Report

Assignment 1: Classification Trees, Bagging and Random Forests

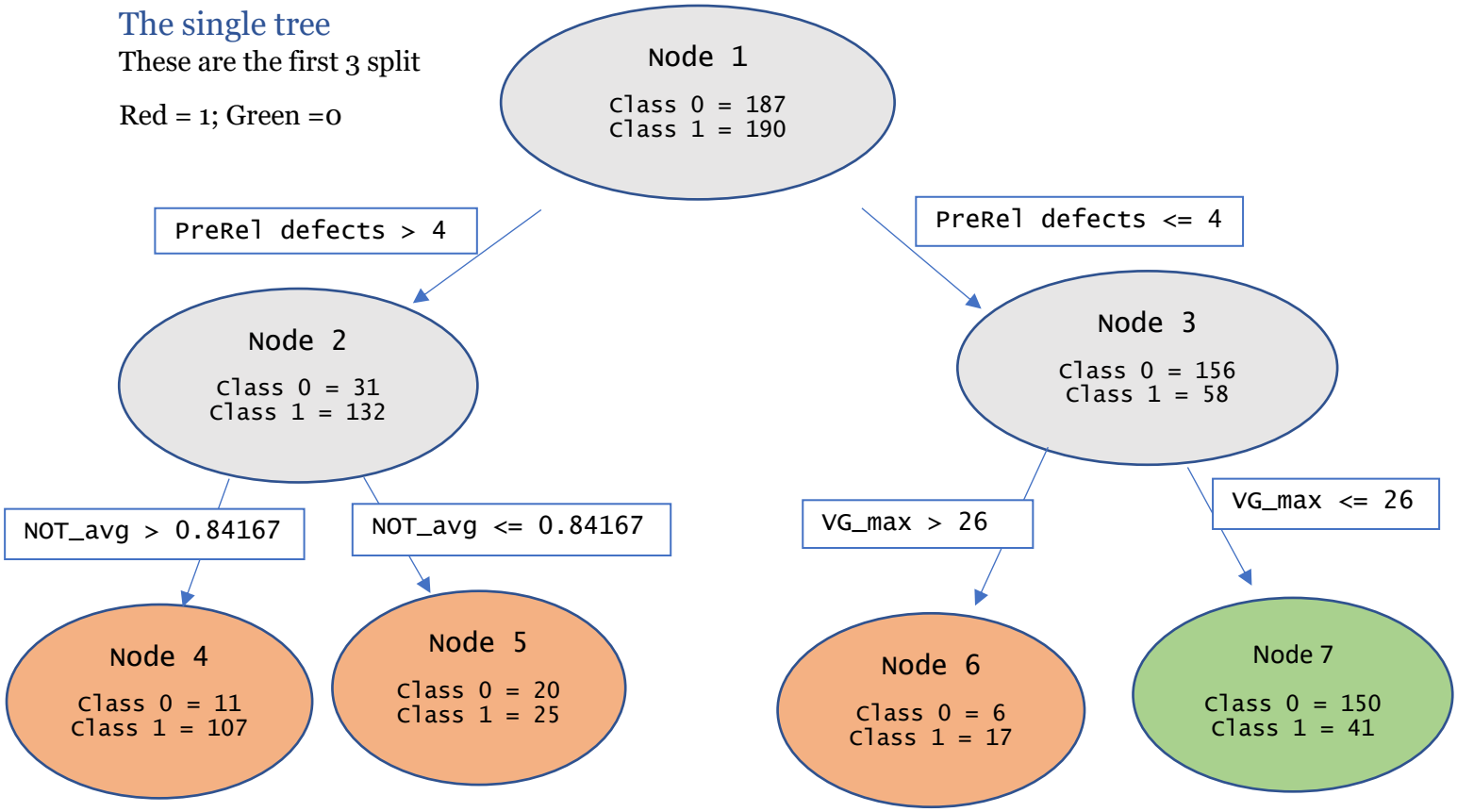
Description of the data

The training set is composed of 377 observations regarding packages in Eclipse release 2.0. The test set is composed of 661 observations regarding packages in Eclipse release 3.0. The used attributes are in total 41 and consist of pre-release defects, the number of files (compilation units) (NOCU) and 13 other complexity metrics for which average, maximum and total values have been computed. The computation of complexity metrics has been carried out by the Java Parser of Eclipse. Moreover, the number of post release defects, i.e. defects observed after the program has been deployed to its users, has been collected for all the observations, so that each package can be classified as *Has_defects = 1* if the number of post release defects is greater than 0 or *Has_defects = 0* if it's equal to 0. These data are used to train, and subsequently test, a binary classification tree to predict whether a package will be reported to have post release defects (*Has_defects =?)*.

