Machine Learning Basics

This introduction will provide general information about machine learning (ML) theory as well as commonly used terms and concepts. We will focus on two ML algorithms: **Random Forest** (RF) and **Support Vector Machines** (SVM).

What is machine learning?

We will have a look at two supervised, non-parametric and nonlinear algorithms in the next two sections, the Random Forest (RF) and Support Vector Machine (SVM). These terms do not tell you anything? Then read the following short intro, which explains the most important terminology.

Machine Learning (ML) is actually a lot of things – it is a generic term for the artificial generation of knowledge or artificial intelligence. An artificial learning system learns from examples and can generalize those after the learning phase is completed. Examples are not simply memorized, but the underlying pattern is recognized. This becomes handy, when new, unknown data should be handled (learning transfer).
ML methods optimize their performances iteratively by learning from the training examples or train data. Those methods can be **descriptive** by distinguishing between several classes of a phenomenon (i.e., classification) or **predictive** by making predictions of a phenomenon (i.e., regression).
Ultimately, ML methods create a function, which tries to map input data x (e.g., Landsat 8 band reflectance) to a desired output response y (e.g., class labels, such as *urban*):

Find $f: x \rightarrow y = f(x)$.

In the field of remote sensing, descriptive ML algorithms are often used for land cover classifications. The range of applications for this is huge, ranging from land use / land cover (LULC) changes of all kind, e.g., urban sprawl, burned area detection, flood area forecast or land degradation. In order to perform a classification, the algorithm must learn to differentiate between the various types of patterns based on **training samples**. After the model has learned, it can then be tested for performance using independent **testing samples**. There are two main ways an algorithm can train - in an unsupervised or a supervised learning manner:

Unsupervised vs. Supervised Algorithms

The difference between unsupervised and supervised algorithms is based on whether they include a priori knowledge during the training phase by using labeled training samples or not.

Unsupervised algorithms have only the training samples *x* available, e.g., reflectance values of all 9 bands of Landsat 8. Thus, the class label information, e.g., *urban*, or *forest*, is missing. The objective of the algorithm is to describe how the data are organized or clustered – it has to find patterns and relationships by itself.

Supervised algorithms make use of a set of labeled training samples, including the samples *x* and the appropriate class labels *y*. The objective here is to predict the value *y* corresponding to a new, unknown, sample *x*. In other words, we teach the algorithm what the individual classes look like in the feature space, e.g., Landsat 8 bands.

Linear vs. Nonlinear Algorithms

This differentiation is straightforward:

Linear algorithms assume that the sample features x and the label output y are linearly related and there is an affine function $f(x) = (w, x) + b$ describing the underlying relationship.

Nonlinear algorithms assume a nonlinear relationship between *x* and *y*. Thus, $f(x)$ can be a function of arbitrary complexity.

Parametric vs. Non-Parametric Algorithms

Some ML methods require the data to follow a specific distribution in its feature space, for example the form of a multivariate normal Gaussian model. A method is called **parametric**, when those assumptions are made. The *Maximimum Likelihood Classification* is such a parametric algorithm.

Non-parametric approaches are not constrained to prior assumptions on the data distribution. Such methods allow application in versatile tasks and data types, such as e.g., RADAR data.

Overfitting vs. Underfitting

In statistics, the term *fit* refers to how well you approximate the function the ML algorithm uses to map input *x* to output *y*.

Overfitting appears, when a model learns to map the training data too well, which negatively impacts the performance of the model on new, unknown, data. Thus, the model lost its ability to generalize.

A trained ML model is **underfitting**, if it neither can model the train nor the test data correctly. Underfitting is easy to detect in the training phase by using given performance metrics. If the problem cannot be solved, another classifier should be considered.

Random Forest

The Random Forest (RF) algorithm is an all-rounder when it comes to data science problems!
It can be used for classification and regression tasks, while being non-linear and non-parametric, they can handle high dimensional, numerical and categorical variables and they have no special requirements or assumptions for the data to be used – so you do not have to worry about data distributing or standardizing your input features. And the best part: The basic concept is relatively easy to understand!

