HEILBRONN UNIVERSITY
OF APPLIED SCIENCES

On the Equivalence of tree-based Ensemble Learners with Decision Trees

Tobias Brock

supervised by
Prof. Dr. Till Florian Kauffeldt

co-supervised by
Prof. Dr. Carsten Lanquillon

A thesis presented for the degree of
Bachelor of Arts

Faculty of International Business
Heilbronn University of Applied Sciences
Germany
October 7, 2022

On the Equivalence of tree-based Ensemble Learners with Decision Trees

Abstract

The endeavor of this thesis is to ameliorate the interpretability of tree-based ensemble learners by proving the existence of prediction equivalent decision trees and the reduction of their order (number of vertices). The structure is twofold. First, a preliminary paper introduces the field of machine learning. Fundamental supervised learning algorithms are discussed, specifically decision trees and tree-based ensemble learners like random forests and gradient boosts. The second part of the thesis is a collaboration of several authors and contains the theorems and algorithms for the tree composition and reduction. As a result, ensemble learners such as gradient boosts can be transformed into a single, prediction equivalent decision tree that can be used to explain the decision-making process on a global and local level.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AdaBoost	Adaptive Boosting
AI	Artificial Intelligence
CART	Classification and Regression Trees
CCP	Cost Complexity Pruning
FN	False Negative
FP	False Positive
HPO	Hyperparameter Optimization
LASSO	Least Absolute Shrinkage and Selection Operator
MCE	Misclassification Error Rate
ML	Machine Learning
MSE	Mean Squared Error
OOB	Out-of-Bag
PAC	Probably Approximately Correct
RSS	Residual Sum of Squares
TN	True Negative
TNR	True Negative Rate
TP	True Positive
TPR	True Positive Rate
WQS	Weighted Quantile Sketch
XGBoost	eXtreme Gradient Boosting

1. Introduction

Artificial Intelligence (AI), specifically Machine Learning (ML) has seen widespread adoption in several fields such as banking and finance, medicine, healthcare and even in education or the solid-state materials sciences over the past couple of years (Khandani et al., 2010, Litjens et al., 2017, Schmidt et al., 2019, Zawacki-Richter et al., 2019). Although the relevance of AI is indisputable, there exists no precise definition of what constitutes machine intelligence. One way of defining AI is "the study of agents that receive percepts from the environment and perform actions" (Russell and Norvig, 2010). An agent, more precisely, an intelligent agent, is an entity that perceives its environment in an intelligent manner by acting rationally and autonomously in accordance with its surroundings. This implies that an intelligent agent has the ability to gather information from its environment to adapt an optimized behavioral strategy. This adaption can be called learning and is heavily dependent on the environment the agent is confronted with. One of the most important subfields of AI is ML. A simple way of defining ML is "the automated detection of meaningful patterns in data (Shalev-Shwartz and Ben-David, 2014). In the specific case of ML, the agent is a computer program or algorithm that learns and adapts for a given task. These algorithms have the ability to extract patterns from large quantities of data, often referred to as "Big Data", that are oblivious to the human mind and therefore pertinent in a highly digitalized world.

An immediate consequence is that ML algorithms require more computational power to deal with the increased complexity of Big Data. Therefore, resulting in complicated and for humans difficult to understand models, often described as "black box models". This, as a consequence led to the uprising of explainable AI, also by some authors referred to as interpretable ML. One may notice that neither AI nor interpretability are rigorously defined, since there exists no precise mathematical formulation (Adadi and Berrada, 2018). Miller (2017) for example, defined interpretability as "the degree to which a human can understand the cause of a decision", which is a reasonable definition in the context of this thesis. However, understanding may very well be subjective and is therefore highly dependent on the individual. Linardatos et al. (2020) described interpretability methods for the creation of white-box models. Contrary to black-box models, white-box models are supposed to be intrinsic and transparent and therefore easy to understand.

Some examples of white-box models are linear models, but also decision trees and rule-based models. In particular, decision trees can provide access to the decision-making process on a local and global level.
This work is separated into two parts. The first part is a preliminary paper, where an introduction to the term supervised learning and its different aspects is given. Classification and Regression Trees (CART) or analogously, decision trees are discussed in more detail. Last, tree-based ensemble learners are to be explained, specifically random forests and gradient boosting. The main paper proposes a new model-class-specific method that transforms tree-based ensemble learners into prediction equivalent decision trees. Theorems for the composition and reduction of trees are provided. Moreover, general algorithms for both theorems are suggested, as an iterative composition-reduction algorithm that combines both procedures.

2. Supervised Learning

This section explains the preliminary concepts of supervised learning as described by Shalev-Shwartz and Ben-David (2014), Hastie et al. (2009) and Murphy (2013). Risk Minimization and different aspects of training and evaluation techniques are introduced. Decision trees and ensemble models are discussed in greater detail in section 3, 4 and 5.

For a supervised machine learning setup, consider a domain set $\mathcal{X} = \bigtimes_{j=1}^{d} X_j$, where X is a random variable. $\mathcal{X}$ is the space of all feature outcomes, also referred to as feature space. A vector $\mathbf{x} \in \mathcal{X}$ is called instance or example. For every $\mathbf{x} \in \mathcal{X}$ there exists a label $y \in \mathcal{Y}$, where $\mathcal{Y}$ denotes the label space. For a supervised learning problem, there is a finite set of labeled instances in the form $(\mathbf{x}_i, y_i)$. A dataset S with n instances can be denoted as

$$S = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\} \subseteq (\mathcal{X} \times \mathcal{Y})^n$$

S is called the training data that is given to a learner as input. Usually, the feature space $\mathcal{X}$ is a subset of $\mathbb{R}^d$. Notice there exist multiple label spaces $\mathcal{Y}$. For example:

- A binary label space $\mathcal{Y} = \{0, 1\}$ or $\mathcal{Y} = \{-1, 1\}$
- A multi class label $\mathcal{Y} = \{1, ..., g\},\ g \geq 3$
- A continuous label space $\mathcal{Y} \subseteq \mathbb{R}$

In the cases (i) and (ii) the learner has to solve a classification problem, (iii) is a regression problem. A learner h, also called hypothesis, predictor or classifier, learns a function $h : \mathcal{X} \rightarrow \mathcal{Y}$ called prediction rule that can label unseen and unlabeled instances. The training data is generated by some underlying probability distribution $\mathcal{D}$. Assume there exists a correct labeling function $f : \mathcal{X} \rightarrow \mathcal{Y}$ such that $y_i = f(x_i)$ for all i. The instances of S are drawn at random from $\mathcal{D}$ and labeled by f. The goal of the learner h is to learn a function so that $f(x_i) = h(x_i)$ for all i.

In a binary classification problem, the error of a hypothesis refers to the probability that if a random instance x is drawn from the underlying probability distribution $\mathcal{D}$ one gets that $f(x)$ does not equal $h(x)$. The error of a prediction rule $h : \mathcal{X} \rightarrow \mathcal{Y}$ is defined as

$$\mathcal{L}_{\mathcal{D},f}(h) = \underset{x \sim \mathcal{D}}{\mathbb{P}}[h(x) \neq f(x)] = \mathcal{D}(\{x : h(x) \neq f(x)\}).$$

The error of a prediction rule h is the probability assigned by the probability measure $\mathbb{P}$ of randomly having an example x such that $f(x) \neq h(x)$, where the error is measured with respect to the underlying distribution $\mathcal{D}$. $\mathcal{L}_{\mathcal{D},f}(h)$ is often called generalization error, risk or true error of h. Notice that the learner h is not aware of the probability distribution $\mathcal{D}$ but aims to learn a probability distribution that minimizes the loss of h. A suitable measure for the risk of h is the expected value

$$\mathcal{R}(h) = \mathbb{E}[\mathcal{L}(y, h(x))] = \int \mathcal{L}(y, h(x)) d\mathcal{D}$$

2.1. Empirical Risk Minimization

Let S be a training set that is sampled at random from the underlying probability distribution $\mathcal{D}$. A learner h is trained on S to learn the prediction rule $h_S : \mathcal{X} \rightarrow \mathcal{Y}$. Notice that h_S is dependent on the specific training examples. Furthermore, h is oblivious to the underlying true probability distribution $\mathcal{D}$ but has to minimize the risk with respect to f and $\mathcal{D}$. This minimization is achieved by learning the hypothesis on the training set S. Given an arbitrary loss function $\mathcal{L}_S(h)$, the minimization problem of finding an ideal h on S is called *Empirical Risk Minimization.* Formally,

$$\mathcal{R}^*(h) = \underset{h \in \mathcal{H}}{\arg\min} \sum_{i=1}^{n} \mathcal{L}_S(y_i, h(\mathbf{x}_i))$$

where arg min is the set of hypotheses that minimize $\mathcal{L}_S(h)$ over $\mathcal{H}$ for an algorithm $\mathcal{A}$. By choosing a specific algorithm, all possible predictors h get restricted by $\mathcal{H}$. This restriction of the hypotheses is often referred to as inductive bias or structural prior because the restrictions are dependent on the particular hypotheses class. In linear regression, for example, it is assumed that the underlying relationship is linear. Usually, a hypotheses class is defined by a set of parameters $\mathbf{\Theta} = \{\boldsymbol{\theta} : \boldsymbol{\theta} \in \mathbf{\Theta}\}$. Therefore, the empirical risk can be expressed as

$$\mathcal{R}^*(\boldsymbol{\theta}) = \sum_{i=1}^{n} \mathcal{L}_S(y_i, h(\mathbf{x}_i|\boldsymbol{\theta}))$$

and the corresponding empirical risk minimization is

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}\in\mathbf{\Theta}}{\arg\min}\ \mathcal{R}^*(\boldsymbol{\theta})$$

Consequently, empirical risk minimization finds an $h \in \mathcal{H}$ that is parameterized by $\boldsymbol{\theta} \in \mathbf{\Theta}$ such that $h_{\boldsymbol{\theta}}$ minimizes $\mathcal{R}^*(\boldsymbol{\theta})$.

2.2. Loss Functions for Regression and Classification

A *Loss Function* $\mathcal{L}(y, h(\mathbf{x}))$ evaluates a hypothesis $h \in \mathcal{H}$ on the training set S by showing how well the algorithm has learned. Thus, it quantifies the quality of the prediction $h(\mathbf{x})$ of a single observation $\mathbf{x}$. A general formulation may be

$$\mathcal{L} : \mathcal{Y} \times \mathbb{R} \to \mathbb{R}.$$

In the following, some of the most common functions are illustrated.

2.2.1. Regression

The l_2 or *Square Loss* is one of the most common loss functions that is used for regression problems.

$$\mathcal{L}_2(y, h(\mathbf{x})) = (h(\mathbf{x}) - y)^2.$$

One main advantages is that the square loss is convex and differentiable. However, the square loss overvalues large deviations and undervalues deviations that are smaller than 1. Correspondingly, outliers may bias the estimation.

Alternatively, the l_1, also called *Absolute Loss*, is used

$$\mathcal{L}_1(y, h(\mathbf{x})) = |h(\mathbf{x}) - y|.$$

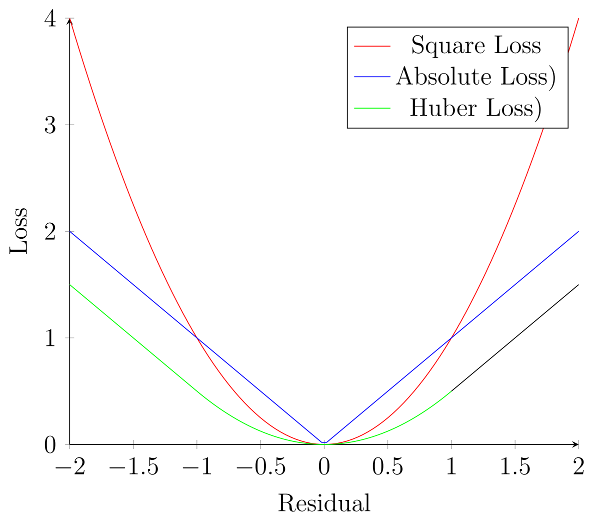
The absolute loss is convex but not differentiable at zero. Therefore, numerical optimization has to be used to find a local minimum.

