Published in Towards Data Science

David Martins Follow

May 14, 2021 · 11 min read ★ · Listen

Save

XGBoost: A Complete Guide to Fine-Tune and Optimize your Model

How to tune XGBoost hyperparameters and supercharge the performance of your model?

Why is XGBoost so popular?

Initially started as a research project in 2014, XGBoost has quickly become one of the most popular Machine Learning algorithms of the past few years.

Many consider it as one of the best algorithms and, due to its great performance for regression and classification problems, would recommend it as a first choice in many situations. XGBoost has become famous for winning tons of Kaggle competitions, is now used in many industry-application, and is even implemented within machine-learning platforms, such as BigQuery ML.

If you're reading this article on XGBoost hyperparameters optimization, you're probably familiar with the algorithm. But to better understand what we want to tune, let's have a recap!

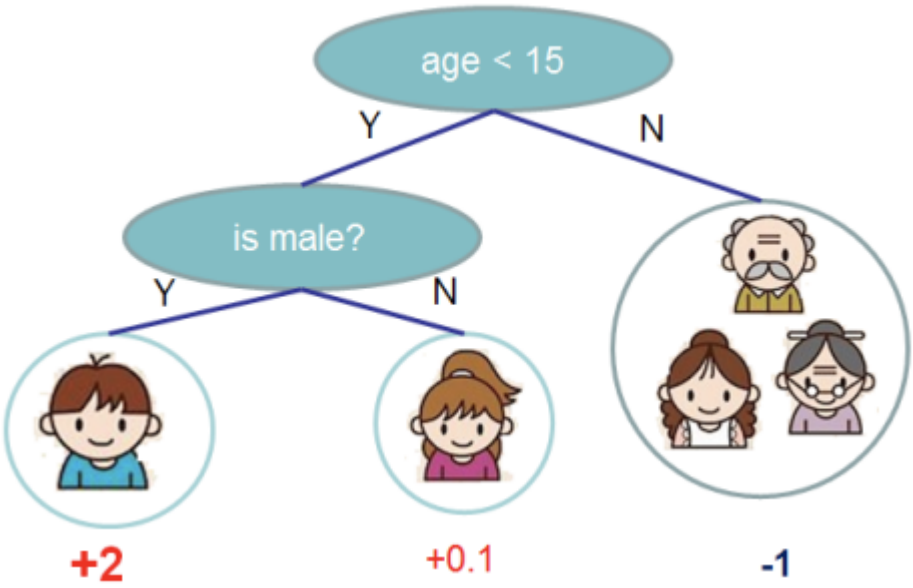
PART 1: Understanding XBGoost

XGBoost (**eXtreme Gradient Boosting**) is not only an algorithm. It's an entire open-source library, designed as an optimized implementation of the Gradient Boosting framework. It focuses on speed, flexibility, and model performances. Its strength doesn't only come from the algorithm, but also from all the underlying system optimization (parallelization, caching, hardware optimization, etc…).

In most cases, data scientist uses XGBoost with a"Tree Base learner", which means that your XGBoost model is based on Decision Trees. But even though they are way less popular, you can also use XGboost with other base learners, such as linear model or Dart. As this is by far the most common situation, we'll focus on Trees for the rest of this article.

> *At that point, you probably have even more questions. What is a Decision Tree? What is Boosting? What the difference with Gradient Boosting?*
> *Don't worry, we'll recap it all!*

What are Decision Trees and CARTs?

CART: Does this person play video games? — Image from XGBoost Documentation

Decision tree is one of the simplest ML algorithms.

It is a way to implement an algorithm that only contains conditional statements.

XGBoost uses a type of decision tree called CART: Classification and Decision Tree.

