



HOME LEARN [ENGAGE](#) [COMPETE](#) [GET HIRED](#) [AVBYTES](#) [CONTACT US](#)

[WE ARE HIRING!](#) [CAMPUS AMBASSADOR](#) [CORPORATE](#)

Home [Machine Learning](#) [Complete Guide to Parameter Tuning in Gradient Boosting \(GBM\) in Python](#)

Complete Guide to Parameter Tuning in Gradient Boosting (GBM) in Python

MACHINE LEARNING PYTHON

AARSHAY JAIN , FEBRUARY 21, 2016 / 22 [SHARE](#) [Facebook](#) [Twitter](#) [LinkedIn](#) [Reddit](#)

TOP ANALYTICS VIDHYA USERS

Rank	Name	Points
1	vopani	8704
2	srk	8287
3	aayushmnit	7128
4	binga	5319
5	sonny	5266

[More Rankings](#)

Introduction

If you have been using GBM as a 'black box' till now, may be it's time for you to open it and see, how it actually works!

This article is inspired by Owen Zhang's (Chief Product Officer at DataRobot and Kaggle Rank 3) approach shared at [NYC Data Science Academy](#). He delivered a ~2 hours talk and I intend to condense it and present the most precious nuggets here.

Boosting algorithms play a crucial role in dealing with bias variance trade-off. Unlike bagging algorithms, which only controls for high variance in a model, boosting controls both the aspects (bias & variance), and is considered to be more effective. A sincere understanding of GBM here should give you much needed confidence to deal with such critical issues.

In this article, I'll disclose the science behind using GBM using Python. And, most important, how you can tune its parameters and obtain incredible results.

Special Thanks: Personally, I would like to acknowledge the timeless support provided by [Mr. Sudalai Rajkumar](#), currently [AV Rank 2](#). This article wouldn't be possible without his guidance. I am sure the whole community will benefit from the same.

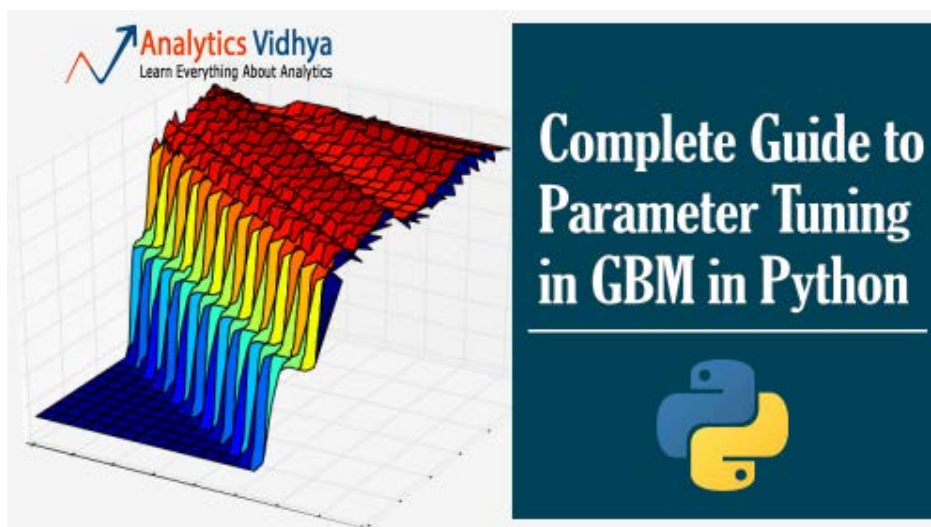


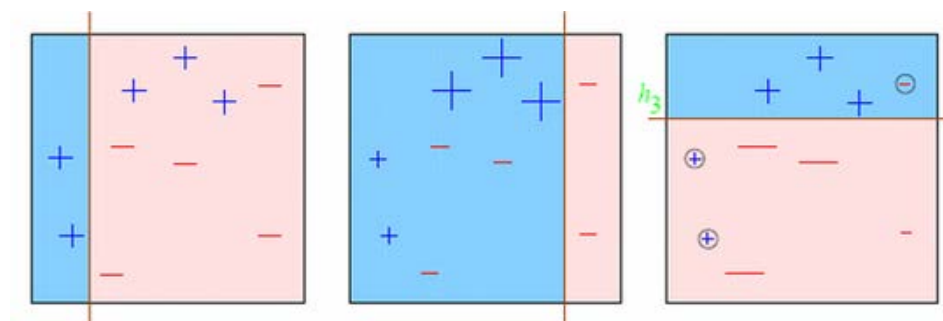
Table of Contents

1. How Boosting Works?
2. Understanding GBM Parameters
3. Tuning Parameters (with Example)

1. How Boosting Works ?

Boosting is a sequential technique which works on the principle of **ensemble**. It combines a set of **weak learners** and delivers improved prediction accuracy. At any instant t , the model outcomes are weighed based on the outcomes of previous instant $t-1$. The outcomes predicted correctly are given a lower weight and the ones miss-classified are weighted higher. This technique is followed for a classification problem while a similar technique is used for regression.

Let's understand it visually:



Observations:

1. **Box 1: Output of First Weak Learner (From the left)**
 - Initially all points have same weight (denoted by their size).
 - The decision boundary predicts 2 +ve and 5 -ve points correctly.
2. **Box 2: Output of Second Weak Learner**
 - The points classified correctly in box 1 are given a lower weight and vice versa.
 - The model focuses on high weight points now and classifies them correctly. But, others are misclassified

POPULAR POSTS

- Essentials of Machine Learning Algorithms (with Python and R Codes)
- A Complete Tutorial to Learn Data Science with Python from Scratch
- 7 Types of Regression Techniques you should know!
- Understanding Support Vector Machine algorithm from examples (along with code)
- 6 Easy Steps to Learn Naive Bayes Algorithm (with codes in Python and R)
- A Complete Tutorial on Tree Based Modeling from Scratch (in R & Python)
- A comprehensive beginner's guide to create a Time Series Forecast (with Codes in Python)
- 10 Free Must-Read Machine Learning E-Books For Data Scientists & AI Engineers

now.

Similar trend can be seen in box 3 as well. This continues for many iterations. In the end, all models are given a weight depending on their accuracy and a consolidated result is generated.

