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

MACHINE LEARNING PYTHON

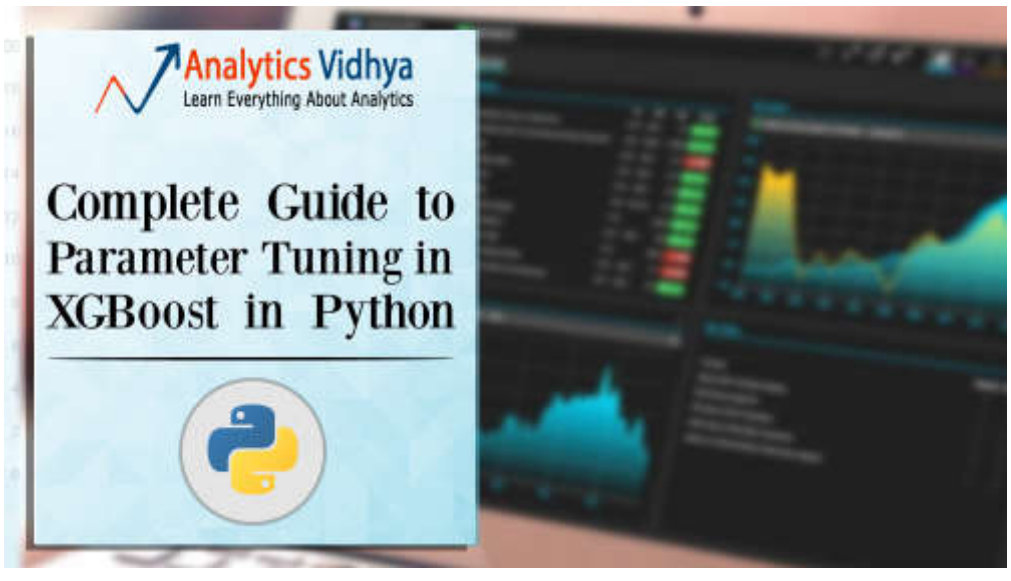
AARSHAY JAIN , MARCH 1, 2016 / 95

Introduction

If things don't go your way in predictive modeling, use XGboost. XGBoost algorithm has become the ultimate weapon of many data scientist a highly sophisticated algorithm, powerful enough to deal with all sorts of irregularities of data.

Building a model using XGBoost is easy. But, improving the model using XGBoost is difficult (at least I struggled a lot). This algorithm uses mul parameters. To improve the model, parameter tuning is must. It is very difficult to get answers to practical questions like – Which set of parameters should tune ? What is the ideal value of these parameters to obtain optimal output ?

This article is best suited to people who are new to XGBoost. In this article, we'll learn the art of parameter tuning along with some useful informa about XGBoost. Also, we'll practice this algorithm using a data set in Python.



What should you know ?

XGBoost (eXtreme Gradient Boosting) is an advanced implementation of gradient boosting algorithm. Since I covered Gradient Boosting Machir detail in my previous article – Complete Guide to Parameter Tuning in Gradient Boosting (GBM) in Python, I highly recommend going through before reading further. It will help you bolster your understanding of boosting in general and parameter tuning for GBM.

Special Thanks: Personally, I would like to acknowledge the timeless support provided by Mr. Sudalai Rajkumar (aka SRK), currently AV Rank 2. article wouldn't be possible without his help. He is helping us guide thousands of data scientists. A big thanks to SRK!

Table of Contents

1. The XGBoost Advantage

I've always admired the boosting capabilities that this algorithm infuses in a predictive model. When I explored more about its performance and scie behind its high accuracy, I discovered many advantages:

1. **Regularization:**
    - Standard GBM implementation has no regularization like XGBoost, therefore it also helps to reduce overfitting.
    - In fact, XGBoost is also known as '**regularized boosting**' technique.
2. **Parallel Processing:**
    - XGBoost implements parallel processing and is **blazingly faster** as compared to GBM.
    - But hang on, we know that boosting is sequential process so how can it be parallelized? We know that each tree can be built only after the prev 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.
3. **High Flexibility**
    - XGBoost allow 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.
4. **Handling Missing Values**
    - XGBoost has an in-built routine to handle missing values.
    - User is required to supply a different value than other observations and pass that as a parameter. XGBoost tries different things as it encounte missing value on each node and learns which path to take for missing values in future.
5. **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 on the other hand make **splits upto the max_depth** specified and then start **pruning** the tree backwards and remove splits beyond w there is no positive gain.
    - Another advantage is that sometimes a split of negative loss say -2 may be followed by a split of positive loss +10. GBM would stop as it encou -2. But XGBoost will go deeper and it will see a combined effect of +8 of the split and keep both.
6. **Built-in Cross-Validation**
    - XGBoost allows user to run a **cross-validation at each iteration** of the boosting process and thus it is easy to get the exact optimum numb boosting iterations in a single run.
    - This is unlike GBM where we have to run a grid-search and only a limited values can be tested.
7. **Continue on Existing Model**
    - User can start training an XGBoost model from its last iteration of previous run. This can be of significant advantage in certain specific applicatior
    - 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 which I could muster. You know a few more? Feel to drop a comment below and I will update the list.