The RF is a representative of the so called “ensemble learning” methods. The word *ensemble* means nothing more than *group* in this context: A standard RF is based on generating a large number (*ntree*) of **decision trees** (DT) during the training phase (we will discuss those in a second), each constructed using a different subset (*n*) of your training dataset (*N*). Those subsets, also called *bootstrap* samples, are selected randomly and with replacement. After the training of the algorithm, each of those DTs is capable to map input data *X* (e.g., reflectance within Landsat 8 bands) to an output response *y* (e.g., class labels, like *urban*, *water* and *forest*). The Random Forest algorithm listens to all the responses of

the DTs and make its final prediction on the basis of the most frequently given response (**majority vote**). In case of regression, the RF takes the **average** of all the DT responses.
By growing multiple DTs, we can overcome or weaken the shortcomings of a single DT by combining them to form a more powerful model. Let us take a closer look at how a single decision tree works.

Decision Tree Method

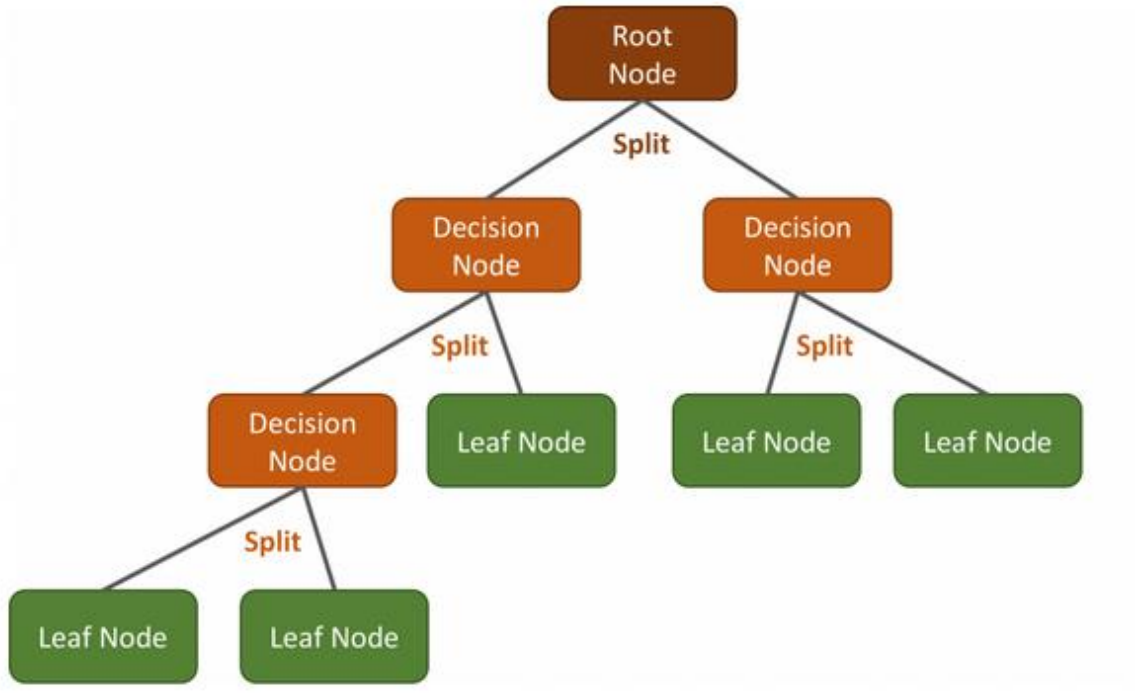
A Decision Tree, also known as *Classification and Regression Tree* (CART), is a supervised learning algorithm first introduced by BREIMAN et al. in 1984. We want to focus on the implementation used by the R package *randomForest*, which we will use late in an assignment.

