Uncovering the Depths of Gradient Boosting

A Hopefully Comprehensive Write-Up (Mainly for me I guess...)

Robert Bajons

Motivation

I found it quite intricate to wrap my head around the full concept of gradient boosting as well as the fine differences between the original approach of Friedman (2001) and the approach of Chen and Guestrin (2016) for their XGBoost package. Thus this work should document my comprehension of this topic as well as provide space to annotate any unclear parts. It may serve as a resource for understanding, coding (mainly in R) and possibly extending the gradient boosting framework.

A Brief History

Note: This is by no means an exhaustive history of existing literature, but it gives an overview of certain very influential or interesting/helpful papers on boosting.

- First works from a computational learning viewpoint (not yet generalized): Freund (1995), Adaboost by Freund and Schapire (1997).
- Generalizations of the Boosting Framework:

Approach 1: Statistical view of Adaboost by Friedman et al. (2000), General Boosting Idea and Algorithm by Friedman (2001) and extensions in Friedman (2002), Comprehensive summary by Ridgeway (1999a,1999b).

Approach 2: Boosting from an Optimization viewpoint by Mason et al. (1999,2000).

Both approaches mathematically revised and to some extent unified more recently by Biau and Cadre (2017).

- A lot of in the 2000s and 2010s, impossible to cite all so omitted for the moment.
- Novel boosting approaches and recent extensions: XGBoost Chen and Guestrin (2016), accelerated gradient boosting Biau et al. (2018), gradient and newton boosting in an unified and mathematical framework Sigrist (2021), Grabit model: a specific model for class imbalance Sigrist and Hirnschall (2019).

Predictive Learning Problem

We mainly use the problem formulation given by Friedman (2001). Consider having a dataset of size N coming from a random output variable y and a set of d random input variables $\boldsymbol{x}$. The goal is to obtain an estimate or approximation $\hat{F}(\boldsymbol{x})$, of the function $F^*(\boldsymbol{x})$ mapping $\boldsymbol{x}$ to y, that minimizes the expected value of some specified loss function $L(y, F(\boldsymbol{x}))$ over the joint distribution of all $(y, \boldsymbol{x})$-values:

$$F^* = \arg\min_F \mathbb{E}[L(y, F(\boldsymbol{x}))]. \tag{1}$$

In typical data science / machine learning problems, the estimation focuses on minimizing the loss over the given set of training data of $(y, \boldsymbol{x})$ available, i.e. solving the empirical analogue to the modified version of equation (1):

$$F^* = \arg\min_F \mathbb{E}_{y|\boldsymbol{x}}[L(y, F(\boldsymbol{x}))|\boldsymbol{x}]. \tag{2}$$

Solving equation (2) is equivalent to solving (1), since the loss function L is usually non-negative and we can use the law of total expectation to rewrite (1) as an expectation w.r.t. $\boldsymbol{x}$ of the conditional expectation in (2).

Add some notes on loss functions!!

Typical loss functions used in the predictive learning problem¸ include the squared error[1] for regression or the negative binomial log-likelihood for classification.

Optimizing a Function

To solve the above problem, one typical approach would be to parametrize $F(\boldsymbol{x})$, such that is belongs to the class of functions $F(\boldsymbol{x}, \boldsymbol{P})$. where $\boldsymbol{P} = \{P_1, P_2, \dots\}$. One simple example would be to choose F as a linear function in the parameters, i.e. a linear regression $F(\boldsymbol{x}, \boldsymbol{P}) = \boldsymbol{P}^\top \boldsymbol{x}$[2]. We will later come back to this simple example. Using a parametrized version of F the optimization problem is changed such that we are searching for[3]

$$\boldsymbol{P}^* = \arg\min_{\boldsymbol{P}} \mathbb{E}[L(y, F(\boldsymbol{x}, \boldsymbol{P}))] = \arg\min_{\boldsymbol{P}} \Phi(\boldsymbol{P}). \tag{3}$$

