集成算法

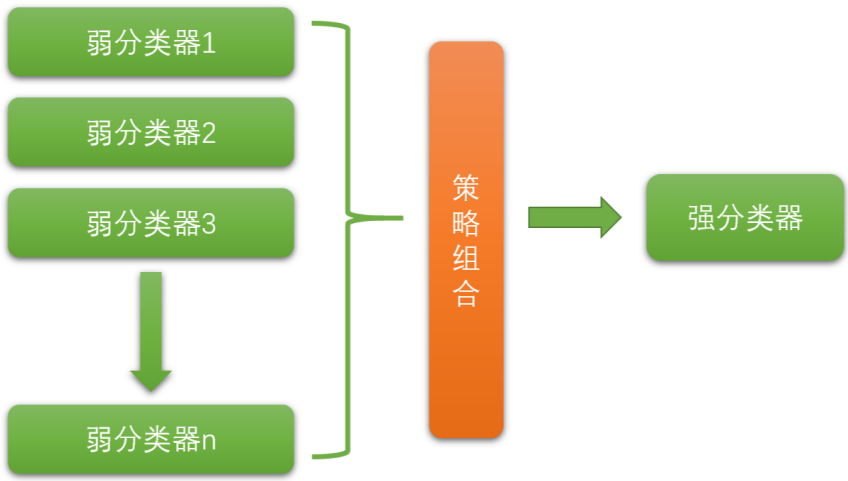
一、概述

1. 什么是集成算法

集成学习（Ensemble learning）就是将若干个弱分类器通过一定的策略组合之后产生一个强分类器，是时下非常流行的机器学习算法，它本身不是一个单独的机器学习算法，而是通过在数据上构建并结合多个机器学习器来完成学习任务，也就是我们常说的 "博采众长"。

基本上所有的机器学习领域都可以看到集成学习的身影，在现实中集成学习也有相当大的作用，它可以用来做市场营销模拟的建模，统计客户来源，保留和流失，也可用来预测疾病的风险和病患者的易感性。在现在的各种算法竞赛中，随机森林，梯度提升树（GBDT），XGBoost等集成算法的身影也随处可见，可见其效果之好，应用之广。

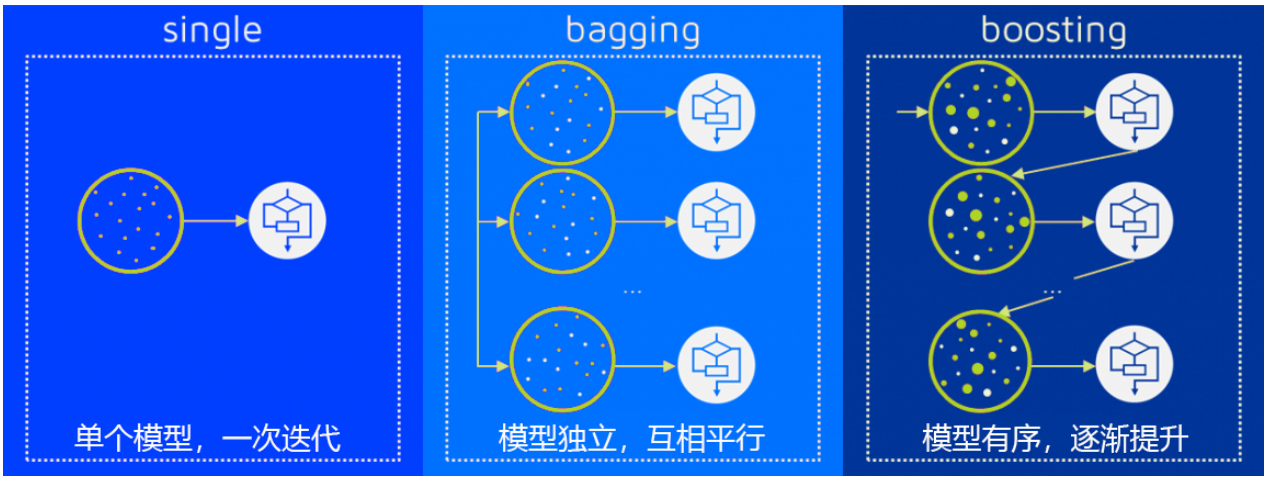
我们可以对集成学习的思想做一个概括。对于训练集数据，我们通过训练若干个「个体学习器」，通过一定的结合策略，就可以最终形成一个强学习器，已达到博采众长的目的。



集成算法的目标
集成算法会考虑多个评估器的建模结果，汇总之后得到一个综合的结果，**以此来获取比单个模型更好的回归或分类表现**。

2. 集成算法种类

多个模型集成在一起的模型叫做集成评估器（ensemble estimator），组成集成评估器的每个模型都叫做基评估器（base estimator）。通常来说，有三类集成算法：装袋法（Bagging）、提升法（Boosting）和堆叠法（stacking）。

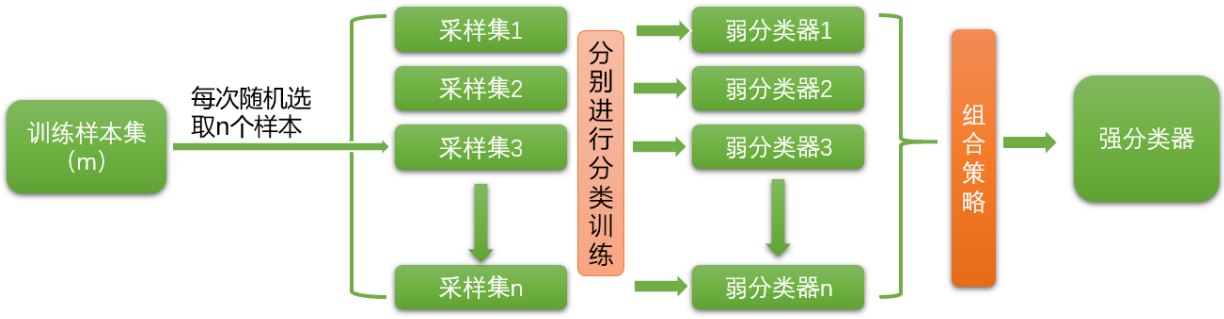


- 装袋法的核心思想是构建多个**相互独立的评估器**，然后对其预测进行平均或多数表决原则来决定集成评估器的结果。装袋法的代表模型就是随机森林。
- 提升法中，**基评估器是相关的**，是按顺序一一构建的。其核心思想是结合弱评估器的力量一次次对难以评估的样本进行预测，从而构成一个强评估器。提升法的代表模型有Adaboost和梯度提升树。

2.1 Bagging（装袋法）

又称自主聚集（bootstrap aggregating），是一种根据均匀概率分布从数据集中重复抽样（有放回的）的技术。每个新数据集和原始数据集的大小相等。由于新数据集中的每个样本都是从原始数据集中**有放回的随机抽样**出来的，所以新数据集中可能有重复的值，而原始数据集中的某些样本可能根本就没出现在新数据集中。

bagging方法的流程，如下图所示：



bagging图示

2.1.1 有放回的随机抽样

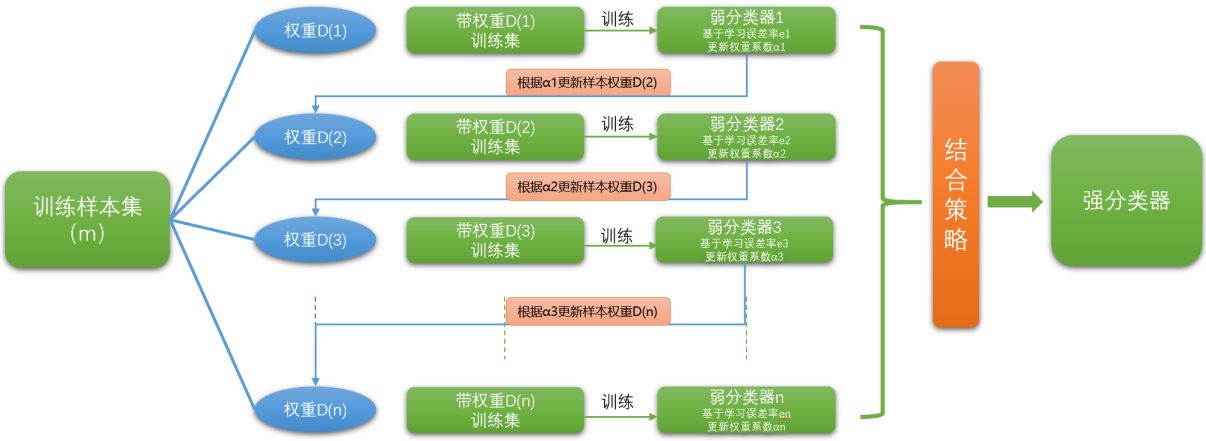
自主采样法（Bootstap sampling），也就是说对于m个样本的原始数据集，每次随机选取一个样本放入采样集，然后把这个样本重新放回原数据集中，然后再进行下一个样本的随机抽样，直到一个采样集中的数量达到m，这样一个采样集就构建好了，然后我们可以重复这个过程，生成n个这样的采样集。也就是说，最后形成的采样集，每个采样集中的样本可能是重复的，也可能原数据集中的某些样本根本就没抽到，并且每个采样集中的样本分布可能都不一样。

根据有放回的随机抽样构造的n个采样集，我们就可以对它们分别进行训练，得到n个弱分类器，然后根据每个弱分类器返回的结果，我们可以采用一定的**组合策略**得到我们最后需要的强分类器。

