

菜菜的机器学习sklearn第十一期

sklearn与XGBoost



小伙伴们晚上好~o(—▽—)ゞ

我是菜菜，这里是我的sklearn课堂第十一期，本周的直播内容是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

1 在学习XGBoost之前

- 1.1 机器学习竞赛的胜利女神
- 1.2 xgboost库与XGB的sklearn API
- 1.3 XGBoost的三大板块

2 梯度提升树

- 2.1 提升集成算法：重要参数n_estimators
- 2.2 有放回随机抽样：重要参数subsample
- 2.3 迭代决策树：重要参数eta

3 XGBoost的智慧

- 3.1 选择弱评估器：重要参数booster
- 3.2 XGB的目标函数：重要参数objective
- 3.3 求解XGB的目标函数
- 3.4 参数化决策树 $f_k(x)$: 参数alpha, lambda
- 3.5 寻找最佳树结构：求解 w 与 T
- 3.6 寻找最佳分枝：结构分数之差
- 3.7 让树停止生长：重要参数gamma

4 XGBoost应用中的其他问题

- 4.1 过拟合：剪枝参数与回归模型调参
- 4.2 XGBoost模型的保存和调用
 - 4.2.1 使用Pickle保存和调用模型
 - 4.2.2 使用Joblib保存和调用模型
- 4.3 分类案例：XGB中的样本不均衡问题
- 4.4 XGBoost类中的其他参数和功能

XGBoost结语

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eXtreme Gradient Boosting，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——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

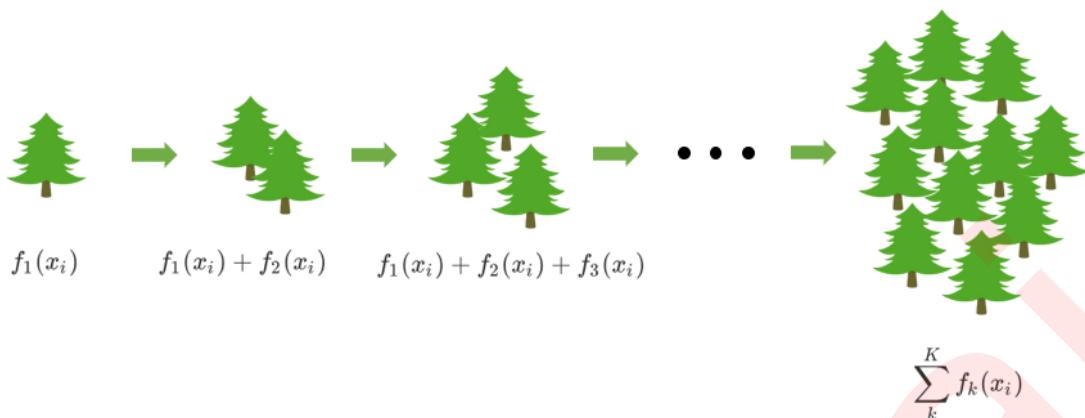
```
class xgboost.XGBRegressor(max_depth=3, learning_rate=0.1, n_estimators=100, silent=True,
objective='reg:linear', booster='gbtree', n_jobs=1, nthread=None, gamma=0, min_child_weight=1, max_delta_step=0,
subsample=1, colsample_bytree=1, colsample_bylevel=1, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,
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

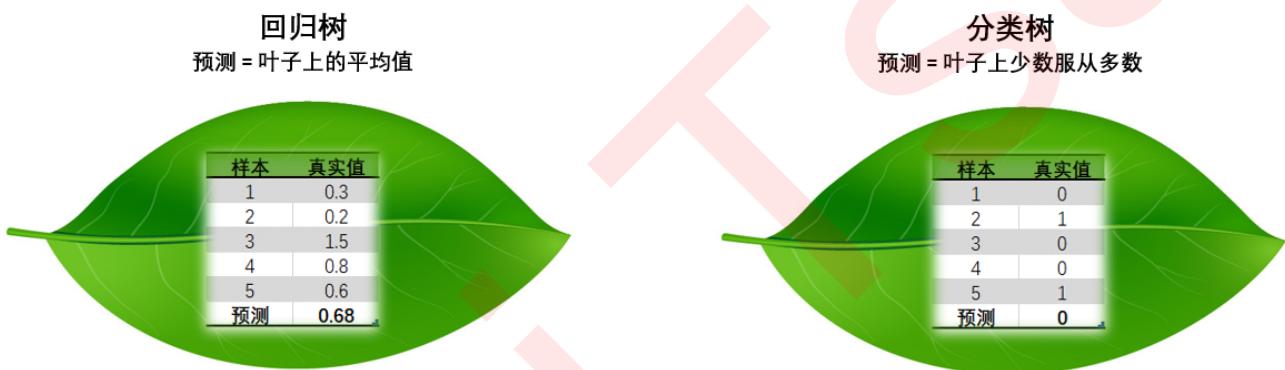
XGBoost的基础是梯度提升算法，因此我们必须先从了解梯度提升算法开始。梯度提升（Gradient boosting）是构建预测模型的最强大技术之一，它是集成算法中提升法（Boosting）的代表算法。**集成算法通过在数据上构建多个弱评估器，汇总所有弱评估器的建模结果，以获取比单个模型更好的回归或分类表现。**弱评估器被定义为是表现至少比随机猜测更好的模型，即预测准确率不低于50%的任意模型。

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()

#开启参数silent: 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

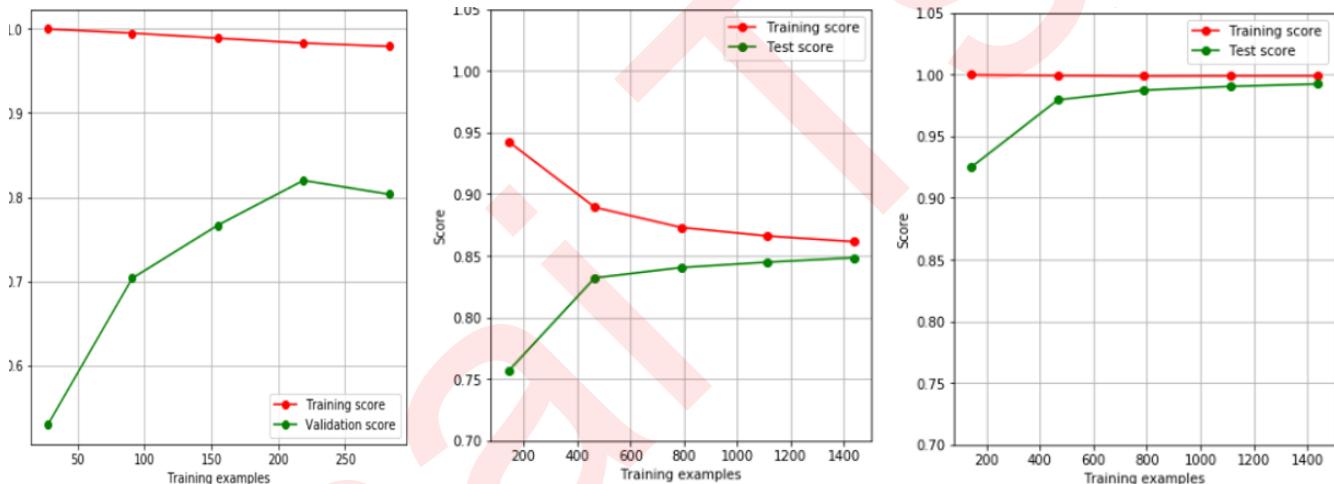
```

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 (Generalization 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(bias)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。

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

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。

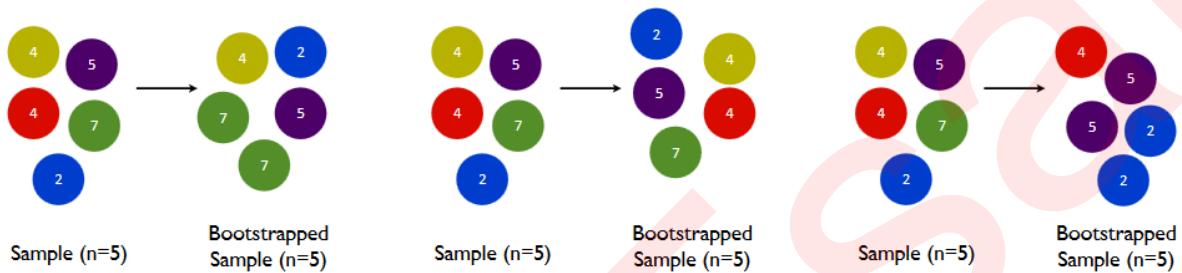
第三，树的数量提升对模型的影响有极限，最开始，模型的表现会随着XGB的树的数量一起提升，但到达某个点之后，树的数量越多，模型的效果会逐步下降，这也说明了暴力增加n_estimators不一定有效果。

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

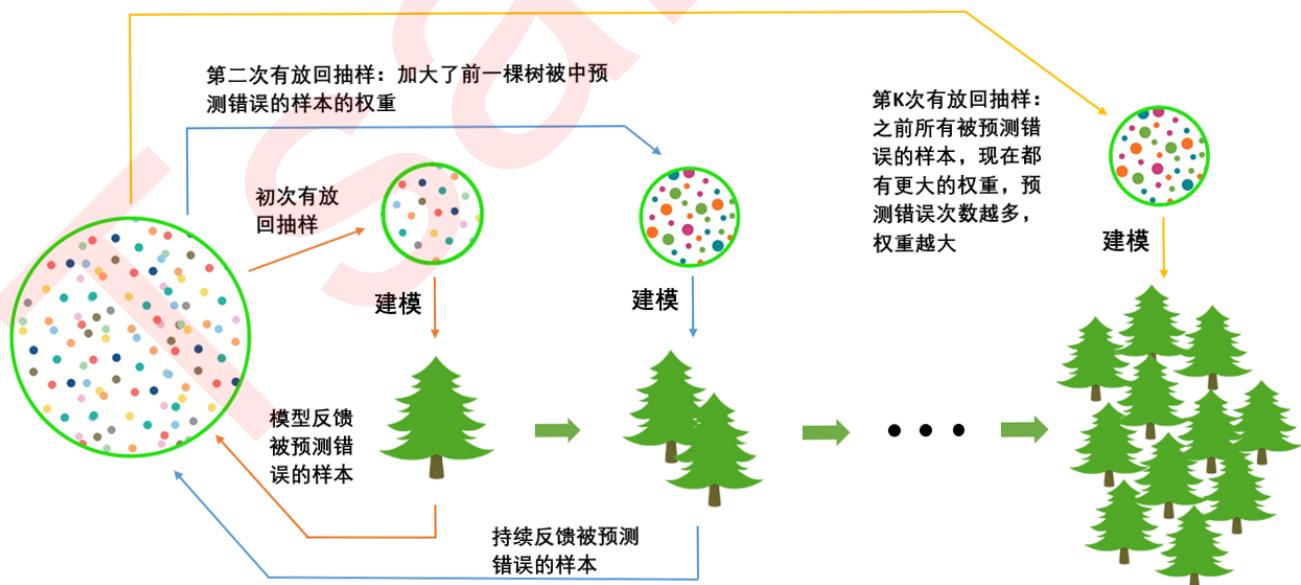
确认了有多少棵树之后，我们来思考一个问题：建立了众多的树，怎么就能够保证模型整体的效果变强呢？集成的目的是为了模型在样本上能表现出更好的效果，所以对于所有的提升集成算法，**每构建一个评估器，集成模型的效果都会比之前更好。**也就是随着迭代的进行，模型整体的效果必须要逐渐提升，最后要实现集成模型的效果最优。要实现这个目标，我们可以首先从训练数据上着手。

我们训练模型之前，必然会有个巨大的数据集。我们都应该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bootstrap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()

#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()

#继续细化学习曲线
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

#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？

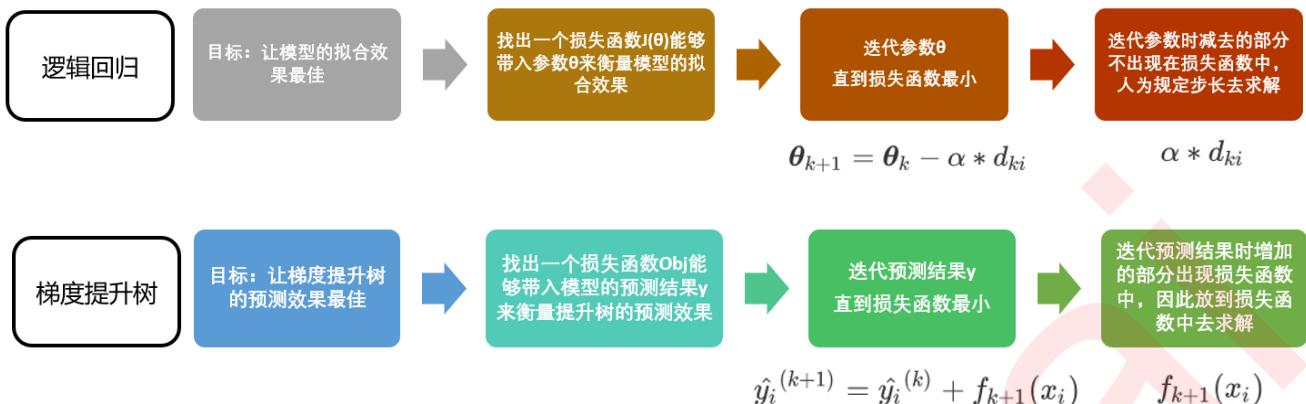
来回顾一下最优化问题的老朋友，我们的逻辑回归模型。在逻辑回归当中，我们有方程：

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

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：

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

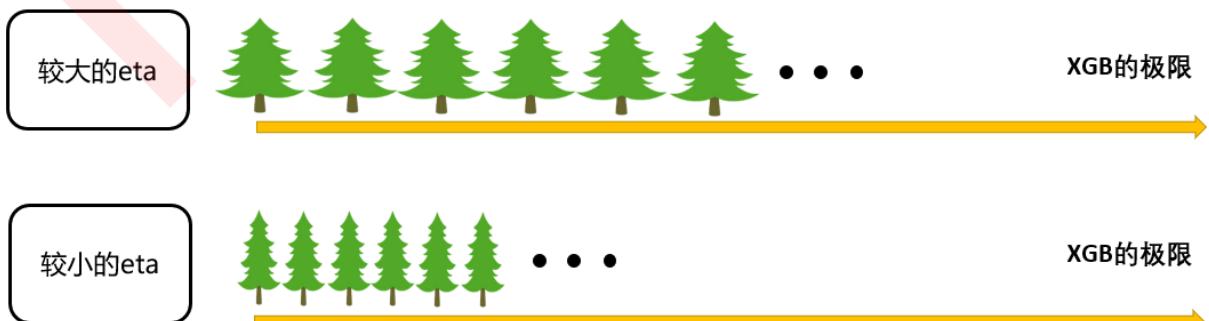
在 k 次迭代后，我们的集成算法中总共有 k 棵树，而我们前面讲明了， k 棵树的集成结果是前面所有树上的叶子权重的累加 $\sum_k^K f_k(x_i)$ 。所以我们让 k 棵树的集成结果 $\hat{y}_i^{(k)}$ 加上我们新建的树上的叶子权重 $f_{k+1}(x_i)$ ，就可以得到第 $k+1$ 次迭代后，总共 $k+1$ 棵树的预测结果 $\hat{y}_i^{(k+1)}$ 了。**我们让这个过程持续下去，直到找到能够让损失函数最小化的 \hat{y} ，这个 \hat{y} 就是我们模型的预测结果。**参数可以迭代，集成的树林也可以迭代，万事大吉！

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摸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。

3 XGBoost的智慧

```

class xgboost.XGBRegressor(kwags, 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')

```

3.1 选择弱评估器：重要参数booster

梯度提升算法中不只是梯度提升树，XGB作为梯度提升算法的进化，自然也不只有树模型一种弱评估器。在XGB中，除了树模型，我们还可以选用线性模型，比如线性回归，来进行集成。虽然主流的XGB依然是树模型，但我们也使用其他的模型。基于XGB的这种性质，我们有参数“booster”来控制我们究竟使用怎样的弱评估器。

| xgb.train() & params | xgb.XGBRegressor() |
|--|---|
| xgb_model | booster |
| 使用哪种弱评估器。可以输入gbtree, gblinear或dart。输入的评估器不同，使用的params参数也不同，每种评估器都有自己的params列表。评估器必须于param参数相匹配，否则报错。 | 使用哪种弱评估器。可以输入gbtree, gblinear或dart。gbtree代表梯度提升树，dart是Dropouts meet Multiple Additive Regression Trees，可译为抛弃提升树，在建树的过程中会抛弃一部分树，比梯度提升树有更好的防过拟合功能。输入gblinear使用线性模型。 |

两个参数都默认为"gbtree"，如果不使用树模型，则可以自行调整。当XGB使用线性模型的时候，它的许多数学过程就与使用普通的Boosting集成非常相似，因此我们在讲解中会重点来讲解使用更多，也更加核心的基于树模型的XGBoost。简单看看现有的数据集上，是什么样的弱评估器表现更好：

```
for booster in ["gbtree", "gblinear", "dart"]:
    reg = XGBR(n_estimators=180
                , learning_rate=0.1
                , random_state=420
                , booster=booster).fit(xtrain, Ytrain)
    print(booster)
    print(reg.score(xtest, Ytest))      #自己找线性数据试试看"gblinear"的效果吧~
```

3.2 XGB的目标函数：重要参数objective

梯度提升算法中都存在着损失函数。不同于逻辑回归和SVM等算法中固定的损失函数写法，**集成算法中的损失函数是可选的**，要选用什么损失函数取决于我们希望解决什么问题，以及希望使用怎样的模型。比如说，**如果我们的目标是进行回归预测，那我们可以选择调节后的均方误差RMSE作为我们的损失函数。如果我们是进行分类预测，那我们可以选择错误率error或者对数损失log_loss。**只要我们选出的函数是一个可微的，能够代表某种损失的函数，它就可以是我们XGB中的损失函数。

在众多机器学习算法中，损失函数的核心是衡量模型的泛化能力，即模型在未知数据上的预测的准确与否，我们训练模型的核心目标也是希望模型能够预测准确。在XGB中，预测准确自然是非常重要的因素，但我们之前提到过，**XGB的是实现了模型表现和运算速度的平衡的算法**。普通的损失函数，比如错误率，均方误差等，都只能够衡量模型的表现，无法衡量模型的运算速度。回忆一下，我们曾在许多模型中使用空间复杂度和时间复杂度来衡量模型的运算效率。XGB因此引入了模型复杂度来衡量算法的运算效率。因此XGB的目标函数被写作：**传统损失函数 + 模型复杂度**。

$$Obj = \sum_{i=1}^m l(y_i, \hat{y}_i) + \sum_{k=1}^K \Omega(f_k)$$

其中*i*代表数据集中的第*i*个样本，*m*表示导入第*k*棵树的数据总量，*K*代表建立的所有树(*n_estimators*)，当只建立了*t*棵树的时候，式子应当为 $\sum_{k=1}^t \Omega(f_k)$ 。第一项代表传统的损失函数，衡量真实标签 y_i 与预测值 \hat{y}_i 之间的差异，通常是RMSE，调节后的均方误差。第二项代表模型的复杂度，使用树模型的某种变换 Ω 表示，这个变化代表了一个从树的结构来衡量树模型的复杂度的式子，可以有多种定义。**注意，我们的第二项中没有特征矩阵 x_i 的介入**。我们在迭代每一棵树的过程中，都最小化 Obj 来力求获取最优的 \hat{y} ，因此我们同时最小化了模型的错误率和模型的复杂度，这种设计目标函数的方法不得不说实在是非常巧妙和聪明。

目标函数：可能的困惑

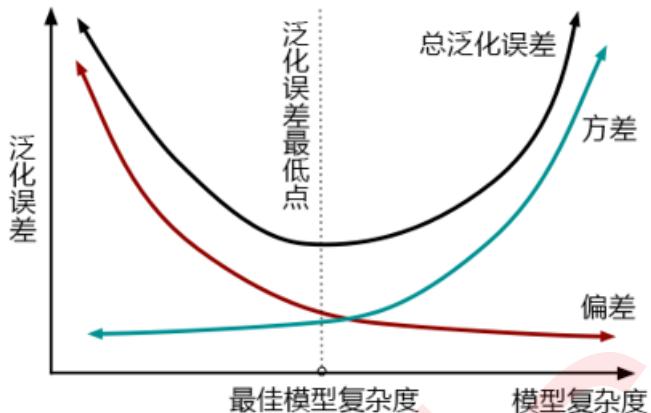
与其他算法一样，我们最小化目标函数以求模型效果最佳，并且我们可以通过限制*n_estimators*来限制我们的迭代次数，因此我们可以看到生成的每棵树所对应的目标函数取值。目标函数中的第二项看起来是与*K*棵树都相关，但我们的第一个式子看起来却只和样本量相关，仿佛只与当前迭代到的这棵树相关，这不是很奇怪么？

其实并非如此，我们的第一项传统损失函数也是与已经建好的所有树相关的，相关在这里：

$$\hat{y}_i^{(t)} = \sum_k^t f_k(x_i) = \sum_k^{t-1} f_k(x_i) + f_t(x_i)$$

我们的 \hat{y}_i 中已经包含了所有树的迭代结果，因此整个目标函数都与 K 棵树相关。

我们还可以从另一个角度去理解我们的目标函数。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（Generalization 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(bias)和噪声(ϵ)共同决定，而泛化误差越小，模型就越理想。从下面的图可以看出来，方差和偏差是此消彼长的，并且模型的复杂度越高，方差越大，偏差越小。



方差可以被简单地解释为模型在不同数据集上表现出来的稳定性，而偏差是模型预测的准确度。那方差-偏差困境就可以对应到我们的 Obj 中了：

$$Obj = \sum_{i=1}^m l(y_i, \hat{y}_i) + \sum_{k=1}^K \Omega(f_k)$$

第一项是衡量我们的偏差，模型越不准确，第一项就会越大。第二项是衡量我们的方差，模型越复杂，模型的学习就会越具体，到不同数据集上的表现就会差异巨大，方差就会越大。所以我们求解 Obj 的最小值，其实是在求解方差与偏差的平衡点，以求模型的泛化误差最小，运行速度最快。我们知道树模型和树的集成模型都是学习天才，是天生过拟合的模型，因此大多数树模型最初都会出现在图像的右上方，我们必须通过剪枝来控制模型不要过拟合。现在XGBoost的损失函数中自带限制方差变大的部分，也就是说XGBoost会比其他的树模型更加聪明，不会轻易落到图像的右上方。可见，这个模型在设计的时候的确是考虑了方方面面，难怪XGBoost会如此强大了。

在应用中，我们使用参数“objective”来确定我们目标函数的第一部分中的 $l(y_i, \hat{y}_i)$ ，也就是衡量损失的部分。

| xgb.train() | xgb.XGBRegressor() | xgb.XGBClassifier() |
|-------------------------------|--------------------------------|-------------------------------------|
| obj: 默认binary:logistic | objective: 默认reg:linear | objective: 默认binary:logistic |

常用的选择有：

| 输入 | 选用的损失函数 |
|-----------------|---------------------------------|
| reg:linear | 使用线性回归的损失函数，均方误差，回归时使用 |
| binary:logistic | 使用逻辑回归的损失函数，对数损失log_loss，二分类时使用 |
| binary:hinge | 使用支持向量机的损失函数，Hinge Loss，二分类时使用 |
| multi:softmax | 使用softmax损失函数，多分类时使用 |

我们还可以选择自定义损失函数。比如说，我们可以选择输入平方损失 $l(y_j, \hat{y}_j) = (y_j - \hat{y}_j)^2$ ，此时我们的XGBoost其实就是算法梯度提升机器（gradient boosted machine）。在xgboost中，我们被允许自定义损失函数，但通常我们还是使用类已经为我们设置好的损失函数。我们的回归类中本来使用的就是reg:linear，因此在这里无需做任何调整。**注意：分类型的目标函数导入回归类中会直接报错。**现在来试试看xgb自身的调用方式。



由于xgb中所有的参数都需要自己的输入，并且objective参数的默认值是二分类，因此我们必须手动调节。试试看在其他参数相同的情况下，我们xgboost库本身和sklearn比起来，效果如何：

```

#默认reg:linear
reg = XGBR(n_estimators=180, random_state=420).fit(Xtrain, Ytrain)
reg.score(Xtest, Ytest)
MSE(Ytest, reg.predict(Xtest))

#xgb实现法
import xgboost as xgb
#使用类Dmatrix读取数据
dtrain = xgb.DMatrix(Xtrain, Ytrain)
dtest = xgb.DMatrix(Xtest, Ytest)
#非常遗憾无法打开来看，所以通常都是先读到pandas里面查看之后再放到DMatrix中
dtrain
#写明参数，silent默认为False，通常需要手动将它关闭
param = {'silent':False, 'objective':'reg:linear', "eta":0.1}
num_round = 180
#类train，可以直接导入的参数是训练数据，树的数量，其他参数都需要通过params来导入
bst = xgb.train(param, dtrain, num_round)
#接口predict
from sklearn.metrics import r2_score
r2_score(Ytest, bst.predict(dtest))
MSE(Ytest, bst.predict(dtest))

```

看得出来，无论是从 R^2 还是从MSE的角度来看，都是xgb库本身表现更优秀，这也许是由于底层的代码是由不同团队创造的缘故。随着样本量的逐渐上升，sklearnAPI中调用的结果与xgboost中直接训练的结果会比较相似，如果希望的话可以分别训练，然后选取泛化误差较小的库。如果可以的话，建议脱离sklearnAPI直接调用xgboost库，因为xgboost库本身的调参要方便许多。

3.3 求解XGB的目标函数

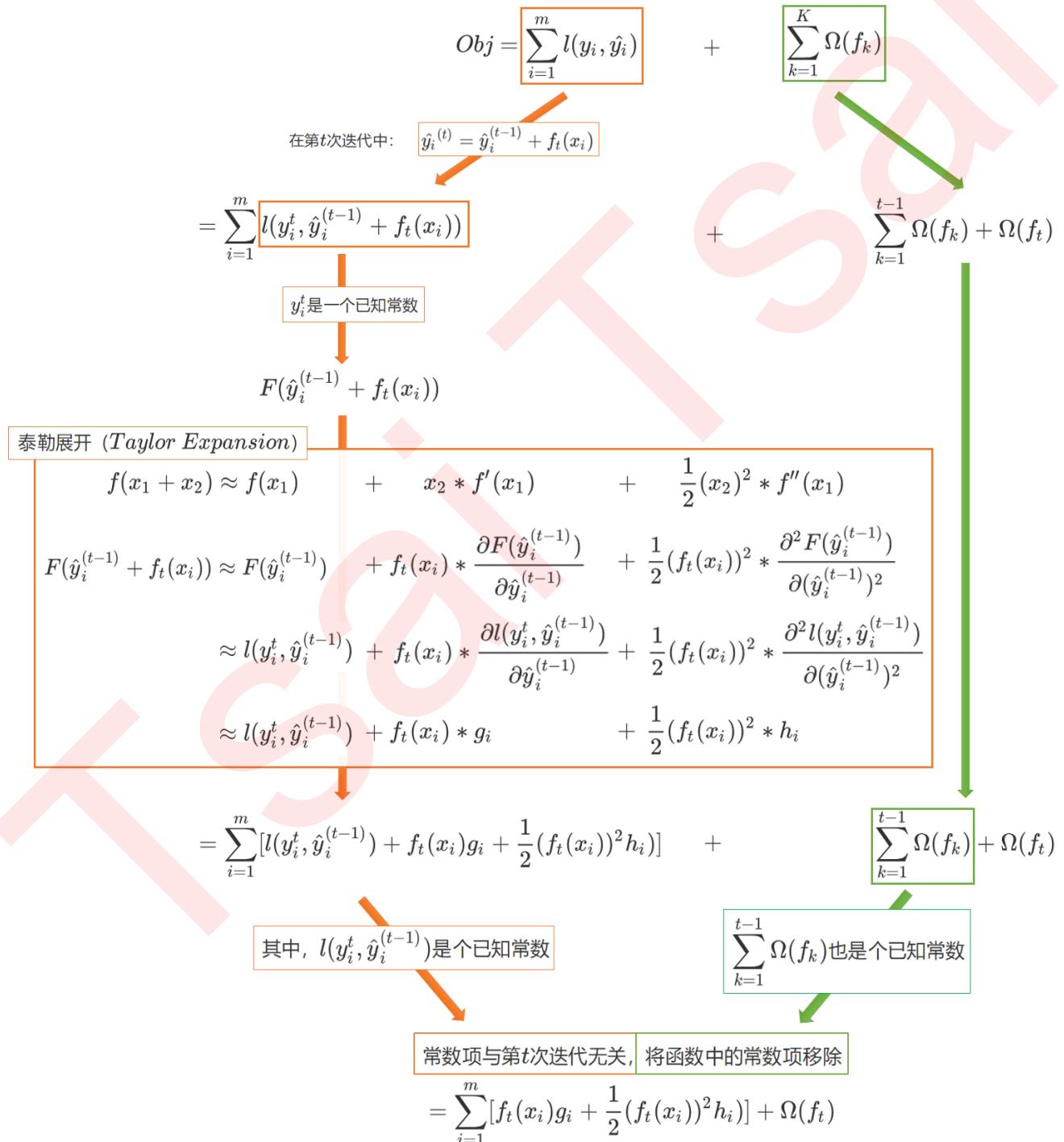
有了如下目标函数，我们来看看如何求解它。时刻记得我们求解目标函数的目的：**为了求得在第t次迭代中最优的树 f_t 。**在逻辑回归和支持向量机中，我们通常先将目标函数转化成一种容易求解的方式（比如对偶），然后使用梯度下降或者SMO之类的数学方法来执行我们的最优化过程。之前我们使用了逻辑回归的迭代过程来帮助大家理解在梯度提升树中树是如何迭代的，那我们是否可以使用逻辑回归的参数求解方式来求解XGB的目标函数呢？

很遗憾，在XGB中我们无法使用梯度下降，原因是XGB的损失函数没有需要求解的参数。我们在传统梯度下降中迭代的是参数，而我们在XGB中迭代的是树，树 f_k 不是数字组成的向量，并且其结构不受到特征矩阵 x 取值大小的直接影响，尽管这个迭代过程可以被类比到梯度下降上，但真实的求解过程却是完全不同。

在求解XGB的目标函数的过程中，我们考虑的是如何能够将目标函数转化成更简单的，与树的结构直接相关的写法，以此来建立树的结构与模型的效果（包括泛化能力与运行速度）之间的直接联系。也因为这种联系的存在，XGB的目标函数又被称为“结构分数”。

$$\hat{y}_i^{(t)} = \sum_k^t f_k(x_i) = \sum_k^{t-1} f_k(x_i) + f_t(x_i) \\ = \hat{y}_i^{(t-1)} + f_t(x_i)$$

首先，我们先来进行第一步转换。



其中 g_i 和 h_i 分别是在损失函数 $l(y_i^t, \hat{y}_i^{(t-1)})$ 上对 $\hat{y}_i^{(t-1)}$ 所求的一阶导数和二阶导数，他们被统称为**每个样本的梯度统计量 (gradient statistic)**。在许多算法的解法推导中，我们求解导数都是为了让一阶导数等于0来求解极值，而现在我们求解导数只是为了配合泰勒展开中的形式，仅仅是简化公式的目的罢了。所以**GBDT和XGB的区别之中，GBDT求一阶导数，XGB求二阶导数，这两个过程根本是不可类比的。XGB在求解极值为目标的求导中也是求解一阶导数**，后面会为大家详解。

可能的困惑：泰勒展开好像不是长这样？

如果大家去单独查看泰勒展开的数学公式，你们会看到的大概是这样：

$$f(x) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}(x - c) + \frac{f''(c)}{2!}(x - c)^2 + \frac{f'''(c)}{3!}(x - c)^3 + \dots$$

其中 $f'(c)$ 表示 $f(x)$ 上对 x 求导后，令 x 的值等于 c 所取得的值。**其中有假设： c 与 x 非常接近， $(x - c)$ 非常接近0**，于是我们可以将式子改写成：

$$f(x + x - c) \approx \frac{f(c)}{0!} + \frac{f'(c)}{1!}(x - c) + \frac{f''(c)}{2!}(x - c)^2 + \frac{f'''(c)}{3!}(x - c)^3 + \dots$$

只取前两项，取约等于：

$$\begin{aligned} &\approx \frac{f(c)}{0!} + \frac{f'(c)}{1!}(x - c) + \frac{f''(c)}{2!}(x - c)^2 \\ &\approx f(c) + f'(c)(x - c) + \frac{f''(c)}{2}(x - c)^2 \end{aligned}$$

令： $x_1 = x, x_2 = x - c$ ：

$$\begin{aligned} f(x_1 + x_2) &\approx f(x_1) + f'(x_1) * x_2 + \frac{f''(x_1)}{2} * x_2^2 \\ F(\hat{y}_i^{(t-1)} + f_t(x_i)) &\approx F(\hat{y}_i^{(t-1)}) + g_i * f_t(x_i) + \frac{h_i}{2} (f_t(x_i))^2 \end{aligned}$$

如刚才所说， $x - c$ 需要很小，与 x 相比起来越小越好，那在我们的式子中，需要很小的这部分就是 $f_t(x_i)$ 。这其实很好理解，对于一个集成算法来说，每次增加的一棵树对模型的影响其实非常小，尤其是当我们有许多树的时候——比如， $n_estimators=500$ 的情况下， $f_t(x_i)$ 与 x 相比总是非常小的，因此这个条件可以被满足，泰勒展开可以被使用。如此，我们的目标函数可以被顺利转化成：

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

这个式子中， g_i 和 h_i 只与传统损失函数相关，核心的部分是我们需要决定的树 f_t 。接下来，我们就来研究一下我们的 f_t 。

3.4 参数化决策树 $f_k(x)$ ：参数alpha, lambda

```
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',
```

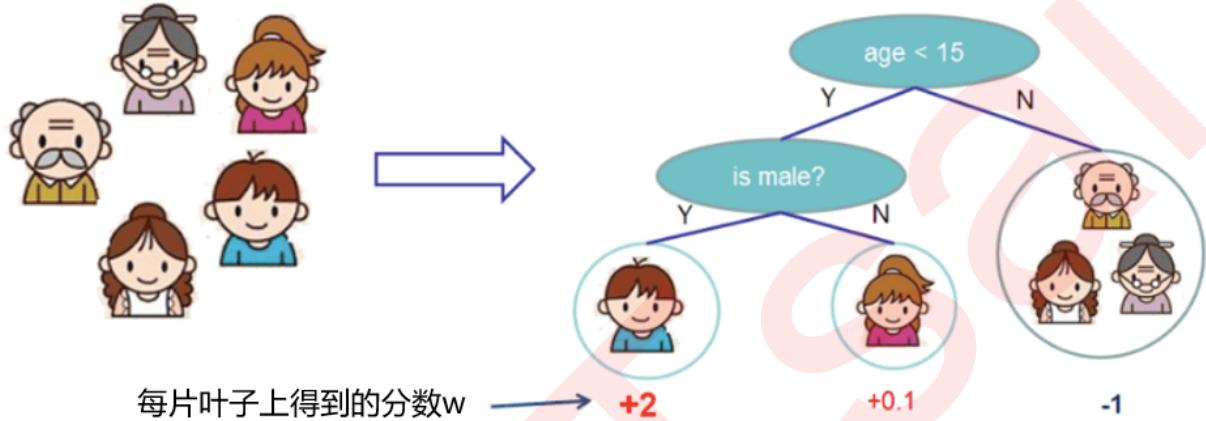
`**kwargs)*`

在参数化决策树之前，我们先来简单复习一下回归树的原理。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。对于回归树，通常来说每个叶子节点上的预测值是这个叶子节点上所有样本的标签的均值。但值得注意的是，XGB作为普通回归树的改进算法，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。

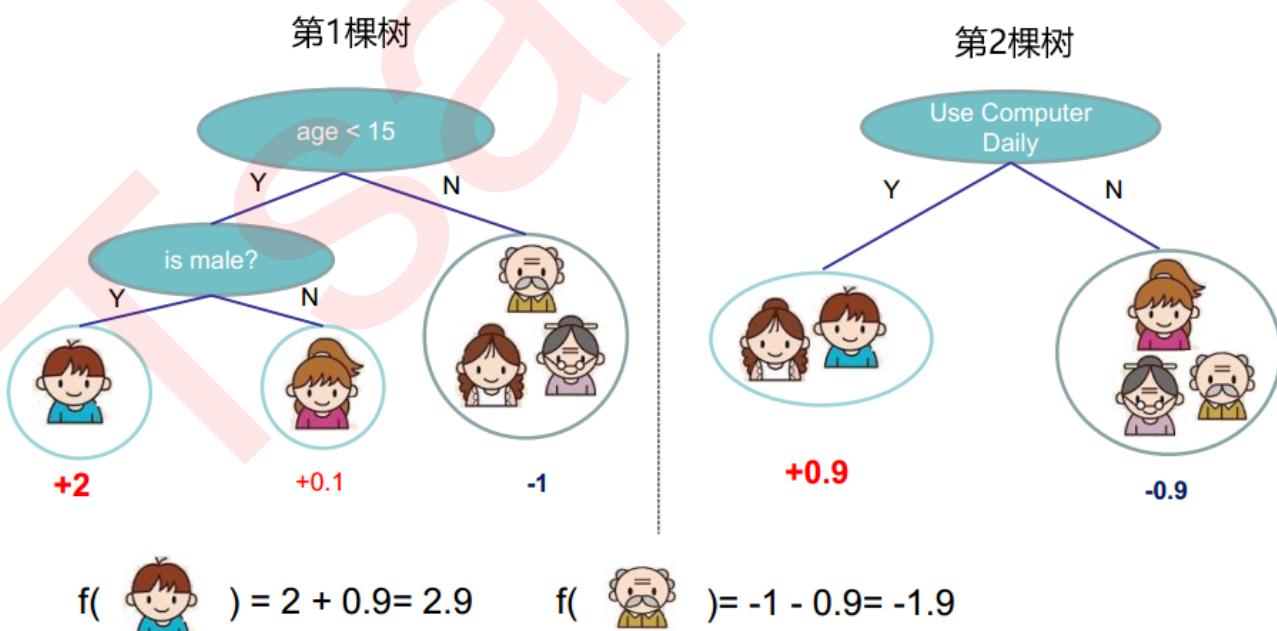
输入特征：年龄，职业，每天使用电脑的时间

这个人有多喜欢电脑游戏呢？



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：

$$\hat{y}_i^{(k)} = \sum_k f_k(x_i)$$



基于这个理解，我们来考虑每一棵树。对每一棵树，它都有自己独特的结构，这个结构即是指叶子节点的数量，树的深度，叶子的位置等等所形成的一个可以定义唯一模型的树结构。在这个结构中，我们使用 $q(x_i)$ 表示样本 x_i 所在的叶子节点，并且使用 $w_{q(x_i)}$ 来表示这个样本落到第*t*棵树上的第 $q(x_i)$ 个叶子节点中所获得的分数，于是有：

$$f_t(x_i) = w_{q(x_i)}$$

这是对于每一个样本而言的叶子权重，然而在一个叶子节点上的所有样本所对应的叶子权重是相同的。设一棵树上总共包含了 T 个叶子节点，其中每个叶子节点的索引为 j ，则这个叶子节点上的样本权重是 w_j 。依据这个，我们定义模型的复杂度 $\Omega(f)$ 为（注意这不是唯一可能的定义，我们当然还可以使用其他的定义，只要满足叶子越多/深度越大，复杂度越大的理论，我们可以自己决定我们的 $\Omega(f)$ 要是一个怎样的式子）：

$$\Omega(f) = \gamma T + \text{正则项 (Regularization)}$$

如果使用 $L2$ 正则项：

$$\begin{aligned} &= \gamma T + \frac{1}{2} \lambda ||w||^2 \\ &= \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \end{aligned}$$

如果使用 $L1$ 正则项：

$$\begin{aligned} &= \gamma T + \frac{1}{2} \alpha |w| \\ &= \gamma T + \frac{1}{2} \alpha \sum_{j=1}^T |w_j| \end{aligned}$$

还可以两个一起使用：

$$= \gamma T + \frac{1}{2} \alpha \sum_{j=1}^T |w_j| + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$

这个结构中有两部分内容，一部分是控制树结构的 γT ，另一部分则是我们的正则项。叶子数量 T 可以代表整个树结构，这是因为在XGBoost中所有的树都是CART树（二叉树），所以我们可以根据叶子的数量 T 判断出树的深度，而 γ 是我们自定的控制叶子数量的参数。

至于第二部分正则项，类比一下我们岭回归和Lasso的结构，参数 α 和 λ 的作用其实非常容易理解，他们都是控制正则化强度的参数，我们可以二选一使用，也可以一起使用加大正则化的力度。当 λ 和 α 都为0的时候，目标函数就是普通的梯度提升树的目标函数。

XGB vs GBDT 核心区别2：正则项的存在

在普通的梯度提升树GBDT中，我们是不在目标函数中使用正则项的。但XGB借用正则项来修正树模型天生容易过拟合这个缺陷，在剪枝之前让模型能够尽量不过拟合。

来看正则化系数分别对应的参数：

| 参数含义 | xgb.train() | xgb.XGBRegressor() |
|--------------------|----------------------------------|--------------------------------------|
| L1正则项的参数 α | alpha， 默认0， 取值范围 $[0, +\infty]$ | reg_alpha， 默认0， 取值范围 $[0, +\infty]$ |
| L2正则项的参数 λ | lambda， 默认1， 取值范围 $[0, +\infty]$ | reg_lambda， 默认1， 取值范围 $[0, +\infty]$ |

根据我们以往的经验，我们往往认为两种正则化达到的效果是相似的，只不过细节不同。比如在逻辑回归当中，两种正则化都会压缩 θ 参数的大小，只不过L1正则化会让 θ 为0，而L2正则化不会。在XGB中也是如此。当 λ 和 α 越大，惩罚越重，正则项所占的比例就越大，在尽全力最小化目标函数的最优化方向下，叶子节点数量就会被压制，模型的复杂度就越来越低，所以对于天生过拟合的XGB来说，正则化可以一定程度上提升模型效果。

对于两种正则化如何选择的问题，从XGB的默认参数来看，我们优先选择的是L2正则化。当然，如果想尝试L1也不是不可。两种正则项还可以交互，因此这两个参数的使用其实比较复杂。在实际应用中，正则化参数往往不是我们调参的最优选择，如果真的希望控制模型复杂度，我们会调整 γ 而不是调整这两个正则化参数，因此大家不必过于在意这两个参数最终如何影响了我们的模型效果。对于树模型来说，还是剪枝参数地位更高更优先。大家只需要理解这两个参数从数学层面上如何影响我们的模型就足够了。如果我们希望调整 λ 和 α ，我们往往会使用网格搜索来帮助我们。在这里，为大家贴出网格搜索的代码和结果供大家分析和学习。

```
#使用网格搜索来查找最佳的参数组合
from sklearn.model_selection import GridSearchCV

param = {"reg_alpha":np.arange(0,5,0.05), "reg_lambda":np.arange(0,2,0.05)}

gscv = GridSearchCV(reg,param_grid = param,scoring = "neg_mean_squared_error",cv=cv)

#=====【TIME WARNING: 10~20 mins】=====
time0=time()
gscv.fit(Xtrain,Ytrain)
print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S.%f"))

gscv.best_params_
gscv.best_score_

preds = gscv.predict(Xtest)

from sklearn.metrics import r2_score, mean_squared_error as MSE
r2_score(Ytest,preds)

MSE(Ytest,preds)

#网格搜索的结果有什么样的含义呢？为什么会出现这样的结果？你相信网格搜索得出的结果吗？试着用数学和你对XGB的理解来解释一下吧。
```

3.5 寻找最佳树结构：求解 w 与 T

在上一节中，我们定义了树和树的复杂度的表达式，树我们使用叶子节点上的预测分数来表达，而树的复杂度则是叶子数目加上正则项：

$$f_t(x_i) = w_{q(x_i)}, \quad \Omega(f_t) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$

假设现在第 t 棵树的结构已经被确定为 q , 我们可以将树的结构带入我们的损失函数, 来继续转化我们的目标函数。转化目标函数的目的是: 建立树的结构 (叶子节点的数量) 与目标函数的大小之间的直接联系, 以求出在第 t 次迭代中需要求解的最优的树 f_t 。注意, 我们假设我们使用的是L2正则化 (这也是参数lambda和alpha的默认设置, lambda为1, alpha为0), 因此接下来的推导也会根据L2正则化来进行。

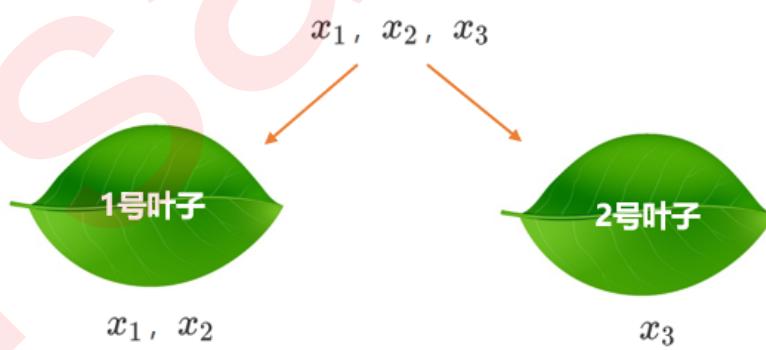
$$\begin{aligned} & \sum_{i=1}^m [f_t(x_i)g_i + \frac{1}{2}(f_t(x_i))^2 h_i] + \Omega(f_t) \\ &= \sum_{i=1}^m [w_{q(x_i)} g_i + \frac{1}{2} w_{q(x_i)}^2 h_i] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \\ &= \boxed{\sum_{i=1}^m w_{q(x_i)} g_i} + \boxed{\sum_{i=1}^m \frac{1}{2} w_{q(x_i)}^2 h_i} + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \end{aligned}$$

其实, 每片叶子上的 w_j 是一致的
唯一不同的是每个样本所对应的 g_i

所有的样本必然会被分到 T 片叶子节点中的任一个节点上
我们定义索引为 j 的叶子上含有的样本的集合是 I_j

$$\begin{aligned} & \sum_{j=1}^T (w_j * \sum_{i \in I_j} g_i) + \frac{1}{2} \sum_{j=1}^T (w_j^2 * \sum_{i \in I_j} h_i) + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \\ &= \sum_{j=1}^T \left[w_j \sum_{i \in I_j} g_i + \frac{1}{2} w_j^2 (\sum_{i \in I_j} h_i + \lambda) \right] + \gamma T \end{aligned}$$

橙色框中的转化是如何实现的?



$$\begin{aligned} & w_{q(x_1)} * g_1 \\ & w_{q(x_2)} * g_2 \end{aligned}$$

$$w_{q(x_3)} * g_3$$

$$w_{q(x_1)} = w_{q(x_2)} = w_1$$

$$w_{q(x_3)} = w_2$$

于是我们可以有:

$$\begin{aligned}
\sum_{i=1}^m w_{q(x_i)} * g_i &= w_{q(x_1)} * g_1 + w_{q(x_2)} * g_2 + w_{q(x_3)} * g_3 \\
&= w_1(g_1 + g_2) + w_2 * g_3 \\
&= \sum_{j=1}^T (w_j \sum_{i \in I_j} g_i)
\end{aligned}$$

如此就实现了这个转化。

对于最终的式子，我们定义：

$$G_j = \sum_{i \in I_j} g_i, \quad H_j = \sum_{i \in I_j} h_i$$

于是可以有：

$$\begin{aligned}
Obj^{(t)} &= \sum_{j=1}^T \left[w_j G_j + \frac{1}{2} w_j^2 (H_j + \lambda) \right] + \gamma T \\
F^*(w_j) &= w_j G_j + \frac{1}{2} w_j^2 (H_j + \lambda)
\end{aligned}$$

其中每个 j 取值下都是一个以 w_j 为自变量的二次函数 F^* ，我们的目标是追求让 Obj 最小，只要单独的每一个叶子 j 取值下的二次函数都最小，那他们的加和必然也会最小。于是，我们在 F^* 上对 w_j 求导，让一阶导数等于0以求极值，可得：

$$\begin{aligned}
\frac{\partial F^*(w_j)}{\partial w_j} &= G_j + w_j(H_j + \lambda) \\
0 &= G_j + w_j(H_j + \lambda) \\
w_j &= -\frac{G_j}{H_j + \lambda}
\end{aligned}$$

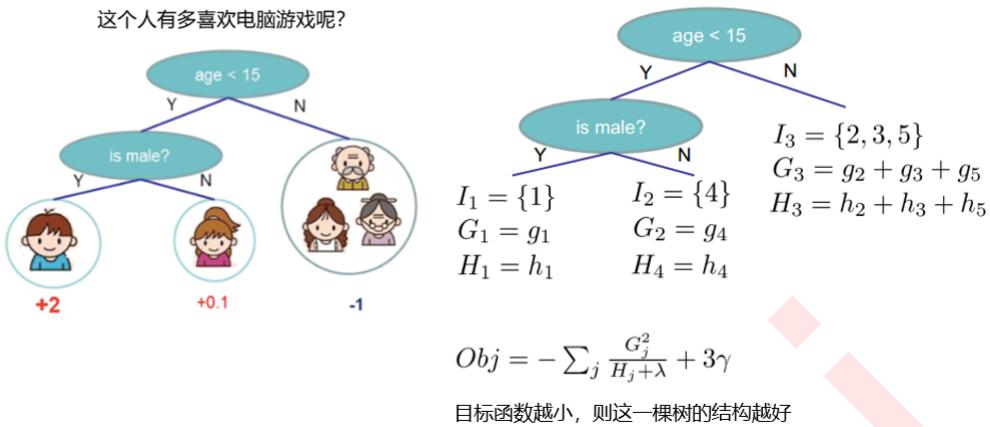
把这个公式带入目标函数，则有：

$$\begin{aligned}
Obj^{(t)} &= \sum_{j=1}^T \left[-\frac{G_j}{H_j + \lambda} * G_j + \frac{1}{2} \left(-\frac{G_j}{H_j + \lambda} \right)^2 (H_j + \lambda) \right] + \gamma T \\
&= \sum_{j=1}^T \left[-\frac{G_j^2}{H_j + \lambda} + \frac{1}{2} * \frac{G_j^2}{H_j + \lambda} \right] + \gamma T \\
&= -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T
\end{aligned}$$

到了这里，大家可能已经注意到了，比起最初的损失函数 + 复杂度的样子，我们的目标函数已经发生了巨大变化。我们的样本量 i 已经被归结到了每个叶子当中去，我们的目标函数是基于每个叶子节点，也就是树的结构来计算。所以，我们的目标函数又叫做“结构分数”（structure score），分数越低，树整体的结构越好。如此，我们就建立了树的结构（叶子）和模型效果的直接联系。

更具体一些，我们来看一个例子：

| 索引 | 样本 | 梯度统计量 |
|----|----|------------|
| 1 | | g_1, h_1 |
| 2 | | g_2, h_2 |
| 3 | | g_3, h_3 |
| 4 | | g_4, h_4 |
| 5 | | g_5, h_5 |



$$Obj = - \left(\frac{g_1^2}{h_1 + \lambda} + \frac{g_4^2}{h_4 + \lambda} + \frac{(g_2 + g_3 + g_5)^2}{h_2 + h_3 + h_5 + \lambda} \right) + 3\gamma$$

所以在XGB的运行过程中，我们会根据 Obj 的表达式直接探索最好的树结构，也就是说找寻最佳的树。从式子中可以看出， λ 和 γ 是我们设定好的超参数， G_j 和 H_j 是由损失函数和这个特定结构下树的预测结果 $\hat{y}_i^{(t-1)}$ 共同决定，而 T 只由我们的树结构决定。则我们通过最小化 Obj 所求解出的其实是 T ，叶子的数量。所以本质也就是求解树的结构了。

在这个算式下，我们可以有一种思路，那就是枚举所有可能的树结构 q ，然后一个个计算我们的 Obj ，待我们选定了最佳的树结构（最佳的 T ）之后，我们使用这种树结构下计算出来的 G_j 和 H_j 就可以求解出每个叶子上的权重 w_j ，如此就找到我们的最佳树结构，完成了这次迭代。

可能的困惑：求解 w_j 的一些细节

这种解法可能会让大家有一些困惑，让我们来看一看：

- 用 w 求解 w ？

一个大家可能会感到困惑的点是， G_j 和 H_j 的本质其实是损失函数上的一阶导数 g_i 和二阶导数 h_i 之和，而一阶和二阶导数本质是：

$$g_i = \frac{\partial l(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}}, \quad h_i = \frac{\partial^2 l(y_i, \hat{y}_i^{(t-1)})}{\partial (\hat{y}_i^{(t-1)})^2}$$

y_i 是已知的标签，但我们有预测值的求解公式：

$$\begin{aligned}\hat{y}_i^{(k)} &= \sum_k^K f_k(x_i) \\ &= \sum_k^K w_{q(x_i)}\end{aligned}$$

这其实是指， G_j 和 H_j 的计算中带有 w ，那先确定最佳的 T ，再求出 G_j 和 H_j ，结合 λ 求出叶子权重 w_j 的思路不觉得有些问题么？仿佛在带入 w 求解 w ？对于有这样困惑的大家，请注意我们的 $\hat{y}_i^{(t-1)}$ 与我们现在要求解的 w_j 其实不是在同一棵树上的。别忘记我们是在一直迭代的，我们现在求解的 w_j 是第 t 棵树上的结果，而 $\hat{y}_i^{(t-1)}$ 是前面的 $(t-1)$ 棵树的累积 w ，是在前面所有的迭代中已经求解出来的已知的部分。

- 求解第一棵树时，没有“前面已经迭代完毕的部分”，怎么办？

那第二个问题又来了：那我们如何求解第一棵树在样本*i*下的预测值 $\hat{y}_i^{(1)}$ 呢？在建立第一棵树时，并不存在任何前面的迭代已经计算出来的信息，但根据公式我们需要使用如下式子来求解 $f_1(x_i)$ ，并且我们在求解过程中还需要对前面所有树的结果进行求导。

$$\hat{y}_i^{(1)} = \hat{y}_i^{(0)} + f_1(x_i)$$

这时候，我们假设 $\hat{y}_i^{(0)} = 0$ 来解决我们的问题，事实是，由于前面没有已经测出来的树的结果，整个集成算法的结果现在也的确为0。

- 第0棵树的预测值假设为0，那求解第一棵树的 g_i 和 h_i 的过程是在对0求导？

这个问题其实很简单。在进行求导时，所有的求导过程都和之前推导的过程相一致，之所以能够这么做，是因为我们其实不是在对0求导，而是对一个变量 $\hat{y}_i^{(t-1)}$ 求导。只是除了求导之外，我们还需要在求导后的结果中带入这个变量此时此刻的取值，而这个取值在求解第一棵树时刚好等于0而已。更具体地，可以看看下面，对0求导，和对变量求导后，变量取值为0的区别：

$$\text{对常数 } 0 \text{ 进行求导: } \frac{\partial(x^2 + x)}{\partial(0)} = 0$$

$$\text{对变量 } x \text{ 进行求导, 但变量 } x \text{ 等于 } 0: \frac{\partial(x^2 + x)}{\partial(x)} = 2x + 1 = 2 * 0 + 1 = 1$$

这些细节都理解了之后，相信大家对于先求解 Obj 的最小值来求解树结构 T ，然后以此为基础求解出 w_j 的过程已经没有什么问题了。回忆一下我们刚才说的，为了找出最小的 Obj ，我们需要枚举所有可能的树结构，这似乎又回到了我们最初的困惑——我们之所以要使用迭代和最优化的方式，就是因为我们不希望进行枚举，这样即浪费资源又浪费时间。那我们怎么办呢？来看下一节：贪婪算法寻找最佳结构。

3.6 寻找最佳分枝：结构分数之差

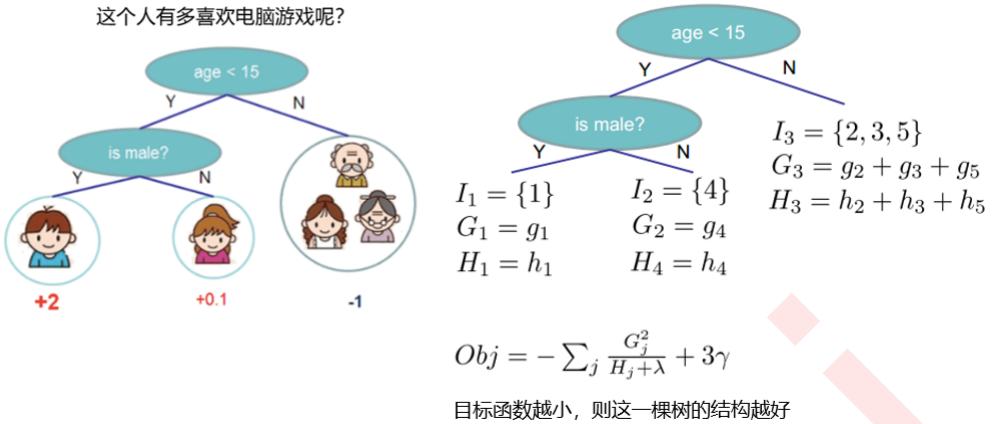
贪婪算法指的是控制局部最优来达到全局最优的算法，决策树算法本身就是一种使用贪婪算法的方法。XGB作为树的集成模型，自然也想到采用这样的方法来进行计算，所以我们认为，如果每片叶子都是最优，则整体生成的树结构就是最优，如此就可以避免去枚举所有可能的树结构。



回忆一下决策树中我们是如何进行计算：我们使用基尼系数或信息熵来衡量分枝之后叶子节点的不纯度，分枝前的信息熵与分枝后的信息熵之差叫做信息增益，信息增益最大的特征上的分枝就被我们选中，当信息增益低于某个阈值时，就让树停止生长。在XGB中，我们使用的方式是类似的：我们首先使用目标函数来衡量树的结构的优劣，然后让树从深度0开始生长，每进行一次分枝，我们就计算目标函数减少了多少，当目标函数的降低低于我们设定的某个阈值时，就让树停止生长。

来个具体的例子，在这张图中，我们有中间节点“是否是男性”，这个中间节点下面有两个叶子节点，分别是样本弟弟和妹妹。我们来看看这个分枝点上，树的结构分数之差如何表示。

| 索引 | 样本 | 梯度统计量 |
|----|----|------------|
| 1 | | g_1, h_1 |
| 2 | | g_2, h_2 |
| 3 | | g_3, h_3 |
| 4 | | g_4, h_4 |
| 5 | | g_5, h_5 |



对于中间节点这个叶子节点而言，我们的 $T = 1$ ，则这个节点上的结构分为：

$$\begin{aligned} I &= \{1, 4\} \\ G &= g_1 + g_4 \\ H &= h_1 + h_4 \\ Score_{middle} &= -\frac{1}{2} \frac{G^2}{H + \lambda} + \gamma \end{aligned}$$

对于弟弟和妹妹节点而言，则有：

$$\begin{aligned} Score_{sis} &= -\frac{1}{2} \frac{g_4^2}{h_4 + \lambda} + \gamma \\ Score_{bro} &= -\frac{1}{2} \frac{g_1^2}{h_1 + \lambda} + \gamma \end{aligned}$$

则分枝后的结构分数之差为：

$$\begin{aligned} Gain &= Score_{sis} + Score_{bro} - Score_{middle} \\ &= -\frac{1}{2} \frac{g_4^2}{h_4 + \lambda} + \gamma - \frac{1}{2} \frac{g_1^2}{h_1 + \lambda} + \gamma - \left(-\frac{1}{2} \frac{G^2}{H + \lambda} + \gamma \right) \\ &= -\frac{1}{2} \frac{g_4^2}{h_4 + \lambda} + \gamma - \frac{1}{2} \frac{g_1^2}{h_1 + \lambda} + \gamma + \frac{1}{2} \frac{G^2}{H + \lambda} - \gamma \\ &= -\frac{1}{2} \left[\frac{g_4^2}{h_4 + \lambda} + \frac{g_1^2}{h_1 + \lambda} - \frac{G^2}{H + \lambda} \right] + \gamma \\ &= -\frac{1}{2} \left[\frac{g_4^2}{h_4 + \lambda} + \frac{g_1^2}{h_1 + \lambda} - \frac{(g_1 + g_4)^2}{(h_1 + h_4) + \lambda} \right] + \gamma \end{aligned}$$

