Data Mining 2018
Assignment 1: Classification Trees, Bagging and Random Forests

Instructions

This assignment should be completed by teams of two students.
Your solution consists of:

1. An `R` workspace (with extension `.RData`) containing the `R` functions that together make up your program.
2. A flat text file containing the documented program code, and
3. A PDF file containing a short report (about 4 pages) of your analysis.

Put your names and student numbers on the first page of the report. The assignment should be handed in through *EduCode* (see the course web page).

Part 1: Programming

Write a function in R 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.classify`. The function `tree.grow` has input arguments `x, y, nmin, minleaf`, and `nfeat`, in that order. Here `x` is a data matrix containing the attribute values. Each row of `x` contains the attribute values of one training example. You may assume that all attributes are numeric, including binary with values coded as `0` and `1`. `y` is the vector 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.classify` has input arguments `x` and `tr`. Here `x` is a data matrix 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.classify` has a single output argument `y`, which is the vector of predicted class labels for the cases in `x`.

For bagging (and random forests), you have to write two auxiliary functions called `tree.grow.bag` and `tree.classify.bag`. They are not much more than repeated applications of `tree.grow` and `tree.classify` respectively. The function `tree.grow.bag` 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 of length `m` containing the `m` trees. Finally, the function `tree.classify.bag` takes as input a list of trees and a data matrix `x` for which predictions are required. The function applies `tree.classify` 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` classifications. 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 algorithms you have made 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 (`as.numeric(post > 0)`). 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 few splits of the single tree. Assign to the majority class in the leaf nodes of this tree. Discuss whether the classification rule you get 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 found on the test set are statistically significant.

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

Handing in the assignment

The R workspace (file with extension `.Rdata`) that you hand in should be tested to work on a Windows machine. This file will be used by us to test the functions you have written.

Also put the functions you have written in a flat text file. Before you give the code of a function, provide the following information (start each line containing this information with the symbol #):

1. Name of the function.
2. Names and types of 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.classify`, and should be the first two functions in the file you submit. Then list `tree.grow.bag` and `tree.classify.bag`. Any other functions you have written that are needed to get things working should be listed below that. Your report on the analysis of the data set should be handed in in PDF format.

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 and elegance of the implementation.
- Quality of the report.

Some Hints

- First read "Getting started with the assignment in R" on the course web page, and make the practice assignments. Play around a little bit with R before you start with the "real work".
- During tree construction, a node in the tree is in fact "nothing more" than a subset of the training examples. Such a subset can be represented by a vector of *row numbers* of the observations contained in the subset.
- To test your algorithm you could 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 from the UCI machine learning repository. 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 (row: true class, column: predicted class):

	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.
- 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*?