Neural Processing Letters (2021) 53:1147–1160
https://doi.org/10.1007/s11063-021-10434-9

KTBoost: Combined Kernel and Tree Boosting

Fabio Sigrist[1]

Accepted: 23 January 2021 / Published online: 11 February 2021

Abstract

We introduce a novel boosting algorithm called 'KTBoost' which combines **k**ernel boosting and **t**ree boosting. In each boosting iteration, the algorithm adds either a regression tree or reproducing kernel Hilbert space (RKHS) regression function to the ensemble of base learners. Intuitively, the idea is that discontinuous trees and continuous RKHS regression functions complement each other, and that this combination allows for better learning of functions that have parts with varying degrees of regularity such as discontinuities and smooth parts. We empirically show that KTBoost significantly outperforms both tree and kernel boosting in terms of predictive accuracy in a comparison on a wide array of data sets.

Keywords Gradient and newton boosting · Reproducing kernel Hilbert space (RKHS) regression · Ensemble learning · Supervised learning

1 Introduction

Boosting algorithms [8,15,17,18,28] enjoy large popularity in both applied data science and machine learning research, among other things, due to their high predictive accuracy observed on a wide range of data sets [11]. Boosting additively combines base learners by sequentially minimizing a risk functional. Despite the fact that there is almost no restriction on the type of base learners in the seminal papers of Freund and Schapire [15] and Freund and Schapire [16], very little research has been done on combining different types of base learners. To the best of our knowledge, except for one reference [22], existing boosting algorithms use only one type of functions as base learners. To date, regression trees are the most common choice of base learners, and a lot of effort has been made in recent years to develop tree-based boosting methods that scale to large data [11,26,35,36].

In this article, we relax the assumption of using only one type of base learners by combining regression trees [7] and reproducing kernel Hilbert space (RKHS) regression functions [4,39] as base learners. In short, RKHS regression is a form of non-parametric regression which shows state-of-the-art predictive accuracy for many data sets as it can, for instance, achieve near-optimal test errors [1,2], and kernel classifiers parallel the behaviors of deep networks as noted in Zhang et al. [46]. As there is now growing evidence that base learners do not

✉ Fabio Sigrist
fabio.sigrist@hslu.ch

[1] Lucerne University of Applied Sciences and Arts, Suurstoffi 1, 6343 Rotkreuz, Switzerland

necessarily need to have low complexity [44], continuous, or smooth, RKHS functions have thus the potential to complement discontinuous trees as base learners.

1.1 Summary of Results

We introduce a novel boosting algorithm denoted by 'KTBoost' which combines **k**ernel and **t**ree boosting. In each boosting iteration, the KTBoost algorithm adds either a regression tree or a penalized RKHS regression function, also known as kernel ridge regression [30], to the ensemble. This is done by first learning both a tree and an RKHS function using one step of functional Newton's method or functional gradient descent, and then selecting the base learner whose addition to the ensemble results in the lowest empirical risk. The KTBoost algorithm thus chooses in each iteration a base learner from two fundamentally different function classes. Functions in an RKHS are continuous and, depending on the kernel function, they also have higher regularity. Trees, on the other hand, are discontinuous functions.

Intuitively, the idea is that the different types of base learners complement each other, and that this combination allows for better learning of functions that exhibit parts with varying degrees of regularity. We demonstrate this effect in a simulation study in Sect. 4.1. To briefly illustrate that the combination of trees and RKHS functions as base learners can achieve higher predictive accuracy, we report in Fig. 1 test mean square errors (MSEs) versus the number of boosting iterations for one data set (wine). The solid lines show average test MSEs over ten random splits into training, validation, and test data sets versus the number of boosting iterations. The confidence bands are obtained after point-wise excluding the largest and smallest MSEs. Tuning parameters of all methods are chosen on the validation data sets. See Sect. 4 for more details on the data set and the choice of tuning parameters.[1] The figure illustrates how the combination of tree and kernel boosting (KTBoost) results in a lower test MSE compared to both tree and kernel boosting. In our extensive experiments in Sect. 4.2, we show on a large collection of data sets that the combination of trees and RKHS functions leads to a lower generalization error compared to both only tree and only kernel boosting. Our approach is implemented in the Python package `KTBoost` which is openly available on the Python Package Index (PyPI) repository.[2]

1.2 Related Work

Combining predictions from several models has been successfully applied in many areas of machine learning such as diversity inducing methods [29] or multi-view learning; see e.g. Peng et al. [34] for a recent example of a boosting application. However, the way boosting combines base learners is different from traditional ensembles consisting of several models trained on potentially different data sets since, for instance, boosting reduces both variance and bias. Very little research has been done on combining different types of base learners in a boosting framework, and, to the best of our knowledge, there is no study which investigates the effect on the predictive accuracy when boosting different types of base learners.

The `mboost` R package of Hothorn et al. [22] allows for combining different base learners which include linear functions, one- and two-dimensional smoothing splines, spatial terms,

[1] For better comparison, the shrinkage parameter ν, see Eq. (4), is set to a fix value ($\nu = 0.1$) in this example. In the experiments in Sect. 4.2, the shrinkage parameter is also chosen using cross-validation.

[2] See https://github.com/fabsig/KTBoost for more information.

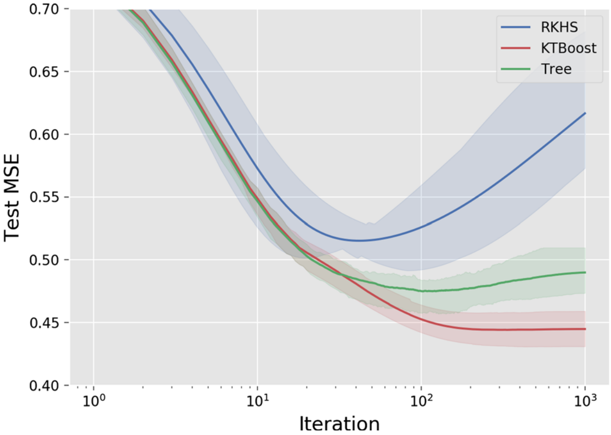

Fig. 1 Test mean square error (MSE) versus the number of boosting iteration for KTBoost in comparison with tree and kernel boosting for one data set (wine)

regression trees, as well as user-defined ones. This approach is different from ours since `mboost` uses a component-wise approach where every base learner typically depends on only a few features, and in each boosting update, the term which minimizes a least squares approximation to the negative gradient of the empirical risk is added to the ensemble. In contrast, in our approach, the tree and the kernel machine depend on all features by default, base learners are learned using Newton's method or gradient descent, and we select the base learner whose addition to the ensemble directly results in the lowest empirical risk.

The idea that machine learning methods should be able learn both smooth as well as non-smooth functions has recently received attention also in other areas of machine learning. For instance, Imaizumi and Fukumizu [24] and Hayakawa and Suzuki [20] argue that one of the reasons for the superior predictive accuracy of deep neural networks, over e.g. kernel methods, is their ability to also learn non-smooth functions.

2 Preliminaries

2.1 Boosting

There exist population as well as sample versions of boosting algorithms. For the sake of brevity, we only consider the latter here. Assume that we have data $\{(x_i, y_i) \in \mathbb{R}^p \times \mathbb{R}, i = 1, \dots, n\}$ from a probability distribution $P_{X,Y}$. The goal of boosting is to find a function $F : \mathbb{R}^p \rightarrow \mathbb{R}$ for predicting y given x, where F is in a function space Ω_S with inner product $\langle \cdot, \cdot \rangle$ given by $\langle F, F \rangle = E_X \left(F(X)^2 \right)$, and the expectation is with respect to the marginal distribution P_X of $P_{X,Y}$. Note that y can be categorical, discrete, continuous, or of mixed type depending on whether the conditional distribution $P_{Y|X}$ is absolutely continuous with respect to the Lebesgue, a counting measure, or a mixture of the two; see, e.g., Sigrist and Hirnschall [42] for an example of the latter. Depending on the data and the goal of the application, the function can also be multivariate. For the sake of notational simplicity, we assume in the following that F is univariate. The extension to the multivariate case $F = (F^k)$, $k = 1, \dots, d$, is straightforward; see, e.g., Sigrist [41].

The goal of boosting is to find a minimizer $F^*(\cdot)$ of the empirical risk functional $R(F)$:

$$F^*(\cdot) = \underset{F(\cdot)\in\Omega_{\mathcal{S}}}{\operatorname{argmin}} R(F) = \underset{F(\cdot)\in\Omega_{\mathcal{S}}}{\operatorname{argmin}} \sum_{i=1}^{n} L(y_i, F(x_i)), \tag{1}$$

where $L(Y, F)$ is an appropriately chosen loss function such as the squared error for regression or the logistic regression loss for binary classification, and $\Omega_{\mathcal{S}} = span(\mathcal{S})$ is the span of a set of base learners $\mathcal{S} = \{f_j : \mathbb{R}^p \to \mathbb{R}\}$. Boosting finds $F^*(\cdot)$ in a sequential way by iteratively adding an update f_m to the current estimate F_{m-1}:

$$F_m(x) = F_{m-1}(x) + f_m(x), \quad f_m \in \mathcal{S}, \quad m = 1, \dots, M, \tag{2}$$

such that the empirical risk is minimized

$$f_m = \underset{f\in\mathcal{S}}{\operatorname{argmin}} R\left(F_{m-1} + f\right). \tag{3}$$

Since this usually cannot be found explicitly, one uses an approximate minimizer. Depending on whether gradient or Newton boosting is used, the update f_m is either obtained as the least squares approximation to the negative functional gradient or by applying one step of functional Newton's method which corresponds to minimizing a second order Taylor expansion of the risk functional; see Sect. 3 or Sigrist [41] for more information. For increased predictive accuracy [18], an additional shrinkage parameter $\nu > 0$ is usually added to the update equation:

$$F_m(x) = F_{m-1}(x) + \nu f_m(x). \tag{4}$$

2.2 Reproducing Kernel Hilbert Space Regression

Assume that $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is a positive definite kernel function. Then there exists a reproducing kernel Hilbert space (RKHS) $\mathcal{H}$ with an inner product $\langle\cdot,\cdot\rangle$ such that (i) the function $K(\cdot, x)$ belongs to $\mathcal{H}$ for all $x \in \mathbb{R}^d$ and (ii) $f(x) = \langle f, K(\cdot, x)\rangle$ for all $f \in \mathcal{H}$. Suppose we are interested in finding the minimizer

$$\underset{f\in\mathcal{H}}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - f(x_i))^2 + \lambda \|f\|_{\mathcal{H}}^2, \tag{5}$$

where $\lambda \geq 0$ is a regularization parameter. The representer theorem [40] then states that there is a unique minimizer of the form

$$f(\cdot) = \sum_{j=1}^{n} \alpha_j K(x_j, \cdot)$$

and (5) can be written as

$$\underset{\alpha\in\mathbb{R}^n}{\operatorname{argmin}} \|y - K\alpha\|^2 + \lambda\alpha^T K\alpha,$$

where $y = (y_1, \dots, y_n)^T$, $K \in \mathbb{R}^{n\times n}$ with $K_{ij} = K(x_i, x_j)$, and $\alpha = (\alpha_1, \dots, \alpha_n)^T$. Taking derivatives and equaling them to zero, we find the explicit solution as

$$\alpha = (K + \lambda I_n)^{-1} y,$$

where I_n denotes the n-dimensional identity matrix.

There is a close connection between Gaussian process regression and kernel regression. The solution to (5) is the posterior mean conditional on the data of a zero-mean Gaussian process with covariance function K. Further, since

$$f(x) = k(x)^T (K + \lambda I_n)^{-1} y,$$

where

$$k(x) = (K(x_1, x), \dots, K(x_n, x))^T, \tag{6}$$

kernel regression can also be interpreted as a two-layer neural network.

2.3 Regression Trees

We denote by $\mathcal{T}$ the space which consists of regression trees [7]. Following the notation used in Chen and Guestrin [11], a regression tree is given by

$$f^T(x) = w_{s(x)},$$

where $s : \mathbb{R}^p \rightarrow \{1, \dots, J\}$, $w \in \mathbb{R}^J$, and $J \in \mathbb{N}$ denotes the number of terminal nodes of the tree $f^T(x)$. s determines the structure of the tree, i.e., the partition of the space, and w denotes the leaf values. As in Breiman et al. [7], we assume that the partition of the space made by s is a binary tree where each cell in the partition is a rectangle of the form $R_j = (l_1, u_1] \times \cdots \times (l_p, u_p] \subset \mathbb{R}^p$ with $-\infty \leq l_m < u_m \leq \infty$ and $s(x) = j$ if $x \in R_j$.

3 Combined Kernel and Tree Boosting

Let $R^2(F_{m-1} + f)$ denote the functional, which is proportional to a second order Taylor approximation of the empirical risk in (1) at the current estimate F_{m-1}:

$$R^2(F_{m-1} + f) = \sum_{i=1}^{n} g_{m,i} f(x_i) + \frac{1}{2} h_{m,i} f(x_i)^2, \tag{7}$$

where $g_{m,i}$ and $h_{m,i}$ are the functional gradient and Hessian of the empirical risk evaluated at the functions $F_{m-1}(x)$ and $I_{\{x=x_i\}}(x)$, where $I_{\{x=x_i\}}(x) = 1$ if $x = x_i$ and 0 otherwise:

$$\begin{aligned} g_{m,i} &= \frac{\partial}{\partial F} L(y_i, F)\Big|_{F=F_{m-1}(x_i)}, \\ h_{m,i} &= \frac{\partial^2}{\partial F^2} L(y_i, F)\Big|_{F=F_{m-1}(x_i)}. \end{aligned} \tag{8}$$

The KTBoost algorithm presented in Algorithm 1 works as follows. In each boosting iteration, a candidate tree $f_m^T(x)$ and RKHS function $f_m^K(x)$ are found as minimizers of the second order Taylor approximation $R^2(F_{m-1} + f)$. This corresponds to applying one step of a functional version of Newton's method. It can be shown that candidate trees $f_m^T(x)$ can be found as weighted least squares minimizers; see, e.g., Chen and Guestrin [11] or Sigrist [41]. Further, the candidate penalized RKHS regression functions $f_m^K(x)$ can be found as shown in Proposition 1 below. The KTBoost algorithm then selects either the tree or the RKHS function such that the addition of the base learner to the ensemble according to Eq. (4) results in the lowest risk. Note that for the RKHS boosting part, the update equation $F_m(x) = F_{m-1}(x) + \nu f_m(x)$ can be replaced by simply updating the coefficients α_m.

Algorithm 1: KTBoost

```
Initialize $F_0(x) = \text{argmin}_{c \in \mathbb{R}^d} R(c)$.
for $m = 1$ to $M$ do
    Compute the gradient $g_{m,i}$ and Hessian $h_{m,i}$ as defined in (8)
    Find the candidate regression tree $f_m^T(x)$ and RKHS function $f_m^K(x)$
        $f_m^T(x) = \text{argmin}_{f \in \mathcal{T}} R^2(F_{m-1} + f)$
        $f_m^K(x) = \text{argmin}_{f \in \mathcal{H}} R^2(F_{m-1} + f) + \frac{1}{2}\lambda \|f\|_{\mathcal{H}}^2$
    where the approximate risk $R^2(F_{m-1} + f)$ is defined in (7)
    if $R\left(F_{m-1} + \nu f_m^T(x)\right) \leq R\left(F_{m-1} + \nu f_m^K(x)\right)$ then
        $f_m(x) = f_m^T(x)$
    else
        $f_m(x) = f_m^K(x)$
    end if
    Update $F_m(x) = F_{m-1}(x) + \nu f_m(x)$
end for
```

If either the loss function is not twice differentiable in its second argument or the second derivative is zero or constant on a non-null set of the support of X, one can alternatively use gradient boosting. The gradient boosting version of KTBoost is obtained as a special case of the Algorithm 1 by setting $h_{m,i} = 1$. Gradient boosting has the advantage that it is computationally less expensive than Newton boosting since, in contrast to (9), the kernel matrix does not depend on the iteration number m; see Sect. 3.1 for more details.

Proposition 1 *The kernel ridge regression solution $f_m^K(x)$ in the regularized Newton boosting update step is given by $f_m^K(x) = k(x)^T \alpha_m$, where $k(x)$ is defined in (6) and*

$$\alpha_m = D_m \left(D_m K D_m + \lambda I_n\right)^{-1} D_m y_m, \tag{9}$$

where $D_m = diag\left(\sqrt{h_{m,i}}\right)$, $h_{m,i} > 0$, $y_m = (-g_{m,1}/h_{m,1}, \ldots, -g_{m,n}/h_{m,n})^T$, and I_n is the identity matrix of dimension n.

Proof We have

$$\begin{aligned}
&\underset{f \in \mathcal{H}}{\text{argmin}} \sum_{i=1}^{n} g_{m,i} f(x_i) + \frac{1}{2} h_{m,i} f(x_i)^2 + \frac{1}{2} \lambda \|f\|_{\mathcal{H}}^2 \\
&= \underset{f \in \mathcal{H}}{\text{argmin}} \sum_{i=1}^{n} h_{m,i} \left(-\frac{g_{m,i}}{h_{m,i}} - f(x_i)\right)^2 + \lambda \|f\|_{\mathcal{H}}^2 \\
&= \underset{\alpha}{\text{argmin}} \|D_m y_m - D_m K \alpha\|^2 + \lambda \alpha^T K \alpha.
\end{aligned}$$

If we take derivatives with respect to α, equal them to zero, and solve for α, we find that

$$\begin{aligned}
\alpha_m &= \left(K D_m^2 K + \lambda K\right)^{-1} K D_m^2 y_m \\
&= \left(D_m^2 K + \lambda I_n\right)^{-1} D_m^2 y_m \\
&= D_m \left(D_m K D_m + \lambda I_n\right)^{-1} D_m y_m.
\end{aligned}$$

□

3.1 Reducing Computational Costs for Large Data

Concerning the regression trees, finding the splits when growing the trees is the computationally demanding part. There are several approaches in the literature on how this can be done efficiently for large data; see, e.g., Chen and Guestrin [11] or Ke et al. [26]. The computationally expensive part for finding the kernel regression updates is the factorization of the kernel matrix which scales with $O(n^3)$ in time. There are several approaches that allow for computational efficiency in the large data case. Examples of this include low rank approximations based on, e.g., the Nyström method [43] and extensions of it such as divide-and-conquer kernel ridge regression [47,48], early stopping of iterative optimization methods [6,27,38,45], stochastic gradient descent [10,12], random feature approximations [37], and compactly supported kernel functions [5,19] which results in a sparse kernel matrix K which can be efficiently factorized.

Note that if gradient descent is used instead of Newton's method, the RKHS function $f_m^K(x)$ can be found efficiently by observing that, in contrast to (9), the kernel matrix $K+\lambda I_n$ does not depend on the iteration number m, i.e., its inverse or a Cholesky factor of it needs to be calculated only once. Further, the two learners can be learned in parallel.

In our empirical analysis, we use the Nyström method for dealing with large data sets. The Nyström method approximates the kernel $K(\cdot,\cdot)$ by first choosing a set of l so-called Nyström samples $x_1^*,\dots,x_l^*$. Often these are obtained by sampling uniformly from the data. Denoting the kernel matrix that corresponds to these points as K^*, the Nyström method then approximates the kernel $K(\cdot,\cdot)$ as

$$K(x,y) \approx k_l(x)^T {K_{l,l}^*}^{-1} k_l(y),$$

where $k_l(x) = (K(x,x_1^*),\dots,K(x,x_l^*))^T$ and $\left(K_{l,l}^*\right)_{j,k} = K(x_j^*,x_k^*)$, $1 \le j,k \le l$. In particular, the reduced-rank Nyström approximation to the full kernel matrix K is given by

$$K \approx K_{n,l}^* {K_{l,l}^*}^{-1} {K_{n,l}^*}^T,$$

where $\left(K_{n,l}^*\right)_{j,k} = K(x_j,x_k^*)$, $1 \le j \le n$, $1 \le k \le l$.

