XGBoost: Short review and methods for hyperparameter optimization

Robin Mittas
M. Sc. in Mathematics in Data Science
Technical University Munich
robin.mittas@gmx.de

Abstract

XGBoost (Extreme Gradient Boosting) has emerged as one of the most powerful and widely used machine learning algorithms for classification and regression tasks. Its popularity stems from its scalability, efficiency, and exceptional predictive performance. This paper presents a short review of XGBoost. It is well known that hyperparameters can have a huge impact on model performance, and therefore we here discuss the main hyperparameters of XGBoost, such as learning rate, maximum depth of trees, number of trees, and regularization parameters. We further discuss strategies for hyperparameter tuning including grid search, random search, and Bayesian optimization techniques.

1 Introduction

Gradient tree boosting has shown to work very well on many tasks and competitions, such as ad click rate through rate prediction [4], smart spam classifiers [2] or in the Netflix prize challenge [1]. Similar to random forests, gradient tree boosting is a decision tree ensemble supervised learning algorithm used for regression or classification. They both use multiple decision trees to obtain a better model, however, the main difference is how the trees are built and combined. While random forests use bagging[1] to build full decision trees in parallel from random bootstrap samples of the data set, gradient tree boosting[2] takes a different approach [6]. In gradient tree boosting, decision trees are built sequentially, with each tree constructed to correct the errors made by the ensemble of previous trees. This sequential nature allows gradient boosting to focus more on difficult-to-classify instances, improving overall predictive performance. Additionally, gradient boosting typically uses shallow trees, often referred to as weak learners, which are combined to form a strong ensemble model. This iterative process continues until a predefined stopping criterion is met, such as reaching a maximum number of trees or when further iterations fail to significantly improve performance. By iteratively refining the model based on the residuals of the previous iterations, gradient tree boosting can effectively handle complex relationships in the data and achieve high predictive accuracy. Furthermore, this setting allows formalizing this procedure as a gradient descent algorithm over an objective function [6][2]. As one of the most popular and most efficient implementations of the gradient tree boosting algorithms, XGBoost has gained much popularity in the recent years. XGBoost is a scalable, distributed gradient-boosted decision tree where trees are constructed in parallel, instead of sequentially like in the setting of gradient boosting trees [6]. The method is based on function approximation by optimizing specific loss functions as well as applying several regularization techniques. Nowadays, XGBoost serves as the one of the main go to choices for ensemble methods [2].

[1] Bagging refers to bootstrap aggregating, where we create new datasets by sampling the training set with replacement. Then on each sampled dataset, a different classifier is trained. The final prediction is then given by combining the predictions, *i.e.* average or majority vote.

[2] Boosting is a method where we incrementally train (weak) classifiers that correct the previous mistakes. Here, a higher weight is given to misclassified examples.

2 Setting and Methods

2.1 Gradient tree boosting

Let us start by shortly reviewing gradient tree boosting. For that, let $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)|x_i \in \mathbb{R}^m, y_i \in \mathbb{R}\}$ be a given dataset with $|\mathcal{D}| = n \in \mathbb{N}$ and $m \in \mathbb{N}$ data-features. A tree ensemble model then uses $K \in \mathbb{N}$ additive functions to predict the output $\hat{y}_i$ for a given data-point $(\boldsymbol{x}_i, y_i) \in \mathcal{D}$ as [2]

$$\hat{y}_i = \phi(\boldsymbol{x}_i) = \sum_{k=1}^{K} f_k(\boldsymbol{x_i}), \tag{1}$$

where $f_k \in \mathcal{F} = \{f(x) = w_{q(x)}|q(x) : \mathbb{R}^m \to T\}$ is element of the space of regression trees (CART). Let q represent the structure of each tree mapping a data point to the corresponding leaf index. T is the number of leaves in the tree and each f_k corresponds to an independent tree structure q and leaf weights w. A main difference to decision trees is that each regression tree has a continuous score on each of the leaves, where w_i represents the score on the i-th leaf. For a given $\boldsymbol{x}_i$, we will use the decision rules in the trees to classify it into the leaves and calculate the final prediction by summing up the score in the respective leaves as defined in equation (1) [2].
The following loss function is minimized to learn the set of functions in the model [2]