The *Huber Loss* is a combination of square loss and absolute loss with some decision rule δ, which squares the deviations for small losses and takes the absolute value for larger deviations.

$$\mathcal{L}_H(y, h(\mathbf{x})) = \begin{cases} \frac{1}{2}(h(\mathbf{x}) - y)^2, & \text{if } |h(\mathbf{x}) - y| < \delta \\ \left(\delta|h(\mathbf{x}) - y| - \frac{\delta}{2}\right), & \text{else} \end{cases}$$

Notice that the Huber loss is convex and once differentiable because the square loss is used in the neighborhood of 0. It is also more robust towards outliers. However, the determination of δ has no closed form solution. Hence, numerical optimization is required for the model fit.

Figure 1: Regression Losses



2.2.2. Classification

A g class classification model can be expressed as $h : \mathcal{X} \rightarrow \mathbb{R}^g$. Although the task is the classification of some discrete label, the output of the hypothesis is continuous and yields a prediction for all g classes. Such a prediction can either be some real number (score) or a probability. The output being continuous has the simple reason that the optimization is substantially more convenient. Moreover, different losses can be used for the training stage and the performance evaluation.

Given g scoring functions $s_1, ..., s_g : \mathcal{X} \to \mathbb{R}$. The class prediction is determined by finding the maximum score of $s_1(\mathbf{x}), ..., s_g(\mathbf{x})$.

$$h(\mathbf{x}) = \underset{k \in \{1,...,g\}}{\arg\max}\ s_k(\mathbf{x}).$$

In the case of a binary classification problem with $g = 2$, a single scoring function s suffices.

For probabilistic classifiers and a g class classification problem, define $p_1, ..., p_g : \mathcal{X} \to [0, 1]$, with $\sum_k p_k = 1$. Analogous to scoring functions, labels are predicted by the class with the highest probability

$$h(\mathbf{x}) = \underset{k \in \{1,...,g\}}{\arg\max}\ p_k(\mathbf{x}).$$

In a binary classification problem, the corresponding class label can also be determined by thresholding, where $h(\mathbf{x}) = [p(\mathbf{x}) > c]$. Usually $c = 0.5$ for probabilistic classifiers and 0 for scoring functions. In higher dimensions, decision regions for all g classes are determined such that

$$\mathcal{X}_k = \{\mathbf{x} \in \mathcal{X} : h(\mathbf{x}) = k\}.$$

The points in $\mathcal{X}$ where the maximal scores are tied are called decision boundaries.

Now given a discrete classifier, the *Zero-One Loss* can be defined in a straightforward manner

$$\mathcal{L}_{0/1}(y, h(\mathbf{x})) = \delta_{h(\mathbf{x}) \neq y} = \begin{cases} 1, & \text{if } h(\mathbf{x}) \neq y \\ 0, & \text{else} \end{cases} .$$

The zero-one loss is neither differentiable nor continuous which renders optimization to be difficult. Therefore, it can be more lucrative to learn a hypothesis with a continuous loss function and use the zero-one loss for evaluation.

A well-known loss function is the *Exponential Loss* given by

$$\mathcal{L}(y, h(\mathbf{x})) = \begin{cases} \exp(-yh(\mathbf{x})) & \text{for } y \in \{-1, +1\} \\ \exp(-(2y-1)h(\mathbf{x})) & \text{for } y \in \{0, 1\} \end{cases} .$$

It is sensitive to strongly misclassified instances due to the exponential function. A more robust loss function for training is the *Binomial Loss* which is derived by the negative log-likelihood of the logistic function. In the binary case, a single scoring function s yields

$$\mathcal{L}_B(y, h(\mathbf{x})) = \begin{cases} \ln(1 + \exp(-2ys(\mathbf{x}))) & \text{for } y \in \{-1, 1\} \\ -ys(\mathbf{x}) + \ln(1 + \exp(s(\mathbf{x})) & \text{for } y \in \{0, 1\} \end{cases} .$$

The *Bernoulli Loss* is convex and differentiable. If the classification decision of $h(\mathbf{x})$ is determined by a probability $p(\mathbf{x})$, then the so called cross-entropy loss can be derived by the Bernoulli loss as

$$\mathcal{L}_{ce}(y, h(\mathbf{x})) = -y \ln(p(\mathbf{x})) - (1-y)\ln(1 - p(\mathbf{x})).$$

The *Cross-Entropy Loss* can be generalized to g classes by

$$\mathcal{L}_{ce}(y, h(\mathbf{x})) = -\sum_{k=1}^{g} \mathbb{1}_{\{y=k\}} \ln(p_k(\mathbf{x})),$$

where $\mathbb{1}$ denotes the indicator function. Another important loss function for probabilistic classifiers is the *Brier Score*

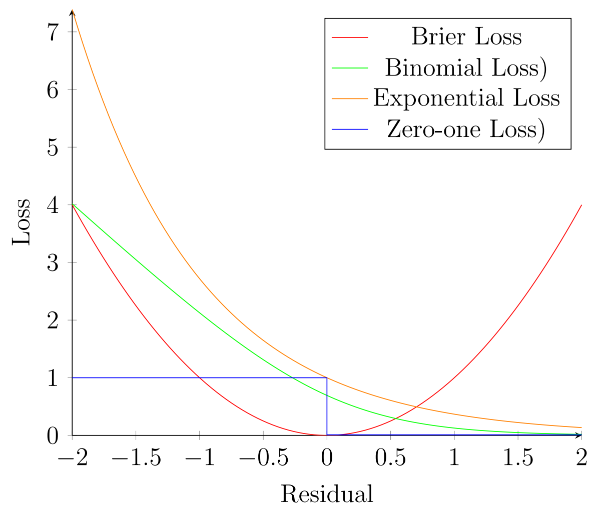
$$\mathcal{L}_B(y, h(\mathbf{x})) = (p(\mathbf{x}) - y)^2.$$

It is easy to see that the brier score is the equivalent of the square loss for classification and therefore holds analogous properties. However, in comparison to the entropy-loss, substantially wrong predictions are punished less severely. Given $p_1(\mathbf{x}), ..., p_g(\mathbf{x})$ class probabilities. The multiclass Brier score is defined as

$$\mathcal{L}(y, h(\mathbf{x})) = \sum_{k=1}^{g} (\mathbb{1}_{\{y=k\}} - p_k(\mathbf{x}))^2.$$

Figure 2 provides a visualization for classification loss functions.

Figure 2: Classification Losses



2.3. Basic Optimization

The goal of optimization is to determine the ideal parameter vector $\boldsymbol{\theta}$ that minimizes the risk for some h in an arbitrary hypotheses space $\mathcal{H}$. Function optimization in high dimensional spaces requires the existence of all partials at a specific point. Because the risk function $\mathcal{R}$ is dependent on the parameter space $\boldsymbol{\Theta}$, the necessary criterion for a minimum is

$$\nabla_{\boldsymbol{\theta}^* \in \boldsymbol{\Theta}} \mathcal{R}(\boldsymbol{\theta}^*) = \frac{\partial}{\partial \boldsymbol{\theta}^*} \mathcal{R}(\boldsymbol{\theta}^*) = 0$$

where $\nabla \mathcal{R}(\boldsymbol{\theta}^*)$ is the gradient at point $\boldsymbol{\theta}^*$. If a minimum exists, then $\nabla \mathcal{R}(\boldsymbol{\theta}^*) = 0$ is called a stationary or critical point of $\mathcal{R}$. Formally, there exists $\epsilon > 0$ such that for each $\boldsymbol{\theta} \in B_\epsilon(\boldsymbol{\theta}^*)$ it holds that $\mathcal{R}(\boldsymbol{\theta}^*) \leq \mathcal{R}(\boldsymbol{\theta})$, where $B_\epsilon(\boldsymbol{\theta}^*)$ is the epsilon ball. This condition is not sufficient for the determination of a minimum. Therefore, assuming $\mathcal{R}$ is twice differentiable, the hessian matrix $\mathfrak{H}_\mathcal{R}$ can be evaluated. If $\mathfrak{H}_\mathcal{R}(\boldsymbol{\theta}^*) = \frac{\partial^2}{\partial^2 \boldsymbol{\theta}^*} \mathcal{R}(\boldsymbol{\theta}^*)$ is positive definite, then $\mathcal{R}$ has a local minimum at $\boldsymbol{\theta}^*$. For a global minimum at a point $\boldsymbol{\theta}^*$, it holds that for all $\boldsymbol{\theta} \in \boldsymbol{\Theta} : \mathcal{R}(\boldsymbol{\theta}^*) \leq \mathcal{R}(\boldsymbol{\theta})$. If a minimum exists and the function is convex, then the minimum is in fact a global minimum. Given that the risk function is twice differentiable on its entire domain, $\mathcal{R}$ is convex if $\mathfrak{H}_\mathcal{R}(\boldsymbol{\theta})$ is positive definite for all $\boldsymbol{\theta} \in \boldsymbol{\Theta}$. However, most machine learning algorithms are not convex optimization problems that can be solved in closed form. Therefore, optimization usually aims to find a suitable local minimum. An important method for numerical optimization is introduced in the following section.

Gradient Descent Optimization

The aim of *Gradient Descent Optimization* is the minimization of some differentiable function f. For every step $s \geq 0$ at a point $x_s \in \mathbb{R}^d$, gradient descent moves in the direction $\Delta x = -\alpha \nabla f(x_s)$, such that $f(x_s + \Delta x) < f(x_s)$. By choosing a small step size α, also called learning rate, gradient descent moves into the negative direction of the gradient. Notice that gradient descent uses the first order Taylor approximation. If f is differentiable and Δx sufficiently small, then

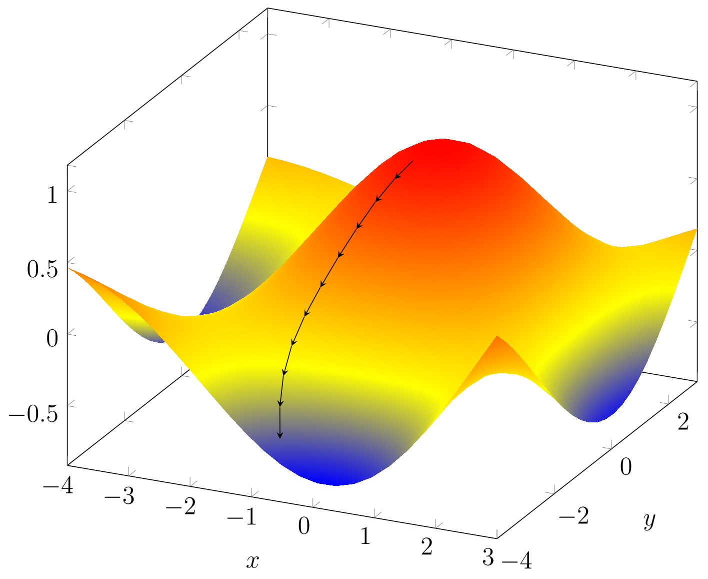
$$f(x_s + \Delta x) \approx f(x_s) + \Delta x^T \nabla f(x).$$

It is obvious that $\Delta x = -\alpha \nabla f(x_s)$ minimizes f stepwise. If $\Delta x = -\alpha \nabla f(x_s)$: $x_{s+1} = x_s - \alpha \nabla f(x_s)$ then

$$f(x_{s+1}) = f(x_s) - \alpha (\nabla f(x_s)^T) \nabla f(x_s) < f(x_s)$$

because $(\nabla f(x_s))^T \nabla f(x_s) > 0$. The procedure is visualized in figure 3.

