Mastering XGBoost Parameters Tuning: A Complete Guide with Python Codes

Aarshay Jain
07 Jan, 2024 • 15 min read

If things don't go your way in predictive modeling, use XGboost. XGBoost algorithm has become the ultimate weapon of many data scientists. It's a highly sophisticated algorithm, powerful enough to deal with all sorts of irregularities of data. It uses parallel computation in which multiple decision trees are trained in parallel to find the final prediction. This article is best suited to people who are new to XGBoost. We'll learn the art of XGBoost parameters tuning and XGBoost hyperparameter tuning. Also, we'll practice this algorithm using a training data set in Python.

Table of contents

What is XGBoost?

Building a machine learning model using the XGBoost classifier is a straightforward process. Naturally, XGBoost simplifies the initial model creation. However, enhancing the model's performance through XGBoost can be challenging—personally, I encountered some struggles in this phase. This algorithm involves various parameters, and achieving an optimized model requires meticulous parameter tuning. Determining the right set of parameters to fine-tune can be perplexing, and discerning the ideal values for these parameters is crucial for obtaining the best possible output. Consequently, obtaining practical

Mastering XGBoost Parameters Tuning: A Complete Guide with Python Codes

XGBoost (eXtreme Gradient Boosting) is an advanced implementation of a gradient boosting algorithm, which has gained popularity for its high performance and efficiency in various machine learning tasks. If you are already familiar with Gradient Boosting Machine (GBM), you may find it insightful to delve into XGBoost's naturally extensive set of parameters. I previously provided an in-depth exploration of parameter tuning for GBM in my article titled "Complete Guide to Parameter Tuning in Gradient Boosting (GBM) in Python." I highly recommend reviewing that material before proceeding further, as it will not only enhance your overall understanding of boosting techniques but also prepare you for a more nuanced comprehension of naturally available XGBoost parameters.

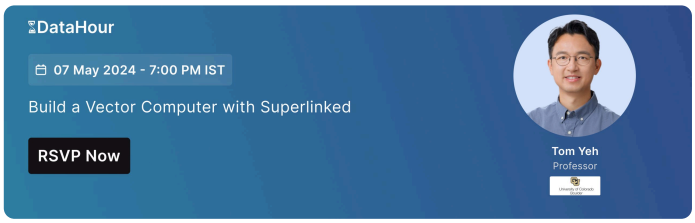
Sample Project to Apply XGBoostProblem StatementHR analytics is revolutionizing the way human resources departments operate, leading to higher efficiency and better results overall. Human resources have been using analytics for years.However, the collection, processing, and analysis of data have been largely manual, and given the nature of human resources dynamics and HR KPIs, the approach has been constraining HR. Therefore, it is surprising that HR departments woke up to the utility of machine learning so late in the game. Here is an opportunity to try predictive analytics in identifying the employees most likely to get promoted. Practice Now

Advantages of XGBoost

I've always admired the boosting capabilities that the XGBoost parameters algorithm infuses into a predictive model. When I explored more about its performance and the science behind its high accuracy, I discovered many advantages, including the flexibility and power of its parameters :

Regularization

Standard GBM implementation has no regularization



Parallel Processing

XGBoost implements parallel processing and is **blazingly faster** as compared to GBM.

But hang on, we know that boosting is a sequential process so how can it be parallelized? We know that each tree can be built only after the previous one, so what stops us from making a tree using all cores? I hope you get where I'm coming from. Check this link out to explore further. XGBoost also supports implementation on Hadoop.

High Flexibility

XGBoost allows users to define **custom optimization objectives and evaluation criteria**. This adds a whole new dimension to the model and there is no limit to what we can do.

Handling Missing Values

XGBoost has an in-built routine to handle missing values. The user is required to supply a different value than other observations and pass that as a parameter. XGBoost tries different things as it encounters a missing value on each node and learns which path to take for missing values in the future.

Tree Pruning

A GBM would stop splitting a node when it encounters a negative loss in the split. Thus it is more of a **greedy algorithm**. XGBoost parameters, on the other hand, makes **splits up to the max_depth** specified and then starts **pruning** the tree backward and removing splits beyond

+10. GBM would stop as it encounters -2. But XGBoost will go deeper, and it will see a combined effect of +8 of the split and keep both.

Built-in Cross-Validation

XGBoost allows the user to run a **cross-validation at each iteration** of the boosting process and thus, it is easy to get the exact optimum number of boosting iterations in a single run. This is unlike GBM, where we have to run a grid search, and only limited values can be tested.

Continue on the Existing Model

Users can start training an XGBoost parameters model from its last iteration of the previous run. This can be of significant advantage in certain specific applications. GBM implementation of sklearn also has this feature, so they are even on this point.

I hope now you understand the sheer power XGBoost algorithm. Note that these are the points that I could muster. Do you know a few more? Feel free to drop a comment below, and I will update the list.