Did I whet your appetite ? Good. Refer to these articles (focus on GBM right now):

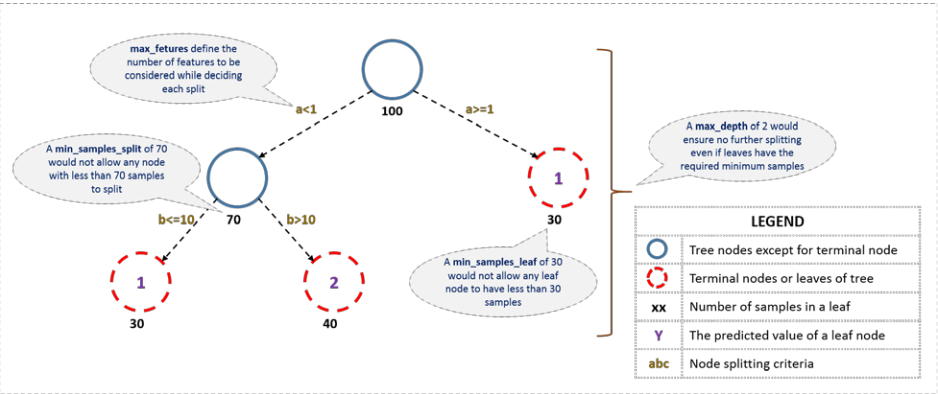
- [Learn Gradient Boosting Algorithm for better predictions \(with codes in R\)](#)
- [Quick Introduction to Boosting Algorithms in Machine Learning](#)
- [Getting smart with Machine Learning – AdaBoost and Gradient Boost](#)

2. GBM Parameters

The overall parameters can be divided into 3 categories:

1. **Tree-Specific Parameters:** These affect each individual tree in the model.
2. **Boosting Parameters:** These affect the boosting operation in the model.
3. **Miscellaneous Parameters:** Other parameters for overall functioning.

I'll start with tree-specific parameters. First, lets look at the general structure of a decision tree:



The parameters used for defining a tree are further explained below. Note that I'm using scikit-learn (python) specific

RECENT POSTS

Interview with Margherita Pagani and Clément Levallois, Program Co-Directors, MSc in Digital Marketing and Data Science, emlyon business school
PRANAV DAR , ...

19 Inspiring Women Blazing a Trail in the Data Science World
PRANAV DAR , ...

terminologies here which might be different in other software packages like R. But the idea remains the same.

1. **min_samples_split**

- Defines the minimum number of samples (or observations) which are required in a node to be considered for splitting.
- Used to control over-fitting. Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree.
- Too high values can lead to under-fitting hence, it should be tuned using CV.

2. **min_samples_leaf**

- Defines the minimum samples (or observations) required in a terminal node or leaf.
- Used to control over-fitting similar to min_samples_split.
- Generally lower values should be chosen for imbalanced class problems because the regions in which the minority class will be in majority will be very small.

3. **min_weight_fraction_leaf**

- Similar to min_samples_leaf but defined as a fraction of the total number of observations instead of an integer.
- Only one of #2 and #3 should be defined.

4. **max_depth**

- The maximum depth of a tree.
- Used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.
- Should be tuned using CV.

5. **max_leaf_nodes**

- The maximum number of terminal nodes or leaves in a tree.
- Can be defined in place of max_depth. Since binary trees are created, a depth of 'n' would produce a maximum of 2^n leaves.
- If this is defined, GBM will ignore max_depth.

6. **max_features**

- The number of features to consider while searching for a best split. These will be randomly selected.
- As a thumb-rule, square root of the total number of features works great but we should check upto 30-40% of the total number of features.
- Higher values can lead to over-fitting but depends on case

How to create a poet / writer using Deep Learning (Text Generation using Python)?

PRANJAL SRIVA...

AVBytes: AI & ML Development s this week – Google Brain's Image Manipulation, Record-Breaking AI, Stanford's ML Model Predicts Poverty

PRANAV DAR , ...

to case.

Before moving on to other parameters, let's see the overall pseudo-code of the GBM algorithm for 2 classes:

1. Initialize the outcome
2. Iterate from 1 to total number of trees
 - 2.1 Update the weights for targets based on previous run (higher for the ones mis-classified)
 - 2.2 Fit the model on selected subsample of data
 - 2.3 Make predictions on the full set of observations
 - 2.4 Update the output with current results taking into account the learning rate
3. Return the final output.

This is an extremely simplified (probably naive) explanation of GBM's working. The parameters which we have considered so far will affect step 2.2, i.e. model building. Let's consider another set of parameters for managing boosting:

1. `learning_rate`

- This determines the impact of each tree on the final outcome (step 2.4). GBM works by starting with an initial estimate which is updated using the output of each tree. The learning parameter controls the magnitude of this change in the estimates.
- Lower values are generally preferred as they make the model robust to the specific characteristics of tree and thus allowing it to generalize well.
- Lower values would require higher number of trees to model all the relations and will be computationally expensive.

2. `n_estimators`

- The number of sequential trees to be modeled (step 2)
- Though GBM is fairly robust at higher number of trees but it can still overfit at a point. Hence, this should be tuned using CV for a particular learning rate.

3. `subsample`

- The fraction of observations to be selected for each tree. Selection is done by random sampling.
- Values slightly less than 1 make the model robust by



GET CONNECTED



reducing the variance.

- Typical values ~ 0.8 generally work fine but can be fine-tuned further.

Apart from these, there are certain miscellaneous parameters which affect overall functionality:

1. **loss**

- It refers to the loss function to be minimized in each split.
- It can have various values for classification and regression case. Generally the default values work fine. Other values should be chosen only if you understand their impact on the model.

2. **init**

- This affects initialization of the output.
- This can be used if we have made another model whose outcome is to be used as the initial estimates for GBM.

3. **random_state**

- The random number seed so that same random numbers are generated every time.
- This is important for parameter tuning. If we don't fix the random number, then we'll have different outcomes for subsequent runs on the same parameters and it becomes difficult to compare models.
- It can potentially result in overfitting to a particular random sample selected. We can try running models for different random samples, which is computationally expensive and generally not used.

4. **verbose**

- The type of output to be printed when the model fits. The different values can be:
 - 0: no output generated (default)
 - 1: output generated for trees in certain intervals
 - >1: output generated for all trees

5. **warm_start**

- This parameter has an interesting application and can help a lot if used judiciously.
- Using this, we can fit additional trees on previous fits of a model. It can save a lot of time and you should explore



this option for advanced applications

6. presort

- Select whether to presort data for faster splits.
- It makes the selection automatically by default but it can be changed if needed.

I know its a long list of parameters but I have simplified it for you in an excel file which you can download from my [GitHub repository](#).

3. Parameter Tuning with Example

We will take the dataset from Data Hackathon 3.x AV hackathon. 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:

1. City variable dropped because of too many categories
2. DOB converted to Age | DOB dropped
3. EMI_Loan_Submitted_Missing created which is 1 if EMI_Loan_Submitted was missing else 0 | Original variable EMI_Loan_Submitted dropped
4. EmployerName dropped because of too many categories
5. Existing_EMI imputed with 0 (median) since only 111 values were missing
6. Interest_Rate_Missing created which is 1 if Interest_Rate was missing else 0 | Original variable Interest_Rate dropped
7. Lead_Creation_Date dropped because made little intuitive impact on outcome
8. Loan_Amount_Applied, Loan_Tenure_Applied imputed with median values
9. Loan_Amount_Submitted_Missing created which is 1 if Loan_Amount_Submitted was missing else 0 | Original variable Loan_Amount_Submitted dropped
10. Loan_Tenure_Submitted_Missing created which is 1 if Loan_Tenure_Submitted was missing else 0 | Original variable Loan_Tenure_Submitted dropped
11. LoggedIn, Salary_Account dropped
12. Processing_Fee_Missing created which is 1 if Processing_Fee was missing else 0 | Original variable Processing_Fee dropped
13. Source – top 2 kept as is and all others combined into different

category

14. Numerical and One-Hot-Coding performed

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

Lets start by importing the required libraries and loading the data:

```
#Import libraries:
import pandas as pd
import numpy as np
from sklearn.ensemble import GradientBoostingClassifier #GBM
algorithm
from sklearn import cross_validation, metrics #Additional sc
klearn functions
from sklearn.grid_search import GridSearchCV #Perforing grid
search

import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib.pyplot import rcParams
rcParams['figure.figsize'] = 12, 4

train = pd.read_csv('train_modified.csv')
target = 'Disbursed'
IDcol = 'ID'
```