Did I whet your appetite ? Good. You can refer to following web-pages for a deeper understanding:

- XGBoost Guide – Introduction to Boosted Trees
- Words from the Author of XGBoost [Video]

2. 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

I will give analogies to GBM here and highly recommend to read this article to learn from the very basics.

General Parameters

These define the overall functionality of XGBoost.

1. **booster [default=gbtree]**
    - Select the type of model to run at each iteration. It has 2 options:
        - 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 maximum number of threads available if not set]**
    - This is used for parallel processing and number of cores in the system should be entered
    - If you wish to run on all cores, value should not be entered and algorithm will detect automatically

There are 2 more parameters which are set automatically by XGBoost and you need not worry about them. Lets 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 later is ra used.

1. **eta [default=0.3]**
    - Analogous to learning rate in GBM
    - Makes the model more robust by shrinking the weights on each step
    - Typical final values to be used: 0.01-0.2
2. **min_child_weight [default=1]**
    - Defines the minimum sum of weights of all observations required in a child.
    - This is similar to **min_child_leaf** in GBM but not exactly. This refers to min "sum of weights" of observations while GBM has min "numb observations".
    - Used to control over-fitting. Higher values prevent a model from learning relations which might be highly specific to the particular sample selecte a tree.
    - Too high values can lead to under-fitting hence, it should be tuned using CV.
3. **max_depth [default=6]**
    - The maximum depth of a tree, same as GBM.
    - 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.
    - Typical values: 3-10
4. **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.
5. **gamma [default=0]**
    - A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction requir make a split.
    - Makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.
6. **max_delta_step [default=0]**
    - In maximum delta step we allow each tree's weight estimation to be. If the value is set to 0, it means there is no constraint. If it is set to a pos value, it can help making the update step more conservative.
    - Usually this parameter is not needed, but it might help in logistic regression when class is extremely imbalanced.
    - This is generally not used but you can explore further if you wish.
7. **subsample [default=1]**
    - Same as the subsample of GBM. Denotes the fraction of observations to be randomly samples for each tree.
    - Lower values make the algorithm more conservative and prevents overfitting but too small values might lead to under-fitting.
    - Typical values: 0.5-1
8. **colsample_bytree [default=1]**
    - Similar to max_features in GBM. Denotes the fraction of columns to be randomly samples for each tree.
    - Typical values: 0.5-1
9. **colsample_bylevel [default=1]**
    - Denotes the subsample ratio of columns for each split, in each level.
    - I don't use this often because subsample and colsample_bytree will do the job for you. but you can explore further if you feel so.
10. **lambda [default=1]**
    - L2 regularization term on weights (analogous to Ridge regression)
    - This used to handle the regularization part of XGBoost. Though many data scientists don't use it often, it should be explored to reduce overfitting
11. **alpha [default=0]**
    - L1 regularization term on weight (analogous to Lasso regression)
    - Can be used in case of very high dimensionality so that the algorithm runs faster when implemented
12. **scale_pos_weight [default=1]**
    - A value greater than 0 should be used in case of high class imbalance as it helps in faster convergence.

