CLASSIFICATION ALGORITHMS FOR PREDICTIVE MUTATION TESTING

Akash Nair, Aparna Kishore, Manasi Kancherla, Pravardhan Nagireddy
TeamNo: 6
Department of Computer Science
University of Virginia

1 PROBLEM DEFINITION

Mutation testing is used to test the efficacy of a given test suite by deriving mutants from the source code and running the tests on the mutants. The mutants are generated by using certain mutant generating operators which perform a minute change on a single statement of the original source code. If a mutant is undetected by the test suite, it is said to have survived and if not, it is killed. Usually, the efficacy of the test suite is determined by the mutation score (which is the ratio of killed mutants to the total number of mutants). Mutation testing proves to be incredibly effective in quantifying the efficiency of the test suite because not only does it check for code coverage, but it also verifies if the test suite checks the functionality of the code. However, mutation generation and testing is generally computationally very expensive, which inhibits the use of this technique. So the problems that we try to address are as follows: How do we make mutation testing faster and more effective by reducing computational overhead? In terms of approach, how do we obtain the results of mutation testing without actually executing the test suite on all the mutants?

The number of mutants that can be generated even for a small snippet of code can be very high and running the entire test suite on each of these mutants is computationally very expensive. An approach to solving this problem is to predict the output of the execution without actually executing each mutant against the test suite in order to reduce the computations required. To overcome the limitations of traditional mutation testing and to embrace the idea of a huge input space. We don't need to restrict the number of mutants that are being generated. We define our input space by selecting features of the source code and test suite such as the number of lines executed by the test suite, number of assert statements in the test suite and so on. We will predict if the mutant gets killed or not. In the paper by Zhang et al. (2018), the authors have implemented a Random Forest Classifier to predict the labels as they were dealing with an imbalanced data set.

Goal: To use different ML classification algorithms and evaluate which of them are able to predict the labels (mutation killed/ survived) with a higher accuracy.

2 PROPOSED IDEA

The paper by Zhang et al. (2018) used Random Forest to predict whether the generated mutant is killed or if it survives (without running the mutant against the test suite). The model was correctly able to determine the label of the mutant in over 85% of the cases. Random Forest was used because the data was imbalanced. We propose to run Predictive Mutation Testing with the other classifiers and determine the best algorithm. Some of the classifiers that we plan to test are: (i) SVM, (ii) Decision tree, (iii) KNN, (iv) Neural Networks.

While judging the efficiency of the different machine learning algorithms, we will not only look at the accuracy but also how the algorithm deals with false negatives and false positives. The problem with mutation testing is that the data set being trained upon is usually imbalanced. False positives would be predicting that mutant is killed when it actually survives whereas false negatives would be predicting that the mutant survives when it is actually killed. We need to reduce both false positives and false negatives. However, we need to focus on reducing the false positives more aggressively because it would otherwise undermine the efficacy of the tool since we don't care much for killed mutants.

We shall implement different classifiers and judge the performance based on the accuracy, the training time, and by taking into account the false positives and the false negatives. We expect to identify the best classifier for the PMT problem.

3 Related Work

1. State of Mutation Testing at Google Petrović & Ivanković (2018): It proposes a scalable mutation analysis framework. It serves as an introduction to adapting mutation testing in the industry by proposing an efficient approach which reduces the number of mutants by selecting only the most interesting ones (at most one per line). The drawback of this paper was that it considers only one mutation per line.

2. Predictive Mutation Testing Zhang et al. (2018): The paper lays out the advantages of Mutation Testing and the challenges that come with it. To address these challenges, the authors propose Predictive Mutation Testing as a solution for achieving computational efficiency. It provides us the data set created from 163 real-world projects and their classifier will act as a baseline for our model testing. However, the paper only explores one ML algorithm (Random Forest Classifier)

3. PMT project in Python PMT (b): In this work, the effectiveness of PMT in another programming language (Python) is evaluated. Again, the drawback is that it explores only the Random Forest classifier.

4. Various classifiers Narayanan et al. (2017) A survey of current machine learning classification algorithms explores Decision tree, Support Vector Machine, Naive Bayes, K-Nearest Neighbour, Neural Network. However, the data sets employed in the survey do not deal with our domain.