A CART trains by splitting all available samples into homogeneous sub-groups of high purity based on a most significant feature. But one thing at a time…

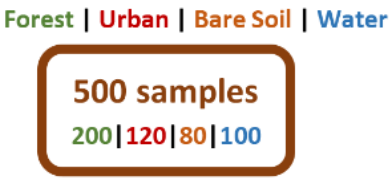
A decision tree conceptionally consists of three elements:

- **One Root Node**: That is the starting point, which includes all training samples. A first split is done here.
- **Multiple Decision Nodes**: Those are nodes where further splits are done.
- **Multiple Leaf (or Terminal) Nodes**: Those are nodes where the assignments to the classes (e.g., *urban*, *water*, etc.) happens for classifications tasks (or the mean response of all observation in this node is calculated for regressions).

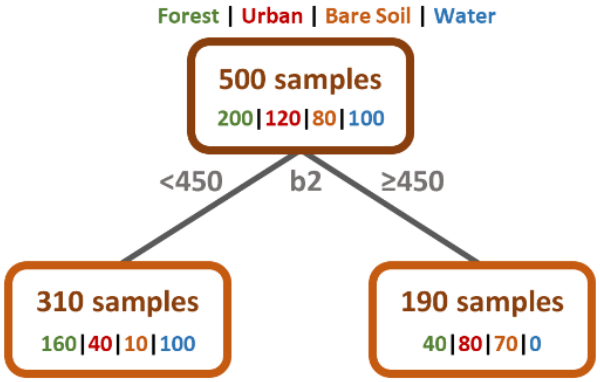
The structure and complexity of each DT can vary, depending on the training samples and the features used. For example, a very simple DT could look like this:



Ok, let us make that clearer with a fictitious example: We want to classify a Landsat 8 scene with 11 multispectral bands into four classes *Forest*, *Urban*, *Bare Soil* & *Water*. As a training dataset we use 500 pixels, 200 pixels belong to the forest class, 120 to urban, 80 to bare soil and to 100 water class, accordingly. We put **all 500 samples in the Root Node** in the top of the DT:



The next step is to split the node into two sub-nodes using a single threshold value. The CART algorithm always uses binary splits, i.e., it splits a node exactly in two sub-nodes, to increase the homogeneity of the resultant sub-nodes. In other words, it increases the purity of the node in respect to our target class. For example, the highest purity would be achieved if only one class was present in the first sub-node, and only one second class in the second sub-node. Of course this is not possible in our four class classification problem with only one split:



The first split could only clearly separate the *Water* samples, otherwise all classes are present at certain proportions in both subnodes. As you can see in the figure above, the DT decided to use the Landsat band 2 (b2) and the reflectance threshold value of 450 to do the split.

But wait a minute, how does the classifier know HOW and WHERE to do a split?!

The DT does not consider all features (Landsat bands) as candidates at each split, but only a certain number of features ($mtry$). By default, the parameter ($mtry$) corresponds to the root of the number of features M available, i.e., $mtry = \sqrt{M}$. Thus, in our example the DT samples 3 out of 11 Landsat bands randomly and with replacement, because $\sqrt{11} \approx 3$.

There are various measures for determining the most significant candidate (e.g., *Gini Index*, *Chi-Square*, *Variance Reduction* or *Information Gain*). Our CART algorithm uses the *Gini Index*, so let us have a look at what it does:
Gini Index says, if we select two samples from our node, then they must be of the same class. If a node contains only one class, i.e., a completely pure node, Gini value would be 0. Thus, the lower the Gini value, the purer the node – which is good, because we want to group the samples according to

their classes. The *Gini Index* is calculated by subtracting the sum of the squared probabilities p of each class $\{1,2,\ldots,C\}$ from one:

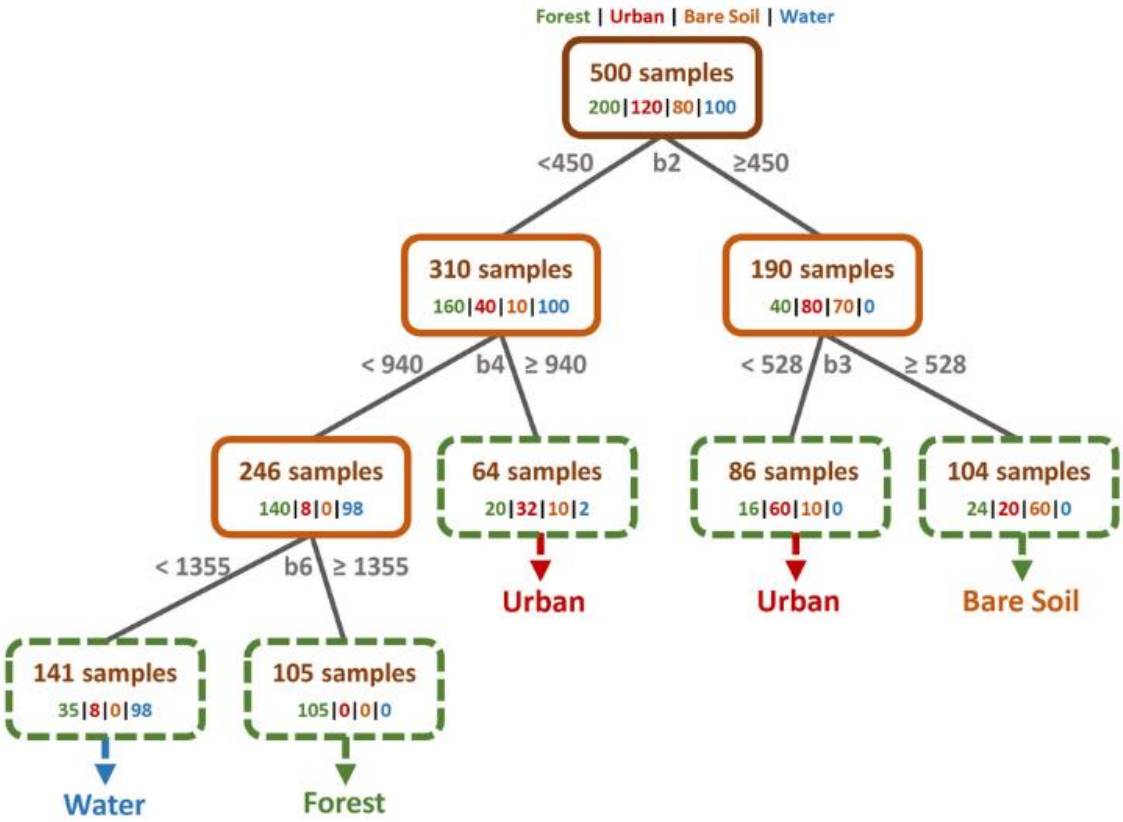
$$Gini = 1 - \sum_{i=1}^{C}(p^2)$$

For the first split in our example shown in the figure above, the math would be:

1. Calculate Gini for left sub-node: $(160/310)^2 + (40/310)^2 + (10/310)^2 + (100/310)^2 = 0.39$
2. Calculate Gini for right sub-node: $(40/190)^2 + (80/190)^2 + (70/190)^2 + (0/190)^2 = 0.36$
3. Calculation of weighted Gini for split on feature (band 2): $(310/500) * 0.39 + (190/500) * 0.36 = 0.38$

The Gini value is thus 0.38 if the Landsat Band 2 and the reflectance value 450 are chosen as as the feature and the threshold value, respectively. Finding a suitable threshold is an iterative process. So the algorithm tries many different values before picking out the one resulting in the smallest Gini, in this case the 450. As $mtry = 3$ in our example, the DT will test two more bands, e.g., band11 and band 1, the same way and compare the Gini values in the end.