$$\mathcal{L}(\phi) = \sum_{i=1}^{n} l(\hat{y}_i, y_i) + \sum_{k} \Omega(f_k), \tag{2}$$

where

$$\Omega(f_k) = \gamma T + \frac{1}{2}\lambda \|w\|^2$$

can be understood as regularization term, penalizing the complexity of the model. The first term penalizes the number of leaves, while the second one can be understood as L2 regularization term. The term $\Omega(f_k)$ smoothes the final learnt weights and avoids overfitting. Furthermore, l denotes a convex, differentiable loss function [2].
However, the loss function given in equation (2) includes functions as parameters and can therefore not be optimized by traditional optimization methods in euclidean space. Therefore, the model is trained in an additive manner instead. Let $\hat{y}_i^{(t)}$ be the prediction made of the i-th instance at the t-th iteration. We can then rewrite $\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(\boldsymbol{x}_i))$ yielding the following objective

$$\mathcal{L}^t = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(\boldsymbol{x}_i)) + \Omega(f_t). \tag{3}$$

In other words, we greedily add f_t that improves the model the most according to the loss definition in equation (2). Now, using Taylor's second order approximation, where for any function $f : A \times B \to C$ and $x \in A, y \in B$ we can write $f(x, y + a)$ as

$$f(x, y + a) \approx f(x, y) + a\frac{df(x, y)}{dy} + \frac{1}{2}a^2\frac{d^2 f(x, y)}{dy}.$$

Applying Taylor's theorem to equation (3) yields

$$\mathcal{L}^t \approx \sum_{i=1}^{n}\left(l(y_i, \hat{y}_i^{(t-1)}) + f_t(\boldsymbol{x}_i)\frac{dl(y_i, \hat{y}_i^{(t-1)})}{d\hat{y}_i^{(t-1)}} + \frac{1}{2}f_t^2(\boldsymbol{x}_i)\frac{d^2 l(y_i, \hat{y}_i^{(t-1)})}{d\hat{y}_i^{(t-1)}}\right) + \Omega(f_t). \tag{4}$$

Now, we use following abbreviations for the first and second order gradient statistics

$$g_i = \frac{dl(y_i, \hat{y}_i^{(t-1)})}{d\hat{y}_i^{(t-1)}}$$

$$h_i = \frac{d^2 l(y_i, \hat{y}_i^{(t-1)})}{d\hat{y}_i^{(t-1)}}.$$

Therefore equation (4) can be simplified by further removing the constant terms (in respect to the function f_t we aim to optimize) as

$$\tilde{\mathcal{L}}^t = \sum_{i=1}^{n} \big(g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{x}_i)\big) + \Omega(f_t).$$

Now, for a leaf j, we define its instance set as $I_j = \{i|q(\boldsymbol{x}_i = j\}$, *i.e.* the set containing the indices to the data-points which are mapped to leaf j. Then, by plugging in the definitions of $\Omega(f_t)$ and $f_t \in \mathcal{F}$, we can simplify

$$\tilde{\mathcal{L}}^t = \sum_{i=1}^{n} \big(g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{x}_i)\big) + \gamma T + \frac{1}{2}\lambda \|w\|^2 \tag{5}$$

$$= \sum_{j=1}^{T} \bigg(\big(\sum_{i \in I_j} g_i\big) w_j + \frac{1}{2} \big(\sum_{i \in I_j} h_i + \lambda\big) w_j^2 \bigg) + \gamma T. \tag{6}$$

