Machine Learning

Notes Part 4, HSG-MiQEF, Spring 2022

David Preinerstorfer

Last updated on 2022-03-16 19:38:43

Contents

Goals:

The goals of this lecture are to study the following methods and concepts:

- Regression and classification trees; bagging, wisdom of crowds, random forests, and boosting.
- Neural networks.

This lecture draws on Chapters 6-7 of Richter [2019] and Chapters 9-11 of Hastie et al. [2009].

1 Trees

1.1 Decision trees and their terminology

In regression and classification problems, the optimal decision rules depend on properties of the conditional distribution of Y given $\boldsymbol{X}$. In the previous lectures, we took a look at various methods to approximate optimal decision rules from data. One important class of algorithms which we did not cover yet are called decision trees, more specifically: classification and regression trees (CARTs).

The essential idea behind a decision tree is to efficiently create (in a data-driven way) a **finite partition** of the feature space $\mathbb{R}^p$ by "growing a tree" (cf. the description box below) and then to fix a decision on each cell in the partition (i.e., on each so-called "leaf"). Recall that a *finite partition* of $\mathbb{R}^p$ is nothing else than a collection of finitely many non-empty sets $A_1, \ldots, A_m$ which satisfy $A_i \cap A_j = \emptyset$ for all pairs $i \neq j$ (i.e., they are pairwise **disjoint**) and satisfy

$$\mathbb{R}^p = A_1 \cup \ldots \cup A_m$$

(i.e., there union equals the whole space). For a new feature vector $\boldsymbol{x}$, say, one would then check into which part of the partition it falls, and would then act according to the decision that was fixed on that part of the partition.

On a general level, to split the data, i.e., to generate a partition (or more colorful: to grow a tree), one does the following (we here focus on the case of binary decision trees, where, at each stage, the splitting is done in only two groups; generalizing the heuristic to more groups is straight-forward):

Grow a tree

1. Choose a feature, that is, a coordinate $j \in \{1, \ldots, p\}$. Call this feature the **split index**.
2. Split the data into two parts

   $$\{i : X_{ij} < s\} \quad \text{and} \quad \{i : X_{ij} \geq s\},$$

   choosing the **split point** s in such a way that the outcomes Y_i on the two sets are "well approximated" by carefully chosen constants c_1 and c_2, say, respectively.
3. Continue this splitting procedure on the just-obtained parts, where we may also decide not to split up a part anymore; in the latter case, the carefully chosen constants would constitute our decision.

If one proceeds in that way, one obtains a partition of the feature space (essentially by partitioning the training sample), and on every partition one obtains a value that intends to approximate the average outcome

(which could be the class in a classification problem or the conditional expectation in a regression problem) on that part of the partition. The best way to illustrate a tree, is probably to draw a diagram such as the one shown in Figure 1. The diagram is not taken from a particular training sample. Its mere purpose is to illustrate how a tree looks like, and (further below) which partition of the feature space corresponds to that tree.

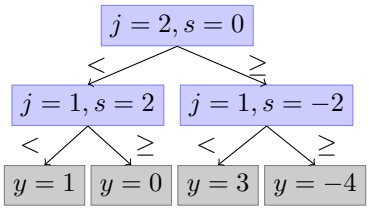


Figure 1: Example of a decision tree.

The tree given in Figure 1 consists of several splits. The feature space is split into the four sets

$$A_1 = \{\boldsymbol{x} : x_2 < 0, x_1 < 2\}, \quad A_3 = \{\boldsymbol{x} : x_2 \geq 0, x_1 < -2\}$$
$$A_2 = \{\boldsymbol{x} : x_2 < 0, x_1 \geq 2\}, \quad A_4 = \{\boldsymbol{x} : x_2 \geq 0, x_1 \geq -2\}.$$

A graphical representation of the partition (A_1 (red), A_2 (green), A_3 (blue), A_4 (pink)) is shown in Figure 2

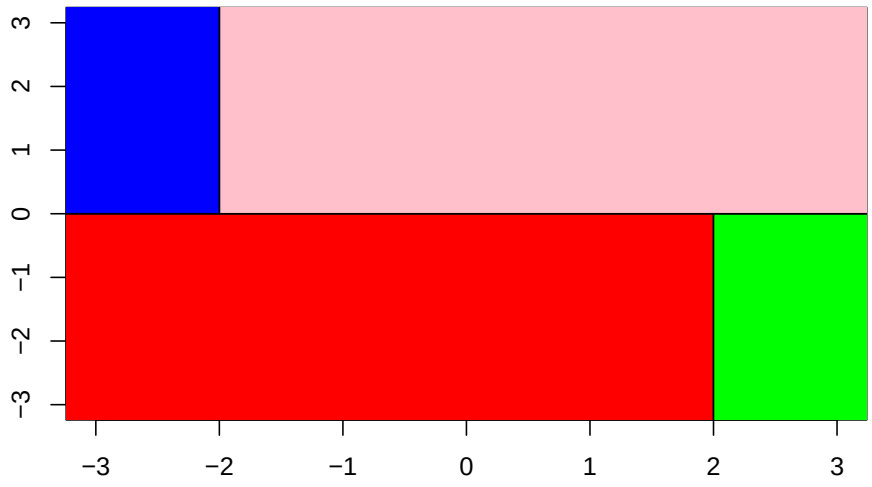


Figure 2: Partition corresponding to the decision tree in Figure 1.

On A_1 we predict $y(1) = 1$, on A_2 we predict $y(2) = 0$, on A_3 we predict $y(3) = 3$ and on A_4 we predict

$y(4) = -4$. In particular, writing $\mathbb{1}_A(\boldsymbol{x})$ for the function that assigns 1 if $x \in A$ and is 0 else (that function $\mathbb{1}_A$ is called the **indicator function** of the set A), we can write the decision rule corresponding to the above decision tree as

$$f(\boldsymbol{x}) = \sum_{i=1}^{4} y(i)\mathbb{1}_{A_i}(\boldsymbol{x}).$$

In general, a decision tree T, say, will partition the feature space in $\#T$ cells, i.e.,

$$\mathbb{R}^p = \bigcup_{i=1}^{\#T} A_i,$$

and the decision rule corresponding to such a tree will then shave the general form

$$f_T(\boldsymbol{x}) = \sum_{i=1}^{\#T} y(i)\mathbb{1}_{A_i}(\boldsymbol{x}).$$

Note that the number $y(i)$ would correspond to the class label assigned to every feature vector in A_i in case of a classification problem, whereas in a regression problem this would be our guess of the conditional expectation of Y given that $\boldsymbol{X}$ falls in A_i.

Note further that the decision rule we get by using a decision tree is always a "step function". The values it takes on each element A_i of the partition is *constant*.

There is some terminology we have to fix. We use the tree in Figure 1 for illustration:

1. The cells (in gray and blue/violet) are called **nodes**.
2. The arrows are called **edges**.

There are **two types of nodes**:

1. Nodes that are not split into further nodes are called **leaves** (which are the cells in gray).
2. Nodes that are split into further leaves are called **inner nodes** or **internal nodes** (of which there are three in the example (all of which are shown in blue/violet).