4 Dataset

The GitHub repository PMT (a) is linked from the PMT for Java research paper and includes all the resources they have used for their project. We aim to use the project_arffs subdirectory which contains the attributes extracted from the source code and test suites in the form of arff files. The projects from which this data is obtained is listed in the project_list subdirectory. The number of training data points are 591,813 and the number of testing data points are 148,492. We could see imbalance in data labels especially in training set as in Figure 1. The number of "No" labels were approximately 4 times to the number of "Yes" labels.

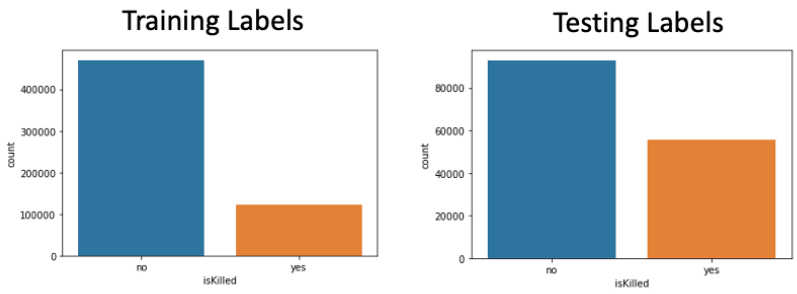

Figure 1: Imbalance in data labels in training and testing set.

In the arff files, the attributes for each possible mutant are listed below:

Inputs

- depthTree: maximum length of a path from the mutated class to a root class in the inheritance structure.

- numSubclass: total number of direct subclasses of the mutated class.
- mcCabe: number of linearly independent paths through the program's source code.
- LOC: number of lines of code in the mutated method.
- depthNested: depth of nested blocks of the mutated method.
- c_a: number of classes outside the mutated package that depend on classes inside the package.
- c_e : number of classes inside the mutated package that depend on classes outside the package.
- instability: computed based on the c_a and c_e value
- numCovered: how many times the mutated statement is being executed
- operator: type of mutation operator
- methodReturn: return type of the mutated method.
- numTestsCover: how many tests are covering the mutated statement
- mutantAssert: total number of assertions in the test methods that cover each mutant.
- classAssert: total number of test assertions inside the mutated class's corresponding test class.

Output

- isKilled: identifies if the mutant is killed or if it survives. Used as data label.

Out of the 14 input features, two features, operator and methodReturn had categorical values. rest of the 12 input features had numerical values. The correlation matrix for the input features is as shown in Figure2. We can see that McCabe and LOC are positive correlated. Similarly, numTestCovered and mutantAssert are also highly positive correlated. Hence, it will be sufficient to use one feature from each of the set. However, for our current analysis we considered all the features to achieve our goal.

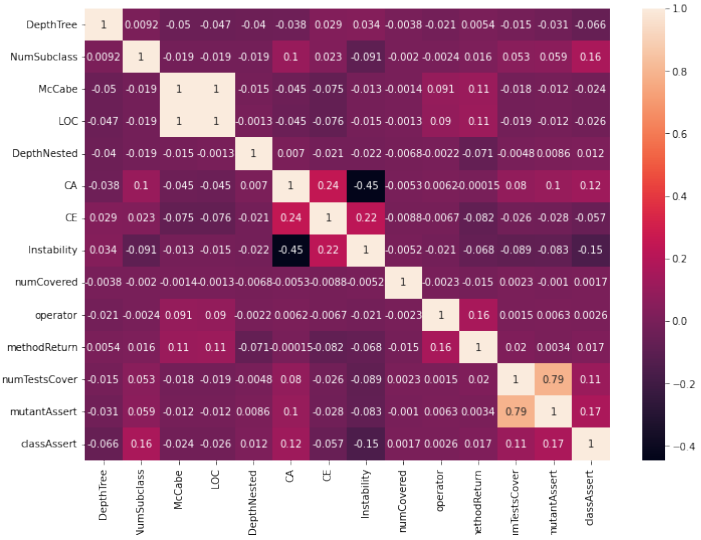

Figure 2: Correlation matrix for the input features

5 EXPERIMENTS

The first step was to replicate the work by Zhang et al Zhang et al. (2018) using the data set provided. This work mainly relied on Random Forest as their classification algorithm and was implemented