Before proceeding further, lets define a function which will help us create GBM models and perform cross-validation.

```
def modelfit(alg, dtrain, predictors, performCV=True, printFea
tureImportance=True, cv_folds=5):
    #Fit the algorithm on the data
    alg.fit(dtrain[predictors], dtrain['Disbursed'])

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

```

dtrain_predprob = alg.predict_proba(dtrain[predictors])[:,
1]

#Perform cross-validation:
if performCV:
    cv_score = cross_validation.cross_val_score(alg, dtrain[predictors], dtrain['Disbursed'], cv=cv_folds, scoring='roc_auc')

#Print model report:
print "\nModel Report"
print "Accuracy : %.4g" % metrics.accuracy_score(dtrain['Disbursed'].values, dtrain_predictions)
print "AUC Score (Train): %f" % metrics.roc_auc_score(dtrain['Disbursed'], dtrain_predprob)

if performCV:
    print "CV Score : Mean - %.7g | Std - %.7g | Min - %.7g | Max - %.7g" % (np.mean(cv_score), np.std(cv_score), np.min(cv_score), np.max(cv_score))

#Print Feature Importance:
if printFeatureImportance:
    feat_imp = pd.Series(alg.feature_importances_, predictors).sort_values(ascending=False)
    feat_imp.plot(kind='bar', title='Feature Importances')
    plt.ylabel('Feature Importance Score')

```

The code is pretty self-explanatory. Please feel free to drop a note in the comments if you find any challenges in understanding any part of it.

Lets start by creating a **baseline model**. In this case, the evaluation metric is AUC so using any constant value will give 0.5 as result. Typically, a good baseline can be a GBM model with default parameters, i.e. without any tuning. Lets find out what it gives:

```
#Choose all predictors except target & IDcols
predictors = [x for x in train.columns if x not in [target, ID
col]]

gbm0 = GradientBoostingClassifier(random_state=10)

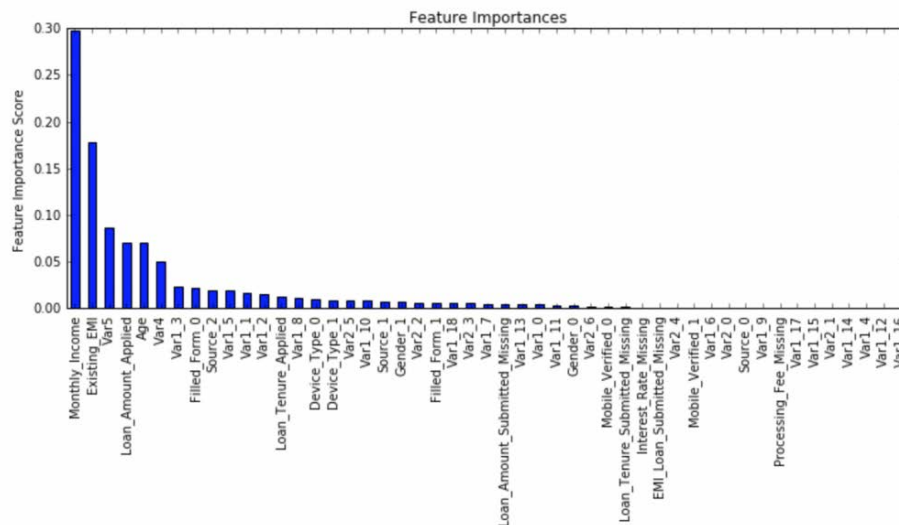
modelfit(gbm0, train, predictors)
```

Model Report

Accuracy : 0.9856

AUC Score (Train): 0.862264

CV Score : Mean - 0.8319 | Std - 0.008757 | Min - 0.8208 | Max - 0.8439



So, the mean CV score is 0.8319 and we should expect our model to do better than this.

General Approach for Parameter Tuning

As discussed earlier, there are two types of parameter to be tuned here – tree based and boosting parameters. There are no optimum values for learning rate as low values always work better, given that we train on sufficient number of trees.

Though, GBM is robust enough to not overfit with increasing trees, but a high number for pa particular learning rate can lead to overfitting. But as we reduce the learning rate and increase trees, the computation becomes expensive and would take a long time to run on standard personal computers.

Keeping all this in mind, we can take the following approach:

1. Choose a relatively **high learning rate**. Generally the default value of 0.1 works but somewhere between 0.05 to 0.2 should work for different problems
2. Determine the **optimum number of trees for this learning rate**. This should range around 40-70. Remember to choose a value on which your system can work fairly fast. This is because it will be used for testing various scenarios and determining the tree parameters.
3. **Tune tree-specific parameters** for decided learning rate and number of trees. Note that we can choose different parameters to define a tree and I'll take up an example here.
4. **Lower the learning rate** and increase the estimators proportionally to get more robust models.

Fix 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. Lets take the following values:

1. **min_samples_split = 500** : This should be ~0.5-1% of total values. Since this is imbalanced class problem, we'll take a small value from the range.
2. **min_samples_leaf = 50** : Can be selected based on intuition. This is just used for preventing overfitting and again a small value because of imbalanced classes.
3. **max_depth = 8** : Should be chosen (5-8) based on the number of observations and predictors. This has 87K rows and 49 columns so lets take 8 here.
4. **max_features = 'sqrt'** : Its a general thumb-rule to start with square root.
5. **subsample = 0.8** : This is a commonly used used start value

Please note that all the above are just initial estimates and will be tuned later. Lets take the default learning rate of 0.1 here and check the optimum number of trees for that. For this purpose, we can do a grid search and test out values from 20 to 80 in steps of 10.