2.2 Boosting（提升法）

boosting是一个迭代的过程，用来自适应地改变训练样本的分布，使得弱分类器聚焦到那些很难分类的样本上。它的做法是给每一个训练样本赋予一个权重，在每一轮训练结束时自动地调整权重。

boosting方法的流程，如下图所示：



boosting图示

2.3 组合策略

2.3.1 平均法

对于数值类的回归预测问题，通常使用的结合策略是平均法，也就是说，对于若干个弱学习器的输出进行平均得到最终的预测输出。

假设我们最终得到的n个弱分类器为$\{h_1, h_2, \dots, h_n\}$

最简单的平均是算术平均，也就是说最终预测是

$$H(x) = \frac{1}{n}\sum_{1}^{n} h_i(x)$$

如果每个弱分类器有一个权重w，则最终预测是

$$H(x) = \frac{1}{n}\sum_{1}^{n} w_i h_i(x)$$

$$s.t.\ \ w_i \geq 0, \sum_{1}^{n} w_i = 1$$

2.3.2 投票法

对于分类问题的预测，我们通常使用的是投票法。假设我们的预测类别是$\{c_1, c_2, \dots c_K\}$,对于任意一个预测样本x，我们的n个弱学习器的预测结果分别是$(h_1(x), h_2(x) \dots h_n(x))$。

最简单的投票法是相对多数投票法，也就是我们常说的少数服从多数，也就是n个弱学习器的对样本x的预测结果中，数量最多的类别c_i为最终的分类类别。如果不止一个类别获得最高票，则随机选择一个做最终类别。

稍微复杂的投票法是绝对多数投票法，也就是我们常说的要票过半数。在相对多数投票法的基础上，不光要求获得最高票，还要求票过半数。否则会拒绝预测。

更加复杂的是加权投票法，和加权平均法一样，每个弱学习器的分类票数要乘以一个权重，最终将各个类别的加权票数求和，最大的值对应的类别为最终类别。

2.3.3 学习法

前两种方法都是对弱学习器的结果做平均或者投票，相对比较简单，但是可能学习误差较大，于是就有了学习法这种方法，对于学习法，代表方法是stacking，当使用stacking的结合策略时， 我们不是对弱学习器的结果做简单的逻辑处理，而是再加上一层学习器，也就是说，我们将训练集弱学习器的学习结果作为输入，将训练集的输出作为输出，重新训练一个学习器来得到最终结果。

在这种情况下，我们将弱学习器称为初级学习器，将用于结合的学习器称为次级学习器。对于测试集，我们首先用初级学习器预测一次，得到次级学习器的输入样本，再用次级学习器预测一次，得到最终的预测结果。

2.4 Bagging VS Boosting

1. **样本选择上**
    - Bagging：训练集是在原始集中有放回选取的，从原始集中选出的各轮训练集之间是独立的。
    - Boosting：每一轮的训练集不变，只是训练集中每个样例在分类器中的权重发生变化，而权值是根据上一轮的分类结果进行调整。
2. **样例权重**
    - Bagging：使用均匀取样，每个样例的权重相等。
    - Boosting：根据错误率不断调整样例的权重，错误率越大则权重越大，因此Boosting的分类精度要优于Bagging。
3. **预测函数**
    - Bagging：所有预测函数的权重相等。
    - Boosting：每个弱分类器都有相应的权重，对于分类误差小的分类器会有更大的权重。
4. **并行计算**
    - Bagging：各个预测函数可以并行生成，对于极为耗时的学习方法，Bagging可通过并行训练节省大量时间开销。
    - Boosting：各个预测函数只能顺序生成，因为后一个模型参数需要前一轮模型的结果。
5. **过拟合和欠拟合**
    - 单个评估器存在过拟合问题的时候，Bagging能在一定程度上解决过拟合问题，而Boosting可能会加剧过拟合的问题。
    - 单个评估其学习能力较弱的时候，Bagging无法提升模型表现，Boosting有一定可能提升模型的表现。
6. **算法目标**
    - Bagging：降低方差，提高模型整体的稳定性。
    - Boosting：降低偏差，提高模型整体的精确度。
    - Bagging和Boosting都可以有效地提高分类的准确性。在大多数数据集中，Boosting的准确性要高于Bagging。

二、使用sklearn实现随机森林分类器

bagging方法的代表算法是**随机森林**，准确的来说，随机森林是bagging的一个特化进阶版，所谓的特化是因为随机森林的弱学习器都是决策树。所谓的进阶是随机森林在bagging的样本随机采样基础上，又加上了特征的随机选择，其基本思想没有脱离bagging的范畴。

分类树组成的森林就叫做随机森林分类器，回归树所集成的森林就叫做随机森林回归器。这一节主要讲解RandomForestClassifier，随机森林分类器。

class `sklearn.ensemble.RandomForestClassifier`(*n_estimators='10'*, *criterion='gini'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *bootstrap=True*, *oob_score=False*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*, *class_weight=None*)

1. 重要参数

1.1 控制基评估器的参数

参数	含义
criterion	不纯度的衡量指标，有基尼系数和信息熵两种选择
max_depth	树的最大深度，超过最大深度的树枝都会被剪掉
min_samples_leaf	一个节点在分枝后的每个子节点都必须包含至少min_samples_leaf个训练样本，否则分枝就不会发生
min_samples_split	一个节点必须要包含至少min_samples_split个训练样本，这个节点才允许被分枝，否则分枝就不会发生
max_features	max_features限制分枝时考虑的特征个数，超过限制个数的特征都会被舍弃，默认值为总特征个数开平方取整
min_impurity_decrease	限制信息增益的大小，信息增益小于设定数值的分枝不会发生

这些参数在随机森林中的含义，和我们在上决策树时说明的内容一模一样，单个决策树的准确率越高，随机森林的准确率也会越高，因为装袋法是依赖于平均值或者少数服从多数原则来决定集成的结果的。

1.2 n_estimators

这是森林中树木的数量，即基评估器的数量。这个参数对随机森林模型的精确性影响是单调的，**n_estimators越大，模型的效果往往越好**。但是相应的，任何模型都有决策边界，n_estimators达到一定的程度之后，随机森林的精确性往往不在上升或开始波动，并且，n_estimators越大，需要的计算量和内存也越大，训练的时间也会越来越长。对于这个参数，我们是渴望在训练难度和模型效果之间取得平衡。

n_estimators的默认值在现有版本的sklearn中是10，但是在即将更新的0.22版本中，这个默认值会被修正为100。这个修正显示出了使用者的调参倾向：要更大的n_estimators。

- **来建立一片森林吧**

树模型的优点是简单易懂，可视化之后的树人人都能够看懂，可惜随机森林是无法被可视化的。所以为了更加直观地让大家体会随机森林的效果，我们来进行一个随机森林和单个决策树效益的对比。我们依然使用红酒数据集。

1. **导入我们需要的包**

```
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_wine
```

2. **导入需要的数据集**

```
wine = load_wine()
wine
wine.data
wine.target
```

3. **复习:sklearn建模的基本流程**

```
#切分训练集和测试集
from sklearn.model_selection import train_test_split
Xtrain, Xtest, Ytrain, Ytest =
train_test_split(wine.data,wine.target,test_size=0.3)

#建立模型
clf = DecisionTreeClassifier(random_state=0)
rfc = RandomForestClassifier(random_state=0)
clf = clf.fit(Xtrain,Ytrain)
rfc = rfc.fit(Xtrain,Ytrain)

#查看模型效果
score_c = clf.score(Xtest,Ytest)
score_r = rfc.score(Xtest,Ytest)

#打印最后结果
print("Single Tree:",score_c)
print("Random Forest:",score_r)
```

4. **画出随机森林和决策树在一组交叉验证下的效果对比**

```
#目的是带大家复习一下交叉验证
#交叉验证：是数据集划分为n分，依次取每一份做测试集，每n-1份做训练集，多次训练模型以观测模型稳定性的方法

from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt

rfc = RandomForestClassifier(n_estimators=25)
rfc_s = cross_val_score(rfc,wine.data,wine.target,cv=10)
```

```
clf = DecisionTreeClassifier()
clf_s = cross_val_score(clf,wine.data,wine.target,cv=10)

plt.plot(range(1,11),rfc_s,label = "RandomForest")
plt.plot(range(1,11),clf_s,label = "DecisionTree")
plt.legend()
plt.show()

#====================一种更加有趣也更简单的写法==================#

"""
label = "RandomForest"
for model in
[RandomForestClassifier(n_estimators=25),DecisionTreeClassifier()]:
    score = cross_val_score(model,wine.data,wine.target,cv=10)
    print("{}:".format(label)),print(score.mean())
    plt.plot(range(1,11),score,label = label)
    plt.legend()
    label = "DecisionTree"

"""
```

5. **画出随机森林和决策树在十组交叉验证下的效果对比**

```
rfc_l = []
clf_l = []

for i in range(10):
    rfc = RandomForestClassifier(n_estimators=25)
    rfc_s = cross_val_score(rfc,wine.data,wine.target,cv=10).mean()
    rfc_l.append(rfc_s)
    clf = DecisionTreeClassifier()
    clf_s = cross_val_score(clf,wine.data,wine.target,cv=10).mean()
    clf_l.append(clf_s)

#绘制结果曲线
plt.plot(range(1,11),rfc_l,label = "RandomForest")
plt.plot(range(1,11),clf_l,label = "DecisionTree")
plt.legend()
plt.show()

#是否有注意到，单个决策树的波动轨迹和随机森林一致?
#再次验证了我们之前提到的，单个决策树的准确率越高，随机森林的准确率也会越高
```

6. **n_estimators的学习曲线**

```
#======【TIME WARNING: 6mins 30 seconds】==========#

superpa = []
for i in range(200):
    rfc = RandomForestClassifier(n_estimators=i+1,n_jobs=-1)
    rfc_s = cross_val_score(rfc,wine.data,wine.target,cv=10).mean()
    superpa.append(rfc_s)
print(max(superpa),superpa.index(max(superpa)))
plt.figure(figsize=[20,5])
plt.plot(range(1,201),superpa)
plt.show()
```

思考

随机森林用了什么方法，来保证集成的效果一定好于单个分类器?

