Data Mining Assignment 1
Classification Trees, Bagging, and Random Forests

Lizzy Brans 1110209
Willemijn Barens 2145480
Anna de Wolff 1498835

1 Introduction

The goal of this assignment is to analyse the Eclipse bug data set and predict the occurrence of post-release bugs using classification models. Three models were implemented: a single classification tree, a bagging ensemble model, and a random forest classifier. The models are evaluated based on their ability to predict post-release bugs using accuracy, precision, and recall metrics. The analysis focuses on two Eclipse software releases: release 2.0 for training and release 3.0 for testing. In software engineering, predicting software defects allows for proactive bug detection and improved software quality. By using machine learning models to analyse past releases, this assignment aims to uncover patterns in code complexity and pre-release defect counts that can predict the likelihood of defects in future versions. The classification models implemented in this analysis differ in their approach to handling overfitting and variance, and their performance will be examined to determine which is most suitable for predicting defects in this context.

2 Data Description

The dataset used is from the Eclipse bug map, containing complexity metrics and defect information for different software packages across several releases of the Eclipse software, including versions 2.0, 2.1, and 3.0 (Zimmermann et al., 2007). The dataset is used to understand the relationship between code complexity and software bugs, helping us predict potential defects in new software releases. This makes the dataset helpful in developing models that can predict where bugs are likely to occur based on past data. Each entry in the dataset corresponds to a software package and includes attributes that provide insights into both the package's complexity and its defect history:
Package name: The name of the software package or file being analysed.
Pre-release defects: The number of non-trivial defects (bugs) reported before the software was officially released.
Post-release defects: The number of non-trivial defects reported in the six months after the software's release.
Complexity metrics: These metrics capture various aspects of the software's structure, including Method Lines of Code (MLOC), Cyclic Complexity (VG), and the Number of Method Calls (FOUT). They provide a measure of the size, complexity, and interaction levels within the code.

The complexity metrics are provided in multiple forms: averages, maximums, and totals across different packages. These values offer an understanding of how complex the code might be to maintain or debug and serve as predictors in identifying potential post-release bugs. For this assignment, we focused on data from the Eclipse 2.0 release as our training set and used data from the 3.0 release as our test set. In total, we got 41 predictor variables, including the complexity metrics and pre-release defect counts, to predict the occurrence of post-release defects. By analysing historical patterns in this data, we aim to build machine learning models to effectively predict software defects in future releases.

3 Model Development

This section details the development of three machine-learning models for predicting post-release defects: a single classification tree, a bagging ensemble model, and a random forest classifier. The Eclipse 2.0 release data is used for training, while the 3.0 release data serves as the test set. The models are built to predict post-release defects based on the number of pre-release defects and software complexity metrics.

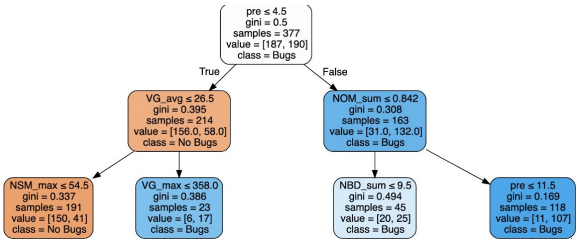

Figure 1: First three splits of a single tree

3.1 Single Classification Tree

The single classification tree is designed to classify whether a software package is likely to have post-release defects. The parameters given in the assignment used to train this model are shown in Table 1. The first split of the classification tree is

Parameter	Value
nmin	15
minleaf	5
nfeat	41

Table 1: Parameters for the single classification tree model.

based on the number of pre-release defects, which captures how many bugs were identified before the software release. The subsequent split used the maximum cyclomatic complexity (**VG max**) to measure code complexity. Software packages with more pre-release defects or high structural complexity will likely experience post-release bugs.