```
#Choose all predictors except target & IDcols
predictors = [x for x in train.columns if x not in [target, ID
```

```
col]]

param_test1 = {'n_estimators':range(20,81,10)}

gsearch1 = GridSearchCV(estimator = GradientBoostingClassifier
(learning_rate=0.1, min_samples_split=500,min_samples_leaf=50,
max_depth=8,max_features='sqrt',subsample=0.8,random_state=10)
,
param_grid = param_test1, scoring='roc_auc',n_jobs=4,iid=False
, cv=5)

gsearch1.fit(train[predictors],train[target])
```

The output can be checked using following command:

```
gsearch1.grid_scores_, gsearch1.best_params_, gsearch1.best_score_
```

```
([mean: 0.83322, std: 0.00985, params: {'n_estimators': 20},
mean: 0.83684, std: 0.00986, params: {'n_estimators': 30},
mean: 0.83752, std: 0.00978, params: {'n_estimators': 40},
mean: 0.83761, std: 0.00991, params: {'n_estimators': 50},
mean: 0.83843, std: 0.00987, params: {'n_estimators': 60},
mean: 0.83832, std: 0.00956, params: {'n_estimators': 70},
mean: 0.83764, std: 0.01001, params: {'n_estimators': 80}],
{'n_estimators': 60},
0.83842766395593704)
```

As you can see that here we got 60 as the optimal estimators for 0.1 learning rate. Note that 60 is a reasonable value and can be used as it is. But it might not be the same in all cases. Other situations:

1. If the value is around 20, you might want to try lowering the learning rate to 0.05 and re-run grid search
2. If the values are too high ~100, tuning the other parameters will take long time and you can try a higher learning rate

Tuning tree-specific parameters

Now lets move onto tuning the tree parameters. I plan to do this in following stages:

1. Tune max_depth and num_samples_split

2. Tune min_samples_leaf
3. Tune max_features

The **order of tuning** variables should be decided carefully. You should take the variables with a higher impact on outcome first. For instance, max_depth and min_samples_split have a significant impact and we're tuning those first.

Important Note: I'll be doing some heavy-duty grid searched 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.

To start with, I'll test max_depth values of 5 to 15 in steps of 2 and min_samples_split from 200 to 1000 in steps of 200. These are just based on my intuition. You can set wider ranges as well and then perform multiple iterations for smaller ranges.

```
param_test2 = {'max_depth':range(5,16,2), 'min_samples_split':
range(200,1001,200)}
gsearch2 = GridSearchCV(estimator = GradientBoostingClassifier
(learning_rate=0.1, n_estimators=60, max_features='sqrt', subs
ample=0.8, random_state=10),
param_grid = param_test2, scoring='roc_auc',n_jobs=4,iid=False
, cv=5)
gsearch2.fit(train[predictors],train[target])
gsearch2.grid_scores_, gsearch2.best_params_, gsearch2.best_sc
ore_
```

```
[{mean: 0.83256, std: 0.01272, params: {'min_samples_split': 200, 'max_depth': 5},
mean: 0.83285, std: 0.01016, params: {'min_samples_split': 400, 'max_depth': 5},
mean: 0.83386, std: 0.01415, params: {'min_samples_split': 600, 'max_depth': 5},
mean: 0.83379, std: 0.01169, params: {'min_samples_split': 800, 'max_depth': 5},
mean: 0.83338, std: 0.01268, params: {'min_samples_split': 1000, 'max_depth': 5},
mean: 0.83390, std: 0.00759, params: {'min_samples_split': 200, 'max_depth': 7},
mean: 0.83660, std: 0.00994, params: {'min_samples_split': 400, 'max_depth': 7},
mean: 0.83481, std: 0.00827, params: {'min_samples_split': 600, 'max_depth': 7},
mean: 0.83788, std: 0.01066, params: {'min_samples_split': 800, 'max_depth': 7},
mean: 0.83769, std: 0.01060, params: {'min_samples_split': 1000, 'max_depth': 7},
mean: 0.83631, std: 0.00942, params: {'min_samples_split': 200, 'max_depth': 9},
mean: 0.83695, std: 0.00923, params: {'min_samples_split': 400, 'max_depth': 9},
mean: 0.83339, std: 0.00893, params: {'min_samples_split': 600, 'max_depth': 9},
mean: 0.83793, std: 0.00965, params: {'min_samples_split': 800, 'max_depth': 9},
mean: 0.83844, std: 0.00954, params: {'min_samples_split': 1000, 'max_depth': 9},
mean: 0.83036, std: 0.00998, params: {'min_samples_split': 200, 'max_depth': 11},
mean: 0.83077, std: 0.00809, params: {'min_samples_split': 400, 'max_depth': 11},
mean: 0.83366, std: 0.00983, params: {'min_samples_split': 600, 'max_depth': 11},
mean: 0.83193, std: 0.00911, params: {'min_samples_split': 800, 'max_depth': 11},
mean: 0.83582, std: 0.01040, params: {'min_samples_split': 1000, 'max_depth': 11},
mean: 0.82198, std: 0.01037, params: {'min_samples_split': 200, 'max_depth': 13},
mean: 0.83055, std: 0.00837, params: {'min_samples_split': 400, 'max_depth': 13},
mean: 0.83139, std: 0.01127, params: {'min_samples_split': 600, 'max_depth': 13},
mean: 0.83403, std: 0.01060, params: {'min_samples_split': 800, 'max_depth': 13},
mean: 0.83288, std: 0.00974, params: {'min_samples_split': 1000, 'max_depth': 13},
mean: 0.82009, std: 0.00691, params: {'min_samples_split': 200, 'max_depth': 15},
mean: 0.82317, std: 0.01017, params: {'min_samples_split': 400, 'max_depth': 15},
mean: 0.82909, std: 0.00904, params: {'min_samples_split': 600, 'max_depth': 15},
mean: 0.82926, std: 0.00944, params: {'min_samples_split': 800, 'max_depth': 15},
mean: 0.83236, std: 0.01421, params: {'min_samples_split': 1000, 'max_depth': 15}},
{'max_depth': 9, 'min_samples_split': 1000},
0.83843938077464664)
```

Here, we have run 30 combinations and the ideal values are 9 for max_depth and 1000 for min_samples_split. Note that, 1000 is an extreme value which we tested. There is a fair chance that the optimum value lies above that. So we should check for some higher values as well.

Here, I'll take the max_depth of 9 as optimum and not try different values for higher min_samples_split. It might not be the best idea always but here if you observe the output closely, max_depth of 9 works better in most of the cases. Also, we can test for 5 values of min_samples_leaf, from 30 to 70 in steps of 10, along with higher min_samples_split.