Figure 3: Gradient Descent Minimization



The first order Taylor approximation only holds for small values of Δx. The minimization is achieved by having a sufficiently small $\alpha > 0$, otherwise the algorithm is not able to converge to a local minimum of f. In machine learning, gradient descent is used to minimize the risk of some hypothesis with respect to a vector of parameters. It can approximate a local minimum by using the first order Taylor approximation, assuming $\mathcal{R}(\boldsymbol{\theta})$ is linear at value $\boldsymbol{\theta}$. Therefore

$$\mathcal{R}(\boldsymbol{\theta}_{s+1}) = \mathcal{R}(\boldsymbol{\theta}_s + (-\alpha(\nabla\mathcal{R}(\boldsymbol{\theta}_s))) = \mathcal{R}(\boldsymbol{\theta}_s) - \alpha(\nabla\mathcal{R}(\boldsymbol{\theta}_s))^\top\nabla\mathcal{R}(\boldsymbol{\theta}_s) < \mathcal{R}(\boldsymbol{\theta}_s).$$

2.4. Regularization

Regularization penalizes model complexity and can therefore be a countermeasure to overfitting, where the model simply "memorizes" the training data. Overfitting occurs when data is particularly noisy, the quantity or quality of data is insufficient or the chosen model has an overly complex hypotheses space. Regularization aims to reduce the generalization error (for a more elaborate explanation, see section 2.7), namely the error the model makes when predicting unseen examples but not its training error. Regularization in its general form can be formulated as

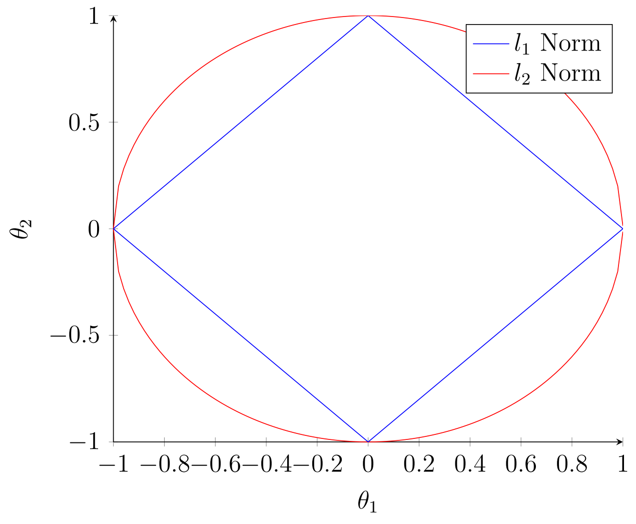
$$\mathcal{R}_r^*(\boldsymbol{\theta}) = \mathcal{R}^*(\boldsymbol{\theta}) + \lambda\mathcal{I}(\boldsymbol{\theta}).$$

λ is a non-negative control parameter and $\mathcal{I}$ is the complexity penalty or regularizer. Notice that this is a constraint optimization problem that tries to minimize $\mathcal{R}$ while simultaneously restricting the model complexity with $\mathcal{I}$. Hence, the risk is determined by the sum of the model risk with the regularization term and the ideal λ can be determined by cross-validation (see section 2.7.3). Quite often, an l_p norm is used for $\mathcal{I}$.

$$\mathcal{R}_{l_p}^*(\boldsymbol{\theta}) = \mathcal{R}^*(\boldsymbol{\theta}) + \lambda||\boldsymbol{\theta}||_p$$

The l_p norm is convex for all $p \geq 1$. Most commonly, the l_1 and l_2 norm are used as regularization penalty (see section 2.5 and 5.3). The l_1 constraint forms a diamond in two-dimensional space. Hence, features with low significance tend to be excluded from the model, whereas the l_2 constraint can be visualized by a circle. Features that contribute less to the optimization receive lower weights during training. Figure 4 visualizes the l_1 and l_2 constraint.

Figure 4: Regularization Constraints



2.5. Linear Regression

Linear Regression is one of the simplest supervised learning algorithms, where a "straight line" is fitted to the examples in two dimensions and a plane/hyper plane in higher dimensions. Given d independent variables and a training set $S = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$, then a linear model for the data can be expressed as

$$\underset{n\times 1}{\mathbf{y}} = \underset{n\times d}{\mathbf{X}}\,\underset{d\times 1}{\boldsymbol{\theta}} + \underset{n\times 1}{\epsilon}$$

The aim of linear regression is to minimize the errors of the model. Applying the l_2 loss $\mathcal{L}(y, h(\mathbf{x})) = (y - h(\mathbf{x}))^2$ is equivalent to minimizing the Residual Sum of Squares (RSS): $RSS = \epsilon^\top \epsilon = \epsilon^2 + ... + \epsilon^2$, also called sum of squared errors. Therefore, the weight vector $\boldsymbol{\theta}$ has to be estimated. Solving the convex optimization problem by minimizing the empirical risk

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \mathcal{R}(\boldsymbol{\theta}) = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{n} \left(\boldsymbol{\theta}^\top \mathbf{x}_i - y_i\right)^2$$
$$= \arg\min_{\boldsymbol{\theta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2$$
$$\frac{\partial}{\partial \boldsymbol{\theta}} \mathcal{R}(\boldsymbol{\theta}) = 0$$

results in the closed form solution $\hat{\boldsymbol{\theta}} = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y}$ and therefore in a global minimum of $\mathcal{R}$. Notice that the l_2 loss is sensitive to outliers due to the quadratic form. Another common loss function used for linear regression is the l_1 loss $\mathcal{L}(y, h(\mathbf{x})) = |y - h(\mathbf{x})|$. The optimization of the l_1 loss is more difficult since the absolute value function is not differentiable at zero. Therefore, numerical optimization methods have to be used, like gradient descent. One major advantage of the l_1 loss is that outliers do not affect the model as drastically as in the l_2 case.

However, many algorithms do not have unique solutions but provide higher accuracy when predicting unseen examples. Linear regression also has a strong inductive bias because it assumes that the underlying relationship is linear. Some variations of linear regression are *Ridge Regression* and *Least Absolute Shrinkage and Selection Operator (LASSO) Regression.*

Ridge Regression bounds the variance of $\boldsymbol{\theta}$ by including a regularization term. The constraint is of the nature $\sum_{j=1}^{d} \theta_j \leq t$. This results in the following optimization problem

$$\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} \left(\boldsymbol{\theta}^\top \mathbf{x}_i - y_i\right)^2 + \lambda ||\boldsymbol{\theta}||_2^2$$

with the unique solution $\hat{\boldsymbol{\theta}} = \left(\mathbf{X}^\top \mathbf{X} + \lambda \mathbb{I}\right)^{-1} \mathbf{X}^\top \mathbf{y}$. As linear regression, ridge regression is a convex optimization problem and hence offers a unique solution. Weights that are not important for the prediction tend asymptotically towards zero.

For *LASSO* the l_1 norm is used as regularization constraint

$$\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} \left(\boldsymbol{\theta}^\top \mathbf{x}_i - y_i\right)^2 + \lambda ||\boldsymbol{\theta}||_1$$

Contrary to regular linear regression and ridge regression, Lasso is not a convex optimization problem and therefore has to be learned by numerical optimization. The Lasso constraint also leads to less important weights becoming zero due to properties of the l_1 norm.

2.6. Logistic Regression

Logistic Regression is a discriminant predictor and a generalized linear model that fits the posterior probabilities $p(y|\mathbf{x}, \boldsymbol{\theta})$ of the labels. The scores are given by $s(y|\mathbf{x}, \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \mathbf{x}$, whereas the transformation into probabilities is achieved by applying the sigmoid function z. For $\mathcal{Y} = \{0, 1\}$, define

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \frac{e^{\boldsymbol{\theta}^\top \mathbf{x}}}{1 + e^{\boldsymbol{\theta}^\top \mathbf{x}}} = \frac{1}{1 + e^{-\left(\boldsymbol{\theta}^\top \mathbf{x}\right)}} = z\left(\boldsymbol{\theta}^\top \mathbf{x}\right).$$

Observe that $s(y|\mathbf{x}, \boldsymbol{\theta}) = \ln\left(\frac{p}{1-p}\right) = \boldsymbol{\theta}^\top \mathbf{x}$ are the log of the odds predictions. The cross-entropy loss is used as loss function

$$\mathcal{L}_{ce}(y_i, h(\mathbf{x}_i)) = -y \ln(p(y|\mathbf{x}, \boldsymbol{\theta})) - (1 - y) \ln(1 - p(y|\mathbf{x}, \boldsymbol{\theta})).$$

The optimization is achieved by estimating the parameter vector $\boldsymbol{\theta} \subseteq \mathbb{R}^d$ with maximum likelihood estimation by assuming that the conditional probabilities $p(y_i|\mathbf{x}_i, \boldsymbol{\theta})$ are independent for all i. Given a training set S of size n, the likelihood function can be formulated as

$$p(\mathbf{y}|S, \boldsymbol{\theta}) = \prod_{i=1}^{n} p(y_i|\mathbf{x}_i, \boldsymbol{\theta})$$

and the empirical risk optimization via the negative log likelihood is

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \boldsymbol{\Theta}}{\arg\min} \sum_{i=1}^{n} \ln\left(1 + e^{-\left(\boldsymbol{\theta}^\top \mathbf{x}_i\right)}\right).$$

Logistic regression does not provide a closed form solution. The best fit is therefore determined by numerical optimization techniques like gradient descent or Newton's method. The generalization of logistic regression is called softmax regression and considers a total of g linear discriminant functions such that $s_k(\mathbf{x}) = \boldsymbol{\theta}_k^\top \mathbf{x}$ for $k \in \{1, ..., g\}$.

The application of the softmax function $\sigma : \mathbb{R}^g \to \mathbb{R}^g$ yields the corresponding probability estimates

$$p(y = k|\mathbf{x}, \boldsymbol{\theta}) = \sigma(s_k(\mathbf{x}))_k = \frac{\exp\left(\boldsymbol{\theta}_k^\top \mathbf{x}\right)}{\sum_{k=1}^{g} \exp\left(\boldsymbol{\theta}_k^\top \mathbf{x}\right)},$$

where $\boldsymbol{\theta} \subseteq \mathbb{R}^{d \times g}$. For training the cross-entropy loss is used

$$\mathcal{L}_{ce}(y_i, h(\mathbf{x}_i)) = -\sum_{k=1}^{g} \mathbb{1}_{\{y_i = k\}} \ln p(y_i = k|\mathbf{x}, \boldsymbol{\theta}).$$

Similarly to the binary case, optimization is achieved by minimizing the negative likelihood

$$\mathcal{R}(\boldsymbol{\theta}) = \arg\min_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} -\prod_{i=1}^{n}\prod_{k=1}^{g} p(y_i = k|\mathbf{x}_i, \boldsymbol{\theta})\mathbb{1}_{\{y_i=k\}}$$

where taking the log results in

$$\mathcal{R}^*(\hat{\boldsymbol{\theta}}) = \arg\min_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} -\sum_{i=1}^{n}\left(\sum_{k=1}^{g}\mathbb{1}_{\{y_i=k\}}\ln p(y_i = k|\mathbf{x}_i, \boldsymbol{\theta})\right)$$

2.7. Model Evaluation

In ML, it is generally of interest how well a model performs on unseen examples. In other words, one may describe this as the ability to generalize. Therefore, model evaluation is a crucial part of machine learning, where the model is usually evaluated after the training process with unseen data. The *Generalization Error* is a theoretical measure that describes the true model performance in statistical terms. This section closely follows Bischl et al. (2021). A more elaborate explanation can be found in their article.

2.7.1. Generalization Error

A given model is learned on a training set S until the training loss is minimized. However, the training loss cannot be used as an estimate for the true performance. Evaluating the model on the training set results in a bias estimate because the examples have already been used in the training process. If a model is evaluated on S, it tends to memorize the examples. Algorithms like boosting (see section 5) can be learned on the training set S until the empirical risk is arbitrarily small. Consequently, resulting in overfitting because the algorithm tends to learn the intrinsic characteristics (noise) of the data and therefore leads to poor generalization. Hence, new examples are required to provide an unbiased estimate of the generalization error. Let $\mathcal{L}$ be an arbitrary loss function and h a trained hypothesis. The true generalization error for an unseen example $(y, \mathbf{x}) \sim \mathcal{D}$ is given by

$$\mathcal{E}(h, \mathcal{L}) = \mathbb{E}[\mathcal{L}(y, h(\mathbf{x}))]$$

A suitable estimate of the generalization error for a test set $\mathcal{T}$ is

$$\hat{\mathcal{E}}(h, \mathcal{L}) := \frac{1}{|\mathcal{T}|}\sum_{(\mathbf{x},y)\in\mathcal{T}}[\mathcal{L}(y, h(\mathbf{x}))]$$

A performance metric ϕ outputs a real number that can be used to evaluate the model performance on the test set $\mathcal{T}$. Often, the loss function applied during training can be used for ϕ, although this does not necessarily have to be the case.