4 Experimental Results

4.1 Simulation Study

We first conduct a small simulation study to illustrate that the combination of discontinuous trees and continuous kernel machines can indeed better learn functions with both discontinuous and smooth parts. We consider random functions $F:[0,1]\rightarrow\mathbb{R}$ with five random jumps in $[0,0.5]$:

$$\begin{gathered} F(x) = \sum_{i=1}^{5} g_i \mathbf{1}_{(t_i,1]}(x) + \sin(8\pi x), \\ t_i \overset{\text{iid}}{\sim} \text{Unif}(0,0.5), \quad g_i \overset{\text{iid}}{\sim} \text{Unif}(0,5) \end{gathered} \tag{10}$$

and data according to

$$y_i = F(x_i) + N(0,0.25^2), \quad x_i \overset{\text{iid}}{\sim} \text{Unif}(0,1), \quad i=1,\dots,1000.$$

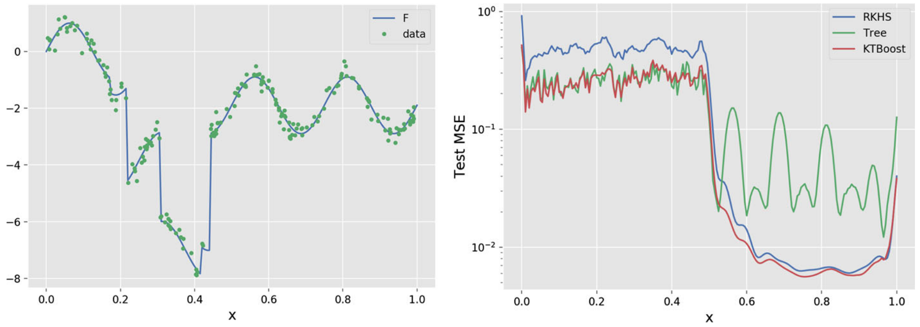

Fig. 2 An example of a random function with five random jumps in [0, 0.5] and corresponding observed data (left plot) and pointwise mean square error (MSE) for tree and kernel boosting as well as the combined KTBoost algorithm (right plot)

In Fig. 2 on the left-hand side, an example of such a function and corresponding data is shown. We simulate 1000 times such random functions as well as training, validation, and test data of size $n = 1000$. For each simulation run, learning is done on the training data. The number of boosting iterations is chosen on the validation data with the maximum number of boosting iterations being $M = 1000$. We use a learning rate of $\nu = 0.1$ as this is a reasonable default value [8] and trees of depth = 1 as there are no interactions. Further, for the RKHS ridge regression, we use a Gaussian kernel

$$K(x_1, x_2) = \exp\left(-\|x_1 - x_2\|^2/\rho^2\right), \tag{11}$$

with $\rho = 0.1$ and $\lambda = 1$. In Fig. 2 on the right-hand side, we show the pointwise test mean square error (MSE) for tree and kernel boosting as well as the combined KTBoost algorithm. We observe that tree boosting performs better than kernel boosting in the area where the discontinuities are located and, conversely, kernel boosting outperforms tree boosting on the smooth part. The figure also clearly shows that KTBoost outperforms both tree and kernel boosting as it achieves the MSE of tree boosting on the interval with jumps and the MSE of kernel boosting on the smooth part.

For the purpose of illustration, we have considered a one-dimensional example. However, in practice discontinuities, or strong non-linearities, as well as smooth parts are likely to occur at the interaction level in higher dimensions of a feature space.

4.2 Real-World Data

In the following, we compare the KTBoost algorithm with tree and kernel boosting using the following Delve, Keel, Kaggle, and UCI data sets: abalone, ailerons, bank8FM, elevators, energy, housing, liberty, NavalT, parkinsons, puma32h, sarcos, wine, adult, cancer, ijcnn, ionosphere, sonar, car, epileptic, glass, and satimage. Detailed information on the number of samples and features can be found in Table 1. We consider both regression as well as binary and multiclass classification data sets. Further, we include data sets of different sizes in order to investigate the performance on both smaller and larger data sets, as small- to moderately-sized data sets continue to be widely used in applied data science despite the recent focus on very large data sets in machine learning research. We use the squared loss for regression, the logistic regression loss for binary classification, and the cross-entropy loss with the softmax function for multiclass classification.

Table 1 Summary of data sets

Data	# classes	Nb. samples	Nb. features
abalone	Regression	4177	10
ailerons	Regression	13,750	40
bank8FM	Regression	8192	8
elevators	Regression	16,599	18
energy	Regression	768	8
housing	Regression	506	13
liberty	Regression	50,999	117
NavalT	Regression	11,934	16
parkinsons	Regression	5875	16
puma32h	Regression	8192	32
sarcos	Regression	48,933	21
wine	Regression	4898	11
adult	2	48,842	108
cancer	2	699	9
ijcnn	2	141,691	22
ionosphere	2	351	34
sonar	2	208	60
car	4	1728	21
epileptic	5	11,500	178
glass	7	214	9
satimage	6	6438	36

For the regression data sets, we use gradient boosting, and for the classification data sets, we use boosting with Newton updates since this can result in more accurate predictions [41]. For some classification data sets (adult, ijcnn, epileptic, and satimage), Newton boosting is computationally infeasible on a standard single CPU computer with the current implementation of KTBoost, despite the use of the Nyström method with a reasonable number of Nyström samples, say 1000, since the weighted kernel matrix in Eq. (9) needs to be factorized in every iteration. We thus also use gradient boosting for these data sets. Technically, it would be possible for these cases to learn the trees using Newton's method, or using the hybrid gradient-Newton boosting version of Friedman [18], but this would result in an unfair comparison that is biased in favor of the base learner which is learned with the better optimization method. For the larger data sets (liberty, sarcos, adult, ijcnn), we use the Nyström method described in Sect. 3.1. Specifically, we use $l = 1000$ Nyström samples, which are uniformly sampled from the training data. In general, the larger the number of Nyström samples, the lower the approximation error but the higher the computational costs. Williams and Seeger [43] reports good results with $l \approx 1000$ for several data sets. All calculations are done with the Python package `KTBoost` on a standard laptop with a 2.9 GHz quad-core processor and 16 GB of RAM.

All data sets are randomly split into three non-overlapping parts of equal size to obtain training, validation and test sets. Learning is done on the training data, tuning parameters are chosen on the validation data, and model comparison is done on the holdout test data. All input features are standardized using the training data to have approximately mean zero and variance one. In order to measure the generalization error and approximately quantify variability in it, we use ten different random splits of the data into training, validation and test sets. We note that when using a resampling approach, standard statistical tests, such as a paired t-test, cannot be used to do a pairwise comparison of the different algorithms on a dataset basis since training and test datasets in different splits are dependent due to overlap [3,13,14]. In particular, this can result in biased standard error estimates for the generalization error.

For the RKHS ridge regression, we use again a Gaussian kernel; see Eq. (11). Concerning tuning parameters, we select the number of boosting iterations M from $\{1, 2, \dots, 1000\}$, the learning rate ν from $\{1, 10^{-1}, 10^{-2}, 10^{-3}\}$, the maximal depth of the trees from $\{1, 5, 10\}$, and the kernel ridge regularization parameter λ from $\{1, 10\}$. Further, the kernel range parameter ρ is chosen using k-nearest neighbors distances as described in the following. We first calculate the average distance of all k-nearest neighbors in the training data, where k is a tuning parameter selected from $\{5, 50, 500, 5000, n - 1\}$ and n is the size of the training data. We then choose ρ such that the kernel function has decayed to a value of 0.01 at this average k-nearest neighbors distance. This is motivated by the fact that for a corresponding Gaussian process with such a covariance function, the correlation has decayed to a level of 1% at this k-nearest neighbor distance. If the training data contains less than 5000 (or 500) samples, we use $n - 1$ as the maximal number for the k-nearest neighbors. In addition, we include ρ which equals the average $(n-1)$-nearest neighbor distance. The latter choice is done in order to also include a range which results in a kernel that decays slowly over the entire space. For the large data sets where the Nyström method is used, we calculate the average k-nearest neighbors distance based on the Nyström samples. I.e., in this case, the maximal k equals $l - 1$.

The results are shown in Table 2. For the regression data sets, we show the average test mean square error (MSE) over the different sample splits, and for the classification data sets, we calculate the average test error rate (=misclassification rate). The numbers in parentheses are approximate standard deviations over the different sample splits. In the last row, we report the average rank of every method over the different data sets. We find that KTBoost achieves higher predictive accuracy than both tree and kernel boosting for the large majority of data sets. Specifically, KTBoost has an average rank of 1.24 and achieves higher predictive accuracy than both tree and kernel boosting for seventeen out of twenty-one data sets. A Friedman test with an Iman and Davenport correction [25] gives a p value of 7.84×10^{-6} which shows that the differences in the three methods are highly significant. We next assess whether the pairwise differences in accuracy between the different methods are statistically significant using a sign test. Further, we apply a Holm–Bonferroni correction [21] to account for the fact that we do multiple tests. Despite the sign test having low power and the application of the conservative Holm–Bonferroni correction, KTBoost is highly significantly better than both tree and kernel boosting with adjusted p values below 0.01. The difference between kernel and tree boosting is not significant with both the adjusted and non-adjusted p values being above 0.1 (result not tabulated).

Note that we do not report the optimal tuning parameters since this is infeasible for all combinations of data sets and sample splits, and aggregate values are not meaningful since different tuning parameters often compensate each other in a non-linear way (e.g., number of iterations, learning rate, and tree depth or kernel regularization λ). Further, it is also difficult to concisely summarize the composition of the ensembles in terms of different base learners as a base learner that is added in an earlier boosting stage is more important than one that is added in a later stage [9], and the properties of the base learners also depend on the chosen tuning parameters. We also note that one can also consider additional tuning parameters. For trees, this includes the minimal number of samples per leaf, row and column sub-sampling,

Table 2 Comparison of KTBoost with tree and kernel boosting using test mean square error (regression) and test error rate (classification)

Data	KTBoost	Tree	Kernel
abalone	4.65 (0.248)	5.07 (0.261)	**4.64** (0.255)
ailerons	**2.64e−08** (6.19e−10)	8.11e−08 (2.39e−09)	2.64e−08 (6.19e−10)
bank8FM	**0.000915** (4.02e−05)	0.000945 (2.47e−05)	0.000945 (5.83e−05)
elevators	**4.83e−06** (2.9e−07)	5.66e−06 (1.44e−07)	5.18e−06 (3.89e−07)
energy	**0.282** (0.0372)	0.335 (0.093)	1.3 (0.377)
housing	**12.7** (3.19)	15.1 (3.23)	13.6 (2.51)
liberty	**14.5** (0.323)	14.5 (0.314)	15.2 (0.345)
NavalT	**6.51e−09** (1.15e−09)	1.15e−06 (1.58e−07)	6.51e−09 (1.15e−09)
parkinsons	**73.3** (1.98)	81.1 (2.44)	73.3 (1.91)
puma32h	**6.5e−05** (2.27e−06)	6.51e−05 (2.13e−06)	0.000695 (2.2e−05)
sarcos	**7.99** (0.206)	9.6 (0.207)	17.8 (0.586)
wine	**0.444** (0.012)	0.471 (0.0169)	0.506 (0.0106)
adult	0.128 (0.00295)	**0.128** (0.00313)	0.163 (0.00512)
cancer	0.0362 (0.00744)	0.0415 (0.0153)	**0.0358** (0.0107)
ijcnn	**0.0122** (0.000685)	0.0123 (0.000702)	0.0387 (0.00516)
ionosphere	**0.0872** (0.017)	0.103 (0.0226)	0.107 (0.0239)
sonar	0.194 (0.0394)	0.223 (0.05)	**0.193** (0.0491)
car	**0.0399** (0.00505)	0.0411 (0.00685)	0.041 (0.00624)
epileptic	**0.354** (0.00612)	0.373 (0.00614)	0.442 (0.0265)
glass	**0.308** (0.0711)	0.315 (0.0589)	0.344 (0.0581)
satimage	**0.089** (0.00452)	0.112 (0.00504)	0.0903 (0.00417)
Average rank	1.24	2.48	2.29
p val Friedman test	7.84e−06		
Adj. *p* val sign test		6.29e−05	0.00885

The smallest value are in boldface. In parentheses are approximate standard deviations. Below are average ranks of the methods over the different datasets. A *p* value of a Friedman test with an Iman and Davenport correction for comparing the different algorithms is also reported. The last row shows Holm–Bonferroni corrected *p* values of sign tests for pairwise comparison of the KTBoost algorithm with tree and kernel boosting

and penalization of leave values, and for the kernel regression, this includes the smoothness of the kernel function, or, in general, the class of kernel functions. One could also use different learning rates for the two types of base learners. Due to limits on computational costs, we have not considered all possible choices and combinations of tuning parameters. However, it is likely that a potential increase in predictive performance in either tree or kernel boosting will also result in an increase in accuracy of the combined KTBoost algorithm. We also note that in our experimental setup, the tuning parameter grid for the KTBoost algorithm is larger compared to the tree and kernel boosting cases. This seems inevitable in order to allow for the fairest possible comparison, though. Restricting one type of tuning parameters for the combined version but not for the single base learner case seems to be no alternative. Somewhat alleviating this concern is the fact that, in the above simulation study, we also find outperformance when not choosing tuning parameters using cross-validation, and on the downside, a larger tuning parameter grid might potentially also lead to overfitting. Finally, we remark that we have also considered to compare the risk of the un-damped base learners

$$R\left(F_{m-1} + f_m^T(x)\right) \leq R\left(F_{m-1} + f_m^K(x)\right)$$

in line 5 of Algorithm 1 when selecting the base learners that is added to the ensemble, and we obtain very similar results (see supplementary material).

5 Conclusions

We have introduced a novel boosting algorithm, which combines trees and RKHS functions as base learners. Intuitively, the idea is that discontinuous trees and continuous RKHS functions complement each other since trees are better suited for learning rougher parts of functions and RKHS regression functions can better learn smoother parts of functions. We have compared the predictive accuracy of the KTBoost algorithm with tree and kernel boosting and have found that KTBoost achieves significantly higher predictive accuracy compared to tree and kernel boosting.

Future research can be done in several directions. First, it would be interesting to investigate to which extent other base learners such as neural networks [23,31] are useful in addition to trees and kernel regression functions. Generalizing the KTBoost algorithm using reproducing kernel Kreĭn space (RKKS) learners [32,33] instead of RKHS learners can also be investigated. Further, theoretical results such as learning rates or bounds on the risk could help to shed further insights on why the combination of trees and kernel machines leads to increased predictive accuracy. Finally, it would be interesting to compare the KTBoost algorithm on very large data sets using different strategies for reducing the computational complexity of the RKHS part. Several potential strategies on how this can be done are briefly outlined in Sect. 3.1.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s11063-021-10434-9.

Acknowledgements We would like to thank Christoph Hirnschall for helpful feedback on the article. This research was partially supported by the Swiss Innovation Agency-Innosuisse (25746.1 PFES-ES).

Funding Open Access funding provided by Hochschule Luzern

References

1. Belkin M, Hsu DJ, Mitra P (2018a) Overfitting or perfect fitting? Risk bounds for classification and regression rules that interpolate. In: Bengio S, Wallach H, Larochelle H, Grauman K, Cesa-Bianchi N, Garnett R (eds) Advances in neural information processing systems, vol 31. pp 2306–2317

2. Belkin M, Ma S, Mandal S (2018b) To understand deep learning we need to understand kernel learning. In: Dy J, Krause A (eds) Proceedings of the 35th international conference on machine learning, volume 80 of proceedings of machine learning research. pp 541–549
3. Bengio Y, Grandvalet Y (2004) No unbiased estimator of the variance of k-fold cross-validation. J Mach Learn Res 5:1089–1105
4. Berlinet A, Thomas-Agnan C (2011) Reproducing kernel Hilbert spaces in probability and statistics. Springer, Berlin
5. Bevilacqua M, Faouzi T, Furrer R, Porcu E et al (2019) Estimation and prediction using generalized Wendland covariance functions under fixed domain asymptotics. Ann Stat 47(2):828–856
6. Blanchard G, Krämer N (2010) Optimal learning rates for kernel conjugate gradient regression. In:Advances in neural information processing systems. pp 226–234
7. Breiman L, Friedman J, Stone CJ, Olshen RA (1984) Classification and regression trees. CRC Press, Boca Raton
8. Bühlmann P, Hothorn T (2007) Boosting algorithms: Regularization, prediction and model fitting. Stat Sci 22:477–505
9. Bühlmann P, Yu B (2003) Boosting with the l 2 loss: regression and classification. J Am Stat Ass 98(462):324–339
10. Cesa-Bianchi N, Conconi A, Gentile C (2004) On the generalization ability of on-line learning algorithms. IEEE Trans Inf Theory 50(9):2050–2057
11. Chen T, Guestrin C (2016) Xgboost: A scalable tree boosting system. In: Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. ACM, pp 785–794
12. Dai B, Xie B, He N, Liang Y, Raj A, Balcan M-FF, Song L (2014) Scalable kernel methods via doubly stochastic gradients. In: Advances in neural information processing systems. pp 3041–3049
13. Demšar J (2006) Statistical comparisons of classifiers over multiple data sets. J Mach Learn Res 7:1–30
14. Dietterich TG (1998) Approximate statistical tests for comparing supervised classification learning algorithms. Neural Comput 10(7):1895–1923
15. Freund Y, Schapire RE (1996) Experiments with a new boosting algorithm. In: ICML, vol 96. Bari, Italy, pp 148–156
16. Freund Y, Schapire RE (1997) A decision-theoretic generalization of on-line learning and an application to boosting. J Comput Syst Sci 55(1):119–139
17. Friedman J, Hastie T, Tibshirani R et al (2000) Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). Ann Stat 28(2):337–407
18. Friedman JH (2001) Greedy function approximation: a gradient boosting machine. Ann Stat 29:1189–1232
19. Gneiting T (2002) Compactly supported correlation functions. J Multivar Anal 83(2):493–508
20. Hayakawa S, Suzuki T (2020) On the minimax optimality and superiority of deep neural network learning over sparse parameter spaces. Neural Netw 123:343–361
21. Holm S (1979) A simple sequentially rejective multiple test procedure. Scand J Stat 6:65–70
22. Hothorn T, Bühlmann P, Kneib T, Schmid M, Hofner B (2010) Model-based boosting 2.0. J Mach Learn Res 11:2109–2113
23. Huang F, Ash J, Langford J, Schapire R (2018) Learning deep resnet blocks sequentially using boosting theory. ICML 80:2058–2067
24. Imaizumi M, Fukumizu K (2019) Deep neural networks learn non-smooth functions effectively. In:The 22nd international conference on artificial intelligence and statistics. pp 869–878
25. Iman RL, Davenport JM (1980) Approximations of the critical region of the fbietkan statistic. Commun Stat Theory Methods 9(6):571–595
26. Ke G, Meng Q, Finley T, Wang T, Chen W, Ma W, Ye Q, Liu T-Y (2017) Lightgbm: a highly efficient gradient boosting decision tree. In: Advances in neural information processing systems. pp 3149–3157
27. Ma S, Belkin M (2017) Diving into the shallows: a computational perspective on large-scale shallow learning. In: Advances in neural information processing systems. pp 3778–3787
28. Mason L, Baxter J, Bartlett PL, Frean MR (2000) Boosting algorithms as gradient descent. In: Advances in neural information processing systems. pp 512–518
29. Mendes-Moreira J, Soares C, Jorge AM, Sousa JFD (2012) Ensemble approaches for regression: a survey. ACM Comput Surv (CSUR) 45(1):10
30. Murphy KP (2012) Machine learning: a probabilistic perspective. The MIT Press. ISBN 0262018020, 9780262018029
31. Nitanda A, Suzuki T (2018) Functional gradient boosting based on residual network perception. ICML 80:3819–3828
32. Oglic D, Gaertner T (2018) Learning in reproducing kernel kreĭ spaces. In: International conference on machine learning. pp 3859–3867