The optimal weight w_j^* can now be computed by taking the derivative and solving $\partial_{w_j} \tilde{\mathcal{L}}^t = 0$ for a fixed structure $q(\boldsymbol{x})$ as

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}.$$

Subsequently, plugging w_j^* into equation (6) yields

$$\tilde{\mathcal{L}}^t(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{\big(\sum_{i \in I_j} g_i\big)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T. \tag{7}$$

This equation can be used as a scoring function measuring the quality of a tree structure q, which can be derived for a wide range of objective functions.
Let us for example consider the case of binary classification. Let $y_i \in \{0, 1\}$ for all $i \in \{1, ..., n\}$. Therefore, we consider the negative log-likelihood (likelihood $p(y|x, w)$), *i.e.* the binary cross-entropy loss, which we aim to minimize. Similar as before, let $\hat{y}_i^{(t-1)}$ denote the the prediction for a sample i for trees up to and including tree number $t-1$. Then $l(y_i, \hat{y}_i^{(t-1)}) = -(y_i \log(\hat{y}_i^{(t-1)}) + (1 - y_i) \log(1 - \hat{y}_i^{(t-1)})$ denotes the binary cross entropy. Following, g_i and h_i can be computed as

$$g_i = \frac{dl(y_i, \hat{y}_i^{(t-1)})}{d\hat{y}_i^{(t-1)}} = \frac{y_i - \hat{y}_i^{(t-1)}}{\hat{y}_i^{(t-1)}(1 - \hat{y}_i^{(t-1)})}$$

$$h_i = \frac{d^2 l(y_i, \hat{y}_i^{(t-1)})}{d\hat{y}_i^{(t-1)}} = \frac{y_i - 1}{(\hat{y}_i^{(t-1)} - 1)^2} - \frac{y_i}{(\hat{y}_i^{(t-1)})^2}$$

2.2 Approximate split algorithm

In many cases it is impossible to iterate over all possible tree structures q, especially when the data does not fully fit into memory or in a distributed setting. Therefore Chen and Gusterin [2] proposed an approximate algorithm for the splitting criteria. We will not go into detail here but refer to [2]. Summarizing in a few words, XGBoost's approximate split finding technique efficiently identifies candidate split points by summarizing the data distribution using quantiles and constructing histograms for each feature. This allows for fast and scalable tree construction while still achieving high predictive accuracy.

3 Hyperparameters

Here, we will investigate the hyperparameters in XGBoost. The hyperparameters listed in Table 1 are often considered as the hyperparameters which have the most impact on model performance [3][5][7].

Hyperparameter	Description
objective	The objective parameter is the loss function to be minimized. For example, binary:logistic for binary classification or reg:squarederror for regression problem.
max_depth	Maximum depth of a tree for base learner.
n_estimators	Number of gradient boosted trees. Equivalent to number of boosting rounds.
max_leaves	Maximum number of leaves.
learning_rate	Boosting learning rate η.
gamma	Minimum loss reduction required to make a further partition on a leaf node of the tree.
reg_alpha	L1 regularization term on weights.
reg_lambda	L2 regularization term on weights.
colsample_bytree	Subsample ratio of columns when constructing each tree.
min_child_weight	Minimum sum of instance weight (hessian) needed in a child.
subsample	Subsample ratio of the training instance.
eval_metric	Metric used for monitoring the training result and early stopping.

Table 1: Most important hyperparameters in XGBoost.

4 Hyperparameter tuning

XGBoost provides a large range of hyperparameters. Therefore, it is crucial to understand how to tune these hyperparameters to improve and take full advantage of the XGBoost model. Here, we will shortly review different methods for hyperparameter tuning.

