Numerical Optimization via Gradient Boosting

Emilija Mirković, Sanja Živanović

December 2022.

1 Introduction

Numerical optimization is a branch of mathematics and computer science that deals with finding the best solution for a specific problem in accordance with certain criteria. It may involve the minimization or maximization of a function or set of functions.

Fast algorithms approximating

$$\hat{\Theta}_m = argmin_{\Theta_m} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + T(x_i; \Theta_m))$$

for any differentiable loss function can be derived by their analogy with numerical optimization. The loss function when using $f(x)$ to predict y on the training set is

$$L(f) = \sum_{i=1}^{N} L(y_i, f(x_i)).$$

The goal is to reduce $L(f)$ with respect to f, where $f(x)$ is restricted to be the sum of trees $f_M(x) = \sum_{m=1}^{M} T(x; \Theta_m)$

Ignoring this limitation, the minimization of $L(f)$ can be understood as a numerical optimization

$$\hat{f} = argmin_f L(f),$$

where the "parameters" $f \in R^N$ are the values of the approximating function $f(x_i)$ at each of the N points $f = \{f(x_1), f(x_2), ..., f(x_N)\}^T$.

Numerical optimization methods solve

$$\hat{f} = argmin_f L(f)$$

as a sum of component vectors $f_M = \sum_{m=0}^{M} h_m, h_m \in R^N$, where $f_0 = h_0$ is the initial attempt, and each subsequent f_m is induced based on f_{m-1}. Numerical optimization methods differ according to the method of calculating h_m ("step").

1.1 Steepest Descent

Steepest Descent is an iterative optimization method. It is based on the idea that in each iteration we move in the direction of the greatest decrease of the function.

Specifically, for each iteration, the gradient of the function at the current point is determined and we move in the direction opposite to the gradient. This step is repeated until the algorithm converges to the minimum of the function.

The Steepest Descent method is often used to solve optimization problems in machine learning and other fields because it is simple to implement. However, there are also several drawbacks, such as slow convergence in cases with many dimensions.

The steepest descent method chooses $h_m = \rho_m g_m$, where ρ_m is a scalar and $g_m \in R^N$ is gradient $L(f)$ calculated for $f = f_{m-1}$. The gradient components g_m are

$$g_{im} = [\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}]_{f(x_i)=f_{m-1}(x_i)}.$$

The step length ρ_m is the solution for

$$\rho_m = argmin_\rho L(f_{m-1} - \rho g_m)$$

The current solution is then updated to $f_m = f_{m-1} - \rho_m g_m$ and the process is repeated in the next iteration.

1.2 Gradient Boosting

Gradient boosting is one of the methods for numerical optimization. It uses the gradient of a function to determine the direction to move in each step.

Gradient boosting is an efficient method for finding optimal solutions for many optimization problems, but it has several drawbacks. For example, it may get stuck in local minimums or maximums, which means it will not find a globally optimal solution. Also, the learning rate plays an important role in the efficiency of gradient boosting and if it is too large, it can lead to divergence, and if it is too small, it can slow down the process of finding the optimal solution.

If minimizing the loss function

$$L(f) = \sum_{i=1}^{N} L(y_i, f(x_i))$$

on the training set is the only goal, steepest descent would be the preferred strategy.

Gradient

$$g_{im} = [\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}]_{f(x_i)=f_{m-1}(x_i)}$$

from the previous chapter is defined only in the points x_i of the training set, while the ultimate goal is the generalization of $f_M(x)$ to new data that are not represented in the training set. A possible solution is to induce the tree $T(x; \Theta_m)$ at the m-th place iterations whose predictions t_m are as close as possible to the negative gradient.

Using squared error we get:

$$\Theta_m = argmin_\Theta \sum_{i=1}^{N} (-g_i m - T(x_i; \Theta))^2.$$

After constructing the previous tree, the corresponding constants in each region are given by:

$$\hat{\gamma}_{jm} = argmin_{\gamma_{jm}} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma_{jm})$$

For classification: each tree T_k^m is adapted to the corresponding negative gradient vector g_{km},

$$-g_{ikm} = [\frac{\partial L(y_i, f_1(x_1), ..., f_K(x_i))}{\partial f_k(x_i)}]_{f(x_i)=f_{m-1}(x_i)} = I(y_i = G_k) - p_k(x_i)$$

so that

$$p_k(x) = \frac{e^{f_k(x)}}{\sum_{l=1}^{K} e^{f_1(x)}}$$

For binary classification (K = 2), only one tree is needed.