2.7.2. Training and Test Error

In real life scenarios, one does not usually have access to a separate training and test set. Therefore, the set S is commonly divided into training set S_{tr} and test set S_{ts}. This strategy is called *Holdout-Splitting*. For estimation purposes, a $2/3, 1/3$ split is often used in practice. Therefore, h is trained on S_{tr} and subsequently evaluated on S_{ts}. The average training loss is given by

$$\mathcal{R}^*(h) = \arg\min_{h \in \mathcal{H}} \frac{1}{|S_{tr}|} \sum_{(\mathbf{x},y) \in S_{tr}} \mathcal{L}(y, h(\mathbf{x})).$$

The model's ability to generalize can then be evaluated with the average test loss

$$\hat{\mathcal{E}} = \frac{1}{|S_{ts}|} \sum_{(\mathbf{x},y) \in S_{ts}} \mathcal{L}(y, h(\mathbf{x})).$$

The training loss is also called inner loss, whereas the test loss can be referred to as outer loss. Given independent and identically distributed examples from $\mathcal{D}$, then $\mathcal{L}(y, h(\mathbf{x}))$ is a random variable. Hence, the expectation can be determined. It is easy to see that the test error is an unbiased estimate for the true generalization error by simply taking the expectation

$$\mathbb{E}\left[\frac{1}{|S_{ts}|} \sum_{(\mathbf{x},y) \in S_{ts}} \mathcal{L}(y, h(\mathbf{x}))\right] = \mathbb{E}[\mathcal{L}(y, h(\mathbf{x}))] = \mathcal{R}(h(\mathbf{x}))$$

Hold-out splitting has to compromise between bias and variance, which are both affected by the training and test split of S. A smaller training set leads to higher (pessimistic) bias because fewer examples are used to learn the model, decreasing its ability to generalize appropriately. On the upside, large training sets tend to decrease the test error in general. Note that using a large training set increases the variance of the test error.

2.7.3. Resampling

As discussed in the previous section, the drawbacks of holdout splitting are the high pessimistic bias when using a relatively large test set, while using a small test set leads to high variance of the generalization error. Again, following Bischl et al. (2012), resampling strategies can be used to balance the trade-off between pessimistic bias and variance. Given $h \in \mathcal{H}$, a set of examples S of size n and a loss-based performance metric $\phi_{\mathcal{L}}$. The goal is the estimation of the true generalization error $\mathcal{E}(h, S, \phi_{\mathcal{L}}) = \mathbb{E}[\mathcal{L}(y, h_S(\mathbf{x}))]$. In resampling, the training set S is repeatedly split into training and test sets. Afterward, the results of all iterations are aggregated, usually by taking the average. Hence, resampling allows for low pessimistic bias by using large training sets while simultaneously reducing the variance of the test error. Formally, let $S_{tr,1}, ..., S_{tr,\mathcal{B}}$ and $S_{ts,1}, ..., S_{tr,\mathcal{B}}$ be a sequence of $\mathcal{B}$ train and test sets that are sampled from S. The collection of splits can be written as a vector

$$\mathcal{S} = ((S_{tr,1}, S_{ts,1}), ..., (S_{tr,\mathcal{B}}, S_{ts,\mathcal{B}}))$$

where the resampling estimator is

$$\hat{\mathcal{E}}(h, \mathcal{S}, \phi) = \text{agr}(\phi(\mathbf{y}_{S_{ts,1}}, \boldsymbol{h}_{S_{ts,1}, h_{S_{tr,1}}}), ..., \phi(\mathbf{y}_{S_{ts,\mathcal{B}}}, \boldsymbol{h}_{S_{ts,b}, h_{S_{tr,\mathcal{B}}}})),$$

with $\boldsymbol{h}$ denoting the vector of individual predictions on the test set. In most cases, the mean is used for the aggregation function. Further, it is assumed that the sizes of the training sets are approximately equal.

A common resampling technique is *$k-$fold Cross-Validation*, where the set S is split into k partitions of roughly equivalent size. Every subset of S is used exactly once as a test set, while the remaining $k - 1$ parts are used for training. Consequently, k test errors are provided and averaged. This procedure is reasonable because every example in S is tested exactly once. Typically, 5-10 folds are used in practice. The case $k = n$ is known as ”leave-one-out” cross-validation, which can drastically reduce bias due to the large training set size of the estimate. However, leave-one-out cross-validation significantly increases the variance of the test error. Therefore, small samples are commonly used to reduce variance. Especially in the case of small n, repeated cross-validation over a high number of folds can improve the bias-variance problematic.

Subsampling is another resampling technique and can simply be described as repeated hold-out splitting that averages the test results. In practice, roughly $80 - 90\%$ of the set S is used for training. Choosing a large training set size reduces pessimistic bias, while the variance is decreased by averaging.

In *Bootstraping*, $\mathcal{B}$ training sets of size n are drawn from S with replacement. The test set are the so called "Out-of-Bag" points $S_{ts}^b = S \backslash S_{tr}^b$ for $b \in \{1, ..., \mathcal{B}\}$. Bootstrap is quite similar to subsampling but the replication of training points can lead to problems with the prediction. On average, each bag contains roughly 63% unique samples.

In resampling, $\mathcal{S}$ is used to learn and test models on different subsets of S. Because the model is fitted on the entirety of S, $\mathcal{E}(h, S)$ functions as a surrogate for $\mathcal{E}(h)$. Given $\hat{\mathcal{E}}(h, \mathcal{S}, \phi_{\mathcal{L}})$ and the argument agr being the average, the ϕ's over the different subsets are simple holdout estimates:

$$\mathbb{E}[\hat{\mathcal{E}}(h, \mathcal{S}, \phi)] \approx \mathbb{E}[\phi(\mathbf{y}_{S_{ts,b}}, \boldsymbol{h}_{S_{ts,b}, h_{S_{tr,b}}})].$$

It is straightforward to see that

$$\mathbb{E}[\phi(\mathbf{y}_{S_{ts,b}}, \boldsymbol{h}_{S_{ts,b}, h_{S_{tr,b}}})] = \mathbb{E}\left[\frac{1}{|S_{ts,b}|} \sum_{(\mathbf{x},y) \in S_{ts,b}} \mathcal{L}(y, h_{S_{tr,b}}(\mathbf{x}))\right] = \mathcal{E}(h_{S_{tr,b}})$$

is an unbiased estimator for the true generalization error of the model trained on $S_{tr,b}$. Because resampling allows for large training sets, the estimate $\mathcal{E}(h_{S_{tr,b}})$ is approximately equivalent to $\mathcal{E}(h_S)$ and yields therefore lower pessimistic bias in comparison to regular holdout splitting. Subsequent averaging reduces the variance of the test errors. However, methods like cross-validation or bagging cannot be used for t-test or other parametric inferential methods because samples are not drawn independently. On the other hand, holdout splitting enables an unbiased calculation of the generalization variance and hence, for the application of the central limit theorem.

2.7.4. Underfitting and Overfitting

Some reasons for the occurrence of overfitting have already been discussed in section 2.4 (noisy data, model complexity etc.). Overfitted models usually have high test errors because the model has been fitted to the intrinsic noise of the data. This can result from a complex hypotheses space or a lack of sufficient training data. If fewer data is available, regularization can be used to restrain the model. Generally speaking, the simplest model with the lowest generalization error should be selected. Also, if the tradeoff between model complexity and the decreased test error is not sufficient, one usually chooses the simpler model. On the other hand, if a model is poorly learned on the training set, then its ability to generalize is significantly affected. Such a model also performs badly during training. One also says the model is "underfitted" (Hastie et al., 2009).

2.7.5. Measures for Performance Evaluation

In regression, one commonly used performance measure is the *Mean Squared Error (MSE)*, which is equivalent to the average square loss as shown in section 2.2.1. Alternatively, the mean absolute error can be used (see also section 2.2.1). Another measure for a test set of size n is given by the *Root* MSE:

$$\phi_{RMSE}(y, h(\mathbf{x})) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - h(\mathbf{x}_i))^2}.$$

Section 2.2.2 introduced the zero-one loss. In the context of performance evaluation, the zero-one loss can be referred to as the *Misclassification Error Rate (MCE)* that counts the number of misclassified examples and divides them by the total number of predictions:

$$\phi_{MCE} = \frac{1}{n}\sum_{i=1}^{n}[y_i \neq h(\mathbf{x}_i)] \in [0, 1]$$

Analogously, accuracy is given by:

$$\phi_A = 1 - \phi_{MCE}$$

Especially in binary classification, the confusion matrix is often used for evaluation. Denote by $\hat{y}$ the predicted class and let y be the true label. Then the confusion matrix is

$\hat{y}/y$	$+$	$-$
$+$	True Positive (TP)	False Positive (FP)
$-$	False Negative (FN)	True Negative (TN)

Table 1: Confusion Matrix

where True Positive (TP), True Negative (TN) are the number of correctly classified positive/negative instances, False Negative (FN) is an incorrectly classified positive instance and False Positive (FP) and incorrectly classified negative instance. The accuracy can be calculated by $(\text{TP} + \text{TN})/n$. One is often also interested in the True Positive Rate (TPR): $\text{TPR} = \text{TP}/(\text{TP}+\text{FN})$ and the True Negative Rate (TNR): $\text{TNR} = \text{TN}/(\text{TN} + \text{FP})$.

2.8. Hyperparameter Tuning

As introduced in section 2.1, models are defined in terms of their parameterization. The ideal vector of parameters $\boldsymbol{\theta}$ is learned during training. On the contrary, *Hyperparameters* are set input parameters that are determined before the training process. Many algorithms are quite sensitive to their hyperparameter configuration. Therefore, the determination of a suitable input is crucial. *Hyperparameter Optimization (HPO)* as described by Bischl et al. (2021) is the process of tuning the parameter configuration and performing subsequent risk minimization to find the ideal model. Let $\boldsymbol{\psi}$ be a hyperparameter configuration and let $\tilde{\boldsymbol{\Psi}}$ be the search space. The set of all hyperparamter configurations is given by $\boldsymbol{\Psi}$, with $\tilde{\boldsymbol{\Psi}} \subset \boldsymbol{\Psi}$. Now define $\tilde{\boldsymbol{\Psi}} = \bigtimes_{j=1}^{k} \tilde{\boldsymbol{\Psi}}_j$. Notice that for all j, $\tilde{\boldsymbol{\Psi}}_j$ is a bounded subset of the jth hyperparameter. Further, $\tilde{\boldsymbol{\Psi}}_j$ is not restricted to being a continuous space. In many cases, the configurations of the hyperparameters are dependent on another. The HPO problem can be formally expressed as

$$\boldsymbol{\psi}^* \in \underset{\boldsymbol{\psi} \in \tilde{\boldsymbol{\Psi}}}{\arg\min}\ \hat{\mathcal{E}}(h, \mathcal{S}, \phi, \boldsymbol{\psi})$$

where $\boldsymbol{\psi}^*$ is the ideal hyperparameter configuration. The generalization error $\hat{\mathcal{E}}(h, \mathcal{S}, \phi, \boldsymbol{\psi})$ for a given h is estimated with respect to a specific configuration vector $\boldsymbol{\psi}$ based on a resampling split $\mathcal{S} = ((S_{tr,1}, S_{ts,1}), ..., (S_{tr,\mathcal{B}}, S_{ts,\mathcal{B}}))$. The evaluation of the hyperparamter configurations can be a difficult process because optimization cannot take place in the usual sense as discussed in section 2.3. Therefore, it is also referred to as a black-box optimization problem.