The "starting point" of the tree is (without much surprise) called the **root** of the tree (staying in this analogy, the tree in Figure 1 is actually standing on its head, i.e., is growing upside down).

In Figure 1 you can also see that every inner node is equipped with two quantities: (i) the split index j, signifying *which* one among the K feature is used to affect the splitting, and (ii) the splitting point s, which tells us *how* the splitting is done at this node.

Given a tree T, its **depth** is *defined* as the **longest path** that one can take from the root to one of the leaves. The depth of the tree in Figure 1 is obviously 2. One can also define the **depth of a node** as the length of

the path from the root to that leave. [One sometimes also defines the **height of a node**, which is the longest path from that node to a leave of the tree.]

1.2 Learning trees from data through penalized empirical risk minimization

Following the empirical risk minimization principle, and given a training sample $(Y_i, \boldsymbol{X}_i)$ for $i = 1, \ldots, n$, we would now like to find an "optimal" decision tree by minimizing the empirical risk (recall the loss functions introduced for regression and classification in Part 1 of the lecture notes)

$$\frac{1}{n} \sum_{i=1}^{n} L(Y_i, f_T(\boldsymbol{X}_i)).$$

For a given tree with partitions A_i for $i = 1, \ldots, \#T$ it is easy to determine the optimal values of the terminal decisions $y(i)$. Because f_T is necessarily constant and equals $y(i)$ on the partition A_i, it is optimal to do the following:

- **Regression problem** (with squared error loss): choose $y(i)$ as the sample average over all Y_i such that the corresponding feature vector $\boldsymbol{X}_i$ is in A_i.
- **Classification problem**: choose $y(i)$ as that class which is most frequent among all outcomes with $\boldsymbol{X}_i \in A_i$; that is, we decide by majority vote.

The most difficult thing in fitting a tree therefore is to find the right splitting variables and values, i.e., to find the right partition.

In order not to overfit the data by solving

$$\frac{1}{n} \sum_{i=1}^{n} L(Y_i, f_T(\boldsymbol{X}_i)),$$

one typically imposes a **maximal depth** of the tree sought and considers the restricted optimization problem obtained.[1] That is, one tries to find the tree that best fits the training sample with the additional constraint that its depth is not larger than some value specified by the user. Given such a maximal depth $t_{\max}$, say, denote the set of all trees of depth not larger than $t_{\max}$ by $\mathcal{T}(t_{\max})$. With this notation, we are trying to find the decision tree which corresponds to the following optimization problem

$$\arg \min_{T \in \mathcal{T}(t_{\max})} \frac{1}{n} \sum_{i=1}^{n} L(Y_i, f_T(\boldsymbol{X}_i)).$$

[1] As a bad example of a too complicated tree, reconsider the step function that was fit in Part 1 of the lecture notes. There, the step function clearly overfits the data. Note that such a step function can always be represented by a tree; admittedly a complicated one in that example.

Two problems arise:

- this optimization problem is very difficult to solve; note in particular that even though we fixed the maximal depth, the split points still vary, so the number of possible trees is infinite, in particular.
- if $t_{\max}$ is chosen too large, then one overfits, whereas one "underfits" if $t_{\max}$ is chosen too small.

To make the problem numerically easier, one often works with so-called **dyadic trees** where one assumes that the i-th feature takes its values only in $[a_i, b_i]$ and whenever one decides to use the i-th feature as the splitting index, one just uses the midpoint of the *current* interval as the split point (note that the interval may already have been split in internal nodes lying on the path between the root and the current node). If the depth is fixed as well, this results in only a finite number of possible trees that we need to consider in the optimization problem. Because we are comparing trees of different "sizes" in the optimization problem, it makes sense to introduce a tuning parameter that penalizes this aspect of a tree. To this end, one can work with penalty terms of the following form:

$$J_1(T) = \#T, \quad J_2(T) = \sqrt{\#T}.$$

One further criterion is how well the tree partition is adapted to the distribution of $\boldsymbol{X}$. In regions where this distribution puts more mass, we would like to have a finer partition, whereas in regions where this distribution puts less mass, we would like to have a coarser partition. One can achieve this by introducing a different penalty, which we do not specify in these notes, cf. Richter [2019] for details.

Given a penalty function, this then leads to the following penalized empirical risk minimization problem:

$$\arg \min_{T \in \mathcal{T}(t_{\max})} \frac{1}{n} \sum_{i=1}^{n} L(Y_i, f_T(\boldsymbol{X}_i)) + \lambda J(T).$$

One can further reduce the set over which the maximum is taken by only maximizing over all dyadic trees (as introduced above). Note that the decision rule obtained depends on the tuning parameter λ, which can be determined in a data-driven way through validation, e.g., cross-validation.

1.3 Pruning trees

The above maximization problems are hard and they might lead to substantial overfit. It is sometimes better to start with a comparably large tree (that is obtained through some greedy heuristic, which we shall detail below), and then to prune this tree, i.e., to cut off branches of the tree. For example, the tree in Figure 3 is

obtained by pruning the tree given previously in Figure 1.

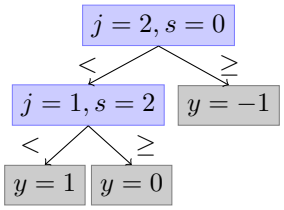


Figure 3: Pruned tree.

The partition corresponding to the pruned tree given in Figure 3 is shown in Figure 4.

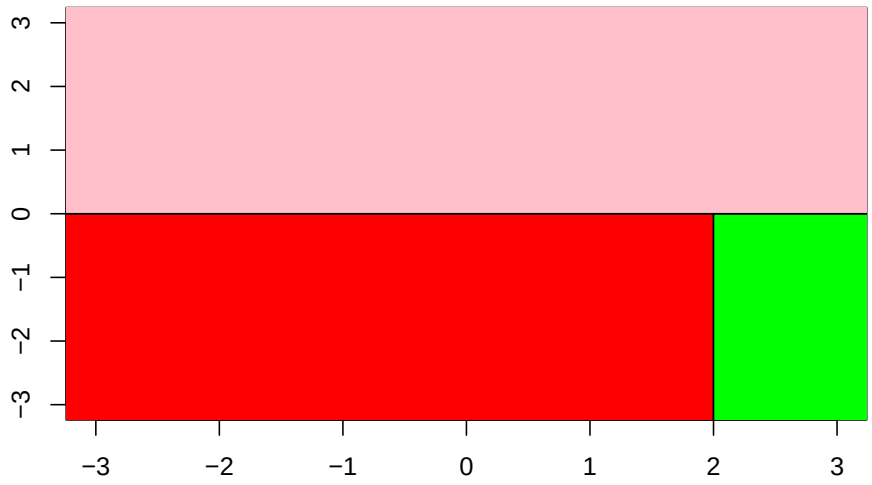


Figure 4: Partition of the pruned tree.