using Weka. They could achieve an accuracy score of 85% using this implementation. The feature importance plot for the work by Zhang et al is in Figure 3a. We used the provided data set and implemented them in Python using Scikit sklearn package. We could achieve an accuracy score of 85.93% on the test set. We also plotted the feature importance as in Figure 3b. We observed that some of the features which had lesser significance as per the feature importance plot by Zhang et al had a higher significance in ours. For example, typeOperator had a significance in 4^{th} place when implemented in Python. Whereas it had 12^{th} place when implemented in Weka. The merit for the most important feature "numExecuteCovered" was around 0.65 in the work by Zhang et al. It is around 0.5 in our implementation. We have the explored Machine Learning Algorithms for PMT in Github ML.

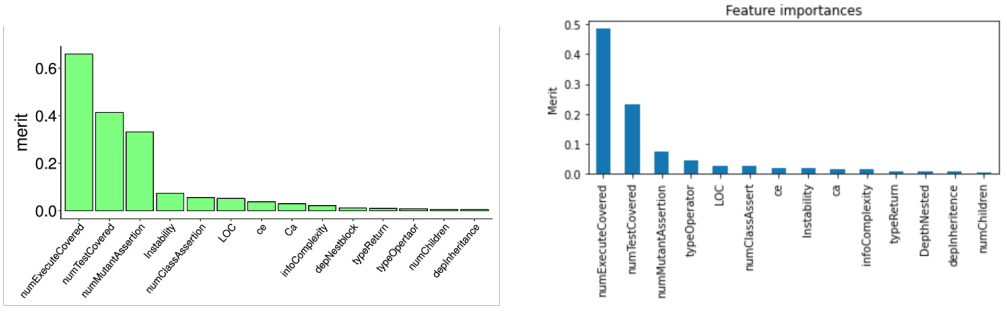

Figure 3: (a) Feature Importance plot implemented in Weka. (b) Feature importance plot implemented in Python.

We implemented different classification algorithms to identify the efficient one to achieve our goal. Percentage of training set and validation set split was not having much impact on the validation or the test accuracy as in Figure 4. Hence we implemented most of the algorithms with different splits.

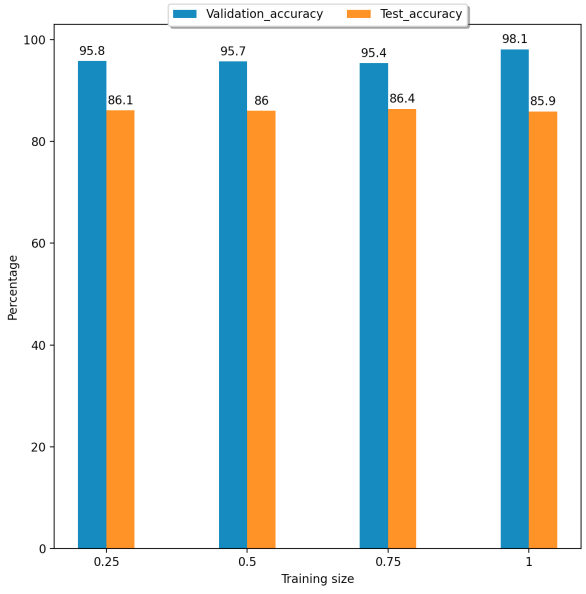

Figure 4: Experimental setup for Random Forest

5.1 EXPERIMENT SETUP

All the ML algorithms that we tried along with the techniques used for implementing them is explained in this section.

5.1.1 RANDOM FOREST

Random forest is a collection of decision trees. Our experimental setup for Random Forest is as in Figure 6. As we had a large sample of data, we had a 50:50 split for training and validation data. Random forest cannot work with categorical data. Hence we converted the two features which had categorical values to numerical values using LabelEncoder, which is a data pre-processing module provided in Scikit-sklearn. We also tried using StandardScaler from Scikit-sklearn to standardize the data values, so that it remains close to the mean value. But this process reduced the accuracy further. The pre-processed data is used for hyper parameter tuning. We tuned 5 hyper parameters namely (i) n_estimators (50,100,200,250,500) (ii) max_depth (10,20, None), (iii) min_samples_split (2,5), (iv) min_samples_leaf (1,2) and (v) criterion (entropy, gini). Dual was set to false during the entire grid search. This was because the number of samples were more than the number of features in the data set. The parameter which gave us the best accuracy in both training and the testing set was considered as the best one. We trained the model using only one sample as the accuracy was not changing with different random samples of train validation split.

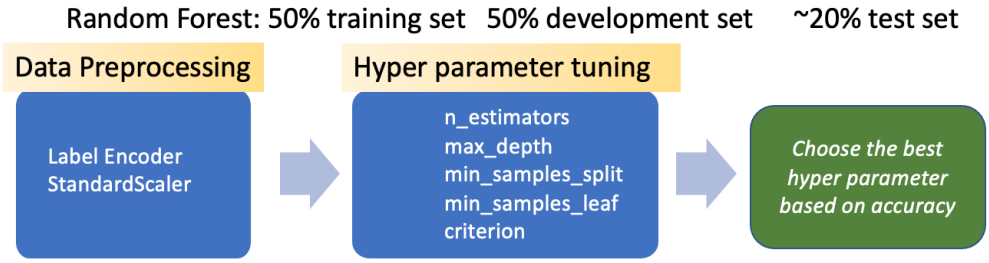

Figure 5: Experimental setup for Random Forest

5.1.2 SUPPORT VECTOR MACHINE (SVM)

For training with SVM, we used three kernels such as (i) Linear, (ii) Gaussian and (iii) Sigmoid. For all the kernels we trained the model for 20 different random samples with each sample having 50:50 training to validation data split. This was done as the sampling yielded different result, especially for Sigmoid kernel. We also used LabelEncoder for pre-processing data. Although we tried using OneHotEncoder for data pre-processing, SVM models were running for a longer duration and was computationally expensive for SVM. Each of the random sample were trained by varying the hyper parameters and computing the best hyperparameter. Thus, we have 20 best hyperparameters. The best hyper parameter was chosen based on majority. For Linear kernel, the cost function, the inverse of the regularization parameter was varied for 5 different values 0.01, 0.1, 1, 10 and 100. For Gaussian kernel, the gamma values and cost function (C) were varied. We tried gamma values of 1e-06, 5e-06, 5e-05, 0.0001 and 0.001 and C values of 0.01, 0.1, 1, 10 and 100. Though the gamma value above 0.001 increased the validation accuracy to 94%, the test accuracy dropped to 47% due to over fitting. For Sigmoid kernel, the independent coefficient (coef0) and cost function (C) were varied. We tried coef0 values of 0.01, 0.1, 0, 1 and 10 and C values of 0.01, 0.1, 1, 10 and 100.

We also tried to normalize the data using Normalizer module from sklearn for data pre-processing. We performed hyper parameter tuning for getting the best hyper parameter for one sample. For Gaussian kernel, we increased the range of gamma values for grid search. It varied between 1e-06 to 10 (15 values). For Sigmoid kernel, we used the same range of values for independent coefficient and cost function as used in the case of training set without normalization. The best hyper parameters and its related accuracy are discussed in Experiment results.