1.3 random_state

随机森林的本质是一种装袋集成算法（bagging），装袋集成算法是对基评估器的预测结果进行平均或用多数表决原则来决定集成评估器的结果。在刚才的红酒例子中，我们建立了25棵树，对任何一个样本而言，平均或多数表决原则下，当且仅当有13棵以上的树判断错误的时候，随机森林才会判断错误。单独一棵决策树对红酒数据集的分类准确率在0.85上下浮动，假设一棵树判断错误的可能性为0.2(ε)，那13棵树以上都判断错误的可能性是：

$$e_{random\_forest} = \sum_{i=13}^{25} C_{25}^{i} \varepsilon^{i} (1-\varepsilon)^{25-i} = 0.000369$$

其中，i是判断错误的次数，也是判错的树的数量，ε是一棵树判断错误的概率，（1-ε）是判断正确的概率，共判对25-i次。采用组合，是因为25棵树中，有任意i棵都判断错误。

```
import numpy as np
from scipy.special import comb

np.array([comb(25,i)*(0.2**i)*((1-0.2)**(25-i)) for i in range(13,26)]).sum()
```

可见，判断错误的几率非常小，这让随机森林在红酒数据集上的表现远远好于单棵决策树。

那现在就有一个问题了：我们说袋装法服从多数表决原则或对基分类器结果求平均，这即是说，我们默认森林中的每棵树应该是不同的，并且会返回不同的结果。设想一下，如果随机森林里所有的树的判断结果都一致（全判断对或全判断错），那随机森林无论应用何种集成原则来求结果，都应该无法比单棵决策树取得更好的效果才对。但我们使用了一样的类DecisionTreeClassifier，一样的参数，一样的训练集和测试集，为什么随机森林里的众多树会有不同的判断结果?

问到这个问题，我们可能就会想到了：sklearn中的分类树DecisionTreeClassifier自带随机性，所以随机森林中的树天生就都是不一样的。我们在讲解分类树时曾提到，决策树从最重要的特征中随机选择出一个特征来进行分枝，因此每次生成的决策树都不一样，这个功能由参数random_state控制。

随机森林中其实也有random_state，用法和分类树中相似，只不过在分类树中，一个random_state只控制生成一棵树，而随机森林中的random_state控制的是生成森林的模式，而非让一个森林中只有一棵树。

```
#随机森林中的random_state控制的是生成森林的模式
rfc = RandomForestClassifier(n_estimators=20,random_state=2)
rfc = rfc.fit(Xtrain, Ytrain)

#随机森林的重要属性之一：estimators，查看森林中树的状况
rfc.estimators_[0].random_state

#打印出森林中所有树的随机模式
for i in range(len(rfc.estimators_)):
    print(rfc.estimators_[i].random_state)
```

我们可以观察到，当random_state固定时，随机森林中生成是一组固定的树，但每棵树依然是不一致的，这是用”随机挑选特征进行分枝“的方法得到的随机性。并且我们可以证明，当这种随机性越大的时候，袋装法的效果一般会越来越好。**用袋装法集成时，基分类器应当是相互独立的，是不相同的。**

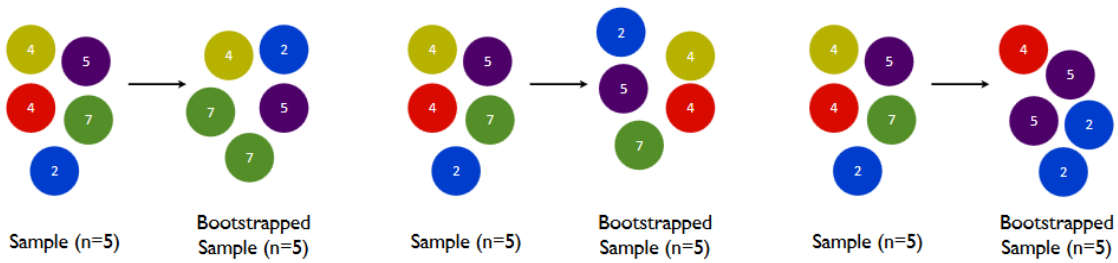
但这种做法的局限性是很强的，当我们需要成千上万棵树的时候，数据不一定能够提供成千上万的特征来让我们构筑尽量多尽量不同的树。因此，除了random_state。我们还需要其他的随机性。

1.4 bootstrap & oob_score

要让基分类器尽量都不一样，一种很容易理解的方法是使用不同的训练集来进行训练，而袋装法正是通过有放回的随机抽样技术来形成不同的训练数据，bootstrap就是用来控制抽样技术的参数。

在一个含有n个样本的原始训练集中，我们进行随机采样，每次采样一个样本，并在抽取下一个样本之前将该样本放回原始训练集，也就是说下次采样时这个样本依然可能被采集到，这样采集n次，最终得到一个和原始训练集一样大的，n个样本组成的自助集。由于是随机采样，这样每次的自助集和原始数据集不同，和其他的采样集也是不同的。这样我们就可以自由创造取之不尽用之不竭，并且互不相同的自助集，用这些自助集来训练我们的基分类器，我们的基分类器自然也就各不相同了。

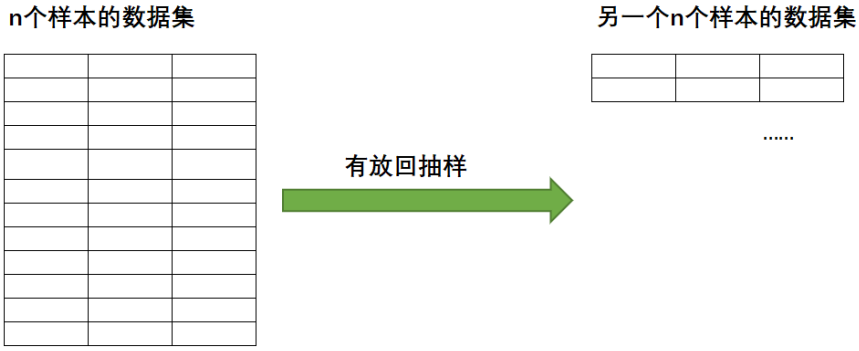
bootstrap参数默认True，代表采用这种有放回的随机抽样技术。通常，这个参数不会被我们设置为False。



然而有放回抽样也会有自己的问题。由于是有放回，一些样本可能在同一个自助集中出现多次，而其他一些却可能被忽略，一般来说，自助集大约平均会包含63%的原始数据。因为每一个样本被抽到某个自助集中的概率为：

$$1-(1-\frac{1}{n})^n$$

这里的数学公式如何理解呢？我们要从一个n个样本的数据集中有放回抽样出另一个n个样本的数据集，也就是总共需要进行n次抽样。只有当一个样本在n次抽样中都没有被抽到的时候，这个样本才是被这个自助集抽到了。在一次抽样中没有被抽到的概率是$1-\frac{1}{n}$，n次都没有抽中的概率就是$(1-\frac{1}{n})^n$，所以被抽中的概率就是1减去n次都没有抽中的概率。



当n足够大时，这个概率收敛于1-(1/e)，约等于0.632。因此，会有约37%的训练数据被浪费掉，没有参与建模， 这些数据被称为袋外数据(out of bag data，简写为oob)。除了我们最开始就划分好的测试集之外，这些数据也可以被用来作为集成算法的测试集。**也就是说，在使用随机森林时，我们可以不划分测试集和训练集，只需要用袋外数据来测试我们的模型即可。**当然，这也不是绝对的，当n和n_estimators都不够大的时候，很可能就没有数据掉落在袋外，自然也就无法使用oob数据来测试模型了。

如果希望用袋外数据来测试，则需要在实例化时就将oob_score这个参数调整为True，训练完毕之后，我们可以用随机森林的另一个重要属性：oob_score_来查看我们的在袋外数据上测试的结果：

```
#无需划分训练集和测试集
rfc = RandomForestClassifier(n_estimators=25,oob_score=True)
rfc = rfc.fit(wine.data,wine.target)

#重要属性oob_score_
rfc.oob_score_
```

2. 重要属性和接口

至此，我们已经讲完了所有随机森林中的重要参数，为大家复习了一下决策树的参数，并通过n_estimators，random_state，boostrap和oob_score这四个参数帮助大家了解了袋装法的基本流程和重要概念。同时，我们还介绍了**.estimators_** 和 **.oob_score_** 这两个重要属性。除了这两个属性之外，作为树模型的集成算法，随机森林自然也有**.feature_importances_**这个属性。

随机森林的接口与决策树完全一致，因此依然有四个常用接口：**apply, fit, predict**和**score**。除此之外，还需要注意随机森林的predict_proba接口，这个接口返回每个测试样本对应的被分到每一类标签的概率，标签有几个分类就返回几个概率。如果是二分类问题，则predict_proba返回的数值大于0.5的，被分为1，小于0.5的，被分为0。传统的随机森林是利用袋装法中的规则，平均或少数服从多数来决定集成的结果，而sklearn中的随机森林是平均每个样本对应的predict_proba返回的概率，得到一个平均概率，从而决定测试样本的分类。

```
#大家可以分别去尝试一下这些属性和接口

rfc = RandomForestClassifier(n_estimators=25)
rfc = rfc.fit(Xtrain, Ytrain)
rfc.score(Xtest,Ytest)

rfc.feature_importances_
rfc.apply(Xtest)
rfc.predict(Xtest)
rfc.predict_proba(Xtest)
```

掌握了上面的知识，基本上要实现随机森林分类已经是没问题了。从红酒数据集的表现上来看，随机森林的效用比单纯的决策树要强上不少，大家可以自己更换其他数据来试试看（比如决策树中的泰坦尼克号数据）。