Visually, it is clear what is meant by pruning a tree. The tree obtained is just a simplification of the initially given tree. In finding the right tree that fits the data best, one then tries to solve the same optimization problem as above, but only considers trees which can be obtained through pruning an initially given, large tree T_0. Denoting any tree that can be obtained by pruning T_0 by $T \preceq T_0$, we can write the optimization problem as (sometimes called **cost-complexity pruning**)

Cost-complexity pruning

$$\arg\min_{T \preceq T_0} \frac{1}{n} \sum_{i=1}^{n} L(Y_i, f_T(\boldsymbol{X}_i)) + \lambda J(T).$$

Again, the decision rule obtained depends on the tuning parameter λ, which can be determined in a data-driven way through validation, e.g., cross-validation.

1.4 Greedy algorithms

Now that we understand what a decision tree is and how optimization problems look like, we can go on to take a look at *algorithms* that allow us to obtain such trees from data. We have already seen that once we have a partition, the determination of the predictions $y(i)$ that we make on each leaf is straight-forward and only depends on whether we want to obtain a regression or a classification tree. Therefore, the main focus is on learning the partition (i.e., learning the split indices and points) .

The main algorithm to grow a tree from training data is to (i) obtain a tree by growing one in a greedy manner (see below) and (ii) to then prune the tree by proceeding as outlined in the previous section.

1.4.1 Regression trees

To grow a regression tree greedily, we do the following:

1. Starting with all of the training sample, we seek the first splitting variable j and the first split point s in such a way as to solve
$$\min_{j,s} \left[\min_{c_1} \sum_{i:X_{ij}<s} (y_i - c_1)^2 + \min_{c_2} \sum_{i:X_{ij}\geq s} (y_i - c_2)^2 \right].$$
2. Note that the inner minimizations over c_1 and c_2 are solved by the average outcomes over all $\boldsymbol{X}_i < s$ and $\boldsymbol{X}_i \geq s$, respectively.[2]
3. Having split the data, we obtain two new data samples, and we repeat the splitting procedure on each of them.

The question now is how large we should grow the tree? In general, one uses heuristics, such as continuing so long until some maximal depth is reached, or the number of training data that falls in each terminal node is smaller than a given value.

1.4.2 Classification trees

To grow a classification tree greedily, we proceed similarly, but we have to adjust the way the splitting is done (because the loss function is now a different one). Recall that for a given partition the classification of the tree equals that class which has the majority among the training sample with feature vectors in that part

[2]This is a special case of the first theorem established in Part 1 of the lecture notes.

of the partition.

One can then

1. start with the whole dataset, and choose the splitting variable j and split point s in such a way that the misclassification error is lowest, i.e., seek
$$\min_{j,s}\left[\min_{c_1}\sum_{i:X_{ij}<s}L(y_i,c_1)+\min_{c_2}\sum_{i:X_{ij}\geq s}L(y_i,c_2)\right].$$
2. split the sample and continue in the same way on each of the new parts of the data.

Note that we use the misclassification rate in the above minimization, which is natural when working with the loss function L. There are also other loss functions one can use, e.g., a Gini index or a cross-entropy measure. These lead to different trees and can typically be chosen by switching a parameter in the implementation of the algorithm. Section 9.2.3 in Hastie et al. [2009] contains some details.

2 Methods for improving and combining regression or classification procedures

In this section we will take a look at several bias or variance reduction methods. The methods are not necessarily restricted to trees, but also apply (potentially after slight adjustment) to the other methods we have already covered.

2.1 Bagging

Bagging is an abbreviation for "bootstrap aggregation". If you are familiar with the bootstrap: the idea is the same. Let's consider how that works for the regression problem:

1. We use the training sample $(Y_i, \boldsymbol{X}_i)$ $(i = 1, \ldots, n)$ to generate "new" datasets $(Y_i^{(b)}, \boldsymbol{X}_i^{(b)})$ $(i = 1, \ldots, n)$ and $b = 1, \ldots, B$. How do we obtain these new datasets? We just draw them (with replacement) from our training sample!
2. To obtain an "improved decision rule" from an algorithm available (e.g., growing a tree by a greedy algorithm and then pruning it; with a **fixed** penalty λ) determine the decision rule for each of the "new datasets" to obtain B decision rules $\hat{f}_{n,b}$ for $b = 1, \ldots B$.
3. Finally, obtain the decision rule
$$\hat{f}_n^{(bagg)}(\boldsymbol{x}) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}_{n,b}(\boldsymbol{x}).$$

One can show that this approach leads to a reduction of variance of the resulting decision rule, while we still approximate the same target. Heuristically, if the cdf of the training sample is close to the true underlying distribution, sampling from the cdf will result in "new" datasets the distribution of which also has this property, which then results in improved decision rules.

For a classification problem the only thing we need to change is how we aggregate. In this case, we do not average, but we take

$$\hat{f}_n^{(bagg)}(\boldsymbol{x}) = \arg\max_{j=1,\dots,K} \#\{b \in \{1,\dots,B\} : \hat{f}_{n,b}(\boldsymbol{x}) = j\},$$

that is we determine the classification as the **majority vote** for $\boldsymbol{x}$ among all the new datasets we generated.

2.2 Wisdom of crowds and random forests

We will now take a look at random forests. Those methods may be considered as improved versions of the bagging algorithm specifically for decision trees. One underlying motivation is the so-called wisdom of crowds phenomenon, which tells us that one can perfectly learn when one combines many **weak learners**. This phenomenon also underlies Boosting, which we shall take a look at further below.

2.2.1 Wisdom of crowds and aggregation of weak lerners

Fix a $\boldsymbol{x}_0 \in \mathbb{R}^p$ and recall that the bagging algorithm for classification is defined via

$$\hat{f}_n^{(bagg)}(\boldsymbol{x}_0) = \arg\max_{j=1,\dots,K} \#\{b \in \{1,\dots,B\} : \hat{f}_{n,b}(\boldsymbol{x}_0) = j\}.$$

That is, the class membership that is decided for $\boldsymbol{x}_0$ is determined through *majority vote* among the decision rules determined from the bootstrap samples $b = 1,\dots,B$.

A similar situation would be given if we had B decision rules $\hat{g}_{n,b}(\boldsymbol{x}_0)$ for $b = 1,\dots,B$, which are **identically distributed** (which is the case for the bootstrapped ones above) and which are **independent** (which is, however, *not* the case for the bootstrapped ones). One could then define the aggregated rule (analogous to the bagged decision rule)

$$\hat{g}_n^{(agg)}(\boldsymbol{x}_0) = \arg\max_{j=1,\dots,K} \#\{b \in \{1,\dots,B\} : \hat{g}_{n,b}(\boldsymbol{x}_0) = j\}.$$

Interpret the individual decision rules $\hat{g}_{n,b}$ as **weak learners**, i.e., they do not have to be very accurate, but we have many of them. We could ask the question:

- if we increase B (the number of weak learners available), does $\hat{g}_n^{(agg)}(\boldsymbol{x}_0) \to f^*(\boldsymbol{x}_0)$ hold in probability, f^* denoting the optimal decision rule (i.e., the risk minimizer we have derived for regression and

classification in Part 1 of the lecture notes)?