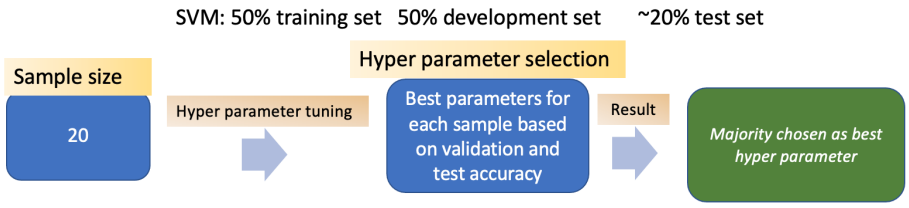

Figure 6: Experimental setup for SVM

5.1.3 NAIVE BAYES

We used Gaussian Naive bayes to check the prediction accuracy. We used Label encoder to clean the data. Initially without parameter tuning we got an accuracy of 79.54% and 63.16% on train and test data respectively. We tried oversampling to see if that improves the prediction accuracy. We saw that oversampling actually made the performance worse. We then tried to do parameter tuning using GridSearchCV. We found the model to work best with a parameter of var_smoothing as 0.0001873. On applying this value and running the model on the data we got a prediction accuracy of 79.53% and 63.16%. Even with parameter tuning we were not able to get much of an improvement in the accuracy. However we got a really low false positive i.e. 42.

5.1.4 DECISION TREE

Our approach for decision tree is similar to random forrest. We use label Encoder to preprocess the data. We spilt the train and validation data 50:50.We noticed that the data was highly unbalanced the yes:no ratio is close to 1:4. To better improve the performance of the algorithm we added duplicate samples of yes to make the ratio closer to 1:1. We oversampled the test data. We got a trainning accuracy of 98.65% along with a test accuracy of 82.63%. The false positives were 7297. We then ran hyperparameter tuning using Random search CV. We ran the tuning on criterion, splitter, max_depth, min_samples_split, min_samples_leaf, max_features. Criterion is the function to measure the quality of a split. Splitter is The strategy used to choose the split at each node. max_depth The maximum depth of the tree. min_samples_split The minimum number of samples required to split an internal node. min_samples_leaf is the minimum number of samples required to be at a leaf node. max_features is the number of features to consider when looking for the best split. The hyper parameter tuning gave us the following values 'splitter': 'best', 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_features': 'sqrt', 'max_depth': 90, 'criterion': 'entropy'. With these values we are able to get 87.97% prediction accuracy on the test data along with 9455 false positives.

5.1.5 KNN

The K nearest neighbor algorithm uses adjacent data points in the training set to compute the label of each data point in the test data set. For this project, we used the scikit library implementation of knn. Label encoder was used to preprocess the data hence enabling the model to compute numerical distances between the data points. Before training the model, a normalizer was used to scale the data. After running a grid search to find the best value of k, the model returned a value of k = 19 as the best number of neighbors for the given dataset. After setting the knn to run on the dataset with weighted distances, a drop in accuracy was recorded. We also tried difference distance metrics: euclidean, manhattan, chebyshev, and minkowski out of which the euclidean distance gave us the best result. Cross validation was performed using a 50:50 data split on the training data.

5.1.6 LOGISTIC REGRESSION

In this classification algorithm, we iteratively try to reduce to the value returned by a predefined cost function (which in turn determines how wrong the model is). For this project, we used the scikit library implementation of logistic regression and stochastic gradient descent to achieve our required results. Before feeding the data to the model, label encoder is used for preprocessing. Then, the first

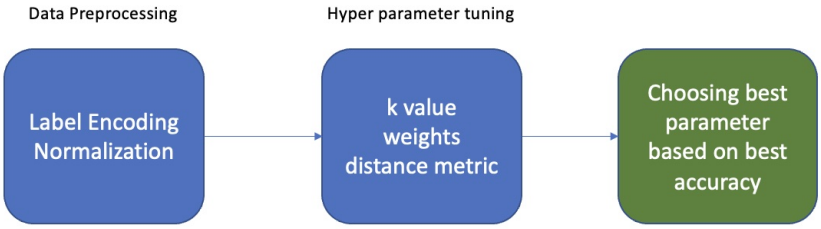

Figure 7: Experimental setup for KNN

parameter that was fine tuned was the solver. We tried the lbfgs, liblinear, and sag solvers out of which lbfgs was able to give us the best result. We tried applying $l1$ and $l2$ regularization to all the solvers but we didn't observe any improvements in accuracy. Final optimization was achieved when we used stochastic gradient descent with logistic regression. The data was split 50:50 for performing cross validation.