3. Bagging的另一个必要条件

之前我们说过，在使用袋装法时要求基评估器要尽量独立。其实，袋装法还有另一个必要条件：基分类器的判断准确率至少要超过随机分类器，即时说，基分类器的判断准确率至少要超过50%。之前我们已经展示过随机森林的准确率公式，基于这个公式，我们画出了基分类器的误差率ε和随机森林的误差率之间的图像。大家可以自己运行一下这段代码，看看图像呈什么样的分布。

```
import numpy as np

x = np.linspace(0,1,20)

y = []
for epsilon in np.linspace(0,1,20):
    E = np.array([comb(25,i)*(epsilon**i)*((1-epsilon)**(25-i))
                  for i in range(13,26)]).sum()
```

```
    y.append(E)

#绘制图形
plt.plot(x,y,"o-",label="when estimators are different")
plt.plot(x,x,"--",color="red",label="if all estimators are same")
plt.xlabel("individual estimator's error")
plt.ylabel("RandomForest's error")
plt.legend()
plt.show()
```

可以从图像上看出，当基分类器的误差率小于0.5，即准确率大于0.5时，集成的效果是比基分类器要好的。相反，当基分类器的误差率大于0.5，袋装的集成算法就失效了。所以在使用随机森林之前，一定要检查，用来组成随机森林的分类树们是否都有至少50%的预测正确率。

三、随机森林回归器

class `sklearn.ensemble.RandomForestRegressor`(*n_estimators='warn', criterion='mse', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False*)

所有的参数，属性与接口，全部和随机森林分类器一致。仅有的不同就是回归树与分类树的不同，不纯度的指标，参数Criterion不一致。

1. 重要参数，属性与接口

- **criterion**

回归树衡量分枝质量的指标，支持的标准有三种：

1）输入"mse"使用均方误差mean squared error(MSE)，父节点和叶子节点之间的均方误差的差额将被用来作为特征选择的标准，这种方法通过使用叶子节点的均值来最小化L2损失。

2）输入“friedman_mse”使用费尔德曼均方误差，这种指标使用弗里德曼针对潜在分枝中的问题改进后的均方误差。

3）输入"mae"使用绝对平均误差MAE（mean absolute error），这种指标使用叶节点的中值来最小化L1损失。

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(f_i - y_i)^2$$

其中N是样本数量，i是每一个数据样本，f_i是模型回归出的数值，y_i是样本点i实际的数值标签。所以MSE的本质，其实是样本真实数据与回归结果的差异。**在回归树中，MSE不只是我们的分枝质量衡量指标，也是我们最常用的衡量回归树回归质量的指标**，当我们在使用交叉验证，或者其他方式获取回归树的结果时，我们往往选择均方误差作为我们的评估（在分类树中这个指标是score代表的预测准确率）。在回归中，我们追求的是，MSE越小越好。

然而，**回归树的接口score返回的是R平方，并不是MSE**。R平方被定义如下：

$$R^2 = 1 - \frac{u}{v}$$

$$u = \sum_{i=1}^{N}(f_i - y_i)^2 \qquad v = \sum_{i=1}^{N}(y_i - \hat{y})^2$$

其中u是残差平方和（MSE * N），v是总平方和，N是样本数量，i是每一个数据样本，f_i是模型回归出的数值，y_i是样本点i实际的数值标签。$\hat{y}$是真实数值标签的平均数。R^2可以为正为负（如果模型的残差平方和远远大于模型的总平方和，模型非常糟糕，R^2就会为负），而均方误差永远为正。

值得一提的是，**虽然均方误差永远为正，但是sklearn当中使用均方误差作为评判标准时，却是计算"负均方误差"（neg_mean_squared_error）**。这是因为sklearn在计算模型评估指标的时候，会考虑指标本身的性质，均方误差本身是一种误差，所以被sklearn划分为模型的一种损失(loss)，因此在sklearn当中，都以负数表示。真正的均方误差MSE的数值，其实就是neg_mean_squared_error去掉负号的数字。

- **重要属性和接口**

  最重要的属性和接口，都与随机森林的分类器相一致，还是apply, fit, predict和score最为核心。值得一提的是，随机森林回归并没有predict_proba这个接口，因为对于回归来说，并不存在一个样本要被分到某个类别的概率问题，因此没有predict_proba这个接口。

- **随机森林回归用法**

  和决策树完全一致，除了多了参数n_estimators。

```
from sklearn.datasets import load_boston
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor

boston = load_boston()
regressor = RandomForestRegressor(n_estimators=100,random_state=0)
cross_val_score(regressor, boston.data, boston.target, cv=10
                ,scoring = "neg_mean_squared_error")

#查看所有可以用的评估指标
sorted(sklearn.metrics.SCORERS.keys())
```

返回十次交叉验证的结果，注意在这里，如果不填写scoring = "neg_mean_squared_error"，交叉验证默认的模型衡量指标是R^2，因此交叉验证的结果可能有正也可能有负。而如果写上scoring，则衡量标准是负MSE，交叉验证的结果只可能为负。

2. 实例：用随机森林回归填补缺失值

我们从现实中收集的数据，几乎不可能是完美无缺的，往往都会有一些缺失值。面对缺失值，很多人选择的方式是直接将含有缺失值的样本删除，这是一种有效的方法，但是有时候填补缺失值会比直接丢弃样本效果更好，即便我们其实并不知道缺失值的真实样貌。在sklearn中，我们可以使用sklearn.impute.SimpleImputer来轻松地将均值，中值，或者其他最常用的数值填补到数据中，在这个案例中，我们将使用均值，0，和随机森林回归来填补缺失值，并验证四种状况下的拟合状况，找出对使用的数据集来说最佳的缺失值填补方法。

1. **导入需要的库**

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestRegressor

from sklearn.model_selection import cross_val_score
```

2. **以波士顿数据集为例，导入完整的数据集并探索**

```
dataset = load_boston()

dataset.data.shape
#总共506*13=6578个数据

X_full, y_full = dataset.data, dataset.target
n_samples = X_full.shape[0]
n_features = X_full.shape[1]
```

3. **为完整数据集放入缺失值**

```
#首先确定我们希望放入的缺失数据的比例，在这里我们假设是50%，那总共就要有3289个数据缺失

rng = np.random.RandomState(0)
missing_rate = 0.5
n_missing_samples = int(np.floor(n_samples * n_features * missing_rate))
#np.floor向下取整，返回.0格式的浮点数

#所有数据要随机遍布在数据集的各行各列当中，而一个缺失的数据会需要一个行索引和一个列索引
#如果能够创造一个数组，包含3289个分布在0~506中间的行索引，和3289个分布在0~13之间的列索引，那我们就可以利用索引来为数据中的任意3289个位置赋空值
#然后我们用0，均值和随机森林来填写这些缺失值，然后查看回归的结果如何
```

```
missing_features = rng.randint(0,n_features,n_missing_samples)
missing_samples = rng.randint(0,n_samples,n_missing_samples)

#missing_samples =
rng.choice(dataset.data.shape[0],n_missing_samples,replace=False)

#我们现在采样了3289个数据，远远超过我们的样本量506，所以我们使用随机抽取的函数randint。但如果我们需要的数据量小于我们的样本量506，那我们可以采用np.random.choice来抽样，choice会随机抽取不重复的随机数，因此可以帮助我们让数据更加分散，确保数据不会集中在一些行中

X_missing = X_full.copy()
y_missing = y_full.copy()

X_missing[missing_samples,missing_features] = np.nan

X_missing = pd.DataFrame(X_missing)
#转换成DataFrame是为了后续方便各种操作，numpy对矩阵的运算速度快到拯救人生，但是在索引等功能上却不如pandas来得好用
```

4. **使用0和均值填补缺失值**

```
#使用均值进行填补
from sklearn.impute import SimpleImputer
imp_mean = SimpleImputer(missing_values=np.nan, strategy='mean')
X_missing_mean = imp_mean.fit_transform(X_missing)

#使用0进行填补
imp_0 = SimpleImputer(missing_values=np.nan, strategy="constant",fill_value=0)
X_missing_0 = imp_0.fit_transform(X_missing)
```

5. **使用随机森林填补缺失值**

```
"""
使用随机森林回归填补缺失值

任何回归都是从特征矩阵中学习，然后求解连续型标签y的过程，之所以能够实现这个过程，是因为回归算法认为，特征矩阵和标签之前存在着某种联系。实际上，标签和特征是可以相互转换的，比如说，在一个“用地区，环境，附近学校数量”预测“房价”的问题中，我们既可以用“地区”，“环境”，“附近学校数量”的数据来预测“房价”，也可以反过来，用“环境”，“附近学校数量”和“房价”来预测“地区”。而回归填补缺失值，正是利用了这种思想。

对于一个有n个特征的数据来说，其中特征T有缺失值，我们就把特征T当作标签，其他的n-1个特征和原本的标签组成新的特征矩阵。那对于T来说，它没有缺失的部分，就是我们的Y_test，这部分数据既有标签也有特征，而它缺失的部分，只有特征没有标签，就是我们需要预测的部分。

特征T不缺失的值对应的其他n-1个特征 + 本来的标签：X_train
特征T不缺失的值：Y_train
```

```
特征T缺失的值对应的其他n-1个特征 + 本来的标签：X_test
特征T缺失的值：未知，我们需要预测的Y_test

这种做法，对于某一个特征大量缺失，其他特征却很完整的情况，非常适用。

那如果数据中除了特征T之外，其他特征也有缺失值怎么办?
答案是遍历所有的特征，从缺失最少的开始进行填补（因为填补缺失最少的特征所需要的准确信息最少）。
填补一个特征时，先将其他特征的缺失值用0代替，每完成一次回归预测，就将预测值放到原本的特征矩阵中，再继续填补下一个特征。每一次填补完毕，有缺失值的特征会减少一个，所以每次循环后，需要用0来填补的特征就越来越少。当进行到最后一个特征时（这个特征应该是所有特征中缺失值最多的），已经没有任何的其他特征需要用0来进行填补了，而我们已经使用回归为其他特征填补了大量有效信息，可以用来填补缺失最多的特征。
遍历所有的特征后，数据就完整，不再有缺失值了。