将负号去除：

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

CART树全部是二叉树，因此这个式子是可以推广的。从这个式子我们可以总结出，其实分枝后的结构分数之差为：

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

其中 G_L 和 H_L 从左节点（弟弟节点）上计算得出， G_R 和 H_R 从右节点（妹妹节点）上计算得出，而 $(G_L + G_R)$ 和 $(H_L + H_R)$ 从中间节点上计算得出。对于任意分枝，我们都可以这样来进行计算。**在现实中，我们会对所有特征的所有分枝点进行如上计算，然后选出让目标函数下降最快的节点来进行分枝。**对每一棵树的每一层，我们都进行这样的计算，比起原始的梯度下降，实践证明这样的求解最佳树结构的方法运算更快，并且在大型数据下也能够表现不错。至此，我们作为XGBoost的使用者，已经将需要理解的XGB的原理理解完毕了。

3.7 让树停止生长：重要参数gamma

在之前所有的推导过程中，我们都没有提到 γ 这个变量。从目标函数和结构分数之差Gain的式子中来看， γ 是我们每增加一片叶子就会被剪去的惩罚项。增加的叶子越多，结构分数之差Gain会被惩罚越重，所以 γ 又被称之为是“复杂性控制”（complexity control），所以 γ 是我们用来防止过拟合的重要参数。**实践证明， γ 是对梯度提升树影响最大的参数之一，其效果丝毫不逊色于n_estimators和防止过拟合的神器max_depth。同时， γ 还是我们让树停止生长的重要参数。**

在逻辑回归中，我们使用参数 tol 来设定阈值，并规定如果梯度下降时损失函数减小量小于 tol 下降就会停止。**在XGB中，我们规定，只要结构分数之差Gain是大于0的，即只要目标函数还能够继续减小，我们就允许树继续进行分枝。**也就是说，我们对于目标函数减小量的要求是：

$$\begin{aligned} \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma > 0 \\ \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] > \gamma \end{aligned}$$

如此，我们可以直接通过设定 γ 的大小来让XGB中的树停止生长。 γ 因此被定义为，在**树的叶节点上进行进一步分枝所需的最小目标函数减少量**，在决策树和随机森林中也有类似的参数（min_split_loss, min_samples_split）。 **γ 越大，算法就越保守，树的叶子数量就越少，模型的复杂度就越低。**

| 参数含义 | xgb.train() | xgb.XGBRegressor() |
|------------------|-------------------------------|-------------------------------|
| 复杂度的惩罚项 γ | gamma，默认0，取值范围 $[0, +\infty]$ | gamma，默认0，取值范围 $[0, +\infty]$ |

如果我们希望从代码中来观察 γ 的作用，使用sklearn中传统的学习曲线等工具就比较困难了。来看下面这段代码，这是一段让参数 γ 在0~5之间均匀取值的学习曲线。其运行速度较缓慢并且曲线的效果匪夷所思，大家若感兴趣可以自己运行一下。

```
#=====【TIME WARNING: 1 min】=====
axisx = np.arange(0, 5, 0.05)
rs = []
var = []
ge = []
for i in axisx:
    reg = XGBR(n_estimators=180, random_state=420, gamma=i)
    result = CVS(reg, Xtrain, Ytrain, cv=cv)
    rs.append(result.mean())
    var.append(result.var())
    ge.append((1 - result.mean())**2 + result.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)*0.1
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()

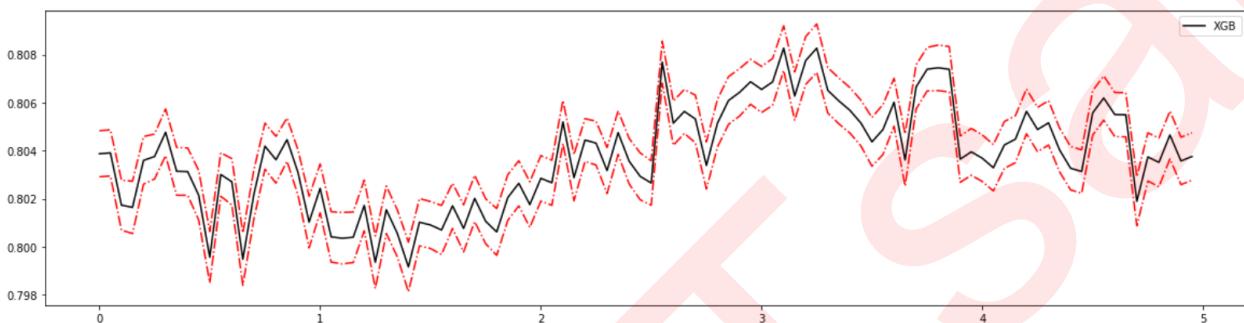
```

以上这段代码的运行结果如下：

```

3.1 0.8082740708121546 0.009289472054282844
2.5500000000000003 0.8076923943232783 0.00871821372693272
2.5500000000000003 0.8076923943232783 0.00871821372693272 0.04570042892804619

```



可以看到，我们完全无法从中看出什么趋势，偏差时高时低，方差时大时小，参数 γ 引起的波动远远超过其他参数（其他参数至少还有一个先上升再平稳的过程，而 γ 则是仿佛完全无规律）。在sklearn下XGBoost太不稳定，如果这样来调整参数的话，效果就很难保证。因此，为了调整 γ ，我们需要来引入新的工具，xgboost库中的类xgboost.cv。

```

xgboost.cv (params, dtrain, num_boost_round=10, nfold=3, stratified=False, folds=None, metrics=(), obj=None,
feval=None, maximize=False, early_stopping_rounds=None, fpreproc=None, as_pandas=True, verbose_eval=None,
show_stdv=True, seed=0, callbacks=None, shuffle=True)

```

```

import xgboost as xgb

#为了便捷，使用全数据
dfull = xgb.DMatrix(X,y)

#设定参数
param1 = {'silent':True,'obj':'reg:linear','gamma':0}
num_round = 180
n_fold=5

#使用类xgb.cv
time0 = time()
cvresult1 = xgb.cv(param1, dfull, num_round,n_fold)
print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S.%f"))

#看看类xgb.cv生成了什么结果？
cvresult1

plt.figure(figsize=(20,5))
plt.grid()

```

```
plt.plot(range(1,181),cvresult1.iloc[:,0],c="red",label="train,gamma=0")
plt.plot(range(1,181),cvresult1.iloc[:,2],c="orange",label="test,gamma=0")
plt.legend()
plt.show()
```

#xgboost中回归模型的默认模型评估指标是什么？

为了使用xgboost.cv，我们必须要熟悉xgboost自带的模型评估指标。xgboost在建库的时候本着大而全的目标，和sklearn类似，包括了大约20个模型评估指标，然而用于回归和分类的其实只有几个，大部分是用于一些更加高级的功能比如ranking。来看用于回归和分类的评估指标都有哪些：

| 指标 | 含义 |
|----------|------------------|
| rmse | 回归用，调整后的均方误差 |
| mae | 回归用，绝对平均误差 |
| logloss | 二分类用，对数损失 |
| mlogloss | 多分类用，对数损失 |
| error | 分类用，分类误差，等于1-准确率 |
| auc | 分类用，AUC面积 |

```
param1 = {'silent':True,'obj':'reg:linear','gamma':0,"eval_metric":"mae"}
cvresult1 = xgb.cv(param1, dfull, num_round,n_fold)

plt.figure(figsize=(20,5))
plt.grid()
plt.plot(range(1,181),cvresult1.iloc[:,0],c="red",label="train,gamma=0")
plt.plot(range(1,181),cvresult1.iloc[:,2],c="orange",label="test,gamma=0")
plt.legend()
plt.show()
```

#从这个图中，我们可以看出什么?
#怎样从图中观察模型的泛化能力?
#从这个图的角度来说，模型的调参目标是什么?

来看看如果我们调整 γ ，会发生怎样的变化：

```
param1 = {'silent':True,'obj':'reg:linear','gamma':0}
param2 = {'silent':True,'obj':'reg:linear','gamma':20}
num_round = 180
n_fold=5

time0 = time()
cvresult1 = xgb.cv(param1, dfull, num_round,n_fold)
print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S:%f"))

time0 = time()
cvresult2 = xgb.cv(param2, dfull, num_round,n_fold)
```

```

print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S:%f"))

plt.figure(figsize=(20,5))
plt.grid()
plt.plot(range(1,181),cvresult1.iloc[:,0],c="red",label="train,gamma=0")
plt.plot(range(1,181),cvresult1.iloc[:,2],c="orange",label="test,gamma=0")
plt.plot(range(1,181),cvresult2.iloc[:,0],c="green",label="train,gamma=20")
plt.plot(range(1,181),cvresult2.iloc[:,2],c="blue",label="test,gamma=20")
plt.legend()
plt.show()

```

#从这里，你看出了gamma是如何控制过拟合了吗？

试一个分类的例子：

```

from sklearn.datasets import load_breast_cancer
data2 = load_breast_cancer()

x2 = data2.data
y2 = data2.target

dfull2 = xgb.DMatrix(x2,y2)

param1 = {'silent':True,'obj':'binary:logistic','gamma':0,"nfold":5}
param2 = {'silent':True,'obj':'binary:logistic','gamma':2,"nfold":5}
num_round = 100

time0 = time()
cvresult1 = xgb.cv(param1, dfull2, num_round,metrics=("error"))
print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S:%f"))

time0 = time()
cvresult2 = xgb.cv(param2, dfull2, num_round,metrics=("error"))
print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S:%f"))

plt.figure(figsize=(20,5))
plt.grid()
plt.plot(range(1,101),cvresult1.iloc[:,0],c="red",label="train,gamma=0")
plt.plot(range(1,101),cvresult1.iloc[:,2],c="orange",label="test,gamma=0")
plt.plot(range(1,101),cvresult2.iloc[:,0],c="green",label="train,gamma=2")
plt.plot(range(1,101),cvresult2.iloc[:,2],c="blue",label="test,gamma=2")
plt.legend()
plt.show()

```

有了xgboost.cv这个工具，我们的参数调整就容易多了。这个工具可以让我们直接看到参数如何影响了模型的泛化能力。接下来，我们将重点讲解如何使用xgboost.cv这个类进行参数调整。到这里，所有关于XGBoost目标函数的原理就讲解完毕了，这个目标函数及这个目标函数所衍生出来的各种数学过程是XGB原理的重中之重，大部分XGB中基于原理的参数都集中在这个模块之中，到这里大家应该已经基本掌握。

4 XGBoost应用中的其他问题

4.1 过拟合：剪枝参数与回归模型调参

```
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)
```

作为天生过拟合的模型，XGBoost应用的核心之一就是减轻过拟合带来的影响。作为树模型，减轻过拟合的方式主要是靠对决策树剪枝来降低模型的复杂度，以求降低方差。在之前的讲解中，我们已经学习了好几个可以用来防止过拟合的参数，包括上一节提到的复杂度控制 γ ，正则化的两个参数 λ 和 α ，控制迭代速度的参数 η 以及管理每次迭代前进的随机有放回抽样的参数subsample。所有的这些参数都可以用来减轻过拟合。但除此之外，我们还有几个影响重大的，专用于剪枝的参数：

| 参数含义 | xgb.train() | xgb.XGBRegressor() |
|--|-----------------------|-----------------------|
| 树的最大深度 | max_depth，默认6 | max_depth，默认6 |
| 每次生成树时随机抽样特征的比例 | colsample_bytree，默认1 | colsample_bytree，默认1 |
| 每次生成树的一层时随机抽样特征的比例 | colsample_bylevel，默认1 | colsample_bylevel，默认1 |
| 每次生成一个叶子节点时随机抽样特征的比例 | colsample_bynode，默认1 | N.A. |
| 一个叶子节点上所需要的最小 h_i 即叶子节点上的二阶导数之和类似于样本权重 | min_child_weight，默认1 | min_child_weight，默认1 |

这些参数中，树的最大深度是决策树中的剪枝法宝，算是最常用的剪枝参数，不过在XGBoost中，最大深度的功能与参数 γ 相似，因此如果先调节了 γ ，则最大深度可能无法展示出巨大的效果。当然，如果先调整了最大深度，则 γ 也有可能无法显示明显的效果。通常来说，这两个参数中我们只使用一个，不过两个都试试也没有坏处。

三个随机抽样特征的参数中，前两个比较常用。在建立树时对特征进行抽样其实是决策树和随机森林中比较常见的一种方法，但是在XGBoost之前，这种方法并没有被使用到boosting算法当中过。Boosting算法一直以抽取样本（横向抽样）来调整模型过拟合的程度，而实践证明其实纵向抽样（抽取特征）更能够防止过拟合。

参数min_child_weight不太常用，它是一片叶子上的二阶导数 h_i 之和，当样本所对应的二阶导数很小时，比如说为0.01，min_child_weight若设定为1，则说明一片叶子上至少需要100个样本。本质上来说，这个参数其实是在控制叶子上所需的最小样本量，因此对于样本量很大的数据会比较有效。如果样本量很小（比如我们现在使用的波士顿房价数据集，则这个参数效用不大）。就剪枝的效果来说，这个参数的功能也被 γ 替代了一部分，通常来说我们会试试看这个参数，但这个参数不是我的优先选择。

通常当我们获得了一个数据集后，我们先使用网格搜索找出比较合适的n_estimators和eta组合，然后使用gamma或者max_depth观察模型处于什么样的状态（过拟合还是欠拟合，处于方差-偏差图像的左边还是右边？），最后再决定是否要进行剪枝。通常来说，对于XGB模型，大多数时候都是需要剪枝的。接下来我们就来看看使用xgb.cv这个类来进行剪枝调参，以调整出一组泛化能力很强的参数。

让我们先从最原始的，设定默认参数开始，先观察一下默认参数下，我们的交叉验证曲线长什么样：

```
dfull = xgb.DMatrix(x,y)

param1 = {'silent':True #并非默认
          , 'obj':'reg:linear' #并非默认
          , "subsample":1
          , "max_depth":6
          , "eta":0.3
          , "gamma":0
          , "lambda":1
          , "alpha":0
          , "colsample_bytree":1
          , "colsample_bylevel":1
          , "colsample_bynode":1
          , "nfold":5}

num_round = 200

time0 = time()
cvresult1 = xgb.cv(param1, dfull, num_round)
print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S:%f"))

fig,ax = plt.subplots(1,figsize=(15,10))
#ax.set_ylim(top=5)
ax.grid()
ax.plot(range(1,201),cvresult1.iloc[:,0],c="red",label="train,original")
ax.plot(range(1,201),cvresult1.iloc[:,2],c="orange",label="test,original")
ax.legend(fontsize="xx-large")
plt.show()
```

从曲线上可以看出，模型现在处于过拟合的状态。我们决定要进行剪枝。我们的目标是：训练集和测试集的结果尽量接近，**如果测试集上的结果不能上升，那训练集上的结果降下来也是不错的选择**（让模型不那么具体到训练数据，增加泛化能力）。在这里，我们要使用三组曲线。一组用于展示原始数据上的结果，一组用于展示上一个参数调节完毕后的结果，最后一组用于展示现在我们在调节的参数的结果。具体怎样使用，我们来看：

```
param1 = {'silent':True
          , 'obj':'reg:linear'
          , "subsample":1
          , "max_depth":6
          , "eta":0.3
          , "gamma":0
          , "lambda":1
          , "alpha":0
          , "colsample_bytree":1
          , "colsample_bylevel":1
          , "colsample_bynode":1
          , "nfold":5}

num_round = 200

time0 = time()
cvresult1 = xgb.cv(param1, dfull, num_round)
print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S:%f"))
```

```

fig,ax = plt.subplots(1,figsize=(15,8))
ax.set_ylim(top=5)
ax.grid()
ax.plot(range(1,201),cvresult1.iloc[:,0],c="red",label="train,original")
ax.plot(range(1,201),cvresult1.iloc[:,2],c="orange",label="test,original")

param2 = {'silent':True
          , 'obj':'reg:linear'
          , "nfold":5}
param3 = {'silent':True
          , 'obj':'reg:linear'
          , "nfold":5}

time0 = time()
cvresult2 = xgb.cv(param2, dfull, num_round)
print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S:%f"))

time0 = time()
cvresult3 = xgb.cv(param3, dfull, num_round)
print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S:%f"))

ax.plot(range(1,201),cvresult2.iloc[:,0],c="green",label="train,last")
ax.plot(range(1,201),cvresult2.iloc[:,2],c="blue",label="test,last")
ax.plot(range(1,201),cvresult3.iloc[:,0],c="gray",label="train,this")
ax.plot(range(1,201),cvresult3.iloc[:,2],c="pink",label="test,this")
ax.legend(fontsize="xx-large")
plt.show()

```

在这里，为大家提供我调出来的结果，供大家参考：

```

#默认设置
param1 = {'silent':True
          , 'obj':'reg:linear'
          , "subsample":1
          , "max_depth":6
          , "eta":0.3
          , "gamma":0
          , "lambda":1
          , "alpha":0
          , "colsample_bytree":1
          , "colsample_bylevel":1
          , "colsample_bynode":1
          , "nfold":5}
num_round = 200

time0 = time()
cvresult1 = xgb.cv(param1, dfull, num_round)
print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S:%f"))

fig,ax = plt.subplots(1,figsize=(15,8))
ax.set_ylim(top=5)
ax.grid()

```

```

ax.plot(range(1,201),cvresult1.iloc[:,0],c="red",label="train,original")
ax.plot(range(1,201),cvresult1.iloc[:,2],c="orange",label="test,original")

#调参结果1
param2 = {'silent':True
          , 'obj':'reg:linear'
          , "subsample":1
          , "eta":0.05
          , "gamma":20
          , "lambda":3.5
          , "alpha":0.2
          , "max_depth":4
          , "colsample_bytree":0.4
          , "colsample_bylevel":0.6
          , "colsample_bynode":1
          , "nfold":5}

#调参结果2
param3 = {'silent':True
          , 'obj':'reg:linear'
          , "max_depth":2
          , "eta":0.05
          , "gamma":0
          , "lambda":1
          , "alpha":0
          , "colsample_bytree":1
          , "colsample_bylevel":0.4
          , "colsample_bynode":1
          , "nfold":5}

time0 = time()
cvresult2 = xgb.cv(param2, dfull, num_round)
print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S:%f"))

ax.plot(range(1,201),cvresult2.iloc[:,0],c="green",label="train,final")
ax.plot(range(1,201),cvresult2.iloc[:,2],c="blue",label="test,final")
ax.legend(fontsize="xx-large")
plt.show()

```

在这个调整过程中，大家可能会有几个问题：

1. 一个个参数调整太麻烦，可不可以使用网格搜索呢？

当然可以！只要电脑有足够的计算资源，并且你信任网格搜索，那任何时候我们都可以使用网格搜索。只是使用的时候要注意，首先XGB的参数非常多，参数可取的范围也很广，究竟是使用np.linspace或者np.arange作为参数的备选值也会影响结果，而且网格搜索的运行速度往往不容乐观，因此建议至少先使用xgboost.cv来确认参数的范围，否则很可能花很长的时间做了无用功。

并且，在使用网格搜索的时候，最好不要一次性将所有的参数都放入进行搜索，最多一次两三个。有一些互相影响的参数需要放在一起使用，比如学习率eta和树的数量n_estimators。

另外，如果网格搜索的结果与你的理解相违背，与你手动调参的结果相违背，选择模型效果较好的一个。如果两者效果差不多，那选择相信手动调参的结果。网格毕竟是枚举出结果，很多时候得出的结果可能会是具体到数据的巧合，我们无法去一一解释网格搜索得出的结论为何是这样。如果你感觉都无法解释，那就不要在意，直接选择结果较好的一个。

2. 调参的时候参数的顺序会影响调参结果吗？

会影响，因此在现实中，我们会优先调整那些对模型影响巨大的参数。在这里，我建议的剪枝上的调参顺序是：n_estimators与eta共同调节，gamma或者max_depth，采样和抽样参数（纵向抽样影响更大），最后才是正则化的两个参数。当然，可以根据自己的需求来进行调整。

3. 调参之后测试集上的效果还没有原始设定上的效果好怎么办？

如果调参之后，交叉验证曲线确实显示测试集和训练集上的模型评估效果是更加接近的，推荐使用调参之后的效果。我们希望增强模型的泛化能力，然而泛化能力的增强并不代表着在新数据集上模型的结果一定优秀，因为未知数据集并非一定符合全数据的分布，在一组未知数据上表现十分优秀，也不一定能够在其他的未知数据集上表现优秀。因此不必过于纠结在现有的测试集上是否表现优秀。当然了，在现有数据上如果能够实现训练集和测试集都非常优秀，那模型的泛化能力自然也会是很强的。

自己找一个数据集，剪枝试试看吧。

4.2 XGBoost模型的保存和调用

在使用Python进行编程时，我们可能会需要编写较为复杂的程序或者建立复杂的模型。比如XGBoost模型，这个模型的参数复杂繁多，并且调参过程不是太容易，一旦训练完毕，我们往往希望将训练完毕后的模型保存下来，以便日后用于新的数据集。在Python中，保存模型的方法有许多种。我们以XGBoost为例，来讲解两种主要的模型保存和调用方法。

4.2.1 使用Pickle保存和调用模型

pickle是python编程中比较标准的一个保存和调用模型的库，我们可以使用pickle和open函数的连用，来将我们的模型保存到本地。以刚才我们已经调整好的参数和训练好的模型为例，我们可以这样来使用pickle：

```
import pickle

dtrain = xgb.DMatrix(Xtrain, Ytrain)

#设定参数，对模型进行训练
param = {'silent':True
          , 'obj':'reg:linear'
          , "subsample":1
          , "eta":0.05
          , "gamma":20
          , "lambda":3.5
          , "alpha":0.2
          , "max_depth":4
          , "colsample_bytree":0.4
          , "colsample_bylevel":0.6
          , "colsample_bynode":1}

num_round = 180
```

```

bst = xgb.train(param, dtrain, num_round)

#保存模型
pickle.dump(bst, open("xgboostonboston.dat", "wb"))
#注意, open中我们往往使用w或者r作为读取的模式, 但其实w与r只能用于文本文件, 当我们希望导入的不是文本文件, 而是模型本身的时候, 我们使用"wb"和"rb"作为读取的模式。其中wb表示以二进制写入, rb表示以二进制读入

#看看模型被保存到了哪里?
import sys
sys.path

#重新打开jupyter lab

from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split as TTS
from sklearn.metrics import mean_squared_error as MSE
import pickle
import xgboost as xgb

data = load_boston()

X = data.data
y = data.target

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

#注意, 如果我们保存的模型是xgboost库中建立的模型, 则导入的数据类型也必须是xgboost库中的数据类型
dtest = xgb.DMatrix(Xtest,Ytest)

#导入模型
loaded_model = pickle.load(open("xgboostonboston.dat", "rb"))
print("Loaded model from: xgboostonboston.dat")

#做预测
ypreds = loaded_model.predict(dtest)

from sklearn.metrics import mean_squared_error as MSE, r2_score
MSE(Ytest,ypreds)

r2_score(Ytest,ypreds)

```

4.2.2 使用Joblib保存和调用模型

Joblib是SciPy生态系统中的一部分, 它为Python提供保存和调用管道和对象的功能, 处理NumPy结构的数据尤其高效, 对于很大的数据集和巨大的模型非常有用。Joblib与pickle API非常相似, 来看看代码:

```

bst = xgb.train(param, dtrain, num_round)

import joblib

```

```

#同样可以看看模型被保存到了哪里
joblib.dump(bst,"xgboost-boston.dat")

loaded_model = joblib.load("xgboost-boston.dat")

ypreds = loaded_model.predict(dtest)

MSE(Ytest, ypreds)

r2_score(Ytest,ypreds)

#使用sklearn中的模型
from xgboost import XGBRegressor as XGBR

bst = XGBR(n_estimators=200
            ,eta=0.05, gamma=20
            ,reg_lambda=3.5
            ,reg_alpha=0.2
            ,max_depth=4
            ,colsample_bytree=0.4
            ,colsample_bylevel=0.6).fit(xtrain,Ytrain)

joblib.dump(bst,"xgboost-boston.dat")
loaded_model = joblib.load("xgboost-boston.dat")

#则这里可以直接导入Xtest
ypreds = loaded_model.predict(Xtest)

MSE(Ytest, ypreds)

```

在这两种保存方法下，我们都可以找到保存下来的dat文件，将这些文件移动到任意计算机上的python下的环境变量路径中（使用sys.path进行查看），则可以使用import来对模型进行调用。注意，模型的保存调用与自写函数的保存调用是两回事，大家要注意区分。