Learning Task Parameters

These parameters are used to define the optimization objective 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 metric 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
        - **logloss** – negative log-likelihood
        - **error** – Binary classification error rate (0.5 threshold)
        - **merror** – Multiclass classification error rate
        - **mlogloss** – Multiclass logloss
        - **auc:** Area under the curve
3. **seed [default=0]**
    - The random number seed.
    - 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. A good news is that xgboost module in python has an skl wrapper called XGBClassifier. It uses sklearn style naming convention. The parameters names which will change are:

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

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

I recommend you to go through the following parts of 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)

3. Parameter Tuning with Example

We will take the data set from Data Hackathon 3.x AV hackathon, same as that taken in the GBM article. The details of the problem can be foun 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 dropp
10. Loan_Tenure_Submitted_Missing created which is 1 if Loan_Tenure_Submitted was missing else 0 | Original variable Loan_Tenure_Submitted droppe
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
import xgboost as xgb
from xgboost.sklearn import XGBClassifier
from sklearn import cross_validation, metrics   #Additional scklearn functions
from sklearn.grid_search import GridSearchCV   #Perforing grid search

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

train = pd.read_csv('train_modified.csv')
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 f GBM

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

```
def modelfit(alg, dtrain, predictors,useTrainCV=True, cv_folds=5, early_stopping_rounds=50):

    if useTrainCV:
```

```
    xgb_param = alg.get_xgb_params()
    xgtrain = xgb.DMatrix(dtrain[predictors].values, label=dtrain[target].values)
    cvresult = xgb.cv(xgb_param, xgtrain, num_boost_round=alg.get_params()['n_estimators'], nfold=cv_folds,
        metrics='auc', early_stopping_rounds=early_stopping_rounds, show_progress=False)
    alg.set_params(n_estimators=cvresult.shape[0])

#Fit the algorithm on the data
alg.fit(dtrain[predictors], dtrain['Disbursed'],eval_metric='auc')

#Predict training set:
dtrain_predictions = alg.predict(dtrain[predictors])
dtrain_predprob = alg.predict_proba(dtrain[predictors])[:,1]

#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)

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

This code is slightly different from what I used for GBM. The focus of this article is to cover the concepts and not coding. Please feel free to drop a in the comments if you find any challenges in understanding any part of it. Note that xgboost's sklearn wrapper doesn't have a "feature_importan metric but a get_fscore() function which does the same job.

General Approach for Parameter 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 diff problems. Determine the **optimum number of trees for this learning rate**. XGBoost has a very useful function called as "cv" which performs c 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 decided learning rate and number of t 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 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. **max_depth = 5** : This should be between 3-10. I've started with 5 but you can choose a different number as well. 4-6 can be good starting points.
2. **min_child_weight = 1** : A smaller value is chosen because it is a highly imbalanced class problem and 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 used start value. Typical values range between 0.5-0.9.
5. **scale_pos_weight = 1**: Because of high class imbalance.

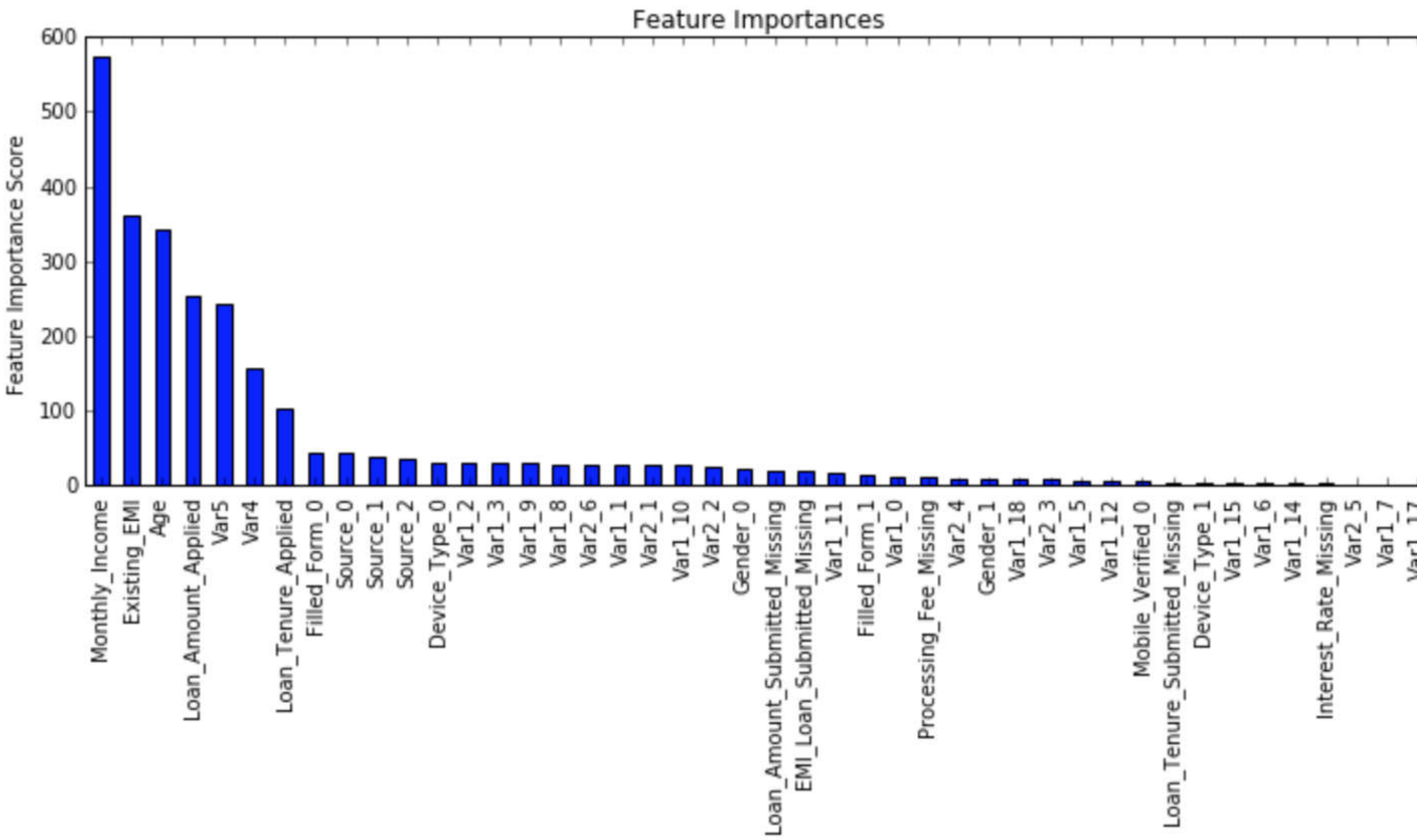
Please note that all the above are just initial estimates and will be tuned later. Lets take the default learning rate of 0.1 here and check the optir number of trees using cv function of xgboost. The function defined above will do it for us.