1.3 Implementations of Gradient Boosting

We will now present the Gradient Boosting Algorithm.
Specific algorithms are obtained by inserting different loss criteria $L(y, f(x))$.
1. set $f_0(x) = argmin_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.

2. for values $m = 1, ..., M$:
(a) for values $i = 1, ..., N$ calculate:

$$r_{im} = -[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}]$$

(b) fit a regression tree to the targets r_{im} giving terminal regions $R_{jm}, j = 1, 2, ..., J_m$. (c) for values $j = 1, ..., J_m$ calculate:

$$\gamma_{jm} = argmin_\gamma \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$$

(d) update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$. 3. output is $\hat{f}(x) = f_M(x)$
The first line of the algorithm is initialized to the optimal constant model, which is only one terminal node of the tree. The negative gradient components calculated in line 2(a) are called generalized or pseudo-residuals.

The classification algorithm is similar. Lines 2 (a)-(d) are repeated K times in each iteration m, once for each class using $-g_{i_{km}}$ from the previous section. The result in line 3 is K different (connected) extensions of the tree $f_{kM}(x), k = 1, 2, ..., K$.

Gradient boosting as described here is implemented in the R gbm package (Ridgeway, 1999, *"Gradient Boosted Models"*), and is freely available. Another R implementation of gradient boosting is mboost (Hothorn and B¨uhlmann, 2006)...

2 Right-Sized Trees for Boosting

It is used to build decision trees with an optimal depth that is selected based on the data of the training set. This method differs from traditional boosting methods, such as AdaBoost, because it uses an algorithm to optimize the depth of the tree instead of increasing the weight of misclassified examples. This leads to better performance of the classifier compared to traditional methods. This method differs from traditional Boosting methods in that it uses "right-sized" trees instead of deep trees.

This method leads to less overfitting and better performance in cases with little data. This technique also allows for faster calculations and easier interpretation of results.

A simple approach is to constrain all trees to be **of the same size J**. Thus, J becomes a metaparameter of the entire boosting procedure that is adjusted to maximize the estimated performance for the current data set. Possible values for J can be obtained by considering the properties of the objective function

$$\eta = argmin_f E_{X,Y} L(Y, f(X)).$$

The objective function $\eta(x)$ is the one that has the lowest prediction risk on future data. That's the function we're trying to approximate. One important property of $\eta(x)$ is the degree of interaction of coordinate variables $X^T = (X_1, X_2, ..., X_p)$ with each other. This is noted in its **ANOVA** (analysis of variance) expansion

$$\eta(X) = \sum_j \eta_j(X_j) + \sum_{jk} \eta_{jk}(X_j, X_k) + \sum_{jkl} \eta_{jkl}(X_j, X_k, X_l) + ...$$

The first sum is over the functions of only one predictor variable X_j. The special functions $\eta_j(X_j)$ are those that together best approximate $\eta(X)$ under the loss criterion used. Each such $\eta_j(X_j)$ is called the **"main effect"** of X_j. The second sum represents second-order interactions, the third sum represents third-order interactions, and so on.

No interaction effects greater than $J - 1$ are possible. A value of J should be chosen that reflects the level of dominant interactions $\eta(x)$. Of course, it is mostly unknown, but in most situations it will tend towards low. Although in many cases $J = 2$ will be insufficient, it is very unlikely that J greater than 10 will be required. Experience so far indicates that $4 \leq J \leq 8$ works well in the context of boosting, and the results are quite insensitive to certain choices in this range. One can fine-tune the value of J by trying several different values and selecting the one that produces the lowest risk on the validation sample.

3 Regularization

In addition to the size of the component trees, J, another meta-parameter of gradient boosting is the **number of boosting iterations** M. Each iteration typically reduces the training risk $L(f_M)$, so for M large enough this risk can be arbitrarily small. However, fitting the training data too well can lead to overfitting, which worsens the risk for future predictions. Therefore, there is an optimal number M^* that minimizes the future risk. A practical way to estimate M^* is to track the prediction risk as a function of M on the validation sample. The value of M that minimizes this risk is considered the estimate of M^*.