4.1 Manual search

Manual search is a method of hyperparameter tuning in which the developer manually selects and adjusts the hyperparameters of the model. It isbest suited when the number of hyperparameters is relatively small and for simple models, allowing the developer to have fine-grained control over the hyperparameters.
To apply manual search, we would define a set of possible values for each hyperparameter, and then manually define and adjust the values of the hyperparameters until the model performance is satisfactory. A main disadvantage is that manual search can be time-consuming and may require significant trial and error to find the optimal combination of hyperparameters. Moreover, manual search is prone to human error, *e.g.* one might overlook a certain combination of hyperparameters or one may not be able to assess the real impact of each parameter on the model's performance [8].

4.2 Grid search

To apply grid search, a developer sets a predefined list of values for each hyperparameter. Then, the model is trained for every possible combination of hyperparameters. Finally, for each combination of hyperparameters, the model is evaluated using a specific metric, for example F1-score, and the model with the best performance is selected. This method is computationally intensive, since it requires training the model on every single combination. Moreover, the trials are limited by the predefined set of of possible values for each hyperparameter, which may not include near optimal values. However, grid search is widely used since it very simple and effective, especially for small and less complex models [8].

4.3 Random search

Random search randomly selects a combination of hyperparameters from a predefined set and trains a model using those hyperparameters. Again, for each hyperparameter a set of possible values is defined. Then one value per hyperparameter is randomly selected from the set and the model is trained using the randomly selected hyperparameters. The process is repeated a predefined number of times, and again, the hyperparameters yielding the best performance is selected as the optimal set of hyperparameters [8]. Random search is often more efficient than exhaustive grid search, especially when dealing with a large hyperparameter space. It explores a broader range of hyperparameters while avoiding the computational cost associated with exhaustively evaluating all possible combinations.

However, random search may not find the optimal hyperparameters. While it efficiently explores the space of possible hyperparameters, there is no guarantee that it will converge to the best possible configuration.

4.4 Bayesian hyperparameter optimization

The hyperparameters listed in Table 1 suggest, that we have many hyperparameters in XGBoost. In this case, applying manual, grid and random search is very cumbersome and time consuming. Also, in these approaches, each run for each set of hyperparameters is independent, *i.e.* they do not use any information of the prior runs. Therefore, the most common approach to tune XGBoost's hyperparameters is via Bayesian optimization as introduced in [9]. Bayesian approaches for hyperparameter tuning, in contrast to random or grid search, keep track of past evaluation results which they use to form a probabilistic model mapping hyperparameters to a probability of a score on the objective function [10]. The goal is to reduce the number of runs until we find the best-performing set of hyperparameters. There are two most common approaches to find these hyperparameters, via Gaussian Processes and via Tree-structured Parzen Estimator.

