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 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 \(D = \{(Y_1, x_1), \ldots, (Y_n, x_n)\}\), with responses \(Y_i \in \mathcal{Y}\), and \(x_i \in \mathcal{X}\) 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.

In the notation of [32], Chapter 2, \( \mathcal{X} \) and \( \mathcal{Y} \) 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 \( \mathcal{Y} \times \mathcal{X} \).

With \( \mathcal{H} = \{ h : \mathcal{X} \rightarrow \mathcal{Y} \} \) 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 \( m = 1, \ldots, M \), 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 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 [15]; and
- the SAMME.R and SAMME adaptive boosting algorithms in [30].

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 \( \mathcal{Y} = \{1, 2, \ldots, k\}, k \geq 2 \) in the multi-class case.
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 $\text{WeakLearn}$, 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 $D = \{(Y_1, x_1), \ldots, (Y_n, x_n)\}$, with labels $Y_i \in \mathcal{Y} = \{0, 1\}$
- weak learning algorithm $\text{WeakLearn}$
- number of iterations $M \geq 1$

1. Initialize $D_1(i) = \frac{1}{n}$, for all $i = 1, \ldots, n$.
2. while $m \in \{1, \ldots, M\}$ do
   3. Fit $\text{WeakLearn}$ to re-weighted training set $(D, D_m)$.
   4. Return a hypothesis $h_m : \mathcal{X} \rightarrow \mathcal{Y}$ and compute its weighted misclassification error $\epsilon_m = \sum_{i=1}^{n} D_m(i) |h_m(x_i) - Y_i|$.
   5. Set $\beta_m = \frac{\epsilon_m}{1-\epsilon_m}$.
   6. Update distribution $D_m$:
      $$D_{m+1}(i) = \frac{D_m(i)}{Z_m} \beta_m 1^{-|h_m(x_i) - Y_i|},$$
      where $Z_m$ is the normalization constant such that $\sum_{i=1}^{n} D_{m+1}(i) = 1$.
3. end

**Output:** Return the final hypothesis

$$H(x) = \begin{cases} 
1 & \text{if } \sum_{m=1}^{M} \log \left( \frac{1}{\beta_m} \right) h_m(x) \geq \frac{1}{2} \sum_{m=1}^{M} \log \left( \frac{1}{\beta_m} \right) \\
0 & \text{otherwise}
\end{cases} \quad (2.1)$$

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 $\beta_m$ 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 vice versa for the so-called ‘hard examples’ \[13\]. In fact, one has

**Lemma 2.1.** Let \( m \in \{1, \ldots, M\} \) 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 \( i \in \{1, \ldots, n\} \) such that \( h_m(x_i) = Y_i \).

**Proof.** The proof can be found in the Appendix.

The final hypothesis (2.1) can be rewritten as

\[
H(x) = \arg \max_{y \in \{0, 1\}} \sum_{m:h_m(x) = y} \log \left( \frac{1}{\beta_m} \right);
\]

this follows from the identities:

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

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

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 \( \beta_m \) (or, equivalently, the lower the misclassification error \( \epsilon_m \)), 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 \[13\]: 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 \( h_1, \ldots, h_M \) with errors \( \epsilon_1, \ldots, \epsilon_M \). Then, the error

\[
\epsilon_f := \sum_{i=1}^n D_{M+1}(i)|H(x_i) - Y_i|
\]

of the final hypothesis \( H \) in (2.2) satisfies

\[
\epsilon_f \leq 2^M \prod_{m=1}^M \sqrt{\epsilon_m(1 - \epsilon_m)}.
\]

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

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

(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 $\gamma_m$ measures how the improvement over random guessing of the $m$-th weak classifier $h_m$, when considering its weighted misclassification error $\epsilon_m$. 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 $D = \{(Y_1, x_1), \ldots, (Y_n, x_n)\}$, with labels $Y_i \in \mathcal{Y} = \{-1, 1\}$
- weak learning algorithm WeakLearn
- number of iterations $M \geq 1$

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

**Output:** Return the final hypothesis

$$F(x) = \text{sign} \left( \sum_{m=1}^{M} \alpha_m h_m(x) \right). \quad (2.5)$$
We also note that the final hypothesis (2.5) can be rewritten in the form
\[
F(x_i) := \arg \max_{y \in \mathcal{Y}} \sum_{m : h_m(x_i) = y} \alpha_m h_m(x_i),
\]  
(2.6)
as \(h_m(x) = \pm 1\), for all \(m \in \{1, \ldots, M\}\). Moreover, \(\alpha_m > 0 \iff \epsilon_m < \frac{1}{2}\); 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 \(x_i\), 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 \(\mathcal{Y} = \{1, \ldots, k\}, k \geq 2\); 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 \(|h_m(x_i) - Y_i|\) replaced by the indicator function \(\mathbb{I}[h_m(x_i) \neq Y_i]\).

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 $\mathcal{D} = \{(Y_1, x_1), \ldots, (Y_n, x_n)\}$, with labels $Y_i \in \mathcal{Y} = \{-1, 1\}$
- number of iterations $M \geq 1$

1. Initialize $f_0(x) = 0$.
2. while $m \in \{1, \ldots, M\}$ do
3. Compute
   
   $p_m(x_i) = \frac{1}{1 + \exp(-f_{m-1}(x_i))}$, \hspace{1cm} (2.7)

   $z_m(x_i) = \begin{cases} 
   \frac{1}{p_m(x_i)} & \text{if } Y_i = +1, \\
   \frac{1}{1-p_m(x_i)} & \text{if } Y_i = -1, 
   \end{cases}$ \hspace{1cm} (2.8)

   $w_m(i) = p_m(x_i)(1 - p_m(x_i))$. \hspace{1cm} (2.9)

4. Fit $f^*_m(x)$ by a weighted least-squares regression to the responses $z_m(x_i)$ in the features $x_i$ with weights $w_m(i)$.
5. Update $f_m(x) = f_{m-1}(x) + f^*_m(x)$
6. end

Output: Return $f_M(x)$.

A classification rule $C$ leveraging the LogitBoost algorithm can be written as $C(x) = \text{sign}(f_M(x))$; 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 $D = \{(Y_1, x_1), \ldots, (Y_n, x_n)\}$, with labels $Y_i \in \mathcal{Y} = \{1, \ldots, k\}, k \geq 2$.
• set $\mathcal{F}$ of weak classifiers
• learning rate $\eta \in (0, 1]$ 
• number of iterations $M \geq 1$

1. Initialize $\omega_i = \frac{1}{n}$, for all $i = 1, \ldots, n$.
2. while $m \in \{1, \ldots, M\}$ do
3. Fit a weak classifier $h_m$ to the weighted training set $(D, \omega)$, with $\omega = (\omega_1, \ldots, \omega_n)$.
4. Compute the weighted misclassification error $\epsilon_m = \sum_{i=1}^{n} \omega_i \mathbb{I}[Y_i \neq h_m(x_i)]$.
5. Set $\alpha_m = \eta \left( \log \left( \frac{1-\epsilon_m}{\epsilon_m} \right) + \log(k-1) \right)$.
6. Update weights:
   \[ \hat{\omega}_i := \omega_i \exp \left( \alpha_m \mathbb{I}[Y_i \neq h_m(x_i)] \right), \quad i = 1, \ldots, n. \]
7. Re-normalize weights: $\omega_i = \frac{\hat{\omega}_i}{\sum_{i=1}^{n} \hat{\omega}_i}$.
8. end

Output: Return the classifier
\[ H(x) = \arg \max_{y \in \mathcal{Y}} \sum_{m=1}^{M} \alpha_m \mathbb{I}[y = h_m(x_i)]. \]

We note that
\[ \alpha_m > 0 \Leftrightarrow \epsilon_m < 1 - \frac{1}{k}, \quad (2.10) \]
in Algorithm 4, i.e. SAMME allows the misclassification training error $\epsilon_m$ 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 \( D = \{(Y_1, x_1), \ldots, (Y_n, x_n)\} \), with labels \( Y_i \in \mathcal{Y} = \{1, \ldots, K\}, K \geq 2 \).
• set \( \mathcal{F} \) of weak classifiers
• learning rate \( \eta \in (0, 1] \)
• number of iterations \( M \geq 1 \)

1. Initialize \( \omega_i = \frac{1}{n} \), for all \( i = 1, \ldots, n \).
2. while \( m \in \{1, \ldots, M\} \) do
   3. Fit a weak classifier \( h_m \) to the weighted training set \( (D, \omega) \), with \( \omega = (\omega_1, \ldots, \omega_n) \).
   4. Compute the weighted class probability estimates
      \[ p_{m,k}(x) = \sum_{i=1}^{n} \omega_i I[h_m(x) = k], \ k = 1, \ldots, K. \]
   5. Set \( h_{m,k}(x) = (K - 1) \left( \log p_{m,k}(x) - \frac{1}{K} \sum_{j=1}^{K} \log p_{m,j}(x) \right), \ k = 1, \ldots, K. \)
   6. Update weights\footnote{The \( K \)-dimensional vector \( y_i \) is encoded as}
      \[ \hat{\omega}_i = \omega_i \exp \left( -\frac{K - 1}{K} \eta \langle y_i, \log p_m(x) \rangle \right), \ i = 1, \ldots, n. \]
   7. Re-normalize weights: \( \omega_i = \frac{\hat{\omega}_i}{\sum_{i=1}^{n} \hat{\omega}_i} \).
3. end

Output: Return the classifier
\[
H(x) = \arg \max_{k \in \mathcal{Y}} \sum_{m=1}^{M} h_{m,k}(x).
\]

\footnotetext{The \( K \)-dimensional vector \( y_i \) is encoded as}
\[
y_i = \begin{cases} 1 & \text{if } Y_i = k, \\ -\frac{1}{K} & \text{otherwise}. \end{cases}
\]

On the other hand, \( p_m(x) = (p_{m,1}(x), \ldots, p_{m,K}(x)) \). We refer to [36] Section 2.1 for all details.