That is, does the rule get better as B increases, and does it eventually converge to the optimal decision rule?

Surely, we have to assume something about the individual rules, in particular, the above convergence does not hold for data-independent rules. It turns out that if the rules have **some** weak predictive power (i.e., are slightly better than flipping a coin), the convergence holds. This is the **wisdom of the crowds** phenomenon. It tells us that when combining many weak learners through aggregation, one can obtain a very **strong** decision rule, i.e., on that is arbitrarily close to the optimal one.

Theorem 1. *Consider a classification problem with two classes, which we call* 0 *and* 1*, respectively. Fix* $\boldsymbol{x}_0 \in \mathbb{R}^p$*, assume that* $f^*(\boldsymbol{x}_0) = 1$ *(without loss of generality), and suppose that*

$$\mathbb{P}(\hat{g}_{n,b} = 1) =: a_n > 1/2.$$

Then $\hat{g}_n^{(agg)}(\boldsymbol{x}_0) \to f^*(\boldsymbol{x}_0)$ *in probability as* $B \to \infty$ *(grant the independence and identical distributedness assumed above hold).*

Proof. We assumed that $\hat{g}_{n,b}(\boldsymbol{x}_0)$, $b = 1, \ldots, B$ are i.i.d. The law of large numbers therefore implies that as $B \to \infty$ we have (in probability)

$$m_B := \frac{1}{B} \sum_{i=1}^{B} \mathbf{1}\{\hat{g}_{n,b}(\boldsymbol{x}_0) = 1\} \to a_n.$$

Furthermore, we note that if $m_B > 1/2$, then (by definition) the aggregated rule equals 1. Therefore,

$$\mathbb{P}(\hat{g}_n^{(agg)}(\boldsymbol{x}_0) = 1) \geq \mathbb{P}(m_B > 1/2) = \mathbb{P}([m_B - a_n] + [a_n - 1/2] > 0).$$

Abbreviate $\varepsilon := a_n - 1/2$ and write

$$\begin{aligned}
\mathbb{P}([m_B - a_n] + [a_n - 1/2] > 0) &= \mathbb{P}([m_B - a_n] + \varepsilon > 0, |m_B - a_n| \geq \varepsilon) + \mathbb{P}([m_B - a_n] + \varepsilon > 0, |m_B - a_n| < \varepsilon) \\
&\geq \mathbb{P}([m_B - a_n] + \varepsilon > 0, |m_B - a_n| < \varepsilon) \\
&= \mathbb{P}(|m_B - a_n| < \varepsilon) \to 1,
\end{aligned}$$

the convergence following from $m_B \to a_n$ in probability. □

The above result is interesting, because it shows that combining many weak learners leads to a decision rule that gets perfect when the number of weak learners gets infinite. However, the question is how to obtain said weak learners. In particular, when we reconsider bagging algorithms, we see that the independence condition is not met, as the individual decision rules are all dependent on the original training sample. Therefore, we

may try to "decrease the degree of dependence" while keeping the same expectation. This leads to what is called a **random forest**.

2.2.2 Random forests

Random forests try to decrease the degree of dependence through two modifications concerning how the trees are grown:

1. Instead of the normal greedy algorithm that is used to grow the tree, one does not consider every feature as a potential split variable in every step, but instead chooses the split variable from a random subset $S \subseteq \{1, \ldots, K\}$ of the set of all features (that is re-drawn at every splitting step). The size of S is denoted by m and it is assumed that $m < n$. At every splitting step, we therefore **force the tree only to use a subset of the features** available.
2. **Reduce the sample size** in the bootstrap samples from n to $A_n < n$. For example, if $A_n = n/2$ it can happen that the trees grown on two bootstrap samples are independent. In any case, this further decreases the dependence between the individual decision functions that are then aggregated.

The individual decision rules are then aggregated by taking the average (regression case) or by majority vote (classification case), in the same way as in the previous section on bagging.

2.3 Boosting

In the spirit of the wisdom of crowds phenomenon, the idea of boosting is to iteratively improve a "weak learner" into a strong one.

Call the set of simple decision rules we make use of in the boosting algorithm $\mathcal{C}$, which is a collection of functions from the feature space $\mathbb{R}^p$ into $\{-1, 1\}$ (assuming that there are only two classes, which we call -1 and 1, respectively). For example, you could think of

$$\mathcal{C}_{stump} = \{c\mathbb{1}_{\{x_j < s\}} - c\mathbb{1}_{\{x_j \geq s\}} : j \in \{1, \ldots, p\},\ s \in \mathbb{R},\ c \in \{-1, 1\}\},$$

which is the set of all decision trees with depth 1 and only 2 terminal nodes. Such trees are called stumps (note that the free parameters that vary between such trees are: the splitting variable j used at the root, the splitting value s used at the root, and the output assigned at the two terminal nodes which is either 1 or -1). *None* of the classifiers in that set will typically be performing very well in isolation, i.e., they are "weak learners".

Boosting aims at combining multiple elements of $\mathcal{C}$ in the best way possible. That is, we are looking for a

classifier of the form

$$G(\boldsymbol{x}) = \text{sign}\left(\sum_{i=1}^{m} \alpha_i C_i(\boldsymbol{x})\right),$$

where each C_i is an element of $\mathcal{C}$, i.e., a special instance of the weak learner. The idea is to start with a first classifier, then to add another one that tries to classify those datapoints that were not well classified by the first, and so on. In this way, we try to gradually improve the decision rule.

There is one immediate question: how to find the weights $\alpha_1, \ldots, \alpha_m$? One very popular and successful algorithm due to Yoav Freund and Robert E. Schapire is called (discrete) **AdaBoost** (many improved versions of the algorithm are available in the literature). This algorithm proceeds as follows (cf. also the description in Section 10 of Hastie et al. [2009] p. 339); the loss function L here is the usual one that we use in classification problems (i.e., it is 1 if the inputs do not coincide and is 0 else) and the classification problem is binary (i.e., Y takes values in $\{-1, 1\}$):

AdaBoost

1. Initialize the weights $w_i = 1/n$ for $i = 1, \ldots, n$.
2. For $j = 1, \ldots, M$ do the following
   - Select a classifier in $\mathcal{C}$ to the training sample using *weighted* empirical risk minimization, i.e., solve
   $$\arg\min_{C \in \mathcal{C}} \sum_{i=1}^{n} w_i L(y_i, C(\boldsymbol{X}_i))$$
   to obtain the classifier $C_j \in \mathcal{C}$, say.
   - Compute the weighted misclassification rate of C_j, i.e., compute
   $$\text{err}_j = \sum_{i=1}^{n} w_i L(Y_i, C_j(\boldsymbol{X}_i)) \Big/ \sum_{i=1}^{n} w_i.$$
   - Compute
   $$\alpha_j = \log\left((1 - \text{err}_j)/\text{err}_j\right).$$
   - For $i = 1, \ldots, n$ update w_i to $w_i \exp(\alpha_j L(Y_i, C_j(\boldsymbol{X}_i)))$.
3. Output the classifier $G(\boldsymbol{x}) = \text{sign}\left(\sum_{j=1}^{M} \alpha_j C_j(\boldsymbol{x})\right)$.

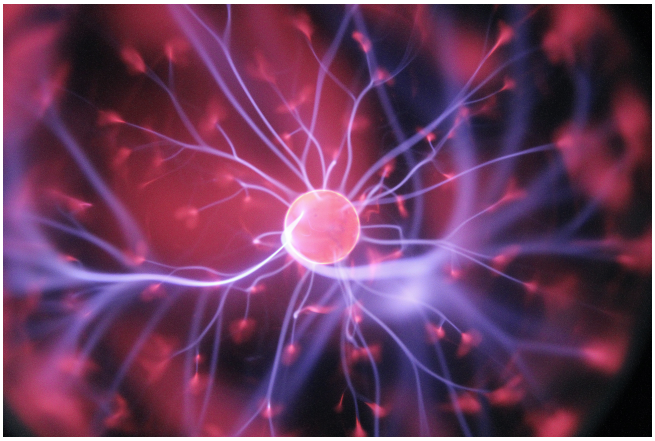
3 Neural Networks

We have already seen in the section on support vector machines that incorporating nonlinear transformations of the feature vectors can be highly relevant in classification problems. There, the choice of a nonlinear function at the end boils down to the choice of a kernel, which allowed us to solve a problem concerning the

dimensionality of the optimization problem considered.

Another, very popular and successful, way to model nonlinearities are **(artificial) neural networks**, which in particular play an important role in the context of **deep learning** in the form of what is called a **deep neural net**. The underlying idea is the following: one starts with a feature vector $\boldsymbol{x}$, say, and then performs a simple **transformation** to the feature vector. Then, one feeds the outcome of this transformation into a next **layer** of transformations. The process is repeated and finally stopped after a few layers of transformations (or many layers of transformations, which is then called deep learning). The final result of the iterated transformations of the feature vector is fed into a function that maps that vector into real values (regression case) or class labels (classification case).

On a heuristic level, in the spirit of **artificial intelligence**, this mechanism tries to emulate how information is transmitted through neurons (or nerve cells), special types of cells that transmit information to other neurons through connections called synapses, or to other types of cells, e.g., muscle cells.



In order to make the above description mathematically precise, we need some ingredients:

- we have to define the transformations used, and
- we have to make specific how the final output is obtained from the transformations.

In any case, we need to specify how a final decision rule corresponding to such a **neural net** would look like.

To this end, one needs to specify:

- the number of **layers** L one wants to use,
- weights matrices $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{L+1}$,

- shift or "bias" vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_L$,
- an **activation function** $\sigma : \mathbb{R} \to \mathbb{R}$.

Starting with a feature vector $\boldsymbol{x}_0$, in every layer $l = 1, \ldots, L$, the feature vector is updated via the transformation

$$\boldsymbol{x}_{l+1} = \sigma(\boldsymbol{v}_l + \boldsymbol{W}_l \boldsymbol{x}_l);$$

here $\sigma(\boldsymbol{v}_l + \boldsymbol{W}_l \boldsymbol{x}_l)$ denotes the component-wise application of the function σ, i.e., for a p-vector $\boldsymbol{x}$ the symbol $\sigma(\boldsymbol{x}) = (\sigma(x_1), \ldots, \sigma(x_p))'$. Note that $\boldsymbol{x}_{l+1}$ may not be of the same dimension as $\boldsymbol{x}_l$ (but must be such that their dimension are such that the multiplication operations with the weights matrices are well defined). Denote the **latent feature dimensions** used in the construction by $\boldsymbol{q} = (q_1, \ldots, q_{L+1})'$ in what follows. In the last step and in the **regression case** one always uses $q_{L+1} = 1$ and obtains the output (i.e., the approximation to the conditional expectation of Y given $\boldsymbol{X} = \boldsymbol{x}$) via

$$y = g(\boldsymbol{x}) = \boldsymbol{W}_{L+1} \boldsymbol{x}_L;$$

in the **classification case** one obtains K outputs (hence, in this case it must hold that $q_{L+1} = K$; sometimes $q_{L+1} = K - 1$, when a reference category is chosen), i.e., K predicted class membership probabilities $g_1(\boldsymbol{x}), \ldots, g_K(\boldsymbol{x})$, and then chooses the class which maximizes those, i.e., $\arg\max_{i=1,\ldots,K} g_i(\boldsymbol{x})$, as the final decision.

One can also write the function obtained (somewhat complicatedly) in a non-iterative way. If one does that, one sees that the decision rule obtained through a neural network with the above specified ingredients is a function $g : \mathbb{R}^p \to \mathbb{R}^{q_{L+1}}$ of the form

$$g(\boldsymbol{x}) = \boldsymbol{W}_{L+1} \sigma\left(\boldsymbol{v}_L + \boldsymbol{W}_L \sigma\left(\ldots\left(\boldsymbol{v}_2 + \boldsymbol{W}_2 \sigma\left(\boldsymbol{v}_1 + \boldsymbol{W}_1 \boldsymbol{x}\right)\right)\right)\right).$$

We make the following observations concerning the updating process $\boldsymbol{x}_{l+1} = \sigma(\boldsymbol{v}_l + \boldsymbol{W}_l \boldsymbol{x}_l)$:

- the weights matrix $\boldsymbol{W}_l$ encodes the amount of the signal that is passed from $\boldsymbol{x}_l$ to each coordinate of $\boldsymbol{W}_l \boldsymbol{x}_l$.
- the bias vector can further translate $\boldsymbol{W}_l \boldsymbol{x}_l$, and
- the activation function σ, which is applied to each coordinate, regulates whether the impulse encoded in the transformed feature vector $\boldsymbol{v}_l + \boldsymbol{W}_l \boldsymbol{x}_l$ is passed onto the next layer or not.

To understand the last item, it is important to know which activation functions are used in practice. Note that we are still trying to formalize the idea of information being passed onto a next layer of neurons. One therefore uses functions that are monotonically increasing in the input. Some examples are given in Figure 5

below: (i) **sigmoid function** also called **logistic function** $\sigma(x) = 1/(1+e^{-x})$; (ii) **hyperbolic tangent** $\sigma(x) = \tanh(x)$; (iii) **ReLU** (stands for rectified linear unit) function $\sigma(x) = \max(0, x)$; (iv) **Leaky ReLU** $\sigma(x) = \max(x/10, x)$; (v) **ELU** $\sigma(x) = x$ if $x \geq 0$ and $\sigma(x) = \alpha(e^x - 1)$, else; ...

