菜菜的机器学习sklearn第十一期

sklearn与XGBoost

小伙伴们晚上好~o(￣▽￣)ブ

我是菜菜，这里是我的sklearn课堂第十一期，今晚的直播内容是XGBoost~

我的开发环境是**Jupyter lab**，所用的库和版本大家参考：

Python 3.7.1（你的版本至少要3.4以上

Scikit-learn 0.20.1 （你的版本至少要0.20

Numpy 1.15.4, **Pandas** 0.23.4, **Matplotlib** 3.0.2, **SciPy** 1.1.0

请扫码进群领取课件和代码源文件，**扫描二维码后回复"K"就可以进群哦~**

菜菜的机器学习sklearn第十一期

sklearn与XGBoost

Tsai Tsai

1 在学习XGBoost之前

1.1 机器学习竞赛的胜利女神

数据领域人才济济，而机器学习竞赛一直都是数据领域中最重要的自我展示平台之一。无数数据工作者希望能够通过竞赛进行修炼，若能斩获优秀排名，也许就能被伯乐发现，一举登上人生巅峰。不过，竞赛不只是数据工作者的舞台，也是算法们激烈竞争的舞台，若要问这几年来各种机器学习比赛中什么算法风头最盛，XGBoost可谓是独孤求败了。从2016年开始，各大竞赛平台排名前列的解决方案逐渐由XGBoost算法统治，业界甚至将其称之为“机器学习竞赛的胜利女神”。Github上甚至列举了在近年来的多项比赛中XGBoost斩获的冠军列表，其影响力可见一斑。

XGBoost全称是e**X**treme **G**radient **B**oosting，可译为极限梯度提升算法。**它由陈天奇所设计，致力于让提升树突破自身的计算极限，以实现运算快速，性能优秀的工程目标。**和传统的梯度提升算法相比，XGBoost进行了许多改进，它能够比其他使用梯度提升的集成算法更加快速，并且已经被认为是在分类和回归上都拥有超高性能的先进评估器。除了比赛之中，高科技行业和数据咨询等行业也已经开始逐步使用XGBoost，了解这个算法，已经成为学习机器学习中必要的一环。

性能超强的算法往往有着复杂的原理，XGBoost也不能免俗，因此它背后的数学深奥复杂。除此之外，XGBoost与多年前就已经研发出来的算法，**比如决策树，SVM等不同，它是一个集大成的机器学习算法，对大家掌握机器学习中各种概念的程度有较高的要求。**虽然要听懂今天这堂课，你不需要是一个机器学习专家，但你至少需要了解树模型是什么。如果你对机器学习比较好的了解，基础比较牢，那今天的课将会是使你融会贯通的一节课。理解XGBoost，一定能让你在机器学习上更上一层楼。

面对如此复杂的算法，我们几个小时的讲解显然是不能够为大家揭开它的全貌的。但我希望这周的课程内容会成为你在梯度提升算法和XGB上的一个向导，一块敲门砖。本周内容中，我会为大家抽丝剥茧，解析XGBoost原理，带大家了解XGBoost库，并帮助大家理解如何使用和评估梯度提升模型。

本周课中，我将重点为大家回答以下问题：

1. XGBoost是什么？它基于什么数学或机器学习原理来实现？
2. XGBoost都有哪些参数？怎么使用这些参数？
3. 是使用XGBoost的sklearn接口好，还是使用原来的xgboost库比较好？
4. XGBoost使用中会有哪些问题？

学完这周课，我会让你们从这里带走在自己的机器学习项目中能够使用的技术和技能。其中，大部分原理会基于回归树来进行讲解，回归树的参数调整会在讲解中解读完毕，XGB用于分类的用法将会在案例中为大家呈现。至于很复杂的数学原理，我不会带大家刨根问底，而是只会带大家了解一些基本流程，只要大家能够把XGB运用在我们的机器学习项目中来创造真实价值就足够了。

1.2 xgboost库与XGB的sklearn API

在开始讲解XGBoost的细节之前，我先来介绍我们可以调用XGB的一系列库，模块和类。陈天奇创造了XGBoost之后，很快和一群机器学习爱好者建立了专门调用XGBoost库，名为xgboost。xgboost是一个独立的，开源的，专门提供梯度提升树以及XGBoost算法应用的算法库。它和sklearn类似，有一个详细的官方网站可以供我们查看，并且可以与C，Python，R，Julia等语言连用，但需要我们单独安装和下载。

xgboost documents：https://xgboost.readthedocs.io/en/latest/index.html

我们课程全部会基于Python来运行。xgboost库要求我们必须要提供适合的Scipy环境，如果你是使用anaconda安装的Python，你的Scipy环境应该是没有什么问题。以下为大家提供在windows中和MAC使用pip来安装xgboost的代码：

```
#windows
pip install xgboost #安装xgboost库
pip install --upgrade xgboost #更新xgboost库

#MAC
brew install gcc@7
pip3 install xgboost
```

安装完毕之后，我们就能够使用这个库中所带的XGB相关的类了。

```
import xgboost as xgb
```

现在，我们有两种方式可以来使用我们的xgboost库。第一种方式，是直接使用xgboost库自己的建模流程。

其中最核心的，是DMtarix这个读取数据的类，以及train()这个用于训练的类。与sklearn把所有的参数都写在类中的方式不同，xgboost库中必须先使用字典设定参数集，再使用train来将参数及输入，然后进行训练。会这样设计的原因，是因为XGB所涉及到的参数实在太多，全部写在xgb.train()中太长也容易出错。在这里，我为大家准备了params可能的取值以及xgboost.train的列表，给大家一个印象。

`params` {*eta, gamma, max_depth, min_child_weight, max_delta_step, subsample, colsample_bytree, colsample_bylevel, colsample_bynode, lambda, alpha, tree_method string, sketch_eps, scale_pos_weight, updater, refresh_leaf, process_type, grow_policy, max_leaves, max_bin, predictor, num_parallel_tree*}

`xgboost.train`(params, *dtrain*, *num_boost_round=10*, *evals=()*, *obj=None*, *feval=None*, *maximize=False*, *early_stopping_rounds=None*, *evals_result=None*, *verbose_eval=True*, *xgb_model=None*, *callbacks=None*, *learning_rates=None*)

或者，我们也可以选择第二种方法，使用xgboost库中的sklearn的API。这是说，我们可以调用如下的类，并用我们sklearn当中惯例的实例化，fit和predict的流程来运行XGB，并且也可以调用属性比如coef_等等。当然，这是我们回归的类，我们也有用于分类，用于排序的类。他们与回归的类非常相似，因此了解一个类即可。

class `xgboost.XGBRegressor`(*max_depth=3*, *learning_rate=0.1*, *n_estimators=100*, *silent=True*, *objective='reg:linear'*, *booster='gbtree'*, *n_jobs=1*, *nthread=None*, *gamma=0*, *min_child_weight=1*, *max_delta_step=0*, *subsample=1*, *colsample_bytree=1*, *colsample_bylevel=1*, *reg_alpha=0*, *reg_lambda=1*, *scale_pos_weight=1*, *base_score=0.5*, *random_state=0*, *seed=None*, *missing=None*, *importance_type='gain'*, ***kwargs*)

看到这长长的参数条目，可能大家会感到头晕眼花——没错XGB就是这门复杂。但是眼尖的小伙伴可能已经发现了，调用xgboost.train和调用sklearnAPI中的类XGBRegressor，需要输入的参数是不同的，而且看起来相当的不同。但其实，**这些参数只是写法不同，功能是相同的。**比如说，我们的params字典中的第一个参数eta，其实就是我们XGBRegressor里面的参数learning_rate，他们的含义和实现的功能是一模一样的。只不过在sklearnAPI中，开发团队友好地帮助我们将参数的名称调节成了与sklearn中其他的算法类更相似的样子。

所以对我们来说，**使用xgboost中设定的建模流程来建模，和使用sklearnAPI中的类来建模，模型效果是比较相似的，但是xgboost库本身的运算速度（尤其是交叉验证）以及调参手段比sklearn要简单。**我们的课是sklearn课堂，因此在今天的课中，我会先使用sklearnAPI来为大家讲解核心参数，包括不同的参数在xgboost的调用流程和sklearn的API中如何对应，然后我会在应用和案例之中使用xgboost库来为大家展现一个快捷的调参过程。如果大家希望探索一下这两者是否有差异，那必须具体到大家本身的数据集上去观察。

1.3 XGBoost的三大板块

XGBoost本身的核心是基于梯度提升树实现的集成算法，整体来说可以有三个核心部分：集成算法本身，用于集成的弱评估器，以及应用中的其他过程。三个部分中，前两个部分包含了XGBoost的核心原理以及数学过程，最后的部分主要是在XGBoost应用中占有一席之地。我们的课程会主要集中在前两部分，最后一部分内容将会在应用中少量给大家提及。接下来，我们就针对这三个部分，来进行一一的讲解。

参数	集成算法	弱评估器	其他过程
n_estimators	√		
learning_rate	√		
silent	√		
subsample	√		
max_depth		√	
objective		√	
booster		√	
gamma		√	
min_child_weight		√	
max_delta_step		√	
colsample_bytree		√	
colsample_bylevel		√	
reg_alpha		√	
reg_lambda		√	
nthread			√
n_jobs			√
scale_pos_weight			√
base_score			√
seed			√
random_state			√
missing			√
importance_type			√

2 梯度提升树

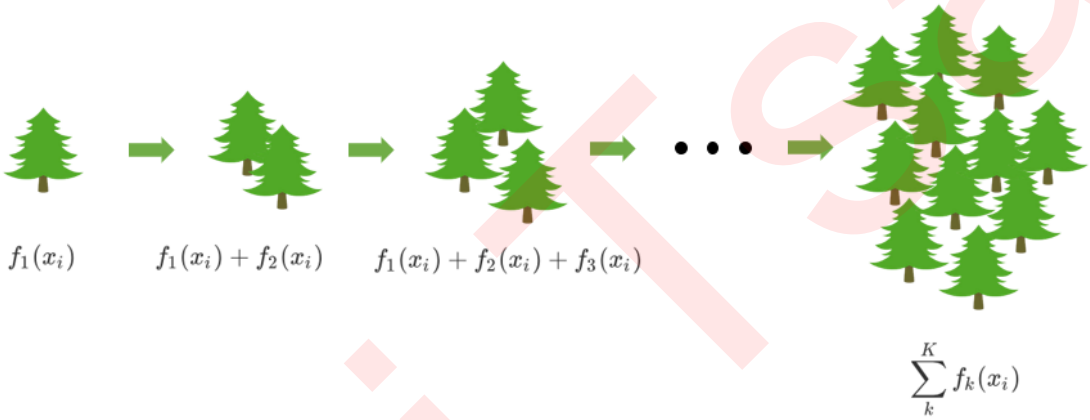
class `xgboost.XGBRegressor`(*max_depth=3*, ***learning_rate=0.1***, ***n_estimators=100***, ***silent=True***, *objective='reg:linear'*, *booster='gbtree'*, *n_jobs=1*, *nthread=None*, *gamma=0*, *min_child_weight=1*, *max_delta_step=0*, ***subsample=1***, *colsample_bytree=1*, *colsample_bylevel=1*, *reg_alpha=0*, *reg_lambda=1*, *scale_pos_weight=1*, *base_score=0.5*, *random_state=0*, *seed=None*, *missing=None*, *importance_type='gain'*, ***kwargs*)

2.1 提升集成算法：重要参数n_estimators

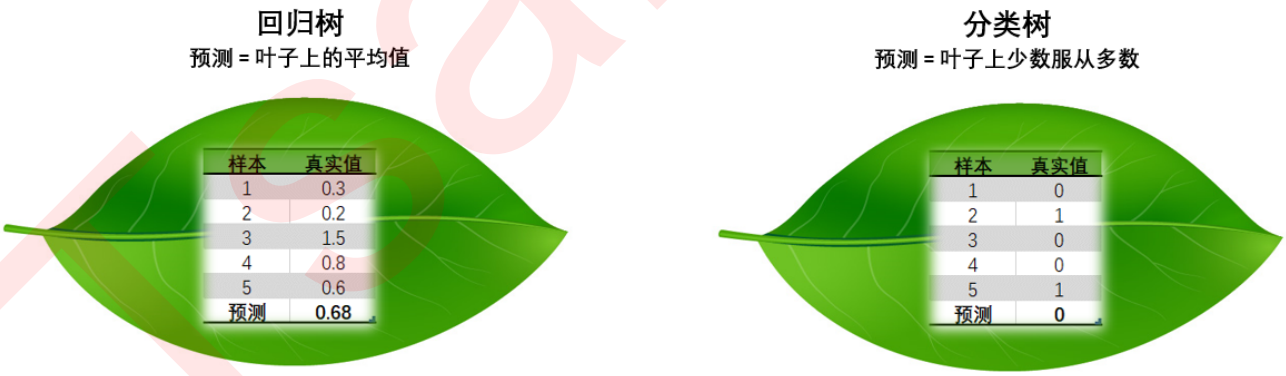
XGBoost的基础是梯度提升算法，因此我们必须先从了解梯度提升算法开始。梯度提升（Gradient boosting）是构建预测模型的最强大技术之一，它是集成算法中提升法（Boosting）的代表算法。**集成算法通过在数据上构建多个弱评估器，汇总所有弱评估器的建模结果，以获取比单个模型更好的回归或分类表现。**弱评估器被定义为是表现至少比随机猜测更好的模型，即预测准确率不低于50%的任意模型。

集成不同弱评估器的方法有很多种。有像我们曾经在随机森林的课中介绍的，一次性建立多个平行独立的弱评估器的装袋法。也有像我们今天要介绍的提升法这样，逐一构建弱评估器，经过多次迭代逐渐累积多个弱评估器的方法。提升法的中最著名的算法包括Adaboost和梯度提升树，XGBoost就是由梯度提升树发展而来的。梯度提升树中可以有回归树也可以有分类树，两者都以CART树算法作为主流，XGBoost背后也是CART树，**这意味着XGBoost中所有的树都是二叉的。**接下来，我们就以梯度提升回归树为例子，来了解一下Boosting算法是怎样工作的。

首先，梯度提升回归树是专注于回归的树模型的提升集成模型，其建模过程大致如下：最开始先建立一棵树，然后逐渐迭代，每次迭代过程中都增加一棵树，逐渐形成众多树模型集成的强评估器。



对于决策树而言，每个被放入模型的任意样本i最终一个都会落到一个叶子节点上。而对于回归树，每个叶子节点上的值是这个叶子节点上所有样本的均值。



对于梯度提升回归树来说，每个样本的预测结果可以表示为所有树上的结果的**加权求和**：

$$\hat{y_i}^{(k)} = \sum_k^K \gamma_k h_k(x_i)$$

其中，K是树的总数量，k代表第k棵树，γ_k是这棵树的权重，h_k表示这棵树上的预测结果。

值得注意的是，XGB作为GBDT的改进，在$\hat{y}$上却有所不同。对于XGB来说，每个叶子节点上会有一个预测分数（prediction score），也被称为叶子权重。**这个叶子权重就是所有在这个叶子节点上的样本在这一棵树上的回归取值，用$f_k(x_i)$或者w来表示**，其中f_k表示第k棵决策树，x_i表示样本i对应的特征向量。当只有一棵树的时候，$f_1(x_i)$就是提升集成算法返回的结果，但这个结果往往非常糟糕。当有多棵树的时候，集成模型的回归结果就是所有树的预测分数之和，假设这个集成模型中总共有K棵决策树，则整个模型在这个样本i上给出的预测结果为：

$$\hat{y_i}^{(k)} = \sum_{k}^{K} f_k(x_i)$$

XGB vs GBDT 核心区别1：求解预测值$\hat{y}$的方式不同
GBDT中预测值是由所有弱分类器上的预测结果的加权求和，其中每个样本上的预测结果就是样本所在的叶子节点的均值。而XGBT中的预测值是所有弱分类器上的叶子权重直接求和得到，**计算叶子权重是一个复杂的过程。**

从上面的式子来看，在集成中我们需要的考虑的第一件事是我们的超参数K，究竟要建多少棵树呢？

参数含义	**xgb.train()**	**xgb.XGBRegressor()**
集成中弱评估器的数量	num_round，默认10	n_estimators，默认100
训练中是否打印每次训练的结果	slient，默认False	slient，默认True

试着回想一下我们在随机森林中是如何理解n_estimators的：n_estimators越大，模型的学习能力就会越强，模型也越容易过拟合。在随机森林中，我们调整的第一个参数就是n_estimators，这个参数非常强大，常常能够一次性将模型调整到极限。在XGB中，我们也期待相似的表现，虽然XGB的集成方式与随机森林不同，但使用更多的弱分类器来增强模型整体的学习能力这件事是一致的。

先来进行一次简单的建模试试看吧。

1. **导入需要的库，模块以及数据**

```
from xgboost import XGBRegressor as XGBR
from sklearn.ensemble import RandomForestRegressor as RFR
from sklearn.linear_model import LinearRegression as LinearR
from sklearn.datasets import load_boston
from sklearn.model_selection import KFold, cross_val_score as CVS, train_test_split as
TTS
from sklearn.metrics import mean_squared_error as MSE
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from time import time
import datetime

data = load_boston()
#波士顿数据集非常简单，但它所涉及到的问题却很多

X = data.data
y = data.target
```

2. **建模，查看其他接口和属性**

```
Xtrain,Xtest,Ytrain,Ytest = TTS(X,y,test_size=0.3,random_state=420)

reg = XGBR(n_estimators=100).fit(Xtrain,Ytrain)
reg.predict(Xtest) #传统接口predict
reg.score(Xtest,Ytest) #你能想出这里应该返回什么模型评估指标么?

MSE(Ytest,reg.predict(Xtest))

reg.feature_importances_ #树模型的优势之一：能够查看模型的重要性分数，可以使用嵌入法进行特征选择
```

3. **交叉验证，与线性回归&随机森林回归进行对比**

```
reg = XGBR(n_estimators=100)
CVS(reg,Xtrain,Ytrain,cv=5).mean()
#这里应该返回什么模型评估指标，还记得么?
#严谨的交叉验证与不严谨的交叉验证之间的讨论：训练集or全数据?

CVS(reg,Xtrain,Ytrain,cv=5,scoring='neg_mean_squared_error').mean()

#来查看一下sklearn中所有的模型评估指标
import sklearn
sorted(sklearn.metrics.SCORERS.keys())

#使用随机森林和线性回归进行一个对比
rfr = RFR(n_estimators=100)
CVS(rfr,Xtrain,Ytrain,cv=5).mean()

CVS(rfr,Xtrain,Ytrain,cv=5,scoring='neg_mean_squared_error').mean()

lr = LinearR()
CVS(lr,Xtrain,Ytrain,cv=5).mean()

CVS(lr,Xtrain,Ytrain,cv=5,scoring='neg_mean_squared_error').mean()

#如果开启参数slient：在数据巨大，预料到算法运行会非常缓慢的时候可以使用这个参数来监控模型的训练进度
reg = XGBR(n_estimators=10,silent=False)
CVS(reg,Xtrain,Ytrain,cv=5,scoring='neg_mean_squared_error').mean()
```

4. **定义绘制以训练样本数为横坐标的学习曲线的函数**

```
def plot_learning_curve(estimator,title, X, y,
                        ax=None, #选择子图
                        ylim=None, #设置纵坐标的取值范围
                        cv=None, #交叉验证
                        n_jobs=None #设定索要使用的线程
                       ):

    from sklearn.model_selection import learning_curve
    import matplotlib.pyplot as plt
    import numpy as np
```

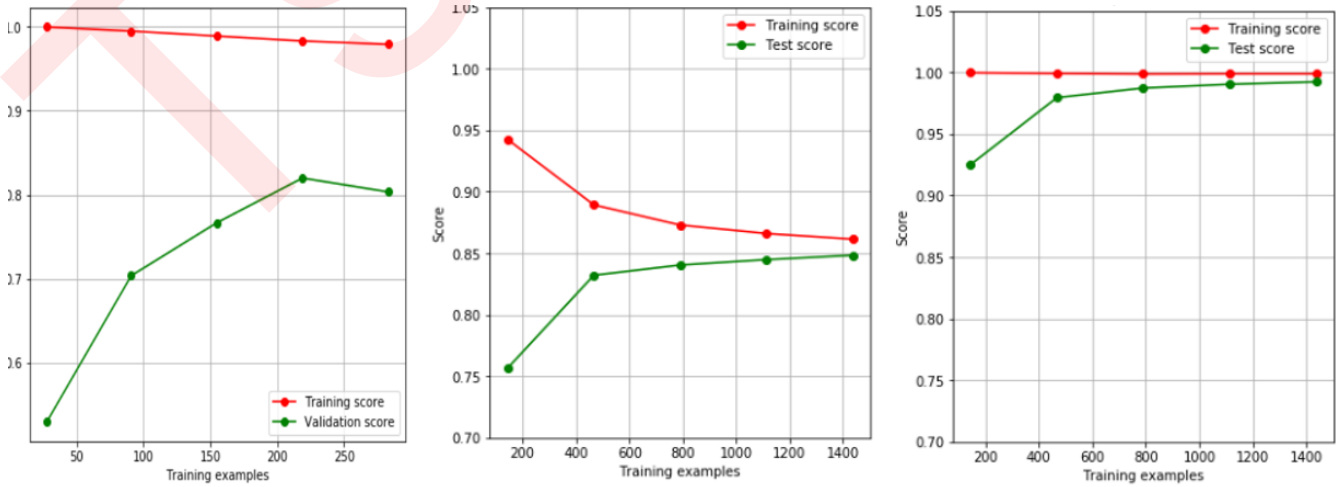
```
    train_sizes, train_scores, test_scores = learning_curve(estimator, X, y
                                                            ,shuffle=True
                                                            ,cv=cv
                                                           # ,random_state=420
                                                            ,n_jobs=n_jobs)
    if ax == None:
        ax = plt.gca()
    else:
        ax = plt.figure()
    ax.set_title(title)
    if ylim is not None:
        ax.set_ylim(*ylim)
    ax.set_xlabel("Training examples")
    ax.set_ylabel("Score")
    ax.grid() #绘制网格，不是必须
    ax.plot(train_sizes, np.mean(train_scores, axis=1), 'o-'
            , color="r",label="Training score")
    ax.plot(train_sizes, np.mean(test_scores, axis=1), 'o-'
            , color="g",label="Test score")
    ax.legend(loc="best")
    return ax
```

5. **使用学习曲线观察XGB在波士顿数据集上的潜力**

```
cv = KFold(n_splits=5, shuffle = True, random_state=42)
plot_learning_curve(XGBR(n_estimators=100,random_state=420)
                    ,"XGB",Xtrain,Ytrain,ax=None,cv=cv)
plt.show()

#多次运行，观察结果，这是怎么造成的?
#在现在的状况下，如何看数据的潜力？还能调上去么？
```

训练集上的表现展示了模型的学习能力，测试集上的表现展示了模型的泛化能力，通常模型在测试集上的表现不太可能超过训练集，因此我们希望我们的测试集的学习曲线能够努力逼近我们的训练集的学习曲线。来观察三种学习曲线组合：我们希望将我们的模型调整成什么样呢？我们能够将模型调整成什么样呢？



6. **使用参数学习曲线观察n_estimators对模型的影响**

```
#=====【TIME WARNING: 25 seconds】=====#

axisx = range(10,1010,50)
rs = []
for i in axisx:
    reg = XGBR(n_estimators=i,random_state=420)
    rs.append(CVS(reg,Xtrain,Ytrain,cv=cv).mean())
print(axisx[rs.index(max(rs))],max(rs))
plt.figure(figsize=(20,5))
plt.plot(axisx,rs,c="red",label="XGB")
plt.legend()
plt.show()

#选出来的n_estimators非常不寻常，我们是否要选择准确率最高的n_estimators值呢?
```

7. **进化的学习曲线：方差与泛化误差**

回忆一下我们曾经在随机森林中讲解过的方差-偏差困境。在机器学习中，我们用来**衡量模型在未知数据上的准确率**的指标，叫做**泛化误差（Genelization error）**。一个集成模型(f)在未知数据集(D)上的泛化误差$E(f;D)$，由方差(var)，偏差(bais)和噪声(ε)共同决定。其中偏差就是训练集上的拟合程度决定，方差是模型的稳定性决定，噪音是不可控的。而泛化误差越小，模型就越理想。

$$E(f;D) = bias^2 + var + \epsilon^2$$

在过去我们往往直接取学习曲线获得的分数的最高点，即考虑偏差最小的点，是因为模型极度不稳定，方差很大的情况其实比较少见。但现在我们的数据量非常少，模型会相对不稳定，因此我们应当将方差也纳入考虑的范围。在绘制学习曲线时，我们不仅要考虑偏差的大小，还要考虑方差的大小，更要考虑泛化误差中我们可控的部分。当然，并不是说可控的部分比较小，整体的泛化误差就一定小，因为误差有时候可能占主导。让我们基于这种思路，来改进学习曲线：

```
#======【TIME WARNING: 20s】=======#
axisx = range(50,1050,50)
rs = []
var = []
ge = []
for i in axisx:
    reg = XGBR(n_estimators=i,random_state=420)
    cvresult = CVS(reg,Xtrain,Ytrain,cv=cv)
    #记录1-偏差
    rs.append(cvresult.mean())
    #记录方差
    var.append(cvresult.var())
    #计算泛化误差的可控部分
    ge.append((1 - cvresult.mean())**2+cvresult.var())
#打印R2最高所对应的参数取值，并打印这个参数下的方差
print(axisx[rs.index(max(rs))],max(rs),var[rs.index(max(rs))])
#打印方差最低时对应的参数取值，并打印这个参数下的R2
print(axisx[var.index(min(var))],rs[var.index(min(var))],min(var))
#打印泛化误差可控部分的参数取值，并打印这个参数下的R2，方差以及泛化误差的可控部分
```

```
print(axisx[ge.index(min(ge))],rs[ge.index(min(ge))],var[ge.index(min(ge))],min(ge))
plt.figure(figsize=(20,5))
plt.plot(axisx,rs,c="red",label="XGB")
plt.legend()
plt.show()
```

8. **细化学习曲线，找出最佳n_estimators**

```
axisx = range(100,300,10)
rs = []
var = []
ge = []
for i in axisx:
    reg = XGBR(n_estimators=i,random_state=420)
    cvresult = CVS(reg,Xtrain,Ytrain,cv=cv)
    rs.append(cvresult.mean())
    var.append(cvresult.var())
    ge.append((1 - cvresult.mean())**2+cvresult.var())
print(axisx[rs.index(max(rs))],max(rs),var[rs.index(max(rs))])
print(axisx[var.index(min(var))],rs[var.index(min(var))],min(var))
print(axisx[ge.index(min(ge))],rs[ge.index(min(ge))],var[ge.index(min(ge))],min(ge))
rs = np.array(rs)
var = np.array(var)*0.01
plt.figure(figsize=(20,5))
plt.plot(axisx,rs,c="black",label="XGB")
#添加方差线
plt.plot(axisx,rs+var,c="red",linestyle='-.')
plt.plot(axisx,rs-var,c="red",linestyle='-.')
plt.legend()
plt.show()

#看看泛化误差的可控部分如何?
plt.figure(figsize=(20,5))
plt.plot(axisx,ge,c="gray",linestyle='-.')
plt.show()
```

9. **检测模型效果**

```
#验证模型效果是否提高了?
time0 = time()
print(XGBR(n_estimators=100,random_state=420).fit(Xtrain,Ytrain).score(Xtest,Ytest))
print(time()-time0)

time0 = time()
print(XGBR(n_estimators=660,random_state=420).fit(Xtrain,Ytrain).score(Xtest,Ytest))
print(time()-time0)

time0 = time()
print(XGBR(n_estimators=180,random_state=420).fit(Xtrain,Ytrain).score(Xtest,Ytest))
print(time()-time0)
```

从这个过程中观察n_estimators参数对模型的影响，我们可以得出以下结论：

首先，XGB中的树的数量决定了模型的学习能力，树的数量越多，模型的学习能力越强。只要XGB中树的数量足够了，即便只有很少的数据， 模型也能够学到训练数据100%的信息，所以XGB也是天生过拟合的模型。但在这种情况下，模型会变得非常不稳定。

第二，XGB中树的数量很少的时候，对模型的影响较大，当树的数量已经很多的时候，对模型的影响比较小，只能有微弱的变化。当数据本身就处于过拟合的时候，再使用过多的树能达到的效果甚微，反而浪费计算资源。当唯一指标R^2或者准确率给出的n_estimators看起来不太可靠的时候，我们可以改造学习曲线来帮助我们。

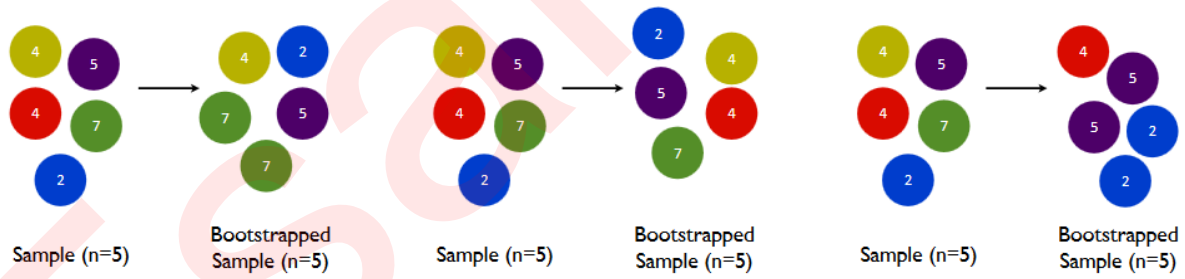
第三，树的数量提升对模型的影响有极限，最开始，模型的表现会随着XGB的树的数量一起提升，但到达某个点之后，树的数量越多，模型的效果会逐步下降，这也说明了暴力增加n_estimators不一定有效果。

这些都和随机森林中的参数n_estimators表现出一致的状态。在随机森林中我们总是先调整n_estimators，当n_estimators的极限已达到，我们才考虑其他参数，但XGB中的状况明显更加复杂，当数据集不太寻常的时候会更加复杂。这是我们要给出的第一个超参数，因此还是建议优先调整n_estimators，一般都不会建议一个太大的数目，300以下为佳。

2.2 有放回随机抽样：重要参数subsample

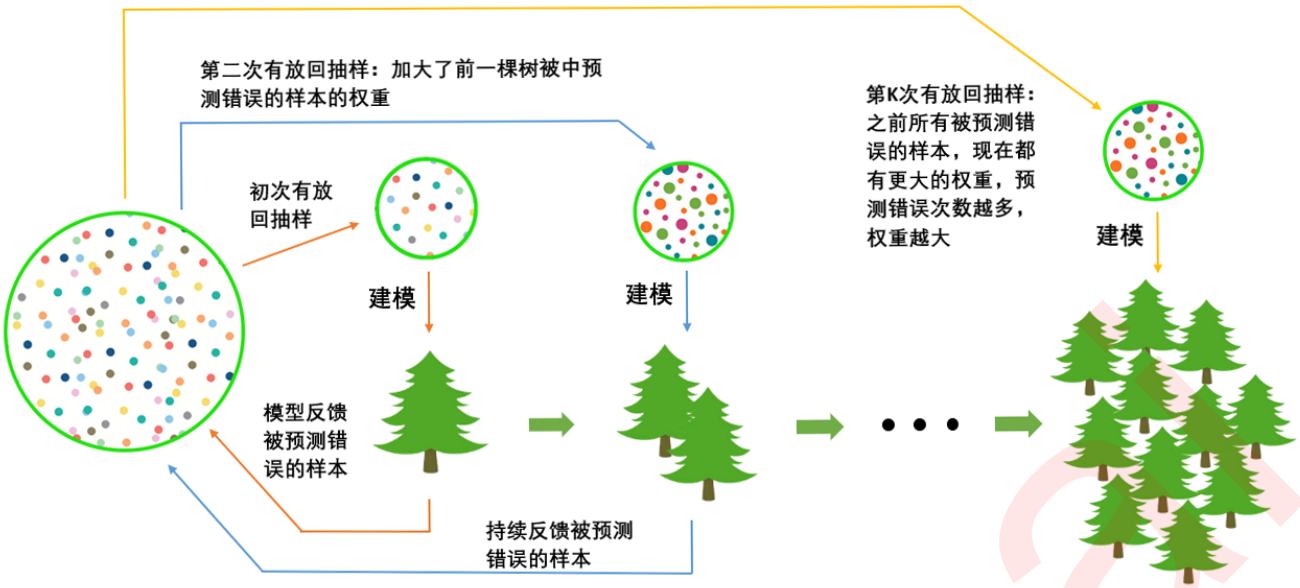
确认了有多少棵树之后，我们来思考一个问题：建立了众多的树，怎么就能够保证模型整体的效果变强呢？集成的目的是为了模型在样本上能表现出更好的效果，所以对于所有的提升集成算法，**每构建一个评估器，集成模型的效果都会比之前更好。**也就是随着迭代的进行，模型整体的效果必须要逐渐提升，最后要实现集成模型的效果最优。要实现这个目标，**我们可以首先从训练数据上着手。**

我们训练模型之前，必然会有一个巨大的数据集。我们都知道树模型是天生过拟合的模型，并且如果数据量太过巨大，树模型的计算会非常缓慢，因此，我们要对我们的原始数据集进行有放回抽样（bootstrap）。有放回的抽样每次只能抽取一个样本，若我们需要总共N个样本，就需要抽取N次。每次抽取一个样本的过程是独立的，这一次被抽到的样本会被放回数据集中，下一次还可能被抽到，因此抽出的数据集中，可能有一些重复的数据。



在无论是装袋还是提升的集成算法中，有放回抽样都是我们防止过拟合，让单一弱分类器变得更轻量的必要操作。实际应用中，每次抽取50%左右的数据就能够有不错的效果了。sklearn的随机森林类中也有名为boostrap的参数来帮助我们控制这种随机有放回抽样。同时，这样做还可以保证集成算法中的每个弱分类器（每棵树）都是不同的模型，基于不同的数据建立的自然是不同的模型，而集成一系列一模一样的弱分类器是没有意义的。

在梯度提升树中，我们每一次迭代都要建立一棵新的树，因此我们每次迭代中，都要有放回抽取一个新的训练样本。不过，这并不能保证每次建新树后，集成的效果都比之前要好。因此我们规定，在梯度提升树中，**每构建一个评估器，都让模型更加集中于数据集中容易被判错的那些样本。**来看看下面的这个过程。



首先我们有一个巨大的数据集，在建第一棵树时，我们对数据进行初次又放回抽样，然后建模。建模完毕后，我们对模型进行一个评估，然后将模型预测错误的样本反馈给我们的数据集，一次迭代就算完成。紧接着，我们要建立第二棵决策树，于是开始进行第二次又放回抽样。但这次有放回抽样，和初次的随机有放回抽样就不同了，**在这次的抽样中，我们加大了被第一棵树判断错误的样本的权重**。也就是说，被第一棵树判断错误的样本，更有可能被我们抽中。

基于这个有权重的训练集来建模，我们**新建的决策树就会更加倾向于这些权重更大的，很容易被判错的样本**。建模完毕之后，我们又将判错的样本反馈给原始数据集。下一次迭代的时候，被判错的样本的权重会更大，新的模型会更加倾向于很难被判断的这些样本。如此反复迭代，越后面建的树，越是之前的树们判错样本上的专家，越专注于攻克那些之前的树们不擅长的数据。对于一个样本而言，它被预测错误的次数越多，被加大权重的次数也就越多。**我们相信，只要弱分类器足够强大，随着模型整体不断在被判错的样本上发力，这些样本会渐渐被判断正确。**如此就一定程度上实现了我们每新建一棵树模型的效果都会提升的目标。

在sklearn中，我们使用参数subsample来控制我们的随机抽样。在xgb和sklearn中，这个参数都默认为1且不能取到0，这说明我们无法控制模型是否进行随机有放回抽样，只能控制抽样抽出来的样本量大概是多少。

参数含义	xgb.train()	xgb.XGBRegressor()
随机抽样的时候抽取的样本比例，范围(0,1]	subsample，默认1	subsample，默认1

那除了让模型更加集中于那些困难样本，采样还对模型造成了什么样的影响呢？采样会减少样本数量，而从学习曲线来看样本数量越少模型的过拟合会越严重，因为对模型来说，数据量越少模型学习越容易，学到的规则也会越具体越不适用于测试样本。所以subsample参数通常是在样本量本身很大的时候来调整和使用。

我们的模型现在正处于样本量过少并且过拟合的状态，根据学习曲线展现出来的规律，我们的训练样本量在200左右的时候，模型的效果有可能反而比更多训练数据的时候好，但这不代表模型的泛化能力在更小的训练样本量下会更强。**正常来说样本量越大，模型才不容易过拟合，现在展现出来的效果，是由于我们的样本量太小造成的一个巧合。**从这个角度来看，我们的subsample参数对模型的影响应该会非常不稳定，大概率应该是无法提升模型的泛化能力的，但也不乏提升模型的可能性。依然使用波士顿房价数据集，来看学习曲线：

```
axisx = np.linspace(0,1,20)
rs = []
for i in axisx:
    reg = XGBR(n_estimators=180,subsample=i,random_state=420)
    rs.append(CVS(reg,Xtrain,Ytrain,cv=cv).mean())
```

```
print(axisx[rs.index(max(rs))],max(rs))
plt.figure(figsize=(20,5))
plt.plot(axisx,rs,c="green",label="XGB")
plt.legend()
plt.show()

#继续细化学习曲线
axisx = np.linspace(0.05,1,20)
rs = []
var = []
ge = []
for i in axisx:
    reg = XGBR(n_estimators=180,subsample=i,random_state=420)
    cvresult = CVS(reg,Xtrain,Ytrain,cv=cv)
    rs.append(cvresult.mean())
    var.append(cvresult.var())
    ge.append((1 - cvresult.mean())**2+cvresult.var())
print(axisx[rs.index(max(rs))],max(rs),var[rs.index(max(rs))])
print(axisx[var.index(min(var))],rs[var.index(min(var))],min(var))
print(axisx[ge.index(min(ge))],rs[ge.index(min(ge))],var[ge.index(min(ge))],min(ge))
rs = np.array(rs)
var = np.array(var)
plt.figure(figsize=(20,5))
plt.plot(axisx,rs,c="black",label="XGB")
plt.plot(axisx,rs+var,c="red",linestyle='-.')
plt.plot(axisx,rs-var,c="red",linestyle='-.')
plt.legend()
plt.show()

#细化学习曲线
axisx = np.linspace(0.75,1,25)

#不要盲目找寻泛化误差可控部分的最低值，注意观察结果

plt.figure(figsize=(20,5))
plt.plot(axisx,ge,c="gray",linestyle='-.')
plt.show()

#看看泛化误差的情况如何
reg = XGBR(n_estimators=180
           ,subsample=0.7708333333333334
           ,random_state=420).fit(Xtrain,Ytrain)
reg.score(Xtest,Ytest)
MSE(Ytest,reg.predict(Xtest))

#这样的结果说明了什么?
```

参数的效果在我们的预料之中，总体来说这个参数并没有对波士顿房价数据集上的结果造成太大的影响，由于我们的数据集过少，降低抽样的比例反而让数据的效果更低，不如就让它保持默认。

2.3 迭代决策树：重要参数eta

从数据的角度而言，我们让模型更加倾向于努力攻克那些难以判断的样本。但是，并不是说只要我新建了一棵倾向于困难样本的决策树，它就能够帮我把困难样本判断正确了。困难样本被加重权重是因为前面的树没能把它判断正确，所以对于下一棵树来说，它要判断的测试集的难度，是比之前的树所遇到的数据的难度都要高的，那要把这些样本都判断正确，会越来越难。如果新建的树在判断困难样本这件事上还没有前面的树做得好呢？如果我新建的树刚好是一棵特别糟糕的树呢？所以，除了保证模型逐渐倾向于困难样本的方向，我们还必须控制新弱分类器的生成，我们必须保证，每次新添加的树一定得是对这个新数据集预测效果最优的那一棵树。

> **思考：怎么保证每次新添加的树一定让集成学习的效果提升？**
>
> 也许我们可以枚举？
> 也许可以学习sklearn中的决策树构建树时一样随机生成固定数目的树，然后生成最好的那一棵？

平衡算法表现和运算速度是机器学习的艺术，我们希望能找出一种方法，直接帮我们求解出最优的集成算法结果。求解最优结果，我们能否把它转化成一个传统的最优化问题呢？

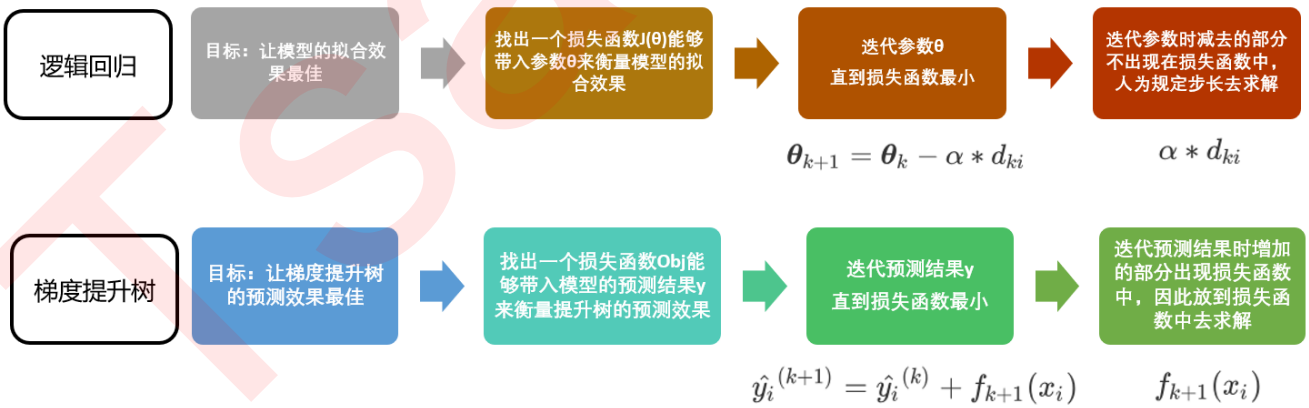
来回顾一下最优化问题的老朋友，我们的逻辑回归模型。在逻辑回归当中，我们有方程：

$$y(x) = \frac{1}{1+e^{-\theta^T x}}$$

我们的目标是求解让逻辑回归的拟合效果最优的参数组合θ。我们首先找出了逻辑回归的损失函数$J(\theta)$，这个损失函数可以通过带入θ来衡量逻辑回归在训练集上的拟合效果。然后，我们利用梯度下降来迭代我们的θ：

$$\theta_{k+1} = \theta_k - \alpha * d_{ki}$$

我们让第k次迭代中的θ_k减去通过步长和特征取值x计算出来的一个量，以此来得到第$k+1$次迭代后的参数向量θ_{k+1}。我们可以让这个过程持续下去，直到我们找到能够让损失函数最小化的参数θ为止。这是一个最典型的最优化过程。这个过程其实和我们现在希望做的事情是相似的。



现在我们希望求解集成算法的最优结果，那我们应该可以使用同样的思路：我们首先找到一个损失函数Obj，这个损失函数应该可以通过带入我们的预测结果$\hat{y}_i$来衡量我们的梯度提升树在样本的预测效果。然后，我们利用梯度下降来迭代我们的集成算法：

$$\hat{y}_i^{(k+1)} = \hat{y}_i^{(k)} + f_{k+1}(x_i)$$

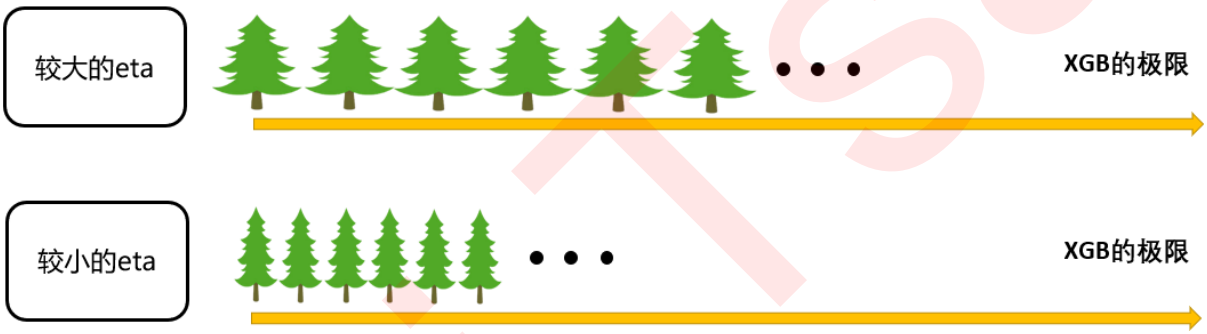
在k次迭代后，我们的集成算法中总共有k棵树，而我们前面讲明了，k棵树的集成结果是前面所有树上的叶子权重的累加$\sum_k^K f_k(x_i)$。所以我们让k棵树的集成结果$\hat{y_i}^{(k)}$加上我们新建的树上的叶子权重$f_{k+1}(x_i)$，就可以得到第$k+1$次迭代后，总共$k+1$棵树的预测结果$\hat{y_i}^{(k+1)}$了。**我们让这个过程持续下去，直到找到能够让损失函数最小化的$\hat{y}$，这个$\hat{y}$就是我们模型的预测结果。**参数可以迭代，集成的树林也可以迭代，万事大吉！

但要注意，在逻辑回归中参数θ迭代的时候减去的部分是我们人为规定的步长和梯度相乘的结果。而在我们的GBDT和XGB中，我们却希望能够求解出让我们的预测结果$\hat{y}$不断迭代的部分$f_{k+1}(x_i)$。但无论如何，我们现在已经有了最优化的思路了，只要顺着这个思路求解下去，我们必然能够在每一个数据集上找到最优的$\hat{y}$。

在逻辑回归中，我们自定义步长α来干涉我们的迭代速率，在XGB中看起来却没有这样的设置，但其实不然。在XGB中，我们完整的迭代决策树的公式应该写作：

$$\hat{y_i}^{(k+1)} = \hat{y_i}^{(k)} + \eta f_{k+1}(x_i)$$

其中η读作"eta"，是迭代决策树时的步长（shrinkage），又叫做学习率（learning rate）。和逻辑回归中的α类似，η越大，迭代的速度越快，算法的极限很快被达到，有可能无法收敛到真正的最佳。η越小，越有可能找到更精确的最佳值，更多的空间被留给了后面建立的树，但迭代速度会比较缓慢。



在sklearn中，我们使用参数learning_rate来干涉我们的学习速率：

参数含义	xgb.train()	xgb.XGBRegressor()
集成中的学习率，又称为步长 以控制迭代速率，常用于防止过拟合	eta，默认0.3 取值范围[0,1]	learning_rate，默认0.1 取值范围[0,1]

让我们来探索一下参数eta的性质：

```
#首先我们先来定义一个评分函数，这个评分函数能够帮助我们直接打印Xtrain上的交叉验证结果
def regassess(reg,Xtrain,Ytrain,cv,scoring = ["r2"],show=True):
    score = []
    for i in range(len(scoring)):
        if show:
            print("{}:{:.2f}".format(scoring[i]
                                     ,CVS(reg
                                          ,Xtrain,Ytrain
                                          ,cv=cv,scoring=scoring[i]).mean()))
        score.append(CVS(reg,Xtrain,Ytrain,cv=cv,scoring=scoring[i]).mean())
    return score

#运行一下函数来看看效果
regassess(reg,Xtrain,Ytrain,cv,scoring = ["r2","neg_mean_squared_error"])
```

```
#关闭打印功能试试看?
regassess(reg,Xtrain,Ytrain,cv,scoring = ["r2","neg_mean_squared_error"],show=False)

#观察一下eta如何影响我们的模型:
from time import time
import datetime

for i in [0,0.2,0.5,1]:
    time0=time()
    reg = XGBR(n_estimators=180,random_state=420,learning_rate=i)
    print("learning_rate = {}".format(i))
    regassess(reg,Xtrain,Ytrain,cv,scoring = ["r2","neg_mean_squared_error"])
    print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S:%f"))
    print("\t")
```

除了运行时间，步长还是一个对模型效果影响巨大的参数，如果设置太大模型就无法收敛（可能导致R^2很小或者MSE很大的情况），如果设置太小模型速度就会非常缓慢，但它最后究竟会收敛到何处很难由经验来判定，在训练集上表现出来的模样和在测试集上相差甚远，很难直接探索出一个泛化误差很低的步长。

```
axisx = np.arange(0.05,1,0.05)
rs = []
te = []
for i in axisx:
    reg = XGBR(n_estimators=180,random_state=420,learning_rate=i)
    score = regassess(reg,Xtrain,Ytrain,cv,scoring =
["r2","neg_mean_squared_error"],show=False)
    test = reg.fit(Xtrain,Ytrain).score(Xtest,Ytest)
    rs.append(score[0])
    te.append(test)
print(axisx[rs.index(max(rs))],max(rs))
plt.figure(figsize=(20,5))
plt.plot(axisx,te,c="gray",label="XGB")
plt.plot(axisx,rs,c="green",label="XGB")
plt.legend()
plt.show()
```

虽然从图上来说，默认的0.1看起来是一个比较理想的情况，并且看起来更小的步长更利于现在的数据，但我们也无法确定对于其他数据会有怎么样的效果。所以通常，我们不调整η，即便调整，一般它也会在[0.01,0.2]之间变动。如果我们希望模型的效果更好，更多的可能是从树本身的角度来说，对树进行剪枝，而不会寄希望于调整η。

梯度提升树是XGB的基础，本节中已经介绍了XGB中与梯度提升树的过程相关的四个参数：n_estimators，learning_rate ，silent，subsample。这四个参数的主要目的，其实并不是提升模型表现，更多是了解梯度提升树的原理。现在来看，我们的梯度提升树可是说是由三个重要的部分组成：

1. 一个能够衡量集成算法效果的，能够被最优化的损失函数Obj
2. 一个能够实现预测的弱评估器$f_k(x)$
3. 一种能够让弱评估器集成的手段，包括我们讲解的迭代方法，抽样手段，样本加权等等过程

XGBoost是在梯度提升树的这三个核心要素上运行，它重新定义了损失函数和弱评估器，并且对提升算法的集成手段进行了改进，实现了运算速度和模型效果的高度平衡。并且，XGBoost将原本的梯度提升树拓展开来，让XGBoost不再是单纯的树的集成模型，也不只是单单的回归模型。只要我们调节参数，我们可以选择任何我们希望集成的算法，以及任何我们希望实现的功能。

Tsai