```
#Choose all predictors except target & IDcols
predictors = [x for x in train.columns if x not in [target, IDcol]]
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 estimators for 0.1 learning rate. Note that this value might be too high for you depending or power of your system. In that case you can increase the learning rate and re-run the command to get the reduced number of estimators.

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 y 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 b removed here.

Step 2: Tune max_depth and min_child_weight

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

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 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)
}
```
control tree complexity
```
gsearch1 = GridSearchCV(estimator = XGBClassifier( learning_rate =0.1, n_estimators=140, 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),
 param_grid = param_test1, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch1.fit(train[predictors],train[target])
gsearch1.grid_scores_, gsearch1.best_params_, gsearch1.best_score_
```

```
([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 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 = {
 'max_depth':[4,5,6],
 'min_child_weight':[4,5,6]
}
```
fine tuning
```
gsearch2 = GridSearchCV(estimator = XGBClassifier( learning_rate=0.1, n_estimators=140, max_depth=5,
 min_child_weight=2, gamma=0, subsample=0.8, colsample_bytree=0.8,
 objective= 'binary:logistic', nthread=4, scale_pos_weight=1,seed=27),
 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_score_
```

```
([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},
 0.8412491517996457 7)
```

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 tha the model performance increases, it becomes exponentially difficult to achieve even marginal gains in performance. You would have noticed that we got 6 as 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( learning_rate=0.1, n_estimators=140, max_depth=4,
 min_child_weight=2, gamma=0, subsample=0.8, colsample_bytree=0.8,
 objective= 'binary:logistic', nthread=4, scale_pos_weight=1,seed=27),
 param_grid = param_test2b, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch2b.fit(train[predictors],train[target])
```

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

```
([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},
 0.8412491517996457 7)
```

We see 6 as the optimal value.