What are XGBoost Parameters?

The overall parameters have been divided into 3 categories by XGBoost authors:

1. **General Parameters:** Guide the overall functioning
2. **Booster Parameters:** Guide the individual booster (tree/regression) at each step
3. **Learning Task Parameters:** Guide the optimization performed

Must Read: [Complete Machine Learning Guide to Parameter Tuning in Gradient Boosting (GBM) in Python]

General Parameters

These define the overall functionality of XGBoost.

- gbtree: tree-based models
- gblinear: linear models

2. **silent [default=0]**
   - Silent mode is activated is set to 1, i.e., no running messages will be printed.
   - It's generally good to keep it 0 as the messages might help in understanding the model.
3. **nthread [default to the maximum number of threads available if not set]**
   - This is used for parallel processing, and the number of cores in the system should be entered
   - If you wish to run on all cores, the value should not be entered, and the algorithm will detect it automatically

There are 2 more parameters that are set automatically by XGBoost, and you need not worry about them. Let's move on to Booster parameters.

Booster Parameters

Though there are 2 types of boosters, I'll consider only **tree booster** here because it always outperforms the linear booster, and thus the latter is rarely used.

Parameter	Description	Typical Values
eta	Analogous to the learning rate in GBM.	0.01-0.2
min_child_weight	Defines the minimum sum of weights of observations required in a child.	Tuned using CV
max_depth	The maximum depth of a tree. Used to control over-fitting.	3-10
max_leaf_nodes	The maximum number of terminal nodes or leaves in a tree.	
gamma	Specifies the minimum loss reduction required to make a split.	Tuned depending on loss function
max_delta_step	Allows each tree's weight estimation to be constrained.	Usually not needed, explore if necessary
	Denotes the fraction of observations to be	

	samples for each tree.	
colsample_bylevel	Denotes the subsample ratio of columns for each split in each level.	Usually not used
lambda	L2 regularization term on weights (analogous to Ridge regression).	Explore for reducing overfitting
alpha	L1 regularization term on weight (analogous to Lasso regression).	Used for high dimensionality
scale_pos_weight	Used in case of high-class imbalance for faster convergence.	> 0

Learning Task Parameters

These parameters are used to define the optimization objective and the metric to be calculated at each step.

1. **objective [default=reg:linear]**
    - This defines the loss function to be minimized. Mostly used values are:
        - **binary: logistic** –logistic regression for binary classification returns predicted probability (not class)
        - **multi: softmax** –multiclass classification using the softmax objective, returns predicted class (not probabilities)
            - you also need to set an additional **num_class** (number of classes) parameter defining the number of unique classes
        - **multi: softprob** –same as softmax, but returns predicted probability of each data point belonging to each class.
2. **eval_metric [default according to objective]**
    - The evaluation metrics are to be used for validation data.
    - The default values are rmse for regression and error for classification.
    - Typical values are:
        - **rmse** – root mean square error
        - **mae** – mean absolute error

- **merror** – Multiclass classification error rate
- **mlogloss** – Multiclass logloss
- **auc:** Area under the curve

3. **seed [default=0]**
   - The random number seed.
   - It can be used for generating reproducible results and also for parameter tuning.

If you've been using Scikit-Learn till now, these parameter names might not look familiar. The good news is that the xgboost module in python has an sklearn wrapper called XGBClassifier parameters. It uses the sklearn style naming convention. The parameters names that will change are:

1. eta –> learning_rate
2. lambda –> reg_lambda
3. alpha –> reg_alpha

You must be wondering why we have defined everything except something similar to the "n_estimators" parameter in GBM. Well, this exists as a parameter in XGBClassifier. However, it has to be passed as "num_boosting_rounds" while calling the fit function in the standard xgboost implementation.

Go through the following parts of the xgboost guide to better understand the parameters and codes:

1. XGBoost Parameters (official guide)
2. XGBoost Demo Codes (xgboost GitHub repository)
3. Python API Reference (official guide)

XGBoost Parameters Tuning With Example

We will take the data set from Data Hackathon 3.x AV hackathon, as taken in the GBM article. The details of the problem can be found on the competition page. You can download the data set from here. I have performed the following steps:

EMI_Loan_Submitted was missing; else 0 | Original variable EMI_Loan_Submitted dropped.