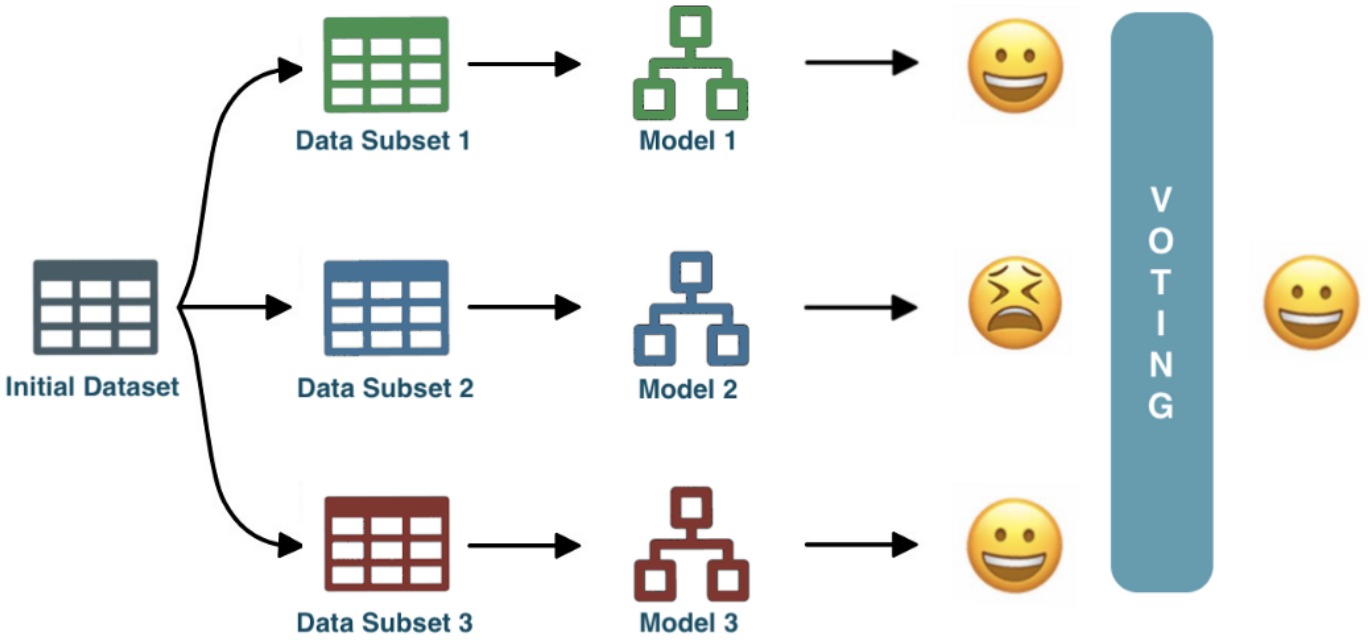
- **Classification Trees**: the target variable is categorical and the tree is used to identify the "class" within which a target variable would likely fall.
- **Regression Trees**: the target variable is continuous and the tree is used to predict its value.

CART leaves don't simply contain final decision values, but also real-valued scores for each leaf, no matter if they are used for classification or regression.

What is Boosting?

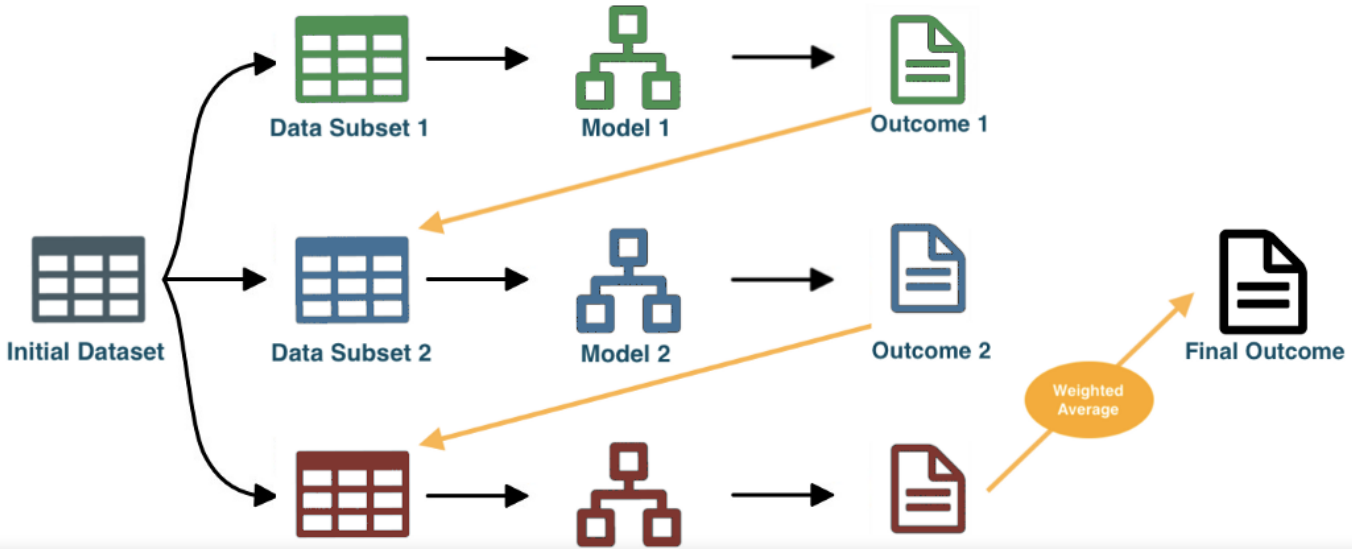
Boosting is just a method that uses the principle of ensemble learning, but in sequential order.