Step 3: Tune gamma

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

```
param_test3 = {
 'gamma':[i/10.0 for i in range(0,5)]
}
gsearch3 = GridSearchCV(estimator = XGBClassifier( learning_rate =0.1, n_estimators=140, 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),
 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_score_
```

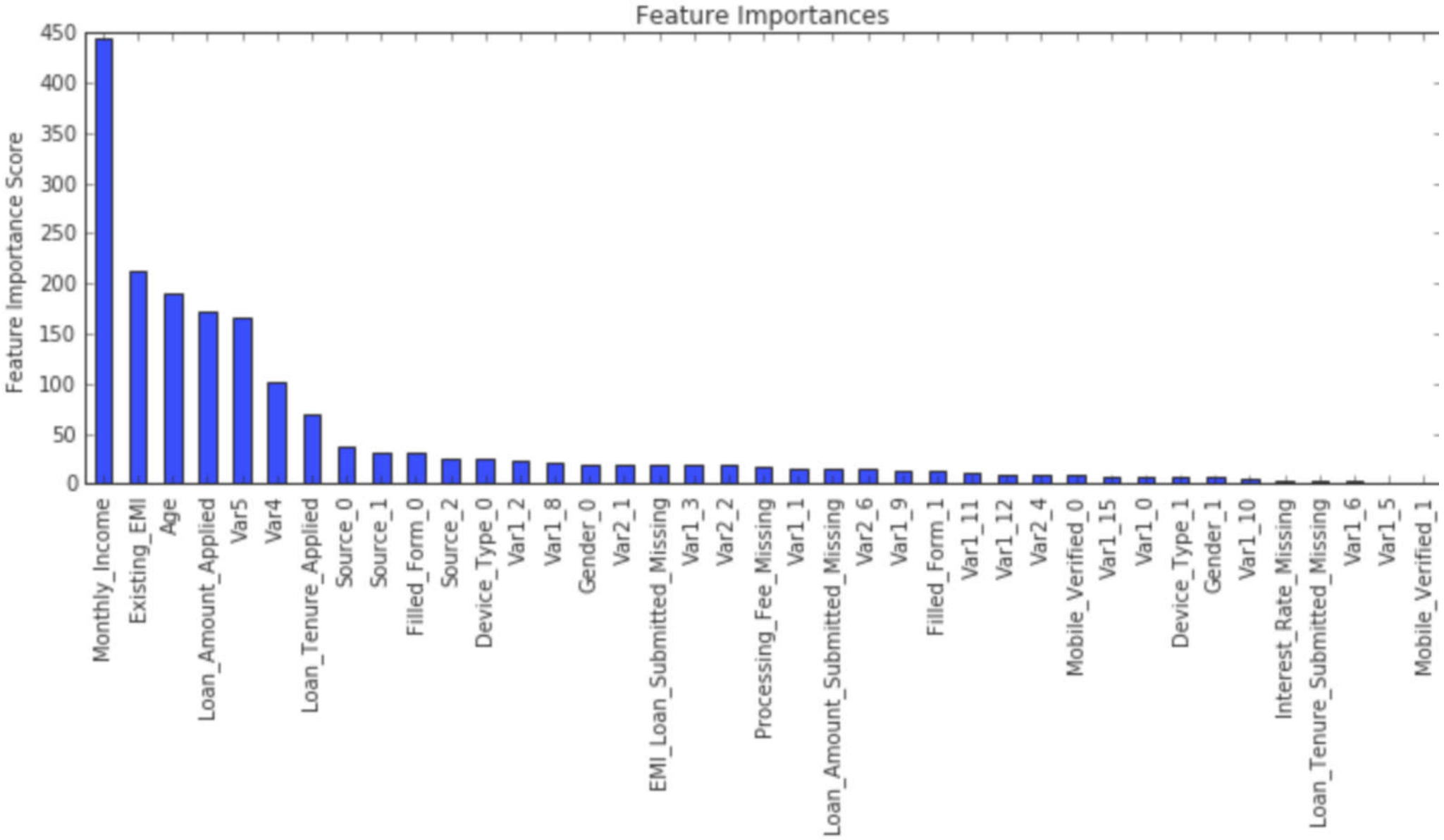
```
([mean: 0.84125, std: 0.00723, params: {'gamma': 0.0},
  mean: 0.83996, std: 0.00695, params: {'gamma': 0.1},
  mean: 0.84045, std: 0.00639, params: {'gamma': 0.2},
  mean: 0.84032, std: 0.00673, params: {'gamma': 0.3},
  mean: 0.84061, std: 0.00692, params: {'gamma': 0.4}],
 {'gamma': 0.0},
 0.8412491517996457)
```

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 improvement in score. So the final parameters are:

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

Step 4: Tune subsample and colsample_bytree

The next step would be try different subsample and colsample_bytree values. Lets do this in 2 stages as well and take values 0.6,0.7,0.8,0.9 for bo 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( learning_rate =0.1, n_estimators=177, 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),
 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_score_
```