33. Ong CS, Mary X, Canu S, Smola AJ (2004) Learning with non-positive kernels. In: Proceedings of the twenty-first international conference on machine learning. pp 81
34. Peng J, Aved AJ, Seetharaman G, Palaniappan K (2018) Multiview boosting with information propagation for classification. IEEE transactions on neural networks and learning systems 29(3):657–669
35. Ponomareva N, Radpour S, Hendry G, Haykal S, Colthurst T, Mitrichev P, Grushetsky A (2017) Tf boosted trees: a scalable tensorflow based framework for gradient boosting. In: Joint European conference on machine learning and knowledge discovery in databases. Springer, pp 423–427
36. Prokhorenkova L, Gusev G, Vorobev A, Dorogush AV, Gulin A (2018) Catboost: unbiased boosting with categorical features. In: Advances in neural information processing systems vol 31. Curran Associates, Inc, pp 6638–6648
37. Rahimi A, Recht B (2008) Random features for large-scale kernel machines. In: Advances in neural information processing systems. pp 1177–1184
38. Raskutti G, Wainwright MJ, Yu B (2014) Early stopping and non-parametric regression: an optimal data-dependent stopping rule. J Mach Learn Res 15(1):335–366
39. Schölkopf B, Smola AJ (2001) Learning with kernels: support vector machines, regularization, optimization, and beyond. MIT Press, Cambridge
40. Schölkopf B, Herbrich R, Smola AJ (2001) A generalized representer theorem. In: International conference on computational learning theory. Springer, pp 416–426
41. Sigrist F (2021) Gradient and newton boosting for classification and regression. Expert Syst Appl (in press)
42. Sigrist F, Hirnschall C (2019) Grabit: Gradient tree-boosted tobit models for default prediction. J Bank Finance 102:177–192
43. Williams CK, Seeger M (2001) Using the Nyström method to speed up kernel machines. In: Advances in neural information processing systems. pp 682–688
44. Wyner AJ, Olson M, Bleich J, Mease D (2017) Explaining the success of adaboost and random forests as interpolating classifiers. J Mach Learn Res 18(48):1–33
45. Yao Y, Rosasco L, Caponnetto A (2007) On early stopping in gradient descent learning. Constr Approx 26(2):289–315
46. Zhang C, Bengio S, Hardt M, Recht B, Vinyals O (2017) Understanding deep learning requires rethinking generalization. In: International conference on learning representations
47. Zhang Y, Duchi J, Wainwright M (2013) Divide and conquer kernel ridge regression. In: Conference on learning theory. pp 592–617
48. Zhang Y, Duchi J, Wainwright M (2015) Divide and conquer kernel ridge regression: a distributed algorithm with minimax optimal rates. J Mach Learn Res 16(1):3299–3340

On Boosting: Theory and Applications

Andrea Ferrario[*], Roger Hämmerli[†]

Prepared for:
Fachgruppe "Data Science"
Swiss Association of Actuaries SAV

Version of June 11, 2019

Abstract

We provide an overview of two commonly used boosting methodologies. We start with the description of different implementations as well as the statistical theory behind selected algorithms which are widely used by the machine learning community, then we discuss a case study focusing on the prediction of car insurance claims in a fixed future time interval. The results of the case study show that, overall, `XGBoost` performs better than `AdaBoost` and it shows best performance when shallow trees, moderate shrinking, the number of iterations increased with respect to default as well as subsampling of both features and training data points are considered.

Keywords. machine learning, boosting, predictive modeling, R, Python, car insurance, Kaggle, Porto Seguro, `AdaBoost`, `XGBoost`.

0 Introduction and overview

This data analytics tutorial has been written for the working group "Data Science" of the Swiss Association of Actuaries SAV, see

`https://www.actuarialdatascience.org/`

The main purpose of this tutorial is to provide an overview of the two most used boosting algorithms, i.e. `AdaBoost` and `XGBoost`. We start with introducing the basic approach of these two algorithms and show various implementations of them. We apply the algorithms to an insurance case study which uses data from the Porto Seguro's Safe Driver Prediction competition[1] hosted on Kaggle's platform[2]. For more statistically oriented literature on machine learning—and in particular on boosting—we refer to the monographs [31], [17], [19] and [5]. A list of references given at the end of this paper can be used to deep dive into the details of boosting.

[*] Mobiliar Lab for Analytics at ETH and Department of Management, Technology and Economics, ETH Zurich, aferrario@ethz.ch

[†] Schweizerische Mobiliar Versicherungsgesellschaft, roger.haemmerli@mobiliar.ch

[1] `https://www.kaggle.com/c/porto-seguro-safe-driver-prediction`

[2] `https://www.kaggle.com/`

1 Boosting: a gentle introduction

Boosting algorithms are so called machine learning ensemble methods, i.e. algorithmic procedures which combine weak learners into a single, high performing one. In [20] and [21] the authors posed the question whether a set of weak classifiers could be converted into a 'strong' one. A positive answer was given by Schapire [30]. Later in the 90s, the first examples of boosting algorithms were introduced. Nowadays boosting has empirically proven its effectiveness in a vast array of classification and regression problems; for example, on the data science competition platform Kaggle, among 29 challenge winning solutions in 2015, 17 used `XGBoost`, a boosting algorithm introduced by Chen and Guestrin in [6]. In the KDD Cup[3] 2015, all top-10 winning team used `XGBoost`[4]. Other gradient tree boosting methods have also shown to give state-of-the-art results on many standard classification benchmarks. In this tutorial we will focus on two prominent examples of boosting algorithms: `AdaBoost` and `XGBoost`. For a general overview of boosting we refer to the monograph [31] and the book [17]; we invite the interested reader to go back to the original literature for each and all the presented algorithms.

2 Boosting algorithms: `AdaBoost`

`AdaBoost` is a boosting algorithm that produces a single ensemble learner[5] from a sequential additive process which involves re-weighting of training data points (therefore the name `AdaBoost`, or 'Adaptive Boosting' of weights) and fitting of weak learners (also named as 'base procedures' in [4]). To produce predictions, the final learner takes as input a linear combination of the predictions from the ensemble of weak learners. The algorithm has been introduced originally in the context of binary classification by Freund and Schapire[6] (in [13] and [14]), and has subsequently been adapted to regression problems. Its success in delivering accurate ensembles and its resistance to overfitting led Breiman to call `AdaBoost` the 'best off-the-shelf classifier in the world' (NIPS Workshop 1996). We refer to [2] for additional considerations on this point.

The structure of this section is as follows: we introduce some notation for the statistical learning problem of relevance for this tutorial, we provide a high-level description of the `AdaBoost` algorithm and we give an overview of main `AdaBoost` formulations from the scientific literature. We discuss the statistical learning theory behind adaptive boosting and we close the section with an overview of `AdaBoost` implementations in both Python and `R`.

2.1 Statistical learning and `AdaBoost`: notation

In this tutorial we consider the statistical learning problem to classify or label data points using machine learning models. We start by considering a data set $\mathcal{D} = \{(Y_1, \mathbf{x}_1), \ldots, (Y_n, \mathbf{x}_n)\}$, with responses $Y_i \in \mathcal{Y}$, and $\mathbf{x}_i \in \mathcal{X}$ for all $i = 1, \ldots, n$.

[3] The KDD (Knowledge Discovery and Data Mining) Cup is a yearly competition hosted by the SIGKDD (`https://www.kdd.org/`), which is the Association for Computing Machinery's (ACM) Special Interest Group (SIG) on Knowledge Discovery and Data Mining.

[4] `https://www.linkedin.com/pulse/present-future-kdd-cup-competition-outsiders-ron-bekkerman/`

[5] In this tutorial, the terms *learner, classifier, hypothesis* are used interchangeably.

[6] An extended abstract of [14] appeared in Freund, Y., Schapire, R.E. (1995). A decision-theoretic generalization of on-line learning and an application to boosting. Proceedings of the Second European Conference on Computational Learning Theory.

① 线性函数 (仿射函数).

② · · · ·

In the notation of [32], Chapter 2, $\mathcal{X}$ and $\mathcal{Y}$[7] are the domain set and the label (finite) set, respectively, of the statistical learning framework at hand, while $\mathcal{D}$ denotes the set of data sampled from $\mathcal{Y} \times \mathcal{X}$. 一堆函数的集合. ? 思考一下为什么叫这个名字.
With $\mathcal{H} = \{h \mid h : \mathcal{X} \longrightarrow \mathcal{Y}\}$ we denote a given set of hypotheses; in the literature it is commonly referred to as the hypothesis class. Heuristically, the set $\mathcal{H}$ has to be rich enough to allow for learning complex decisions, but not excessively large to avoid overfitting. In this tutorial, we will consider the set $\mathcal{H}$ of finite classification trees.
Let M denote a positive integer; given a training data set $\mathcal{D}$, a set of classifiers $\mathcal{H}$ and the number of steps M, in its most general formulation `AdaBoost` computes weak learners h_m as well as real coefficients through re-weighting of training data, and returns a strong classifier by considering a majority weighted-rule, which depends, among others, on the label space $\mathcal{Y}$. At each step $m = 1, \dots M$, the algorithm performs the fitting of a weak classifier and the computation of the real coefficients relying only on results from the previous step, i.e. $m - 1$. Two distinct approaches to re-weighting of data points and subsequent fitting of classifiers in $\mathcal{H}$ are presented in the literature (e.g. in [31], [11]). With *boosting by resampling* one would sample the original training data using an adaptive empirical distribution provided in the `AdaBoost` algorithm formulation and feed the sampled unweighted data to the weak learners h_m's at each step m. On the other hand, with *boosting by re-weighting*, one assumes that the elements of $\mathcal{H}$ allow us to perform weighted fitting through modification of their base algorithm, instead. This is the case for decision trees as base learners; boosting by re-weighting is the re-weighting approach considered in this tutorial.

2.2 On `AdaBoost`: an *excursus* in the literature

Since its original formulation by Freund and Schapire [14], multiple versions of adaptive boosting algorithms have appeared in the scientific literature. Therefore, in this section we introduce and compare the most relevant `AdaBoost` variants 变种 to provide the reader with a map to move through different adaptive boosting algorithm formulations both in the available literature and their implementations in software like Python and R. We believe that such an exposition could be beneficial as the actuarial application requires a detailed understanding of the algorithm and its limitations. We will briefly discuss:

- the original `AdaBoost` formulation from [14];
- the `AdaBoost` formulation from Schapire and Freund in [31];
- the adaptive boosting variant denoted by `AdaBoost.M1` in [14] and [13];
- the 'real', 'discrete' and 'gentle' adaptive boosting algorithms together with `LogitBoost` in [16]; and
- the SAMME.R and SAMME adaptive boosting algorithms in [36].

Additional adaptive boosting algorithm variants like `TotalBoost` [34], `BrownBoost` [12], `LPBoost` [7] and `SmoothBoost` [33] and those in [8] will not be considered in this tutorial, as well as cost-sensitive boosting in asymmetric statistical learning problems ([25], [24] and references therein, in particular those presented in Table 3.1, Chapter 3).

[7] Common choices in the literature are $\mathcal{Y} = \{-1, 1\}$, $\mathcal{Y} = \{0, 1\}$ or $\mathcal{Y} = \{1, 2, \dots, k\}, k \geq 2$ in the multi-class case.

这TM倒底是啥啊. $h_m \in \mathcal{H}$.

2.2.1 Original AdaBoost formulation from [14]

The original adaptive boosting algorithm AdaBoost has been introduced in [14], in the context of binary classification. The algorithm generates an ensemble classifier through sequential training of weak learners on weighted data; the weak learners, called 'hypotheses', are returned as the output of a given, yet unspecified, learning algorithm, denoted by **WeakLearn**, at each boosting iteration. Overall, the goal of each weak learner is to minimize the misclassification error on training weighted data; at each iteration both weights and the distribution over data are updated taking into account the performance of the weak learner at the previous iteration. Finally, AdaBoost returns a final hypothesis by combining the outputs of all weak hypotheses *via* a majority voting rule. In summary, AdaBoost is a meta-algorithm which uses a base algorithm to train weak learners on reweighted data sequentially and combines their contributions to produce a stronger classifier.

We provide the original AdaBoost from [14] in **Algorithm 1** with a slightly modified notation, to facilitate comparison with AdaBoost.M1 in [13]. In this section, the expressions *original* AdaBoost and **Algorithm 1** are used interchangeably.

```
Algorithm 1: Original AdaBoost algorithm [14] for binary classification
Input :
    • training data $\mathcal{D} = \{(Y_1, \mathbf{x}_1), \ldots, (Y_n, \mathbf{x}_n)\}$, with labels $Y_i \in \mathcal{Y} = \{0, 1\}$
    • weak learning algorithm WeakLearn          实际中这个Hypothesis是什么.
    • number of iterations $M \geq 1$
Initialize $D_1(i) = \frac{1}{n}$, for all $i = 1, \ldots, n$. 权重
while $m \in \{1, \ldots, M\}$ do
    Fit WeakLearn to re-weighted training set $(\mathcal{D}, D_m)$. 得到了一个 $h_1 \in \mathcal{H}$.
    Return a hypothesis $h_m : \mathcal{X} \to \mathcal{Y}$ and compute its weighted misclassification error
      $\epsilon_m = \sum_{i=1}^{n} D_m(i)|h_m(\mathbf{x}_i) - Y_i|$. $> 0$ .
    Set $\beta_m = \frac{\epsilon_m}{1-\epsilon_m}$. 单增. $\epsilon_m < \frac{1}{2}$ implies. $\beta_m < 1$. (都是误差的度量).
    Update distribution $D_m$:
        $$D_{m+1}(i) = \frac{D_m(i)}{Z_m} {\beta_m}^{1-|h_m(\mathbf{x}_i)-Y_i|},$$
      where $Z_m$ is the normalization constant such that $\sum_{i=1}^{n} D_{m+1}(i) = 1$.
end
Output: Return the final hypothesis
    $$H(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_{m=1}^{M} \log\left(\frac{1}{\beta_m}\right) h_m(\mathbf{x}) \geq \frac{1}{2} \sum_{m=1}^{M} \log\left(\frac{1}{\beta_m}\right) \\ 0 & \text{otherwise} \end{cases} \quad (2.1)$$
```

If in **Algorithm 1** the weak learner/hypothesis h_m at step $m \in \{1, \ldots, M\}$ is such that $\epsilon_m < \frac{1}{2}$, then $\beta_m \in [0, 1)$. Therefore, weights of data points which have been correctly classified by h_m are reduced by a factor β_m and they are left unchanged otherwise, before re-normalization. This ends up in reducing the weights corresponding to data points correctly classified by multiple weak learners, and viceversa for the so called 'hard examples' [13]. In fact, one has

Lemma 2.1. *Let $m \in \{1, \dots, M\}$ and $n > 1$; with h_m we denote the m-th weak hypothesis in* **Algorithm 1** *with misclassification error $\epsilon_m < \frac{1}{2}$. Then*

$$D_{m+1}(i) < D_m(i),$$

for all $i \in \{1, \dots, n\}$ such that $\underline{h_m(\mathbf{x}_i) = Y_i}$. 相当于是，预测对了，则下期对误差的贡献

Proof. The proof can be found in the Appendix.

The final hypothesis (2.1) can be rewritten as

使得误差之和最小

$$H(\mathbf{x}) = \underset{y \in \{0,1\}}{\arg\max} \sum_{m : h_m(\mathbf{x}) = y} \underline{\log\left(\frac{1}{\beta_m}\right)}; \tag{2.2}$$

this follows from the identities:

对 hm 的一个 weighted,

$$\sum_{m=1}^{M} \log\left(\frac{1}{\beta_m}\right) \underline{h_m(\mathbf{x})} = \sum_{m : h_m(\mathbf{x}) = 1} \log\left(\frac{1}{\beta_m}\right),$$

$$\sum_{m=1}^{M} \log\left(\frac{1}{\beta_m}\right) = \sum_{m : h_m(\mathbf{x}) = 1} \log\left(\frac{1}{\beta_m}\right) + \sum_{m : h_m(\mathbf{x}) = 0} \log\left(\frac{1}{\beta_m}\right).$$

Therefore, (2.2) shows that the final hypothesis outputs the label of a data point corresponding to the maximum of the sum of weights of all those weak learners that correctly classified the data point itself. As the weight is of the form $\log\left(\frac{1}{\beta_m}\right) = -\log \beta_m$ (with $\beta_m \in [0, 1)$), then the lower the β_m (or, equivalently, the lower the misclassification error ϵ_m), the higher the corresponding weight for the weak learner h_m.
A bound on the misclassification training error of the final hypothesis output by **Algorithm 1** is well known and proven in [14]: we show below the original result by Freund and Schapire.

Theorem 2.2 (Theorem 6, [14])**.** *Suppose the weak learning algorithm* **WeakLearn** *when called by* **Algorithm 1**, *generates hypotheses $h_1, \dots, h_M$ with errors $\epsilon_1, \dots, \epsilon_M$. Then, the error*

$$\epsilon_f := \sum_{i=1}^{n} D_{M+1}(i) |H(\mathbf{x}_i) - Y_i|$$

of the final hypothesis H in (2.2) satisfies

$$\epsilon_f \leq 2^M \prod_{m=1}^{M} \sqrt{\epsilon_m (1 - \epsilon_m)}. \tag{2.3}$$

Equivalently, one can introduce the edge γ_m of the m-th weak classifier h_m with error ϵ_m as $\gamma_m := \frac{1}{2} - \epsilon_m$ and rewrite (2.3) as

$$\epsilon_f \leq \prod_{m=1}^{M} \sqrt{1 - 4\gamma_m^2} \leq \exp\left(-2 \sum_{m=1}^{M} \gamma_m^2\right), \tag{2.4}$$

where for the second inequality we use $1+x \leq e^x$, $\forall x \in \mathbb{R}$. The error bound formulation (2.4) is reported, for example, in [31]. The edge γ_m measures how the improvement over random guessing of the m-th weak classifier h_m, when considering its weighted misclassification error ϵ_m. Theorem 2.2 shows that the original `AdaBoost` misclassification *training* error drops exponentially fast as a function of the number of steps M.

In the literature the resistance to overfitting shown by adaptive boosting algorithms has been discussed extensively. However, a theoretical analysis of the generalization (or out-of sample or test) error of `AdaBoost` is out of scope of this tutorial, as well as the discussion of strong/weak PAC (Probably Approximately Correct) learning algorithms which lie behind the use of the weak learner **WeakLearn** in **Algorithm 1**; we refer to both [14] and [31] for additional details.

2.2.2 `AdaBoost` formulation from [31]

The `AdaBoost` formulation from the classical textbook on boosting [31] and there denoted as **Algorithm 1.1** is presented in **Algorithm 2** below. We discuss it for sake of completeness, before moving to multi-class generalizations of the original `AdaBoost` algorithm in Section 2.2.3.