and then

$$F^*(\boldsymbol{x}) = F(\boldsymbol{x}, \boldsymbol{P}^*). \tag{4}$$

In lack of an explicit solution one usually has to resort to numerical optimization in order to solve the problem equation (3). Friedman states that "this often involves expressing the solution for the parameters in the form" of a sum, starting with an initial guess successively adding increments ("boosts"). In essence this refers to Newton type methods, where one starts with an initial guess and then iterates via a specific rule to find a better approximation of the extremum by following a specific rule. These methods try to use a clever approach to find the direction in which one should move to get to the extremum. In a one dimensional case (see below), this is rather simple as there are only 2 ways to go (left or right, imagine a simple plot), however in higher dimensions there are infinite possible direction to move (360 degree radius, imagine an x,y plane in the two-dimensinal case).

The most traditional Newton method (also called Newton Raphson method) for finding a one dimensional extremum use the following iteration

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{5}$$

Thus the final value for the extremum could in this case be expressed as sum of the increments:

$$x^* = x_0 + \sum_{i=1}^{M} \hat{x}_i, \tag{6}$$

with a starting value x_0 and $\hat{x}_i = -\frac{f(x_{i-1})}{f'(x_{i-1})}$ and maximum number of iterations M. This can be generalized to higher dimensions by using the gradient and the hessian, where finding the search direction (i.e. the increments) amounts to solving a linear system of the form:

$$Hf(x_n)p_n = -\nabla f(x_n). \tag{7}$$

[1] The sample analog of the expected squared error loss is the well known mean squared error (MSE)

[2] Usually the parameters in linear regression are denoted by β...

[3] Typically again consider sample analogue of the modified equation (2).

Thus the iteration amounts to

$$x_{n+1} = x_n + p_n. \tag{8}$$

and we obtain the solution as sum of increments

$$x^* = x_0 + \sum_{i=1}^{M} p_i. \tag{9}$$

The main idea for any kind of Newton-type algorithm is based on Taylor series approximation. In essence, the idea is that instead of optimizing the original function, one tries to optimize a Taylor approximation of the function up to a certain order (usually order 2 is sufficient) at a starting value x_{k-1}. For more details a more thorough analysis of non linear optimization books is required (see e.g. Griva et al. (2008), chapters 2,11,12).

Steepest Descent

The steepest descent, sometimes also called gradient descent, is one of the simplest Newton-type methods out there. The basis of these approach is that the gradient of a function provides the direction of the steepest ascent of the curve[4]. In order to find the minimum[5] one idea is to move in the direction of the maximal decrease, i.e. the negative of the gradient. If the function is well behaved and the step size is not too big one should with every iteration move to a value that is lower than the original value. This is the most basic implementation of a newton method, which compared to the Newton Raphson method describe above only takes gradient information (but not the hessian) into account.

In order to solve the initial problem as given by equation (3), we thus need to compute the gradient and evaluate it at the value of the previous step of the iteration:

$$g_n = \nabla\Phi(\boldsymbol{P})\big|_{\boldsymbol{P}=\boldsymbol{P}_{n-1}} \tag{10}$$

and evaluate it at each step at $\boldsymbol{P}_{n-1}$, where we start with some starting value $\boldsymbol{P}_0$ and $\boldsymbol{P}_n = \boldsymbol{P}_{n-1} - g_n$[6]. Thus the each $\boldsymbol{P}_n$ can again be written as sum:

$$\boldsymbol{P}_n = \sum_{i=0}^{n} p_i, \tag{11}$$

where p_0 is some starting value and $p_i = -g_i$. Usually the steepest descent algorithm is combined with a line search algorithm, such that the step p_n and $\boldsymbol{P}_n$ are given by

$$p_n = -\rho_n g_n, \quad \text{resp.} \quad \boldsymbol{P}_n = \boldsymbol{P}_{n-1} - \rho_n g_n. \tag{12}$$

The optimal value of ρ at each iteration is derived by solving the Problem

$$\rho_n = \arg\min_{\rho} \Phi(\boldsymbol{P}_{n-1} - \rho g_n). \tag{13}$$

Example

We illustrate the steepest descent algorithm in the context of a predictive learning problem for probably the most common simple parametrized function F, the linear regression. Note that for the case of linear regression, an explicit solution to the least squares problem, which is equivalent to minimizing the MSE is available, so from a computational point of view it would not make sense to use the steepest descent algorithm. However it is a good example to visualize how the method would work.

[4]Hint: A nice intuitive explanation can be found at wikipedia. The proof is quite straightforward using the definition of the directional derivative

[5]Optimization here mostly refers to minimization as maximization can be achieved by minimizing the negative of the function.

[6]We stick to the notation used by Friedman and describe the steepest descent method with respect to $\boldsymbol{P}$, one would usually use x, but in our case this would clash with the notation above.

```r
##############################
### Simulate data
##############################

set.seed(123)
x1 <- rnorm(500,10,5)
x2 <- rnorm(500,1,1)

y <- 1.5*x1-3*x2+rnorm(500)


##############################
### Fit linear Model
##############################


mod <- lm(y ~ x1+x2-1)

MSE <- function(beta,y,x1,x2){
  mean((y-(beta[1]*x1+beta[2]*x2))^2)
}

grad <- function(beta,y,x1,x2){
  c(mean(-x1*2*(y-(beta[1]*x1+beta[2]*x2))),mean(-x2*2*(y-(beta[1]*x1+beta[2]*x2))))
}

##############################
### Gradient Descent with line search
### without line search no convergence!!
### line search taken from Wikipedia
##############################


bold <- c(1,1)
#bold <- c(1.523738,-3.527437)
eps = 1
counter = 1
while(eps > 10^(-7)){
  g <- grad(bold,y,x1,x2)
  alpha <- 1
  ### line search part very important
  while(MSE(bold,y,x1,x2)-MSE(bold-alpha*g,y,x1,x2) < -alpha/4*crossprod(bold,g)){
    alpha = alpha/2
  }
  bnew <- bold - alpha*grad(bold,y,x1,x2)
  eps <- abs(MSE(bnew,y,x1,x2)-MSE(bold,y,x1,x2))
  #eps <- max(abs(bold-bnew)) ## choose some stopping criterion
  #eps <- abs(MSE(bnew,y,x1,x2))
  bold <- bnew
  counter = counter+1
  if(counter %% 10000 == 0){
    cat("prec = ",eps," round = ",counter," MSE = ", MSE(bnew,y,x1,x2))
  }
}
```

```
##
##  Results from steepest descent:
## prec =  0  round =  68  MSE =  1.013114
##  coeff1 =  1.489883  coeff2 =  -2.812129
##
##
##  Linear Model (least squares output):
##
## Call:
## lm(formula = y ~ x1 + x2 - 1)
##
## Coefficients:
##     x1      x2
##  1.502  -2.972
## MSE for least squares:  0.9811374
```

The above code provides a way for using the steepest descent algorithm in a linear regression setup (without intercept). A few important notes are necessary:

- The result of the gradient descent algorithm are close to the linear model results, however least squares provides a more accurate result (in terms of MSE and when comparing to the real values).
- The steepest descent algorithm only works when using an appropriate line search algorithm otherwise we overshoot the steps length and the algorithm does not converge.
- The result of the steepest descent algorithm might be improved by adjusting the stopping criterion, as well as the line search algorithm (through the choice of criterion and adjustment of alpha in each line search step).

Optimization in Function Space

When using a non parametric approach to, one could still try to solve problem (2) using similar arguments as above. Instead of considering parameters of a function one would consider the function $F(\boldsymbol{x})$ at each point $\boldsymbol{x}$ as a parameter and use a stepwise updating procedure starting with an initial value $f_0(\boldsymbol{x})$. Using a steepest descent approach one would thus update the estimate $\hat{F}$ at each step m by

$$F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}) - \rho_m g_m, \quad g_m = \nabla\Phi(F)|_{F=F_{m-1}}.$$

The optimal step length ρ would again be determined by performing a line search similar to equation (13). Thus $F_m(\boldsymbol{x})$ could again be expressed as a sum

$$F_m(\boldsymbol{x}) = \sum_{i=0}^{m} f_i(\boldsymbol{x}),$$

where $f_i(\boldsymbol{x}) = -\rho_i g_i$.

As pointed out by many authors (Friedman (2001), Ridgeway (1999b), Hastie et al. (2009)), this procedure is far from optimal in a finite data setting, where all estimation has to be done on the sample analog of equation (2), since we would only estimate the function on the set of data points available. Thus using this procedure would only be desirable when the goal would be to minimize the loss on the training data set. However one is usually interesting in predictions for unseen values of $\boldsymbol{x}$, i.e. generalization of the function to to new data outside of the training set.

Gradient Boosting and its Implementations

Gradient Boosting build on the ideas described in the previous section. Having a training data set $(\boldsymbol{x}, y)_{i=1}^{N}$ at hand one is interested in estimating a function F^* by minimizing the sample analog of equation (2):

$$F^* = \arg\min_F \sum_{i=1}^{N} [L(y, F(\boldsymbol{x}))]. \tag{14}$$

To do so, we take the "natural"[7] approach of fitting additive expansions in a greedy stagewise process, i.e. the estimate for optimal function can be written as sum

$$\hat{F}(\boldsymbol{x}) = \sum_{m=1}^{M} \beta_m h(\boldsymbol{x}, a_m), \tag{15}$$

where each component of above sum is derived at each stage of the process. Note that we have already expressed each term of the sum in a specific form. This is the usual approach in gradient boosting, where each component is assumed to be a member of a class of function usually termed as "weak learners" (Friedman (2001)) or "basis functions" (Hastie et al. (2009)), i.e. simple function that on its own do not possess great predictive power. The idea of using additive expansion is not a unique characteristic of gradient boosting, but rather is a very common approach in many machine learning techniques, e.g. (single layer) neural networks, regression splines, etc. (cf. Hastie et al. (2009)).

Parameter optimization in such a model with respect to some loss functions amounts to solving the problem

$$\min_{\beta,a} \sum_{i=1}^{N} L(y, \sum_{m=1}^{M} \beta_m h(\boldsymbol{x}, a_m)). \tag{16}$$

Solving the above optimization is usually infeasible, especially for highly non linear loss functions. To avoid solving the expensive problem the idea is to use a simple alternative, where a forward stagewise algorithm is employed, such that each weak learner is fit sequentially. Thus, at each iteration, the new component of the sum is estimated without affecting/modifying the coefficients of previous terms. The algorithm therefore only needs to fit a single base learner at each iteration, which can be computationally much more efficient. For each $m = 1, \ldots, M$ one solves the problem

$$\min_{\beta,a} \sum_{i=1}^{N} L(y, F_{m-1}(\boldsymbol{x}) + \beta_m h(\boldsymbol{x}, a_m)). \tag{17}$$

For squared error loss the above problem leads to fitting a regression tree to the current residuals[8], as can be seen by plugging in:

$$\min_{\beta,a} \sum_{i=1}^{N} L(y, F_{m-1}(\boldsymbol{x}) + \beta_m h(\boldsymbol{x}, a_m)) = \min_{\beta,a} \sum_{i=1}^{N} \left((y - F_{m-1}(\boldsymbol{x})) - \beta_m h(\boldsymbol{x}, a_m)\right)^2 \tag{18}$$

Solving problem (17) can be quite difficult as well for a number of loss functions. This is where the usual implementations try to find a clever way to come up with a generalized solution that works for any (possibly new) data point $\boldsymbol{x}$. While there are different ways to come up with a clever approach for solving (17) (as we will see further on), the general idea is the same. At each step the model selects a tree that locally minimizes an approximate of the loss function (linear approximation for

[7] Natural in that sense means that it comes from the previously derived concepts for optimizing functions.