2.8.1. HPO Algorithms

Algorithms for HPO typically suggest different hyperparameter configurations $\boldsymbol{\psi}$ and evaluate their performance afterward. Configuration and performance are then saved in a given archive. Two simple but very common methods are *Grid* and *Random Search*. In grid search, the possible hyperparameter configurations are determined beforehand on a high-dimensional grid. Every fixed point is then evaluated during the training process. For every possible hyperparamter in a hyperparameter vector $\boldsymbol{\psi}$, a specific set of configurations is determined. Then all the possible combinations with the other hyperparameters are evaluated exhaustively. The main advantage of grid search is its straightforward application. However, it cannot be considered to be particularly efficient because the number of possible combinations increases exponentially and areas with unsuitable configurations are not omitted. Random search on the other hand randomly tries different hyperparameter configurations based on a predetermined probability distribution (e.g. uniform distribution). It can be stopped and reconfigured anytime and hence may not search through irrelevant areas as exhaustively as grid search with its predetermined configurations. However, since random search is based on randomness, it is still not an efficient optimization technique. Other more advanced HPO techniques are e.g. Bayesian optimization based on stochastic processes or Hyperbands. For more in-depth information, the interested reader is referred to Bischl et al. (2021) where a more elaborate discussion of these approaches can be found.

2.8.2. Nested Resampling

In section 2.7.3 resampling methods for improved performance evaluation have been discussed. The introduction of fixed hyperparameters increases the complexity of finding the ideal model because aspects like configurations, preprocessing, feature selection and model learning have to be considered. HPO can be a challenging task, for example using cross-validation for model evaluation over different hyperparameter configurations may bias the test error because information about the repeatedly used test sets can influence the model fit. A suitable countermeasure is a threefold split of the data set S into the training set S_{tr}, a validation set S_{vd} and the test set S_{ts}. S_{ts} can only be used for the evaluation of the final model to provide an unbiased estimate of the generalization error. Training and HPO are performed on the training and validation set. A model is learned with a specific hyperparameter configuration that is subsequently evaluated on the validation set. During tuning, several configurations are evaluated on the validation set (for example via cross-validation). After determining the best parameter vector $\boldsymbol{\psi}$, the model is again trained on the union of the training and validation set. Afterward, the estimate of the true performance can be determined on the unseen test set (Bischl et al., 2012).

However, this procedure may again limit the amount of training data that can be used to learn the model and therefore increase the pessimistic bias. *Nested resampling* can ameliorate this problematic. First, the data set S is separated in an outer resampling procedure like k-fold cross-validation. Within every fold, an untouched test set $S_{ts,j}, j \in \{1, ..., k\}$ is determined. On the remaining training set $S_{tr,j}$, p-fold cross-validation is used to find the ideal $\boldsymbol{\psi}$. Every p-fold split is evaluated on a validation set that is a subset of $S_{tr,j}$. For each fold, the ideal $\boldsymbol{\psi}^*$ is determined and the model is subsequently learned on the entire training set of fold j. Then, the performance is evaluated on $S_{ts,j}$. Therefore, k unbiased estimates of the generalization error are provided, which as a consequence, can decrease the pessimistic bias and the variance by simple averaging (Bischl et al., 2012).

3. Decision Trees

Decision Trees, often referred to by the term CART as introduced by Breiman et al. (1984), are simple tree-based models. A decision tree (for a rigorous description, one is referred to the sequel) is a rooted tree that partitions the training data at each split. The (in almost all cases binary) splits are created in a top-down greedy approach that starts at the root node v_0 and iterates over all possible features and values. At every node, a feature partitions the space by some condition. Predictions are then determined in the leaves of the tree (also called terminal nodes) at the end of every path. For every terminal node, there exists a specific vector of feature conditions. Notice that terminal nodes do not partition the training set/feature space. Depending on a task, a decision can either be a classification decision or the output of a real number. Furthermore, for every example, there is exactly one terminal node that outputs a prediction. Also, the prediction of a terminal node is identical for all examples in the node.

Every point in the feature space $\mathcal{X}$ is assigned to exactly one terminal node, and for every terminal node, there exists a vector of unique conditions that is determined by axis-parallel splits. Therefore, the hypothesis space of a decision tree is the set of all step functions over rectangular partitions in the feature space. Formally, the output values of a decision tree are given by

$$h(\mathbf{x}) = \sum_{t \in T} c_t \mathbb{1}(\mathbf{x} \in R_t)$$

where a tree with $|T|$ terminal nodes has $|T|$ "rectangles" R_t. Then, c_t is the predicted output for some $t \in T$. The splits are again determined by empirical risk minimization. Let $S' \subseteq S$ be the data assigned to a terminal node $t \in T$. Denote by c the predicted constant value for the data in t : $\hat{y} = c \; \forall \; (\mathbf{x}, y) \in S'$. Hence, the empirical risk $\mathcal{R}^*(t)$ for a leaf is simply the average loss of the examples in t for some loss function $\mathcal{L}$. In a slight abuse of notation, denote the random variable X_j by x_j. A split with feature x_j at a specific value w divides a parent node $v \in V$, where V is the set of nodes into $v_1 = \{(\mathbf{x}, y) \in S' : x_j \leq w\}$ and $v_2 = \{(\mathbf{x}, y) \in S' : x_j > w\}$. The quality of the split is evaluated by summing up the risk in both child nodes

$$\mathcal{R}^*(v, x_j, w) = \frac{1}{|v|} \left(\sum_{(\mathbf{x},y) \in v_1} \mathcal{L}(y, c_1) + \sum_{(\mathbf{x},y) \in v_2} \mathcal{L}(y, c_2) \right).$$

Therefore, the best split of v into v_1, v_2 is determined by solving

$$\arg\min_{j,w} \mathcal{R}^*(v, x_j, w).$$

The growing process starts with an empty root node that contains the entire data set S. Then, by recursive binary splitting, all possible feature splits and values are evaluated in a greedy approach and the ideal split that minimizes the empirical risk with respect to $\mathcal{R}^*(v, x_j, w)$ is determined. After splitting, the examples are distributed to the child nodes based on their feature conditions. Afterward, the split finding process is repeated recursively for each child node.

3.1. Classification Trees

Let S be a training set of size n and $\mathcal{Y} = \{1, ..., g\}$ a multiclass label space. The aim of a *Classification Tree* is to minimize the impurity (risk) at each leaf to achieve an accurate classification decision. There exists different split criterion that determine the impurity of a split.

The *Gini Impurity* measures the probability that a new example is incorrectly classified and can be derived by minimizing the Brier score introduced in section 2.2.2. For the split at node v write

$$\begin{aligned}\mathcal{R}^*_G(v) = \frac{1}{|v|}\sum_{(\mathbf{x}_i,y_i)\in v}\mathcal{L}(y_i,h(\mathbf{x}_i)) &= \frac{1}{|v|}\sum_{(\mathbf{x}_i,y_i)\in v}\sum_{k=1}^{g}\left(\mathbb{1}_{\{y_i=k\}} - p^v_k(\mathbf{x})\right)^2 \\ &= \sum_{k=1}^{g} p^v_k(\mathbf{x})\left(1-p^v_k(\mathbf{x})\right) \\ &= 1-\sum_{k=1}^{g} p^v_k(\mathbf{x}) = G(v)\end{aligned}$$

where $p_k(\mathbf{x}) = \frac{1}{|v|}\sum_{(\mathbf{x},y)\in v}\mathbb{1}_{\{y_i=k\}}$. The total impurity of a binary split over v is

$$\mathcal{R}^*(v) = \frac{|v_1|}{|v|}G(v_1) + \frac{|v_2|}{|v|}G(v_2).$$

The minimization of the cross-entropy loss is also used as splitting criterion and is equivalent to minimizing the entropy impurity

$$\begin{aligned}\mathcal{R}^*_E(v) = \frac{1}{|v|}\sum_{(\mathbf{x}_i,y_i)\in v}\mathcal{L}(y_i,h(\mathbf{x}_i)) &= \frac{1}{|v|}\sum_{(\mathbf{x}_i,y_i)\in v}\left(-\sum_{k=1}^{g}\mathbb{1}_{\{y_i=k\}}\log p^v_k(\mathbf{x})\right) \\ &= \sum_{k=1}^{g} p^v_k(\mathbf{x})\log p^v_k(\mathbf{x})) = E(v)\end{aligned}$$

where $p_k(\mathbf{x}) = \frac{1}{v}\sum_{(\mathbf{x},y)\in v}\mathbb{1}_{\{y_i=k\}}$. The total entropy follows analogously.

Information Gain is the reduction of information entropy in comparison to its previous state by the split of some feature x_j, which assumes a specific value w. The training set S is separated by a split with feature x_j such that $v_{x_j}(w) = \{\mathbf{x}\in v | x_j = w\}$ with $w \in x_j$. Therefore,

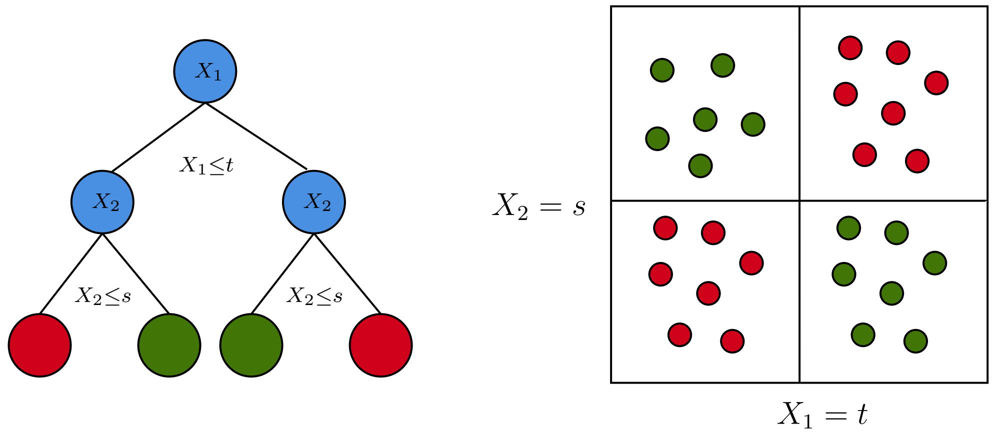
$$IG(v,x_j) = \mathcal{R}^*_E(v) - \mathcal{R}^*_E(v|x_j)$$

The conditional entropy $\mathcal{R}^*(v|x_j)$ is defined as:

$$\mathcal{R}^*_E(v|w) = \sum_{w\in x_j}\frac{|v_{x_j}(w)|}{|v|}\cdot E(v_{x_j}(w)).$$

A classification tree maximizes the information gain at each split by finding the ideal feature with an ideal threshold value w so that $IG(v, x_j)$ is maximized. Rectangular partitions of a binary classification tree are demonstrated in figure 5. A straightforward weakness of decision trees are that trends are difficult to model because the partitions of decision trees are not smooth functions. Especially in regression problems, trees do not extrapolate well.

Figure 5: Binary Classification Tree



3.2. Regression Trees

The task of a *Regression Tree* is the prediction of some continuous label space $\mathcal{Y} \subseteq \mathbb{R}$. The best split is usually determined by the feature that minimizes the sum of squares for a subset $S' \subseteq S$. Let v be the node that contains S', then

$$\mathcal{R}^*(v) = \frac{1}{|v|} \sum_{(\mathbf{x},y)\in v} (y - c)^2$$

where $\arg\min_{c\in\mathbb{R}} = \frac{1}{|v|} \sum_{(\mathbf{x},y)\in v} y = \overline{y}_v$ is the average value of the label y that results from solving the minimization problem of the square loss. If the l_1 loss is applied, the median can be used as the risk minimizing constant. For the split of a continuous feature x_j, the mean of every ordered pair of examples is calculated and the square loss is determined until the ideal split is found such that $\min_{\overline{y}_v} \mathcal{R}^*(v)$. Formally, the label space $\mathcal{Y}$ is separated into $|T|$ disjoint terminal regions $R_1, ..., R_{|T|}$. For every region R_t the mean is calculated as prediction. It is convenient to imagine that the label space is separated into high-dimensional rectangles. The task is to find $R_1, ..., R_{|T|}$ terminal regions that minimize the residual sum of squares (RSS):

$$RSS = \sum_{t \in T} \sum_{i \in R_t} (y_i - \overline{y}_{R_t})^2$$

$\overline{y}_{R_t} = \frac{1}{|S_{R_t}|} \sum_{i \in R_t} y$ is the mean of label y in the tth rectangle.