Algorithm 2: `AdaBoost` algorithm in [31] for binary classification

```
Input :
    • training data $\mathcal{D} = \{(Y_1, \mathbf{x}_1), \ldots, (Y_n, \mathbf{x}_n)\}$, with labels $Y_i \in \mathcal{Y} = \{-1, 1\}$
    • weak learning algorithm WeakLearn
    • number of iterations $M \geq 1$

Initialize $D_1(i) = \frac{1}{n}$, for all $i = 2, \ldots, n$.
while $m \in \{1, \ldots, M\}$ do
    Fit WeakLearn to re-weighted training set $(\mathcal{D}, D_m)$.
    Return a hypothesis $h_m : \mathcal{X} \to \mathcal{Y}$ and compute its weighted misclassification error
      $\epsilon_m = \sum_{i=1}^{n} D_m(i) \mathbb{I}\left[Y_i \neq h_m(\mathbf{x}_i)\right]$.
    Set $\alpha_m = \frac{1}{2} \log\left(\frac{1-\epsilon_m}{\epsilon_m}\right)$.
    Update distribution $D_m$:
        $D_{m+1}(i) = \frac{D_m(i)}{Z_m} \exp\big(-\alpha_m Y_i h_m(\mathbf{x}_i)\big),$
      where $Z_m$ is the normalization constant such that $\sum_{i=1}^{n} D_{m+1}(i) = 1$.
end
Output: Return the final hypothesis
    $F(\mathbf{x}) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m h_m(\mathbf{x})\right).$        (2.5)
```

We also note that the final hypothesis (2.5) can be rewritten in the form

$$F(\mathbf{x}_i) := \arg\max_{y \in \mathcal{Y}} \sum_{m: h_m(\mathbf{x}_i) = y} \alpha_m h_m(\mathbf{x}_i), \tag{2.6}$$

as $h_m(\mathbf{x}) = \pm 1$, for all $m \in \{1, \dots, M\}$. Moreover, $\alpha_m > 0 \Leftrightarrow \epsilon_m < \frac{1}{2}$; in that case, the weights $D_m(i)$ are multiplied (before normalization) to a factor greater or smaller than one in case of wrong or correct classification of the data point $\mathbf{x}_i$, respectively. Typically, one assumes that each weak learner h_m is (slightly) better than random guessing, which would give a misclassification rate of $\frac{1}{2}$: this is equivalent to $\alpha_m = 0$.

Algorithm 1 and **Algorithm 2** differ in both label set and update rules. However, it can be proved that they are formally equivalent, i.e. at each iteration they compute the same weights and they return the same final hypothesis, under mild assumptions and introducing the bijection $\varphi : \{0, 1\} \to \{-1, 1\}, \varphi(Y_i) := 2Y_i - 1$. Details are left to the reader.

2.2.3 `AdaBoost.M1` in [14] and [13]

The adaptive boosting algorithm `AdaBoost.M1` in [14] and [13] is a boosting algorithm conceived for multiclass problems, with label set $\mathcal{Y} = \{1, \dots, k\}, k \geq 2$; in the literature it is often cited as the original `AdaBoost` algorithm. It is easy to show that it is a direct generalization of **Algorithm 1**, with the quantity $|h_m(\mathbf{x_i}) - Y_i|$ replaced by the indicator function $\mathbb{I}\left[h_m(\mathbf{x}_i) \neq Y_i\right]$. The `AdaBoost.M1` final hypothesis is presented in the same form as (2.2).

2.2.4 `AdaBoost` algorithms in [16]: 'real', 'discrete', 'gentle' adaptive boosting and `LogitBoost`

In the work of [16], the authors introduce new boosting algorithms denoted by 'Real AdaBoost', 'Gentle AdaBoost', `LogitBoost` as well as 'Discrete AdaBoost' (which is the algorithm presented in [13]), and discuss the statistical theory of boosting based on maximum likelihood, additive modeling and optimization problems. They consider the set of labels $\mathcal{Y} = \{-1, 1\}$.

In this section we only briefly discuss the `LogitBoost` algorithm, as in Section 2.3 we will provide some results concerning its statistical theory. For sake of readability, we present `LogitBoost` in **Algorithm 3** following the notation from [31]; the implementation of `LogitBoost` in [16] can be obtained with minor changes. For the description of and the motivation behind the algorithms 'Real AdaBoost', 'Discrete AdaBoost' and 'Gentle AdaBoost' we refer to the original paper [16].

Algorithm 3: `LogitBoost` algorithm in [31] (simplified version)

```
Input :
    • training data $\mathcal{D} = \{(Y_1, \mathbf{x}_1), \ldots, (Y_n, \mathbf{x}_n)\}$, with labels $Y_i \in \mathcal{Y} = \{-1, 1\}$
    • number of iterations $M \geq 1$

Initialize $f_0(\mathbf{x}) = 0$.
while $m \in \{1, \ldots, M\}$ do
    Compute
        $p_m(\mathbf{x}_i) = \frac{1}{1 + \exp(-f_{m-1}(\mathbf{x}_i))}$,    (2.7)
        $z_m(\mathbf{x}_i) = \begin{cases} \frac{1}{p_m(\mathbf{x}_i)} & \text{if } Y_i = +1, \\ -\frac{1}{1-p_m(\mathbf{x}_i)} & \text{if } Y_i = -1, \end{cases}$    (2.8)
        $w_m(i) = p_m(\mathbf{x}_i)(1 - p_m(\mathbf{x}_i))$.    (2.9)
    Fit $f_m^*(\mathbf{x})$ by a weighted least-squares regression to the responses $z_m(\mathbf{x}_i)$ in the
      features $\mathbf{x}_i$ with weights $w_m(i)$.
    Update $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + f_m^*(\mathbf{x})$
end
Output: Return $f_M(\mathbf{x})$.
```

A classification rule C leveraging the `LogitBoost` algorithm can be written as $C(\mathbf{x}) = \text{sign}(f_M(\mathbf{x}))$; the structure of the `LogitBoost` implementation will be clarified in some detail in Section 2.3.

2.2.5 Adaptive boosting in [36]: SAMME and SAMME.R algorithms

The adaptive boosting algorithms SAMME—i.e. *Stagewise Additive Modeling using a Multi-class Exponential loss function*—and SAMME.R have been introduced in [36]; they are defined in the context of multi-class classification, where most of the boosting algorithms in the literature have been restricted to reduce the multi-class classification problem to multiple two-class problems. We consider them in this tutorial as they are used in the main Python implementation of adaptive boosting.

SAMME algorithm is presented in **Algorithm 4**, while SAMME.R is given in **Algorithm 5**; we present both as in the original paper [36] with the addition of learning rates (which are not specified in [36]) because these latter are implemented by default in the `AdaBoostClassifier()` function of the `scikit-learn` Python package. We refer to Section 3.1 for additional details.

Algorithm 4: SAMME algorithm in [36] for multi-class classification

```
Input :
    • training data $\mathcal{D} = \{(Y_1, \mathbf{x}_1), \dots, (Y_n, \mathbf{x}_n)\}$, with labels $Y_i \in \mathcal{Y} = \{1, \dots, k\}, k \geq 2$.
    • set $\mathcal{F}$ of weak classifiers
    • learning rate $\eta \in (0, 1]$
    • number of iterations $M \geq 1$

Initialize $\omega_i = \frac{1}{n}$, for all $i = 1, \dots, n$.
while $m \in \{1, \dots, M\}$ do
    Fit a weak classifier $h_m$ to the weighted training set $(\mathcal{D}, \omega)$, with $\omega = (\omega_1, \dots, \omega_n)$.
    Compute the weighted misclassification error $\epsilon_m = \sum_{i=1}^n \omega_i \mathbb{I}\left[Y_i \neq h_m(\mathbf{x}_i)\right]$.
    Set $\alpha_m = \eta \left( \log \left( \frac{1-\epsilon_m}{\epsilon_m} \right) + \log(k-1) \right)$.
    Update weights:
        $\hat{\omega}_i := \omega_i \exp\left(\alpha_m \mathbb{I}\left[Y_i \neq h_m(\mathbf{x}_i)\right]\right), \; i = 1, \dots, n.$
    Re-normalize weights: $\omega_i = \frac{\hat{\omega}_i}{\sum_{i=1}^n \hat{\omega}_i}$.
end
Output: Return the classifier
    $H(\mathbf{x}) = \arg\max_{y \in \mathcal{Y}} \sum_{m=1}^{M} \alpha_m \mathbb{I}[y = h_m(\mathbf{x}_i)].$
```

We note that

$$\alpha_m > 0 \Leftrightarrow \epsilon_m < 1 - \frac{1}{k}, \tag{2.10}$$

in **Algorithm 4**, i.e. SAMME allows the misclassification training error ϵ_m of the m-th weak classifier h_m to be higher than $\frac{1}{2}$, which is the threshold above which `AdaBoost.M1` stops. Equivalently, the in-sample accuracy of h_m has to be better than random guessing. Therefore, SAMME is more robust than multi-class `AdaBoost.M1`, and we refer to [36] for additional discussions on this point.

In the case of binary classification, i.e. $k = 2$, and using a learning rate $\eta = 1$ it is possible to show the equivalence between SAMME algorithm presented in **Algorithm 4** and the original `AdaBoost` implementation in **Algorithm 1**.

We turn now the attention to SAMME.R, which is a multi-class generalization of the 'Real Adaboost' algorithm in [16]; both 'Real AdaBoost' and SAMME.R are *ad-hoc* boosting algorithms, as they compute weighted probabilities estimates at each boosting iteration, which are then used to generate coefficients used to define the final classifier; on the other hand, weighted probabilities are used for the weight update of training data points, as well.

Algorithm 5: SAMME.R algorithm in [36] for multi-class classification

```
Input :
    • training data $\mathcal{D} = \{(Y_1, \mathbf{x}_1), \dots, (Y_n, \mathbf{x}_n)\}$, with labels $Y_i \in \mathcal{Y} = \{1, \dots, K\}, K \geq 2$.
    • set $\mathcal{F}$ of weak classifiers
    • learning rate $\eta \in (0, 1]$
    • number of iterations $M \geq 1$
Initialize $\omega_i = \frac{1}{n}$, for all $i = 1, \dots, n$.
while $m \in \{1, \dots, M\}$ do
    Fit a weak classifier $h_m$ to the weighted training set $(\mathcal{D}, \omega)$, with $\omega = (\omega_1, \dots, \omega_n)$.
    Compute the weighted class probability estimates
     $p_{m,k}(\mathbf{x}) = \sum_{i=1}^n \omega_i \mathbb{I}[h_m(\mathbf{x}) = k],\ k = 1, \dots, K$.
    Set $h_{m,k}(\mathbf{x}) = (K-1)\left(\log p_{m,k}(\mathbf{x}) - \frac{1}{K}\sum_{j=1}^K \log p_{m,j}(\mathbf{x})\right),\ k = 1, \dots, K$.
    Update weights[a,b]:
        $$\hat{\omega}_i = \omega_i \exp\left(-\frac{K-1}{K}\eta\langle \mathbf{y}_i, \log \mathbf{p}_m(\mathbf{x})\rangle\right),\ i = 1, \dots, n.$$
    Re-normalize weights: $\omega_i = \frac{\hat{\omega}_i}{\sum_{i=1}^n \hat{\omega}_i}$.
end
Output: Return the classifier
        $$H(\mathbf{x}) = \arg\max_{k \in \mathcal{Y}} \sum_{m=1}^M h_{m,k}(\mathbf{x}).$$
```

[a] The K-dimensional vector $\mathbf{y}_i$ is encoded as

$$\mathbf{y}_i = \begin{cases} 1 & \text{if } Y_i = k, \\ -\frac{1}{k-1} & \text{otherwise.} \end{cases}$$

On the other hand, $\mathbf{p}_m(\mathbf{x}) = (p_{m,1}(\mathbf{x}), \cdots, p_{m,k}(\mathbf{x}))$. We refer to [36] Section 2.1 for all details.

[b] With $\langle \cdot, \cdot \rangle$ we indicate the standard scalar product in $\mathbb{R}^K$.

We will encounter the SAMME.R algorithm again in Section 3.1.

2.3 Statistical theory of boosting

The statistical theory of boosting sees the specification of a loss functional and describes the algorithmic boosting procedure as an infinite-dimensional optimization problem. In the original formulation of AdaBoost [14] no loss functional is considered. In fact, it is only after the seminal work of [16] that a statistical formulation of AdaBoost was introduced, greatly supporting the introduction of new boosting algorithms by specifying new loss functionals and describing their minimization procedures. In this section, we keep the notation and the mathematics as simple as possible. We refer to [1] for a mathematically more complete exposition on the statistical theory of gradient boosting.

We consider a *functional gradient descent* approach to loss functional minimization for boosting. In other words, considering a boosting problem we specify a given space of functions and specify a sequential procedure—called **gradient** or **Newton boosting algorithm**—to compute an approximation of the minimum of the given loss functional. All boosting algorithms described so far can be re-written as either gradient or Newton boosting algorithms. We will describe this result at the end of the current section.

Let us introduce the in-sample loss functional $\mathcal{L}_L : \mathbb{F} \to \mathbb{R}_+$ over a given space of functions $\mathbb{F} \subset \mathbb{R}^{\mathcal{X}}$, where

泛函.

$\{f \mid f: \mathcal{X} \to \mathbb{R}\}.$

$$\mathcal{L}_L[F] := \frac{1}{n}\sum_{i=1}^{n} L(Y_i, F(\mathbf{x}_i)), \;\; F \in \mathbb{F}, \tag{2.11}$$

for a given loss function L measuring the cost incurred in predicting the target Y_i with the prediction $F(\mathbf{x}_i)$, for all $(Y_i, \mathbf{x}_i) \in \mathcal{D}$. The loss function $L : \mathcal{Y} \times \mathbb{R} \to \mathbb{R}_+$ is assumed to be convex in the second argument; it may be differentiable or not. For regression problems, one typically uses the loss function $L(y, x) = (y - x)^2$; on the other hand, for classification exercises where the label space is $\mathcal{Y} = \{-1, 1\}$ we can consider the losses of the form $L(y, x) = \psi(yx)$, with $\psi : \mathbb{R} \to \mathbb{R}_+$ being convex. Classical examples of losses of the above form comprise the logistic, exponential and hinge losses:

$$\begin{aligned}
\psi_{\text{logit}}(u) &= \log(1 + \exp(-u)), \\
\psi_{\exp}(u) &= \exp(-u), \\
\psi_{\text{hinge}}(u) &= \max(1 - u, 0).
\end{aligned}$$

ϕ

Note that the hinge loss is not differentiable. For an interesting discussion on convexity and strong convexity of loss functions we refer to [1]. Both the logistic and exponential loss will play a major role in this tutorial.

The goal is to find F that minimizes $\mathcal{L}_L$. To do so, we need to specify a procedure to compute F at points $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathcal{X}$ and, at the same time, introduce a methodology to avoid overfitting as the whole minimization is clearly performed in-sample.

In a functional gradient descent approach to loss functional minimization for boosting, we compute (2.11) iteratively by considering the functions F in the subspace $\mathbb{F} := \langle \mathcal{H} \rangle$ of finite linear combinations over $\mathbb{R}$ of weak learners in $\mathcal{H}$. In other words, in presence of M boosting iterations, we estimate both coefficients α_k and base learners f_k of the function $F = \sum_{k=1}^{M} \alpha_k f_k \in \mathbb{F}$ either by gradient or Newton approximations of the loss function L at each step m. In the former case we speak of gradient boosting algorithm, Netwon boosting algorithm in the latter. Let us discuss both in detail.

确实是子空间

Let $\mathbf{f}_m \in \mathbb{F}_M$, with $\mathbf{f}_m = \sum_{k=1}^{m} \alpha_k f_k$ denote the function at the m-th step of the gradient or Newton boosting algorithm used to approximate the function $F = \sum_{k=1}^{M} \alpha_k f_k$. As $\mathbf{f}_m$ is observed only at points $\mathbf{x}_i \in \mathcal{X}$ we can write

$$\mathbf{f}_m = (f_{m,1}, \ldots, f_{m,n}) \in \mathbb{R}^n,$$

$$\mathbf{f}_m(\mathbf{x}_i) = f_{m,i} := \sum_{k=1}^{m} \alpha_k f_k(\mathbf{x}_i).$$

The gradient $\nabla\mathcal{L}_L(\mathbf{f}_m)$ of $\mathcal{L}_L$[8] at $\mathbf{f}_m$ reads

$$\nabla\mathcal{L}_L(\mathbf{f}_m) = \frac{1}{n}\left(\frac{\partial L(Y_1, f_1)}{\underline{\partial f_1}}\bigg|_{f_1=f_{m,1}}, \dots, \frac{\partial L(Y_n, f_n)}{\underline{\partial f_n}}\bigg|_{f_n=f_{m,n}} \right),$$

while the Hessian matrix $\nabla^2\mathcal{L}_L(\mathbf{f}_m) = \left\{ \left(\nabla^2\mathcal{L}_L(\mathbf{f}_m)\right)_{ij} \right\}_{i,j=1,\dots,n}$ of $\mathcal{L}_L$ at $\mathbf{f}_m$ is such that

应该就是对第二个变量求二阶导数.

$$\left(\nabla^2\mathcal{L}_L(\mathbf{f}_m)\right)_{ij} = \frac{1}{n}\delta_{ij}\boxed{\frac{\partial^2 L}{\partial f_{m,i}^2}}(Y_i, f_{m,i}),$$

and denoting by δ_{ij} the Kronecker delta. Classically, for the gradient descent we consider the approximation around $\mathbf{f}_m$

$$\mathcal{L}_L(\mathbf{f}) = \mathcal{L}_L(\mathbf{f}_m) + \langle \nabla\mathcal{L}(\mathbf{f}_m), \mathbf{f} - \mathbf{f}_m \rangle + o(\|\mathbf{f} - \mathbf{f}_m\|),$$

and the quadratic approximation

$$\begin{aligned} \mathcal{L}_L(\mathbf{f}) = \mathcal{L}(\mathbf{f}_m) + \langle \nabla\mathcal{L}(\mathbf{f}_m), \mathbf{f} - \mathbf{f}_m \rangle + \\ \frac{1}{2}\langle \nabla^2\mathcal{L}(\mathbf{f}_m)(\mathbf{f} - \mathbf{f}_m), \mathbf{f} - \mathbf{f}_m \rangle + o(\|\mathbf{f} - \mathbf{f}_m\|^2) \end{aligned}$$

for the Newton approximation, instead, as $o(\|\mathbf{f} - \mathbf{f}_m\|) \to 0$. Choosing a small step size $\rho > 0$, we can perform the updates [26]:

$$\mathbf{f}_m \to \mathbf{f}_{m+1} = \mathbf{f}_m - \rho\nabla\mathcal{L}(\mathbf{f}_m), \text{ (gradient boosting)} \tag{2.12}$$

$$\mathbf{f}_m \to \mathbf{f}_{m+1} = \mathbf{f}_m - \left(\nabla^2\mathcal{L}(\mathbf{f}_m)\right)^{-1}\nabla\mathcal{L}(\mathbf{f}_m). \text{ (Newton boosting)} \tag{2.13}$$

To avoid overfitting, in gradient and Newton boosting algorithms the updates (2.12) and (2.13) are replaced as follows, with notation from [15] and [23]. In case of gradient boosting, at each step m one selects the base learner h_m^* which is most highly correlated with the negative gradient $\nabla\mathcal{L}(\mathbf{f}_m)$, i.e.

$$h_m^* = \underset{h\in\mathcal{F},\beta}{\arg\min} \sum_{i=1}^{n} \left(g_m(\mathbf{x}_i) - \beta h(\mathbf{x}_i)\right)^2,$$

where $g_m(\mathbf{x}_i) := -\frac{\partial L(Y_i, f_i)}{\partial f_i}\Big|_{f_i=f_{m,i}}$. The small step size ρ_m to take in the direction of h_m^* is determined by line search

$$\rho_m = \underset{\rho}{\arg\min} \sum_{i=1}^{n} L(Y_i, f_{m-1}(\mathbf{x}_i) + \rho h_m^*(\mathbf{x}_i)),$$

instead. For the interested reader, the gradient boosting algorithm is presented in [15] and denoted by **Gradient_Boost**. We just note that in the same paper, in Section 5, a coefficient $0 < \eta \leq 1$ is also introduced for regularization, after the estimation of the step size ρ_m. The methodology is called shrinkage, and η is commonly referred to as learning rate. Therefore, we propose **Gradient_Boost** in **Algorithm 6** with learning rate η for sake of completeness.