4.4.1 Bayesian Optimization with Gaussian Process Priors

We are interested in finding the minimum of an objective function $f(\boldsymbol{x})$ over a bounded set $\mathcal{X} \subseteq \mathbb{R}^D$ denoting the set of hyperparameters. Compared to other optimization algorithms, we here construct a probabilistic $f(\boldsymbol{x})$ to then exploit the model to make a decision about where in $\mathcal{X}$ to next evaluate the function, while integrating out uncertainty. The key idea is to use the information available from prior evaluations of $f(\boldsymbol{x})$, and not only rely on local gradient and hessian approximations [9].
To perform Bayesian optimization, we need to first choose a prior over functions that will express assumptions about the function being optimized. Here, we choose the Gaussian process prior since it is flexible and tractable. Afterward, an acquisition function is chosen to construct a utility function from the model posterior, allowing us to determine which point to evaluate next [9].
Let $f : \mathcal{X} \to \mathbb{R}$ be a Gaussian process (GP). Then, it holds that any finite set of points $\{\boldsymbol{x}_n \in \mathcal{X}\}_{n=1}^N$ induces a multivariate Gaussian distribution on $\mathbb{R}^N$. The marginalization properties of the Gaussian distribution allow us to compute marginals and conditionals in closed form [9]. Then for some $n \in \{1, ..., N\}$, it holds $f(\boldsymbol{x}_n) \sim \mathcal{N}(m, K)$, where $m : \mathcal{X} \to \mathbb{R}$ and a positive definite covariance function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. For further information about the impact of kernel functions, we here refer to [9].
Assuming that $f(\boldsymbol{x})$ is drawn from a Gaussian process prior and let the observations be of the form $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$, where $y_n \sim \mathcal{N}(f(\boldsymbol{x}_n), \nu)$ and ν is the variance of introduced noise into the function observations. The prior and the data induce a posterior over functions, referred to as the acquisition function, which is denoted by $a : \mathcal{X} \to \mathbb{R}^+$, determining which point to evaluate next via a proxy approximation $\boldsymbol{x}_{\text{next}} = \operatorname{argmax} a(\boldsymbol{x})$. The acquisition function depends on the previous observations and on the GP parameters, denoted as $a(\boldsymbol{x}; \{\boldsymbol{x}_n, y_n\}, \theta)$. There are many choices for the acquisition function, most commonly used is the expected improvement function. Under the GP prior, the acquisition function depends on the model solely through its predictive mean function $\mu(\boldsymbol{x}; \{\boldsymbol{x}_n, y_n\}, \theta)$ and predictive variance function $\sigma^2(\boldsymbol{x}; \{\boldsymbol{x}_n, y_n\}, \theta)$. The current best value is denoted as $x_{\text{best}} = \operatorname{argmin}_{\boldsymbol{x}_n} f(\boldsymbol{x}_n)$. Furthermore, let $\boldsymbol{\Phi}()$ denote the cumulative distribution of a standard normal, and let $\phi()$ denote the standard normal density function.
The first intuitive way is to maximize the probability of improving over the best current value. This can be computed analytically as

$$a_{\text{PI}}(\boldsymbol{x}; \{\boldsymbol{x}_n, y_n\}, \theta) = \boldsymbol{\Phi}(\gamma(\boldsymbol{x})) \quad \gamma(\boldsymbol{x}) = \frac{\boldsymbol{x}_{\text{best}} - \mu(\boldsymbol{x}; \{\boldsymbol{x}_n, y_n\}, \theta)}{\sigma(\boldsymbol{x}; \{\boldsymbol{x}_n, y_n\}, \theta)}.$$

The expected improvement can also be maximized over the current best value and has a closed-form solution given by

$$a_{\text{EI}}(\boldsymbol{x}; \{\boldsymbol{x}_n, y_n\}, \theta) = \sigma(\boldsymbol{x}; \{\boldsymbol{x}_n, y_n\}, \theta)(\gamma(\boldsymbol{x})\boldsymbol{\Phi}(\gamma(\boldsymbol{x})) + \mathcal{N}(\gamma(\boldsymbol{x}); 0, 1))$$

4.4.2 Tree-structured Parzen Estimator (TPE)

The Tree-structured Parzen Estimator constructs a model by applying the Bayes rule, instead of directly representing $f(\boldsymbol{x}_n)$. In short, let $\boldsymbol{x}$ denote the hyperparameters and $y \in \mathbb{R}$ some score that we want to minimize. The surrogate function is the probability representation of the objective score

function built using previous evaluations [10]. In the case of TPE, the selection function is the criteria by which the next set of hyperparameters are chosen from the surrogate function. The most common choice of criteria is Expected Improvement defined as

$$\mathrm{EI}_{y^*}[\boldsymbol{x}|\mathcal{D}] = \int_{-\infty}^{y^*} (y^* - y) p(y|\boldsymbol{x}, \mathcal{D}) dy,$$

where y^* is a threshold value of the objective function, $\boldsymbol{x}$ is the proposed set of hyperparameters, y is the actual value of the objective function using these hyperparameters, and $p(y|\boldsymbol{x})$ is the surrogate probability model expressing the probability of y given $\boldsymbol{x}$ [10]. With Bayes rule we can rewrite