[8] Note that fitting a regression tree is usually (by default) done by minimising squared error loss (similiar to linear regression), thus this approach is in some sense very intuitive. See also the section on decision trees below.

gradient descent, quadratic approximation for Newton descent as in XGBoost), i.e. the procedure is analogous to a Newton type method.

To be more concrete, in its most general form the initial approach by (Friedman (2001)) uses at each step m a linear approximation of the loss function at the point F_{m-1}, i.e. it searches for the step direction that minimizes the loss under the constraint, that the new step direction is a member of the class of weak learners. As already mentioned the "best" step direction is in this case the data-based analogue of the negative gradient, evaluated at F_{m-1}. However, as we are restricted to a class of weak learners and we want to generalize the gradient information to values of $\boldsymbol{x}$ that are not contained in the training data set, the idea is to choose the weak learner $h(\boldsymbol{x}, a)$, such that evaluated at training values it is most parallel to the negative gradient. Thus we simply regress the weak learner on the negative gradient using a least squares criterion[9]:

$$\min_{a} \sum_{i=1}^{N} (-g_m(\boldsymbol{x_i}) - h(\boldsymbol{x_i}, a))^2. \tag{19}$$

Similarly to basic gradient descent the above step is accompanied by a line search

$$\min_{\rho} \sum_{i=1}^{N} L(y_i, F_{m-1}(\boldsymbol{x}) + \rho h(\boldsymbol{x}, a_m)) \tag{20}$$

in order to scale the optimal step direction and to guarantee function descent at each iteration.

In general the solution to equation (19) will not be the same as the solution of (17), however they should be similar enough and due to the simple loss function (MSE) the numerical optimization is usually much less expensive (Hastie et al. (2009)). In the case of squared error as initial loss function L, both approaches lead to the same result, since the gradient of L amounts to the residuals $y_i - F_{m-1}(\boldsymbol{x})$[10].

While the intuition of the above procedure is (at least for me) to some extent clear, the subtle details of gradient descent and Taylor expansions are (at least to me) not that obvious. Thankfully Sigrist (2021) provides a unifying approach to gradient and Newton (i.e. higher order approximation) boosting. Furthermore Biau and Cadre (2017) investigate some mathematical aspects of gradient boosting[11] by discussing the optimization principles of two main approaches, which (at least to me) was very helpful in understanding and unifying existing gradient boosting methodologies. We briefly discuss the main takeaways from these 2 papers. The main building block is the Taylor approximation of the empirical loss functional around F_{m-1}

$$\begin{aligned} R^e(F + f) &= \frac{1}{n} \sum_{i=1}^{N} L\big(y_i, F_{m-1}(\boldsymbol{x}_i) + f(\boldsymbol{x}_i)\big) \\ &\approx \frac{1}{n} \sum_{i=1}^{N} L(y_i, F_{m-1}(\boldsymbol{x}_i)) + f(\boldsymbol{x}_i) G_{m,i} + \frac{1}{2} f(\boldsymbol{x}_i)^2 H_{m,i}, \end{aligned} \tag{21}$$

where $G_{m,i} = \frac{\partial}{\partial F} L(y_i, F_{m-1}(\boldsymbol{x}_i))$ and $H_{m,i} = \frac{\partial^2}{\partial F^2} L(y_i, F_{m-1}(\boldsymbol{x}_i))$. The gradient G and the Hessian H in this context are treated as simple derivatives of the functional $\Psi(F) = L(y, F)$, and then evaluated at each $\boldsymbol{x}_i$. Each G_i and H_i is thus merely a scalar. The mathematical justification for that is given in more detail by Sigrist (2021). Equation (21) shows the Taylor approximation of order 2, which is used for newton boosting algorithms. For the classical gradient boosting approach the second order term is dropped.