Figure 1, the visualisation of the single tree can be found. In the first leaf node (**NSM max**), we see that software that has a low number of submodules, indicating the code is less complex, is reasonably classified as 'no bugs'. In the second leaf (**VG max**), the majority of cases are classified as 'bugs'. This classification is also reasonable in line with the notion that complex software is more prone to error. The third leaf (**NBD sum**) classifies software with a moderate sum of block depth as containing bugs. This seems intuitive as well, considering that nesting code brings higher complexity. Lastly, the fourth leaf (**pre**) classifies the software as containing bugs when the number of pre-release bugs is relatively high (but 11.5). Overall, given the attributes involved, the classification rules seem reasonable and align with general expectations of software complexity.

The confusion matrix for the single classification tree is shown in Table 2. It shows the model's ability to correctly classify packages with and without post-release defects.

Actual/Predicted	Predicted 0	Predicted 1
Actual 0	266	82
Actual 1	128	185

Table 2: Confusion Matrix for Single Classification Tree.

3.2 Bagging

Bagging, or Bootstrap Aggregating, reduces variance and improves prediction accuracy by averaging the predictions from multiple trees. In this method, 100 bootstrap samples were generated from the training data, and a classification tree was trained on each sample. The parameters used in the bagging model are provided in Table 3. By

Parameter	Value
nmin	15
minleaf	5
m (number of trees)	100

Table 3: Parameters for the bagging model.

aggregating the predictions from 100 trees using majority voting, bagging reduces overfitting and increases the generalisation of the model. The confusion matrix for the bagging model, showing the model's performance on the test set, is shown in Table 4.

Actual/Predicted	Predicted 0	Predicted 1
Actual 0	306	42
Actual 1	105	208

Table 4: Confusion Matrix for Single Classification Tree.

3.3 Random Forest

Random Forest builds upon the principles of bagging but adds extra randomness by selecting a random subset of features for each tree split. Instead of considering all 41 predictors at each split, only six (calculated as the square root of 41) are selected randomly. The parameters for the random forest model are shown in Table 5. The added randomness in feature selection helps decorate individual trees, reducing overfitting and improving accuracy. The table 6 shows the confusion matrix for the random forest model, showing its effectiveness in predicting post-release defects.

Parameter	Value
nmin	15
minleaf	5
nfeat (random subset)	6
m (number of trees)	100

Table 5: Parameters for the random forest model.

Actual/Predicted	Predicted 0	Predicted 1
Actual 0	282	66
Actual 1	92	221

Table 6: Confusion Matrix for Random Forest.

4 Results

4.1 Confusion Matrices and Performance Metrics

The performance of each model is summarised in Table 7. The metrics evaluated are accuracy, precision, recall, and the F1 score, using the test set (release 3.0) to assess the model's prediction abilities.

4.2 Statistical Significance of Accuracy Differences

In order to verify the statistical significance of our resulting accuracy scores, we conducted an ANOVA test using a k-fold test evaluation approach. The models were first trained on a separate training dataset, and we applied a k-fold test evaluation on the test data, dividing it into five folds. Before splitting the test data into folds, the rows of data were shuffled to ensure that the folds contain a random subset of the data. Without shuffling, possible dependencies in subsequent rows could lead to biased data folds and results. This resulted in multiple accuracy scores per model, which we compared using an ANOVA test. The ANOVA results showed a significant difference in performance between the models, with an F-statistic of 6.06 and a p-value of 0.015, indicating that the differences in accuracy between at least some of the models are statistically significant ($p < 0.05$).

To understand which models result in significant differences a Tukey post-hoc test was performed (Figure 2). The test revealed that the differences between the single classification tree and both the bagging model and the random forest model were significant at $p = 0.0175$ and $p = 0.0479$ respectively. However, no significant difference was found between the bagging model and the random forest model. Altogether, these results show that methods like bagging and random forests significantly improve accuracy in comparison to a single classification tree. However, performance between bagging and random forests does not differ significantly.

```
      Multiple Comparison of Means - Tukey HSD, FWER=0.05
=================================================================
    group1        group2    meandiff p-adj   lower   upper  reject
-----------------------------------------------------------------
      Bagging Random Forest  -0.0165 0.8409 -0.0946  0.0616  False
      Bagging   Single Tree  -0.0954 0.0175 -0.1735 -0.0173   True
Random Forest   Single Tree  -0.0788 0.0479 -0.1569 -0.0007   True
-----------------------------------------------------------------
```