3.3. Cost Complexity Pruning

Pruning is the procedure of building a large tree τ_0 that gets pruned to obtain some subtree $\tau \subseteq \tau_0$. Some positive tuning parameter α is used, where α is determined by k-fold cross-validation. *Cost Complexity Pruning (CCP)* considers a sequence of subtrees and selects the subtree that minimizes the testing error, over different values of α.

A pruning algorithm can be described as follows: (1) A tree τ_0 is built by recursive binary splitting such that every terminal node $t \in T$, where T is the set of terminal nodes, has to include less or equal to a specific number of examples. (2) CCP is applied to determine the best subtree as a function of α, where α is determined by k-fold cross-validation. Steps (1) and (2) are executed for all $k - 1$ training subsets. Afterward, the RSS of the kth subset is evaluated as a function of α. The average loss for each value of α has to be determined. Subsequently, α is selected such that the average loss is minimized.

Let $|T|$ be the number of terminal nodes, R_t the terminal region of the tth terminal node and $\overline{y}_{R_t}$ the associated prediction. α operates as a trade-off between the subtree complexity and fit to the training set S. For every value of α there exists a subtree $\tau \subseteq \tau_0$, such that

$$RSS = \sum_{t=1}^{|T|} \sum_{i \in R_t} (y_i - \overline{y}_{R_t})^2 + \alpha |T|$$

is as small as possible. This is equivalent to bounding $|T|$:

$$\min_{\overline{y}_{R_t}} \sum_{i} (y_i - \overline{y}_{R_t})^2, \ |T| \leq c_\alpha$$

With Lagrange Multipliers it can be formulated that

$$\Delta_g = \sum_i (y_i - \overline{y}_{R_t})^2 + \lambda(|T| - c_\alpha)$$

which is a discrete optimization problem that has to be solved for T and λ. Therefore, the minimization is independent of c_α, but every c_α implies an ideal value for λ. Therefore, a value for λ can be chosen arbitrarily to optimize the equation

$$\Delta_{g'} = \sum_i (y_i - \overline{y}_{R_t})^2 + \lambda(|T|)$$

Letting $\lambda = \alpha$ results in the previous optimization problem.

4. Bagging and Feature Importance

Bootstrap Aggregation or in short *Bagging* reduces the variance of the predictor and has been developed by Breiman (1996). As an example, the bias-variance decomposition of the square loss is given by:

$$\mathbb{E}_{\mathbf{x},y,S}\left[(h_S(\mathbf{x}) - y)^2\right] = \underbrace{\mathbb{E}_{\mathbf{x},S}\left[(h_S(\mathbf{x}) - \overline{h}(\mathbf{x}))^2\right]}_{Variance} + \underbrace{\mathbb{E}_{\mathbf{x},y}[((\overline{h}(\mathbf{x}) - y)^2]}_{Bias^2} + \mathcal{N}$$

where $\mathcal{N}$ is a noise term. Reducing the variance is equal to letting the term $\mathbb{E}_{\mathbf{x},S}\left[(h_S(\mathbf{x}) - \overline{h}(\mathbf{x}))^2\right]$ be as small as possible, in other notation: $h_S \to \overline{h}$. Applying the weak law of large numbers yields that the average hypothesis tends towards the expected hypothesis. Let $S_1, ..., S_\mathcal{B}$ be training sets that are drawn from the distribution $\mathcal{D}$. Learning a hypothesis (base learner) on every training set and taking the average prediction gives

$$\hat{h} = \frac{1}{\mathcal{B}} \sum_{b=1}^{\mathcal{B}} h_{S_b} \to \overline{h}.$$

and consequently

$$\lim_{\mathcal{B}\to\infty} P\left(|\frac{1}{\mathcal{B}}\sum\nolimits_{b=1}^{\mathcal{B}} h_{S_b} - \overline{h}| \geq \epsilon\right) = 0 \Longrightarrow \mathbb{E}_{\mathbf{x},S}\left[(\hat{h}_S(\mathbf{x}) - \overline{h}(\mathbf{x}))^2\right] \to 0.$$

However, usually in practice only one training set S is available. Therefore, repeatedly drawing from $\mathcal{D}$ has to be simulated by drawing samples from

S with replacement. Although the samples are not drawn independently, bagging can reduce the variance of a hypothesis drastically. A sequence of bagged base learners $h_1, ..., h_{\mathcal{B}}$ is called an ensemble. The final model is

$$H_{\mathcal{B}}(\mathbf{x}) = \frac{1}{\mathcal{B}} \sum_{b=1}^{\mathcal{B}} h_b(\mathbf{x}).$$

One advantage of bagging is that the average and variance of the hypotheses can be determined. For a regression problem, the average prediction is calculated. In a classification task, the majority vote of the ensemble is used. Another advantage of bagging is that an unbiased estimate of the testing error is provided, typically referred as *Out-of-Bag (OOB) error.* The idea is quite simple. Not every pair $(\mathbf{x}_i, y_i)$ is drawn for all the sample S_b. Let H_k be the ensemble of all hypotheses that did not include the example $(\mathbf{x}_i, y_i)$ in their training set, with $k < \mathcal{B}$. The result is a test sample, since H_k has not been learned on the example $(\mathbf{x}_i, y_i)$. Calculating the OOB error for all examples results in an estimate for the true testing error.

Formally, for every example $(\mathbf{x}_i, y_i) \in S$, denote by $\mathcal{S}_k = \{k : (\mathbf{x}_i, y_i) \notin S_k\}$ the set of training sets S_k that do not include $(\mathbf{x}_i, y_i)$. Let the average hypothesis over these sets be

$$H_k(\mathbf{x}) = \overline{h}_k(\mathbf{x}) = \frac{1}{|\mathcal{S}_k|} \sum_{k \in \mathcal{S}_k} h_k(\mathbf{x})$$

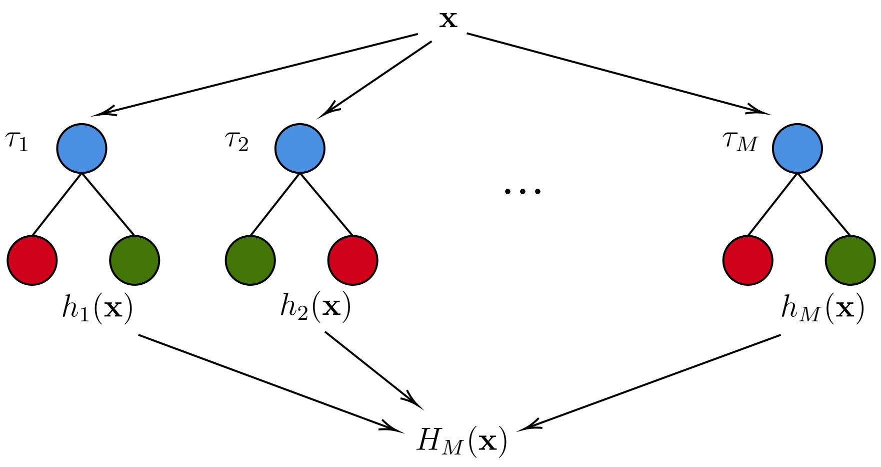
Therefore, the OOB error is simply the average loss over all these hypotheses:

$$\hat{\mathcal{E}}_{OOB}(H_{\mathcal{B}}) = \frac{1}{n} \sum_{(\mathbf{x}_i, y_i) \in S} \mathcal{L}(y_i, \overline{h}_k(\mathbf{x}_i))$$

which is an estimate of the test error that uses a subset of hypotheses that has not been learned on the example $(\mathbf{x}_i, y_i)$. If $\mathcal{B}$ is sufficiently large, neglecting some of the hypotheses does not substantially influence the estimate. Bagging maximizes its potential if the base learners are more variable (small pairwise correlations) given that their risk is not increased. Therefore, bagging of decision trees may be an effective way to increase performance.

One of the most popular bagging algorithms is the *Random Forest* that has been developed by Ho (1995) and Breiman (2001). A random forest uses decision trees as base learners and randomly samples a subset of k features at each split. A popular choice for k is $k = \sqrt{d}$. The random subsampling of the features decorrelates the base learners and consequently increases their variance, which yields better predictions because the errors are averaged out (the learners are wrong in different ways, roughly speaking). Figure 6 demonstrates the decision-making process of an unlabeled instance $\mathbf{x}$ for a random forest of size M. Notice how the individual decisions of the base learners lead to an aggregated decision. Also, the contributions of the base learners are weighted by the factor $1/M$ which is equivalent to the size of the ensemble.

Figure 6: Random Forest



The pseudocode of the random forest can be denoted by:

Algorithm 1: Random Forest Algorithm

```
1 sample $M$ data sets $\mathcal{S}_1...,\mathcal{S}_M$ with replacement;
2 for m=1 to M do
3     train a full decision tree $h_m()$ on $\mathcal{S}_m$;
4     for each split do
5         randomly subsample $k \leq d$ features without replacement;
6     end
7 end
8 return $h(\mathbf{x}) = \frac{1}{M}\sum_{m=1}^{M} h_m(\mathbf{x})$
```

Feature Importance
As shown in section 4, bagging can drastically improve performance by averaging over a set of variable base learners and thus reduce variance of the prediction. Decision trees are a common choice for bagging because their growing process is sensitive to the training set (consequently resulting in variable base learners). A single tree is quite easy to interpret but its predictive power is rather mediocre. Random forests are one of the most common implementations of tree ensembles. But the high accuracy of random forest comes at the expense of interpretability since the interpretation of several trees at the same time is not a feasible process (Hastie et al., 2009). One way to ameliorate the interpretability of random forests is the evaluation of measures that analyze the contributions of the different features in the ensemble. The so-called *Feature Importance* aims to determine how much a feature affects the predictive validity of the forest by removing a said feature from the ensemble. The determination of relevant features is a difficult process, especially when a large set of features is at hand that are dependent on another (Gregorutti et al., 2013). One way of identifying feature importance in a random forest is the summation of the splitting criterion over all trees and all feature splits. Splitting criteria for classification are the gini index or entropy, whereas the residual sum of squares can be used in the regression case (as discussed in detail in sections 3.1 and 3.2). The amount of risk reduction can be averaged over all trees and hence provides an estimate of the average feature importance for the ensemble. Generally speaking, the larger the reduction in risk, the greater the feature importance (Hastie et al., 2009).

But measuring feature importance based on the improvement of the split criterion can lead to bias when using the Gini index with categorical features, as they tend to be favored during the tree-building process (Breiman et al., 1984). Hence, permutation-based feature importance measures can be used instead that aim to either permutate the feature or response vectors (Gregorutti et al., 2013).

5. Boosting

Boosting originates from the formal Probably Approximately Correct (PAC) framework that has been developed by Valiant (1984) and Kearns and Vazirani (1994). This theoretical study of machine learning investigates under what condition an algorithm is able to output a *strong learner*, in other words, a learner that minimizes the training error until it is arbitrarily small. Naturally, the question arose whether an algorithm that performs slightly better than chance (*weak learner*), can be transformed into a strong learner, by combining the predictions of an ensemble of weak learners (Freund and Schapire, 1995). This process is called boosting.

Boosting is a sequential process, where weak learners are built in an iterative fashion in dependence on the previous outputs. In comparison, bagging trains base learners simultaneously and independently of another on bagged subsets of the training set. Particularly for tree boosting, stumps are commonly used (trees with limited depth). Each stump aims to improve the performance of the aggregated predictions of the previous stumps by accounting for their mistakes. Further, boosting is typically performed on a single modified version of the training set (Hastie et al., 2009). In section 5.1, Adaptive Boosting (AdaBoost), introduced by Freund and Schapire (1995) is discussed. AdaBoost was the first algorithm that demonstrated that an ensemble of weak learners can reduce the training loss to an arbitrary margin. Section 5.2 elaborates on Gradient Boosting, which is a generalization of the boosting process as gradient descent optimization in function space is explained in greater detail.