"""
```

```
X_missing_reg = X_missing.copy()
sortindex = np.argsort(X_missing_reg.isnull().sum(axis=0)).values

for i in sortindex:

    #构建我们的新特征矩阵和新标签
    df = X_missing_reg
    fillc = df.iloc[:,i]
    df = pd.concat([df.iloc[:,df.columns != i],pd.DataFrame(y_full)],axis=1)

    #在新特征矩阵中，对含有缺失值的列，进行0的填补
    df_0 =SimpleImputer(missing_values=np.nan,
                        strategy='constant',fill_value=0).fit_transform(df)

    #找出我们的训练集和测试集
    Ytrain = fillc[fillc.notnull()]
    Ytest = fillc[fillc.isnull()]
    Xtrain = df_0[Ytrain.index,:]
    Xtest = df_0[Ytest.index,:]

    #用随机森林回归来填补缺失值
    rfc = RandomForestRegressor(n_estimators=100)
    rfc = rfc.fit(Xtrain, Ytrain)
    Ypredict = rfc.predict(Xtest)

    #将填补好的特征返回到我们的原始的特征矩阵中
    X_missing_reg.loc[X_missing_reg.iloc[:,i].isnull(),i] = Ypredict
```

6. **对填补好的数据进行建模**

```
#对所有数据进行建模，取得MSE结果

X = [X_full,X_missing_mean,X_missing_0,X_missing_reg]

mse = []
std = []
for x in X:
    estimator = RandomForestRegressor(random_state=0, n_estimators=100)
    scores =
cross_val_score(estimator,x,y_full,scoring='neg_mean_squared_error',
cv=5).mean()
    mse.append(scores * -1)
```

7. **用所得结果画出条形图**

```
x_labels = ['Full data',
            'Zero Imputation',
            'Mean Imputation',
            'Regressor Imputation']
colors = ['r', 'g', 'b', 'orange']

plt.figure(figsize=(12, 6))
ax = plt.subplot(111)
for i in np.arange(len(mse)):
    ax.barh(i, mse[i],color=colors[i], alpha=0.6, align='center')
ax.set_title('Imputation Techniques with Boston Data')
ax.set_xlim(left=np.min(mse) * 0.9,
             right=np.max(mse) * 1.1)
ax.set_yticks(np.arange(len(mse)))
ax.set_xlabel('MSE')
ax.set_yticklabels(x_labels)
plt.show()
```

四、AdaBoost的Scikit-learn实现

AdaBoost，是英文"Adaptive Boosting"（自适应增强）的缩写，由 Yoav Freund 和 Robert Schapire 在1995年提出。它的自适应在于：前一个基本分类器分错的样本会得到加强（也就是得到更高的权重），加权后的全体样本再次被用来训练下一个基本分类器。同时，在每一轮中加入一个新的弱分类器，直到达到某个预定的足够小的错误率或达到预先指定的最大迭代次数，算法停止。

具体说来，整个Adaboost 迭代算法就3步：

- **初始化训练数据的权重。**

如果有N个样本，则每一个训练样本最开始时都被赋予相同的权值：1/N。

- **训练弱分类器。**

  具体训练过程中，如果某个样本点已经被准确地分类，那么在构造下一个训练集中，它的权值就被降低；相反，如果某个样本点没有被准确地分类，那么它的权值就得到提高。然后，权值更新过的样本集被用于训练下一个分类器，整个训练过程如此迭代地进行下去。

- **将各个训练得到的弱分类器组合成强分类器。**

  各个弱分类器的训练过程结束后，加大分类误差率小的弱分类器的权重，使其在最终的分类函数中起着较大的决定作用，而降低分类误差率大的弱分类器的权重，使其在最终的分类函数中起着较小的决定作用。换言之，误差率低的弱分类器在最终分类器中占的权重较大，否则较小。

顺便提一下，原则上，只要表现略好于随机猜测的算法都可以作为弱学习器，大家需要知道的是，以决策树作为弱学习器的AdaBoost通常被称为最佳开箱即用的分类器~

Scikit-Learn中AdaBoost类库比较直接，就是AdaBoostClassifier和AdaBoostRegressor两个，从名字就可以看出AdaBoostClassifier用于分类，AdaBoostRegressor用于回归。 AdaBoostClassifier使用了两种AdaBoost分类算法的实现，SAMME和SAMME.R。而AdaBoostRegressor则使用了Adaboost.R2。

当我们对Adaboost调参时，主要要对两部分内容进行调参，第一部分是对我们的Adaboost的框架进行调参， 第二部分是对我们选择的弱分类器进行调参。两者相辅相成。下面就对AdaBoost的两个类：AdaBoostClassifier和AdaBoostRegressor从这两部分做一个介绍。

1. AdaBoost分类器

class `sklearn.ensemble.``AdaBoostClassifier` (*base_estimator = None*，*n_estimators = 50*，*learning_rate = 1.0*，*algorithm ='SAMME.R'*，*random_state = None*)

1.1 核心参数

参数	参数说明
base_estimator	None或输入字符串，默认为None 即我们的弱分类器。 理论上可以选择任何一个分类学习器，不过需要支持样本权重。我们常用的一般是CART决策树或者神经网络MLP。 如果为None，则弱分类器为DecisionTreeClassifier（max_depth = 1）
n_estimators	整数，默认为50 弱学习器的最大迭代次数，或者说最大的弱学习器的个数。 一般来说n_estimators太小，容易欠拟合，n_estimators太大，又容易过拟合，一般选择一个适中的数值。默认是50。在实际调参的过程中，我们常常将n_estimators和参数learning_rate一起考虑。
learning_rate	浮点数，默认为1. 即每个弱学习器的权重缩减系数。对于同样的训练集拟合效果，较小的权重缩减系数意味着我们需要更多的弱学习器的迭代次数。通常我们用步长和迭代最大次数一起来决定算法的拟合效果。所以这两个参数n_estimators和learning_rate要一起调参。一般来说，可以从一个小一点的learning_rate开始调参，默认是1。
algorithm	可选择{ ‘SAMME’ ， ‘SAMME.R’ }，默认是’ SAMME.R’ scikit-learn实现了两种Adaboost分类算法，SAMME和SAMME.R。两者的主要区别是弱学习器权重的度量，SAMME使用了对样本集分类效果作为弱学习器权重，而SAMME.R使用了对样本集分类的预测概率大小来作为弱学习器权重。 由于SAMME.R使用了概率度量的连续值，迭代一般比SAMME快，因此AdaBoostClassifier的默认算法algorithm的值也是SAMME.R。我们一般使用默认的SAMME.R就够了，但是要注意的是使用了SAMME.R， 则弱分类学习器参数base_estimator必须限制使用**支持概率预测**的分类器。SAMME算法则没有这个限制。
random_state	整数，sklearn中设定好的RandomState实例，或None，可不填，默认为None 1）输入整数，random_state是随机数生成器生成的随机数种子 2）输入RandomState实例，则random_state是一个随机数生成器 3）输入None，随机数生成器会是np.random模块中的RandomState实例

1.2 弱学习器参数

由于使用不同的弱学习器，则对应的弱学习器参数各不相同。这里我们仅仅讨论默认的决策树弱学习器的参数。即回顾下CART分类树DecisionTreeClassifier和CART回归树DecisionTreeRegressor。

参数	参数说明
max_features	划分时考虑的最大特征数，可以使用很多种类型的值，默认是"None" 1）"None"：划分时考虑所有的特征数 2）"log2"：划分时最多考虑$\log_2 N$个特征 3）"sqrt"或者"auto"：划分时最多考虑$\sqrt{N}$个特征 4）整数：代表考虑的特征绝对数 5）浮点数：代表考虑特征百分比，即考虑（百分比xN）取整后的特征数 其中N为样本总特征数。一般来说，如果样本特征数不多，比如小于50，我们用默认的"None"就可以了，如果特征数非常多，我们可以灵活使用刚才描述的其他取值来控制划分时考虑的最大特征数，以控制决策树的生成时间。
max_depth	决策树最大深，默认可以不输入 如果不输入的话，决策树在建立子树的时候不会限制子树的深度。一般来说，数据少或者特征少的时候可以不管这个值。如果模型样本量多，特征也多的情况下，推荐限制这个最大深度，具体的取值取决于数据的分布。常用的可以取值10-100之间。
min_samples_split	内部节点再划分所需最小样本数。 这个值限制了子树继续划分的条件，如果某节点的样本数少于min_samples_split，则不会继续再尝试选择最优特征来进行划分。 默认是2.如果样本量不大，不需要管这个值。如果样本量数量级非常大，则推荐增大这个值。
min_samples_leaf	叶子节点最少样本数。 这个值限制了叶子节点最少的样本数，如果某叶子节点数目小于样本数，则会和兄弟节点一起被剪枝。 默认是1,可以输入最少的样本数的整数，或者最少样本数占样本总数的百分比。如果样本量不大，不需要管这个值。如果样本量数量级非常大，则推荐增大这个值。
min_weight_fraction_leaf	叶子节点最小的样本权重和。 这个值限制了叶子节点所有样本权重和的最小值，如果小于这个值，则会和兄弟节点一起被剪枝。 默认是0，就是不考虑权重问题。一般来说，如果我们有较多样本有缺失值，或者分类树样本的分布类别偏差很大，就会引入样本权重，这时我们就要注意这个值了。
max_leaf_nodes	最大叶子节点数。 通过限制最大叶子节点数，可以防止过拟合，默认是"None”，即不限制最大的叶子节点数。如果加了限制，算法会建立在最大叶子节点数内最优的决策树。如果特征不多，可以不考虑这个值，但是如果特征分成多的话，可以加以限制，具体的值可以通过交叉验证得到。

1.3 AdaBoostClassifier实战

这里我们用一个具体的例子来讲解AdaBoostClassifier的使用。 首先我们载入需要的类库：

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_gaussian_quantiles
```

- 接着我们生成一些随机数据

```
# 生成2维正态分布，生成的数据按分位数分为两类，500个样本,2个样本特征，协方差系数为2
X1, y1 = make_gaussian_quantiles(cov=2.0,n_samples=500,
n_features=2,n_classes=2, random_state=1)

# 生成2维正态分布，生成的数据按分位数分为两类，400个样本,2个样本特征均值都为3，协方差系数为2
X2, y2 = make_gaussian_quantiles(mean=(3, 3), cov=1.5,n_samples=400,
n_features=2, n_classes=2, random_state=1)

#将两组数据合成一组数据
X = np.concatenate((X1, X2))
y = np.concatenate((y1, - y2 + 1))
```

- 我们通过可视化看看我们的分类数据，它有两个特征，两个输出类别，用颜色区别。

```
plt.scatter(X[:, 0], X[:, 1], marker='o', c=y)
```

- 可以看到数据有些混杂，我们现在用基于决策树的Adaboost来做分类拟合。

```
bdt = AdaBoostClassifier(DecisionTreeClassifier(max_depth=2,
min_samples_split=20,min_samples_leaf=5),algorithm="SAMME",n_estimators=200,
learning_rate=0.8)
bdt.fit(X, y)
```

- 这里我们选择了SAMME算法，最多200个弱分类器，步长0.8，在实际运用中你可能需要通过交叉验证调参而选择最好的参数。拟合完了后，我们用网格图来看看它拟合的区域。

```
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                     np.arange(y_min, y_max, 0.02))

Z = bdt.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
cs = plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
plt.scatter(X[:, 0], X[:, 1], marker='o', c=y)
plt.show()
```

- 从图中可以看出，Adaboost的拟合效果还是不错的，现在我们看看拟合分数：

```
print("Score:" ,bdt.score(X,y))
```

也就是说拟合训练集数据的分数还不错。当然分数高并不一定好，因为可能过拟合。

- 现在我们将最大弱分离器个数从200增加到300。再来看看拟合分数。

```
bdt = AdaBoostClassifier(DecisionTreeClassifier(max_depth=2,
min_samples_split=20
                                                ,
min_samples_leaf=5),algorithm="SAMME"
                                                , n_estimators=300,
learning_rate=0.8)
bdt.fit(X, y)
print("Score:" ,bdt.score(X,y))
```

这印证了我们前面讲的，弱分离器个数越多，则拟合程度越好，当然也越容易过拟合。

- 现在我们降低步长，将步长从上面的0.8减少到0.5，再来看看拟合分数。

```
bdt = AdaBoostClassifier(DecisionTreeClassifier(max_depth=2,
min_samples_split=20
                                                ,
min_samples_leaf=5),algorithm="SAMME"
                                                ,n_estimators=300,
learning_rate=0.5)
bdt.fit(X, y)
print("Score:" ,bdt.score(X,y))
```

可见在同样的弱分类器的个数情况下，如果减少步长，拟合效果会下降。

- 最后我们看看当弱分类器个数为700，步长为0.7时候的情况：

```
bdt = AdaBoostClassifier(DecisionTreeClassifier(max_depth=2,
min_samples_split=20, min_samples_leaf=5),algorithm="SAMME",n_estimators=600,
learning_rate=0.7)
bdt.fit(X, y)
print("Score:"), bdt.score(X,y)
```

此时的拟合分数和我们最初的300弱分类器，0.8步长的拟合程度相当。也就是说，在我们这个例子中，如果步长从0.8降到0.7，则弱分类器个数要从300增加到700才能达到类似的拟合效果。

2. AdaBoost回归器

接下来我们一起看一下AdaBoost回归器如何工作~

2.1 核心参数

参数	参数说明
base_estimator	None或输入字符串，默认为None 即我们的弱分类器。 理论上可以选择任何一个分类学习器，不过需要支持样本权重。我们常用的一般是CART决策树或者神经网络MLP。 如果为None，则弱分类器为**DecisionTreeRegressor(max_depth=3)**
n_estimators	整数，默认为50 弱学习器的最大迭代次数，或者说最大的弱学习器的个数。 一般来说n_estimators太小，容易欠拟合，n_estimators太大，又容易过拟合，一般选择一个适中的数值。默认是50。在实际调参的过程中，我们常常将n_estimators和参数learning_rate一起考虑。
learning_rate	浮点数，默认为1. 即每个弱学习器的权重缩减系数。对于同样的训练集拟合效果，较小的权重缩减系数意味着我们需要更多的弱学习器的迭代次数。通常我们用步长和迭代最大次数一起来决定算法的拟合效果。所以这两个参数n_estimators和learning_rate要一起调参。一般来说，可以从一个小一点的learning_rate开始调参，默认是1。
loss	可选择{ ‘linear’ ， ‘square’ ， ‘exponential’ }，默认是’ linear’ 一般使用线性就足够了，除非你怀疑这个参数导致拟合程度不好。这个值对应了我们对第k个弱分类器的中第i个样本的误差的处理
random_state	整数，sklearn中设定好的RandomState实例，或None，可不填，默认为None 1）输入整数，random_state是随机数生成器生成的随机数种子 2）输入RandomState实例，则random_state是一个随机数生成器 3）输入None，随机数生成器会是np.random模块中的RandomState实例

至于弱分类器我们还是使用决策树，相关参数上面已经写过，这里我们就不再赘述啦~

接下来我们用一个小例子一起来看看AdaBoostRegressor的使用。

```
#导入所需的模块和包
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor

#创造数据集
rng = np.random.RandomState(1)
X = np.linspace(0, 6, 100).reshape(-1,1)
y = np.sin(X).ravel() + np.sin(6 * X).ravel() + rng.normal(0, 0.1, X.shape[0])

#训练回归模型
regr_1 = DecisionTreeRegressor(max_depth=4)

regr_2 = AdaBoostRegressor(DecisionTreeRegressor(max_depth=4),
```

```
                          n_estimators=300, random_state=rng)

regr_1.fit(X, y)
regr_2.fit(X, y)

#预测结果
y_1 = regr_1.predict(X)
y_2 = regr_2.predict(X)

#绘制可视化图形
plt.figure()
plt.scatter(X, y, c="k", label="training samples")
plt.plot(X, y_1, c="g", label="DTR", linewidth=2)
plt.plot(X, y_2, c="r", label="ABR", linewidth=2)
plt.xlabel("data")
plt.ylabel("target")
plt.title("Boosted Decision Tree Regression")
plt.legend()
plt.show()
```

*五、机器学习中调参的基本思想

市面上大多数的机器学习相关的书都是遍历各种算法和案例，为大家讲解各种各样算法的原理和用途，但却对调参探究甚少。这中间有许多原因，其一是因为，调参的方式总是根据数据的状况而定，所以没有办法一概而论；其二是因为，其实大家也都没有特别好的办法。

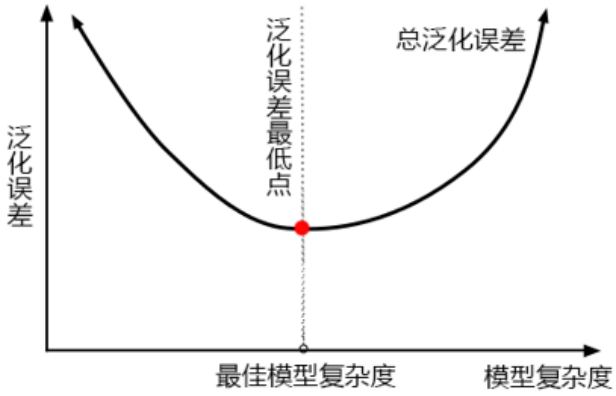
通过画学习曲线，或者网格搜索，我们能够探索到调参边缘（代价可能是训练一次模型要跑三天三夜），但是在现实中，高手调参恐怕还是多依赖于经验，而这些经验，来源于：1）非常正确的调参思路和方法，2）对模型评估指标的理解，3）对数据的感觉和经验，4）用洪荒之力去不断地尝试。

我们也许无法学到高手们多年累积的经验，但我们可以学习他们对模型评估指标的理解和调参的思路。

那我们首先来讲讲正确的调参思路。模型调参，第一步是要找准目标：我们要做什么？一般来说，这个目标是提升某个模型评估指标，比如对于随机森林来说，我们想要提升的是模型在未知数据上的准确率（由score或oob_score_来衡量）。找准了这个目标，我们就需要思考：模型在未知数据上的准确率受什么因素影响？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）**。

1. 泛化误差

当模型在未知数据（测试集或者袋外数据）上表现糟糕时，我们说模型的泛化程度不够，泛化误差大，模型的效果不好。泛化误差受到模型的结构（复杂度）影响。看下面这张图，它准确地描绘了泛化误差与模型复杂度的关系，当模型太复杂，模型就会过拟合，泛化能力就不够，所以泛化误差大。当模型太简单，模型就会欠拟合，拟合能力就不够，所以误差也会大。只有当模型的复杂度刚刚好的才能够达到泛化误差最小的目标。



那模型的复杂度与我们的参数有什么关系呢？对树模型来说，树越茂盛，深度越深，枝叶越多，模型就越复杂。所以树模型是天生位于图的右上角的模型，随机森林是以树模型为基础，所以随机森林也是天生复杂度高的模型。随机森林的参数，都是向着一个目标去：减少模型的复杂度，把模型往图像的左边移动，防止过拟合。当然了，调参没有绝对，也有天生处于图像左边的随机森林，所以调参之前，我们要先判断，模型现在究竟处于图像的哪一边。