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 \( L_L : F \to \mathbb{R}_+ \) over a given space of functions \( F \subset \mathbb{R}^X \), where

\[
L_L[F] := \frac{1}{n} \sum_{i=1}^{n} L(Y_i, F(x_i)), \quad F \in F,
\]

for a given loss function \( L \) measuring the cost incurred in predicting the target \( Y_i \) with the prediction \( F(x_i) \), for all \((Y_i, x_i) \in D \). 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:

\[
\begin{align*}
\psi_{\text{logit}}(u) &= \log(1 + \exp(-u)), \\
\psi_{\text{exp}}(u) &= \exp(-u), \\
\psi_{\text{hinge}}(u) &= \max(1 - u, 0).
\end{align*}
\]

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 \( L_L \). To do so, we need to specify a procedure to compute \( F \) at points \( x_1, \ldots, x_n \in X \) 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 \( F := \langle \mathcal{H} \rangle \) 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 \( \alpha_k \) and base learners \( f_k \) of the function \( F = \sum_{k=1}^{M} \alpha_k f_k \in F \) 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.

Let \( f_m \in F_M \), with \( f_m = \sum_{k=1}^{m} \alpha_k f_k \) 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 \( f_m \) is observed only at points \( x_i \in X \) we can write

\[
\begin{align*}
f_m &= (f_{m,1}, \ldots, f_{m,n}) \in \mathbb{R}^n, \\
f_m(x_i) &= f_{m,i} := \sum_{k=1}^{m} \alpha_k f_k(x_i).
\end{align*}
\]
The gradient $\nabla L(f_m)$ of $L_L$ at $f_m$ reads

$$\nabla L(f_m) = \frac{1}{n} \left( \left. \frac{\partial L(Y_1, f_1)}{\partial f_1} \right|_{f_1 = f_{m,1}} , \cdots , \left. \frac{\partial L(Y_n, f_n)}{\partial f_n} \right|_{f_n = f_{m,n}} \right),$$

while the Hessian matrix $\nabla^2 L(f_m) = \left\{ \left( \nabla^2 L(f_m) \right)_{ij} \right\}_{i,j=1,\ldots,n}$ of $L_L$ at $f_m$ is such that

$$(\nabla^2 L(f_m))_{ij} = \frac{1}{n} \delta_{ij} \frac{\partial^2 L}{\partial f_{m,i}^2} (Y_i, f_{m,i}),$$

and denoting by $\delta_{ij}$ the Kronecker delta. Classically, for the gradient descent we consider the approximation around $f_m$

$$L_L(f) = L_L(f_m) + \langle \nabla L(f_m), f - f_m \rangle + o(\|f - f_m\|),$$

and the quadratic approximation

$$L_L(f) = L(f_m) + \langle \nabla L(f_m), f - f_m \rangle + \frac{1}{2} \langle \nabla^2 L(f_m)(f - f_m), f - f_m \rangle + o(\|f - f_m\|^2)$$

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

$$f_m \to f_{m+1} = f_m - \rho \nabla L(f_m), \quad \text{(gradient boosting)} \quad (2.12)$$

$$f_m \to f_{m+1} = f_m - (\nabla^2 L(f_m))^{-1} \nabla L(f_m). \quad \text{(Newton boosting)} \quad (2.13)$$

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 $\nabla L(f_m)$, i.e.

$$h^*_m = \arg \min_{h \in \mathcal{H}, \beta} \sum_{i=1}^n (g_m(x_i) - \beta h(x_i))^2,$$

where $g_m(x_i) := -\left. \frac{\partial L(Y_i, f_{m,i})}{\partial f_{m,i}} \right|_{f_{i}=f_{m,i}}$. The small step size $\rho_m$ to take in the direction of $h^*_m$ is determined by line search

$$\rho_m = \arg \min_{\rho} \sum_{i=1}^n L(Y_i, f_{m-1}(x_i) + \rho h^*_m(x_i)),$$

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

---

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

**Input**:
- training data $D = \{(Y_1, x_1), \ldots, (Y_n, x_n)\}$, with labels $Y_i \in \mathcal{Y} = \{-1, 1\}$
- number of iterations $M \geq 1$
- learning rate $\eta \in (0, 1]$.