- EmployerName dropped because of too many categories.
- Existing_EMI imputed with 0 (median) since only 111 values were missing.
- Interest_Rate_Missing created, which is 1 if Interest_Rate was missing; else 0 | Original variable Interest_Rate dropped.
- Lead_Creation_Date dropped because it made a little intuitive impact on the outcome.
- Loan_Amount_Applied, Loan_Tenure_Applied imputed with median values.
- Loan_Amount_Submitted_Missing created, which is 1 if Loan_Amount_Submitted was missing; else 0 | Original variable Loan_Amount_Submitted dropped.
- Loan_Tenure_Submitted_Missing created, which is 1 if Loan_Tenure_Submitted was missing; else 0 | Original variable Loan_Tenure_Submitted dropped.
- LoggedIn, Salary_Account dropped.
- Processing_Fee_Missing created, which is 1 if Processing_Fee was missing; else 0 | Original variable Processing_Fee dropped.
- Source – top 2 kept as is, and all others are combined into a different category.
- Numerical and One-Hot-Coding performed.

For those who have the original data from the competition, you can check out these steps from the data_preparation iPython notebook in the repository.

Let's start by importing the required libraries and loading the data.

Python Code

Mastering XGBoost Parameters Tuning: A Complete Guide with Python Codes

main.py

```
import numpy as np
import xgboost as xgb
from xgboost.sklearn import XGBClassifier
from sklearn import metrics   #Additional scklearn functions
from sklearn.model_selection import GridSearchCV
import matplotlib.pylab as plt

from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 12, 4

train = pd.read_csv('Train_Modified.csv', encoding='ISO-8859-1')
target = 'Disbursed'
IDcol = 'ID'
```

Note that I have imported 2 forms of XGBoost:

1. **xgb** – this is the direct xgboost library. I will use a specific function, “cv” from this library
2. **XGBClassifier** – this is an sklearn wrapper for XGBoost. This allows us to use sklearn’s Grid Search with parallel processing in the same way we did for GBM.

Before proceeding further, let’s define a function that will help us create XGBoost models and perform cross-validation. The best part is that you can take this function as it is and use it later for your own models.

```
def modelfit(alg, dtrain, predictors,useTrainCV=True, cv_

    if useTrainCV:
        xgb_param = alg.get_xgb_params()
        xgtrain = xgb.DMatrix(dtrain[predictors].values,
        cvresult = xgb.cv(xgb_param, xgtrain, num_boost_r
            metrics='auc', early_stopping_rounds=early_st
        alg.set_params(n_estimators=cvresult.shape[0])

    #Fit the algorithm on the data
    alg.fit(dtrain[predictors], dtrain['Disbursed'],eval_

    #Predict training set:
    dtrain_predictions = alg.predict(dtrain[predictors])
    dtrain_predprob = alg.predict_proba(dtrain[predictors

    #Print model report:
    print "\nModel Report"
    print "Accuracy : %.4g" % metrics.accuracy_score(dtra
    print "AUC Score (Train): %f" % metrics.roc_auc_score

    feat_imp = pd.Series(alg.booster().get_fscore()).sort_
    feat_imp.plot(kind='bar', title='Feature Importances'
    plt.ylabel('Feature Importance Score')
```

Note that xgboost's sklearn wrapper doesn't have a "feature_importances" metric but a get_fscore() function, which does the same job.

General Approach for XGBoost Parameters Tuning

We will use an approach similar to that of GBM here. The various steps to be performed are:

1. Choose a relatively **high learning rate**. Generally, a learning rate of 0.1 works, but somewhere between 0.05 to 0.3 should work for different problems. Determine the **optimum number of trees for this learning rate**. XGBoost has a very useful function called "cv" which performs cross-validation at each boosting iteration and thus returns the optimum number of trees required.
2. **Tune tree-specific parameters** (max_depth, min_child_weight, gamma, subsample, colsample_bytree) for the decided learning rate and the number of trees. Note that we can choose different parameters to define a tree, and I'll take up an example here.
3. Tune **regularization parameters** (lambda, alpha) for xgboost, which can help reduce model complexity and enhance performance.
4. **Lower the learning rate** and decide the optimal parameters.

Let us look at a more detailed step-by-step approach.

Step 1: Fix the learning rate and number of estimators for tuning tree-based parameters.

In order to decide on boosting parameters, we need to set some initial values of other parameters. Let's take the following values:

1. **max_depth = 5**: This should be between 3-10. I've started with 5 but you can choose a different number

Mastering XGBoost Parameters Tuning: A Complete Guide with Python Codes

leaf nodes can have smaller size groups.

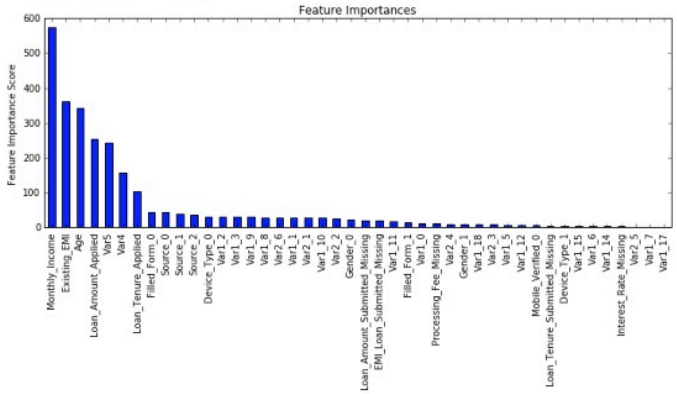
3. **gamma = 0**: A smaller value like 0.1-0.2 can also be chosen for starting. This will, anyways, be tuned later.
4. **subsample, colsample_bytree = 0.8**: This is a commonly used start value. Typical values range between 0.5-0.9.
5. **scale_pos_weight = 1**: Because of high-class imbalance.

