INFOMDM Assignment 1

Pascal Verkade
6045057

Idan, Grady
7304447

Marc de Fluiter
5928087

October 15, 2021

1 Data description

For the data analyses using a Classification tree, Bagging and Random forest, we used the dataset that was used in 'Predicting Defects for Eclipse', by Thomas Zimmermann, Rahul Premraj and Andreas Zeller. This dataset contains the pre- and post-release defects for every package in Eclipse and the goal is to predict whether or not any post-release bugs have been reported. The data was collected and created by Zimmerman et al., by analysing and collecting data from archives and bug tracking systems. In order to make a proper prediction and evaluation, we will train our classifiers using the package level data of release 2.0 (consisting of 377 items/packages), while testing on the 3.0 release (consisting of 661 items). Instead of using all the derived and anotated data, we will only use a subset consisting only of only the 13 metrics with their corresponding avg, max and total levels that Zimmerman et al. marked in their article. Three examples of the 13 metrics are *PAR_total*, which is a metric for the total number of parameters in a method, *NOM_avg*, which indicates the average number of methods in a class, and *TLOC_max*, which stands for the maximum total lines of code for a file. The idea is that these 39 metrics together with the single value metric for the number of files and the number of pre-release bugs, are important predictor variables for the prediction of post-release presence of bugs with our three classifiers. Prior to the model implementation and testing, we performed some data prepossessing, so that it is tailored to our needs:

- Extracted both the 41 predictor variables (hereafter noted as X), which serves as our features, as well as the number of post release bugs (hereafter noted as y), which will serve as our label to predict.
- Since the values of the post release bugs are numeric, we transformed it back to a $0, 1$ representation, in order to use it for classification predictions. Here, 0 means that not a single bug was found and 1 means that at least 1 bug was found..

For a more in depth view of the data, how it was collected and how it can be used, please refer to the original article of Zimmerman et al.

2 Data analysis

As explained above, all three algorithms will be using the Eclipse 2.0 dataset for training and the Eclipse 3.0 dataset for testing. The algorithms have similar parameters and we will shortly describe them here. The first parameter is *nmin*, which indicates the number of observations that a node must contain at least, for it to be allowed to split. The second parameter, *minleaf*, also stops growing the tree, since it is the minimum number of observations required for a leaf node. These first two parameters will be equal across all algorithms (15 and 5 respectively), to better compare them. The third parameter is *nfeat* and this is one is rather important for the random forest classifier since it denotes the number of features that should be considered for each split. For the other algorithms it's the maximum amount of features (41), but for random forest it is set to the conventional square root of the maximum amount of features, resulting in 6. The last parameter m is for bagging and random forests and indicates the number of bootstrap samples to be drawn. For all three algorithms the confusion matrix and the accuracy, recall and precision were calculated to get an idea of the performance of the individual algorithms on this data. Before going over these numbers however, a more in depth description of the first two splits in a single tree is given, in which we go over and say something about the default majority classification rule.

2.1 First two splits on a single tree

The single classification tree is contains a lot of splits. In this section we will look at the first two splits, the split in the root node and the split in its left child. These splits are visualized in Figure 1. The split in the root node of the classification tree is on the $'pre'$ column. This column contains the number of non-trivial defects reported in the last six months before release. The second split is on the complexity metric $'VG\_max'$. This is the maximum of the McCabe cyclomatic complexity over all the methods in the considered package. It measures the complexity of the code structure of the method by looking at the control-flow. If the control-flow can only follow one execution path then this complexity is at the lowest. If the control-flow can follow many different execution paths, for example when there are a lot of nested loops and conditions, a method can quickly get a very complex structure.

Looking at the classification tree we see that packages with 5 or more non-trivial defects in the last six months before release are classified as defect-prone. For packages with 4 or less non-trivial defects in this period we see that there is a second split on the maximum McCabe cyclomatic complexity of the package. If the maximum complexity is above the threshold value of 26.5, then the package is classified as defect-prone again. If this threshold-value is not reached the package is classified as defect-free. We would expect that packages which show a considerate amount (in this specific case 5) of non-trivial defects before release to still show defects after its release. If such defects are encountered in testing, there could still be similar defects in the code which have not been encountered

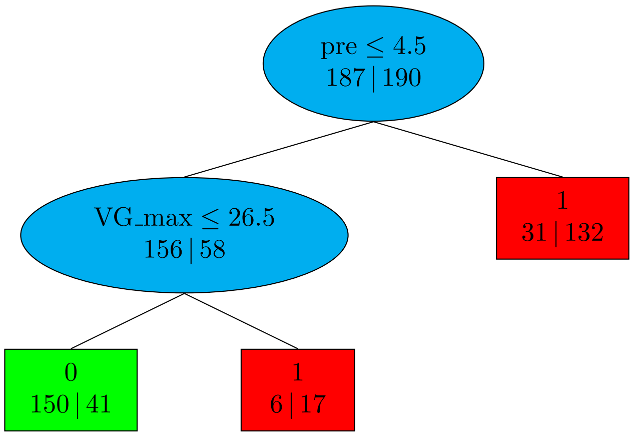

Figure 1: The first two splits of the single classification tree. Nodes which split the data show the condition on which the data is split, providing the name of the column and the threshold value. Data entries that satisfy this condition are sent to the left and the other data entries are sent to the right. Below that are respectively the distribution of data entries with class label 0 and data entries with class label 1 separated by a |-sign. Leaf nodes show their majority class label 0 or 1, which is also visualized by their green or red color. Below this condition the distribution of data entries with class label 0 and 1 are again given as is shown in the other nodes.