1. Initialize $f_0(x) = \arg \min_{\theta \in \mathbb{R}} \sum_{i=1}^{n} L(Y_i, \theta)$.

2. while $m \in \{1, \ldots, M\}$ do

3. $g_m(x_i) := -\frac{\partial L(Y_i, f_{m-1}(x_i))}{\partial f_{m-1, i}}|_{f_{m-1, i}}$.

4. Compute $h^*_m = \arg \min_{h \in \mathcal{F}, \beta} \sum_{i=1}^{n} (g_m(x_i) - \beta h(x_i))^2$.

5. Compute $\rho_m = \arg \min_{\rho > 0} \sum_{i=1}^{n} L(Y_i, f_{m-1}(x_i) + \rho h^*_m(x_i))$.

6. Scale $f^*_m(x) = \eta \rho_m h^*_m(x)$.

7. Update $f_m(x) = f_{m-1}(x) + f^*_m(x)$.

8. end

**Output**: Return $f_M(x)$.

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

$$h^*_m = \arg \min_{h \in \mathcal{F}} \sum_{i=1}^{n} \left( g_m(x_i) h(x_i) + \frac{1}{2} H_m(x_i) h^2(x_i) \right),$$  \hspace{1cm} (2.14)

where $H_m(x_i) := (\nabla^2 L_L(f_m))_{ii}$. Minimization in (2.14) is equivalent to

$$h^*_m = \arg \min_{h \in \mathcal{F}} \sum_{i=1}^{n} \frac{1}{2} H_m(x_i) \left( -\frac{g_m(x_i)}{H_m(x_i)} - h(x_i) \right)^2,$$  \hspace{1cm} (2.15)

i.e. a weighted least squares regression problem. We present Newton boosting algorithm in **Algorithm 7** with learning rate $\eta$, for sake of completeness.
Algorithm 7: Newton boosting algorithm in [23].

Input:
- training data \( D = \{(Y_1, x_1), \ldots, (Y_n, x_n)\} \), with labels \( Y_i \in \mathcal{Y} = \{-1, 1\} \)
- number of iterations \( M \geq 1 \)
- learning rate \( \eta \)

1. Initialize \( f_0(x) = \arg \min_{\theta} \sum_{i=1}^{n} L(Y_i, \theta) \).

2. while \( m \in 1, \ldots, M \) do
   3. \( g_m(x_i) := -\frac{\partial L(Y_i, f_{m-1})}{\partial f_{m-1,i}} \bigg|_{f_{m-1,i}} \)
   4. \( H_m(x_i) := \left( \nabla^2 L(f_{m-1}) \right)_{ii} \).
   5. Compute \( h_m^* = \arg \min_{h \in F} \sum_{i=1}^{n} \frac{1}{2} H_m(x_i) \left( -\frac{g_m(x_i)}{H_m(x_i)} - h(x_i) \right)^2 \).
   6. Scale \( f_m^*(x) = \eta h_m^*(x) \).
   7. Update \( f_m(x) = f_{m-1}(x) + f_m^*(x) \).

end

Output: Return \( f_M(x) \).

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 \( \eta \), 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 \((Y_i, x_i)\) in \(X' \subset X\) has response \(y\) is given by

\[
\hat{p}_\omega(y|X') := \frac{n_\omega(y,X';D)}{\sum_y n_\omega(y,X';D)} = \frac{\sum_i n_i \left[ Y_i = y, x_i \in X' \right]}{\sum_i n_i \left[ x_i \in X' \right]},
\]

(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 \(\hat{p}(y|X')\) in formula (5.25) of [35] and DecisionTreeClassifier() reduces to classical classification tree algorithm. Note that we assume that there is at least one \(x_i\) such that \(x_i \in X'\).

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

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\footnote{https://github.com/scikit-learn/scikit-learn/blob/7389dba/sklearn/tree/tree.py#L598} we have:

*The weighted impurity decrease equation is the following:

\[
\frac{N_t}{N} \cdot (\text{impurity} - \frac{N_{t,R}}{N_t} \cdot \text{right_impurity} - \frac{N_{t,L}}{N_t} \cdot \text{left_impurity})
\]