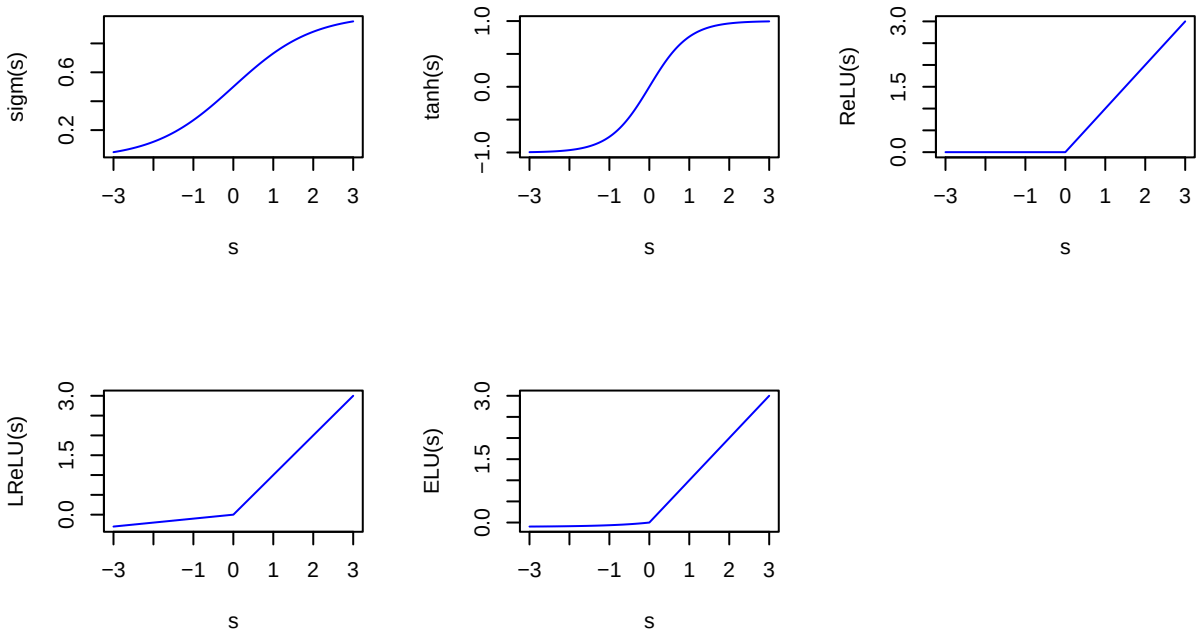


Figure 5: Different activiation functions.

For illustration purposes, a neural network is often depicted as shown in Figure 3 (regression case). A similar figure could also be drawn for the classification case, which we do not detail.

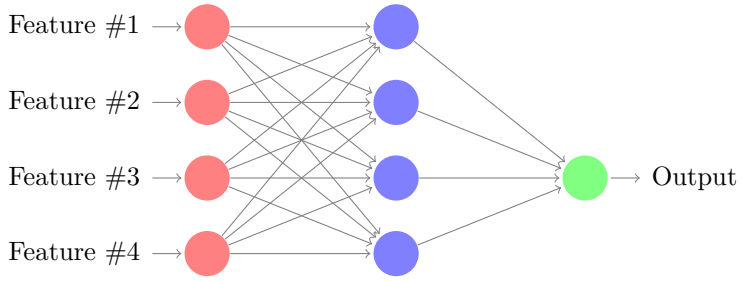


Figure 6: A neural network with one layer $L = 1$ for regression.

Building up decision rules in that way results in functions that are quite complicated if one wants to write them up. But in essence, they are only a certain class of nonlinear transformations of the feature vector $\boldsymbol{x}$ of the particular form

$$g(\boldsymbol{x}) = \boldsymbol{W}_{L+1}\sigma\left(\boldsymbol{v}_L + \boldsymbol{W}_L\sigma\left(\ldots\left(\boldsymbol{v}_2 + \boldsymbol{W}_2\sigma\left(\boldsymbol{v}_1 + \boldsymbol{W}_1\boldsymbol{x}\right)\right)\right)\right)$$

as we have seen above.

To learn those functions from data one fixes L and σ and the dimensions $\boldsymbol{q}$ and then learns weights matrices and the bias vectors from a training sample. One proceeds by penalized empirical risk minimization, where the penalization is over the weights matrices used in the construction of the neural net. We do not want these matrices to be too "complicated". Hence, similar as in the ridge estimator, we can for example sum the squares of all entries of the weights matrices involved in a given neural net, and use this as the penalty (typically one does not penalize the shift vectors; one could also use a Lasso type penalty, but then the computational complexity would be much higher). Call the penalty function J, which we shall leave unspecified below. One then proceeds as follows:

Neural network for regression

- Fix the number of layers L, fix the activation function σ, and fix the latent feature dimensions $\boldsymbol{q} = (q_1, \ldots, q_L)$.
- Denote the set of all neural networks with L layers, activation function σ and latent feature dimensions $\boldsymbol{q}$ by $\mathcal{F}(L, \boldsymbol{q})$ (the freely varying variables here are the weights matrices $\boldsymbol{W}_i$ and the bias vectors $\boldsymbol{v}_i$ for $i = 1, \ldots, L$).
- Fix a penalty term $\lambda \geq 0$ and solve the optimization problem

$$\hat{f}_{n,\lambda}^{NN} \in \arg \min_{g \in \mathcal{F}(L,\boldsymbol{q},\sigma)} \left\{ \frac{1}{n} \sum_{i=1}^{n} (Y_i - g(\boldsymbol{X}_i))^2 + \lambda J(g) \right\}.$$

The optimization problem in that procedure is typically solved by gradient descent (cf. the description of the method in Part 1 of the lecture notes). In the context of neural networks, this method is referred to as **back-propagation** due to the specific form gradient descent takes in that problem. A description of this optimization method in the present context can be found in Hastie et al. [2009] p. 395-397; the same chapter also contains comments on numerical issues and an additional data example.

The procedure for classification is similar and hence skipped.