泛化误差的背后其实是“偏差-方差困境”，在下一节偏差vs方差中，我用最简单易懂的语言为大家解释了泛化误差背后的原理，大家选读。那我们只需要记住这四点：

1）模型太复杂或者太简单，都会让泛化误差高，我们追求的是位于中间的平衡点

2）模型太复杂就会过拟合，模型太简单就会欠拟合

3）对树模型和树的集成模型来说，树的深度越深，枝叶越多，模型越复杂

4）树模型和树的集成模型的目标，都是减少模型复杂度，把模型往图像的左边移动

那具体每个参数，都如何影响我们的复杂度和模型呢？我们一直以来调参，都是在学习曲线上轮流找最优值，盼望能够将准确率修正到一个比较高的水平。然而我们现在了解了随机森林的调参方向：降低复杂度，我们就可以将那些对复杂度影响巨大的参数挑选出来，研究他们的单调性，然后专注调整那些能最大限度让复杂度降低的参数。对于那些不单调的参数，或者反而会让复杂度升高的参数，我们就视情况使用，大多时候甚至可以退避。基于经验，我对各个参数对模型的影响程度做了一个排序。在我们调参的时候，大家可以参考这个顺序。

参数	对模型在未知数据上的评估性能的影响	影响程度
n_estimators	提升至平稳，n_estimators↑，不影响单个模型的复杂度	⭐⭐⭐⭐
max_depth	有增有减，默认最大深度，即最高复杂度，向复杂度降低的方向调参max_depth↓，模型更简单，且向图像的左边移动	⭐⭐⭐
min_samples_leaf	有增有减，默认最小限制1，即最高复杂度，向复杂度降低的方向调参min_samples_leaf↑，模型更简单，且向图像的左边移动	⭐⭐
min_samples_split	有增有减，默认最小限制2，即最高复杂度，向复杂度降低的方向调参min_samples_split↑，模型更简单，且向图像的左边移动	⭐⭐
max_features	有增有减，默认auto，是特征总数的开平方，位于中间复杂度，既可以向复杂度升高的方向，也可以向复杂度降低的方向调参 max_features↓，模型更简单，图像左移 max_features↑，模型更复杂，图像右移 max_features是唯一的，既能够让模型更简单，也能够让模型更复杂的参数，所以在调整这个参数的时候，需要考虑我们调参的方向	⭐
criterion	有增有减，一般使用gini	看具体情况

有了以上的知识储备，我们现在也能够通过参数的变化来了解，模型什么时候到达了极限，当复杂度已经不能再降低的时候，我们就不必再调整了，因为调整大型数据的参数是一件非常费时费力的事。除了学习曲线和网格搜索，我们现在有了基于对模型和正确的调参思路的“推测”能力，这能够让我们的调参能力更上一层楼。

2. 偏差 vs 方差【选读】

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。

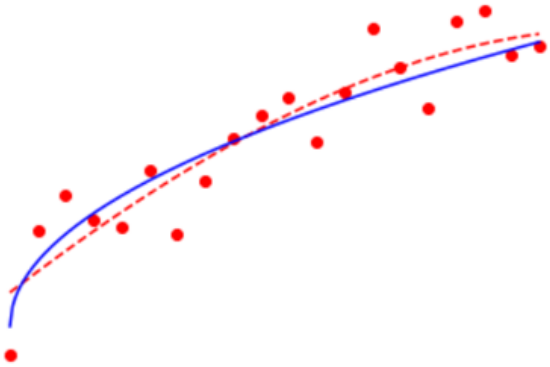
$$E(f;D) = bias^2(x) + var(x) + \varepsilon^2$$

关键概念：偏差与方差

观察下面的图像，每个点就是集成算法中的一个基评估器产生的预测值。红色虚线代表着这些预测值的均值，而蓝色的线代表着数据本来的面貌。

偏差：模型的预测值与真实值之间的差异，即每一个红点到蓝线的距离。在集成算法中，每个基评估器都会有自己的偏差，集成评估器的偏差是所有基评估器偏差的均值。模型越精确，偏差越低。

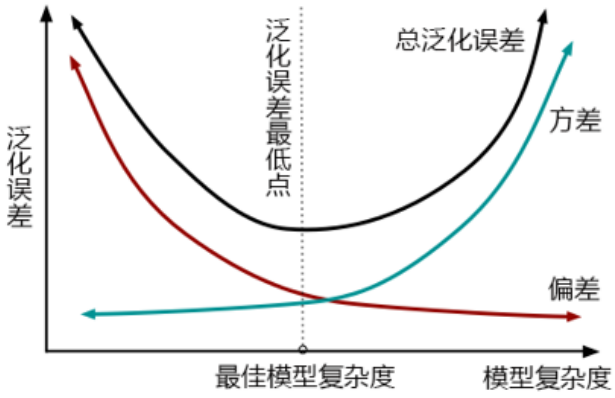
方差：反映的是模型每一次输出结果与模型预测值的平均水平之间的误差，即每一个红点到红色虚线的距离，衡量模型的稳定性。模型越稳定，方差越低。



其中偏差衡量模型是否预测得准确，偏差越小，模型越“准”；而方差衡量模型每次预测的结果是否接近，即是说方差越小，模型越“稳”；噪声是机器学习无法干涉的部分，为了让世界美好一点，我们就不去研究了。**一个好的模型，要对大多数未知数据都预测得“准”又“稳”。**即是说，当偏差和方差都很低的时候，模型的泛化误差就小，在未知数据上的准确率就高。

	偏差大	**偏差小**
方差大	模型不适合这个数据 换模型	过拟合 模型很复杂 对某些数据集预测很准确 对某些数据集预测很糟糕
方差小	欠拟合 模型相对简单 预测很稳定 但对所有的数据预测都不太准确	泛化误差小，我们的目标

通常来说，方差和偏差有一个很大，泛化误差都会很大。然而，方差和偏差是此消彼长的，不可能同时达到最小值。这个要怎么理解呢？来看看下面这张图：



从图上可以看出，模型复杂度大的时候，方差高，偏差低。偏差低，就是要求模型要预测得“准”。模型就会更努力去学习更多信息，会具体于训练数据，这会导致，模型在一部分数据上表现很好，在另一部分数据上表现却很糟糕。模型泛化性差，在不同数据上表现不稳定，所以方差就大。而要尽量学习训练集，模型的建立必然更多细节，复杂程度必然上升。**所以，复杂度高，方差高，总泛化误差高。**

相对的，复杂度低的时候，方差低，偏差高。方差低，要求模型预测得“稳”，泛化性更强，那对于模型来说，它就不需要对数据进行一个太深的学习，只需要建立一个比较简单，判定比较宽泛的模型就可以了。结果就是，模型无法在某一类或者某一组数据上达成很高的准确度，所以偏差就会大。**所以，复杂度低，偏差高，总泛化误差高。**

我们调参的目标是，达到方差和偏差的完美平衡！虽然方差和偏差不能同时达到最小值，但他们组成的泛化误差却可以有一个最低点，而我们就是要寻找这个最低点。对复杂度大的模型，要降低方差，对相对简单的模型，要降低偏差。随机森林的基评估器都拥有较低的偏差和较高的方差，因为决策树本身是预测比较“准“，比较容易过拟合的模型，装袋法本身也要求基分类器的准确率必须要有50%以上。**所以以随机森林为代表的装袋法的训练过程旨在降低方差，即降低模型复杂度，所以随机森林参数的默认设定都是假设模型本身在泛化误差最低点的右边。**

所以，我们在降低复杂度的时候，本质其实是在降低随机森林的方差，随机森林所有的参数，也都是朝着降低方差的目标去。有了这一层理解，我们对复杂度和泛化误差的理解就更上一层楼了，对于我们调参，也有了更大的帮助。

关于方差-偏差的更多内容，大家可以参考周志华的《机器学习》。

数据挖掘导论

作者: (美)Pang-Ning Tan / Michael Steinbach / Vipin Kumar
出版社: 机械工业出版社
副标题: （英文版）
出版年: 2010-9
页数: 769
定价: 59.00元
丛书: 经典原版书库
ISBN: 9787111316701

机器学习

作者: 周志华
出版社: 清华大学出版社
出版年: 2016-1-1
页数: 425
定价: 88.00元
装帧: 平装
ISBN: 9787302423287

*六、实例：随机森林在乳腺癌数据上的调参

我们了解了随机森林，并且学习了机器学习中调参的基本思想，了解了方差和偏差如何受到随机森林的参数们的影响。这一节，我们就来使用我们刚才学的，基于方差和偏差的调参方法，在乳腺癌数据上进行一次随机森林的调参。乳腺癌数据是sklearn自带的分类数据之一。接下来就用乳腺癌数据，来看看我们的调参代码。

1. **导入需要的库**

```
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

2. **导入数据集，探索数据**

```
data = load_breast_cancer()

data

data.data.shape

data.target

#可以看到，乳腺癌数据集有569条记录，30个特征，单看维度虽然不算太高，但是样本量非常少。过拟合的情况可能存在。
```

3. **进行一次简单的建模，看看模型本身在数据集上的效果**

```
rfc = RandomForestClassifier(n_estimators=100,random_state=90)
score_pre = cross_val_score(rfc,data.data,data.target,cv=10).mean()

score_pre

#这里可以看到，随机森林在乳腺癌数据上的表现本就还不错，在现实数据集上，基本上不可能什么都不调就看到95%以上的准确率
```

4. **随机森林调整的第一步：无论如何先来调n_estimators**

```
"""
```

```
在这里我们选择学习曲线，可以使用网格搜索吗？可以，但是只有学习曲线，才能看见趋势
我个人的倾向是，要看见n_estimators在什么取值开始变得平稳，是否一直推动模型整体准确率的上升等信息
第一次的学习曲线，可以先用来帮助我们划定范围，我们取每十个数作为一个阶段，来观察n_estimators的变化如何引起模型整体准确率的变化
"""

