**CS 6356.001: Software Maintenance, Evolution & Re-Engineering**

**Assignment 6: Defect Prediction**

**Team 17**

**Team Members:**

* Shivani Talatam
* Zamaan Bawa
* Pritul Dave

**Setting Up the Training Data:**

In order to create the training data, we first downloaded the source code for each of the necessary versions. Using the command prompt, we were able to run a command to create a text file with all the java class files. The command was: grep -rl --include="\*.java" "class " . > class\_files.txt where class\_files.txt is the name of an output text file where the data was stored. After this, we needed to add a unique identifier to each of the classes since we would have all the classes for each version in one csv file. This was done using a Python script. The script is available in the file “createUniqueIdentifier.py”. It simply adds the version number to the start of each line.

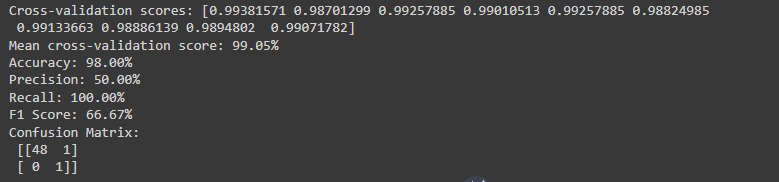
Now that we had got the preliminary files setup, we could calculate the desired metrics. For this project, we looked to calculate the Coupling Between Objects (CBO), Lines of Code (LOC), Average Operation Complexity (OCavg), Response for a Class (RFC), Weighted Metric Complexity (WMC) and whether it is buggy or not. To solve this, we used IntelliJ’s Metrics Reloaded plugin. We created a custom profile to calculate these results. This was done for each version needed for the code. After saving each of them into separate CSV files, we combined them into one CSV file. Next, we needed to get the buggy values. This was done via downloading the Jira issue reports for each version. After this, we wrote a script “link.py” which looped through all the data we had and compared it with the Jira issue reports. If the class name and issue’s class name are the same, then it extracts the issue key. Next, it will check if that issue key is found in the other versions, it sets the buggy value to 1. Lastly, it saves the value to a new column in an output file.

Now that the buggy values are calculated, we need to balance the dataset. This is due to the fact that the values for bugs are pretty low which is causing incorrect results. In order to resolve this, the professor suggested we balance the positive and negative data in the training set. We wrote another script, “balance\_dataset.py” which read the dataset and took a count of the buggy and non-buggy instances. Following the professor’s suggestion, we duplicated the buggy values to balance the ratios for 20%, 30%, 40% and 50%. After this was finished, we were finally ready to train the data.

**Training the Data:**

To train the data, we used two Python files using the scikit-learning library since we were running into issues with Weka. The two files used were “train\_test\_split.py” and “model.py”. We used the machine learning algorithm where the “model.py” code imports “Decision Tree Classifier” and “Grid Search” libraries from Scikit-Learning for data analysis. Initially we had to preprocess the training and test data from CSV files, by removing unnecessary columns and replacing the “n/a” values with 0. The “train\_test\_split.py” splits the data into training and test datasets . After loading the training data and splitting it into input and target variables, the code uses grid search to determine the optimal depth for the decision tree model. The grid search is performed by trying out a range of depth values from 1 to 10 and evaluating the performance of the model for each depth using cross-validation. The depth that gives the best performance is then selected as the optimal hyperparameter for the decision tree model. The code applies the trained decision tree model to the test data to make predictions. Then, it evaluates the performance of the model on the test data using various metrics such as accuracy, precision, recall, F1 score, and confusion matrix. Finally, it calculates the correlation matrix for the training data.

**Results:**



Looking at the results, our scores were decent. The cross-validation scores were high across the board for all ten folds. This means the model was able to learn the patterns found in the training data. We also had a high accuracy, but the precision was not as high as we wanted. This is due to the bug balancing we did earlier. It reached 50% which was something we came to expect given our script to balance the buggy values. Looking at the F1 score, the result was adequate. A score of 66.67% most likely means we are able to have a balance between precision and recall. Lastly, the confusion matrix lets us know we were able to classify 48 negative and 1 positive case correctly, but misclassified 1 negative case as a positive. It can be determined the model is biased towards predicting negative scores which can be seen in the 50% precision score.



Looking at the table with our correlation coefficients, we can see how a majority are positive, but low. There aren’t really any coefficients over 0.40. CBO and RFC, for example, have a decent positive correlation at 0.37, but CBO and OCavg have a weak positive correlation at 0.09. An important correlation is between CBO and bug which is 0.21. This means as the CBO increases, the odds of a bug occurring is also higher.

For OCavg, we can see its correlations. Its correlation with bug is a negative value which means a higher OCavg can lead to a lower number of bugs. RFC has a very strong positive correlation with WMC at 0.94 which means they are highly related. Similar to OCavg, RFC has a negative correlation with bug meaning a higher RFC can lead to a lower number of bugs. Lastly, WMC has a weak negative correlation with bugs which could indicate a lower number of bugs with a higher WMC.

Overall, the results we found were interesting. We would have preferred having a higher precision, but given the predictions, it was solid. We had very accurate results looking at the cross-validation scores.

**Team Contributions:**

* Shivani Talatam
  + Created csv files for training
  + Worked on report
  + Effort: 30%
* Zamaan Bawa
  + Extracted classnames and added unique identifiers
  + Created csv files for training
  + Worked on report
  + Effort: 30%
* Pritul Dave
  + Wrote scripts to train data
  + Wrote scripts to predict buggy values
  + Effort: 40%