4 Data example

Let's take a look at a dataset containing survival statistics for the titanic disaster, where we also drop some of the variables that have lots of missing values (for simplicity). The challenge here is to classify the outcome (survival) based on several passenger characteristics. We shall see how decision trees and related methods and neural networks can be implemented in `R` through readily available functions.

```
set.seed(123)
path <- 'https://raw.githubusercontent.com/guru99-edu/R-Programming/master/titanic_data.csv'
titanic <-read.csv(path, na.strings = "?", header = TRUE)
dim(titanic)
```

```
## [1] 1309   13
```

```
head(titanic)
```

```
##   x pclass survived                                            name    sex
## 1 1      1        1                   Allen, Miss. Elisabeth Walton female
## 2 2      1        1                  Allison, Master. Hudson Trevor   male
## 3 3      1        0                    Allison, Miss. Helen Loraine female
## 4 4      1        0            Allison, Mr. Hudson Joshua Creighton   male
## 5 5      1        0 Allison, Mrs. Hudson J C (Bessie Waldo Daniels) female
## 6 6      1        1                              Anderson, Mr. Harry   male
##       age sibsp parch ticket     fare   cabin embarked
## 1 29.0000     0     0  24160 211.3375      B5        S
## 2  0.9167     1     2 113781 151.5500 C22 C26        S
## 3  2.0000     1     2 113781 151.5500 C22 C26        S
## 4 30.0000     1     2 113781 151.5500 C22 C26        S
## 5 25.0000     1     2 113781 151.5500 C22 C26        S
## 6 48.0000     0     0  19952  26.5500     E12        S
##                         home.dest
## 1                     St Louis, MO
## 2 Montreal, PQ / Chesterville, ON
## 3 Montreal, PQ / Chesterville, ON
## 4 Montreal, PQ / Chesterville, ON
## 5 Montreal, PQ / Chesterville, ON
## 6                     New York, NY
```

```
titanic <- titanic[,-1]
```

Next, we turn several variables in `titanic` (columns: 1, 2, 4, 6, 7, 8, 10, 11, 1) into factors

```
for(i in c(1, 2, 4, 6, 7, 8, 10, 11, 12)){
  titanic[,i] <- factor(titanic[,i])
}
```

Next, we drop the variables: `name`, `ticket`, `cabin` and `home.dest` because they contain many missing entries. We also drop all subjects for which we do not have all entries for the remaining variables.

```
titanic <- titanic[,-c(3, 8, 10, 12)]
titanic <- titanic[complete.cases(titanic),]
```

Our goal now is to classify passengers as to whether they survived the disaster or not. We will use a decision tree and fit a neural network and see which method gives better results.

As usual, we select a training sample and a testing sample.

```
n <- dim(titanic)[1]
tsa <- sample(1:n, size = floor(0.8*n), replace = FALSE)
titanic_tr <- titanic[tsa,]
```

To fit a tree, we can use the function `rpart` from the package **rpart** via `rpart(formula, data=, method='')`. If we specify method as `"class"`, we obtain a classification tree, whereas if we specify the method as `"anova"`, we obtain a regression tree. In this exercise, we are interested in obtaining a classification tree, therefore we specify this parameters as `"class"`. The decision tree that is built is shown in Figure 8.

```
library("rpart")
library("rpart.plot")
fit_t <- rpart(survived ~ ., data = titanic_tr, method = "class")
rpart.plot(fit_t)
```

We can now compute the confusion matrix (on the testing sample) for this tree using the function `predict(fitted_model, df, type = 'class')` as usual.

```
titanic_te <- titanic[-tsa,]
pre_t <- predict(fit_t, newdata = titanic_te, type = 'class')
conf_tab <- table(titanic_te$survived, pre_t)
conf_tab
```

```
##    pre_t
##       0   1
```

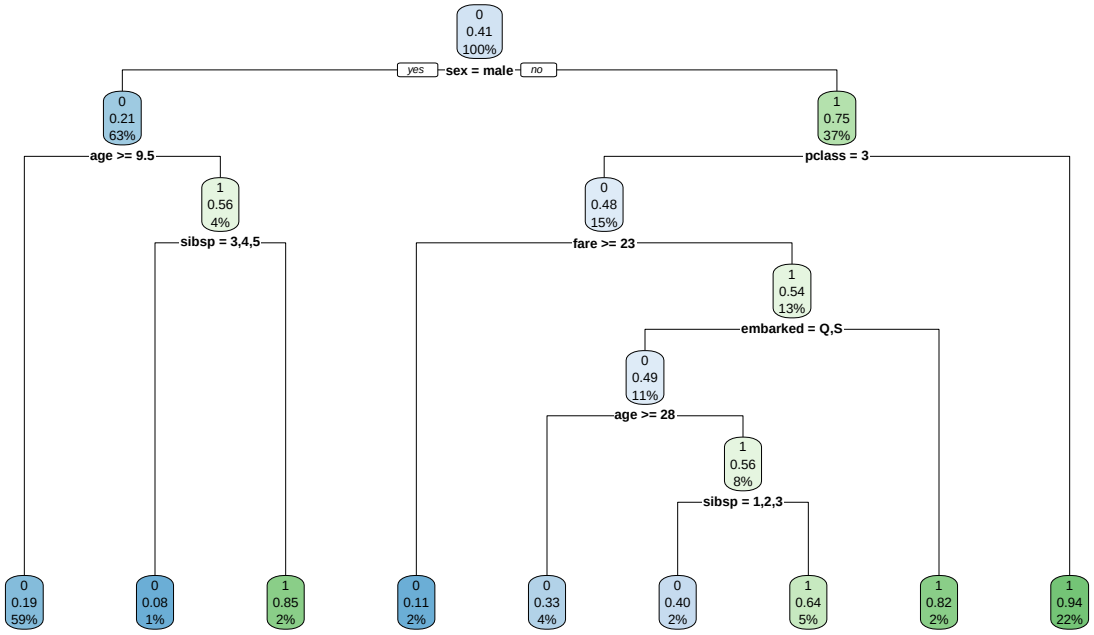


Figure 7: Titanic disaster survival tree.

```
##   0 117  11
##   1  21  60
```

```
sum(diag(conf_tab))/(sum(conf_tab))
```

```
## [1] 0.84689
```

We can further try to improve that by using bagging. This can be done using the package **ipred** (which stands for improved predictors) with its function `bagging` which is applied via `bagging(formula, data, subset, na.action=na.rpart, nbagg)`. Let's try that:

```
library("ipred")
fit_bag <- bagging(survived ~ ., data = titanic_tr, nbagg = 70)
fit_bag
```

```
##
## Bagging classification trees with 70 bootstrap replications
##
## Call: bagging.data.frame(formula = survived ~ ., data = titanic_tr,
```

```
##     nbagg = 70)
```

```
pre_tb <- predict(fit_bag, newdata = titanic_te, type = 'class')
conf_tab2 <- table(titanic_te$survived, pre_tb)
conf_tab2
```

```
##    pre_tb
##       0   1
##   0 102  26
##   1  23  58
```

```
sum(diag(conf_tab2))/(sum(conf_tab2))
```

```
## [1] 0.7655502
```

In this case, bagging does not lead to an improvement. Let's try whether random forests improve the classification performance. This can be done with the package **randomForest**, which also comes with a plot that allows us to assess the importance of the individual variables in the forest.

```
library("randomForest")
fit_rf <- randomForest(survived ~ ., data = titanic_tr,
                       na.action = na.roughfix, ntree = 2000,
                       importance = TRUE)
varImpPlot(fit_rf)
```

```
pre_rf <- predict(fit_rf, newdata = titanic_te, type = 'class')
conf_tab3 <- table(titanic_te$survived, pre_rf)
conf_tab3
```

```
##    pre_rf
##       0   1
##   0 113  15
##   1  20  61
```

```
sum(diag(conf_tab3))/(sum(conf_tab3))
```

```
## [1] 0.8325359
```