[8] For sake of simplicity, we only consider twice differentiable convex loss functions $(f_{m,1}, \cdots, f_{m,n}) \mapsto \mathcal{L}_L(f_{m,1}, \cdots, f_{m,n}) = \frac{1}{n}\sum_{i=1}^{n} L(Y_i, f_{m,i})$.

Other useful references for the gradient boosting algorithm are [35], where the algorithm is called 'gradient boosting machine', and [23], Chapter 4.

Algorithm 6: Gradient boosting algorithm in [15] (including learning rate η).

```
Input :
    • training data $\mathcal{D} = \{(Y_1, \mathbf{x}_1), \ldots, (Y_n, \mathbf{x}_n)\}$, with labels $Y_i \in \mathcal{Y} = \{-1, 1\}$
    • number of iterations $M \geq 1$
    • learning rate $\eta \in (0, 1]$.

Initialize $f_0(\mathbf{x}) = \underset{\theta \in \mathbb{R}}{\arg\min} \sum_{i=1}^{n} L(Y_i, \theta)$.
while $m \in \{1, \ldots, M\}$ do
    $g_m(\mathbf{x}_i) := -\left.\frac{\partial L(Y_i, f_i)}{\partial f_i}\right|_{f_i = f_{m-1,i}}$.

    Compute $h_m^* = \underset{h \in \mathcal{F}, \beta}{\arg\min} \sum_{i=1}^{n} (g_m(\mathbf{x}_i) - \beta h(\mathbf{x}_i))^2$.
    Compute $\rho_m = \underset{\rho > 0}{\arg\min} \sum_{i=1}^{n} L(Y_i, f_{m-1}(\mathbf{x}_i) + \rho h_m^*(\mathbf{x}_i))$.
    Scale $f_m^*(\mathbf{x}) = \eta \rho_m h_m^*(\mathbf{x})$.
    Update $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + f_m^*(\mathbf{x})$.
end
Output: Return $f_M(\mathbf{x})$.
```

On the other hand, Newton boosting algorithms replace the update rule (2.13) by selecting the base learner h_m^* such that

$$h_m^* = \underset{h \in \mathcal{F}}{\arg\min} \sum_{i=1}^{n} \left(g_m(\mathbf{x}_i) h(\mathbf{x}_i) + \frac{1}{2} H_m(\mathbf{x}_i) h^2(\mathbf{x}_i) \right), \tag{2.14}$$

where $H_m(\mathbf{x}_i) := \left(\nabla^2 \mathcal{L}_L(\mathbf{f}_m)\right)_{ii}$. Minimization in (2.14) is equivalent to

$$h_m^* = \underset{h \in \mathcal{F}}{\arg\min} \sum_{i=1}^{n} \frac{1}{2} H_m(\mathbf{x}_i) \left(-\frac{g_m(\mathbf{x}_i)}{H_m(\mathbf{x}_i)} - h(\mathbf{x}_i) \right)^2, \tag{2.15}$$

i.e. a weighted least squares regression problem. We present Newton boosting algorithm in **Algorithm 7**, with learning rate η, for sake of completeness.

```
Algorithm 7: Newton boosting algorithm in [23].
Input :
    • training data $\mathcal{D} = \{(Y_1, \mathbf{x}_1), \ldots, (Y_n, \mathbf{x}_n)\}$, with labels $Y_i \in \mathcal{Y} = \{-1, 1\}$
    • number of iterations $M \geq 1$
    • learning rate $\eta$
Initialize $f_0(\mathbf{x}) = \arg\min_{\theta} \sum_{i=1}^{n} L(Y_i, \theta)$.
while $m \in 1, \ldots, M$ do
    $g_m(\mathbf{x}_i) := -\frac{\partial L(Y_i, f_i)}{\partial f_i}\Big|_{f_i = f_{m-1,i}}$.
    $H_m(\mathbf{x}_i) := \left(\nabla^2 \mathcal{L}_L(\mathbf{f}_{m-1})\right)_{ii}$.
    Compute $h_m^* = \arg\min_{h \in \mathcal{F}} \sum_{i=1}^{n} \frac{1}{2} H_m(\mathbf{x}_i) \left(-\frac{g_m(\mathbf{x}_i)}{H_m(\mathbf{x}_i)} - h(\mathbf{x}_i)\right)^2$.
    Scale $f_m^*(\mathbf{x}) = \eta h_m^*(\mathbf{x})$.
    Update $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + f_m^*(\mathbf{x})$.
end
Output: Return $f_M(\mathbf{x})$.
```

Newton boosting is a forward additive algorithm adding at each iteration m a weak learner h_m^* rescaled by a fixed coefficient, i.e. the learning rate η, to the output of the previous iteration. This is a difference with respect to gradient boosting, as in the latter algorithm rescaling includes a coefficient resulting from line search (together with learning rate). We refer to both [35] and [23] for additional details.
We introduce now the main result of this section.

Theorem 2.3. *Original* `AdaBoost`*, i.e.* **Algorithm 1***, is a gradient boosting algorithm with exponential loss* $L(Y_i, F(x_i)) = \exp(-Y_i F(x_i))$ *and learning rate* $\eta = 1$*.* `LogitBoost` *from* **Algorithm 3** *is a Newton boosting algorithm with logistic loss* $L(Y_i, F(x_i)) = \log(1+\exp(-Y_i F(x_i)))$ *and learning rate* $\eta = 1$*.*

Proof. For the proof of `AdaBoost` we refer to Sections 7.1 and 7.4 in [31]. The proof for `LogitBoost` is sketched in the Appendix.

Theorem 2.3 asserts that the original `AdaBoost` respectively `LogitBoost` algorithms are equivalent to gradient, respectively Newton optimization routines, once we specify exponential, respectively logistic loss functions. This concludes our analysis of the statistical theory of boosting.

3 `AdaBoost` in Python and R

In this section we present an overview of the main packages in Python and R providing `AdaBoost` algorithms. In what follows we focus on classification problems only and we provide links to the package descriptions as well as selected online resources for additional details.

3.1 AdaBoost in Python

AdaBoost algorithms can be used using the AdaBoostClassifier() function[9], which is available in the sklearn.ensemble module[10] of the scikit-learn[11] package (often abbreviated as sklearn). AdaBoostClassifier() allows to select either SAMME or SAMME.R algorithms to train the ensemble through the algorithm parameter; in the next few sections we collect key topics on both SAMME and SAMME.R implementations in AdaBoostClassifier().

3.1.1 AdaBoostClassifier(): base learners

AdaBoostClassifier() can use different classes of base learners, if they support re-weighting of data points during learning. The parameter initializing the sequential re-weighting of data points is sample_weight: it is discussed in Section 3.1.2. By default, AdaBoostClassifier() uses decision tree stumps as base learners, as the default specification base_estimator=None is equivalent to use DecisionTreeClassifier(max_depth=1). However, as remarked in [10], it is often necessary to increase tree depth in order to capture nonlinearities in data. The scikit-learn function DecisionTreeClassifier()[12] is commonly used to grow trees in Python: we recommend the official page[13] for additional information on DecisionTreeClassifier(). As decision trees (and, in particular, their stumps) do support calculation of class probabilities and re-weighting of training data points during learning, they are suitable for both SAMME and SAMME.R adaptive boosting algorithms.
Below we summarize selected information on DecisionTreeClassifier() and the use of weighting of training data points during decision tree growth:

- Weighting of training data points (which is specified through sample_weight, as in the case of AdaBoostClassifier()) is used to compute weighted empirical probabilities for tree growing (i.e. sequential splitting). Following the notation of Section 5.3.1 in [35], the weighted empirical probability that a randomly chosen training data point $(Y_i, \mathbf{x}_i)$ in $\mathcal{X}' \subset \mathcal{X}$ has response y is given by

$$\hat{p}_\omega(y|\mathcal{X}') := \frac{n_\omega(y, \mathcal{X}'; \mathcal{D})}{\sum_{y \in \mathcal{Y}} n_\omega(y, \mathcal{X}'; \mathcal{D})} = \frac{\sum_{i=1}^n \omega_i \mathbb{I}[Y_i = y, \mathbf{x}_i \in \mathcal{X}']}{\sum_{i=1}^n \omega_i \mathbb{I}[\mathbf{x}_i \in \mathcal{X}']}, \tag{3.1}$$

  where the vector of weights $\omega = (\omega_1, \ldots, \omega_n)$ is initialized in DecisionTreeClassifier() by sample_weight. If all training data points are equally weighted, then the weighted empirical probabilities (3.1) reduce to the empirical probabilities $\hat{p}(y|\mathcal{X}')$ in formula (5.25) of [35] and DecisionTreeClassifier() reduces to classical classification tree algorithm. Note that we assume that there is at least one $\mathbf{x}_i$ such that $\mathbf{x}_i \in \mathcal{X}'$.

- DecisionTreeClassifier() supports the following impurity measures for split decisions: Gini (default choice) and information gain-based impurity. Both are functions of weighted empirical probabilities.

[9] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#id2

[10] http://scikit-learn.org/stable/modules/ensemble.html#ensemble

[11] http://scikit-learn.org/stable/index.html

[12] http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

[13] https://scikit-learn.org/stable/modules/tree.html#tree-classification

- To perform splits, `DecisionTreeClassifier()` computes the weighted impurity decrease as follows: let us consider a node and its left and right child nodes; as described under the `min_impurity_decrease` at line 598 in the section in the official `DecisionTreeClassifier()` documentation[14], we have:
  '*The weighted impurity decrease equation is the following:*

  ```
  N_t / N * (impurity - N_t_R / N_t * right_impurity
                      - N_t_L / N_t * left_impurity)
  ```

  where `N` *is the total number of samples,* `N_t` *is the number of samples at the current node,* `N_t_L` *is the number of samples in the left child, and* `N_t_R` *is the number of samples in the right child.* `N`*,* `N_t`*,* `N_t_R` *and* `N_t_L` *all refer to the weighted sum, if* `sample_weight` *is passed.'*
  With `impurity`, `left_impurity` and `right_impurity` the evaluation of the impurity function at the current node, left child node and child right node is meant. The scalars `N`, `N_t`, `N_t_L` and `N_t_L` denote the weighted (again, using `sample_weight`) number of samples in the training data set, parent node, left child node and right child node.

- `DecisionTreeClassifier()` does not support categorical variables[15]. This means that the function `DecisionTreeClassifier()` tries to convert categorical variables into numerical ones, before running tree growing routines. At the time of writing, an active pull-request[16] is taking care of the implementation of categorical splits for tree-based learners. *De facto*, encodings are introduced to use categorical variables in predictive modeling with `DecisionTreeClassifier()`. In `sklearn`, main choices are `LabelEncoding()` and `OneHotEncoder()`. We also note that the official `sklearn` documentation[17] mentions the use of the CART[18] algorithm for growing decision trees, although without support for categorical variables, as highlighted above. This is a major difference with the original formulation of the CART algorithm, which supports the use of categorical variables without introducing encodings. We refer to the monograph [3] for all details on the original CART algorithm and to Section 9.2.4 of [17] for a short discussion on computational issues arising from categorical variables with many levels.

3.1.2 `AdaBoostClassifier()`: use of `sample_weight`

Re-weighting of training data points is performed in `AdaBoostClassifier()` by specifying the parameter `sample_weight`; by default, if `sample_weight==None` is passed, the algorithm assigns a weight equal to $\frac{1}{n}$ to each training data point, where n denotes the cardinality of the training data set under consideration. We remark that this is the choice of weighting at the first boosting step for **Algorithm 1**, **Algorithm 2**, `LogitBoost`, SAMME and SAMME.R.

It is interesting to note that also negative values for `sample_weight` can be initialized in `AdaBoostClassifier()`, although the sum of all `sample_weight` has to be non-negative or otherwise the algorithm stops. For the interested reader, a discussion on the use of negative

[14] https://github.com/scikit-learn/scikit-learn/blob/7389dba/sklearn/tree/tree.py#L598

[15] https://github.com/scikit-learn/scikit-learn/issues/12398

[16] https://github.com/scikit-learn/scikit-learn/pull/4899

[17] https://scikit-learn.org/stable/modules/tree.html#tree-algorithms-id3-c4-5-c5-0-and-cart

[18] Classification And Regression Trees.

sample weightings in machine learning problems stemming from high energy physics research can be found on `sklearn` issues online discussions[19].

3.1.3 `AdaBoostClassifier()`: learning rate and number of boosting steps

The performance trade-off between the number of base classifiers and the learning rate during the ensemble training can be checked empirically by controlling the `n_estimators` and `learning_rate` parameters in `AdaBoostClassifier()`.
By default, learning rate in `AdaBoostClassifier()` is set to 1. We also note that learning rates are implemented in the computation of the coefficients α_m for SAMME and the update of weights ω_i for SAMME.R in lines 570 and 522 in the source code for `AdaBoostClassifier` available at the `sklearn` GitHub repository[20]. For these reasons, we explicitly inserted learning rates at line **5** and **6** in the SAMME and SAMME.R implementations presented in **Algorithm 4** and **Algorithm 5**.

3.1.4 `AdaBoostClassifier()`: early stops

Both SAMME and SAMME.R implementations present few early stop conditions that allow us to terminate if conditions critical for the sequential boosting computations are not met; for example, the SAMME algorithm implementation stops computations if, at any boosting step, the base learner shows a misclassification error `estimator_error` such that `estimator_error <= 0` or `estimator_error >= 1. - (1. / n_classes)`. In the first equality one consider the possibility of having a negative misclassification error as `AdaBoostClassifier` allows for negative weighting of training data points (as discussed in Section 3.1.2) or a perfect classifier (such that the misclassification training error is zero); the second inequality is simply the implementation of the bound (2.10).

More details on both SAMME and SAMME.R implementations in `AdaboostClassifier()` are given in the code for the class `AdaBoostClassifier(BaseWeightBoosting, ClassifierMixin)` at lines 536 and 478 in the official `sklearn` Python script[21].
Apart from `scikit-learn`, many Python implementations of `AdaBoost` algorithms are available on GitHub repositories or personal blogs of data science practitioners. These resources are recommended for self study.

3.2 `AdaBoost` in R

R presents multiple packages for the computation of adaptive boosting ensembles. Relevant examples are provided in the discussions below; the reader is encouraged to read the package vignettes and relevant literature for an in-depth analysis of the relevant functions and corresponding algorithms.
The package `fastadaboost`[22] contains the following functions for adaptive boosting:

[19] https://github.com/scikit-learn/scikit-learn/issues/3774

[20] https://github.com/scikit-learn/scikit-learn/blob/bac89c2/sklearn/ensemble/weight_boosting.py

[21] Cfr. footnote 20.

[22] https://cran.r-project.org/web/packages/fastAdaboost/fastAdaboost.pdf

1. `adaboost` - implementation of `AdaBoost.M1` algorithm from [13];
2. `real_adaboost` - implementation of the SAMME.R algorithm from [36];
3. `fastAdaboost` - fast implementation of both `AdaBoost.M1` and SAMME.R algorithms, with C++ as back end programming language.

Decision trees are selected as weak classifiers; they are grown using `rpart`. On the other hand, the package `ada`[23] contains the function `ada` which encodes one adaptive (stochastic) boosting algorithms in [16]; in particular, the argument `type` allows to select the values 'discrete', 'real' and 'gentle' corresponding to discrete, real and gentle adaptive boosting. Ensemble training procedures can be run leveraging both the exponential and the logistic losses, by specifying values for the `ada` argument `loss` accordingly; we refer to the package vignette for additional details.

Finally, the package `JOUSBoost`[24](we refer to [27] for additional considerations on the methodologies therein contained) contains the function `adaboost`, which computes ensembles via adaptive boosting following the algorithm presented in [14]. Decision trees are grown using the CART algorithm implemented in the `rpart` package.

4 Boosting algorithms: `GradientBoost`

After a description of `AdaBoostClassifier()`, we turn briefly our attention to a specific gradient boosting software implementation. In Python, gradient boosting is implemented in the `sklearn` library by the `GradientBoostingClassifier()` and `GradientBoostingRegressor()` functions[25]; both functions use trees as base learners. Let us focus on classification problems only. `GradientBoostingClassifier()` allows one to select both `loss='exponential'` and `loss='deviance'` losses. In the first case, one retrieves the `AdaBoostClassifier()` algorithm[26]; on the other hand, we remind that `LogitBoost` is a Netwon descent algorithm (and not a gradient descent one); it is implemented in Python in the `logitboost` package[27]. Below we show how the `loss='deviance'` case is handled in `GradientBoostingClassifier()`. Let us consider a gradient boosting step $m \in \{1, M\}$; the binomial deviance loss functional can be written as

$$\begin{aligned} \mathcal{L}_{L_{\text{bin}}}(\mathbf{f}_m) &= \frac{1}{n}\sum_{i=1}^{n} L_{\text{bin}}(Y_i, f_m(\mathbf{x}_i)), \\ L_{\text{bin}}(Y_i, f_m(\mathbf{x}_i)) &= -2\left(Y_i \log(p_i^m) + (1-Y_i)\log(1-p_i^m)\right) \\ &= -2\left(Y_i f_m(\mathbf{x}_i) - \log(1+\exp(f_m(\mathbf{x}_i)))\right), \end{aligned} \tag{4.1}$$

where $p_i^m = \frac{1}{1+\exp(-f_m(\mathbf{x}_i))}$, for all $i \in \{1, \dots, n\}$, where we choose the logit output function. Informally, the binomial loss is equal to '2 * negative log likelihood'. Eq. (4.1) is implemented

[23] `https://cran.r-project.org/web/packages/ada/ada.pdf`

[24] `https://cran.r-project.org/web/packages/JOUSBoost/JOUSBoost.pdf`

[25] `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html`

[26] As specified in the official documentation `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html`

[27] `https://logitboost.readthedocs.io/index.html`

in Python for `GradientBoostingClassifier()`[28] as shown in the following code chunk, where `pred` is **not** the vector of probabilities p_i^m, but of predictions $(f_m(\mathbf{x}_1), \ldots, f_m(\mathbf{x}_n))$ at boosting step m, `.ravel()` is an array manipulation routine and `np` denotes the use of the `numpy` package:

```
def __call__(self, y, pred, sample_weight=None):
        """Compute the deviance (= 2 * negative log-likelihood). """
        # logaddexp(0, v) == log(1.0 + exp(v))
        pred = pred.ravel()
        if sample_weight is None:
            return -2.0 * np.mean((y * pred) - np.logaddexp(0.0, pred))
        else:
            return (-2.0 / sample_weight.sum() *
                    np.sum(sample_weight * ((y * pred) - np.logaddexp(0.0, pred))))
```

In case of weighted training samples, the binomial loss takes weights into account leveraging the parameter `sample_weight` as shown above. If the training samples are all equally weighted, the binomial loss reduces to $\mathcal{L}_{L_{\text{bin}}}$, with L_{bin} in (4.1).
The gradient $\nabla \mathcal{L}_{L_{bin}}(\mathbf{f}_m)$ of the loss functional $\mathcal{L}_{L_{bin}}$ in case of binomial loss (4.1) is then:

$$\nabla \mathcal{L}_{L_{bin}}(\mathbf{f}_m) \quad = \quad -\frac{2}{n}\left(Y_1 - p_1^m, \ldots, Y_n - p_n^m\right).$$

With these considerations we end this short section on the Python `sklearn` gradient tree boosting implementation.