5.1.7 NEURAL NETWORKS

For classification using Neural Networks, we used fully connected layers. There was very high imbalance in the dataset. The 'no' class was 4x more common than the 'yes' class. If we do the training normally although we might get high accuracy, we might have a lot of false positives. The model might not be able to learn the 'yes' class particularly well. For handling this issue, we made the loss function weighted. As the 'yes' class is 4x less frequent, the model will be penalized 4x more if it makes a mistake on this class. This helped in getting a much better false positive score without hurting the accuracy too much. Next, the learning rate was made adaptive because with a fixed learning rate the training was not converging. MultiplicativeLR gave the best result. It simply divides the learning rate by half for every epoch. The parameters that had the biggest effect were the activation function and the optimizer. When we tried tanh over relu and adam over sgd, the accuracy increased drastically. Coming to the layers, we tried many combinations, we even tried to overfit the training set by having lots of layers with lots of units but the accuracy didn't increase much compared to this shallow network.

Activation Function – tanh
Optimizer – Adam
Fully Connected Layers – (49, 50, 10, 1)
Learning Rate – 0.01
Loss Function - BCEWithLogitsLoss
MultiplicativeLR - 0.5

5.2 EXPERIMENT RESULTS

The best accuracy scores for each algorithm we used are listed below:

1. **Random Forest:** The best validation and test accuracy were obtained for the following hyper parameter: {n_estimators: 50 max_depth:10 min_samples_split: 5 min_samples_leaf': 2 'max_features': 'auto, criterion: entropy}.
   Best accuracy during cross validation: 94.74%
   Best accuracy during testing: 87.2%
2. **SVM**
   - Linear Kernel

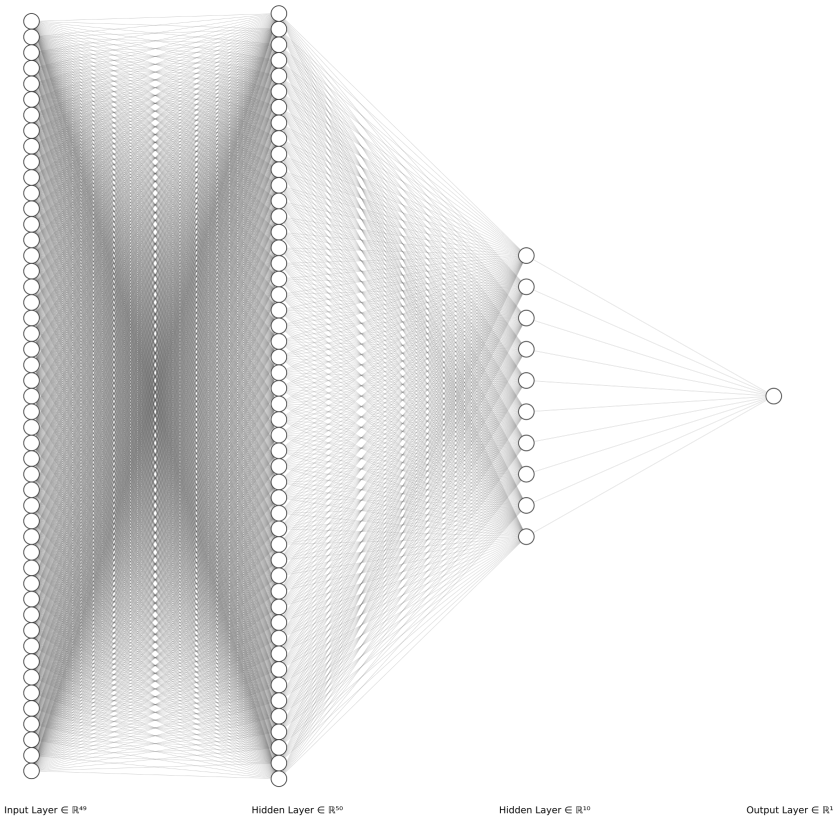

Figure 8: Neural Network Model

All the samples irrespective of their cost functions yielded same accuracy.
Best accuracy during cross validation: 94.21%
Best accuracy during testing: 89.29%

- Gaussian Kernel

  All the samples had the same best hyper parameter for Gaussian kernel. Output from one of the sample is in Figure 9. In the figure, we can see that as the gamma value is increased we can see that the validation accuracy is increasing. However, the test accuracy begins to fall down. This is due to over fitting. Hence the optimum range selected from this figure is when gamma value is 5e-06 and C value is 10/100.