As we can see from the confusion matrix, we somehow cannot beat the first tree we built. However, the

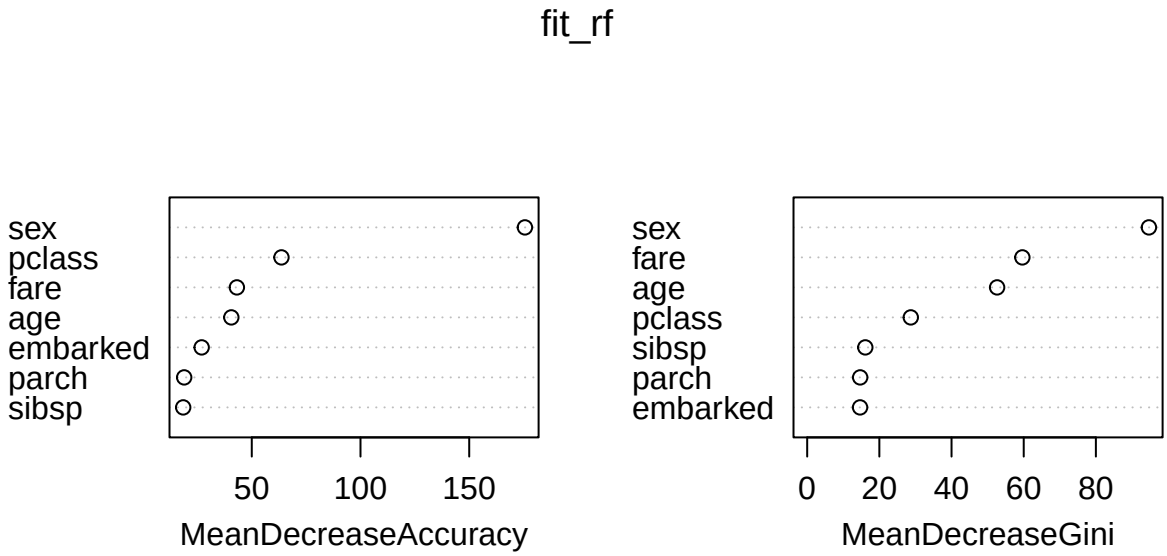


Figure 8: Variable importance in the random forest.

performance is better than the one of obtained through bagging

Finally, let's take a look at a neural net. We fit the network through the package **neuralnet**, in particular using the function `neuralnet`. This function allows us to specify the dimensions (and number) of hidden layers through the parameter `hidden = ...` and also allows us to choose the activation function through the parameter `act.fct = ...`. The neural network is fit by a numerical optimization procedure. The maximal number of steps used in that procedure is regulated through the parameter `stepmax = ...`. If the algorithm does not converge, there is no neural network that is returned by the function. One should then increase the maximal numbers of steps and try to run the function again.

```
library("neuralnet")
options(na.action='na.pass')
X <- model.matrix(survived ~ ., data = titanic)[,-1]
titanic_n <- data.frame("survived" = (titanic[,2] == 1)*1, X)
titanic_ntr <- titanic_n[tsa,]
titanic_nte <- titanic_n[-tsa,]
fit_nnet <- neuralnet(survived ~ ., data = titanic_ntr,
                      hidden = c(2, 2), act.fct =  "logistic",
                      linear.output = FALSE, stepmax = 1000000)
```

```r
pre_nnet <- (predict(fit_nnet, newdata = titanic_nte) > 1/2)*1
conf_tab4 <- table(titanic_nte$survived, pre_nnet)
conf_tab4
```

```
##    pre_nnet
##      0  1
##   0 93 35
##   1 20 61
```

```r
sum(diag(conf_tab4))/(sum(conf_tab4))
```

```
## [1] 0.7368421
```

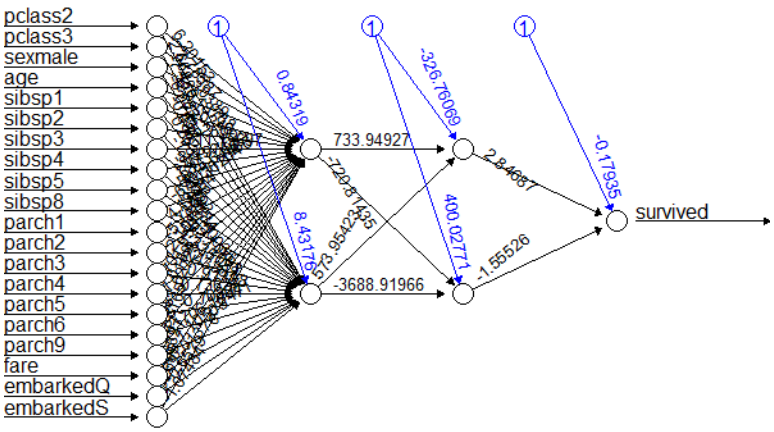
```r
fit_nnet_tanh <- neuralnet(survived ~ ., data = titanic_ntr,
                      hidden = c(2, 2), act.fct =  "tanh",
                      linear.output = FALSE, stepmax = 1000000)

pre_nnet_tanh <- (predict(fit_nnet_tanh, newdata = titanic_nte) > 1/2)*1
conf_tab5 <- table(titanic_nte$survived, pre_nnet_tanh)
conf_tab5
```

```
##    pre_nnet_tanh
##       0   1
##   0 126   2
##   1  78   3
```

```r
sum(diag(conf_tab5))/(sum(conf_tab5))
```

```
## [1] 0.6172249
```



5 Exercises

This last exercise asks you to implement the AdaBoost algorithm as discussed in the above section and based on stumps. More specifically, you are asked to do the following:

Write an `R` function that takes as input any training sample (and the tuning parameter M). Based on this training sample, the output of the function should be the classifier trained through AdaBoost using as weak learners the class $\mathcal{C}_{stump}$. That is, classification trees of the form

$$C(\boldsymbol{x}) = \begin{cases} c & \text{if } x_j < s \\ -c & \text{if } x_j \geq s, \end{cases}$$

where c is either 1 or -1 (the classification label), s is a real number (the splitting criterion), and j is an index in $\{1, \ldots, p\}$.

This means that in the course of implementing AdaBoost, you also need to implement a function that returns for each of those weak classifiers the optimal constant c and the splitting value s in the weighted empirical risk minimization used in the AdaBoost algorithm; i.e., you need to write a function that solves (for given weights $w_1, \ldots, w_n$) the optimization problem

$$\min_{C \in \mathcal{C}_{stump}} n^{-1} \sum_{i=1}^{n} w_i L(Y_i, C(\boldsymbol{X}_i)).$$

Given this function, you can now proceed as in the above section on Boosting, where the AdaBoost algorithm is described. Illustrate your implementation on the titanic dataset (changing the class label 0 to -1) for different values of M, and plot the resulting testing error in dependence on M.

References

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer: New York, 2009.

Stefan Richter. *Statistisches und maschinelles Lernen*. Springer: Berlin, 2019.