Please note that all the above are just initial estimates and will be tuned later. Let's take the default learning rate of 0.1 here and check the optimum number of trees using the cv function of xgboost. The function defined above will do it for us.

```
#Choose all predictors except target & IDcols
predictors = [x for x in train.columns if x not in [target
xgb1 = XGBClassifier(
 learning_rate =0.1,
 n_estimators=1000,
 max_depth=5,
 min_child_weight=1,
 gamma=0,
 subsample=0.8,
 colsample_bytree=0.8,
 objective= 'binary:logistic',
 nthread=4,
 scale_pos_weight=1,
 seed=27)
modelfit(xgb1, train, predictors)
```

```
Will train until cv error hasn't decreased in 50 rounds.
Stopping. Best iteration:
[140] cv-mean:0.843638  cv-std:0.0141274405467

Model Report
Accuracy : 0.9854
AUC Score (Train): 0.899857
AUC Score (Test): 0.847934
```



As you can see that here we got 140 as the optimal estimator for a 0.1 learning rate. Note that this value might

Note: You will see the test AUC as "AUC Score (Test)" in the outputs here. But this would not appear if you try to run the command on your system as the data is not made public. It's provided here just for reference. The part of the code which generates this output has been removed here.

Step 2: Tune **max_depth** and **min_child_weight.**

We tune these first as they will have the highest impact on the model outcome. To start with, let's set wider ranges, and then we will perform another iteration for smaller ranges.

Important Note: I'll be doing some heavy-duty grid searches in this section, which can take 15-30 mins or even more time to run, depending on your system. You can vary the number of values you are testing based on what your system can handle.

```
param_test1 = {
 'max_depth':range(3,10,2),
 'min_child_weight':range(1,6,2)
}
gsearch1 = GridSearchCV(estimator = XGBClassifier( learni
 min_child_weight=1, gamma=0, subsample=0.8, colsample_by
 objective= 'binary:logistic', nthread=4, scale_pos_weigh
 param_grid = param_test1, scoring='roc_auc',n_jobs=4,iid=
gsearch1.fit(train[predictors],train[target])
gsearch1.grid_scores_, gsearch1.best_params_, gsearch1.be
```

```
([mean: 0.83690, std: 0.00821, params: {'max_depth': 3, 'min_child_weight': 1},
  mean: 0.83730, std: 0.00858, params: {'max_depth': 3, 'min_child_weight': 3},
  mean: 0.83713, std: 0.00847, params: {'max_depth': 3, 'min_child_weight': 5},
  mean: 0.84051, std: 0.00748, params: {'max_depth': 5, 'min_child_weight': 1},
  mean: 0.84112, std: 0.00595, params: {'max_depth': 5, 'min_child_weight': 3},
  mean: 0.84123, std: 0.00619, params: {'max_depth': 5, 'min_child_weight': 5},
  mean: 0.83772, std: 0.00518, params: {'max_depth': 7, 'min_child_weight': 1},
  mean: 0.83672, std: 0.00579, params: {'max_depth': 7, 'min_child_weight': 3},
  mean: 0.83658, std: 0.00355, params: {'max_depth': 7, 'min_child_weight': 5},
  mean: 0.82690, std: 0.00622, params: {'max_depth': 9, 'min_child_weight': 1},
  mean: 0.82909, std: 0.00560, params: {'max_depth': 9, 'min_child_weight': 3},
  mean: 0.83211, std: 0.00707, params: {'max_depth': 9, 'min_child_weight': 5}],
 {'max_depth': 5, 'min_child_weight': 5},
 0.8412329282025758 9)
```

Here, we have run 12 combinations with wider intervals between values. The ideal values are **5 for max_depth** and **5 for min_child_weight**. Let's go one step deeper and look for optimum values. We'll search for values 1 above and below the optimum values because we took an interval of two.

```
param_test2 = {
```