  Best accuracy during cross validation: 91%
  Best accuracy during testing: 83.9%

  For Gaussian kernel, the best hyperparameters were with gamma value of 5 and a cost function of 100. However, the test accuracy remains the same as the one without normalization, i.e. 84%, although the validation accuracy was 95%.

- Sigmoid Kernel

  17 out of 20 samples had a different best hyperparameter of coef0=1 and C=0.01, whereas 3 out of 20 samples had the best hyperparameter as coef0=10 and C=100. One of the sample using Sigmoid kernel is as in Figure 10. Normalization did not yield better result for this kernel.

  Best accuracy during cross validation: 92.5%
  Best accuracy during testing: 84.8%

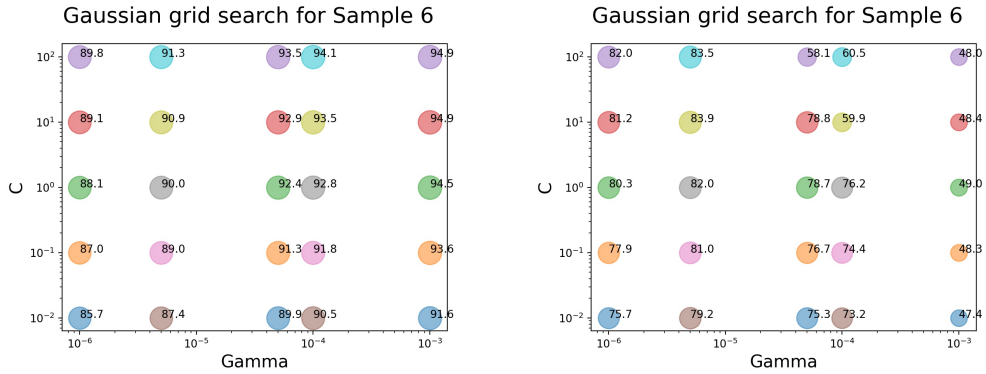

Figure 9: Validation accuracy and test accuracy for Sample 6 using Gaussian Kernel in SVM.

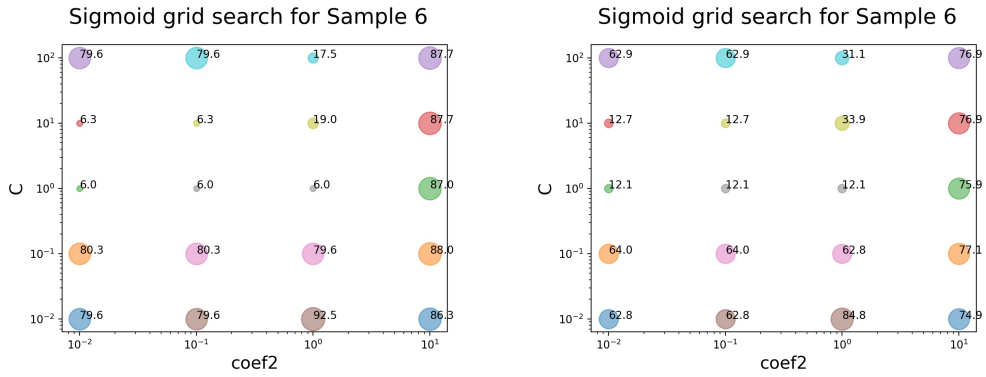

Figure 10: Validation accuracy and test accuracy for Sample 6 using Sigmoid Kernel in SVM.