If you're not familiar with ensemble learning, it's a process that combines decisions from



Ensemble Learning example with the Bagging method and a majority-vote strategy — Image by author

Boosting is a type of ensemble learning that uses the previous model's result as an input to the next one. Instead of training models separately, boosting trains models sequentially, each new model being trained to correct the errors of the previous ones. At each iteration (round), the outcomes predicted correctly are given a lower weight, and the ones wrongly predicted a higher weight. It then uses a weighted average to produce a final outcome.



What is Gradient Boosting?

Finally, Gradient Boosting is a boosting method where errors are minimized using a gradient descent algorithm. Simply put, Gradient descent is an iterative optimization algorithm used to minimize a loss function.

The loss function quantifies how far off our prediction is from the actual result for a given data point. The better the predictions, the lower will be the output of your loss function.

$$MSE = \frac{1}{n}\Sigma\underbrace{\left(y - \hat{y}\right)^2}_{\text{The square of the difference between actual and predicted}}$$

Example of loss function: Mean Square Error

When we construct our model, the goal is to minimize the loss function across all of the data points. For example, Mean squared error (MSE) is the most commonly used loss function for regression.

Contrary to classic Boosting, Gradient boosting not only weight higher wrongly predicted outcomes, but also adjust those weights based on a gradient — given by the direction in the loss function where the loss "decreases the fastest". If you want to learn more about Gradient Boosting, you can check out this video.

And as we said in the intro, XGBoost is an optimized implementation of this Gradient Boosting method!

- Learning API: It is the basic, low-level way of using XGBoost. Simple and powerful, it includes a built-in cross-validation method.

```
import xgboost as xgb

X, y = #Import your data
dmatrix = xgb.DMatrix(data=x, label=y) #Learning API uses a dmatrix

params = {'objective':'reg:squarederror'}
cv_results = xgb.cv(dtrain=dmatrix,
                    params=params,
                    nfold=10,
                    metrics={'rmse'})

print('RMSE: %.2f' % cv_results['test-rmse-mean'].min())
```

- Scikit-Learn API: It is a Scikit-Learn wrapper interface for XGBoost. It allows using XGBoost in a scikit-learn compatible way, the same way you would use any native scikit-learn model.

```
import xgboost as xgb

X, y = # Import your data

xtrain, xtest, ytrain, ytest = train_test_split(X, y, test_size=0.2)

xgbr = xgb.XGBRegressor(objective='reg:squarederror')

xgbr.fit(xtrain, ytrain)

ypred = xgbr.predict(xtest)
mse = mean_squared_error(ytest, ypred)
print("RMSE: %.2f" % (mse**(1/2.0)))
```

Note that when using the Learning API you can input and access an evaluation metric, whereas when using the Scikit-learn API you have to calculate it.


XGBoost is a great choice in multiple situations, including regression and classification problems. Based on the problem and how you want your model to learn, you'll choose a different objective function.

The most commonly used are:

- **reg:squarederror**: for linear regression
- **reg:logistic**: for logistic regression
- **binary:logistic**: for logistic regression — with output of the probabilities

PART 2: Hyperparameter tuning

Why should you tune your model?

How would an untuned model perform compared to a tuned model? Is it worth the effort? Before going deeper into XGBoost model tuning, let's highlight the reasons why **you have to tune your model**.

As a demo, we will use the well-known Boston house prices dataset from sklearn, and try to predict the prices of houses.

Here how would perform our model without hyperparameter tuning:

```
import xgboost as xgb
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

boston = load_boston()
X, y = boston.data, boston.target
dmatrix = xgb.DMatrix(data=x, label=y)

params={'objective':'reg:squarederror'}

cv_results = xgb.cv(dtrain=dmatrix, params=params, nfold=10, metrics=
```

```
## Result : RMSE: 3.38
```

Without any tuning, we've got a ***RMSE of 3.38***. Which isn't bad, but let's see how it would perform with just a few tuned hyperparameters:

```
import xgboost as xgb
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

boston = load_boston()
X, y = boston.data, boston.target
dmatrix = xgb.DMatrix(data=x, label=y)

params={ 'objective':'reg:squarederror',
         'max_depth': 6,
         'colsample_bylevel':0.5,
         'learning_rate':0.01,
         'random_state':20}

cv_results = xgb.cv(dtrain=dmatrix, params=params, nfold=10, metrics=
{'rmse'}, as_pandas=True, seed=20, num_boost_round=1000)

print('RMSE: %.2f' % cv_results['test-rmse-mean'].min())

## Result : RMSE: 2.69
```

With just a little bit of tuning, we've now got a ***RMSE of 2.69***. It's a 20% improvement! And we could probably improve even more. Let's see how!