Mastering XGBoost Parameters Tuning: A Complete Guide with Python Codes

```
gsearch2.fit(train[predictors],train[target])
gsearch2.grid_scores_, gsearch2.best_params_, gsearch2.bes
```

```
([mean: 0.84031, std: 0.00658, params: {'max_depth': 4, 'min_child_weight': 4},
  mean: 0.84061, std: 0.00700, params: {'max_depth': 4, 'min_child_weight': 5},
  mean: 0.84125, std: 0.00723, params: {'max_depth': 4, 'min_child_weight': 6},
  mean: 0.83988, std: 0.00612, params: {'max_depth': 5, 'min_child_weight': 4},
  mean: 0.84123, std: 0.00619, params: {'max_depth': 5, 'min_child_weight': 5},
  mean: 0.83995, std: 0.00591, params: {'max_depth': 5, 'min_child_weight': 6},
  mean: 0.83905, std: 0.00635, params: {'max_depth': 6, 'min_child_weight': 4},
  mean: 0.83904, std: 0.00656, params: {'max_depth': 6, 'min_child_weight': 5},
  mean: 0.83844, std: 0.00682, params: {'max_depth': 6, 'min_child_weight': 6}],
 {'max_depth': 4, 'min_child_weight': 6},
 0.84124915179964577)
```

Here, we get the optimum values as **4 for max_depth** and **6 for min_child_weight**. Also, we can see the CV score increasing slightly. Note that as the model performance increases, it becomes exponentially difficult to achieve even marginal gains in performance. You would have noticed that here we got 6 as the optimum value for min_child_weight, but we haven't tried values more than 6. We can do that as follow:

```
param_test2b = {
 'min_child_weight':[6,8,10,12]
}
gsearch2b = GridSearchCV(estimator = XGBClassifier( learn
 min_child_weight=2, gamma=0, subsample=0.8, colsample_by
 objective= 'binary:logistic', nthread=4, scale_pos_weigh
 param_grid = param_test2b, scoring='roc_auc',n_jobs=4,ii
gsearch2b.fit(train[predictors],train[target])
```

```
modelfit(gsearch3.best_estimator_, train, predictors)
gsearch2b.grid_scores_, gsearch2b.best_params_, gsearch2b.
```

```
([mean: 0.84125, std: 0.00723, params: {'min_child_weight': 6},
  mean: 0.84028, std: 0.00710, params: {'min_child_weight': 8},
  mean: 0.83920, std: 0.00674, params: {'min_child_weight': 10},
  mean: 0.83996, std: 0.00729, params: {'min_child_weight': 12}],
 {'min_child_weight': 6},
 0.84124915179964577)
```

We see 6 as the optimal value.

Step 3: Tune gamma.

Now let's tune the gamma value using the parameters already tuned above. Gamma can take various values, but I'll check for 5 values here. You can go into more precise values.

```
param_test3 = {
 'gamma':[i/10.0 for i in range(0,5)]
}
gsearch3 = GridSearchCV(estimator = XGBClassifier( learni
 min_child_weight=6, gamma=0, subsample=0.8, colsample_by
 objective= 'binary:logistic', nthread=4, scale_pos_weigh
```

Mastering XGBoost Parameters Tuning: A Complete Guide with Python Codes

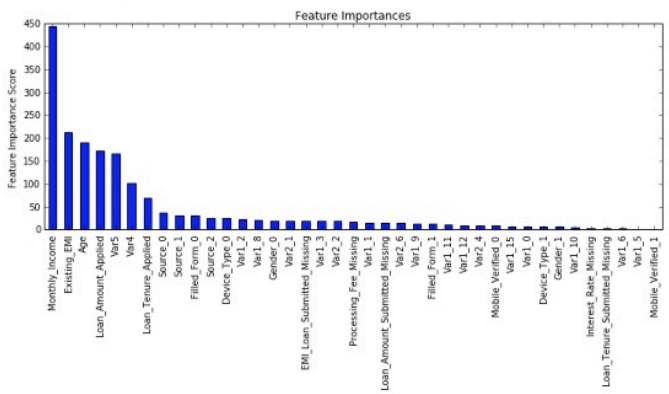
```
([mean: 0.84125, std: 0.00723, params: {'gamma': 0.0},
  mean: 0.83996, std: 0.00695, params: {'gamma': 0.1},
  mean: 0.84045, std: 0.00639, params: {'gamma': 0.2},
  mean: 0.84032, std: 0.00673, params: {'gamma': 0.3},
  mean: 0.84061, std: 0.00692, params: {'gamma': 0.4}],
 {'gamma': 0.0},
 0.84124915179964577)
```

This shows that our original value of gamma, i.e., **0 is the optimum one**. Before proceeding, a good idea would be to re-calibrate the number of boosting rounds for the updated parameters.

```
xgb2 = XGBClassifier(
 learning_rate =0.1,
 n_estimators=1000,
 max_depth=4,
 min_child_weight=6,
 gamma=0,
 subsample=0.8,
 colsample_bytree=0.8,
 objective= 'binary:logistic',
 nthread=4,
 scale_pos_weight=1,
 seed=27)
modelfit(xgb2, train, predictors)
```

```
Will train until cv error hasn't decreased in 50 rounds.
Stopping. Best iteration:
[177] cv-mean:0.8451166 cv-std:0.0123406045006
```

```
Model Report
Accuracy : 0.9854
AUC Score (Train): 0.883836
AUC Score (Test): 0.848967
```