\[
\text{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 \text{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 \( N, N.t, N.t.R \) and \( N.t.L \) denote the weighted (again, using \text{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\footnote{https://github.com/scikit-learn/scikit-learn/issues/12398}. 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\footnote{https://github.com/scikit-learn/scikit-learn/pull/4899} 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\footnote{https://scikit-learn.org/stable/modules/tree.html#tree-algorithms-id3-c4-5-c5-0-and-cart} mentions the use of the CART\footnote{Classification And Regression Trees.} 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 \cite{Classification And Regression Trees.} for all details on the original CART algorithm and to Section 9.2.4 of \footnote{https://scikit-learn.org/stable/modules/tree.html#tree-algorithms-id3-c4-5-c5-0-and-cart} 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\footnote{https://github.com/scikit-learn/scikit-learn/blob/7389dba/sklearn/tree/tree.py#L598} Algorithm\footnote{https://github.com/scikit-learn/scikit-learn/blob/7389dba/sklearn/tree/tree.py#L598} 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
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()\(^{20}\): 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 \(\alpha_m\) for SAMME and the update of weights \(\omega_i\) 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 \(\text{estimator.error} \leq 0\) or \(\text{estimator.error} \geq 1. - \left(\frac{1}{\text{n.classes}}\right)\). 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 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 (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, 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; 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. 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

$$
\mathcal{L}_{\text{bin}}(f_m) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_{\text{bin}}(Y_i, f_m(x_i)),
$$

$$
\mathcal{L}_{\text{bin}}(Y_i, f_m(x_i)) = -2 \left( Y_i \log(p^m_i) + (1 - Y_i) \log(1 - p^m_i) \right) = -2 \left( Y_i f_m(x_i) - \log(1 + \exp(f_m(x_i))) \right),
$$