That is all the magic behind splitting procedure! In the same way, the sub-nodes are split further, making the tree "grow":



All samples within a leaf node are assigned to the class most frequently represented. The figure above shows, that the leaf nodes are not perfectly pure, e.g., there are 35 forest and 8 urban samples assigned to the water-node on the far left. These will result in misclassifications in your final classification map.

Working with Random Forests

In the case of a Random Forest approach, the tree shown above would continue growing until almost all nodes contain only one class at a time. While that would most likely result in overfitting in a single DT, this effect is relativized by the majority vote in the end when using a RF. Anyway, there are some more advantages when using a RF:

Out of Bag (OOB) Error:
There is no need for a cross-validation or a separate test set, as the OOB Error is estimated internally in the training phase as an unbiased estimate of the classification error. Remind: each DT is created based on a small sample of your original training data (the so-called bootstrap sample). About one-third of the samples within this bootstrap sample are left out (out-of-bag) and not used for the construction of the DT. After the construction of the DT, the left out samples are used to test the tree. The Leaf Nodes then provide the class that is most represented by the OOB samples in each node. The proportion of samples that does not correspond to the class in the Leaf Node, averaged over all Leaf Nodes, is the OOB Error (overall accuracy for classifications and mean absolute error for regressions).

Variable Importance (VI)::
A Random Forest can identify the most significant/ important features for your classification or regression task. Therefore it can also be considered as a method for dimensionality reduction.
If you plot the variable importance with R, you will get two measures in the *randomforest*-package:
MeanDecreaseAccuracy: we permute all threshold values of a specific feature (e.g., Landsat band 2) and see how the purity in a Leaf Node changes. In detail: we count the number of correctly classified OOB samples once the tree is grown. After that, we randomly permute the values of a specific feature for the OOB samples and classify the OOB samples again. The difference of correctly classified samples between permuted and original OOB data, averaged over all trees, is the MeanDecreaseAccuracy.
MeanDecreaseGini: each time a split is done, both sub-nodes become more pure and the Gini values decrease. The sum of all Gini decreases for each individual feature (e.g., Landsat band) over all trees is the MeanDecreaseAccuracy.

Support Vector Machine

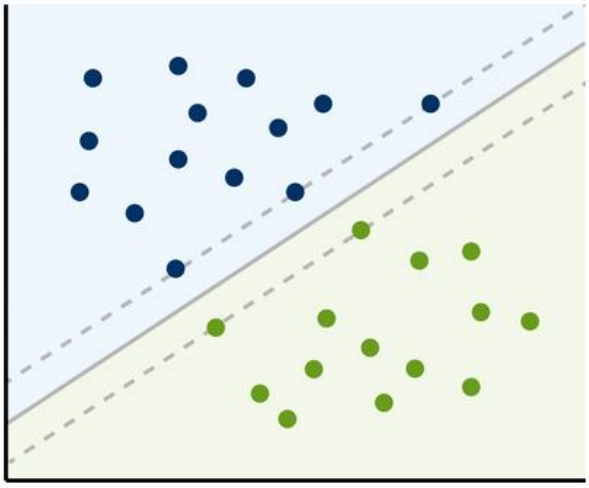
The concept of Support Vector Machines (SVM) was introduced in the context of pattern recognition and machine learning in 1979 by VAPNIK & CHERVONENKIS. A SVM is a powerful tool for high dimensional, linear and non-linear data science problems. It is always a good choice to consider a SVM for classification and regression tasks due to its high customizability and settings options. However, the search procedure for critical parameter is more complex compared to the RF.

Let us have a look at the basic concepts of a supervised SVM implementation for a classification task.