"Trick": Gradient/Hessian in function space is treated as 1-dim derivative!

[9] As pointed out by Friedman any fitting criterion could be used, but least squares offers computational advantages and, as can be seen by Sigrist, it is from a mathematical point of view very natural.

[10] Note that the gradient evaluated at $F_{m-1}(\boldsymbol{x})$ for an arbitrary loss function is also sometimes referred to as pseudo-residuals.

[11] While they mainly focus on first order approximation the mathematical details do not change drastically when considering second order boosting methods.

For gradient descent we thus seek to find

$$\min_f \frac{1}{n} \sum_{i=1}^{N} L(y_i, F_{m-1}(\boldsymbol{x}_i)) + f(\boldsymbol{x}_i) G_{m,i}, \tag{22}$$

where f is in the class of weak learners. It can be noted, that the only relevant part to optimize is (the sum of) $f(\boldsymbol{x}_i)G_{m,i}$, however this problem in general does not necessarily has a finite minimum (Griva et al. (2008)). Thus Sigrist (2021)[12] uses a trick and adds a constraint on the norm of f. The new objective is (omitting the irrelevant parts of the equations)

$$\min_f \frac{1}{n} \sum_{i=1}^{N} f(\boldsymbol{x}_i) G_{m,i} + \frac{1}{2} f(\boldsymbol{x}_i)^2, \tag{23}$$

which can be expressed equivalently as

$$\min_f \frac{1}{n} \sum_{i=1}^{N} (-G_{m,i} - f(\boldsymbol{x}_i))^2. \tag{24}$$

Thus, this leads to the same procedure as described by Friedman (2001), cp. equation (19).

Newton boosting uses the second order term as well, which leads to the problem (omitting irrelevant terms)

$$\begin{aligned} &\min_f \frac{1}{n} \sum_{i=1}^{N} f(\boldsymbol{x}_i) G_{m,i} + \frac{1}{2} f(\boldsymbol{x}_i)^2 H_{m,i} \\ \Leftrightarrow &\min_f \frac{1}{n} \sum_{i=1}^{N} H_{m,i} (-\frac{G_{m,i}}{H_{m,i}} - f(\boldsymbol{x}_i))^2. \end{aligned} \tag{25}$$

Finding the optimal add on at each iteration thus relates to fitting a weak learner using a weighted least squares criterion on the negative ratio of gradient to Hessian, weighted by H_i. This Newton boosting approach in essence leads to the same procedure as used by Chen and Guestrin (2016) for their XGBoost algorithm. The somewhat only difference is that they do not use weighted least squares but directly optimize the loss function. Since they are using tree models, this however is equivalent, that is, for tree modelling, estimation via weighted least squares is the same as just using a least squares criterion.

A Note on Decision Trees

As advocated by Hastie et al. (2009) (chapter 10.7), decision or regression trees are a very attractive choice for using in combination with gradient boosting, i.e. for usage as weak learners. Often also referred to as "off-the-shelf" method, trees fulfills a few desiderata for data mining and machine learning methods: they are able to handle combinations of datatypes (continuous, binary, categorical data), they are able to handle missing values naturally, they are interpretable and finally they are to some extend computationally inexpensive. The latter however only holds when using heuristics to grow a tree. While optimizing along the whole space of trees is usually infeasible, a greedy approach is employed which leads to fast tree growing, at the cost of only obtaining suboptimal trees. As a result of the off-the-shelf properties of trees, there is no need for tedious data preprocessing steps, scaling or transforming the data. Furthermore trees are resistant to predictor outliers and perform feature selection, omitting the inclusion of irrelevant predictors. Finally another advantage is the interpretability of these models (Hastie et al. (2009)). However there is one main disadvantage, which prevents them from being the optimal tool for predictive modelling, namely their variance[13]. That is, the tree is likely to overfit to training data and thus not generalize well to unseen data. In other words , a small change in the data can cause a large change in the

Maybe provide more Info on Bias-Variance tradeoff here!