```
param_test3 = {'min_samples_split':range(1000,2100,200), 'min_
samples_leaf':range(30,71,10)}

gsearch3 = GridSearchCV(estimator = GradientBoostingClassifier
(learning_rate=0.1, n_estimators=60,max_depth=9,max_features='
sqrt', subsample=0.8, random_state=10),
param_grid = param_test3, scoring='roc_auc',n_jobs=4,iid=False
, cv=5)

gsearch3.fit(train[predictors],train[target])

gsearch3.grid_scores_, gsearch3.best_params_, gsearch3.best_sc
ore_
```

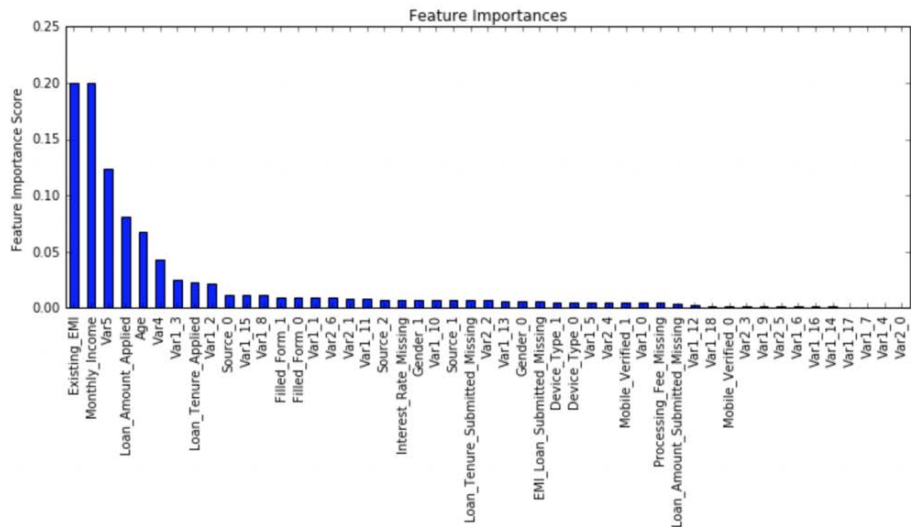


```
[{mean: 0.83821, std: 0.01092, params: {'min_samples_split': 1000, 'min_samples_leaf': 30},
mean: 0.83889, std: 0.01271, params: {'min_samples_split': 1200, 'min_samples_leaf': 30},
mean: 0.83552, std: 0.01024, params: {'min_samples_split': 1400, 'min_samples_leaf': 30},
mean: 0.83683, std: 0.01429, params: {'min_samples_split': 1600, 'min_samples_leaf': 30},
mean: 0.83958, std: 0.01233, params: {'min_samples_split': 1800, 'min_samples_leaf': 30},
mean: 0.83783, std: 0.01137, params: {'min_samples_split': 2000, 'min_samples_leaf': 30},
mean: 0.83821, std: 0.00872, params: {'min_samples_split': 1000, 'min_samples_leaf': 40},
mean: 0.83740, std: 0.01280, params: {'min_samples_split': 1200, 'min_samples_leaf': 40},
mean: 0.83714, std: 0.01019, params: {'min_samples_split': 1400, 'min_samples_leaf': 40},
mean: 0.83771, std: 0.01188, params: {'min_samples_split': 1600, 'min_samples_leaf': 40},
mean: 0.83738, std: 0.01370, params: {'min_samples_split': 1800, 'min_samples_leaf': 40},
mean: 0.83765, std: 0.01221, params: {'min_samples_split': 2000, 'min_samples_leaf': 40},
mean: 0.83575, std: 0.01017, params: {'min_samples_split': 1000, 'min_samples_leaf': 50},
mean: 0.83744, std: 0.01224, params: {'min_samples_split': 1200, 'min_samples_leaf': 50},
mean: 0.83892, std: 0.01234, params: {'min_samples_split': 1400, 'min_samples_leaf': 50},
mean: 0.83814, std: 0.01354, params: {'min_samples_split': 1600, 'min_samples_leaf': 50},
mean: 0.83824, std: 0.01116, params: {'min_samples_split': 1800, 'min_samples_leaf': 50},
mean: 0.83821, std: 0.01014, params: {'min_samples_split': 2000, 'min_samples_leaf': 50},
mean: 0.83626, std: 0.01111, params: {'min_samples_split': 1000, 'min_samples_leaf': 60},
mean: 0.83959, std: 0.00989, params: {'min_samples_split': 1200, 'min_samples_leaf': 60},
mean: 0.83735, std: 0.01217, params: {'min_samples_split': 1400, 'min_samples_leaf': 60},
mean: 0.83685, std: 0.01325, params: {'min_samples_split': 1600, 'min_samples_leaf': 60},
mean: 0.83589, std: 0.01101, params: {'min_samples_split': 1800, 'min_samples_leaf': 60},
mean: 0.83769, std: 0.01173, params: {'min_samples_split': 2000, 'min_samples_leaf': 60},
mean: 0.83792, std: 0.00994, params: {'min_samples_split': 1000, 'min_samples_leaf': 70},
mean: 0.83712, std: 0.01053, params: {'min_samples_split': 1200, 'min_samples_leaf': 70},
mean: 0.83777, std: 0.01186, params: {'min_samples_split': 1400, 'min_samples_leaf': 70},
mean: 0.83812, std: 0.01126, params: {'min_samples_split': 1600, 'min_samples_leaf': 70},
mean: 0.83812, std: 0.01055, params: {'min_samples_split': 1800, 'min_samples_leaf': 70},
mean: 0.83677, std: 0.01190, params: {'min_samples_split': 2000, 'min_samples_leaf': 70}},
{'min_samples_leaf': 60, 'min_samples_split': 1200},
0.83959087132827259)
```

Here we get the optimum values as 1200 for min_samples_split and 60 for min_samples_leaf. Also, we can see the CV score increasing to 0.8396 now. Let’s fit the model again on this and have a look at the feature importance.

```
modelfit(gsearch3.best_estimator_, train, predictors)
```

Model Report
Accuracy : 0.9854
AUC Score (Train): 0.896453
CV Score : Mean - 0.8395909 | Std - 0.009890497 | Min - 0.8259075 | Max - 0.8527672



If you compare the feature importance of this model with the

baseline model, you'll find that now we are able to derive value from many more variables. Also, earlier it placed too much importance on some variables but now it has been fairly distributed.

Now let's tune the last tree-parameters, i.e. `max_features` by trying 7 values from 7 to 19 in steps of 2.

```
param_test4 = {'max_features':range(7,20,2)}
gsearch4 = GridSearchCV(estimator = GradientBoostingClassifier
(learning_rate=0.1, n_estimators=60,max_depth=9, min_samples_s
plit=1200, min_samples_leaf=60, subsample=0.8, random_state=10
),
param_grid = param_test4, scoring='roc_auc',n_jobs=4,iid=False
, cv=5)
gsearch4.fit(train[predictors],train[target])
gsearch4.grid_scores_, gsearch4.best_params_, gsearch4.best_sc
ore_
```

```
([mean: 0.83959, std: 0.00989, params: {'max_features': 7},
mean: 0.83648, std: 0.00988, params: {'max_features': 9},
mean: 0.83919, std: 0.01042, params: {'max_features': 11},
mean: 0.83738, std: 0.01017, params: {'max_features': 13},
mean: 0.83820, std: 0.01017, params: {'max_features': 15},
mean: 0.83495, std: 0.00957, params: {'max_features': 17},
mean: 0.83499, std: 0.00996, params: {'max_features': 19}],
{'max_features': 7},
0.83959087132827259)
```

Here, we find that optimum value is 7, which is also the square root. So our initial value was the best. You might be anxious to check for lower values and you should if you like. I'll stay with 7 for now. With this we have the final tree-parameters as:

- `min_samples_split`: 1200
- `min_samples_leaf`: 60
- `max_depth`: 9
- `max_features`: 7

Tuning subsample and making models with

lower learning rate

The next step would be try different subsample values. Lets take values 0.6,0.7,0.75,0.8,0.85,0.9.