Basic Concept

Consider a set of labelled training samples X, e.g., pixels of an RS image. Each sample is a single vector in the feature space, e.g. one grey scale value for each of the 11 spectral bands of a Landsat 8 scene gives a vector with dimensions $1\ x\ 8$. The concept of a SVM is based on an optimal linear

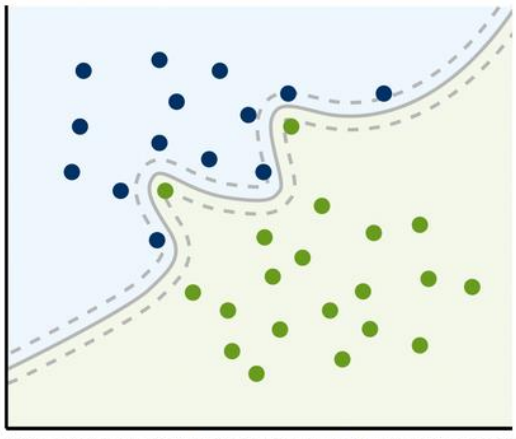
separating hyperplane that is fitted to the training samples of two classes within this multidimensional feature space. Thereby, the optimization problem that has to be solved is aiming at a maximization of the margins between this hyperplane and the closest training samples, the so-called and name giving **Support Vectors**. Only those Support Vectors are needed to describe the hyperplane mathematically exactly, vectors farther from the hyperplane remain hidden. The following figure shows a hyperplane separating two classes in two-dimensional space, e.g. Band 3 on the x-axis and Band 4 on the y-axis:



A linear hyperplane separating two classes (blue and green dots) in a two-dimensional feature space, e.g., Landsat bands 3 and 4

This wide and empty border area surrounding the hyperplane should ensure that new vectors, which are not in our training dataset, i.e., pixels in RS imagery, will be classified as reliably as possible in a later prediction process.

Ok, the classification problem in the figure above is very easy to solve linearly. However, the condition of linear separation is generally not met in real-life data. In the case of non-linear separable samples, the SVM needs a so-called kernel. A **kernel** is actually a function, which maps the data into a higher dimensional space where the data is separable. There are several of these kernels, such as linear, polynomial, radial basis and sigmoid kernels, with the radial basis function (RBF) kernel most commonly used for non-linear problems. In doing so, even the most nested training samples become linear separable and the hyperplane can be determined. After a transformation back into the initial lower-dimensional space, this linear hyperplane can become a non-linear or even un-connected hypersurface:



Non-linear hyperplane after inverse transformation from higher dimensionality separating two classes

In the context of remote sensing, binary classification problems are not common. There are several approaches to solve multi-class problems, the most frequently used approach is the one-against-one rule, which is also implemented in the *e1071*-package, which we use in the next assignment. According to this one-against-one approach, a binary classifier is trained for each pair-wise class combination, and the most frequently assigned class label is assigned in the end.

Common Parameters

Regularization parameter *C*: As the hyperplane can be of arbitrary dimensionality, it could be fitted perfectly to match the training dataset. However, this would result in extreme overfitting. The regularization parameter *C* allows the SVM to misclassify individual samples, but at the same time punish them. Large *C* values cause the hyperplane margin to shrink, which may help to increase the amount of correctly classified samples. Conversely, a small *C* value will result in a larger-margin hyperplane, leading to lower punishments and maybe more misclassified samples.

Kernel parameter gamma γ: This parameter defines how far the influence of a single training sample reaches, i.e. its radius of influence. Low values mean “far”, and high values mean “close”. The model performance is quite sensitive to this parameter: when gamma is too large, the radii of influence of the SVs are too big and only affect the SVs itself, making the regularization with *C* impossible and thus lead to overfitting. When gamma is too small, the model is too constrained and may not be capable to capture the complexity of our training dataset.

Parameter epsilon ε (only in regressions): The epsilon parameter is an additional value of tolerance where no penalty is given to errors. The larger epsilon is, the larger errors you allow in your model. When epsilon is zero, all errors will be penalized.

Questions / prove your knowledge:

- What is the overall goal of machine learning methods in remote sensing?
- What is the difference between supervised and unsupervised algorithms?
- Define the Out Of Bag (OOB) error!