# Machine Learning - Homework 1

Tim Kmecl, 13.3.2024

## Implementation

### Classification Tree

TreeNodes can be of two kinds – leafs, which only contain an information about what to return as prediction, and others, which contain threshold, column index and the two children nodes. The function predict returns the saved prediction for leaves, or recursively the prediction of either the left or the right node. Inside the class Tree, I implemented a function that calculates the Gini index for a given numpy matrix of data, and a function that calculates the gain given the current Gini, a threshold, and a column to use for splitting – it subtracts the given current Gini from the weighted average Ginis for splits (weighted by the amount of samples on each side of threshold)

The tree is built recursively. On every step, if every sample belongs to the same class or if amount of samples is less than the minimum, it returns a TreeNode leaf predicting the majority class. This would also happen in the case no split produces positive gain. Current Gini is calculated. To check all splits, the function loops over all the columns returned by get_candidate_columns at that step of recursion. In each iteration, that column is sorted, and the half-point between every two non-equal consecutive samples is considered as a potential threshold. If gain is better then the previous best, the threshold and the column id are saved. The data is then split into two based on this, two sub-trees are recursively built, and the function then returns a TreeNode with the threshold, column id and the two recursively built subtrees.

As a sanity check, I tried building trees on the train dataset both using all the features, or by using just a subset. In the second case the misclassification rate is always lower, which is the desired behaviour, since the best feature to split by may not be available.

### Random Forest

RandomForest builds the forest by bootstrapping n times. Every time it saves the indices of samples in the bag, those not in the bag (OOB) and the tree (TreeNode) built on that bag. It then returns RFModel containing all those arrays together with a copy of the data. For prediction, RFModel loops over all the trees, averages their predictions and then rounds them. When testing it on the provided data, it gives a way better result (details in the next section) then just a single tree, and increasing the amount of trees in the forest steadily improves the result, which speaks in favour of correctness.

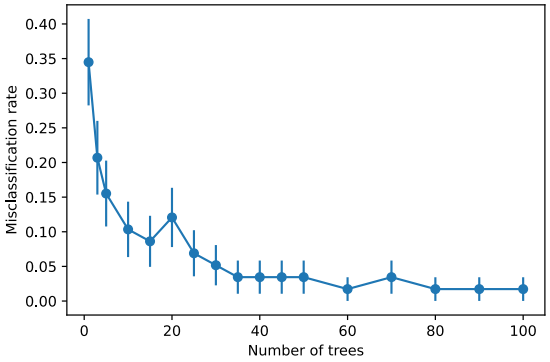
For variable importance, it goes over every feature separately in a loop. For each, it loops over all the bags. It first calculates the misclassification rate for the corresponding tree on for the corresponding OOB data. Then it shuffles the column of that feature, and calculates misclassification rate again. The final variable importance is the average of differences between those two rates.

## Using the dataset

I calculated the misclassification rate by counting the number of samples in the set where the prediction differs from the truth. To quantify the uncertainty, the standard deviation of the singular errors is first calculated. Since misscl. rate is the average of the errors, the standard deviation of it is obtained by dividing that standard deviation by the square root of the amount of samples. I use this to report uncertainty.

Missclasification rate of the tree is equal to 19.0% with deviation of 5.1%. For the forest, it is 1.7% with deviation of 1.7% (there is only one missclassified sample)

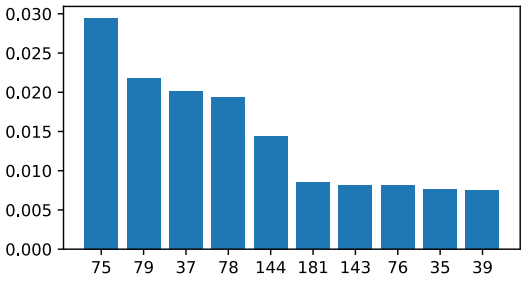
I tried building forests with 2, 5 and 10 trees, then by 5 more until 50, and then by 10 more until a 100. Their misscl. rates together with uncertainties are shown in the plot below.



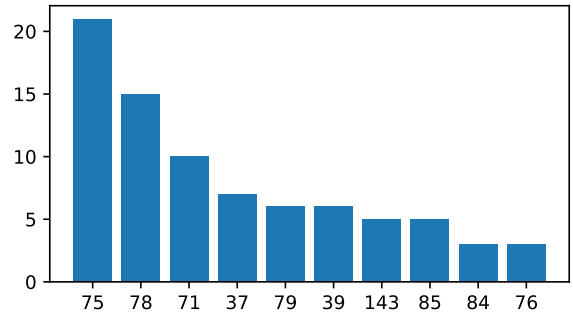

We can see that the performance quickly increases – quickly to around n=40, then more slowly, however the increase is not completely monotonous due to randomness. The analysis could be improved by averaging the rates over different random seeds.

## Variable importance

A new random forest was built on the entire dataset. I calculated the variable importance as described earlier. We can see in the last plot that some of them actually have negative importance, however they are few and their magnitude is small compared to the positive ones. 10 most important ones are shown:




I also built 100 non random trees 100 different random samples from the entire dataset, of size 2*sqrt(n), where n is the number of all samples, and checked how often each variable is in the root – since I only needed the root, building the tree could be stopped after one split, which sped things up. 10 most common ones are:




By varying the amount of samples in the train data for the trees, different results are obtained. Increasing the amount of samples increases the frequency of those variables that already have higher frequencies, and vice versa. Decreasing it reduces the amount of variables that appear in roots at all.

Frequency of variables in roots is highly correlated with their variable importance. Those that have the most importance appear more often, and those with low importance don't appear at all. Variables with low importance do not contain much information for classification, therefore they don't appear often as the root, and vice-versa. The comparison is shown here: