AI学习笔记--sklearn--XGBoost算法

- XGBoost算法简介

XGBoost是**Extreme Gradient Boosting**的简称，Gradient Boosting是论文"Greedy Function Approximation: A Gradient Boosting Machine"中介绍的梯度提升算法。Boosting Tree树数据挖掘和机器学习中常用算法之一，其对输入要求不敏感，效果好，在工业界用的较多。

XGBoost是一个优化的分布式梯度增强库，旨在实现高效，灵活和便携。 它在Gradient Boosting框架下实现机器学习算法。 XGBoost提供了并行树提升（也称为GBDT，GBM），可以快速准确地解决许多数据学科问题。 相同的代码在主要的分布式环境（Hadoop，SGE，MPI）上运行，并且可以解决超过数十亿个样例的问题。XGBoost正是本教程中介绍的boosted tree原则所激发的工具！ 更重要的是，它在系统优化和机器学习原理方面都得到了深入的考虑。该库的目标是推动机器计算限制的极端，以提供可扩展，可移植和准确的库。XGBoost成功背后最重要的因素是它在所有场景中的可扩展性。 该系统在单台机器上运行速度比现有流行解决方案快十倍以上，并且在分布式或内存限制设置中可扩展至数十亿个示例。XGBoost的可扩展性归功于几个重要的系统和算法优化。 这些创新包括：

- 一种新颖的用于处理稀疏数据的树学习算法;
- 理论上合理的**weighted quantile sketch**过程使得能够在近似树学习中处理实例权重；
- 引入一个新颖的稀疏感知（**sparsity-aware**）算法用于并行树学习；并行和分布式计算使学习更快，从而实现更快的模型探索；
- 我们提出了一种有效地缓存感知块结构用于核外树学习；

关于XGBoost原理算法，请参考：

- **XGBoost 英文简介（原理）**

Introduction to Boosted Trees

XGBoost stands for “Extreme Gradient Boosting”, where the term “Gradient Boosting” originates from the paper *Greedy Function Approximation: A Gradient Boosting Machine*, by Friedman. This is a tutorial on gradient boosted trees, and most of the content is based on these slides by Tianqi Chen, the original author of XGBoost.
The **gradient boosted trees** has been around for a while, and there are a lot of materials on the topic. This tutorial will explain boosted trees in a self-contained and principled way using the elements of supervised learning. We think this explanation is cleaner, more formal, and motivates the model formulation used in XGBoost.

Elements of Supervised Learning

XGBoost is used for supervised learning problems, where we use the training data (with multiple features) x_i
to predict a target variable y_i. Before we learn about trees specifically, let us start by reviewing the basic elements in supervised learning.

Model and Parameters

The **model** in supervised learning usually refers to the mathematical structure of by which the prediction y_i is made from the input x_i. A common example is a *linear model*, where the prediction is given as $\hat{y}_i = \sum_j \theta_j x_{ij}$, a linear combination of weighted input features. The prediction value can have different interpretations, depending on the task, i.e., regression or classification. For example, it can be logistic transformed to get the probability of positive class in logistic regression, and it can also be used as a ranking score when we want to rank the outputs.
The **parameters** are the undetermined part that we need to learn from data. In linear regression problems, the parameters are the coefficients θ. Usually we will use θ to denote the parameters (there are many parameters in a model, our definition here is sloppy).

Objective Function: Training Loss + Regularization

With judicious choices for y_i, we may express a variety of tasks, such as regression, classification, and ranking. The task of **training** the model amounts to finding the best parameters θ that best fit the training data x_i and labels y_i. In order to train the model, we need to define the **objective function** to measure how well the model fit the training data.A salient characteristic of objective functions is that they consist two parts: **training loss** and **regularization term**:

$$\text{obj}(\theta) = L(\theta) + \Omega(\theta)$$

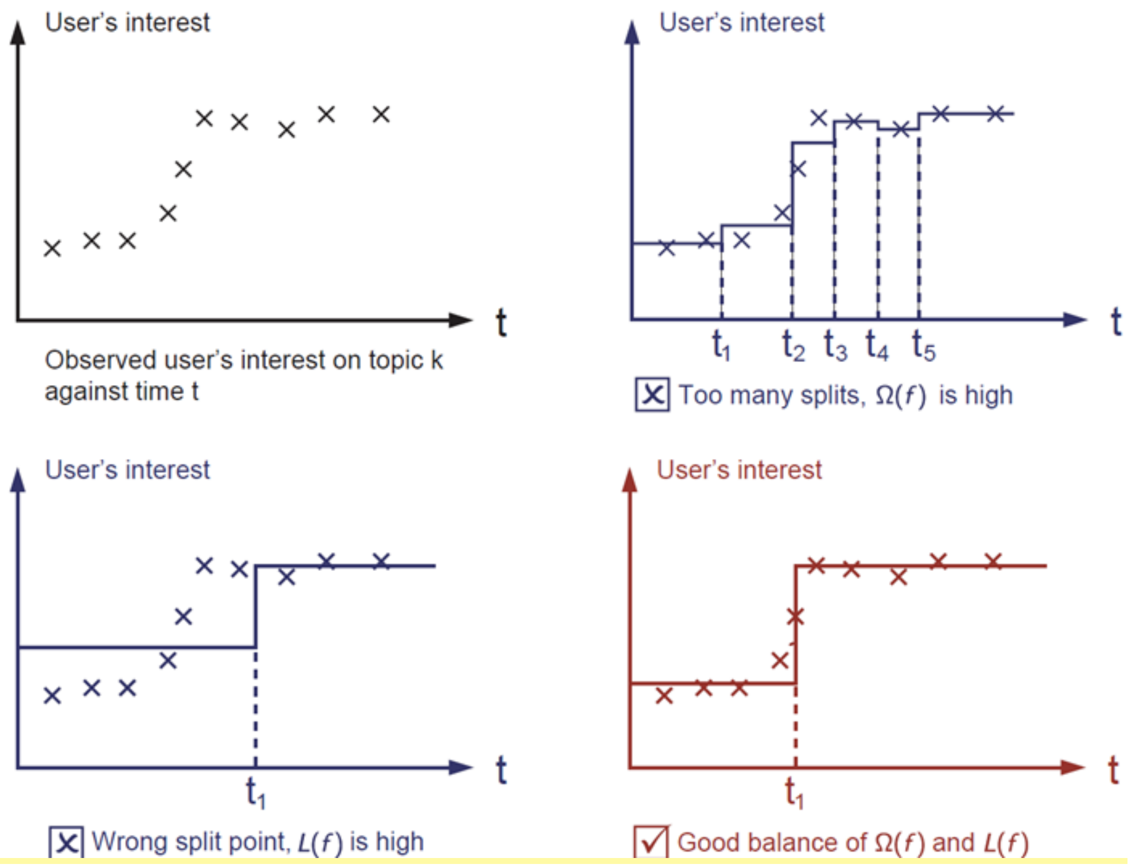
where L is the training loss function, and Ω is the regularization term. The training loss measures how *predictive* our model is with respect to the training data. A common choice of L is the *mean squared error*, which is given by

$$L(\theta) = \sum_i (y_i - \hat{y}_i)^2$$

Another commonly used loss function is logistic loss, to be used for logistic regression:

$$L(\theta) = \sum_i [y_i \ln(1 + e^{-\hat{y}_i}) + (1 - y_i) \ln(1 + e^{\hat{y}_i})]$$

The **regularization term** is what people usually forget to add. The regularization term controls the complexity of the model, which helps us to avoid overfitting. This sounds a bit abstract, so let us consider the following problem in the following picture. You are asked to *fit* visually a step function given the input data points on the upper left corner of the image. Which solution among the three do you think is the best fit?



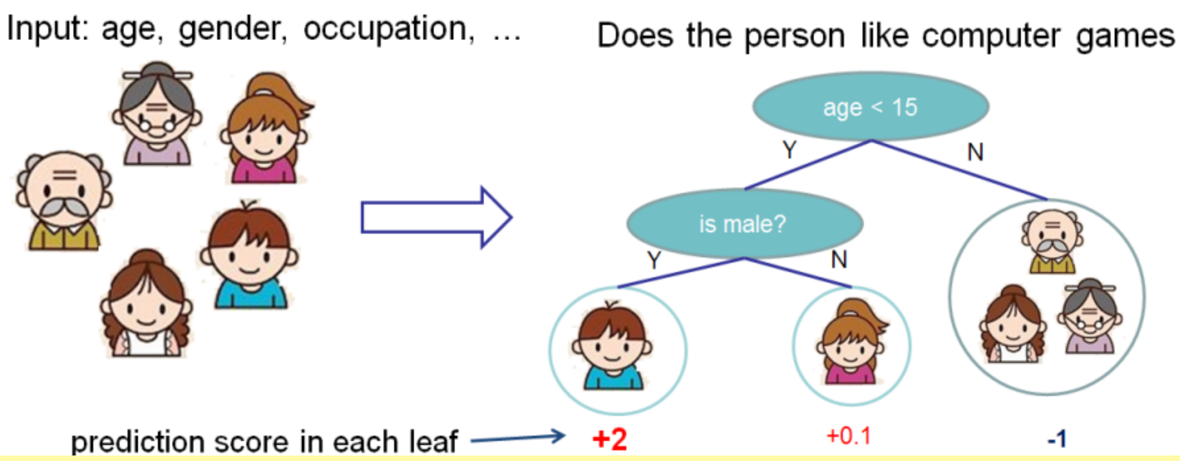
The correct answer is marked in red. Please consider if this visually seems a reasonable fit to you. The general principle is we want both a *simple* and *predictive* model. The tradeoff between the two is also referred as **bias-variance tradeoff** in machine learning.

Why introduce the general principle?

The elements introduced above form the basic elements of supervised learning, and they are natural building blocks of machine learning toolkits. For example, you should be able to describe the differences and commonalities between gradient boosted trees and random forests. Understanding the process in a formalized way also helps us to understand the objective that we are learning and the reason behind the heuristics such as pruning and smoothing.

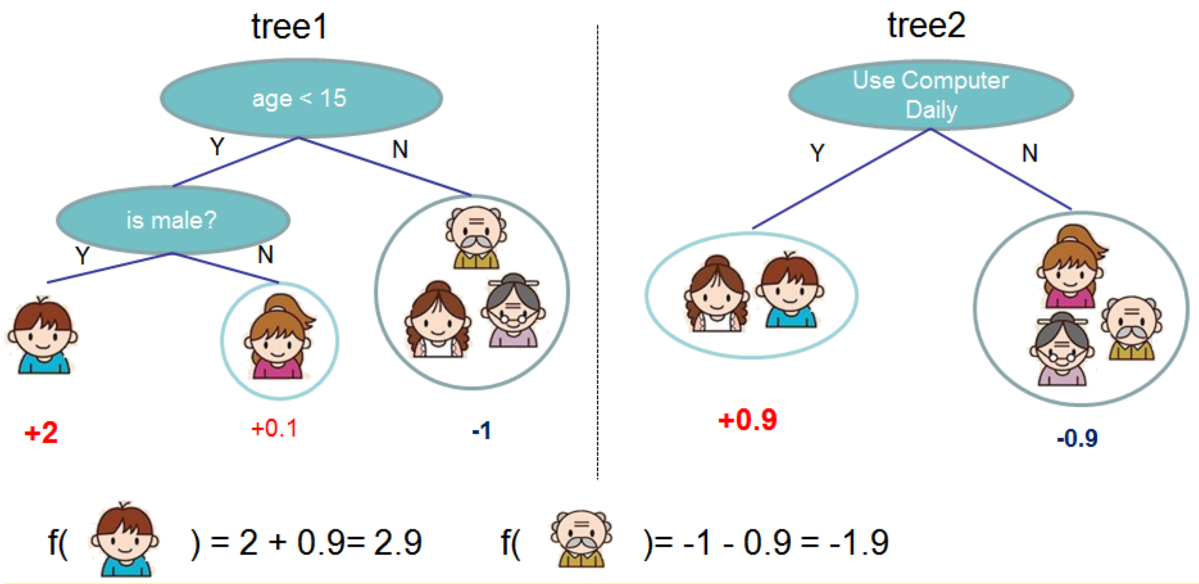
Decision Tree Ensembles

Now that we have introduced the elements of supervised learning, let us get started with real trees. To begin with, let us first learn about the model choice of XGBoost: **decision tree ensembles**. The tree ensemble model consists of a set of classification and regression trees (CART). Here's a simple example of a CART that classifies whether someone will like computer games.


Input: age, gender, occupation, …
Does the person like computer games
age < 15
Y
N
is male?
Y
N
prediction score in each leaf
+2
+0.1
-1

We classify the members of a family into different leaves, and assign them the score on the corresponding leaf. A CART is a bit different from decision trees, in which the leaf only contains decision values. In CART, a real score is associated with each of the leaves, which gives us richer interpretations that go beyond classification. This also allows for a pricipled, unified approach to optimization, as we will see in a later part of this tutorial.

Usually, a single tree is not strong enough to be used in practice. What is actually used is the ensemble model, which sums the prediction of multiple trees together.



Here is an example of a tree ensemble of two trees. The prediction scores of each individual tree are summed up to get the final score. If you look at the example, an important fact is that the two trees try to *complement* each other. Mathematically, we can write our model in the form

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), f_k \in \mathcal{F}$$

y^i=∑k=1Kfk(xi),fk∈F

where K is the number of trees, f is a function in the functional space $\mathcal{F}$, and $\mathcal{F}$ is the set of all possible CARTs. The objective function to be optimized is given by

$$\text{obj}(\theta) = \sum_{i}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k)$$

Now here comes a trick question: what is the *model* used in random forests? Tree ensembles! So random forests and boosted trees are really the same models; the difference arises from how we train them. This means that, if you write a predictive service for tree ensembles, you only need to write one and it should work for both random forests and gradient boosted trees. (See Treelite for an actual example.) One example of why elements of supervised learning rock.

Tree Boosting

Now that we introduced the model, let us turn to training: How should we learn the trees? The answer is, as is always for all supervised learning models: *define an objective function and optimize it*!
Let the following be the objective function (remember it always needs to contain training loss and regularization):

$$\text{obj} = \sum_{i=1}^n l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^t \Omega(f_i)$$

Additive Training

The first question we want to ask: what are the **parameters** of trees? You can find that what we need to learn are those functions f_i, each containing the structure of the tree and the leaf scores. Learning tree structure is much harder than traditional optimization problem where you can simply take the gradient. It is intractable to learn all the trees at once. Instead, we use an additive strategy: fix what we have learned, and add one new tree at a time. We write the prediction value at step t

as $\hat{y}_i^{(t)}$. Then we have

$$\begin{aligned}\hat{y}_i^{(0)} &= 0\\ \hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i)\\ \hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i)\\ &\dots\\ \hat{y}_i^{(t)} &= \sum_{k=1}^t f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)\end{aligned}$$

It remains to ask: which tree do we want at each step? A natural thing is to add the one that optimizes our objective.

$$\begin{aligned}\text{obj}^{(t)} &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^t \Omega(f_i)\\ &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + \mathrm{constant}\end{aligned}$$

If we consider using mean squared error (MSE) as our loss function, the objective becomes

$$\begin{aligned}\text{obj}^{(t)} &= \sum_{i=1}^n (y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)))^2 + \sum_{i=1}^t \Omega(f_i)\\ &= \sum_{i=1}^n [2(\hat{y}_i^{(t-1)} - y_i)f_t(x_i) + f_t(x_i)^2] + \Omega(f_t) + \mathrm{constant}\end{aligned}$$

The form of MSE is friendly, with a first order term (usually called the residual) and a quadratic term. For other losses of interest (for example, logistic loss), it is not so easy to get such a nice form. So in the general case, we take the *Taylor expansion of the loss function up to the second order*:

$$\text{obj}^{(t)} = \sum_{i=1}^n [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) + \mathrm{constant}$$

where the g_i and h_i are defined as

$$\begin{aligned}g_i &= \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})\\ h_i &= \partial_{\hat{y}_i^{(t-1)}}^2 l(y_i, \hat{y}_i^{(t-1)})\end{aligned}$$

After we remove all the constants, the specific objective at step t becomes

$$\sum_{i=1}^n [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

This becomes our optimization goal for the new tree. One important advantage of this definition is that the value of the objective function only depends on g_i and h_i. This is how XGBoost supports custom loss

functions. We can optimize every loss function, including logistic regression and pairwise ranking, using exactly the same solver that takes
g_i and h_i as input!

Model Complexity

We have introduced the training step, but wait, there is one important thing, the **regularization term**! We need to define the complexity of the tree $\Omega(f)$. In order to do so, let us first refine the definition of the tree $f(x)$ as

$$f_t(x)=w_{q(x)}, w\in R_T, q:R_d\to\{1,2,\cdots,T\}.$$

Here w is the vector of scores on leaves, q is a function assigning each data point to the corresponding leaf, and T is the number of leaves. In XGBoost, we define the complexity as

$$\Omega(f)=\gamma T+12\lambda\sum_{j=1T}w_{2j}$$

Of course, there is more than one way to define the complexity, but this one works well in practice. The regularization is one part most tree packages treat less carefully, or simply ignore. This was because the traditional treatment of tree learning only emphasized improving impurity, while the complexity control was left to heuristics. By defining it formally, we can get a better idea of what we are learning and obtain models that perform well in the wild.

The Structure Score

Here is the magical part of the derivation. After re-formulating the tree model, we can write the objective value with the$t$
-th tree as:

$$\text{obj}_{(t)}\approx\sum_{i=1n}[g_iw_{q(x_i)}+12h_iw_{2q(x_i)}]+\gamma T+12\lambda\sum_{j=1T}w_{2j}=\sum_{j=1T}[(\sum_{i\in I_j}g_i)w_j+12(\sum_{i\in I_j}h_i+\lambda)w_{2j}]+\gamma T$$

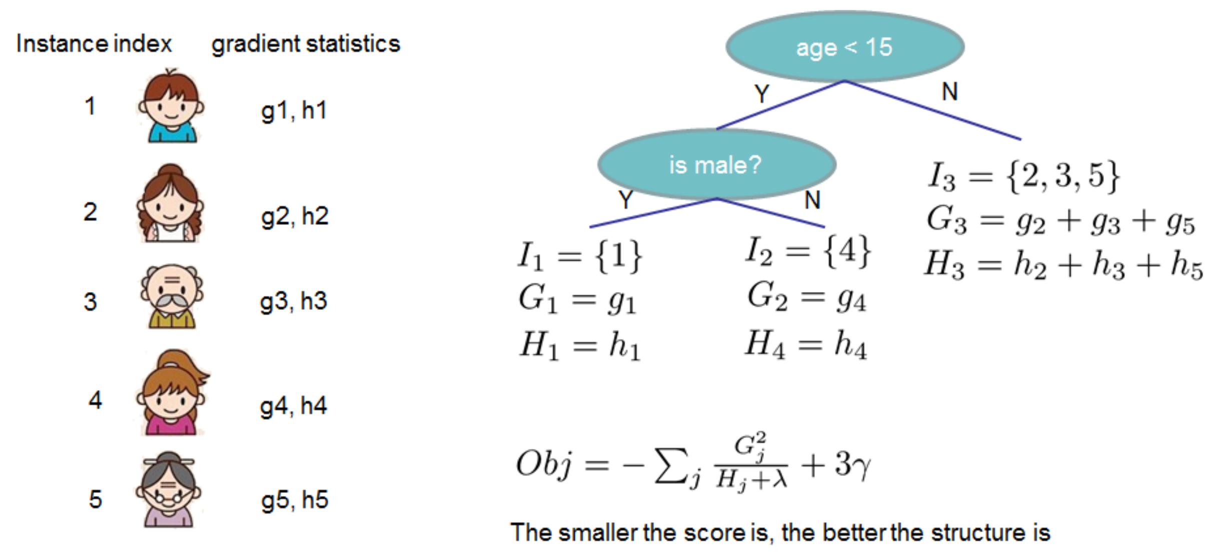
where $I_j=\{i|q(x_i)=j\}$ is the set of indices of data points assigned to the j -th leaf. Notice that in the second line we have changed the index of the summation because all the data points on the same leaf get the same score. We could further compress the expression by defining $G_j=\sum_{i\in I_j}g_i$ and $H_j=\sum_{i\in I_j}h_i$:

$$\text{obj}_{(t)}=\sum_{j=1T}[G_jw_j+12(H_j+\lambda)w_{2j}]+\gamma T$$

In this equation, w_j are independent with respect to each other, the form $G_jw_j+12(H_j+\lambda)w_{2j}$ is quadratic and the best w_j
for a given structure $q(x)$ and the best objective reduction we can get is:

$$w_{*j}\text{obj}_*=-G_jH_j+\lambda=-12\sum_{j=1T}G_{2j}H_j+\lambda+\gamma T$$

The last equation measures *how good* a tree structure


Instance index
gradient statistics
1
g1, h1
2
g2, h2
3
g3, h3
4
g4, h4
5
g5, h5
age < 15
Y
N
is male?
Y
N
$I_1 = \{1\}$ $G_1 = g_1$ $H_1 = h_1$
$I_2 = \{4\}$ $G_2 = g_4$ $H_4 = h_4$
$I_3 = \{2, 3, 5\}$ $G_3 = g_2 + g_3 + g_5$ $H_3 = h_2 + h_3 + h_5$
$Obj = -\sum_j \frac{G_j^2}{H_j+\lambda} + 3\gamma$
The smaller the score is, the better the structure is

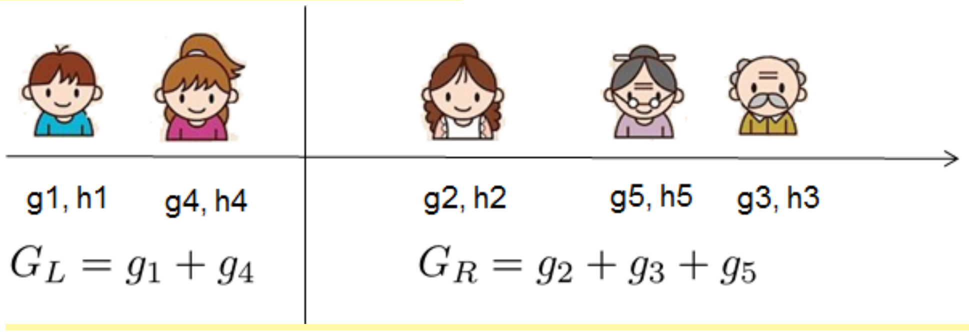
$q(x)$ is. If all this sounds a bit complicated, let's take a look at the picture, and see how the scores can be calculated. Basically, for a given tree structure, we push the statistics g_i and h_i to the leaves they belong to, sum the statistics together, and use the formula to calculate how good the tree is. This score is like the impurity measure in a decision tree, except that it also takes the model complexity into account.

Learn the tree structure

Now that we have a way to measure how good a tree is, ideally we would enumerate all possible trees and pick the best one. In practice this is intractable, so we will try to optimize one level of the tree at a time. Specifically we try to split a leaf into two leaves, and the score it gains is

$$Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right]-\gamma$$

This formula can be decomposed as 1) the score on the new left leaf 2) the score on the new right leaf 3) The score on the original leaf 4) regularization on the additional leaf. We can see an important fact here: if the gain is smaller than γ, we would do better not to add that branch. This is exactly the **pruning** techniques in tree based models! By using the principles of supervised learning, we can naturally come up with the reason these techniques work :)
For real valued data, we usually want to search for an optimal split. To efficiently do so, we place all the instances in sorted order, like the following picture.



A left to right scan is sufficient to calculate the structure score of all possible split solutions, and we can find the best split efficiently.

Final words on XGBoost

Now that you understand what boosted trees are, you may ask, where is the introduction for XGBoost? XGBoost is exactly a tool motivated by the formal principle introduced in this tutorial! More importantly, it is developed with both deep consideration in terms of **systems optimization** and **principles in machine learning**. The goal of this library is to push the extreme of the computation limits of machines to provide a **scalable**, **portable** and **accurate** library. Make sure you try it out, and most importantly, contribute your piece of wisdom (code, examples, tutorials) to the community!

- iPython 应用实例

Python默认情况下是不带XGBoost的，需要利用pip下载对应的模块，如果出现没有模块的提示，可以用下面的方式解决：

```
➜  xgboost git:(master) ✗ ipython test.py
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
/Users/genesis/Documents/GitHub/Machine-Learning/常用的机器学习算法/xgboost/test.py in <module>()
      1 from numpy import loadtxt
----> 2 from xgboost import XGBClassifier
      3 from sklearn.model_selection import train_test_split
      4 from sklearn.metrics import accuracy_score
      5
ImportError: No module named xgboost
➜  xgboost git:(master) ✗ pip install xgboost
Collecting xgboost
  Downloading
https://files.pythonhosted.org/packages/4f/4c/4969b10939c4557ae46e5569d07c0c7ce772b3d6b9c1401a6ed07059fdee/xgboost-0.81.tar.gz (636kB)
    100% |████████████████████████████████| 645kB 9.4kB/s
Requirement already satisfied: numpy in /usr/local/lib/python2.7/site-packages (from xgboost) (1.15.4)
Requirement already satisfied: scipy in /usr/local/lib/python2.7/site-packages (from xgboost) (1.2.0)
Building wheels for collected packages: xgboost
  Running setup.py bdist_wheel for xgboost … \
Stored in directory:
/Users/genesis/Library/Caches/pip/wheels/8a/89/0c/4870bb69132698f40889fa956f92e630a36c1fd7173fcc759f
Successfully built xgboost
Installing collected packages: xgboost
Successfully installed xgboost-0.81
```

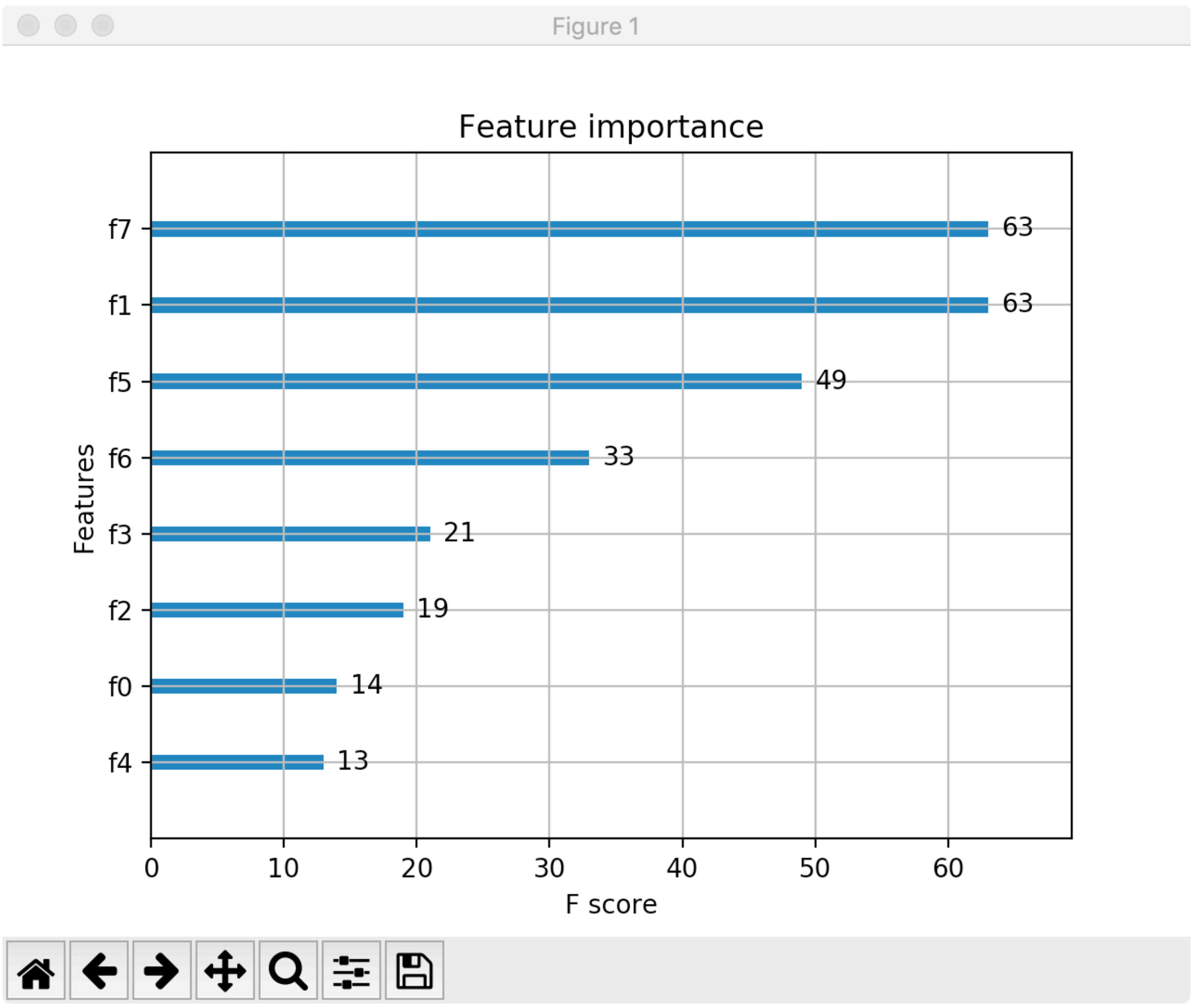
安装完成后，编写代码如下：

```
原始数据：
1。怀孕次数
2。口服葡萄糖耐量试验中2小时的血糖浓度
3。舒张压（mm Hg）
4。三头肌皮褶厚度（mm）
5。2小时血清胰岛素（μu/ml）
6。体重指数（重量单位：kg/（高度单位：m）^2）
7。糖尿病谱系功能
8。年龄（年）
9。类变量（0或1）

# 英文原始
# 1. Number of times pregnant
# 2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test
# 3. Diastolic blood pressure (mm Hg)
# 4. Triceps skin fold thickness (mm)
# 5. 2-Hour serum insulin (mu U/ml)
# 6. Body mass index (weight in kg/(height in m)^2)
# 7. Diabetes pedigree function
# 8. Age (years)
# 9. Class variable (0 or 1)

6    148    72    35    0      33.6    0.627    50    1
1    85     66    29    0      26.6    0.351    31    0
8    183    64    0     0      23.3    0.672    32    1
1    89     66    23    94     28.1    0.167    21    0
0    137    40    35    168    43.1    2.288    33    1
5    116    74    0     0      25.6    0.201    30    0
3    78     50    32    88     31      0.248    26    1
10   115    0     0     0      35.3    0.134    29    0
2    197    70    45    543    30.5    0.158    53    1
8    125    96    0     0      0       0.232    54    1
4    110    92    0     0      37.6    0.191    30    0
10   168    74    0     0      38      0.537    34    1
10   139    80    0     0      27.1    1.441    57    0
1    189    60    23    846    30.1    0.398    59    1
5    166    72    19    175    25.8    0.587    51    1
7    100    0     0     0      30      0.484    32    1
0    118    84    47    230    45.8    0.551    31    1
7    107    74    0     0      29.6    0.254    31    1
1    103    30    38    83     43.3    0.183    33    0
1    115    70    30    96     34.6    0.529    32    1
3    126    88    41    235    39.3    0.704    27    0
8    99     84    0     0      35.4    0.388    50    0


#source code
from numpy import loadtxt
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from xgboost import plot_importance
from matplotlib import pyplot
print ("ok")
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
X = dataset[:,0:8]
Y = dataset[:,8]
seed = 7
test_size = 0.33
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=test_size,
random_state=seed)
#---------------------------二选一执行-----------------------------#
#choice one
#model = XGBClassifier()
#model.fit(X_train, y_train)
#choice two
model = XGBClassifier()
eval_set = [(X_test, y_test)]
model.fit(X_train, y_train, early_stopping_rounds=10, eval_metric="logloss",
eval_set=eval_set, verbose=True)
#---------------------------   END   -----------------------------#
#可视化
plot_importance(model)
pyplot.show()
y_pred = model.predict(X_test)
predictions = [round(value) for value in y_pred]
accuracy = accuracy_score(y_test, predictions)
print("Accuracy: %.2f%%" % (accuracy * 100.0))


执行结果:

➜  Python_Workplace ipython test.py
[0]     validation_0-logloss:0.660186
Will train until validation_0-logloss hasn't improved in 10 rounds.
[1]     validation_0-logloss:0.634854
```

```
[2]	validation_0-logloss:0.612239
[3]	validation_0-logloss:0.593118
[4]	validation_0-logloss:0.578303
[5]	validation_0-logloss:0.564942
[6]	validation_0-logloss:0.555113
[7]	validation_0-logloss:0.54499
[8]	validation_0-logloss:0.539151
[9]	validation_0-logloss:0.531819
[10]	validation_0-logloss:0.526065
[11]	validation_0-logloss:0.51977
[12]	validation_0-logloss:0.514979
[13]	validation_0-logloss:0.50927
[14]	validation_0-logloss:0.506086
[15]	validation_0-logloss:0.503565
[16]	validation_0-logloss:0.503591
[17]	validation_0-logloss:0.500805
[18]	validation_0-logloss:0.497605
[19]	validation_0-logloss:0.495328
[20]	validation_0-logloss:0.494777
[21]	validation_0-logloss:0.494274
[22]	validation_0-logloss:0.493333
[23]	validation_0-logloss:0.492211
[24]	validation_0-logloss:0.491936
[25]	validation_0-logloss:0.490578
[26]	validation_0-logloss:0.490895
[27]	validation_0-logloss:0.490646
[28]	validation_0-logloss:0.491911
[29]	validation_0-logloss:0.491407
[30]	validation_0-logloss:0.488828
[31]	validation_0-logloss:0.487867
[32]	validation_0-logloss:0.487297
[33]	validation_0-logloss:0.487562
[34]	validation_0-logloss:0.487788
[35]	validation_0-logloss:0.487962
[36]	validation_0-logloss:0.488218
[37]	validation_0-logloss:0.489582
[38]	validation_0-logloss:0.489334
[39]	validation_0-logloss:0.490969
[40]	validation_0-logloss:0.48978
[41]	validation_0-logloss:0.490704
[42]	validation_0-logloss:0.492369
Stopping. Best iteration:
[32]	validation_0-logloss:0.487297

Accuracy: 77.56%
```

pima-indians-diabetes.csv

最后的执行结果：



网上也找了一个原始的例子以供参考。有关于XGBoost算法去学习并且预测对应值的例子。

```
# -*- coding=utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
import xgboost as xgb

# train data
def get_train_data(data_size=100):
    data_label = np.zeros((2*data_size, 1))
    # class 1
    x1 = np.reshape(np.random.normal(3, 1, data_size), (data_size, 1))
    y1 = np.reshape(np.random.normal(4, 1, data_size), (data_size, 1))
    data_train = np.concatenate((x1, y1), axis=1)
    data_label[0:data_size, :] = 0
    # class 2
    x2 = np.reshape(np.random.normal(1, 1, data_size), (data_size, 1))
    y2 = np.reshape(np.random.normal(0.5, 1, data_size), (data_size, 1))
    data_train = np.concatenate((data_train, np.concatenate((x2, y2), axis=1)), axis=0)
    data_label[data_size:2*data_size, :] = 1

    return data_train, data_label

# test data
def get_test_data(start, end, data_size=100):
    data1 = (end - start) * np.random.random((data_size, 1)) + start
    data2 = (end - start) * np.random.random((data_size, 1)) + start
    data_test = np.concatenate((data1, data2), axis=1)
    return data_test
```

```python
# show data distribution
def plot_data(train_data, data_size, test_data):
    plt.figure()
    plt.plot(train_data[0:data_size, 0], train_data[0:data_size, 1], 'g.',
             train_data[data_size:2*data_size, 0], train_data[data_size:2*data_size, 1], 'b*',
             test_data[:, 0], test_data[:, 1], 'rs')
    plt.legend(['class1', 'class 2', 'test_data'])
    plt.title('Distribution')
    plt.grid(True)
    plt.xlabel('axis1')
    plt.ylabel('axis2')
    plt.show()

# plot predict res
def plot_predict_data(train_data, data_size, test_data, predict_res1, predict_res2):
    plt.figure()
    plt.subplot(1, 2, 1)
    plt.plot(train_data[0:data_size, 0], train_data[0:data_size, 1], 'g.',
             train_data[data_size:2*data_size, 0], train_data[data_size:2*data_size, 1], 'b*',
             test_data[:, 0], test_data[:, 1], 'ms')
    plt.legend(['class1', 'class2', 'test_data'])
    plt.title('Distribution')
    plt.grid(True)
    plt.xlabel('axis1')
    plt.ylabel('axis2')

    plt.subplot(1, 2, 2)
    plt.plot(train_data[0:data_size, 0], train_data[0:data_size, 1], 'g.',
             train_data[data_size:2 * data_size, 0], train_data[data_size:2 * data_size, 1], 'b*',
             predict_res1[:, 0], predict_res1[:, 1], 'ro',
             predict_res2[:, 0], predict_res2[:, 1], 'rs')
    plt.legend(['class1', 'class2', 'predict1', 'predict2'])
    plt.title('Predict res')
    plt.grid(True)
    plt.xlabel('axis1')
    plt.ylabel('axis2')
    plt.show()

# main function
if __name__ == '__main__':
    # 训练个数
    data_size = 100
    train_data0, label_data = get_train_data(data_size)  # train data generate 训练模型，训练模型数据个数
    test_data0 = get_test_data(-1, 5, 10)  # test data 分别是取样范围 -1~5，取样个数
    # print train_data0
    # 显示原始数据
    plot_data(train_data0, data_size, test_data0)  # plot
    # data convert 转换数据模型
    train_data = xgb.DMatrix(train_data0, label=label_data)
    test_data = xgb.DMatrix(test_data0)

    # data training 训练数据个数
    num_round = 5
    param = {'booster': 'gbtree', 'eta': 0.1, 'max_depth': 5, 'objective': 'binary:logistic'}
    # 训练数据
    bst = xgb.train(param, train_data, num_round)

    # make prediction
    predict_res = bst.predict(test_data)
    print predict_res
    # 阈值范围
    index1 = predict_res > 0.5
    res1 = test_data0[index1, :]
    res2 = test_data0[~index1, :]

    # plot prediction result 结果预判输出
    plot_predict_data(train_data0, data_size, test_data0, res1, res2)
```

原始数据：

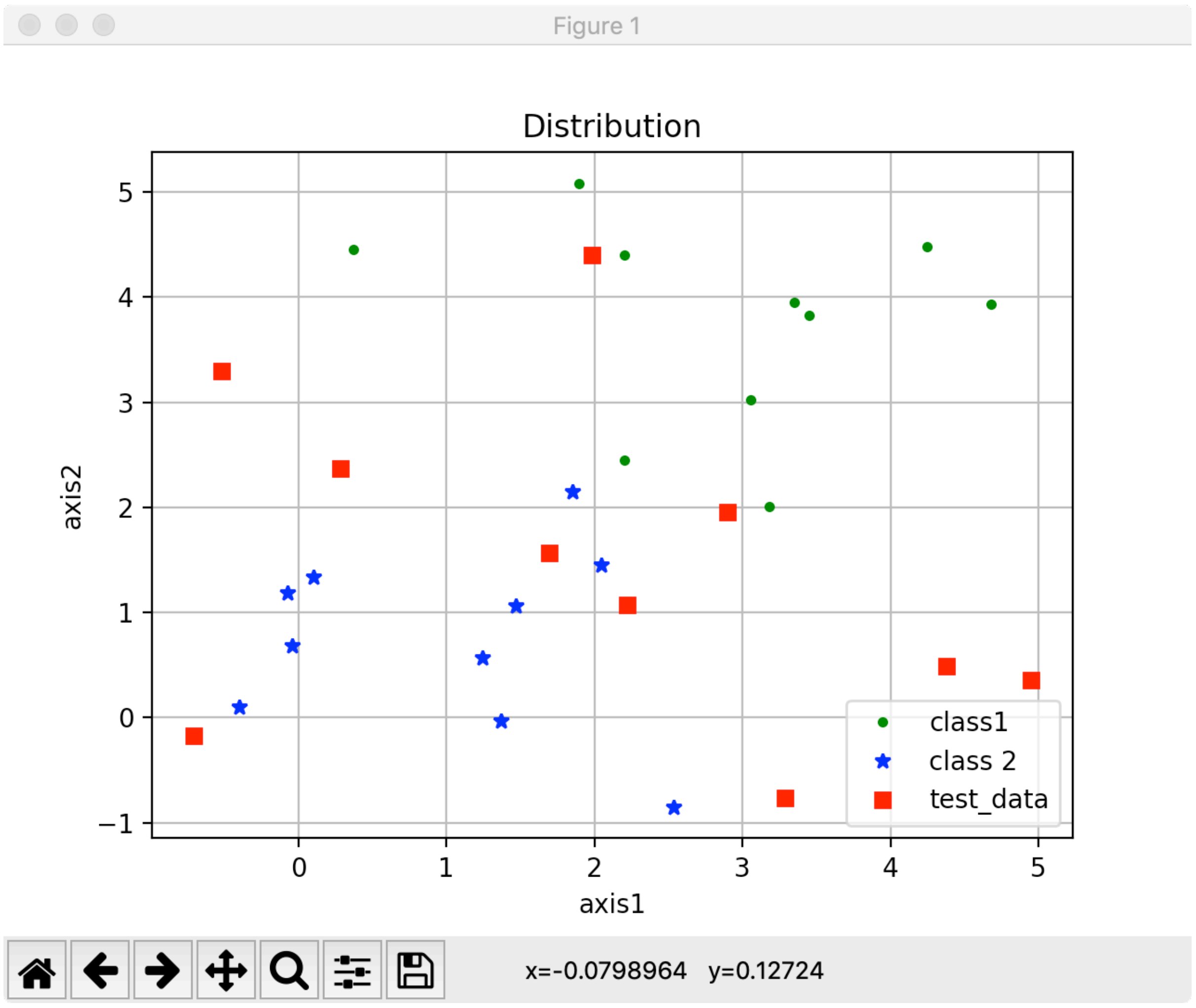
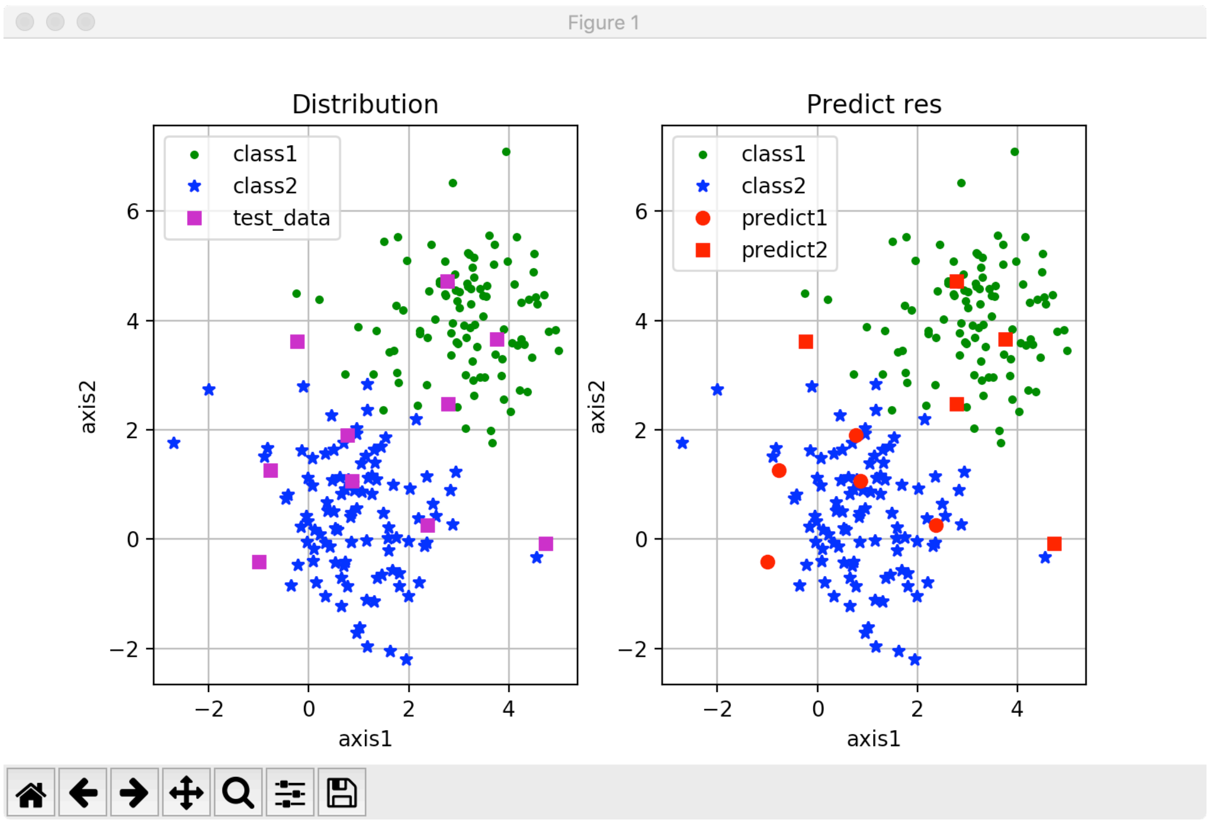


Figure 1
Distribution
axis1
axis2
class1
class 2
test_data
x=-0.0798964   y=0.12724

测绘预测数据：



我们可以看到，他对原始的范围值和实际的值进行了分类，红色圆圈属于分类蓝色，红色方框属于绿色阵营。

By Genesis.Ling
2019.01.30