where $p^m_i = \frac{1}{1 + \exp(-f_m(x_i))}$, for all $i \in \{1, \ldots, n\}$, 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](https://cran.r-project.org/web/packages/ada/ada.pdf)
24. [https://cran.r-project.org/web/packages/JOUSBoost/JOUSBoost.pdf](https://cran.r-project.org/web/packages/JOUSBoost/JOUSBoost.pdf)
27. [https://logitboost.readthedocs.io/index.html](https://logitboost.readthedocs.io/index.html)
in Python for `GradientBoostingClassifier()` as shown in the following code chunk, where `pred` is not the vector of probabilities $p^m_i$, but of predictions $(f_m(x_1), \ldots, f_m(x_n))$ at boosting step $m$. `.ravel()` is an array manipulation routine and `np` denotes the use of the `numpy` package:

```python
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 $L_{bin}$, with $L_{bin}$ in (4.1).

The gradient $\nabla L_{bin}(f_m)$ of the loss functional $L_{bin}$ in case of binomial loss (4.1) is then:

$$\nabla L_{bin}(f_m) = -\frac{2}{n} (Y_1 - p^m_1, \ldots, Y_n - p^m_n).$$

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;
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`

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

\[ \hat{y}_i := \varphi \left( \sum_{m=1}^{M} f_m(x_i) \right), \]

for all \((x_i, y_i) \in \mathcal{D}\), where \(f_m \in \mathcal{H} = \{ f = f(\cdot, q|T) : \mathcal{X} \to \mathbb{R}, x_i \mapsto \omega_q(x_i) \}\), denotes the space of regression or classification/decision tree models.\(^{30}\) \(\omega_q(x_i)\) is the score of the \(q(x_i)\)-th leaf of the tree \(f(\cdot, q|T)\), and \(\varphi\) 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 \(\varphi\) is the logit function. In other words, for regression type problems, the tree ensemble score \(\sum_{m=1}^{M} f_m(x_i)\) represents the predicted target variable for the data point \(x_i\), while for binary classification problems the tree ensemble score is transformed into probabilities \(\text{via} \) a logit transformation.

\textbf{XGBoost} generates a tree ensemble by Newton approximations of the regularized objective

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

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 \(\omega_i\)'s, where \(\omega = (\omega_1, \ldots, \omega_T)\) and \(\omega_t\) denoting the value at leaf \(t = 1, \ldots, T\)). 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

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

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

\[ g_i = \frac{\partial L}{\partial \hat{y}_i^{m-1}} \bigg|_{(\hat{y}_i^{m-1}, y_i)}, \]

\[ h_i = \frac{\partial^2 L}{\partial \hat{y}_i^{m-1} \partial \hat{y}_i^{m-1}} \bigg|_{(\hat{y}_i^{m-1}, y_i)} , \]

\(^{30}\) In \cite{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) $\omega^*_j$ of leaf $j \in \{1, \ldots, T\}$ for the tree minimizing (5.2) is (cf. formula (5) in [6])

$$\omega^*_j = -\frac{\sum_{i=1}^n g_i I[q(x_i) = j]}{\lambda + \sum_{i=1}^n h_i I[q(x_i) = j]},$$

(5.3)

while the tree structure function $q : \mathcal{X} \rightarrow T$ 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])

\[
\text{Loss reduction} = [\text{Impurity Node} - \text{Impurity Left Child} - \text{Impurity Right Child}] - \gamma,
\]

where, with a slight abuse of notation, impurity at node $N$ to be split reads as $\text{Impurity Node} = -\frac{1}{2} \left( \frac{\sum_{i \in N} g_i}{\lambda + \sum_{i \in N} h_i} \right)^2 + \gamma T$. 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; 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


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


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. XGBClассifier() 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](https://cran.r-project.org/web/packages/xgboost/vignettes/discoverYourData.html)

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. 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. 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].

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

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. 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 dependences) and activate it. The installation of the package xgboost is notoriously non-easy. 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. 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 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
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

[^40]: https://www.kaggle.com/c/ClaimPredictionChallenge/discussion/703
[^41]: https://www.kaggle.com/c/porto-seguro-safe-driver-prediction/kernels
<table>
<thead>
<tr>
<th>Public Leaderboard Rank</th>
<th>Normalized Gini Coefficient - Score</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.29154</td>
<td>83</td>
</tr>
<tr>
<td>2</td>
<td>0.29034</td>
<td>293</td>
</tr>
<tr>
<td>3</td>
<td>0.28993</td>
<td>252</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2'591</td>
<td>0.28</td>
<td>22</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3'525</td>
<td>0.27</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3'846</td>
<td>0.26</td>
<td>8</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5'169</td>
<td>-0.24553</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>#</th>
<th>Column Name</th>
<th>#</th>
<th>Column Name</th>
<th>#</th>
<th>Column Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>id</td>
<td>21</td>
<td>ps_reg_01</td>
<td>41</td>
<td>ps_calc_02</td>
</tr>
<tr>
<td>2</td>
<td>target</td>
<td>22</td>
<td>ps_reg_02</td>
<td>42</td>
<td>ps_calc_03</td>
</tr>
<tr>
<td>3</td>
<td>ps_ind_01</td>
<td>23</td>
<td>ps_reg_03</td>
<td>43</td>
<td>ps_calc_04</td>
</tr>
<tr>
<td>4</td>
<td>ps_ind_02_cat</td>
<td>24</td>
<td>ps_car_01_cat</td>
<td>44</td>
<td>ps_calc_05</td>
</tr>
<tr>
<td>5</td>
<td>ps_ind_03</td>
<td>25</td>
<td>ps_car_02_cat</td>
<td>45</td>
<td>ps_calc_06</td>
</tr>
<tr>
<td>6</td>
<td>ps_ind_04_cat</td>
<td>26</td>
<td>ps_car_03_cat</td>
<td>46</td>
<td>ps_calc_07</td>
</tr>
<tr>
<td>7</td>
<td>ps_ind_05_cat</td>
<td>27</td>
<td>ps_car_04_cat</td>
<td>47</td>
<td>ps_calc_08</td>
</tr>
<tr>
<td>8</td>
<td>ps_ind_06_bin</td>
<td>28</td>
<td>ps_car_05_cat</td>
<td>48</td>
<td>ps_calc_09</td>
</tr>
<tr>
<td>9</td>
<td>ps_ind_07_bin</td>
<td>29</td>
<td>ps_car_06_cat</td>
<td>49</td>
<td>ps_calc_10</td>
</tr>
<tr>
<td>10</td>
<td>ps_ind_08_bin</td>
<td>30</td>
<td>ps_car_07_cat</td>
<td>50</td>
<td>ps_calc_11</td>
</tr>
<tr>
<td>11</td>
<td>ps_ind_09_bin</td>
<td>31</td>
<td>ps_car_08_cat</td>
<td>51</td>
<td>ps_calc_12</td>
</tr>
<tr>
<td>12</td>
<td>ps_ind_10_bin</td>
<td>32</td>
<td>ps_car_09_cat</td>
<td>52</td>
<td>ps_calc_13</td>
</tr>
<tr>
<td>13</td>
<td>ps_ind_11_bin</td>
<td>33</td>
<td>ps_car_10_cat</td>
<td>53</td>
<td>ps_calc_14</td>
</tr>
<tr>
<td>14</td>
<td>ps_ind_12_bin</td>
<td>34</td>
<td>ps_car_11_cat</td>
<td>54</td>
<td>ps_calc_15_bin</td>
</tr>
<tr>
<td>15</td>
<td>ps_ind_13_bin</td>
<td>35</td>
<td>ps_car_11</td>
<td>55</td>
<td>ps_calc_16_bin</td>
</tr>
<tr>
<td>16</td>
<td>ps_ind_14_bin</td>
<td>36</td>
<td>ps_car_12</td>
<td>56</td>
<td>ps_calc_17_bin</td>
</tr>
<tr>
<td>17</td>
<td>ps_ind_15_bin</td>
<td>37</td>
<td>ps_car_13</td>
<td>57</td>
<td>ps_calc_18_bin</td>
</tr>
<tr>
<td>18</td>
<td>ps_ind_16_bin</td>
<td>38</td>
<td>ps_car_14</td>
<td>58</td>
<td>ps_calc_19_bin</td>
</tr>
<tr>
<td>19</td>
<td>ps_ind_17_bin</td>
<td>39</td>
<td>ps_car_15</td>
<td>59</td>
<td>ps_calc_20_bin</td>
</tr>
<tr>
<td>20</td>
<td>ps_ind_18_bin</td>
<td>40</td>
<td>ps_calc_01</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 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.

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.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Number of Missing Entries</th>
<th>Percentage of Missing Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>ps_ind_02_cat</td>
<td>216</td>
<td>0.04</td>
</tr>
<tr>
<td>ps_ind_04_cat</td>
<td>83</td>
<td>0.01</td>
</tr>
<tr>
<td>ps_ind_05_cat</td>
<td>5,809</td>
<td>0.98</td>
</tr>
<tr>
<td>ps_reg_03</td>
<td>107,772</td>
<td>18.11</td>
</tr>
<tr>
<td>ps_car_01_cat</td>
<td>107</td>
<td>0.02</td>
</tr>
<tr>
<td>ps_car_02_cat</td>
<td>5</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>ps_car_03_cat</td>
<td>411,231</td>
<td>69.09</td>
</tr>
<tr>
<td>ps_car_05_cat</td>
<td>266,551</td>
<td>44.78</td>
</tr>
<tr>
<td>ps_car_07_cat</td>
<td>11,489</td>
<td>1.93</td>
</tr>
<tr>
<td>ps_car_09_cat</td>
<td>569</td>
<td>0.10</td>
</tr>
<tr>
<td>ps_car_11</td>
<td>5</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>ps_car_12</td>
<td>1</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>ps_car_14</td>
<td>42,620</td>
<td>7.16</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Binary Variable</th>
<th>% of 1’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>ps_ind_10_bin</td>
<td>0.04</td>
</tr>
<tr>
<td>ps_ind_11_bin</td>
<td>0.17</td>
</tr>
<tr>
<td>ps_ind_12_bin</td>
<td>0.94</td>
</tr>
<tr>
<td>ps_ind_13_bin</td>
<td>0.09</td>
</tr>
</tbody>
</table>

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, ··· , 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 3. 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.

<table>
<thead>
<tr>
<th>target Level</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>96.36</td>
</tr>
<tr>
<td>1</td>
<td>3.64</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variable</th>
<th>Pearson Correlation</th>
<th>Spearman Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ps_car_12</td>
<td>ps_car_13</td>
<td>0.67</td>
<td>0.66</td>
</tr>
<tr>
<td>ps_reg_01</td>
<td>ps_reg_03</td>
<td>0.64</td>
<td>-</td>
</tr>
<tr>
<td>ps_car_13</td>
<td>ps_car_15</td>
<td>0.53</td>
<td>0.68</td>
</tr>
<tr>
<td>ps_reg_03</td>
<td>ps_reg_02</td>
<td>0.52</td>
<td>0.64</td>
</tr>
<tr>
<td>ps_reg_01</td>
<td>ps_reg_02</td>
<td>-</td>
<td>0.54</td>
</tr>
</tbody>
</table>

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 readability, 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
A straightforward 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 $X_{\text{train}}$ 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 $X_{\text{train}}$ data set.

Lastly, we finish by reporting the performance on $X_{\text{test}}$ of the ‘best’ model fitted on $X_{\text{train}}$ 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

$$Gini_{\text{CAP}} = \frac{a_R}{a_P},$$

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

$$a_P = \frac{1}{2}(1 - freq_{\text{pos}}),$$

where $freq_{\text{pos}}$ denotes the percentage of records associated to the ‘signal’ in the provided data set; for example, on test data $y_{\text{test}}$ one would have