[12] Similarly Biau and Cadre (2017) norms the gradient to provide a solution.

[13] This is also referred to as instability in Hastie et al. (see section 9.2.4. at the end).

final estimated tree (James et al. (2021)). This is where the combination of decision trees with boosting has the most effect. It drastically improves accuracy of the model while maintaining most of the desired off-the-shelf properties[14].

We mainly follow Hastie et al. (2009) and briefly describe the fitting (or growing) of trees. A tree divides each training example $\boldsymbol{x}$ into distinct non overlapping regions, producing high-dimensional rectangles[15]. Mathematically it is typically written as a step function in the following form.

$$f(\boldsymbol{x}) = \sum_{j=1}^{J} \gamma_j \mathbb{1}_{\{\boldsymbol{x} \in R_j\}}. \tag{26}$$

Thus the goal is to find a partition $R_1, \ldots, R_J$ and coefficients $\gamma_1, \ldots, \gamma_J$ such that the function f minimizes some specific loss criterion. For regression trees the usual criterion is the square error, i.e. minimizing $\sum_(y_i - f(\boldsymbol{x}_i))^2$, for classification trees usually the misclassification rate is used. Thus in essence one tries to find

$$\arg\min_{\Theta} \sum_{j=1}^{J} \sum_{\boldsymbol{x}_i \in R_j} L(y_i, \gamma_j), \tag{27}$$

where $\Theta = \{y_j, R_j\}_{j=1}^{J}$. Solving this problem for any partition is typically infeasible and thus a greedy top down approach is typically employed, leading to a suboptimal solution. Given the regions R_j the estimation of γ_j is usually easy, for squared error loss it is simply the average of all examples falling into region R_j, for misclassification error it is the majority class in region R_j. The derivation of the Regions R_j however is the tricky part, where a greedy approach is necessary. Typically the idea is used is termed recursive binary splitting, where first a splitting variable x_j and a split point s is determined, and the example space is split into two regions $R_{1,(j,s)} = \{\boldsymbol{x} | x_j < s\}$ and $R_{2,(j,s)} = \{\boldsymbol{x} | x_j \geq s\}$. The splitting variable x_j as well as the cutpoint s are not chosen arbitrarily but rather such that they minimize the equation

$$\sum_{i:\boldsymbol{x}_i \in R_1} \tilde{L}(y_i, \hat{\gamma}_{R_1}) + \sum_{i:\boldsymbol{x}_i \in R_2} \tilde{L}(y_i, \hat{\gamma}_{R_2}), \tag{28}$$

where $\hat{\gamma}_{R_i}$ are the estimates for γ in region R_i. Note that we have used a loss function $\tilde{L}$ which is not necessary the same as the original L, that is the loss for determining the γ can sometimes be different from the loss for estimating the regions R_j. This is especially done in the classification case where the misclassification error is replaced by a smoother loss function such as the Gini index or cross entropy. The greedy step from equation (28) is then repeated for each subregion R_1 and R_2, i.e. for each R_i, $i = 1, 2$, we again split it into two regions using the best splitting variable x_{j_i} and s_i for the particular region R_i. This process is then continued until a specified stopping criterion is reached, e.g. until no region contains more than c observations, where c is a tuning parameter (James et al. (2021)).

As noted by Hastie et al. (2009) and James et al. (2021), the above procedure might lead to a very large an complex tree, which might overfit the training data. To cite James et al. (2021): "A smaller tree with fewer splits might lead to lower variance and better interpretation at the cost of a little bias". An easy but ineffective strategy would be to build the tree until the loss reduction is high enough, i.e. exceeds some predefined threshold. However a seemingly worthless split might subsequently lead to a very effective spit (Hastie et al. (2009)).

Bias-Variance tradeoff, see also above.