5.1. AdaBoost

AdaBoost has been developed for binary classification problems of the kind $\mathcal{Y} = \{-1, +1\}$. Consider a hypothesis space of weak base learners (e.g. tree stumps) $h \in \mathcal{H} : h(\mathbf{x}) \in \{-1, +1\} \forall \mathbf{x} \in \mathcal{X}$, the exponential loss $\mathcal{L}(h) = e^{-yh(\mathbf{x})}$ and the initial weight of $w_i = \frac{1}{n}$ for all i given S. An ensemble classifier of size M outputs a weighted average

$$H_M(\mathbf{x}) = \sum_{m=1}^{M} \beta_m h(\mathbf{x}, \boldsymbol{\theta}_m)$$

as prediction, where the weak learners are sequentially created in the training process. In each iteration, AdaBoost allocates larger weights to misclassified examples and fits a weak learner to the newly weighted training set. Moreover, weak learners with higher accuracy receive larger weights in the final prediction. The classification error of iteration m can be denoted by $\epsilon_m = \sum_{i:h_m(\mathbf{x}) \neq y_i} w_{m,i}$. The weak learner in iteration $m + 1$ is found by minimizing the weights of the misclassified examples.

$$h_{m+1} = \arg\min_{h \in \mathcal{H}} \sum_{i:h_m(\mathbf{x}) \neq y_i} w_{m,i}.$$

In AdaBoost, the ideal step size can be determined in closed form by the equation

$$\beta_m = \frac{1}{2} \ln \left(\frac{1 - \epsilon_m}{\epsilon_m} \right)$$

and the new weights are updated with

$$w_{m+1,i} = w_{m,i} \cdot \exp(-\beta_m \cdot y_i \cdot h_m(\mathbf{x}_i)).$$

Afterward, the updated weights are normalized such that $\sum_{i=1}^{n} w_{m+1,i} = 1$ and the loop is continued until iteration M is reached. It can be shown that the training error of AdaBoost decreases exponentially fast. Furthermore, the number of iterations required for the training loss to converge to 0 can be determined. Also, AdaBoost uses weak learners (decision trees with restricted depth) and reduces bias, whereas random forests uses unrestricted base learners and reduces variance by averaging (all base learners are weighted equally).

5.2. Gradient Boosting

Friedman (2001) demonstrated that boosting can be described as gradient descent optimization in function space by the development of the *Gradient Boosting Machine*. Consider again an additive model

$$H_M(\mathbf{x}) = \sum_{m=1}^{M} \beta_m h(\mathbf{x}, \boldsymbol{\theta}_m).$$

The risk minimization of $\mathcal{R}(H_M(\mathbf{x})) = \sum_{i=1}^{n} \mathcal{L}(y_i, H_M(\mathbf{x}_i))$ of such an additive model is dependent on the weights and parameters of the chosen base learners and can therefore be difficult to obtain. Friedman proposes a greedy forward stagewise additive modeling approach that tries to find the best weak learner at each iteration m, without readjusting the learners of the previous iterations. Given a hypothesis space $\mathcal{H}$ of weak base learners, for every stage m find $\beta_m, \boldsymbol{\theta}_m$ such that

$$(\beta_m, \boldsymbol{\theta}_m) = \underset{\beta, \boldsymbol{\theta}}{\arg\min} \sum_{i=1}^{n} \mathcal{L}(y_i, H_{m-1}(\mathbf{x}_i) + \beta h(\mathbf{x}_i, \boldsymbol{\theta}))$$

and update the model with

$$H_m(\mathbf{x}) = H_{m-1}(\mathbf{x}) + \beta_m h(\mathbf{x}, \boldsymbol{\theta}_m)$$

where $H_{m-1}(\mathbf{x}) = \sum_{j=1}^{m-1} \beta_j h(\mathbf{x}, \boldsymbol{\theta}_j)$ denotes the weighted ensemble of the first $m-1$ iterations. Optimization is achieved by gradient descent, where for some non-parametric model and a differentiable loss function $\mathcal{L}$, the negative gradient is computed by

$$-\nabla \mathcal{L}_m(\mathbf{x}_i) = -\left[\frac{\partial \mathcal{L}(y_i, H(\mathbf{x}_i))}{\partial H(\mathbf{x}_i)}\right]_{H(\mathbf{x}) = H_{m-1}(\mathbf{x})} = r_{im}.$$

Notice that the gradient $\nabla \mathcal{L}_m(\mathbf{x}_i)$ is only valid for the specific example $\mathbf{x}_i$ and correspondingly does not generalize to the entire data set. One solution to this problem is to find a parameterized base learner that outputs $h(\mathbf{x}_i, \boldsymbol{\theta}_m)$ most parallel to the pseudo residuals r_{im} for all i. In other words, a base learner is fitted to the pseudo residuals that minimizes the loss. Therefore, the optimization

$$\boldsymbol{\theta}_m = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{n} \mathcal{L}(r_{im}, h(\mathbf{x}, \boldsymbol{\theta}))$$

provides the ideal parameter that minimizes the loss with respect to the responses of the previous iterations. Then, the line search for the ideal step size is determined by

$$\beta_m = \arg\min_{\beta} \sum_{i=1}^{n} \mathcal{L}(y_i, H_{m-1}(\mathbf{x}_i) + \beta h(\mathbf{x}_i, \boldsymbol{\theta}_m))$$

and the update is

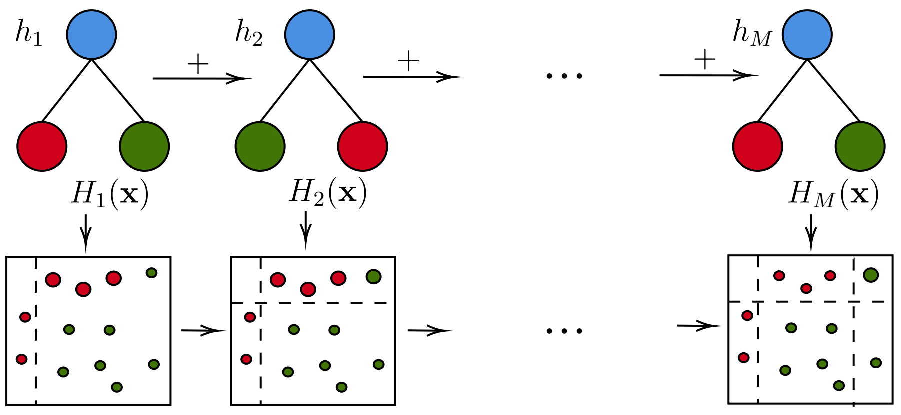
$$H_m(\mathbf{x}) = H_{m-1}(\mathbf{x}) + \beta_m h(\mathbf{x}, \boldsymbol{\theta}_m).$$

Using a common least squares function for the minimization of the pseudo residuals yields that most regression base learners can be used for gradient boosting. Then, the parameter search reduces to

$$\boldsymbol{\theta}_m = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{n} (r_{im} - h(\mathbf{x}, \boldsymbol{\theta}))^2.$$

A common choice for the base learners are regression trees of limited depth. The training procedure is illustrated in figure 7. Notice how the weak base learners emphasize on correcting the errors of previous iterations. The output is an aggregated strong learner.

Figure 7: Boosting Procedure



Trees can deal with categorical variables and provide an integrated feature selection mechanism. The general algorithm can be formulated as:

Algorithm 2: Gradient Boosted Regression Trees

```
1  input training data $\{\{\mathbf{x}_i, y_i\}\}_{i=1}^n$, differentiable loss function $\mathcal{L}(y,\gamma)$,
     number of iterations $M$, learning rate $\alpha$;
2  initialize $H_0(\mathbf{x}) = \arg\min_{\gamma} \sum_{i=1}^n \mathcal{L}(y_i,\gamma)$;
3  for $m = 1$ to $M$ do
4      $r_{im} = -\left[\frac{\partial \mathcal{L}(y_i, H(\mathbf{x}_i))}{\partial (H(\mathbf{x}_i))}\right]_{H(\mathbf{x})=H_{m-1}(\mathbf{x})}$ for $i = 1, ..., n$
5      Fit a regression tree $h_m(\mathbf{x})$ to pseudo-residuals by training on the
         set $\{(\mathbf{x}_i, r_i)\}_{i=1}^n$ to create terminal regions $R_{jm}$
6      for $j = 1...J_m$ do
7          $\gamma_{jm} = \arg\min_{\gamma} \sum_{x \in R_{jm}} \mathcal{L}(y_i, H_{m-1}(\mathbf{x}_i) + \gamma)$
8      end
9      $H_m(\mathbf{x}) = H_{m-1}(\mathbf{x}) + \alpha \sum_{j=1}^{J_m} \gamma_{jm} \mathbb{1}_{(\mathbf{x} \in R_{jm})}$
10 end
11 return $H_M(\mathbf{x}) = h_0(\mathbf{x}) + \alpha \sum_{m=1}^M h_m(\mathbf{x})$
```

Notice that the line search of β_m is replaced by setting some fixed learning rate α for all base learners.

Gradient Boost Classification

Gradient Boost Classification can be performed by using the logistic function for binary classification and the softmax function for multiclass classification, as described in section 2.6. Recall that the loss function of softmax regression was given by the cross-entropy

$$\mathcal{L}(y_k, h_k(\mathbf{x})) = -\sum_{k=1}^{g} \mathbb{1}_{\{y_i=k\}} \ln(p_k(\mathbf{x}))$$

and the posterior probability $p_k(\mathbf{x})$ is

$$p_k(\mathbf{x}) = \sigma(s_k(\mathbf{x}))_k = \frac{\exp(h_k(\mathbf{x}))}{\sum_{k=1}^{g} \exp(h_k(\mathbf{x}))}.$$

The pseudo residuals are then calculated by

$$r_{ik,m} = -\left[\frac{\partial \mathcal{L}(y_{ik}, H_k(\mathbf{x}_i))}{\partial H_k(\mathbf{x}_i)}\right]_{H_k(\mathbf{x})=H_{k,m-1}(\mathbf{x})} = y_{ik} - p_{k,m-1}(\mathbf{x}_i).$$

In each iteration m, g base learners are fitted to the pseudo residuals to predict the responses for class k. Given terminal nodes $R_{1k,m}, ..., R_{Jk,m}$ for the classes $k \in \{1, ..., g\}$ the updates $\gamma_{jk,m}$ are given by

$$\gamma_{jk,m} = \underset{\gamma_{jk,m}}{\arg\min} \sum_{i=1}^{n} \sum_{k=1}^{g} \phi\left(y_{ik}, H_{k,m-1}(\mathbf{x}_i) + \sum_{j=1}^{J_m} \gamma_{jk,m} \mathbb{1}_{(\mathbf{x}_i \in R_{jk,m})}\right)$$

where $\phi(y_k, H_k(\mathbf{x}))) = y_k \ln p_k(\mathbf{x})$ and hence

$$\gamma_{jk,m} = \underset{\gamma_{jk,m}}{\arg\min} \sum_{i=1}^{n} \sum_{k=1}^{g} \mathbb{1}_{(y_i=k)} \ln p_{k,m}(\mathbf{x}_i).$$

Because there exists no closed from solution for this equation and the regions of different class trees overlap, a single Newton-Raphson step is used for approximation. Therefore, resulting in

$$\gamma_{jk,m} = \frac{g-1}{g} \frac{\sum_{\mathbf{x}_i \in R_{jk,m}} r_{ik,m}}{\sum_{\mathbf{x}_i \in R_{jk,m}} |r_{ik,m}|(1 - r_{ik,m})}$$

where the updated model is given by

$$H_{k,m}(\mathbf{x}) = H_{k,m-1}(\mathbf{x}) + \sum_{j=1}^{J_m} \gamma_{jk,m} \mathbb{1}_{(\mathbf{x} \in R_{jk,m})}.$$

Multiclass Gradient Boosting outputs a total of g ensembles that make one hot encoded predictions. The tree that outputs the largest value is typically used for prediction. In the case of binary classification, the calculation of the pseudo residuals is given by $r_{im} = y_i - p_{m-1}(\mathbf{x}_i)$ and the minimization in the terminal regions are

$$\gamma_{jm} = \frac{\sum_{\mathbf{x}_i \in R_{jm}} r_{im}}{\sum_{\mathbf{x}_i \in R_{jm}} p_{m-1}(\mathbf{x}_i)(1 - p_{m-1}(\mathbf{x}_i))}$$

where it is easy to see that in the binary case a single ensemble suffices for classification. Note that gradient boost classification is not restricted to the softmax function, e.g. the exponential loss can also be implemented.