yet and show up after the release. So it makes sense to split on the amount of non-trivial pre-release defects and the first split seems to be a very good split. When there is a method in the package with a very complex structure, we expect it to be very prone to defects, because there are many possible execution paths which could all hide a defect. For a complex control-flow structure it is indeed very hard to validate that no single execution will result in a defect. Despite the fact that not too many non-trivial defects are encountered in the period leading up to the release, there could likely still be many possible execution paths that have not been tested yet and which contain a defect. Therefore the second split seems to be a good split as well, signifying that packages with a very complex method could hide many defects in the complex structure of this method, despite not having encountered many defects yet in the period prior to the release.

2.2 Confusion Matrix and Quality Measures

The table below (table 1) consists of the confusion matrices for the three models, with the parameters set as described above. The scores for this table describe the individual facets of the predictive power of the algorithm. Below table 1, is table 2, which plots the quality measures Accuracy, Precision and Recall. By comparing these quality measures, we can more easily relate and differentiate our models in the evaluation below.

Model	Observed Negative	Observed Positive
Single Tree (15, 5, 41, -)		
Predicted Negative	266	128
Predicted Positive	82	185
Bagging (15, 5, 41, 100)		
Predicted Negative	308	104
Predicted Positive	40	209
Random Forest (15, 5, 6, 100)		
Predicted Negative	287	92
Predicted Positive	61	221

Table 1: Table Matrix

Model	Accuracy	Recall	Precision
Single Tree	0.682	0.591	0.692
Bagging Tree	0.782	0.667	0.839
Random Forest	0.7685	0.706	0.783

Table 2: Accuracy, Recall. Precision

3 Evaluation of the three algorithms

When we look at the confusion matrices for the three algorithms, we immediately see that the bagging and random forest model outperform the single classifier tree, which is no surprise of course. The difference between the Bagging and Random forest is more interesting as we can see the bagging classifier performing better on the observed negatives, while the random forest classifier performs better on the observed positives. Although a specific reason can't be pointed out, it might be the case that this is due to the *nfeat* parameter that limits the features to be considered for each split of the random forest classifier. It could be the case that the limited amount of features has skewed the model towards a more negative model. This could have happened if the majority of features tend to have a more negative but smaller bias. In this case, the bagging algorithm will focus on the best splits (being skewed more positive) and the strongest features, while the random forest model has to work with six random features, and thus having a higher chance of being skewed more to the negative.

If we now look at the quality measures of the three algorithms, we do find outcomes that match the findings on the confusion matrices. We see that the performance of the single classification tree is the lowest on all three measures, with an accuracy of 68%, a recall of 59% and a precision of 69%. Both the bagging model and the random forest model have higher values (A: 78%, R: 67%, P: 84% for bagging and A: 77%, R: 71%, P: 78% for random forests).

Here we can see the biggest difference in recall and precision, where bagging has a higher precision and a lower recall than random forest (and vice versa). This is in line with the findings that bagging performs better on observed negatives (precision is based on false positives) and random forest performing better on observed positive (recall is based on false negatives).

Based on these findings we do not believe that with the current parameters one of the three models performs best in general, but it's clear that thee single tree model is the weakest. To test for the best model, we use a statistical test called McNemar's[1] test. The test is based on and makes use of a contingency table, which is depicted in table 3. This table marks how many items were correctly classified by each algorithm. The more commonly used and by Edwards corrected (continuity correction) formula to calculate the statistical result is as follows:

$$x^2 = \frac{(|B - C| - 1)^2}{(B + C)}$$

As you can see the test calculates a significance over the differences of the models, as it is only interested in items that the models performed different on. The statistic is reporting on the different correct or incorrect predictions between the two models, not the accuracy or error rates. If the B and C cells have counts that are similar, it shows us that both models make errors in much the same proportion, just on different instances. This means the test would not be significant and the null hypothesis that the models perform similarly would not be rejected. We will be looking at a significant level of 0.05, since it is standard for statistical analysis. If we fill in the values, we have a resulting p-value of 3.44, which is not significant and therefore, the null hypothesis can not be rejected, meaning that we can't differentiate between the performance of the bagging and the random forest model.

Model	**Model 2 Correct**	**Model 2 Wrong**
Model 1 Correct	A=483	B=35
Model 1 Wrong	C=22	D=121

Table 3: McNemar Visual Explanation with model 1 = Bagging and model 2 = Random Forest

[1]https://wikistatistiek.amc.nl/index.php/$McNemar_toets$/