$$p(y|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|y)p(y)}{p(\boldsymbol{x})} \qquad p(\boldsymbol{x}|y) = \begin{cases} l(\boldsymbol{x}) & \text{if } y < y^* \\ g(\boldsymbol{x}) & \text{else} \end{cases}$$

where $y < y^*$ represents a lower value of the objective function than the threshold. This equation explains that we make two different distributions for the hyperparameters: one where the value of the objective function is less than the threshold, $l(x)$, and one where the value of the objective function is greater than the threshold, $g(x)$.
Finally, with Bayes rule, and some reformulations, the expected improvement equation (which we are trying to maximize) becomes

$$\mathrm{EI}_{y^*}[\boldsymbol{x}|\mathcal{D}] = \frac{\gamma y^* l(\boldsymbol{x}) - l(\boldsymbol{x}) \int_{-\infty}^{y^*} p(y) dy}{\gamma l(\boldsymbol{x}) + (1-\gamma) g(\boldsymbol{x})} \propto \left(\gamma + \frac{g(\boldsymbol{x})}{l(\boldsymbol{x})}(1-\gamma)\right)^{-1}$$

The expected improvement is proportional to the ratio $l(\boldsymbol{x})/g(\boldsymbol{x})$ (due to the negative exponent) and therefore to maximize the expected improvement, we should maximize this ratio. The TPE works by drawing sample hyperparameters from $l(\boldsymbol{x})$, evaluating them in terms of $l(\boldsymbol{x})/g(\boldsymbol{x})$, and returning the set that yields the highest value under $l(\boldsymbol{x})/g(\boldsymbol{x})$ corresponding to the greatest expected improvement. These hyperparameters are then evaluated on the objective function. If the surrogate function is correct, then these hyperparameters should yield a better value when evaluated [10].

The Python library "Hyperopt" provides a framework to easily apply Bayesian optimization to find the optimal hyperparameters. Bayesian optimization should be the go-to method for hyperparameter tuning in XGBoost.

References

[1] J. Bennett and S. Lanning. "The Netflix Prize". In: *Proceedings of the KDD Cup Workshop 2007*. New York: ACM, Aug. 2007, pp. 3–6. URL: http://www.cs.uic.edu/~liub/KDD-cup-2007/NetflixPrize-description.pdf.

[2] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: (2016). DOI: 10.48550/ARXIV.1603.02754. URL: https://arxiv.org/abs/1603.02754.

[3] Eric Luellen. *Mastering XGBoost*. https://towardsdatascience.com/mastering-xgboost-2eb6bce6bc76. Accessed: 2024-03-18.

[4] Xinran He et al. "Practical Lessons from Predicting Clicks on Ads at Facebook". In: *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*. KDD '14. ACM, Aug. 2014. DOI: 10.1145/2648584.2648589. URL: http://dx.doi.org/10.1145/2648584.2648589.

[5] Matt Harrison. *681: XGBoost: The Ultimate Classifier*. https://sds-platform-private.s3-us-east-2.amazonaws.com/uploads/PT681-Transcript.pdf. Accessed: 2024-03-18.

[6] Nvidia. *XGBoost*. https://www.nvidia.com/en-us/glossary/xgboost/. Accessed: 2024-03-18.

[7] RITHP. *The main parameters in XGBoost and their effects on model performance*. https://medium.com/@rithpansanga/the-main-parameters-in-xgboost-and-their-effects-on-model-performance-4f9833cac7c. Accessed: 2024-03-18.

[8] Run:ai. *Hyperparameter Tuning*. https://www.run.ai/guides/hyperparameter-tuning. Accessed: 2024-03-18.

[9] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. *Practical Bayesian Optimization of Machine Learning Algorithms*. 2012. DOI: 10.48550/ARXIV.1206.2944. URL: https://arxiv.org/abs/1206.2944.

[10] Will Koehrsen. *A Conceptual Explanation of Bayesian Hyperparameter Optimization for Machine Learning*. https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f. Accessed: 2024-03-18.