3. **Naive Bayes:**
   Best accuracy during cross validation: 79.54%
   Best accuracy during testing: 63.16%
4. **Decision tree:**
   Best accuracy during cross validation: 98.65%
   Best accuracy during testing: 87.97%
5. **Logistic Regression:**
   Best accuracy during cross validation: 94.21%
   Best accuracy during testing: 89.29%
6. **K Nearest Neighbors:**
   Best accuracy during cross validation: 94.95%
   Best accuracy during testing: 83.08%
7. **Neural Network:**
   Best accuracy during training: 94.32%
   Best accuracy during testing: 89.01%

5.3 Result Analysis

Comparing all the models together we can see in Figure 11 that logistic regression gives the best overall accuracy while the neural networks, decision tree and random forest come very close. This is expected as the neural network is very shallow and would be close to logistic regression. If we look at the false positives Linear SVM and Naive Bayes achieve very low false positives but they come at the cost of low overall accuracy.

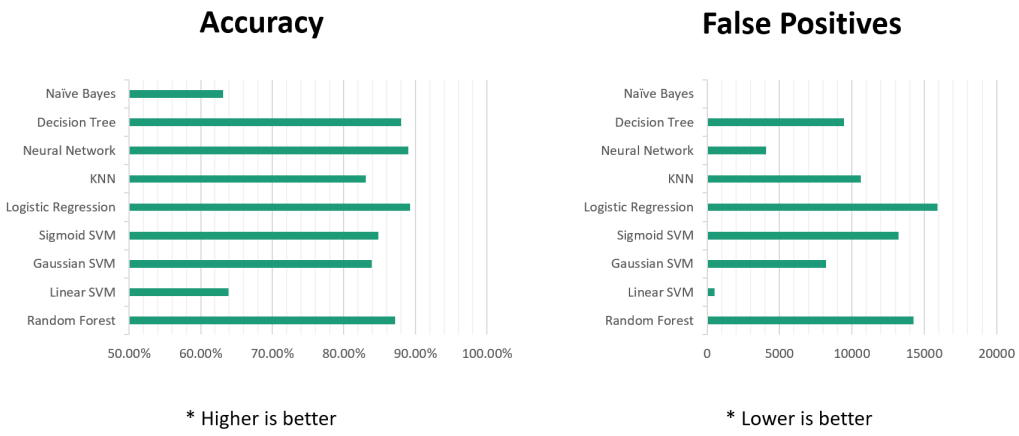

Figure 11: Accuracy and False Positive scores of the models

If we look at the feature importance in Figure 3b, only few features contribute the majority share. Even if we look at the categorical variables, a lot of the categories are very sparse, this could even hurt the models when we use one hot encoding.

6 Conclusion

Over the course of the project we gained really useful insights and experience working with various classification algorithms. We understood the challenges and tried to come up with solutions to tackle those problems. Working with python using the sklearn library was very convenient and it was also more optimized compared to Weka as by using just the default parameters for Random forest, the sklearn implementation achieved higher accuracy compared to the tuned model implemented using Weka in the original paper.

The main challenge we had to face was the imbalance in the data. The ‘no’ class was 4 times that of the ‘yes’ class. The models were achieving good accuracy on the test data but they were getting a lot of false positives. To combat this we explored various strategies like oversampling and weighted loss function and observed their effects.

Hyper-parameter tuning increased the accuracy of the models across the board. It was interesting to see that after tuning, a lot of the classification algorithms performed similarly in terms of accuracy. In the end, by exploring various classification algorithms we were able to find models like Neural Network and Logistic Regression that outperformed Random Forest. In future, we hope to create a better dataset with good quality features that can better characterize the mutations and help the models make better predictions.

Our Project code can be found here

REFERENCES

Explore Machine Learning Algorithms for PMT. https://github.com/pravardhanreddy/MLProject.

PMT. https://github.com/SEITest/PMT, a.

Predictive Mutation Testing For Python. https://github.com/aparnakishore/Predictive-Mutation-Testing-For-Python, b.

Uma Narayanan, Athira Unnikrishnan, Varghese Paul, and Shelbi Joseph. A survey on various supervised classification algorithms. In *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, pp. 2118–2124. IEEE, 2017.

Goran Petrović and Marko Ivanković. State of mutation testing at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, pp. 163–171, 2018.

Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. Predictive mutation testing. *IEEE Transactions on Software Engineering*, 45(9):898–918, 2018.