5 Boosting algorithms: `XGBoost`

`XGBoost`, i.e. eXtreme Gradient Boosting, is a highly scalable *tree* boosting algorithm introduced by Chen and Guestrin in [6]. Since its introduction, `XGBoost` has rapidly become one of the most used and best performing boosting algorithms available to the machine learning community. Among most relevant `XGBoost` features we mention:

1. the use of a loss function + regularization formalism for tree boosting;
2. a unified approach to both regression and classification problems: what `XGBoost` computes are scores (or weights) depending on the chosen loss function and regularization parameters;
3. the use of Newton approximations to solve the loss + regularization optimization problem (instead of a first order approach, like in gradient boosting implementations);
4. the possibility of choosing multiple algorithms to grow trees, depending on data volumes, computing infrastructure including procedures to handle sparsity in data and parallelization capabilities[29];
5. an approach to subsampling including both feature space (like in random forest algorithms) and training samples to avoid overfitting, together with regularization and shrinkage.

Before starting with a short introduction to `XGBoost`, we list the most important resources for learning:

[28] The interested reader can check line 762 at `https://github.com/scikit-learn/scikit-learn/blob/7389dba/sklearn/ensemble/gradient_boosting.py#L762`

[29] `http://www.parallelr.com/parallel-computation-with-r-and-xgboost/`

- Official online reference: `http://xgboost.readthedocs.io/en/latest/`.
- Official pdf guide: `https://media.readthedocs.org/pdf/xgboost/latest/xgboost.pdf`.
- Recommended exposition: `https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf`

The following short introduction to `XGBoost` is based on both content and notation in [6], which is obviously recommended for all details. As usual, let us consider the data set $\mathcal{D} = \{(Y_1, \mathbf{x}_1), \ldots, (Y_n, \mathbf{x}_n)\}$, described in Section 2.1, where $\mathcal{Y} = \mathbb{R}$ or $\mathcal{Y} = \{0,1\}$; in other words, the following exposition comprises both regression and binary classification learning problems. A regression or classification/decision tree on $\mathcal{D}$ is a pair (q, T), with $q : \mathcal{X} \to \{1, \ldots, T\}$ denoting the function that maps each data point $\mathbf{x}_i \in \mathcal{X}$ to the positive integer $q(\mathbf{x}_i) \in \{1, \ldots, T\}$ labeling the leaf of the decision tree it belongs to, where the integer $T \geq 2$ denotes to the number of leaves in the tree under consideration.

As discussed previously in this tutorial, and with a minor change in notation, a *tree ensemble* is a linear combination of predictions from base learners

$$\hat{y}_i := \varphi\left(\sum_{m=1}^{M} f_m(\mathbf{x}_i)\right),$$

for all $(\mathbf{x}_i, Y_i) \in \mathcal{D}$, where $f_m \in \mathcal{H} = \{f = f(\cdot, q|T) : \mathcal{X} \to \mathbb{R},\ \mathbf{x}_i \mapsto \omega_{q(\mathbf{x}_i)}\}$, denotes the space of regression or classification/decision trees[30], $\omega_{q(\mathbf{x}_i)}$ is the score of the $q(\mathbf{x}_i)$-th leaf of the tree $f(\cdot, q|T)$, and φ is a function depending on the learning problem at hand. For example, in case of regression problems, $\varphi(x) = x$ or $\varphi(x) = \exp(x)$, while for binary classification φ is the logit function. In other words, for regression type problems, the tree ensemble score $\sum_{m=1}^{M} f_m(\mathbf{x}_i)$ represents the predicted target variable for the data point $\mathbf{x}_i$, while for binary classification problems the tree ensemble score is transformed into probabilities *via* a logit transformation.

`XGBoost` generates a tree ensemble by Netwon approximations of the regularized objective

$$\mathcal{L} = \sum_{i=1}^{n} L(\hat{y}_i, y_i) + \sum_{m=1}^{M} \Omega(f_m), \tag{5.1}$$

where $L(\hat{y}_i, y_i)$ is a differentiable convex loss function and $\Omega(f_m) := \gamma T + \frac{1}{2}\lambda\|\omega\|^2$ is a regularization term to penalize model complexity (in this case, tree complexity due to number of leaves T and the L^2-norm of the leaf scores ω_i's, where $\omega = (\omega_1, \ldots, \omega_T)$ and ω_t denoting the value at leaft $t = 1, \ldots, T$). The choice of the loss function depends on the learning problem at hand. At step $m \in \{1, \ldots, M\}$, the Newton approximation of (5.1) leads to the simplified objective function

$$\tilde{\mathcal{L}}^m = \sum_{i=1}^{n} \left[L(\hat{y}_i^{m-1}, y_i) + g_i f_m(\mathbf{x}_i) + \frac{1}{2} h_i f_m^2(\mathbf{x}_i)\right] + \Omega(f_m), \tag{5.2}$$

where $\hat{y}_i^{m-1} = \sum_{k=1}^{m-1} f_k(\mathbf{x}_i)$ and with

$$\begin{aligned} g_i &= \left.\frac{\partial L}{\partial \hat{y}_i^{m-1}}\right|_{(\hat{y}_i^{m-1}, y_i)}, \\ h_i &= \left.\frac{\partial^2 L}{\partial \hat{y}_i^{m-1} \partial \hat{y}_i^{m-1}}\right|_{(\hat{y}_i^{m-1}, y_i)}, \end{aligned}$$

[30] In [6], the terminology 'regression tree' comprises regression, classification and ranking trees.

denoting the i-th and ii-th component of the gradient and the Hessian matrix of the loss function L evaluated at step $m-1$, respectively.
The optimal score (or weight) ω_j^* of leaf $j \in \{1, \ldots, T\}$ for the tree minimizing (5.2) is (cf. formula (5) in [6])

$$\omega_j^* = -\frac{\sum_{i=1}^n g_i \mathbb{I}[q(\mathbf{x}_i) = j]}{\lambda + \sum_{i=1}^n h_i \mathbb{I}[q(\mathbf{x}_i) = j]}, \tag{5.3}$$

while the tree structure function $q : \mathcal{X} \to T$ is obtained by a greedy algorithm starting from a single leaf and adding branches by evaluating split candidates with loss reduction function given by (cf. formula (7) in [6])

$$\texttt{Loss reduction} = [\texttt{Impurity\_Node} - \texttt{Impurity\_Left\_Child} - \texttt{Impurity\_Right\_Child}] - \gamma,$$

where, with a slight abuse of notation, impurity at node N to be split reads as $\texttt{Impurity\_Node} = -\frac{1}{2}\frac{(\sum_{i\in\texttt{N}}^n g_i)^2}{\lambda+\sum_{i\in\texttt{N}} h_i} + \gamma T$. Similar formulæ hold both for the left and right child nodes.
In summary, XGBoost performs a sequential Newton approximation of the regularized functional (5.1) to boost base learners like in **Algorithm 7**, although in presence of a regularization term $\Omega(f)$, default learning rate $\eta = 1$ and the solution of the minimization problem in **Algorithm 7** line **5** given by the computation of scores (5.3) as well as the search of q through iterative minimization of loss reduction at every split using formula (7) in [6].
As mentioned in the introduction to this section, XGBoost implements multiple tree learning algorithms; the default choice is the computationally demanding *exact greedy algorithm* that enumerates all possible split points for all features selecting the one that leads to the maximum loss reduction; on the other hand, approximate algorithms use percentiles of feature distributions either *globally*, i.e. once per tree learning procedure, or *locally*, i.e. at each proposed split during the tree learning procedure. Also for this topic we refer to the original paper [6] for all details. XGBoost can deal only with numerical data columns[31]; therefore, encoding of categorical variables is needed.

5.1 XGBoost in Python and R

Information on the official XGBoost Python implementation is contained in the official guide

https://xgboost.readthedocs.io/en/latest/python/python_intro.html

XGBoost has been ported also in R; the official guide can be accessed at

https://xgboost.readthedocs.io/en/latest/R-package/index.html

In this section, we will collect selected information on the sklearn Python API for XGBoost, as this will be used for the case study presented later in this tutorial. XGBClassifier()[32] is the default sklearn Python API XGBoost function for classification problems; in its default state it shows the following list of parameters

[31] See, for example, https://cran.r-project.org/web/packages/xgboost/vignettes/discoverYourData.html

[32] https://xgboost.readthedocs.io/en/latest/python/python_api.html#xgboost.XGBClassifier

```
XGBClassifier(base_score=0.5, booster=’gbtree’, colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=3, min_child_weight=1, missing=None, n_estimators=100,
       n_jobs=1, nthread=None, objective=’binary:logistic’, random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1)
```

Let us comment the most relevant ones, referring to the aforementioned documentation for all details. The string `booster=‘gbtree’` specifies that the booster under use is the default `‘gbtree’`. The different tree learning algorithms can be specified using the `tree_method` tree parameter to be specified beforehand[33]. By default, the exact greedy algorithm is used for learning; we will use it for the case study in this tutorial.

We continue with `learning_rate` and `n_estimators`, which denote the learning rate to prevent overfitting and the number of boosting iterations, as already seen for `AdaBoostClassifier()`. We note that the default formulation of `XGBClassifier()` proposes a stronger shrinkage and a higher number of estimators with respect to the default `AdaBoostClassifier()`. Default depth of trained trees is specified through `max_depth=3`: default `XGBoostClassifier()` does not learn tree stumps. This is another difference with respect to default `AdaBoostClassifier()`, where `max_depth=1` is used, instead. Global subsampling ratio of features (which occurs only once) is controlled by `colsample_bytree`; on the other hand, `colsample_bylevel` controls local subsampling ratio occurring at every tree split during learning; both parameters lie in $(0, 1]$. The subsampling ratio of training samples is specified by `subsample`, instead. Subsampling of both features and training samples are useful to avoid overfitting and speed-up computations: we will use them when accessing `XGBoost` in the proposed case study.

6 Case study: Porto Seguro's Safe Driver Prediction Competition

We provide a case study that compares the performance of different boosting algorithms using data from the Porto Seguro's Safe Driver Prediction competition hosted on Kaggle's platform. In this chapter we describe the case study and we highlight the machine learning pipeline which is discussed in the Jupyter notebook 'Boosting_PS_Case_Study.ipynb' on the GitHub repository of the Fachgruppe "Data Science" of the Swiss Association of Actuaries (SAV), which is available at

`https://github.com/JSchelldorfer/ActuarialDataScience`

This exposition is meant to be a self-study tool to approach an end-to-end machine learning pipeline including boosting algorithms; therefore, we avoid to setup distributed computing architectures which allow for a more extensive analysis and, for example, more complete hyperparameter tuning procedures[34]. We invite the reader to consider the tool as a starting point for further in-depth learning experiences: for this purpose, we recommend the monographs [28], [22] and [29].

[33] As discussed on this page `https://xgboost.readthedocs.io/en/latest/parameter.html`

[34] All computations have been performed on a laptop computer, with Intel(R) Core(TM) i7-8550U CPU 1.80GHz 1.99GHz and 16GB RAM.

6.1 Getting started

6.1.1 Python and Jupyter Notebook

First of all, Python 3.X.Y has to be installed on the working machine. We recommend to install it *via* the Anaconda platform[35]. Secondly, we need to access the Jupyter Notebook from the Fachgruppe "Data Science" GitHub by downloading it into a folder of preference. We continue by creating an Anaconda environment (e.g. called `boosting`) comprising of the Python packages used in the notebook (it is recommended to install them all together for a better management of depencences) and activate it. The installation of the package `xgboost` is notoriously non-easy[36]. One possibility is to install it using Anaconda by the command

```
conda install -c anaconda py-xgboost=0.60
```

which installs `xgboost` version 0.60. Another possibility is to run the cell

```
# importing xgboost
import pip
pip.main(['install', 'xgboost'])
```

in the notebook, before loading other packages. Additional information on installation options can be found on the `XGBoost` official guide[37]. Finally, by typing the command

```
jupyter notebook
```

in the Anaconda Prompt we activate the notebook. For additional information on how to perform the environment management tasks described above we refer to the Anaconda guide

`https://conda.io/docs/user-guide/tasks/manage-environments.html`

6.1.2 Kaggle data

Kaggle data for the Porto Seguro's Safe Driver Prediction competition can be accessed from

`https://www.kaggle.com/c/porto-seguro-safe-driver-prediction/data`

Once saved in a folder of preference, the analysis on the Jupyter Notebook can start. Before that, we need a short introduction to the Porto Seguro competition[38] to properly frame the machine learning exercise.

6.2 Introduction to the Porto Seguro's Safe Driver Prediction Competition

The Porto Seguro's Safe Driver Prediction competition has been hosted on the Kaggle platform in 2017. The goal is to predict the probability that a driver will cause a car insurance claim in the next year, by training machine learning algorithms on an anonymized data set with a binary target variable provided by Porto Seguro. In other words, not only the records, but also the meaning of the features have been anonymized. Due to the peculiar Kaggle competition setting, non trivial constraints characterize the data analytics and machine learning routines; we collect the most relevant ones in the short list below.

[35] `https://www.anaconda.com/`

[36] See `https://stackoverflow.com/questions/35139108/how-to-install-xgboost-in-anaconda-python-windows-platform`

[37] `https://xgboost.readthedocs.io/en/latest/build.html`

[38] In the remainder of this tutorial, we will often abbreviate 'Porto Seguro's Safe Driver Prediction Competition' into 'Porto Seguro competition', for sake of readability.

- **Planning of predictive modeling**

The business and technical motivations behind the necessity to predict the probability of a car claim, together with both the data pipeline and performance measurement until deployment of the 'best' model (where 'best' refers to criteria that are undisclosed, as well) are not known. Selection of a model with respect to complexity and in function of end users' profiles is not possible. Criteria or desiderata on explainability and transparency of the models, description of existing models as well as lesson learned from previous data science initiatives are not available for the purpose of the current analysis.

- **Data pipeline**

Aggregation processes of different data sources, data preprocessing, feature engineering and time-frame specifications for both features and target variable are not known. The use of additional sources of information to augment the competition data set is not possible.

- **Anonymization of features**

Anonymization of features results in relevant limitations on business-driven considerations for feature preprocessing, engineering and selection. Considerations on representativity of proposed data set with respect to motor insurance business portfolios are not possible.

- **Error type I vs. type II and performance measures**

The lack of business-driven information on the predictive modeling exercise does not allow us to infer much on the errors arising in the binary classification problem. Therefore, the introduction of ad-hoc performance measures aimed at penalizing selected error types or customized weighting in model estimation is not possible.

According to the Porto Seguro competition public leaderboard[39] 5'169 teams of data scientists participated to the competition; submitted predictions have been evaluated by the competition hosts using an *ad-hoc* performance measure, called Normalized Gini Coefficient[40]. Table 1 provides an overall summary of submission scores, according to the aforementioned measure. The number of entries, i.e. the number of submissions per team is also shown.
We note that the score difference among the top three teams are small; the number of entries is high, suggesting that a considerable effort in fine tuning the learning algorithms to further climb the leaderboard ranks has been made. It is also worthy to mention that the top three solutions are the results of ensembling multiple algorithms, from both neural networks and boosting families most probably trained on high performance computing infrastructures. For additional details we refer to the competition Kaggle kernels[41], which are an excellent source of learning material.

The reader may argue how could a use case on fully anonymized data stemming from a highly competitive machine learning competition be useful for learning boosting methodologies.
Given the above limitations, we still consider the proposed exercise as appropriate for an audience of practitioners approaching the *art* of machine learning and boosting by framing it in a purely

[39] https://www.kaggle.com/c/porto-seguro-safe-driver-prediction/leaderboard

[40] https://www.kaggle.com/c/ClaimPredictionChallenge/discussion/703

[41] https://www.kaggle.com/c/porto-seguro-safe-driver-prediction/kernels

Public Leaderboard Rank	Normalized Gini Coefficient - Score	Entries
1	0.29154	83
2	0.29034	293
3	0.28993	252
...		
2’591	0.28	22
...		
3’525	0.27	3
...		
3’846	0.26	8
...		
5’169	-0.24553	1

Table 1: Kaggle Porto Seguro competition: public leaderboard final scores

performance driven-context, i.e. structuring the data analysis and the subsequent modeling around the chase of improvements of out-of-sample performance values.

This framing sacrifices interpretability of post-modeling results (which is compromised already by the presence of anonymized features and, in a real world scenario, it is an iterative exercise seeing the collaboration of different experts, including data scientists responsible for modeling), but given the end-to-end structure of the machine learning pipeline presented in the notebook, it nonetheless constitutes a useful ‘sand box’ to test an array of boosting algorithm implementations without having as goal the deployment of a fully functional and agreed-on predictive model.

We have also to mention that an one-to-one comparison between the results of the machine learning pipelines in this case study and those presented in Table 1 is not possible; models are evaluated on different test data sets. In fact, all submissions of teams participating to the competition are evaluated on a test data set that is not disclosed by the organizers of the competition.

We structure the remainder of the paper as follows; each section of the notebook is backed-up by a corresponding section in this document, and viceversa. In this tutorial we just summarize and provide background of the content for all sections in the notebook, where Python code to replicate those results is presented, instead.

6.3 Analysis

In what follows, we summarize the main sections of the notebook by providing some comments to results and analysis; we recommend to use the notebook and come back to this document whenever needed to compare findings and produced results.

6.3.1 Getting started with Python and Jupyter Notebook

In this section of the notebook we introduce some settings and we load the Python packages needed for the machine learning exercise.

6.3.2 Data import

This section is only about data import; the data set comprises of 595'212 records and 59 columns; no duplicate records are found. The variable `target` is the target/label variable used to train the classifiers; the unique identifier of data samples is denoted by `id`. No duplicate records have been detected. In Table 2 we show the complete list of columns in provided data.

#	Column Name	#	Column Name	#	Column Name
1	id	21	ps_reg_01	41	ps_calc_02
2	target	22	ps_reg_02	42	ps_calc_03
3	ps_ind_01	23	ps_reg_03	43	ps_calc_04
4	ps_ind_02_cat	24	ps_car_01_cat	44	ps_calc_05
5	ps_ind_03	25	ps_car_02_cat	45	ps_calc_06
6	ps_ind_04_cat	26	ps_car_03_cat	46	ps_calc_07
7	ps_ind_05_cat	27	ps_car_04_cat	47	ps_calc_08
8	ps_ind_06_bin	28	ps_car_05_cat	48	ps_calc_09
9	ps_ind_07_bin	29	ps_car_06_cat	49	ps_calc_10
10	ps_ind_08_bin	30	ps_car_07_cat	50	ps_calc_11
11	ps_ind_09_bin	31	ps_car_08_cat	51	ps_calc_12
12	ps_ind_10_bin	32	ps_car_09_cat	52	ps_calc_13
13	ps_ind_11_bin	33	ps_car_10_cat	53	ps_calc_14
14	ps_ind_12_bin	34	ps_car_11_cat	54	ps_calc_15_bin
15	ps_ind_13_bin	35	ps_car_11	55	ps_calc_16_bin
16	ps_ind_14	36	ps_car_12	56	ps_calc_17_bin
17	ps_ind_15	37	ps_car_13	57	ps_calc_18_bin
18	ps_ind_16_bin	38	ps_car_14	58	ps_calc_19_bin
19	ps_ind_17_bin	39	ps_car_15	59	ps_calc_20_bin
20	ps_ind_18_bin	40	ps_calc_01		

Table 2: Kaggle Porto Seguro competition: columns in provided data set

6.3.3 Structural data analysis

This is a section of structural checks on the feature space and missing values. As provided in the official data description of the Porto Seguro competition[42] one has:

- Features that belong to similar groupings are tagged as such in the feature names (e.g., `ind, reg, car, calc`);
- Feature names include the postfix `bin` to indicate binary features and `cat` to indicate categorical features;
- Features without these designations are either continuous or ordinal;
- Values of -1 indicate that the feature was missing from the observation.