4.3 分类案例：XGB中的样本不均衡问题

在之前的学习中，我们一直以回归作为演示的例子，这是由于回归是XGB的常用领域的缘故。然而作为机器学习中的大头，分类算法也是不可忽视的，XGB作为分类的例子自然也是非常多。存在分类，就会存在样本不平衡问题带来的影响，XGB中存在着调节样本不平衡的参数**scale_pos_weight**，这个参数非常类似于之前随机森林和支持向量机中我们都使用到过的**class_weight**参数，通常我们在参数中输入的是负样本量与正样本量之比 $\frac{\text{sum}(\text{negative instances})}{\text{sum}(\text{positive instances})}$ 。

| 参数含义 | xgb.train() | xgb.XGBClassifier() |
|------------------------------------|-----------------------|-----------------------|
| 控制正负样本比例，表示为负/正样本比例 在样本不平衡问题中使用 | scale_pos_weight, 默认1 | scale_pos_weight, 默认1 |

来看看如何使用这个参数吧。

1. 导库，创建样本不均衡的数据集

```

import numpy as np
import xgboost as xgb
import matplotlib.pyplot as plt
from xgboost import XGBClassifier as XGBC
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split as TTS
from sklearn.metrics import confusion_matrix as cm, recall_score as recall, roc_auc_score as auc

class_1 = 500 #类别1有500个样本
class_2 = 50 #类别2只有50个
centers = [[0.0, 0.0], [2.0, 2.0]] #设定两个类别的中心
clusters_std = [1.5, 0.5] #设定两个类别的方差, 通常来说, 样本量比较大的类别会更加松散
X, y = make_blobs(n_samples=[class_1, class_2],
                   centers=centers,
                   cluster_std=clusters_std,
                   random_state=0, shuffle=False)

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

(y == 1).sum() / y.shape[0]

```

2. 在数据集上建模：sklearn模式

```

#在sklearn下建模#
clf = XGBC().fit(Xtrain,Ytrain)
y whole pred = clf.predict(Xtest)
clf.score(Xtest,Ytest)

cm(Ytest,y whole pred,labels=[1,0])

recall(Ytest,y whole pred)

auc(Ytest,clf.predict_proba(Xtest)[:,1])

#负/正样本比例
clf_ = XGBC(scale_pos_weight=10).fit(Xtrain,Ytrain)
y whole pred_ = clf_.predict(Xtest)
clf_.score(Xtest,Ytest)

cm(Ytest,y whole pred_,labels=[1,0])

recall(Ytest,y whole pred_)

auc(Ytest,clf_.predict_proba(Xtest)[:,1])

#随着样本权重逐渐增加, 模型的recall, auc和准确率如何变化?
for i in [1,5,10,20,30]:
    clf_ = XGBC(scale_pos_weight=i).fit(Xtrain,Ytrain)

```

```

ypred_ = clf_.predict(xtest)
print(i)
print("\tAccuracy:{}".format(clf_.score(xtest,Ytest)))
print("\tRecall:{}".format(recall(Ytest,ypred_)))
print("\tAUC:{}".format(auc(Ytest,clf_.predict_proba(xtest)[:,1])))

```

3. 在数据集上建模：xgboost模式

```

dtrain = xgb.DMatrix(Xtrain,Ytrain)
dtest = xgb.DMatrix(Xtest,Ytest)

#看看xgboost库自带的predict接口
param= {'silent':True,'objective':'binary:logistic',"eta":0.1,"scale_pos_weight":1}
num_round = 100

bst = xgb.train(param, dtrain, num_round)

preds = bst.predict(dtest)

#看看preds返回了什么?
preds

#自己设定阈值
ypred = preds.copy()
ypred[preds > 0.5] = 1
ypred[ypred != 1] = 0

#写明参数
scale_pos_weight = [1,5,10]
names = ["negative vs positive: 1",
         "negative vs positive: 5",
         "negative vs positive: 10"]

#导入模型评估指标
from sklearn.metrics import accuracy_score as accuracy, recall_score as recall,
roc_auc_score as auc

for name,i in zip(names,scale_pos_weight):
    param= {'silent':True,'objective':'binary:logistic',
            "eta":0.1,"scale_pos_weight":i}
    clf = xgb.train(param, dtrain, num_round)
    preds = clf.predict(dtest)
    ypred = preds.copy()
    ypred[preds > 0.5] = 1
    ypred[ypred != 1] = 0
    print(name)
    print("\tAccuracy:{}".format(accuracy(Ytest,ypred)))
    print("\tRecall:{}".format(recall(Ytest,ypred)))
    print("\tAUC:{}".format(auc(Ytest,preds)))

#当然我们也可以尝试不同的阈值
for name,i in zip(names,scale_pos_weight):
    for thres in [0.3,0.5,0.7,0.9]:

```

```

param= {'silent':True,'objective':'binary:logistic'
        , "eta":0.1,"scale_pos_weight":i}
clf = xgb.train(param, dtrain, num_round)
preds = clf.predict(dtest)
yprob = preds.copy()
yprob[preds > thres] = 1
yprob[yprob != 1] = 0
print("{} , thresholds:{}".format(name,thres))
print("\tAccuracy:{}".format(accuracy(Ytest,yprob)))
print("\tRecall:{}".format(recall(Ytest,yprob)))
print("\tAUC:{}".format(auc(Ytest,preds)))

```

可以看出，在xgboost库和sklearnAPI中，参数scale_pos_weight都非常有效。本质上来说，scale_pos_weight参数是通过调节预测的概率值来调节，大家可以通过查看bst.predict(Xtest)返回的结果来观察概率受到了怎样的影响。因此，当我们只关心预测出的结果是否准确，AUC面积或者召回率是否足够好，我们就可以使用scale_pos_weight参数来帮助我们。然而xgboost除了可以做分类和回归，还有其他的多种功能，在一些需要使用精确概率的领域（比如排序ranking），我们希望能够保持概率原有的模样，而提升模型的效果。这种时候，我们就无法使用scale_pos_weight来帮助我们。来看看xgboost官网是怎么说明这个问题的：

Handle Imbalanced Dataset

For common cases such as ads clickthrough log, the dataset is extremely imbalanced. This can affect the training of XGBoost model, and there are two ways to improve it.

- If you care only about the overall performance metric (AUC) of your prediction
 - Balance the positive and negative weights via `scale_pos_weight`
 - Use AUC for evaluation
- If you care about predicting the right probability
 - In such a case, you cannot re-balance the dataset
 - Set parameter `max_delta_step` to a finite number (say 1) to help convergence

官网上说，如果我们只在意模型的整体表现，则使用AUC作为模型评估指标，使用scale_pos_weight来处理样本不平衡问题，如果我们在意预测出正确的概率，**那我们就无法通过调节scale_pos_weight来减轻样本不平衡问题带来的影响。**

这种时候，我们需要考虑另一个参数：`max_delta_step`。

这个参数非常难以理解，它被称之为是“树的权重估计中允许的单次最大增量”，既可以考虑成是影响 w_j 的估计的参数。xgboost官网上认为，如果我们在处理样本不均衡问题，并且十分在意得到正确的预测概率，则可以设置max_delta_step参数为一个有限的数（比如1）来帮助收敛。max_delta_step参数通常不进行使用，二分类下的样本不均衡问题时这个参数唯一的用途。

4.4 XGBoost类中的其他参数和功能

到目前为止，我们已经讲解了XGBoost类中的大部分参数和功能。这些参数和功能主要覆盖了XGBoost中的梯度提升树的原理以及XGBoost自身所带的一些特性。还有一些其他的参数和用法，是算法实际应用时需要考虑的问题。接下来，我们就来看看这些参数。

```
class xgboost.XGBRegressor
```

| 参数 | 参数含义 | 集成算法 | 弱评估器 | 其他过程 |
|-------------------|--------------------------------------|------|------|------|
| n_estimators | 集成算法中的弱分类器的数量 | √ | | |
| learning_rate | 集成中的学习率 | √ | | |
| silent | 是否在运行集成时进行流程的打印 | √ | | |
| subsample | 从样本中进行采样的比例 | √ | | |
| max_depth | 弱分类器的最大树深度 | | √ | |
| objective | 指定学习目标函数与学习任务 | | √ | |
| booster | 指定要使用的弱分类器 | | √ | |
| gamma | 在树的叶节点上进行进一步分枝所需的小的目标函数的下降 | | √ | |
| min_child_weight | 一个叶节点上所需的最小样本权重(hessian) | | √ | |
| max_delta_step | 树的权重估计中允许的单次最大增量 | | √ | |
| colsample_bytree | 构造每一棵树时随机抽样出的特征占所有特征的比例 | | √ | |
| colsample_bylevel | 在树的每一层进行分支时随机抽样出的特征占所有特征的比例 | | √ | |
| reg_alpha | 目标函数中使用L1正则化时控制正则化强度 | | √ | |
| reg_lambda | 目标函数中使用L2正则化时控制正则化强度 | | √ | |
| nthread | 用于运行xgboost的并行线程数 (已弃用, 请使用n_jobs) | | | √ |
| n_jobs | 用于运行xgboost的并行线程数 (nthread取代) | | | √ |
| scale_pos_weight | 处理标签中的样本不平衡问题 | | | √ |
| base_score | 所有实例的初始预测分数, 全局偏差 | | | √ |
| seed | 随机数种子 (已弃用, 请使用random_state) | | | √ |
| random_state | 随机数种子 (取代种子) | | | √ |
| missing | 需要作为缺失值存在的数据中的值。如果为None, 则默认为np.nan。 | | | √ |
| importance_type | feature_importances_属性的特征重要性类型 | | | √ |

更多计算资源: n_jobs

nthread和n_jobs都是算法运行所使用的线程, 与sklearn中规则一样, 输入整数表示使用的线程, 输入-1表示使用计算机全部的计算资源。如果我们的数据量很大, 则我们可能需要这个参数来为我们调用更多线程。

降低学习难度: base_score

base_score是一个比较容易被混淆的参数, 它被叫做全局偏差, 在分类问题中, 它是我们希望关注的分类的先验概率。比如说, 如果我们有1000个样本, 其中300个正样本, 700个负样本, 则base_score就是0.3。对于回归来说, 这个分数默认0.5, 但其实这个分数在这种情况下并不有效。许多使用XGBoost的人已经提出, 当使用回归的时候base_score的默认应该是标签的均值, 不过现在xgboost库尚未对此做出改进。使用这个参数, 我们便是在告诉模型一些我们了解但模型不一定能够从数据中学习到的信息。通常我们不会使用这个参数, **但对于严重的样本不均衡问题, 设置一个正确的base_score取值是很有必要的。**

生成树的随机模式: random_state

在xgb库和sklearn中, 都存在空值生成树的随机模式的参数random_state。在之前的剪枝中, 我们提到可以通过随机抽样样本, 随机抽样特征来减轻过拟合的影响, 我们可以通过其他参数来影响随机抽样的比例, 却无法对随机抽样干涉更多, 因此, 真正的随机性还是由模型自己生成的。如果希望控制这种随机性, 可以在random_state参数中输入固定整数。需要注意的是, xgb库和sklearn库中, 在random_state参数中输入同一个整数未必表示同一个随机模式, 不一定会得到相同的结果, 因此导致模型的feature_importances也会不一致。

自动处理缺失值: missing

XGBoost被设计成是能够自动处理缺失值的模型，这个设计的初衷其实是为了让XGBoost能够处理稀疏矩阵。我们可以在参数missing中输入一个对象，比如np.nan，或数据的任意取值，表示将所有含有这个对象的数据作为空值处理。XGBoost会将所有的空值当作稀疏矩阵中的0来进行处理，因此在使用XGBoost的时候，我们也可以不处理缺失值。当然，通常来说，如果我们了解业务并且了解缺失值的来源，我们还是希望手动填补缺失值。

XGBoost结语

作为工程能力强，效果优秀的算法，XGBoost应用广泛并且原理复杂。不过在我们为大家解读完毕XGBoost之后，相信大家已经意识到，XGBoost的难点在于它是一个集大成的模型。它所涉及到的知识点和模型流程，多半在其他常用的机器学习模型中出现过：比如树的迭代过程，其实可以和逻辑回归中的梯度下降过程进行类比；比如剪枝的过程，许多都可以与随机森林和决策树中的知识进行对比。当然，XGBoost还有很多可以进行探索，能够使用XGB算法的库和模块也不止sklearn和xgboost，许多库对xgboost的底层原理进行了更多优化，让它变得更快更优秀（比如lightGBM，比如使用分布式进行计算等等）。本周我们学习了许多内容，大家已经对XGBoost算法有了基本的认识，了解了如何使用它来建立模型，如何使用它进行基本的调整。本周的课程只不过是梯度提升算法和XGB上的一个向导，希望大家继续探索XGBoost这个可爱的算法，再接再厉。