3.1 Shrinkage

Controlling the value of $M$is not the only possible regularization strategy. The simplest implementation of downscaling in the context of boosting is to *scale the contribution of each tree by a factor* $0 < \nu < 1$ when added to the current approximation.

Smaller values of ν (more downscaling) result in higher training risk for the same number of iterations M. Therefore, both ν and M control the prediction risk on the training data. However, these parameters do not act independently. Smaller values of ν lead to larger values of M for the same training risk, so there is a trade-off between the two.

3.2 Subsampling

The idea is that at each iteration of the algorithm, a fraction η of the training data (without repetition) is taken to be used to grow the next tree.

This technique is called *stochastic gradient boosting* (Friedman, 1999). A typical value for η can be $\frac{1}{2}$, although for large N, η can be significantly less than $\frac{1}{2}$.

Using **subsampling** reduces the computational time, but also in many cases provides a more accurate model. The downside is that we now have four parameters to set: J, M, ν and η. Usually, some earlier research determines appropriate values for J, ν, and η, leaving M as the main parameter.

4 Interpretation

Single decision trees are very interpretive. The whole model can be completely represented by a simple two-dimensional graphic (binary tree) that is easy to visualize.

4.1 Relative Importance of Predictor Variables

It is often useful to learn the relative importance or contribution of each input variable in predicting the response.

For a single decision tree T, Breiman (1984) suggested

$$I_l^2(T) = \sum_{t=1}^{J-1} \hat{\imath}_t^2 I(v(t) = l)$$

as a measure of importance for each predictive variable X_l. The sum is over $J-1$ internal nodes of the tree. At each such node t, one of the input variables $X_{v(t)}$ is used to divide the region into two subregions; a special constant response is fitted within each. The particular variable chosen is the one that gives the maximum estimated improvement $\hat{\imath}_t^2$ in the squared error risk relative to a constant fit over the entire region. The squared relative importance of the variable X_l is the sum of such squared improvements at all internal nodes for which it is chosen as the partition variable.

For K-class classification, K separate models $f_k(x), k = 1, 2, ...,$K are induced, each of which consists of a sum of trees $f_k(x) = \sum_{m=1}^{M} T_{km}(x)$.

4.2 Partial Dependence Plot (PDP)

Plotting $f(X)$ as a function of its arguments provides an overview of the dependence of $f(X)$ on the common value of the input variables. Unfortunately, such visualization is limited to lower dimensional views. We can

easily show functions of one or two arguments, either continuous, discrete or combined. For more than two or three variables a useful alternative can sometimes be to display a collection of plots, each showing the partial dependence of the approximation $f(X)$ on a selected small subset of the input variables. Example:

$$\eta(X) = \sum_j \eta_j(X_j) + \sum_{jk} \eta_{jk}(X_j, X_k) + \sum_{jkl} \eta_{jkl}(X_j, X_k, X_l) + ...$$

Consider a subvector X_S of $l < p$ input variables $X_T = (X_1, X_2, ..., X_p)$, indexed by $S \subset 1, 2, ..., p$. Let $S \cup C = 1, 2, ..., p$. The general function $f(X)$ will depend on all input variables: $f(X) = f(X_S, X_C)$. The partial dependence of $f(X)$ on X_S is: $f_S(X_S) = E_{X_C} f(X_S, X_C)$.

The partial dependence function is estimated by calculating the average on the training data (*Monte Carlo method*):

$$\hat{f}_S(X_S) = \frac{1}{N} \sum_{i=1}^{N} f(X_S, x_{iC}),$$

where $x_{1C}, x_{2C}, ..., x_{NC}$ are the values of X_C occurring in the training set.

PDP are a simple and intuitive way to visualize relationships between variables and can be a valuable tool for gaining a deeper understanding of complex machine learning models. It is a *global method*: The method considers all instances and talks about the global relationship of some characteristic with the predicted outcome. It is useful for questions such as: How much of the wage gap between men and women is due solely to gender, as opposed to differences in education or work experience?