Deep dive into XGBoost Hyperparameters

A hyperparameter is a type of parameter, external to the model, set before the learning process begins. It's tunable and can directly affect how well a model performs.

To find out the best hyperparameters for your model, you may use rules of thumb, or specific methods that we'll review in this article

For Tree base learners, the most common parameters are:

- **max_depth**: The maximum depth per tree. A deeper tree might increase the performance, but also the complexity and chances to overfit.
  The value must be an integer greater than 0. Default is 6.

- **learning_rate**: The learning rate determines the step size at each iteration while your model optimizes toward its objective. A low learning rate makes computation slower, and requires more rounds to achieve the same reduction in residual error as a model with a high learning rate. But it optimizes the chances to reach the best optimum.
  The value must be between 0 and 1. Default is 0.3.

- **n_estimators**: The number of trees in our ensemble. Equivalent to the number of boosting rounds.
  The value must be an integer greater than 0. Default is 100.
  NB: In the standard library, this is referred as **num_boost_round**.

- **colsample_bytree**: Represents the fraction of columns to be randomly sampled for each tree. It might improve overfitting.
  The value must be between 0 and 1. Default is 1.

- **subsample**: Represents the fraction of observations to be sampled for each tree. A lower values prevent overfitting but might lead to under-fitting.
  The value must be between 0 and 1. Default is 1.

Regularization parameters:

- **alpha** (reg_alpha): L1 regularization on the weights (Lasso Regression). When working with a large number of features, it might improve speed performances. It can be any integer. *Default is 0.*

- **lambda** (reg_lambda): L2 regularization on the weights (Ridge Regression). It might help to reduce overfitting. It can be any integer. *Default is 1.*

- **gamma**: Gamma is a pseudo-regularisation parameter (Lagrangian multiplier), and

Approach 1: Intuition and reasonable values

A first approach would be to start with reasonable parameters and to play along. If you understood the meanings of each hyperparameter above, you should be able to intuitively set some values.

Let's start with reasonable values. It would usually be:

> ***max_depth****: 3–10*
> ***n_estimators****: 100 (lots of observations) to 1000 (few observations)*
> ***learning_rate****: 0.01–0.3*
> ***colsample_bytree****: 0.5–1*
> ***subsample****: 0.6–1*

Then, you can focus on optimizing **max_depth** and **n_estimators**.

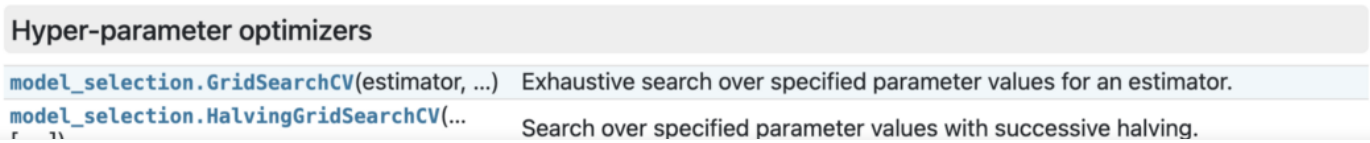
You can then play along with the **learning_rate**, and increase it to speed up the model without decreasing the performances. If it becomes faster without losing in performances, you can increase the number of estimators to try to increase the performances.

Finally, you can work with your regularization parameters, usually starting with alpha and lambda. For gamma, 0 would mean no regularization, 1–5 are commonly used values, whereas 10+ would be considered as very high.

Approach 2: Optimization Algorithms

A second approach to find the best hyperparameters is through Optimization Algorithm. Since XGBoost is available in a Scikit-learn compatible way, you can work with Scikit-learn's hyperparameter optimizer functions!

The two most common are Grid Search and Random Search.



Scikit-learn hyper-parameter optimizers

Grid Search

A Grid Search is an exhaustive search over every combination of specified parameter values. If you specify 2 possible values for **max_depth** and 3 for **n_estimators**, Grid Search will iterate over 6 possible combinations:

> *max_depth: [3,6],*
> *n_estimators:[100, 200, 300]*
>
> *Would result in the following possibilities:*
> *max_depth: 3, n_estimators: 100*
> *max_depth: 3, n_estimators: 200*
> *max_depth: 3, n_estimators: 300*
> *max_depth: 6, n_estimators: 100*
> *max_depth: 6, n_estimators: 200*
> *max_depth: 6, n_estimators: 300*

Let's use **GridSearchCV()** from Scikit-learn to tune our XGBoost model!