```
param_test5 = {'subsample':[0.6,0.7,0.75,0.8,0.85,0.9]}
gsearch5 = GridSearchCV(estimator = GradientBoostingClassifier
(learning_rate=0.1, n_estimators=60,max_depth=9,min_samples_sp
lit=1200, min_samples_leaf=60, subsample=0.8, random_state=10,
max_features=7),
param_grid = param_test5, scoring='roc_auc',n_jobs=4,iid=False
, cv=5)
gsearch5.fit(train[predictors],train[target])
gsearch5.grid_scores_, gsearch5.best_params_, gsearch5.best_sc
ore_
```

```
([mean: 0.83621, std: 0.00950, params: {'subsample': 0.6},
mean: 0.83648, std: 0.01181, params: {'subsample': 0.7},
mean: 0.83601, std: 0.01074, params: {'subsample': 0.75},
mean: 0.83959, std: 0.00989, params: {'subsample': 0.8},
mean: 0.83989, std: 0.01078, params: {'subsample': 0.85},
mean: 0.83827, std: 0.01076, params: {'subsample': 0.9}],
{'subsample': 0.85},
0.83988852960292915)
```

Here, we found 0.85 as the optimum value. Finally, we have all the parameters needed. Now, we need to lower the learning rate and increase the number of estimators proportionally. Note that these trees might not be the most optimum values but a good benchmark.

As trees increase, it will become increasingly computationally expensive to perform CV and find the optimum values. For you to get some idea of the model performance, I have included the private leaderboard scores for each. Since the data is not open, you won't be able to replicate that but it'll good for understanding.

Lets decrease the learning rate to **half, i.e. 0.05** with twice (120) the number of trees.

```
predictors = [x for x in train.columns if x not in [target, ID
col]]

gbm_tuned_1 = GradientBoostingClassifier(learning_rate=0.05, n
_estimators=120,max_depth=9, min_samples_split=1200,min_sample
s_leaf=60, subsample=0.85, random_state=10, max_features=7)

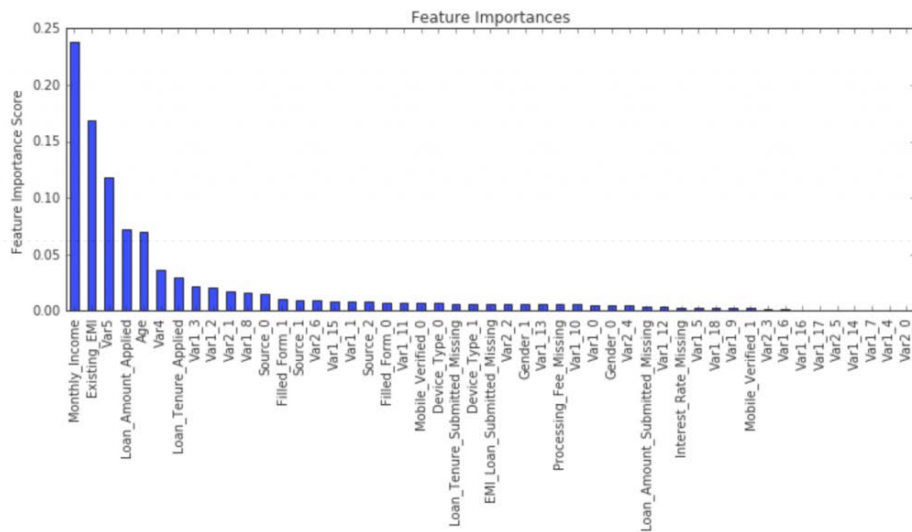
modelfit(gbm_tuned_1, train, predictors)
```

Model Report

Accuracy : 0.9854

AUC Score (Train): 0.897471

CV Score : Mean - 0.8396 | Std - 0.009514 | Min - 0.8266 | Max - 0.8516



Private LB Score: 0.844139

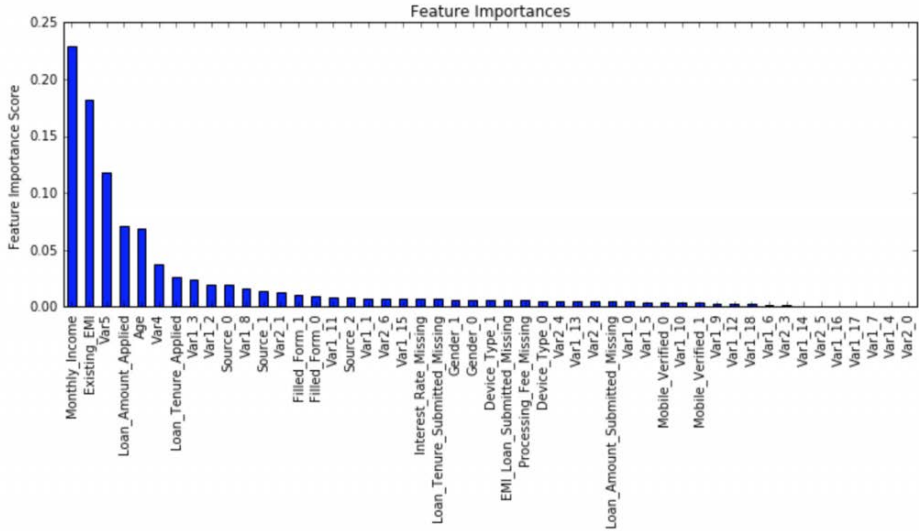
Now lets reduce to **one-tenth** of the original value, i.e. 0.01 for 600 trees.

```
predictors = [x for x in train.columns if x not in [target, ID
col]]

gbm_tuned_2 = GradientBoostingClassifier(learning_rate=0.01, n
_estimators=600,max_depth=9, min_samples_split=1200,min_sample
s_leaf=60, subsample=0.85, random_state=10, max_features=7)

modelfit(gbm_tuned_2, train, predictors)
```

Model Report
Accuracy : 0.9854
AUC Score (Train): 0.899927
CV Score : Mean - 0.8409339 | Std - 0.01035658 | Min - 0.8258238 | Max - 0.8529458



Private LB Score: 0.848145

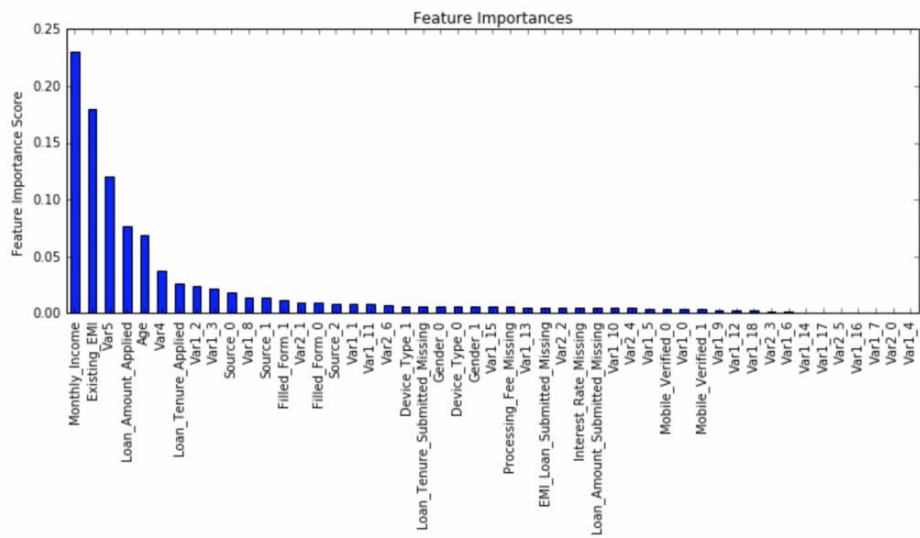
Lets decrease to **one-twentieth** of the original value, i.e. 0.005 for 1200 trees.

