Data Mining 2021
Assignment 1
Classification Trees, Bagging and Random Forests

Instructions

This assignment must be completed by teams of 3 students, and handed in by e-mail to `a.j.feelders@uu.nl`.

Part 1: Programming

The code for this part should be written in R or Python.

Write a function to grow a classification tree. Also write a function that uses this tree to predict the class label for given attribute values. More specifically you should write two main functions, with the names `tree_grow` and `tree_pred`. The function `tree_grow` has input arguments `x, y, nmin, minleaf`, and `nfeat`, in that order. Here `x` is a data matrix (2-dimensional array) containing the attribute values. Each row of `x` contains the attribute values of one training example. You may assume that all attributes are numeric. `y` is the vector (1-dimensional array) of class labels. The class label is binary, with values coded as `0` and `1`. Furthermore, you may assume there are no missing values (either in training or prediction).

The parameters `nmin` and `minleaf` (both integers) are used to stop growing the tree early, to prevent overfitting and/or to save computation. `nmin` is the number of observations that a node must contain at least, for it to be allowed to be split. In other words: if a node contains fewer cases than `nmin`, it becomes a leaf node. `minleaf` is the minimum number of observations required for a leaf node; hence a split that creates a node with fewer than `minleaf` observations is not acceptable. If the algorithm performs a split, it should be the best split that meets the `minleaf` constraint. If there is no split that meets the minleaf constraint, the node becomes a leaf node. Use the gini-index for determining the quality of a split. The parameter `nfeat` denotes the number of features that should be considered for each split. Every time we compute the best split in a particular node, we first draw at random `nfeat` features from which the best split is to be selected. For "normal" tree growing, `nfeat` is equal to the total

number of predictors (the number of columns of `x`). For random forests, `nfeat` is smaller than the total number of predictors.

The function `tree_grow` should return a "tree object" that can be used for predicting new cases. You are free to choose the data structure for the tree object, as long as it can be used for predicting new cases in the following way. A new case is dropped down the tree, and assigned to the majority class of the leaf node it ends up in. More precisely, the function `tree_pred` has input arguments `x` and `tr`, in that order. Here `x` is a data matrix (2-dimensional array) containing the attribute values of the cases for which predictions are required, and `tr` is a tree object created with the function `tree_grow`. The function `tree_pred` has a single output argument `y`, which is the vector (1-dimensional array) of predicted class labels for the cases in `x`.

For bagging (and random forests), you have to write two auxiliary functions called `tree_grow_b` and `tree_pred_b`. They are not much more than repeated applications of `tree_grow` and `tree_pred` respectively. The function `tree_grow_b` has all the arguments of `tree_grow` and in addition an argument `m` which denotes the number of bootstrap samples to be drawn. On each bootstrap sample a tree is grown. The function returns a list containing these `m` trees. Finally, the function `tree_pred_b` takes as input a list of trees and a data matrix `x` for which predictions are required. The function applies `tree_pred` to `x` using each tree in the list in turn. For each row of `x` the final prediction is obtained by taking the majority vote of the `m` predictions. The function returns a vector `y`, where `y[i]` contains the predicted class label for row `i` of `x`.

Part 2: Data Analysis

Use the functions you have created in part 1 to analyse the Eclipse bug data set. See the course web page for links to the data, and an accompanying article.

We will analyse the package level data, using release 2.0 as the training set, and release 3.0 as the test set. We will try to predict whether or not any post-release bugs have been reported. To predict whether or not bugs have been reported we will use the metrics listed in Table 1 of the accompanying article, and the number of pre-release bugs. We will not use the features derived from the abstract syntax tree. You should end up with 41 predictor variables in total. The results obtained with logistic regression by the authors (with the same set of predictors) can be found in Table 5 of the article.
Perform the following analyses with the code you have written:

1. Train a single classification tree on the training set with `nmin` = 15, `minleaf` = 5 (we have pre-selected reasonable values for you), and `nfeat` = 41. Compute the accuracy, precision and recall on the test set.

2. Use bagging with the same parameter settings as under (1), and `m` = 100. Compute the accuracy, precision and recall on the test set.

3. Use random forests with the same parameter settings as under (2), except

that `nfeat = 6`, that is $\sqrt{41}$ rounded to the nearest integer. Compute the accuracy, precision and recall of the random forest on the test set.

Describe your analysis in a report of about 3 or 4 pages.
The report should contain:

1. A short description of the data.
2. A picture of the first two splits of the single tree (the split in the root node, and the split in its left or right child). Consider the classification rule that you get by assigning to the majority class in the three leaf nodes of this heavily simplified tree. Discuss whether this classification rule makes sense, given the meaning of the attributes.
3. Confusion matrices and the requested quality measures for all three models (single tree, bagging, random forest).
4. A discussion of whether the differences in accuracy (that is, the proportion of correct predictions) found on the test set are statistically significant. Find a statistical test that is suited for this purpose.

You are *not* supposed to describe the tree algorithm or its implementation in the report.

Handing in the assignment

You should hand in:

1. The documented program code (a `.py` file for Python, a `.R` file for R).
2. A `.pdf` file of the report.

Put your names and student numbers at the top of the code file, and on the first page of the report. The documentation should provide the following information:

1. Name of the function.
2. Names and types of its input arguments.
3. The result returned by the function.
4. A short description of what it does.

The main functions should be called `tree_grow` and `tree_pred`, and should be the first two functions in the code file. Then list `tree_grow_b` and `tree_pred_b`. Any other required functions that you have written should be listed below that.

Grading

The following considerations are taken into account to determine the grade for this assignment:

- Does the program work, and does it return the correct result?
- Efficiency of the implementation.
- Has the code been properly documented?
- Quality of the report.

Some Hints

- First read "Getting started with the assignment" on the course web page, and make the practice assignments.
- To test your algorithm, first apply it to the credit scoring data set used in the lectures. With `nmin = 2` and `minleaf = 1` you should get the same tree as presented in the lecture slides.
- For a more elaborate test, use the Pima indians data (see the course webpage). If you grow the tree on the complete data set with `nmin = 20` and `minleaf = 5`, and you use this tree to predict the training sample itself, you should get the following confusion matrix:

Class \ Pred	0	1
0	444	56
1	54	214

  If the confusion matrix produced by your algorithm differs substantially from this one, there is probably an error in your code. There might be slight differences due to different orders in processing the attributes when computing the quality of splits, or due to other minor variations in the code.
- We have shown in the lecture slides that optimal splits can only occur at the borders of segments. Is this still true for the best split *that meets the* `minleaf` *constraint*?