In the following examples, we'll use a processed version of the Life Expectancy dataset available on Kaggle.

```
import pandas as pd
import xgboost as xgb
from sklearn.model_selection import GridSearchCV

data = pd.read_csv("life_expectancy_clean.csv")

X, y = data[data.columns.tolist()[:-1]],
       data[data.columns.tolist()[-1]]

params = { 'max_depth': [3,6,10],
           'learning_rate': [0.01, 0.05, 0.1],
           'n_estimators': [100, 500, 1000],
```

```
clf = GridSearchCV(estimator=xgbr,
                   param_grid=params,
                   scoring='neg_mean_squared_error',
                   verbose=1)

clf.fit(X, y)

print("Best parameters:", clf.best_params_)
print("Lowest RMSE: ", (-clf.best_score_)**(1/2.0))
```

- **estimator:** GridSearchCV is part of *sklearn.model_selection*, and works with any scikit-learn compatible estimator. We use *xgb.XGBRegressor()*, from XGBoost's Scikit-learn API.
- **param_grid:** GridSearchCV takes a list of parameters to test in input. As we said, a Grid Search will test out every combination.
- **scoring**: It's the metric(s) that will be used to evaluate the performance of the cross-validated model. In this case, *neg_mean_squared_error* is used in replacement for *mean_squared_error*. GridSearchCV is simply using a negative version of MSE for technical reasons — so it makes the function generalizable to other metrics where we aim for the higher score instead of the lower.
- **verbose:** Controls the verbosity. The higher, the more messages.

More parameters as available, as you can find out in the documentation.

Finally, the lowest RMSE based on the negative value of *clf.best_score_*
And the best parameters with *clf.best_params_*

```
Best parameters: {'colsample_bytree': 0.7, 'learning_rate': 0.05,
'max_depth': 6, 'n_estimators': 500}
```

Random Search

If you input 10 possible values for **max_depth**, 200 possible values for **n_estimators**, and choose to do 10 **iterations**:

> *max_depth: np.arrange(1,10,1),*
> *n_estimators: np.arrange(100,400,2)*
>
> *Example of random possibilities with 10 iterations:*
> *1: max_depth: 1, n_estimators: 110*
> *2: max_depth: 3, n_estimators: 222*
> *3: max_depth: 3, n_estimators: 306*
> *4: max_depth: 4, n_estimators: 102*
> *5: max_depth: 1, n_estimators: 398*
> *6: max_depth: 6, n_estimators: 290*
> *7: max_depth: 9, n_estimators: 102*
> *8: max_depth: 6, n_estimators: 310*
> *9: max_depth: 3, n_estimators: 344*
> *10: max_depth: 6, n_estimators: 202*

Now, let's use **RandomSearchCV()** from Scikit-learn to tune our model!

```
import pandas as pd
import numpy as np
import xgboost as xgb
from sklearn.model_selection import RandomizedSearchCV

data = pd.read_csv("life_expectancy_clean.csv")

X, y = data[data.columns.tolist()[:-1]],
       data[data.columns.tolist()[-1]]

params = { 'max_depth': [3, 5, 6, 10, 15, 20],
           'learning_rate': [0.01, 0.1, 0.2, 0.3],
           'subsample': np.arange(0.5, 1.0, 0.1),
           'colsample_bytree': np.arange(0.4, 1.0, 0.1),
           'colsample_bylevel': np.arange(0.4, 1.0, 0.1),
           'n_estimators': [100, 500, 1000]}
```

```
                            scoring='neg_mean_squared_error',
                            n_iter=25,
                            verbose=1)

clf.fit(X, y)

print("Best parameters:", clf.best_params_)
print("Lowest RMSE: ", (-clf.best_score_)**(1/2.0))
```

- Most *RandomizedSearchCV's* parameters are similar to *GridSearchCV's*.
- **n_iter:** It's the number of parameter combinations that are sampled.
  The higher, the more combinations you'll be testing. It trades off runtime and quality of the solution.

As for *GridSearchCV*, we print the best parameters with *clf.best_params_*
And the lowest RMSE based on the negative value of *clf.best_score_*

Conclusion

In this article, we explained how XGBoost operates to better understand how to tune its hyperparameters. As we've seen, tuning usually results in a big improvement in model performances.

Using our intuition to tune our model might sometimes be enough. It is also worth trying Optimization Algorithms like GridSearch and RandomSearch.
But most of the time, you'll get an even better result with a mix of Algorithms and adjustments through testing and intuition!


You might also be interested...

10 Best Practices to Write Readable and Maintainable SQL Code

How to write SQL queries that your team can easily read and maintain?

towardsdatascience.com