```
predictors = [x for x in train.columns if x not in [target, ID
col]]

gbm_tuned_3 = GradientBoostingClassifier(learning_rate=0.005,
n_estimators=1200,max_depth=9, min_samples_split=1200, min_sam
ples_leaf=60, subsample=0.85, random_state=10, max_features=7,
warm_start=True)

modelfit(gbm_tuned_3, train, predictors, performCV=False)
```

Model Report
Accuracy : 0.9854
AUC Score (Train): 0.900688



Private LB Score: 0.848112

Here we see that the score reduced very slightly. So lets run for 1500 trees.

```
predictors = [x for x in train.columns if x not in [target, ID
col]]

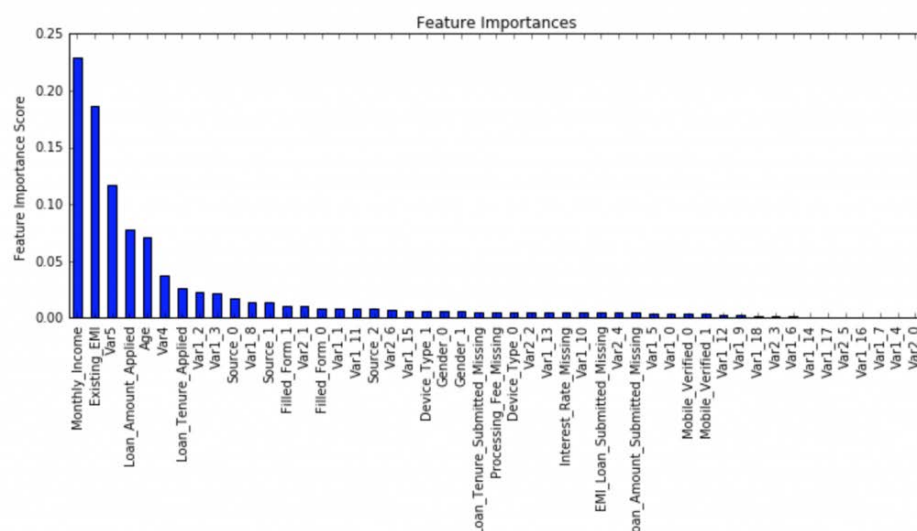
gbm_tuned_4 = GradientBoostingClassifier(learning_rate=0.005,
n_estimators=1500,max_depth=9, min_samples_split=1200, min_sam
ples_leaf=60, subsample=0.85, random_state=10, max_features=7,
warm_start=True)

modelfit(gbm_tuned_4, train, predictors, performCV=False)
```

Model Report

Accuracy : 0.9854

AUC Score (Train): 0.906346



Private LB Score: 0.848747

Therefore, now you can clearly see that this is a very important step as private LB scored improved from ~0.844 to ~0.849 which is a significant jump.

Another hack that can be used here is the 'warm_start' parameter of GBM. You can use it to increase the number of estimators in small steps and test different values without having to run from starting always. You can also download the iPython notebook with all these model codes from my [GitHub account](#).

If you like this article and want to read a similar post for XGBoost, check this out – [Complete Guide to Parameter Tuning in XGBoost](#)

End Notes

This article was based on developing a GBM model end-to-end. We started with an **introduction to boosting** which was followed by detailed discussion on the **various parameters** involved. The parameters were divided into 3 categories namely the tree-specific, boosting and miscellaneous parameters depending on

their impact on the model.

Finally, we discussed the **general approach** towards tackling a problem with GBM and also worked out the **AV Data Hackathon 3.x problem** through that approach.

I hope you found this useful and now you feel more confident to apply GBM in solving a data science problem. You can try this out in our upcoming signature hackathon [Date Your Data](#).

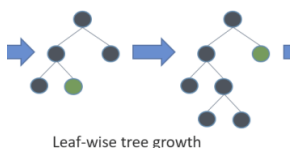
Did you like this article? Would you like to share some other hacks which you implement while making GBM models? Please feel free to drop a note in the comments below and I'll be glad to discuss.

You want to apply your analytical skills and test your potential? Then [participate in our Hackathons](#) and compete with Top Data Scientists from all over the world.

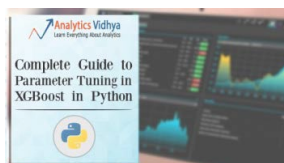
Share this:



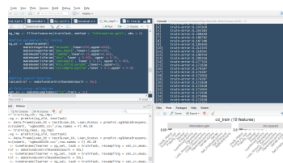
RELATED



Which algorithm takes the crown: Light GBM vs XGBOOST?



Complete Guide to Parameter Tuning in XGBoost (with codes in Python)



Practicing Machine Learning Techniques in R with MLR Package

TAGS:

BML Munjal
University launches
MBA in Business
Analytics to create
future leaders!

India Exclusive: Analytics and Big Data Salary Report 2016



Aarshay Jain

Aarshay is a ML enthusiast, pursuing MS in Data Science at Columbia University, graduating in Dec 2017. He is currently exploring the various ML techniques and writes articles for AV to share his knowledge with the community.

This article is quite old now and you might not get a prompt response from the author. We would request you to post this comment on Analytics Vidhya **Discussion portal** to get your queries resolved.

22 COMMENTS



anurag says:

REPLY

FEBRUARY 22, 2016 AT 7:33 AM

Great Article!!

Can you do this for SVM,XGBoost, deep learning and neural networks.



Aarshay Jain says:

REPLY

FEBRUARY 22, 2016 AT 8:10 AM

Absolutely!! I plan to do an entire series if people like the first ones. XGBoost is definitely coming up next week.. There are a few others along with SVM and Deep Learning in my mind. But I haven't thought about a specific order yet.. Feel free to push in more suggestions.. It'll be easier for me to decide ☐

Also, it'll be great if you can share this article with others in your network. The more people read these, higher will be the preference we can set on articles of this kind!!



Mohammed Niyas says:

REPLY

FEBRUARY 23, 2016 AT 11:39 AM

Waiting for XGBoost, in python if possible.



Aarshay Jain says: REPLY

FEBRUARY 23, 2016 AT 11:40 AM

Yes it'll be coming up in Python!



swathi says:

REPLY

FEBRUARY 23,
2016 AT 6:09 PM

Thank you for
this amazing
article. XGBoost
in Python please!



Aarsh

Jain

says:

FEBRUARY
23,
2016 AT
6:10 PM

for
sure



Jignesh Vyas says:

REPLY

FEBRUARY 23, 2016 AT 1:24 AM

Wow great article, pretty much detailed and easy to
understand. Am a great fan of articles posted on this site.
Keep up the good work !



Aarshay Jain says:

REPLY

FEBRUARY 23, 2016 AT 8:13 AM

Thanks Jignesh 



Pallavi says:

REPLY

FEBRUARY 23, 2016 AT 10:04 AM

absolutely fantastic article. Loved the step by step
approach. Would love to read more of these on SVM, deep
learning. Also, it would be fantastic to have R code.



Aarshay Jain says:

REPLY

FEBRUARY 23, 2016 AT 10:44 AM

glad you liked it.. stay tuned for more!!



Aarshay Jain says:

REPLY

MARCH 2, 2016 AT 7:19 AM

Guys,

The article for XGBoost is out –

<http://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/>

I've also updated the same towards the end of this article.

Enjoy and share your feedback!

Cheers,
Aarshay



Don Fox says:

REPLY

MARCH 7, 2016 AT 12:00 AM

Great example! One question;
Where do you define 'dtrain'? I see it as an argument for the function.



Aarshay Jain says:

REPLY

MARCH 7, 2016 AT 6:11 AM

Hi Don,

Thanks for reaching out. So dtrain is a function argument and copies the passed value into dtrain. To explain further, a function is defined using following:

```
def modelfit(alg, dtrain, predictors,
performCV=True, printFeatureImportance=True,
cv_folds=5):
```

This tells that modelfit is a function which takes

arguments as `alg`, `dtrain`, `predictors` and so on..

I need not define these arguments explicitly.

When I call a function using:

```
modelfit(gbm_tuned_4, train, predictors,  
performCV=False)
```

The passed values get mapped to arguments. In this case:

```
alg = gbm_tuned_4
```

```
dtrain = train
```

```
predictors = predictors, and so on..
```

One thing to note is that for some arguments I have defined a default value, like “`performCV=True`, `printFeatureImportance=True`, `cv_folds=5`”. So you need not pass these always in a function. If not passed, the default values are taken.

Cheers,
Aarshay



Praveen Gupta Sanka says:

REPLY

MARCH 15, 2016 AT 3:39 AM

Hi Aarshay,

The article is simply superb.

I have the following doubts.. Please clarify :

1. what is the difference between `pylab` and `pyplot` module in `matplotlib` library?

(I searched in google and found certain results that `pylab` should not be used any further) can you please give some additional details what is the difference and which is preferred.

2. `modelfit(gbm0,train,predictors,printOOB=False)` is written by you at a specific place. `modelfit` does not accept ‘`printOOB`’ parameter. Can you please explain what is the use of this parameter?



Aarshay Jain says:

REPLY

MARCH 15, 2016 AT 3:58 AM

Thanks Praveen! Regarding your concerns.

1. I didn't thought about this earlier but a quick search tells me that pyplot is the recommended option as you rightly mentioned. You can get some more intuition from this comment which I found online:

"The pyplot interface is generally preferred for non-interactive plotting (i.e., scripting). The pylab interface is convenient for interactive calculations and plotting, as it minimizes typing. Note that this is what you get if you use the ipython shell with the -pylab option, which imports everything from pylab and makes plotting fully interactive."

2. I was using this earlier but I removed it later. I've updated the code with this removed. It was being used earlier to plot the OOB (out-of-bag) improvement curve. GBM returns OOB improvement scores which can be plotted to check how well the model is fitting. But later I found that the estimates provided are biased and many people recommend it should not be used. So I removed that plot from my codes.



Jay says:

REPLY

MARCH 31, 2016 AT 10:21 AM

That was amazing content, thanks for your efforts.

What's your Kaggle account, if you don't mind sharing?



Aarshay Jain says:

REPLY

MARCH 31, 2016 AT 10:23 AM

Thanks Jay.. Here is my kaggle account –
<https://www.kaggle.com/aarshayjain>
Even I am pretty new at Kaggle. This was my first serious participation..



Xu Zhang says:

REPLY

SEPTEMBER 27, 2016 AT 11:07 PM

Great article. Thank you so much for sharing. May I ask you some questions?

1.. You tuned the parameters with an order, first `n_estimate` and `learning_rate`, then keep these parameters as the constants when you tuned other tree related parameters. Does the order really matter? I think it matters. But how did you figure out this order and what kinds of theories did you based on?

2. In sklearn, there is a tool called `GridSearchCV`. If we don't consider the cost of computing, we put every parameter together as a `dict{ }`, can we get a final, optimized set of parameters? I doubt it. What is your opinion?

3. Is your tuning method suitable for other datasets and other problems?

Your response is highly appreciated. Thank you so much for your great work.



Serhiy says:

REPLY

OCTOBER 16, 2016 AT 8:51 PM

Amazing article!

There is one small issue – in section Tuning tree-specific parameters the first stage called “Tune `max_depth` and `num_samples_split`”, I guess it should be “`min_samples_split`”



Serhiy says:

REPLY

OCTOBER 16, 2016 AT 9:03 PM

And one more thing:

“Here, I'll take the `max_depth` of 9 as optimum and not try different values for higher `min_samples_split`” should be “Here, I'll take the `max_depth` of 9 as optimum and NOW try different values for higher `min_samples_split`”



Suleyman Sahal says:

REPLY

OCTOBER 18, 2016 AT 9:28 PM

Great article. Thanks for sharing. One question: How do you adjust steps for parameter tuning? For instance you set 10 as step size for n_estimators parameter and found out that 60 has the highest CV accuracy. What if 55 provides significantly(!) better prediction accuracy and we missed it. Which approach should we follow to adjust step size? For example, should we try every number for integer parameter if computationally possible (10-20 hours maybe)? Thanks.



Andy says:

REPLY

NOVEMBER 8, 2016 AT 11:05 PM

Hi,

Please can you rename your variables in terms of X and y?
Let X = training data(matrix) and y = target(a vector). I assume dtrain[predictors] = X and dtrain['Disbursed'] = y.

In this section:

#Print Feature Importance:

if printFeatureImportance:

```
feat_imp = pd.Series(alg.feature_importances_,  
predictors).sort_values(ascending=False)
```

What is predictors?

I know these questions sound basic but I'd really appreciate some help

LEAVE A REPLY

Your email address will not be published.



DATA SCIENTISTS



COMPANIES



Don't have an account? Sign up here. **JOIN OUR COMMUNITY :**

14774

© Copyright 2013-2018 Analytics Vidhya.

2608
5975