The single tree

These are the first 3 split

Red = 1; Green =0



- PreRel defects = number of non-trivial defects that were reported in the first six months after release
- NOT_avg = average number of classes per package
- VG_max = McCabe cyclomatic complexity

This classification rule, even if heavily simplified, makes sense. In details, the first split of the classification tree is the most reasonable (and obvious) one: the number of defects reported before the release of the package is strongly correlated to the number of defects reported after the release. Some common causes could be the (low) quality of the work of the programmer of that package, or

the complexity of the tasks that package is supposed to do. Since all the two leaf nodes on the left have more Class 0 than Class 1 observations, for this classification rule the attribute *Not_avg* is useless and the two nodes can be pruned. It follows that if a package has had more than 4 defects reported before releasing, it will be predicted to have other defects reported after releasing. Still splitting on *Not_avg* makes sense, since a high number of classes is an index of a complex code which is more exposed to defects. If the pre release defects were less or equal to 4, the rule takes into account the McCabe cyclomatic complexity then. It's a measure of the number of linearly independent paths within a source code (caused by control flow statements). A high value is an index of an intricated code, which again is more exposed to defects. Consequently, if this value is above 26, a post release defect in the code is predicted.

Values

In this part there are the confusion matrices and the quality measures of the analysis.

Analysis 1 (single tree)

		Defects are observed	
		True	False
Model predicts defect.	Positive	229	93
	Negative	84	255

Accuracy	Precision	Recall
0.7322	0.7111	0,7316

Analysis 2 (bagging)

		Defects are observed	
		True	False
Model predicts defect.	Positive	221	47
	Negative	92	301

Accuracy	Precision	Recall
0.7897	0.8246	0,7060

Analysis 3 (random forests)

		Defects are observed	
		True	False
Model predicts defect.	Positive	221	59
	Negative	92	289

Accuracy	Precision	Recall
0.7715	0.7893	0,7060

Discussion of the differences

As expected, bagging improves the performances of the classification algorithm. Instead, unlike the expectances, random forests gain a little worsening of the performances. To find out in a rigorous way whether the differences between the three models are statistically significant a McNemar test has been performed between the three models, splitting the predictions in correct ones and incorrect ones. The results are as following.

Single Tree vs Bagging

		Bagging	
		Correct	Incorrect
Single tree	Correct	456	28
	Incorrect	66	111

```
McNemar's chi-squared = 14.564, df = 1, p-value = 0.0001355
```

We refuse the null ipothesis (the differences are not significant) with an error probability of 0,01%: there is strong evidence that the differences are statistically different, i.e. there is a substantial improvement moving from the single tree model to random forests.

Single Tree vs Random Forests

		Random Forests	
		Correct	Incorrect
Single tree	Correct	450	34
	Incorrect	60	117

```
McNemar's chi-squared = 6.6489, df = 1, p-value = 0.009922
```

We refuse the null ipothesis (the differences are not significant) with an error probability of 1%: the re is still some evidence that the differences are statistically significant, i.e. there is a little improve ment moving from the single tree model to random forests.

Bagging vs Random Forests

		Random Forests	
		Correct	Incorrect
Bagging	Correct	488	34
	Incorrect	22	117

```
McNemar's chi-squared = 2.1607, df = 1, p-value = 0.1416
```

We refuse the null ipothesis (the differences are not significant) with an error probability of 14%: as expected, there is no evidence that random forests improve the performances with respect to bagging.

Note

According to the implementation of the algorithm, every time a leaf node contains exactly the same number of observations of class 0 and class 1, the node is considered to be of class 1 and so are classified all the observations of the test set which fall under that node (see row 125 in the code). If instead we consider the node of class 0 (which anyway appears to be a less conservative approach), the quality measures of *Analysis 1* decrease significantly. This could be caused by the presence in the tree of a leaf node (or even more than one) with this kind of tie which has a strong influence on the test set (i.e., a lot of observations go there). After having inspected the tree, there have been found tree nodes (all leaf nodes) with a tie:

- Node 13 (5 observations per each class)
- Node 21 (7)
- Node 35 (4)

For comparison, here there are the results with the less conservative approach.

		Defects are observed	
		True	False
Model predicts defect.	Positive	185	82
	Negative	128	266

Accuracy = 0.6823
Precision = 0.6228
Recall = 0,5910

The same phenomenon does not occur in *Analysis 2* and *Analysis 3*.

Appendix 1: Metrics in the Eclipse Data Set

- FOUT Number of method calls (fan out)
- MLOC Method lines of code
- NBD Nested block depth
- PAR Number of parameters
- VG McCabe cyclomatic complexity NOF Number of fields
- NOM Number of methods
- NSF Number of static fields NSM Number of static methods
- ACD Number of anonymous type declarations NOI Number of interfaces
- NOT Number of classes TLOC Total lines of code
- NOCU Number of files (compilation units)