Algorithm 3: Multiclass Gradient Boosting

```
1  input $H_{0,k}(x) = 0, k = 1, ..., g$, number of iterations $M$ and learning
     rate $\alpha$;
2  for $m = 1$ to $M$ do
3      Set $p_{k,m}(\mathbf{x}) = \frac{\exp(H_{k,m}(\mathbf{x}))}{\sum_{k=1}^{g} \exp(H_{k,m}(\mathbf{x}))}, k = 1, ..., g$
4      for $k$ to $g$ do
5          Calculate $r_{ik,m} = y_{ik} - p_{k,m-1}(\mathbf{x}_i), i = 1, ..., n$
6          Fit a regression tree to pseudo residuals by creating terminal
             regions $R_{1k,m}, ..., R_{Jk,m}$
7          for $j = 1...J_m$ do
8              $\gamma_{jk,m} = \frac{g-1}{g} \frac{\sum_{\mathbf{x}_i \in R_{jk,m}} r_{ik,m}}{\sum_{\mathbf{x}_i \in R_{jk,m}} |r_{ik,m}|(1 - r_{ik,m})}$
9          end
10         Update $H_{k,m}(\mathbf{x}) = H_{k,m-1}(\mathbf{x}) + \sum_{j=1}^{J_m} \gamma_{jk,m} \mathbb{1}_{(\mathbf{x} \in R_{jk,m})}$
11     end
12 end
13 return $H_{1,M}(\mathbf{x}), ..., H_{g,M}(\mathbf{x})$
```

5.3. XGBoost

eXtreme Gradient Boosting (XGBoost), introduced by Chen and Guestrin (2016), is an extension of the common gradient boost algorithm and provides a specific pruning procedure that penalizes the complexity of the base learners. Most commonly, XGBoost is used with decision trees but also generalizes to other types of base learners. However, in this section, mainly XGBoost with decision trees is discussed.

Consider a regularization term $\Omega(h) = \gamma|T| + \frac{1}{2}\lambda\|c\|^2$, where γ, λ are both regularization hyperparameters, $|T|$ is the cardinality of the set of terminal nodes and c is the specific output value for a terminal node $t \in T$. The risk of such a regularized, additive model can then be expressed as

$$\mathcal{R}(H_m) = \sum_{i=1}^{n} \mathcal{L}(y_i, H_{m-1}(\mathbf{x}_i) + h_m(\mathbf{x}_i)) + \Omega(h_m).$$

The base learner of the m iteration is therefore added in a greedy manner with respect to the penalty term $\Omega(h_m)$. The optimization can be achieved by using a second-order Taylor approximation, which results in

$$\mathcal{R}(H_m) \approx \sum_{i=1}^{n} \left(\mathcal{L}(y_i, H_{m-1}) + r_i h_m(\mathbf{x}_i) + \frac{1}{2} l_i h_m^2(\mathbf{x}_i) \right) + \Omega(h_m)$$

by letting $r_i = \frac{\partial \mathcal{L}(y_i, H_{m-1})(\mathbf{x}_i)}{\partial H_{m-1}(\mathbf{x}_i)}$ and $l_i = \frac{\partial^2 \mathcal{L}(y_i, H_{m-1}(\mathbf{x}_i))}{\partial^2 H_{m-1}(\mathbf{x}_i)}$. Notice that r_i again denotes the pseudo residual of observation i with respect to the gradient. In XGBoost regression using l_2 loss, the residuals are given by $r_i = H_{m-1}(\mathbf{x}_i) - y_i$. Because $\mathcal{L}(y_i, H_{m-1})$ is a constant term it can be omitted for the optimization process. Therefore

$$\mathcal{R}(H_m) \approx \sum_{i=1}^{n} \left(r_i h_m(\mathbf{x}_i) + \frac{1}{2} l_i h_m^2(\mathbf{x}_i) \right) + \Omega(h_m).$$

Define R_j as the jth terminal region of base learner $h_m(\mathbf{x})$. Inserting $\Omega(h) = \gamma|T| + \frac{1}{2}\lambda\|c\|^2$ into the previous equation gives

$$\begin{aligned}
\mathcal{R}(H_m) &\approx \sum_{i=1}^{n} \left(r_i h_m(\mathbf{x}_i) + \frac{1}{2} l_i h_m^2(\mathbf{x}_i) \right) + \gamma|T| + \frac{1}{2}\lambda \sum_{j=1}^{|T|} c_j^2 \\
&= \sum_{j=1}^{|T|} \left[\left(\sum_{\mathbf{x}_i \in R_j} r_i \right) c_j + \frac{1}{2} \left(\sum_{\mathbf{x}_i \in R_j} l_i + \lambda \right) c_j^2 \right] + \gamma|T|.
\end{aligned}$$

Then, it is straightforward to see that the optimal output value w_j of terminal region R_j can be determined by

$$c_j' = -\frac{\sum_{\mathbf{x}_i \in R_j} r_i}{\sum_{\mathbf{x}_i \in R_j} l_i + \lambda}.$$

The scoring criterion of a base learner (tree) can now be evaluated by

$$\mathcal{R}(h_m) = -\frac{1}{2}\sum_{j=1}^{|T|} -\frac{\left(\sum_{\mathbf{x}_i \in R_j} r_i\right)^2}{\sum_{\mathbf{x}_i \in R_j} l_i + \lambda} + \gamma|T|$$

To make the computation more feasible, a greedy approach is used for tree splitting that evaluates the impurity of each split. This procedure restricts the number of possible base learners substantially. Now, denote by v_1 and v_2 the partitions of the binary split at node v. Splits are evaluated by the criterion

$$\mathcal{R}(v_{1,2}) = \frac{1}{2}\left[\frac{\left(\sum_{\mathbf{x}_i \in v_1} r_i\right)^2}{\sum_{\mathbf{x}_i \in v_1} l_i + \lambda} + \frac{\left(\sum_{\mathbf{x}_i \in v_2} r_i\right)^2}{\sum_{\mathbf{x}_i \in v_2} l_i + \lambda} - \frac{\left(\sum_{\mathbf{x}_i \in v} r_i\right)^2}{\sum_{\mathbf{x}_i \in v} l_i + \lambda}\right] - \gamma.$$

Residuals that lie in the same direction result in a large loss reduction, since the individual terms are not squared. The magnitude of a required loss reduction is dependent on the chosen γ. Also, observe that larger values of λ require more similarity of the residuals. As in regular gradient boosting, a constant learning rate is applied that scales the output values of the base learners (shrinkage) to prevent overfitting. Moreover, feature subsampling can be used similarly to random forests to further decrease the overfitting risk.

Furthermore, XGBoost provides efficient splitting criteria for large data sets. The common greedy tree splitting approach is effective but requires a substantial amount of resources because for each split, the algorithm has to iterate over all features and all possible split points. This can lead to memory exhaustion. Therefore, XGBoost implements a *Weighted Quantile Sketch (WQS)*. In a regular quantile sketch, several subsets of a large data set are sampled and an approximate feature distribution is created by parallelization. However, such an approach allocates the same "weight" (usually the number of examples) to all quantiles. The weighted WQS proposed by Chen and Guestrin (2016) tries to find the ideal candidate split points by considering all feature values $S_k = \{x_{1k}, ..., x_{nk}\}$ with respect to the second order gradient statistic l_i. A rank function creates candidate splits whose values are smaller than some w by

$$r_k(w) = \frac{1}{\sum_{(x,l)\in S_k} l} \sum_{(x,l)\in S_k, x<w} l$$

representing the proportion of examples with feature value k being strictly less than w. XGBoost aims to find a set of possible splits $\{s_{k,1}, ..., s_{k,p}\}$ such that for any $j, j+1 \in \{1, ..., p\}$

$$|r_k(s_{k,j}) - r_k(s_{k,j+1})| < \epsilon$$

where ϵ is an approximation factor and s_{k1}, s_{kp} denote the quantile with the smallest and larges value of feature k respectively over all i. If the data set is not large, XGBoost uses the regular greedy algorithm for split finding. XGBoost also provides efficient algorithms for missing feature values and parallel learning. These topics are beyond the scope of this thesis. Therefore, for the interested reader, the paper by Chen and Guestrin (2016) can provide further insight.

6. Proposing a new Interpretability-Method

Algorithms like random forest and tree-based gradient boosts are powerful predictive tools. However, the complex ensemble structure loses the straightforward interpretability of simple decision trees. In the following sequel, a method that transforms tree-based ensemble learners into a single, prediction equivalent decision tree is introduced. Proofs for the existence and reduction of the tree's order are provided. Further, general algorithms for the implementation are described. This method helps to improve the global interpretability of ensemble methods by giving access to the entire structure of the decision-making process. Also, decisions on the local level can be understood by examining individual paths and decisions.

References

Adadi, A. and Berrada, M. (2018). Peeking inside the black-box: A survey on explainable artificial intelligence (xai). *IEEE Access*, 6:52138–52160.

Bischl, B., Binder, M., Lang, M., Pielok, T., Richter, J., Coors, S., Thomas, J., Ullmann, T., Becker, M., Boulesteix, A.-L., Deng, D., and Lindauer, M. (2021). Hyperparameter optimization: Foundations, algorithms, best practices and open challenges.

Bischl, B., Mersmann, O., Trautmann, H., and Weihs, C. (2012). Resampling methods for meta-model validation with recommendations for evolutionary computation. *Evolutionary computation*, 20(2):249–275.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

Breiman, L., Friedman, J., Stone, C. J., and Olshen, R. (1984). *Classification and Regression Trees*. Chapman and Hall.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system.

Freund, Y. and Schapire, R. E. (1995). A desicion-theoretic generalization of on-line learning and an application to boosting. In Vitányi, P., editor, *Computational Learning Theory*, pages 23–37, Berlin, Heidelberg. Springer Berlin Heidelberg.

Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5).

Gregorutti, B., Michel, B., and Saint-Pierre, P. (2013). Correlation and variable importance in random forests.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning*. Springer New York, New York, NY.

Ho, T. K. (1995). Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, pages 278–282. IEEE Comput. Soc. Press.

Kearns, M. J. and Vazirani, U. V. (1994). *An Introduction to Computational Learning Theory.* MIT Press, Cambridge, MA, USA.

Khandani, A. E., Kim, A. J., and Lo, A. W. (2010). Consumer credit-risk models via machine-learning algorithms. *Journal of Banking & Finance*, 34(11):2767–2787.

Linardatos, P., Papastefanopoulos, V., and Kotsiantis, S. (2020). Explainable ai: A review of machine learning interpretability methods. *Entropy (Basel, Switzerland)*, 23(1).

Litjens, G., Kooi, T., Bejnordi, B. E., Setio, A. A. A., Ciompi, F., Ghafoorian, M., van der Laak, J. A. W. M., van Ginneken, B., and Sánchez, C. I. (2017). A survey on deep learning in medical image analysis. *CoRR*, abs/1702.05747.

Miller, T. (2017). Explanation in artificial intelligence: Insights from the social sciences.

Murphy, K. P. (2013). *Machine learning: A probabilistic perspective.* Adaptive computation and machine learning series. MIT Press, Cambridge, Mass., 4. print. (fixed many typos) edition.

Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach.* Prentice Hall, 3 edition.

Schmidt, J., Marques, M. R. G., Botti, S., and Marques, M. A. L. (2019). Recent advances and applications of machine learning in solid-state materials science. *npj Computational Materials*, 5(1).

Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms.* Cambridge University Press.

Valiant, L. G. (1984). A theory of the learnable.

Zawacki-Richter, O., Marín, V., Bond, M., and Gouverneur, F. (2019). Systematic review of research on artificial intelligence applications in higher education -where are the educators? *International Journal of Educational Technology in Higher Education*, 16:1–27.