Here, We can see the score improvement. so, the final parameters are:

- max_depth: 4
- min_child_weight: 6
- gamma: 0

Step 4: Tune subsample and colsample_bytree

and take values 0.6,0.7,0.8,0.9 for both to start with.

```
param_test4 = {
 'subsample':[i/10.0 for i in range(6,10)],
 'colsample_bytree':[i/10.0 for i in range(6,10)]
}
gsearch4 = GridSearchCV(estimator = XGBClassifier( learni
 min_child_weight=6, gamma=0, subsample=0.8, colsample_by
 objective= 'binary:logistic', nthread=4, scale_pos_weigh
 param_grid = param_test4, scoring='roc_auc',n_jobs=4,iid=
gsearch4.fit(train[predictors],train[target])
gsearch4.grid_scores_, gsearch4.best_params_, gsearch4.bes
```

```
([mean: 0.83688, std: 0.00849, params: {'subsample': 0.6, 'colsample_bytree': 0.6},
  mean: 0.83834, std: 0.00772, params: {'subsample': 0.7, 'colsample_bytree': 0.6},
  mean: 0.83946, std: 0.00813, params: {'subsample': 0.8, 'colsample_bytree': 0.6},
  mean: 0.83845, std: 0.00831, params: {'subsample': 0.9, 'colsample_bytree': 0.6},
  mean: 0.83816, std: 0.00651, params: {'subsample': 0.6, 'colsample_bytree': 0.7},
  mean: 0.83797, std: 0.00668, params: {'subsample': 0.7, 'colsample_bytree': 0.7},
  mean: 0.83956, std: 0.00824, params: {'subsample': 0.8, 'colsample_bytree': 0.7},
  mean: 0.83892, std: 0.00626, params: {'subsample': 0.9, 'colsample_bytree': 0.7},
  mean: 0.83914, std: 0.00794, params: {'subsample': 0.6, 'colsample_bytree': 0.8},
  mean: 0.83974, std: 0.00687, params: {'subsample': 0.7, 'colsample_bytree': 0.8},
  mean: 0.84102, std: 0.00715, params: {'subsample': 0.8, 'colsample_bytree': 0.8},
  mean: 0.84029, std: 0.00645, params: {'subsample': 0.9, 'colsample_bytree': 0.8},
  mean: 0.83881, std: 0.00723, params: {'subsample': 0.6, 'colsample_bytree': 0.9},
  mean: 0.83975, std: 0.00706, params: {'subsample': 0.7, 'colsample_bytree': 0.9},
  mean: 0.83975, std: 0.00648, params: {'subsample': 0.8, 'colsample_bytree': 0.9},
  mean: 0.83954, std: 0.00698, params: {'subsample': 0.9, 'colsample_bytree': 0.9}],
 {'colsample_bytree': 0.8, 'subsample': 0.8},
 0.8410246925643593)
```

Here, we found **0.8 as the optimum value for both** subsample and colsample_bytree. Now we should try values in 0.05 intervals around these.

```
param_test5 = {
 'subsample':[i/100.0 for i in range(75,90,5)],
 'colsample_bytree':[i/100.0 for i in range(75,90,5)]
}
gsearch5 = GridSearchCV(estimator = XGBClassifier( learni
 min_child_weight=6, gamma=0, subsample=0.8, colsample_by
 objective= 'binary:logistic', nthread=4, scale_pos_weigh
 param_grid = param_test5, scoring='roc_auc',n_jobs=4,iid=
gsearch5.fit(train[predictors],train[target])
```

```
([mean: 0.83881, std: 0.00795, params: {'subsample': 0.75, 'colsample_bytree': 0.75},
  mean: 0.84037, std: 0.00638, params: {'subsample': 0.8, 'colsample_bytree': 0.75},
  mean: 0.84013, std: 0.00685, params: {'subsample': 0.85, 'colsample_bytree': 0.75},
  mean: 0.83967, std: 0.00694, params: {'subsample': 0.75, 'colsample_bytree': 0.8},
  mean: 0.84102, std: 0.00715, params: {'subsample': 0.8, 'colsample_bytree': 0.8},
  mean: 0.84087, std: 0.00693, params: {'subsample': 0.85, 'colsample_bytree': 0.8},
  mean: 0.83836, std: 0.00738, params: {'subsample': 0.75, 'colsample_bytree': 0.85},
  mean: 0.84067, std: 0.00698, params: {'subsample': 0.8, 'colsample_bytree': 0.85},
  mean: 0.83978, std: 0.00689, params: {'subsample': 0.85, 'colsample_bytree': 0.85}],
 {'colsample_bytree': 0.8, 'subsample': 0.8},
 0.8410246925643593)
```