[42] `https://www.kaggle.com/c/porto-seguro-safe-driver-prediction/data`

The target column `target` shows whether or not a claim has been filed for that policyholder. It is a binary variable (but not an input variable for modeling). In Table 3 we collect a summary of missing values in the data. Both features `ps_car_03_cat` and `ps_car_05_cat` are characterized by a high percentage of missing values. Approaches for missing data imputation are discussed in Section 6.3.6.

Variable	Number of Missing Entries	Percentage of Missing Entries
`ps_ind_02_cat`	216	0.04
`ps_ind_04_cat`	83	0.01
`ps_ind_05_cat`	5’809	0.98
`ps_reg_03`	107’772	18.11
`ps_car_01_cat`	107	0.02
`ps_car_02_cat`	5	< 0.01
`ps_car_03_cat`	411’231	69.09
`ps_car_05_cat`	266’551	44.78
`ps_car_07_cat`	11’489	1.93
`ps_car_09_cat`	569	0.10
`ps_car_11`	5	< 0.01
`ps_car_12`	1	< 0.01
`ps_car_14`	42’620	7.16

Table 3: Missing values

6.3.4 Univariate analysis

In order to facilitate the subsequent analysis, in this section we introduce the encoding of the feature data type by means of meta-information, resulting into `target`, `id`, `categorical`, `interval`, `ordinal` features.

We continue by proposing a univariate analysis (frequency tables, bar plots and histograms); this is followed by a short discussion of `target` class imbalance, which ends the section. We collect below selected considerations on the above points.

Binary variables Among the 17 (excluding `target`) binary variables in the data set, those with highest class imbalance are collected in Table 4.

Binary Variable	% of 1’s
`ps_ind_10_bin`	0.04
`ps_ind_11_bin`	0.17
`ps_ind_12_bin`	0.94
`ps_ind_13_bin`	0.09

Table 4: Binary features: highest class imbalances

Categorical variables The 14 categorical variables show different number of distinct levels; for example `ps_car_11_cat` comprises of 104 levels. Tabulation shows strong skewness of the level distribution for `ps_ind_05_cat`, `ps_car_04_cat`, `ps_car_02_cat`, `ps_car_07_cat`, `ps_car_10_cat` with predominance of the 0 level for the first two variables, and of the 1 level for the remaining three, as well as a high percentage of missing values for `ps_car_03_cat` and `ps_car_05_cat` (as shown in Table 3). Moreover, although encoded as categorical, `ps_car_08_cat` is binary; similarly, `ps_car_07_cat`, `ps_car_05_cat`, `ps_car_03_cat`, `ps_car_02_cat` and `ps_ind_04_cat` are binary as well, once the missing values are removed. The original categorical encoding is kept for the purpose of the forthcoming analysis.

Interval or numerical variables Interval or numerical variables are characterized by quite different distributions; in fact some variables are semi-continuous, while others are discretized. Therefore, we do produce both bar charts and histograms and we refer to the notebook for all visualizations. Here we just note that the `calc` variables `ps_calc_01`, `ps_calc_02`, `ps_calc_03` show an highly homogeneous distribution and that inverse engineering of `car` variables (which, by definition are vehicle-related features) is in principle possible: we show an example in the notebook by considering `ps_car_15`, which one could suggest to be related to vehicle manufacture age. Moreover, the variables `ps_calc_01`, `ps_calc_02`, `ps_calc_03`, `ps_reg_01` and `ps_reg_02` show 10 distinct values denoted by $0.0, 0.1, \cdots, 0.9$. We could argue that they correspond to deciles of a given *a priori* distribution.

Ordinal variables The 16 ordinal variables are all 11 `calc` variables (from `ps_calc_04` to `ps_calc_14`), and `ps_ind_01`, `ps_ind_03`, `ps_ind_14`, `ps_ind_15` and `ps_car_11`. Considerations on the monotonicity of the levels and visualizations are contained in the notebook.

The target variable `target` The target variable `target` shows a highly imbalanced level distribution, as shown in Table 5. Class imbalance is a quite common feature of data analysis projects from different domains, and it often encodes important characteristics of the problem domain itself. The level of class imbalance could depend on different business-driven considerations, e.g. the Line of Business with low claims frequency under consideration.

General strategies to deal with class imbalance—in this case for a classification problem—suggest to apply statistical methods on data like under/over-sampling or sample weighting, select performance measures alternative to accuracy for modeling and introduce cost-sensitive training. An interesting overview of modern methodologies (with a rather different level of sophistication) is provided in [18].

Due to the constraints of the current analysis highlighted at the beginning of this section, no business-driven considerations on the target variable imbalance and performance measure engineering can be performed. Therefore, class imbalance will be treated only from a purely statistical and algorithmic point of view.

`target` Level	Percentage
0	96.36
1	3.64

Table 5: `target`: levels and percentages

6.3.5 Data visualization and multivariate analysis

Correlations for interval features There are some strong correlations (i.e. in absolute value > 0.5) between interval variables; they are shown in Table 6.

target Variable	Variable	Pearson Correlation	Spearman Correlation
ps_car_12	ps_car_13	**0.67**	0.66
ps_reg_01	ps_reg_03	0.64	-
ps_car_13	ps_car_15	0.53	**0.68**
ps_reg_03	ps_reg_02	0.52	0.64
ps_reg_01	ps_reg_02	-	0.54

Table 6: Correlations

The calc interval features ps_calc_01, ps_calc_02 and ps_calc_03 show no correlation with all other variables, for both Pearson's and Spearman's correlation analysis. Correlations are computed considering only those records which do not present missing values for both variables in the computation; this becomes particularly relevant when considering ps_reg_03 as it presents 18.11% of missing records.

target vs. features Visualizations and cross-tabulations of features vs. target are provided in the notebook. Here we just mention that the interval variables ps_calc_01, ps_calc_02 and ps_calc_03 show no discrimination with respect to the target variable.

6.3.6 Feature engineering & data preparation for modeling

On calc variables and univariate screening Given the above considerations, we decide to remove the 14 interval and ordinal calc variables to simplify our analysis and speed-up the computations of the modeling procedures, although in absence of documentation to guide feature selection before modeling and given nonetheless the possibility of implementing techniques to allow for multivariate feature selection. Before the removal, the original data set contains 59 features; after removal we retrieve 45 features.

Imputation of missing values The presence of missing values has been already highlighted. Here imputation strategies are taken into account, as casewise record deletion is deprecated; it easily leads to strong bias and high uncertainty in model estimates. For sake of readibility, one starts to recalling the list of all variables with corresponding type and percentage of missing values in Table 3.

In absence of a detailed description of the variables under analysis, business-driven imputation schemata are not feasible. Therefore, a practical (and rather straightforward) approach to imputation is chosen. The categorical variables ps_car_03_cat and ps_car_05_cat are removed due to the high number of missing values; for the remaining categorical variables the missing value -1 is treated as an additional level. For continuous/interval variables (i.e. ps_reg_03, ps_car_12 and ps_car_14) mean imputation is applied; on the ordinal variable ps_car_11 mode imputation is chosen. Of course, alternative imputation schemata based on regressions, kNN models are applicable as well (see [22] for additional details). However, for simplicity only the above straighforward imputation scheme is chosen. Therefore, before imputation we have 45 features; after imputation only 43 remain, as we removed `ps_car_03_cat` and `ps_car_05_cat`.

Dummy Variables As `AdaBoost` and `XGBoost` Python implementations do not handle categorical variables, as noted in Section 3 and 5, dummy coding for categorical features is introduced. The Python function `get_dummies` converts categorical variables into dummy variables dropping both the original categorical variable from which the corresponding dummy variables are created and the first level, as we specified the `drop_first = True` parameter. After creation of dummy variables, we end up with 197 features. We have chosen the dummy encoding for all the categorical features as we do not have any documentation to infer if, for example, label encoding could have been used, as well. Note that all binary and categorical variables in the data set show levels corresponding to non negative integers. So, theoretically, one can avoid dummification and let algorithms run on data without encoding. Of course, handling categorical variables as numerical poses problems to correctness of splitting procedures and interpretability of results. Therefore, in absence of additional information on categorical variables, including binary ones, we proceed with dummy coding.

Training and test data sets For the purpose of modeling, the original data set is randomly split into a training resp. test data set, with a standard 80-20 ratio using the function `train_test_split()`. The resulting data sets are denoted by `X_train` and `X_test`. The random seed is reported for reproducibility of results. A non-stratified approach is chosen. After the split, the training data set consists of 476'169 samples; the test data set of 119'043. Both data sets comprise of 195 features.

6.3.7 Modeling strategies

Modeling strategies: short description We are now ready to define a modeling strategy for all boosting algorithms under consideration. As described in Section 6.3.6, we have randomly split the original data set into the sub-samples `X_train` and `X_test`. This implies that the modeling strategies discussed below follow a single unstratified random split of the original data set. Hence, no modeling with repeated cross-validation on the full data sample is performed, for sake of simplicity and due to high computational time this procedure would necessitate. The overall modeling strategy consists of the following steps:

1. **Baseline modeling.** A baseline model is fitted for benchmarking purposes, on `X_train` data. Performance of the baseline model is analyzed on `X_test`, i.e. test data, instead. *De facto*, baseline models end up to be the boosting algorithms with default choice of parameters. Baseline modeling is denoted by the **ST0** strategy label.
2. **In-depth modeling: hyperparameters tuning.** The baseline model is followed by more in-depth modeling routines, with hyperparameter tuning, cross-validation and out-of-sample performance testing. In-depth modeling strategies are denoted by **ST1**, **ST2**, etc. Models are trained on `X_train`, hyperparameter tuning is performed through `sklearn GridSearch()` function and cross-validation routines. The whole training procedure takes care of the following steps:

- consider a grid of hyperparameters (defined by the strategy under consideration);

- for each vector of the grid, perform K-fold cross-validation on `X_train` and compute performance measures on each of the K-validation subsets;
- compute the average of the performance measure values for the current hyperparameter vector;
- select best hyperparameter vector based on best averaged performance measure;
- fit the model corresponding to the best hyperparameter vector on the whole `X_train` data set.

Lastly, we finish by reporting the performance on `X_test` of the 'best' model fitted on `X_train` and corresponding to the best hyperparameter vector. As already discussed in this tutorial, the adjective 'best' refer to the out-of-sample performance only.

On the Porto Seguro competition performance metric: normalized Gini coefficient

The evaluation of the fitted models on out-of-sample data is performed in the Porto Seguro competition by considering an ad-hoc performance measure, called normalized Gini coefficient, which will be used in the notebook as performance measure on validation subsets, as explained in the preceding section.

The Gini coefficient—also known as Accuracy Ratio—is a metric used in the context of Cumulative Accuracy Profile (CAP) curve analysis of binary classifiers; the CAP curve plots the empirical cumulative distribution of data points associated to 'signal' (i.e. a default in credit scoring, a claim in claim propensity analysis etc.) against the empirical cumulative distribution of all data points in consideration. Usually, the classifier is able to produce scores, or probabilities, which are used to draw the CAP curves, after ranking. The Gini coefficient is given by the ratio

$$\text{Gini}_{\text{CAP}} = \frac{a_R}{a_P},$$

where a_R is the area between the CAP curve of the classifier and the CAP curve of the random classifier, while a_P denotes the area between the CAP curve of the perfect classifier and the CAP curve of the random one. We refer to [9], Chapter 13 for additional details. Geometric considerations show that

$$a_P = \frac{1}{2}(1 - \texttt{freq\_pos}),$$

where `freq_pos` denotes the percentage of records associated to the 'signal' in the provided data set; for example, on test data `y_test` one would have

```
freq_pos=y_test[y_test==1].count()/y_test.count()
```

where the 'signal'—or a claim, in our case study—is characterized by the value 1 (therefore the selection of the samples with `y_test==1`).
Similarly, one can prove that $\text{Gini}_{\text{CAP}} = 2\text{AUC} - 1$, where AUC denotes the Area Under Curve of the binary classifier. Again, we refer to [9] for additional details. The code for the computation of the normalized Gini coefficient is provided in the notebook; the code computes the coefficients `gini(actual, pred)` and `gini_normalized_score(actual, pred)`, which are introduced in the next paragraph.

On Gini coefficient, `gini(actual, pred)` and the `gini_normalized_score(actual, pred)` The code to compute both the Gini coefficient `gini(actual, pred)` and the normalized Gini coefficient `gini_normalized_score(actual, pred)` is shown in the notebook; it has been provided for C+ by the Kaggle competition organizers and translated to other software languages, including Python. In this section we present a short analysis of the mathematical relation between a_R, a_P, `gini(actual, pred)` and `gini_normalized_score(actual, pred)`. The reader not interested in these details can skip this section and continue with the case study.

Let D be the $K \times 2$ matrix whose i-th row D_i is given by $D_i = (x_i, p_i)$, where $x_i \in \{0, 1\}$ and $p_i \in [0, 1]$, such that $p_1 \geq \cdots \geq p_K$; moreover, $L = \#\{j \in \{1, \ldots, K\} \mid x_j = 1\}$.

Let us consider the computation of $\text{Gini}_{\text{CAP}} = \frac{a_R}{a_P}$, using the matrix D. Geometrically we have:

$$a_R + \frac{1}{2} = \sum_{i=1}^{K} \frac{1}{2} \frac{1}{K} \left[\sum_{j=1}^{i} \frac{x_j}{L} + \sum_{j=1}^{i-1} \frac{x_j}{L} \right],$$

where the factor $\frac{1}{2}$ on the l.h.s. corresponds to the area under the random classifier. Therefore:

$$\begin{aligned} a_R + \frac{1}{2} &= \sum_{i=1}^{K} \frac{1}{2} \frac{1}{K} \frac{1}{L} \left[\sum_{j=1}^{i-1} 2x_j + x_i \right] = \sum_{i=1}^{K} \frac{1}{KL} \left[\sum_{j=1}^{i-1} x_j \right] + \sum_{i=1}^{K} \frac{1}{2KL} x_i \\ &= \sum_{i=1}^{K} \frac{1}{KL} \left[\sum_{j=1}^{i-1} x_j \right] + \frac{L}{2KL} = \frac{1}{K} \left[\sum_{i=1}^{K} \sum_{j=1}^{i-1} \frac{x_j}{L} + \frac{1}{2} \right]. \end{aligned}$$

Then

$$\begin{aligned} a_R &= \frac{1}{K} \left[\sum_{i=1}^{K} \sum_{j=1}^{i-1} \frac{x_j}{L} + \frac{1}{2} - \frac{K}{2} \right] = \frac{1}{K} \left[\sum_{i=1}^{K} \sum_{j=1}^{i} \frac{x_j}{L} - \sum_{i=1}^{K} \frac{x_i}{L} + \frac{1}{2} - \frac{K}{2} \right] \\ &= \frac{1}{K} \left[\sum_{i=1}^{K} \sum_{j=1}^{i} \frac{x_j}{L} - 1 + \frac{1}{2} - \frac{K}{2} \right] = \frac{1}{K} \left[\sum_{i=1}^{K} \sum_{j=1}^{i} \frac{x_j}{L} - \frac{1+K}{2} \right], \end{aligned}$$

arriving at

$$a_R = \texttt{gini(x,p)},$$

where `x`= $(x_1, \ldots, x_K)$ and `p`= $(p_1, \ldots, p_K)$ denote the columns of D. In the context of the case study, the elements of the ordered vector `p` are the empirical probabilities computed by the classifier under consideration, while `x` encodes the corresponding labels. As

$$a_P = \texttt{gini(x, x)}$$

by definition of the perfect model, and $\text{Gini}_{\text{CAP}} = \frac{a_R}{a_P}$ = `gini_normalized_score(x,p)`, then it follows

$$\texttt{gini\_normalized\_score(x,p)} = 2\text{AUC} - 1.$$

In summary, there is an affine transformation between `gini_normalized_score(x,p)` and the Area Under Curve; therefore, one can report any of the two measures when considering, for example, out-of-sample performance of any given classifier.

6.3.8 Modeling: `AdaBoost`

We start by using the Python `sklearn` implementation for `AdaBoost`. We present the baseline strategy **ST0** followed by strategies **ST1** and **ST2**.

1. **ST0: default `AdaBoost` with `AdaBoostClassifier()`.**
In **ST0** we choose the tree stumps as base learners and we apply both SAMME and SAMME.R algorithms. By default, `AdaBoostClassifier()` performs 50 boosting iterations and selects the learning rate `learning_rate=1`. Results show that SAMME.R `AdaBoost` outperforms SAMME `AdaBoost`; in the former case we have an out-of-sample (i.e on test data) AUC equal to 0.635, while in the latter we arrive at an out-of-sample AUC equal to 0.612.
As a side note, we remark that AUC=0.635 is equivalent to a normalized Gini coefficient `gini_normalized_score(x,p)` = 0.27; such score corresponds to the top 68% of the Public Leaderboard shown in Table 1 (although we have discussed already that an one-to-one comparison with the public leaderboard is not possible). This is not a surprising result, considering we have just used a boosting algorithm in its default implementation, without any hyperparameter tuning or additional analysis.

2. **ST1 or *interlude*: comparison between SAMME and SAMME.R**
In **ST1** we perform a comparison between SAMME and SAMME.R algorithms, by increasing `n_estimators` (which is equivalent to increase the number of boosting iterations) up to 500 and considering different tree depths, with default `learning_rate=1` (i.e. no shrinking is applied). Strategy **ST1** is adapted from an analysis presented on the official `sklearn` documentation[43]. Figure 1 collects all results of the analysis: we plot the AUC on training and test data for both SAMME and SAMME.R as function of the number of boosting iterations. In presence of shallow trees (i.e. `max_depth=1`), SAMME.R consistently outperforms SAMME; the maximum value of the AUC on test data for SAMME.R is 0.639; it is reached at `n_estimators=267` iterations. However, increasing the depth of trees by selecting `max_depth=3`, SAMME.R overfits after few boosting iterations. The maximum values of the AUC on test data for SAMME is 0.624 reached at `n_estimators=485` iterations, while for SAMME.R the maximum value of the AUC on test data is 0.63, but reached at only `n_estimators=8` iterations. A third run of SAMME and SAMME.R in presence of `max_depth=5` base learners confirms the severe overfitting of SAMME.R after few boosting iterations. Therefore, increasing the base learner complexity (`max_depth`) and considering a wide range of boosting iterations, shrinking is needed to control overfitting. This is performed in strategy **ST2**.

3. **ST2: hyper parameter tuning: `n_estimators`, `max_depth` and `learning_rate`**
Motivated by **ST1**, we search for the optimal out-of-sample performance in a trade-off between base learner complexity (`max_depth`), boosting iterations (`n_estimators`) and shrinking (`learning_rate`) in three separate runs. We choose the SAMME.R algorithm for simplicity, collecting results in Table 7. We consider a rather simple grid of hyperparameters for each run, due to the highly time-consuming computations performed. As shown by run I., tree stumps overperform deeper trees, but in presence of considerably more boosting iterations: the maximum out-of-sample AUC among all runs is reached by tree stumps with mild learning rate `learning_rate`=0.1 and `n_estimators`=400.