```python
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 $y_{\text{test}}=1$).

Similarly, one can prove that $Gini_{\text{CAP}} = 2\text{AUC} - 1$, 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 $gini(\text{actual}, \text{pred})$ and $gini_{\text{normalized}}\text{score}(\text{actual}, \text{pred})$, which are introduced in the next paragraph.
On Gini coefficient, \( gini(actual, pred) \) and the \( gini_{normalized}(actual, pred) \)

The code to compute both the Gini coefficient \( gini(actual, pred) \) and the normalized Gini coefficient \( gini_{normalized}(actual, pred) \) 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 \( gini_{normalized}(actual, pred) \). 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 Gini\_CAP = \( \frac{a_R}{a_P} \), using the matrix \( D \). Geometrically we have:

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

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

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

Then

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

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

arriving at

\[
a_R = gini(x, p),
\]

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

\[
a_P = gini(x, x)
\]

by definition of the perfect model, and Gini\_CAP = \( \frac{a_R}{a_P} = gini_{normalized}(x, p) \), then it follows

\[
gini_{normalized}(x, p) = 2AUC - 1.
\]

In summary, there is an affine transformation between \( gini_{normalized}(x, p) \) 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.
   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.

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 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 $\text{gini}_\text{normalized}(x,p) = 0.286$ 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.

<table>
<thead>
<tr>
<th>Run</th>
<th>max_depth</th>
<th>learning_rate and n_estimators</th>
<th>Best hyperparameters</th>
<th>AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.</td>
<td>1</td>
<td>[0.001, 0.01, 0.1, 1], [100, 300, 500]</td>
<td>(1, 0.1, 500)</td>
<td>0.639</td>
</tr>
<tr>
<td>II.</td>
<td>2</td>
<td>[0.001, 0.01, 0.1, 1], [100, 300, 500]</td>
<td>(2, 0.1, 500)</td>
<td><strong>0.643</strong></td>
</tr>
<tr>
<td>III.</td>
<td>(3, 4)</td>
<td>[0.001, 0.01, 0.1, 1], [100, 300, 500]</td>
<td>(3, 0.1, 300)</td>
<td><strong>0.643</strong></td>
</tr>
<tr>
<td>IV.</td>
<td>5</td>
<td>[0.001, 0.01, 0.1, 1], [100, 300, 500]</td>
<td>(5, 0.1, 100)</td>
<td>0.641</td>
</tr>
</tbody>
</table>

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 XGBoost 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].

[^44]: https://stats.stackexchange.com/questions/162162/relative-variable-importance-for-boosting
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
\[
D_{m+1}(i) = \frac{\beta_m}{Z_m} D_m(i) < D_m(i) \iff \beta_m < Z_m.
\]
As the misclassification error \( \epsilon_m \) of \( h_m \) is such that \( 0 < \epsilon_m < \frac{1}{2} \), then \( 0 < \beta_m < 1 \); therefore we arrive at
\[
Z_m = \sum_{k:h_m(x_k)=Y_k} D_m(k)\beta_m + \sum_{k:h_m(x_k)\neq Y_k} D_m(k) > \sum_{k:h_m(x_k)=Y_k} D_m(k)\beta_m + \sum_{k:h_m(x_k)\neq Y_k} D_m(k)\beta_m
\]
\[
= \sum_{k=1}^n D_m(k)\beta_m = \beta_m,
\]
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 logistic loss. To do so, let assume that we are at step \( m \leq M \), with estimated function \( f_m \); the loss functional (2.11) is
\[
\mathcal{L}_L(f_m) = \mathcal{L}_L(f_{m,1}, \ldots, f_{m,n}) = \frac{1}{n} \sum_{i=1}^n \log(1 + \exp(-Y_if_{m,i}))
\]
by the definition of the logistic loss. The gradient \( \nabla \mathcal{L}_L(f_m) \) and the Hessian \( \nabla^2 \mathcal{L}_L(f_m) \) of \( L \) at \( f_m \) read as
\[
\nabla \mathcal{L}_L(f_m) = -\frac{1}{n} \left( \frac{Y_1}{1 + \exp(Y_1f_{m,1})}, \ldots, \frac{Y_n}{1 + \exp(Y_nf_{m,n})} \right),
\]
\[
\nabla^2 \mathcal{L}_L(f_m) = \frac{1}{n} \begin{pmatrix}
\frac{\exp(Y_1f_{m,1})}{(1 + \exp(Y_1f_{m,1}))^2} & \cdots & 0 \\
\cdots & \ddots & \cdots \\
0 & \cdots & \frac{\exp(Y_nf_{m,n})}{(1 + \exp(Y_nf_{m,n}))^2}
\end{pmatrix},
\]
where we used \( Y_i^2 = 1 \), for all \( i \)'s. Again, as \( Y_i \in \{-1, +1\} \), we arrive at
\[
(\nabla^2 \mathcal{L}_L(f_m))_{ii} = \frac{1}{n} \frac{\exp(-f_{m,i})}{(1 + \exp(-f_{m,i}))^2},
\]
i.e. \( (\nabla^2 \mathcal{L}_L(f_m))_{ii} = \frac{1}{n} \omega_m(i) \), 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
\[
-\frac{g_m(x_i)}{H_m(x_i)} = -\frac{\nabla_i \mathcal{L}_L(f_m)}{(\nabla^2 \mathcal{L}_L(f_m))_{ii}},
\]
for all \( i = 1, \ldots, n \). Therefore, we arrive at
\[
-\frac{g_m(x_i)}{H_m(x_i)} = \begin{cases}
\frac{1 + \exp(f_{m,i})}{\exp(f_{m,i})} & \text{if } Y_i = +1 \\
\frac{1 - \exp(-f_{m,i})}{\exp(-f_{m,i})} & \text{if } Y_i = -1
\end{cases}
\]
where \( p_m(x_i) \) and \( z_m(x_i) \) 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