The go-to approach in this case is cost-complexity-pruning. The strategy is to initially grow a big tree T_0, and then prune it back in order to obtain a smaller subtree. In a nutshell, the idea is to minimize the cost-complexity criterion

$$C_\alpha(T) = \sum_{m=1}^{|T|} Q_m(T) + \alpha |T|, \tag{29}$$

[14] The main sacrifice of boosting trees is the loss of interpretability.

[15] In two dimension this can be nicely visualized, see Hastie et al..

where $|T|$ is the number of (final) nodes in the subtree T of the initial big tree T_0 (the size of the tree), α is a tuning parameter that governs the tradeoff between tree size and its goodness of fit to the data and $Q_m(T)$ is the error in Node m (e.g. for squared error $\sum_{\boldsymbol{x}_i \in R_m} (y_i - \hat{\gamma}_m)^2$). Thus the tree T_α which minimized equation (29) is used for prediction instead of T_0. The parameter α is usually determined via cross validation (see also algorithm 8.1. from James et al. (2021), page 309).

The Classical Implementation (Friedman (2001))

Maybe add Algorithm here

In principle Friedman provides an generic algorithm for pure gradient boosting which is in essence described above (cp. equation (19) and (20)). However in his paper, he also provides specific use cases, especially focusing on trees as base learners. Furthermore their adoption of the Adaboost algorithm in Friedman et al. (2000) actually uses a hybrid gradient-Newton algorithm (cf. Sigrist (2021)). In this section we present some specifics of the implementations from the famous Friedman (2001) paper.

XGBoost (Chen and Guestrin (2016))

Extensions to Gradient Boosting

Accelerated Gradient Boosting (AGB, Biau et al. (2018))

Grabit (Sigrist and Hirnschall (2019))

References

[1] Biau, G. and Cadre, B. (2017). Optimization by gradient boosting.

[2] Biau, G., Cadre, B., and Rouvière, L. (2018). Accelerated gradient boosting.

[3] Chen, T. and Guestrin, C. (2016). XGBoost. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM.

[4] Freund, Y. (1995). Boosting a weak learning algorithm by majority. *Information and Computation*, 121(2):256–285.

[5] Freund, Y. and Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139.

[6] Friedman, J., Hastie, T., and Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting (With discussion and a rejoinder by the authors). *The Annals of Statistics*, 28(2):337 – 407.

[7] Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189 – 1232.

[8] Friedman, J. H. (2002). Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378. Nonlinear Methods and Data Mining.

[9] Greenwell, B., Boehmke, B., Cunningham, J., and Developers, G. (2020). *gbm: Generalized Boosted Regression Models*. R package version 2.1.8.

[10] Griva, I., Nash, S. G., and Sofer, A. (2008). *Linear and Nonlinear Optimization (2. ed.)*. SIAM.

[11] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer Series in Statistics. Springer New York.

[12] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2021). *An Introduction to Statistical Learning: with Applications in R*. Springer Texts in Statistics. Springer US.

[13] Mason, L., Baxter, J., Bartlett, P., and Frean, M. (1999). Boosting algorithms as gradient descent. In Solla, S., Leen, T., and M"uller, K., editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press.

[14] Mason, L., Baxter, J., Bartlett, P. L., and Frean, M. (2000). Functional Gradient Techniques for Combining Hypotheses. In *Advances in Large-Margin Classifiers*. The MIT Press.

[15] Ridgeway, G. (1999a). *Generalization of boosting algorithms and applications of Bayesian inference for massive datasets*. PhD thesis, University of Washington.

[16] Ridgeway, G. (1999b). The state of boosting. *Computing Science and Statistics*, 31:172–181.

[17] Ridgeway, G. (2007). Generalized boosted models: A guide to the gbm package.

[18] Sigrist, F. (2021). Gradient and newton boosting for classification and regression. *Expert Systems with Applications*, 167:114080.

[19] Sigrist, F. and Hirnschall, C. (2019). Grabit: Gradient tree-boosted tobit models for default prediction. *Journal of Banking & Finance*, 102:177–192.