[43] https://scikit-learn.org/stable/auto_examples/ensemble/plot_adaboost_hastie_10_2.html#sphx-glr-auto-examples-ensemble-plot-adaboost-hastie-10-2-py

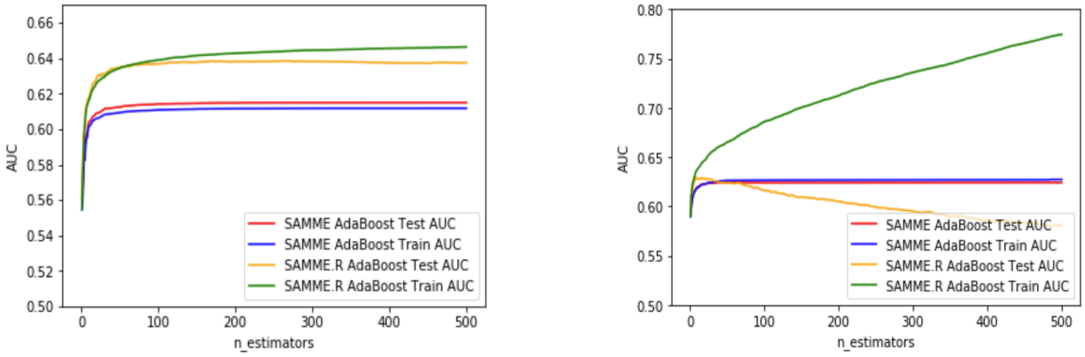

Figure 1: Comparison of AUC on training and test data for both SAMME and SAMME.R algorithms, with different base learner complexity (on the left: `max_depth=1`; on the right: `max_depth=3`).

Run	`max_depth`	`learning_rate` and `n_estimators`	Best hyperparameters	AUC
I.	1	[0.001, 0.01, 0.1, 1] [100, 200, 300, 400]	(0.1, 400)	**0.637**
II.	3	[0.001, 0.01, 0.1, 1] [100, 200, 300, 400]	(0.1, 100)	0.634
III.	3	[0.1, 0.15, 1] [50, 75, 100]	(0.15, 75)	0.634

Table 7: **ST2**: `AdaBoost`: comparison of hyperparameter tuning results.

Run I. outperforms the default `AdaBoost` in **ST0**, although run II. and III. do not. However, SAMME.R in **ST1** (for `max_depth=1` base learners) outperforms SAMME.R in **ST2**, run I.

6.3.9 Modeling: `XGBoost`

We turn now our attention to `XGBoost`, starting with the Python `sklearn` implementation for `XGBoost`. We present the baseline strategy **ST0** followed by **ST1**.

1. **ST0: default `XGBoost` with `XGBClassifier()`.** The default `XGBClassifier()` function has been discussed in Section 5.1; we remind that the learning rate is equal to 0.1 and max tree depth is equal to 3. The number of estimators, or boosting rounds, is equal to 100. Default `XGBClassifier()` delivers an out-of-sample AUC = 0.638. It already outperforms `AdaBoost` **ST0** and all the `AdaBoost` strategies of Table 7, but not the best SAMME.R from `AdaBoost` **ST1**.

2. **ST1: hyper parameter tuning: `max_depth`, `learning_rate`, `n_estimators` with fixed subsampling.** Similarly to what we did in **ST2** for `AdaBoost`, we implement a `GridSearchCV()` function with tuning of `max_depth`, `learning_rate`, `n_estimators`. The novelty is to use the `XGBoostClassifier()` subsampling capabilities by fixing feature subsampling to the value `colsample_bytree=0.75` and data sample subsampling to `subsample=0.5`; hence, we drop 25% of features and 50% of training data points at each tree learning routine to avoid overfitting (we remind that the `X_train` is characterized by a high number of features due to dummy coding); the subsampling of training data allows for quicker computations. The interested reader can replicate **ST1** for `XGBoost` using different feature and data sample subsampling ratios and compare results.

As shown in Table 8, we consider 3 different runs, each characterized by different max_depth values and the same grid for learning_rate and n_estimators. The idea was to fine tune the max_depth parameter, by identifying the value that delivers the highest out-of-sample performance, without keeping the tree complexity unnecessarily high.

The best configurations are reached in runs II. and III.; they outperform all boosting procedures so far with an out-of-sample AUC equal to 0.643. This value is equivalent to a Gini normalized coefficient gini_normalized_score(x,p) = 0.286 which is, heuristically, a score in the the top 24% of the public leaderboard shown in Table 1. Run II. is characterized by max_depth=2 and n_estimators= 500; increasing tree depth (i.e. moving to run III.) allows to reduce the number of boosting iterations considerably and keep the same out-of-sample performance. We note that for both runs, a moderate (default) shrinking through learning_rate=0.1 is applied. Run IV. shows that an increase of tree depth does not deliver any improvement in out-of-sample performance, in the given hyperparameter grid.

Run	max_depth	learning_rate and n_estimators	Best hyperparameters	AUC
I.	1	[0.001, 0.01, 0.1, 1], [100, 300, 500]	(1, 0.1, 500)	0.639
II.	2	[0.001, 0.01, 0.1, 1], [100, 300, 500]	(2, 0.1, 500)	**0.643**
III.	(3, 4)	[0.001, 0.01, 0.1, 1], [100, 300, 500]	(3, 0.1, 300)	**0.643**
IV.	5	[0.001, 0.01, 0.1, 1], [100, 300, 500]	(5, 0.1, 100)	0.641

Table 8: **ST2**: Comparison of hyperparameter tuning results. We report the triples (max_depth, learning_rate, n_estimators) under 'Best hyperparameters'. All runs are characterized by colsample_bytree=0.75 and subsample=0.5.

In Figure 2 we show the top 10 feature importance values as computed by the best XGBoost algorithm in run III., i.e. the one characterized by max_depth=3, learning_rate=0.1 and n_estimators= 300. For all details on the computation of feature importance by XBGboost we refer to the official documentation[44]. The plot shows a homogeneous distribution of variables from the car, reg and ind groups. We note that no dummy variable appears in the plot and that ps_car_13 and ps_reg_03 show the highest importance values. We also remind that both ps_reg_03 and ps_car_14 have been pre-processed through mean imputation (as they originally showed 18% and 7% of missing values). We leave the analysis of the dependence of feature importance on imputation schemata to further studies.

7 Conclusions

We have provided the reader with an overview of the two most common boosting algorithms, i.e. AdaBoost and XGBoost. We have described how different boosting algorithms are related by providing multiple examples from the literature, and we introduced the statistical theory of boosting as well as gradient/Newton approximation routines. A discussion on the implementation of AdaBoost and XGBoost in Python (including their limitations) followed. Finally, we have applied different boosting algorithms in a case study stemming from a Kaggle competition hosted

[44] https://stats.stackexchange.com/questions/162162/relative-variable-importance-for-boosting

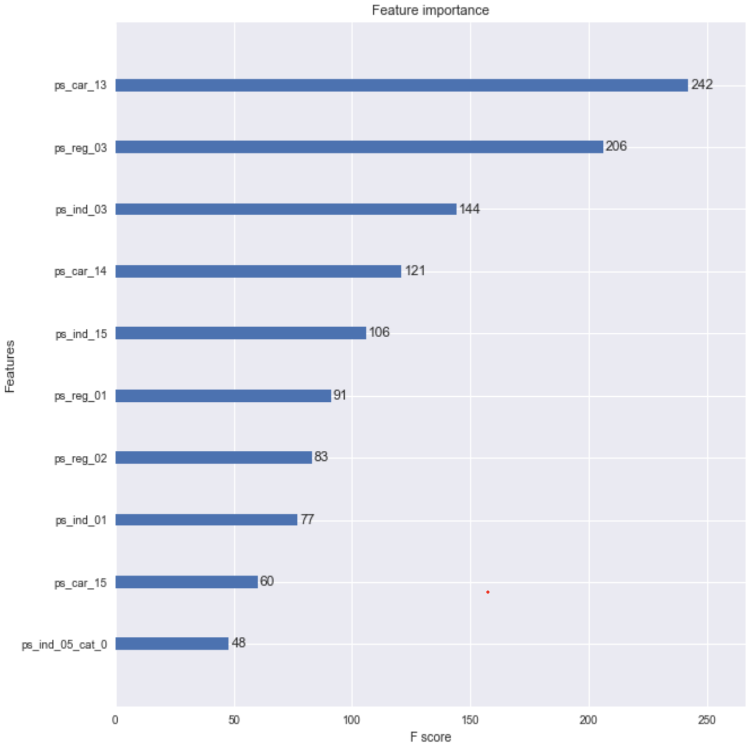

Figure 2: `XGBoost` best configuration in run III.: feature importance

by Porto Seguro and based on claims propensity modeling. The case study results in a machine learning classification problem. We have highlighted the limitations of the case study and its 'performance driven' setup, providing the reader with a notebook to fully reproduce results and extend the analysis, if needed. Empirical results show that SAMME.R `AdaBoost` outperforms SAMME `AdaBoost`; SAMME.R `AdaBoost` reaches its highest performance with tree stumps, moderate shrinking and an increased number of boosting iterations with respect to default (i.e. 400 iterations vs. 50). On the other hand, default `XGBoost` outperforms most `AdaBoost` strategies. Overall, the highest performance is reached by `XGBoost` routines with shallow trees (i.e. with tree depth equal to 2 or 3), moderate (default) shrinking, number of iterations increased with respect to default (i.e. 300 iterations vs. 100) and subsampling of both features and training data points.

7.1 Acknowledgment.

We would like to kindly thank Jürg Schelldorfer (Swiss Re) and Mario Wuthrich (ETH Zurich) for detailed comments that have helped us to substantially improve this tutorial.

8 Appendix

8.1 Proof of Lemma 2.1

We start with noting that $D_m(i) > 0$ and $Z_m > 0$ for all $m \in \{1, \ldots, M\}$ and $i \in \{1, \ldots, n\}$, as it can be easily proven by induction. We are left to prove that $D_{m+1}(i) \leq D_m(i)$ for all i's such that $h_m(x_i) = Y_i$; this is equivalent to prove

$$D_{m+1}(i) = \frac{\beta_m}{Z_m} D_m(i) < D_m(i) \;\Leftrightarrow\; \beta_m < Z_m.$$

As the misclassification error ϵ_m of h_m is such that $0 < \epsilon_m < \frac{1}{2}$, then $0 < \beta_m < 1$; therefore we arrive at

$$\begin{aligned} Z_m &= \sum_{k:h_m(\mathbf{x}_k)=Y_k} D_m(k)\beta_m + \sum_{k:h_m(\mathbf{x}_k)\neq Y_k} D_m(k) > \sum_{k:h_m(\mathbf{x}_k)=Y_k} D_m(k)\beta_m + \sum_{k:h_m(\mathbf{x}_k)\neq Y_k} D_m(k)\beta_m \\ &= \sum_{k=1}^{n} D_m(k)\beta_m = \beta_m, \end{aligned}$$

where last equality follows by normalization of the weights $D_m(k)$. □

8.2 Proof of Theorem 2.3

We want to prove that **Algorithm 3** can be rewritten as **Algorithm 7**, in presence of logisitc loss. To do so, let assume that we are at step $m \leq M$, with estimated function $\mathbf{f}_m$; the loss functional (2.11) is

$$\mathcal{L}_L(\mathbf{f}_m) = \mathcal{L}_L(f_{m,1}, \ldots, f_{m,n}) = \frac{1}{n}\sum_{i=1}^{n} \log(1 + \exp(-Y_i f_{m,i})),$$

by the definition of the logistic loss. The gradient $\nabla\mathcal{L}_L(\mathbf{f}_m)$ and the Hessian $\nabla^2\mathcal{L}_L(\mathbf{f}_m)$ of L at $\mathbf{f}_m$ read as

$$\begin{aligned} \nabla\mathcal{L}_L(\mathbf{f}_m) &= -\frac{1}{n}\left(\frac{Y_1}{1+\exp(Y_1 f_{m,1})}, \ldots, \frac{Y_n}{1+\exp(Y_n f_{m,n})}\right), \\ \nabla^2\mathcal{L}_L(\mathbf{f}_m) &= \frac{1}{n}\begin{pmatrix} \frac{\exp(Y_1 f_{m,1})}{(1+\exp(Y_1 f_{m,1}))^2} & & 0 \\ & \ddots & \\ 0 & & \frac{\exp(Y_n f_{m,n})}{(1+\exp(Y_n f_{m,n}))^2} \end{pmatrix}, \end{aligned}$$

where we used $Y_i^2 = 1$, for all i's. Again, as $Y_i \in \{-1, +1\}$, we arrive at

$$(\nabla^2\mathcal{L}_L(\mathbf{f}_m))_{ii} = \frac{1}{n}\frac{\exp(-f_{m,i})}{(1+\exp(-f_{m,i}))^2},$$

i.e. $(\nabla^2\mathcal{L}_L(\mathbf{f}_m))_{ii} = \frac{1}{n}\omega_m(i)$, where $\omega_m(i)$ has been defined in (2.9), in line **3** of **Algorithm 3**. In our notation, the ratio $-\frac{g_m(x_i)}{H_m(x_i)}$ from (2.15) can be written as

$$-\frac{g_m(x_i)}{H_m(x_i)} = -\frac{\nabla_i\mathcal{L}_L(\mathbf{f}_m)}{(\nabla^2\mathcal{L}_L(\mathbf{f}_m))_{ii}},$$

for all $i = 1, \ldots, n$. Therefore, we arrive at

$$-\frac{g_m(x_i)}{H_m(x_i)} = \begin{cases} \frac{1+\exp(f_{m,i})}{\exp(f_{m,i})} & \text{if } Y_i = +1 \\ -\frac{1+\exp(-f_{m,i})}{\exp(-f_{m,i})} & \text{if } Y_i = -1 \end{cases} = \begin{cases} \frac{1}{p_m(\mathbf{x}_i)} & \text{if } Y_i = +1 \\ -\frac{1}{1-p_m(\mathbf{x}_i)} & \text{if } Y_i = -1 \end{cases} = z_m(\mathbf{x}_i),$$

where $p_m(\mathbf{x}_i)$ and $z_m(\mathbf{x}_i)$ are defined in (2.7) and (2.8). Therefore, both procedures on line **4** of **Algorithm 3** and on line **5** of **Algorithm 7** refer to the same second order/Newton approximation problem. □

References

[1] Biau, G., Cadre, B. (2017). Optimization by gradient boosting. Available at `https://arxiv.org/abs/1707.05023`.

[2] Breiman, L. (1998).. Arcing classifiers (with discussion). Annals of Statistics, 26, 801-849.

[3] Breiman, L., Friedman, J., Stone, C.J., Olshen, R.A. (1984). Classification and Regression Trees. Chapman and Hall/CRC.

[4] Bühlmann, P., Hothorn, T. (2007). Boosting algorithms: regularization, prediction and model fitting. Statistical Science 22, 477-505.

[5] Bühlmann, P., van de Geer, S. (2011). Statistics for High-Dimensional Data: Methods, Theory and Applications. Springer, Berlin.

[6] Chen, T., Guestrin, C. (2016). XGBoost: a scalable tree boosting system. Available at `http://www.kdd.org/kdd2016/papers/files/rfp0697-chenAemb.pdf`.

[7] Demiriz, A., Bennett, K.P., Shawe-Taylor, J. (2002). Linear programming boosting via column generation. Machine Learning, 46, 225-254.

[8] Duchi, J., Singer, N. (2009). Boosting with structural sparsity. Proceedings of the 26th International Conference on Machine Learning.

[9] Engelmann, B., Rauhmeier, R. (2006) The Basel II risk parameters: estimation, validation, and stress testing. Springer Berlin Heidelberg New York.

[10] Ferrario, A., Noll, A., Wuthrich, M., V. (2018). Insights from Inside Neural Networks. Available at SSRN: https://ssrn.com/abstract=3226852 or http://dx.doi.org/10.2139/ssrn.3226852

[11] Freund, Y. (1995). Boosting a weak learning algorithm by majority. Inform. and Comput. 121, 256-285.

[12] Freund, Y. (2001). An adaptive version of the boost by majority algorithm. Machine Learning, 43(3):293-318.

[13] Freund, Y., Schapire, R.E. (1996). Experiments with a new boosting algorithm. Machine Learning: Proceedings of the Thirteenth International Conference.

[14] Freund, Y., Schapire, R.E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. J. Comput. System Sci. 55, 119-139.

[15] Friedman, J. (2001). Greedy function approximation: a gradient boosting machine. Annals of Statistics, 29, 1189-1232.

[16] Friedman, J., Hastie, T., Tibshirani, R. (1998). Additive logistic regression: a statistical view of boosting. Annals of Statistics, 28, 337-407.

[17] Friedman, J., Hastie, T., Tibshirani, R. (2009). The Elements of Statistical Learning - Data Mining, Inference, and Prediction. Springer.

[18] He, H., Ma, Y. (2013). Imbalanced Learning: Foundations, Algorithms, and Applications. Wiley Online Library.

[19] James, G., Witten, D., Hastie, T., Tibshirani, R. (2013). An Introduction to Statistical Learning. Springer.

[20] Kearns, M. (1988). Thoughts on hypothesis boosting, unpublished.

[21] Kearns, M., Valiant, L.G. (1989). Cryptographic limitations on learning boolean formulae and finite automata. Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing.

[22] Kuhn, M., Johnson, K. (2013). Applied Predictive Modeling. Springer.

[23] Nielsen, D. (2016). Tree Boosting with XGBoost - Why Does XGBoost Win 'Every' Machine Learning Competition?. MsC Thesis, available at https://brage.bibsys.no/xmlui/handle/11250/2433761.

[24] Nikolaou, N. (2016). Cost-Sensitive Boosting: A unified Approach. PhD Thesis at University of Manchester.

[25] Nikolaou, N., Edakunni, N., Kull, M. et al. (2016). Cost-sensitive boosting algorithms: Do we really need them? Machine Learning 104, 359-384.

[26] Nocedal, J., Wright, S.J. (2006). Numerical Optimization, Springer.

[27] Olson, M. (2017). JOUSBoost: An R package for improving machine learning classifier probability estimates. Available at https://cran.r-project.org/web/packages/JOUSBoost/vignettes/JOUS.pdf.

[28] Raschka, S., Mirjalili, V. (2017). Python Machine Learning. Packt.

[29] Richert, W., Coelho, L.P. (2013). Building Machine Learning Systems with Python. Packt.

[30] Schapire, R. (1990). The strength of weak learnability, Machine Learning 5, 197-227.

[31] Schapire, R.E., Freund, Y. (2012). Boosting: Foundations and Algorithms. MIT Press.

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

[33] Servedio, R.A. (2001). Smooth boosting and learning with malicious noise. In D.P. Helmbold and B. Williamson, Editors, Conference on Learning Theory, Springer.

[34] Warmuth, M.K., Liao, J., Rätsch, G. (2006). Totally corrective boosting algorithms that maximize the margin. In W.W. Cohen and A. Moore, Editors, International Conference on Machine Learning, vol. 148 of ACM International Conference Proceeding Series, ACM.

[35] Wuthrich, M.V., Buser, C. (2016). Data Analytics for Non-Life Insurance Pricing. SSRN Manuscript ID 2870308. Version October 24, 2017.

[36] Zhu, J., Zou, H., Rosset, S., et al. (2006). Multi-class adaboost. Stat. Interface, 2, 349360.