Again we got the same values as before. Thus the optimum values are:

- subsample: 0.8
- colsample_bytree: 0.8

Step 5: Tuning regularization parameters

The next step is to apply regularization to reduce overfitting. However, many people don't use this

Mastering XGBoost Parameters Tuning: A Complete Guide with Python Codes

```
param_test6 = {
 'reg_alpha':[1e-5, 1e-2, 0.1, 1, 100]
}
gsearch6 = GridSearchCV(estimator = XGBClassifier( learni
 min_child_weight=6, gamma=0.1, subsample=0.8, colsample_
 objective= 'binary:logistic', nthread=4, scale_pos_weigh
 param_grid = param_test6, scoring='roc_auc',n_jobs=4,iid=
gsearch6.fit(train[predictors],train[target])
gsearch6.grid_scores_, gsearch6.best_params_, gsearch6.bes
```

```
([mean: 0.83999, std: 0.00643, params: {'reg_alpha': 1e-05},
  mean: 0.84084, std: 0.00639, params: {'reg_alpha': 0.01},
  mean: 0.83985, std: 0.00831, params: {'reg_alpha': 0.1},
  mean: 0.83989, std: 0.00707, params: {'reg_alpha': 1},
  mean: 0.81343, std: 0.01541, params: {'reg_alpha': 100}],
 {'reg_alpha': 0.01},
 0.84084269674772316)
```

We can see that the CV score is less than in the previous case. But the values tried are very widespread. We should try values closer to the optimum here (0.01) to see if we get something better.

```
param_test7 = {
 'reg_alpha':[0, 0.001, 0.005, 0.01, 0.05]
}
gsearch7 = GridSearchCV(estimator = XGBClassifier( learni
 min_child_weight=6, gamma=0.1, subsample=0.8, colsample_
 objective= 'binary:logistic', nthread=4, scale_pos_weigh
 param_grid = param_test7, scoring='roc_auc',n_jobs=4,iid=
gsearch7.fit(train[predictors],train[target])
gsearch7.grid_scores_, gsearch7.best_params_, gsearch7.bes
```

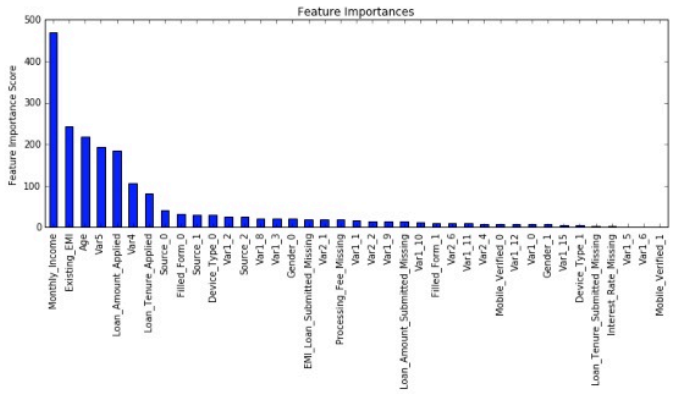
```
([mean: 0.83999, std: 0.00643, params: {'reg_alpha': 0},
  mean: 0.83978, std: 0.00663, params: {'reg_alpha': 0.001},
  mean: 0.84118, std: 0.00651, params: {'reg_alpha': 0.005},
  mean: 0.84084, std: 0.00639, params: {'reg_alpha': 0.01},
  mean: 0.84008, std: 0.00690, params: {'reg_alpha': 0.05}],
 {'reg_alpha': 0.005},
 0.84118352535245489)
```

You can see that we got a better CV. Now we can apply this regularization in the model and look at the impact:

```
xgb3 = XGBClassifier(
 learning_rate =0.1,
 n_estimators=1000,
 max_depth=4,
 min_child_weight=6,
 gamma=0,
 subsample=0.8,
 colsample_bytree=0.8,
 reg_alpha=0.005,
 objective= 'binary:logistic',
 nthread=4,
 scale_pos_weight=1,
 seed=27)
modelfit(xgb3, train, predictors)
```

Mastering XGBoost Parameters Tuning: A Complete Guide with Python Codes

```
AUC Score (Train): 0.887149
AUC Score (Test): 0.848972
```



Again we can see a slight improvement in the score.

