XGBoost (eXtreme Gradient Boosting) (regularized boosting)

Литература

XGBoost Parameters (official guide)

http://xgboost.readthedocs.org/en/latest/parameter.html#general-parameters

XGBoost Demo Codes (xgboost GitHub repository)

https://github.com/dmlc/xgboost/tree/master/demo/guide-python

Python API Reference (official guide)

http://xgboost.readthedocs.org/en/latest/python/python_api.html

Постоить модель с помощью XGBoost легко.

Подобрать оптимальные значения параметров XGBoost очень сложно.

Какие параметры настраивать, какие значения оптимальны?

Однозначного ответа нет.

Есть даже «стакинг XGBoost’ов»

1. XGBoost лучше GBM

1. Есть регуляризация
2. Лучше реализована параллелизация, Hadoop
3. Эмпирические приемы работы с пропусками
4. Обрезание деревьев.

Сначала строится дерево с максимальным числом слоев. Допускается даже увеличение загрязнения при расщеплении. Затем отбрасываются неэффективные узлы.

Параметры XGBoost

Авторы и разработчики XGBoost’а делят параметры на 3 группы.

General Parameters

Booster Parameters

Learning Task Parameters

General Parameters

booster [default=gbtree]

Выбор слабой модели. Выбираем один из 2-х вариантов:

gbtree: деревья

gblinear: линейные модели (либо хуже, либо очень долго)

silent [default=0]:

Вывод промежуточных результатов в ходе обучения модели

1 : промежуточные результаты не выдаются

0 : промежуточные результаты выдаются.

nthread [default to maximum number of threads available if not set]

Число ядер, используемых при вычислениях.

… есть еще параметры ...

Booster Parameters (для варианта booster =gbtree)

num_boosting_rounds

Число деревьев

eta [default=0.3]

Скорость обучения. Обычно используют значения в интервале 0.01-0.2

min_child_weight [default=1]

Минимальное значение для суммы весов в узле потомке.

Аналогично минимально возможному числу наблюдений в узле потомке **min_child_leaf** в GBM, но есть отличие. суммы весов не то же самое, что число наблюдений. Предназначено для предотвращения переподгонки. Слишком большое значение ухудшит модель. Интуиция не работает, надо подбирать, используя кросс-валидацию.

max_depth [default=6]

Максимальное число слоев дерева.

Предотвращает переподгонку. Значение может быть разным в разных задачах. Надо подбирать, используя кросс-валидацию. Обычно используют значения в интервале 3-10.

max_leaf_nodes

Максимальное количество конечных узлов в дереве. Если используется, заменяет **max_depth**.

gamma [default=0]

Запрещает расщепление узла, если загрязнение потомков уменьшилось менее, чем на **gamma**. Значение параметра зависит от используемого критерия качества. Надо подбирать, используя кросс-валидацию.

max_delta_step [default=0]

Не знаю. Мало, кто знает. Используется значение по умолчанию.

subsample [default=1]

Доля наблюдений, попадающих в случайную подвыборку при построении очередного дерева. Маленькие значения препятствуют переподгонке, очень маленькие ухудшают качество модели. Обычно используют значения в интервале 0.5-1.

colsample_bytree [default=1]

Доля переменных, попадающих в случайную подвыборку при построении очередного дерева. Маленькие значения препятствуют переподгонке, очень маленькие ухудшают качество модели. Обычно используют значения в интервале 0.5-1.

colsample_bylevel [default=1]

Доля переменных, попадающих в случайную подвыборку при расщеплении очередного узла дерева. Маленькие значения препятствуют переподгонке, очень маленькие ухудшают качество модели. Забава для параноиков. Замедляет обучение.

lambda [default=1]

Коэффициент при L2 регуляризационном слагаемом (как в Ridge регрессии). Надо подбирать, используя кросс-валидацию.

alpha [default=0]

Коэффициент при L1 регуляризационном слагаемом (как в Lasso регрессии). Надо подбирать, используя кросс-валидацию. Рекомендуется использовать при большом числе переменных.

scale_pos_weight [default=1]

Не знаю. Но параметр важный. Используется при анализе несбалансированных выборок.

Learning Task Parameters

Описывают критерий качества, используемый при обучении.

objective [default=reg:linear]

Задается критерий качества, используемый при обучении. Чаще всего используются:

binary:logistic – когда имеется два класса, выходные значения — вероятности принадлежать классу, не код класса

multi:softmax – когда имеется больше двух классов, выходные значения — код класса, не вероятности принадлежать классу.

Надо еще дополнительно задать **num_class** — число классов в задаче.

multi:softprob – когда имеется больше двух классов, выходные значения — вероятности принадлежать классу, не код класса.

eval_metric [default according to objective]

Метрика, используемая при валидации. По умолчанию используется rmse в задаче регрессии и error в задаче классификации.

Популярные варианты:

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

seed [default=0]

Зерно датчика случайных чисел. Полезен для воспроизводимости результатов, в частности при подборе параметров.

Scikit-Learn содержит XGBClassifier обертку для модуля(?) xgboost

При этом некоторые параметры приобретают другое имя:

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

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 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
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 for 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 can 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 note 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_importances" 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 different problems. Determine the **optimum number of trees for this learning rate**. XGBoost has a very useful function called as "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 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.
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 optimum 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)
```

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 on the 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 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 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 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.

```
param_test1 = {
  'max_depth':range(3,10,2),
  'min_child_weight':range(1,6,2)
}
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_
```

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**. Lets 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 = {
  'max_depth':[4,5,6],
  'min_child_weight':[4,5,6]
}
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_
```

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 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_
```

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 can go 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_
```

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)
```

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 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( 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_
```

Here, we found **0.8 as the optimum value for both** subsample and colsample_bytree. Now we should try values in 0.05 interval 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( 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_test5, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch5.fit(train[predictors],train[target])
```

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

Next step is to apply regularization to reduce overfitting. Though many people don't use this parameters much as gamma provides a substantial way of controlling complexity. But we should always try it. I'll tune 'reg_alpha' value here and leave it upto you to try different values of 'reg_lambda'.

```
param_test6 = {
  'reg_alpha':[1e-5, 1e-2, 0.1, 1, 100]
}
gsearch6 = GridSearchCV(estimator = XGBClassifier( learning_rate =0.1, n_estimators=177, max_depth=4,
  min_child_weight=6, gamma=0.1, subsample=0.8, colsample_bytree=0.8,
  objective= 'binary:logistic', nthread=4, scale_pos_weight=1,seed=27),
  param_grid = param_test6, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch6.fit(train[predictors],train[target])
gsearch6.grid_scores_, gsearch6.best_params_, gsearch6.best_score_
```

We can see that the CV score is less than 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( learning_rate =0.1, n_estimators=177, max_depth=4,
```

```
  min_child_weight=6, gamma=0.1, subsample=0.8, colsample_bytree=0.8,
  objective= 'binary:logistic', nthread=4, scale_pos_weight=1,seed=27),
  param_grid = param_test7, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch7.fit(train[predictors],train[target])
gsearch7.grid_scores_, gsearch7.best_params_, gsearch7.best_score_
```

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)
```

Again we can see slight improvement in the score.

Step 6: Reducing Learning Rate

Lastly, we should lower the learning rate and add more trees. Lets 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)
```

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

As we come to the 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 **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.