Figure 2: Tukey HSD

Based on the aforementioned metrics, both Bagging and Random Forest outperform the Single Classification Tree in accuracy, precision, recall, and F1-score. The Random Forest model has the highest overall performance, especially regarding recall and F1-score, while the Bagging model demonstrates high precision. The Single Classification Tree model achieved an accuracy of 69%, with a precision of 70%, recall of 59%, and an F1 score of 64%. This model demonstrates reasonable precision in predicting post-release defects, but the lower recall suggests that it tends to miss many defect cases, limiting its usefulness when identifying as many bugs as possible is crucial. In contrast, the Bagging model showed improvements in all metrics, with an accuracy of 77%, precision of 81%, recall of 67%, and an F1 score of 73%. This indicates that Bagging effectively reduces false positives, making it a good option when avoiding incorrect defect predictions is important. However, recall remains lower than the Random Forest model, indicating some missed defect cases. The Random Forest model achieved the best balance across all metrics, with an accuracy of 77%, precision of 78%, recall of 71%, and an F1 score of 74%. This model is the most effective at detecting post-release defects, because of high recall and precision. Therefore, Random Forest can be considered the best model overall, especially when both defect detection (recall) and minimising false positives (precision) are equally important.

5 Discussion

The results showed that the Random Forest model outperforms the Single Classification Tree and Bagging models in terms of overall performance, with a better balance between precision and recall. While Bagging shows higher precision, its lower recall than Random Forest suggests it may not capture

Model	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
Single Tree	69	70	59	64
Bagging	77	81	67	73
Random Forest	77	78	71	74

Table 7: Performance Metrics for Single Classification Tree, Bagging, and Random Forest Models

as many defect cases. To properly test if these differences in precision and recall are significant, an additional ANOVA and Tukey test would need to be done for these metrics.

In contrast, the Single Classification Tree model struggled to maintain a balance between precision and recall, missing a significant number of defect cases.

5.1 Limitations

One limitation of this study is that the models were evaluated using a specific dataset from Eclipse software releases. This limits the generalisability of the results to other software systems or defect datasets.

Additionally, the models were tested on post-release defect data only, which means they may not perform as well when predicting defects in real-time or with other release timelines. The models also relied heavily on code complexity and defect counts, potentially missing other significant factors like developer behaviour, which refers to the decision-making processes that specific developers exhibit, or the specifics of the software architecture that could influence defect occurrence.

Another limitation is the use of k-fold test evaluation on the test data only instead of performing a traditional cross-validation approach. Normally, cross-validation splits both the training and test set multiple times and trains the model on each fold. However, in our case the models are already trained on a separate dataset, while only the test data is divided into folds. The choice of validation was due to the assignment restrictions and the pre-selected datasets. This could limit our insights into the model's stability across different training sets.

5.2 Future studies

Future studies could incorporate additional features like developer activity to improve prediction performance. By including developer activity, we could more accurately identify which sections of the code are frequently reviewed or altered, which could highlight high-risk areas sensitive to bugs. Moreover, validating these models on diverse software systems would provide stronger insights into their applicability across different contexts.

Additionally, in this study we only focused on the use of decision trees in order to analyse and predict the occurrence of post-release bugs. Future studies could focus on employing different types of machine learning methods instead. The original study used a logistic regression model, which did have a lot of potential, but was far from being perfect (Zimmermann et al., 2007). A Naive Bayes classifier may be able to estimate the probability of a bug occurring, instead of solely making a binary decision on its presence or absence. This would provide a more detailed insight into the risks of each software component, allowing developers to prioritize debugging of riskier components. Nevertheless, in order for this to work more research would need to be done as it may not perform well if features turn out to be highly interdependent.

References

Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pages 9–9.