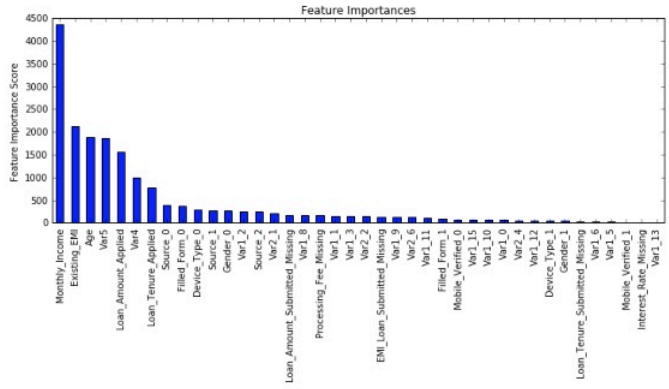
Step 6: Reducing the learning rate

Lastly, we should lower the learning rate and add more trees. Let's use the cv function of XGBoost to do the job again.

```
xgb4 = XGBClassifier(
 learning_rate =0.01,
 n_estimators=5000,
 max_depth=4,
 min_child_weight=6,
 gamma=0,
 subsample=0.8,
 colsample_bytree=0.8,
 reg_alpha=0.005,
 objective= 'binary:logistic',
 nthread=4,
 scale_pos_weight=1,
 seed=27)
modelfit(xgb4, train, predictors)
```

```
Will train until cv error hasn't decreased in 50 rounds.
Stopping. Best iteration:
[1732] cv-mean:0.8452782        cv-std:0.0126670016879
```

```
Model Report
Accuracy : 0.9854
AUC Score (Train): 0.885261
AUC Score (Test): 0.849430
```



Here is a live coding window where you can try different

main.py

```
                nthread=4,
                scale_pos_weight=1,
                seed=27),
                param_grid = param_test,
                scoring='roc_auc',
                n_jobs=4,
                iid=False,
                cv=2,
                verbose=10)

gsearch.fit(train[predictors],train[target])

print('Best Grid Search Parameters
:',gsearch.best_params_)
print('Best Grid Search Score : ',gsearch.best_score_)
```

Now we can see a significant boost in performance, and the effect of parameter tuning is clearer.

As we come to an end, I would like to share 2 key thoughts:

1. It is **difficult to get a very big leap** in performance by just using **parameter tuning** or **slightly better models**. The max score for GBM was 0.8487, while XGBoost gave 0.8494. This is a decent improvement but not something very substantial.
2. A significant jump can be obtained by other methods like **feature engineering**, creating an **ensemble** of models, **stacking**, etc.

You can also download the iPython notebook with all these model codes from my GitHub account. For codes in R, you can refer to this article.

Master XGBoost Parameters

This tutorial was based on developing an XGBoost machine learning model end-to-end. We started by discussing **why XGBoost Parameters has superior performance over GBM,** which was followed by a detailed discussion of the **various parameters** involved. We also defined a generic function that you can reuse for making models. Finally, we discussed the **general approach** towards tackling a problem with XGBoost and also worked

algorithm, especially where speed and accuracy are concerned.

- We need to consider different parameters and their values to be specified while implementing an XGBoost model.
- The XGBoost model requires parameter tuning to improve and fully leverage its advantages over other algorithms.

If you're looking to take your machine learning skills to the next level, consider enrolling in our Data Science Black Belt program. The curriculum covers all aspects of data science, including advanced topics like XGBoost parameter tuning. With hands-on projects and mentorship, you'll gain practical experience and the skills you need to succeed in this exciting field. Enroll today and take your XGBoost parameters tuning skills and overall data science expertise to the next level!

Frequently Asked Questions

Gradient Boosting | grid search | live coding | parameter tuning in xgboost | python | sklearn | xgbclassifier parameters | XGBoost | XGBoost classifer | xgboost model

Aarshay Jain
07 Jan 2024

Aarshay graduated from MS in Data Science at Columbia University in 2017 and is currently an ML Engineer at Spotify New York. He works at an intersection or applied research and engineering while designing ML solutions to move product metrics in the required direction. He specializes in designing ML system architecture, developing offline models and deploying them in production for both batch and real time prediction use cases.

Python Structured Data

Frequently Asked Questions

Q1. What parameters should you use for XGBoost?

A. The choice of XGBoost parameters depends on the specific task. Commonly adjusted parameters include learning rate (eta), maximum tree depth (max_depth), and minimum child weight (min_child_weight).

What does the 'N_estimators' parameter signify in XGBoost?

Q3. How do you define a hyperparameter in XGBoost?

Q4. What purpose do regularization parameters serve in XGBoost?

Responses From Readers

What are your thoughts?...

Submit reply

Prateek
02 Mar, 2016

Please provide the R code as well. Thnkx

Ankur Bhargava
02 Mar, 2016

It is a great article , but if you could provide codes in R , it would be more beneficial to us. Thanks

Mastering XGBoost Parameters Tuning: A Complete Guide with Python Codes

article (http://www.analyticsvidhya.com/blog/2016/01/xgboost-algorithm-easy-steps/) in my above article. This has

Mastering XGBoost Parameters Tuning: A Complete Guide with Python Codes