#####【TIME WARNING: 30 seconds】#####

scorel = []
for i in range(0,200,10):
    rfc = RandomForestClassifier(n_estimators=i+1,
                                 n_jobs=-1,
                                 random_state=90)
    score = cross_val_score(rfc,data.data,data.target,cv=10).mean()
    scorel.append(score)

#打印出最高score和最优n_estimators，并绘制学习曲线
print('最高score:',max(scorel))
print('最优n_estimators:',(scorel.index(max(scorel))*10)+1)
plt.figure(figsize=[20,5])
plt.plot(range(1,201,10),scorel)
plt.show()

#list.index([object])
#返回这个object在列表list中的索引
```

5. **在确定好的范围内，进一步细化学习曲线**

```
scorel = []
for i in range(35,45):
    rfc = RandomForestClassifier(n_estimators=i,
                                 n_jobs=-1,
                                 random_state=90)
    score = cross_val_score(rfc,data.data,data.target,cv=10).mean()
    scorel.append(score)

print(max(scorel),([*range(35,45)][scorel.index(max(scorel))]))
plt.figure(figsize=[20,5])
plt.plot(range(35,45),scorel)
plt.show()
```

调整n_estimators的效果显著，模型的准确率立刻上升了0.005。接下来就进入网格搜索，我们将使用网格搜索对参数一个个进行调整。为什么我们不同时调整多个参数呢？

原因有两个：

- 同时调整多个参数会运行非常缓慢，在课堂上我们没有这么多的时间。

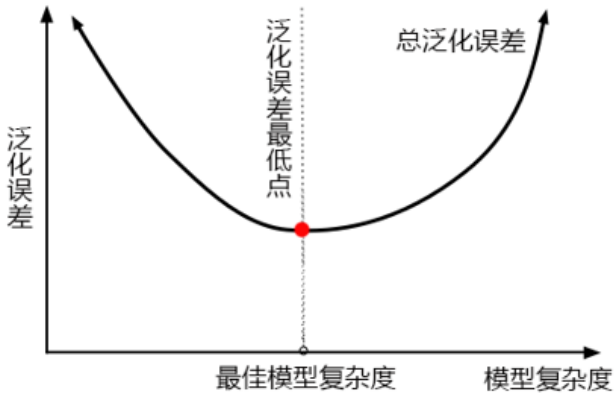
- 同时调整多个参数，会让我们无法理解参数的组合是怎么得来的，所以即便网格搜索调出来的结果不好，我们也不知道从哪里去改。在这里，为了使用复杂度-泛化误差方法（方差-偏差方法），我们对参数进行一个个地调整。

6. **为网格搜索做准备，书写网格搜索的参数**

```
"""
有一些参数是没有参照的，很难说清一个范围，这种情况下我们使用学习曲线，看趋势
从曲线跑出的结果中选取一个更小的区间，再跑曲线

param_grid = {'n_estimators':np.arange(0, 200, 10)}

param_grid = {'max_depth':np.arange(1, 20, 1)}

param_grid = {'max_leaf_nodes':np.arange(25,50,1)}
  对于大型数据集，可以尝试从1000来构建，先输入1000，每100个为一个区间，再逐渐缩小范围

有一些参数是可以找到一个范围的，或者说我们知道他们的取值和随着他们的取值，模型的整体准确率会如何变化，这样的参数我们就可以直接跑网格搜索
param_grid = {'criterion':['gini', 'entropy']}

param_grid = {'min_samples_split':np.arange(2, 2+20, 1)}

param_grid = {'min_samples_leaf':np.arange(1, 1+10, 1)}

param_grid = {'max_features':np.arange(5,30,1)}

"""
```

7. **开始按照参数对模型整体准确率的影响程度进行调参，首先调整max_depth**

```
#调整max_depth

param_grid = {'max_depth':np.arange(1, 20, 1)}

# 一般根据数据的大小来进行一个试探，乳腺癌数据很小，所以可以采用1~10，或者1~20这样的试探
# 但对于像digit recognition那样的大型数据来说，我们应该尝试30~50层深度（或许还不足够
#    更应该画出学习曲线，来观察深度对模型的影响

rfc = RandomForestClassifier(n_estimators=39,random_state=90)
GS = GridSearchCV(rfc,param_grid,cv=10)
GS.fit(data.data,data.target)

GS.best_params_
GS.best_score_
```



在这里，我们注意到，将max_depth设置为有限之后，模型的准确率下降了。限制max_depth，是让模型变得简单，把模型向左推，而模型整体的准确率下降了，即整体的泛化误差上升了，这说明模型现在位于图像左边，即泛化误差最低点的左边（偏差为主导的一边）。通常来说，随机森林应该在泛化误差最低点的右边，树模型应该倾向于过拟合，而不是拟合不足。这和数据集本身有关，但也有可能是我们调整的n_estimators对于数据集来说太大，因此将模型拉到泛化误差最低点去了。然而，既然我们追求最低泛化误差，那我们就保留这个n_estimators，除非有其他的因素，可以帮助我们达到更高的准确率。

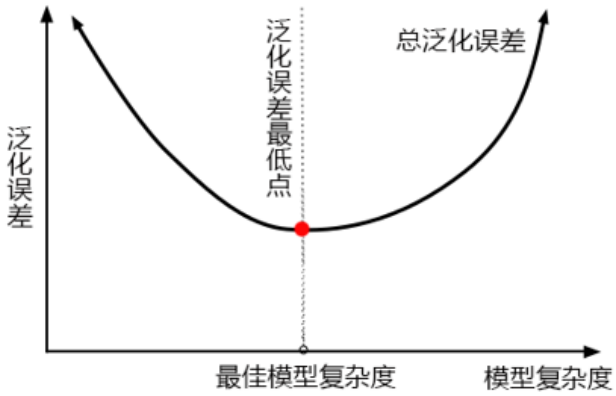
当模型位于图像左边时，我们需要的是增加模型复杂度（增加方差，减少偏差）的选项，因此max_depth应该尽量大，min_samples_leaf和min_samples_split都应该尽量小。这几乎是在说明，除了max_features，我们没有任何参数可以调整了，因为max_depth，min_samples_leaf和min_samples_split是剪枝参数，是减小复杂度的参数。在这里，我们可以预言，我们已经非常接近模型的上限，模型很可能没有办法再进步了。

那我们这就来调整一下max_features，看看模型如何变化。

8. **调整max_features**

```
#调整max_features

param_grid = {'max_features':np.arange(5,30,1)}

"""

max_features是唯一一个即能够将模型往左（低方差高偏差）推，也能够将模型往右（高方差低偏差）推的参数。我们需要根据调参前，模型所在的位置（在泛化误差最低点的左边还是右边）来决定我们要将max_features往哪边调。现在模型位于图像左侧，我们需要的是更高的复杂度，因此我们应该把max_features往更大的方向调整，可用的特征越多，模型才会越复杂。max_features的默认最小值是sqrt(n_features)，因此我们使用这个值作为调参范围的最小值。

"""

rfc = RandomForestClassifier(n_estimators=39,random_state=90)
GS = GridSearchCV(rfc,param_grid,cv=10)
GS.fit(data.data,data.target)
```

```
GS.best_params_
GS.best_score_
```



网格搜索返回了max_features的最小值，可见max_features升高之后，模型的准确率降低了。这说明，我们把模型往右推，模型的泛化误差增加了。前面用max_depth往左推，现在用max_features往右推，泛化误差都增加，这说明模型本身已经处于泛化误差最低点，已经达到了模型的预测上限，没有参数可以左右的部分了。剩下的那些误差，是噪声决定的，已经没有方差和偏差的舞台了。

如果是现实案例，我们到这一步其实就可以停下了，因为复杂度和泛化误差的关系已经告诉我们，模型不能再进步了。调参和训练模型都需要很长的时间，明知道模型不能进步了还继续调整，不是一个有效率的做法。如果我们希望模型更进一步，我们会选择更换算法，或者更换做数据预处理的方式。但是在课上，出于练习和探索的目的，我们继续调整我们的参数，让大家观察一下模型的变化，看看我们预测得是否正确。

依然按照参数对模型整体准确率的影响程度进行调参。

9. **调整min_samples_leaf**

```
#调整min_samples_leaf

param_grid={'min_samples_leaf':np.arange(1, 1+10, 1)}

#对于min_samples_split和min_samples_leaf,一般是从他们的最小值开始向上增加10或20
#面对高维度高样本量数据，如果不放心，也可以直接+50，对于大型数据，可能需要200~300的范围
#如果调整的时候发现准确率无论如何都上不来，那可以放心大胆调一个很大的数据，大力限制模型的复杂度

rfc = RandomForestClassifier(n_estimators=39,random_state=90)
GS = GridSearchCV(rfc,param_grid,cv=10)
GS.fit(data.data,data.target)

GS.best_params_
GS.best_score_
```

可以看见，网格搜索返回了min_samples_leaf的最小值，并且模型整体的准确率还降低了，这和max_depth的情况一致，参数把模型向左推，但是模型的泛化误差上升了。在这种情况下，我们显然是不要把这个参数设置起来的，就让它默认就好了。

10. **不懈努力，继续尝试min_samples_split**

```
#调整min_samples_split

param_grid={'min_samples_split':np.arange(2, 2+20, 1)}

rfc = RandomForestClassifier(n_estimators=39
                             ,random_state=90
                            )
GS = GridSearchCV(rfc,param_grid,cv=10)
GS.fit(data.data,data.target)

GS.best_params_
GS.best_score_
```

和min_samples_leaf一样的结果，返回最小值并且模型整体的准确率降低了。

11. **最后尝试一下criterion**

```
#调整Criterion

param_grid = {'criterion':['gini', 'entropy']}

rfc = RandomForestClassifier(n_estimators=39,random_state=90)
GS = GridSearchCV(rfc,param_grid,cv=10)
GS.fit(data.data,data.target)

GS.best_params_
GS.best_score_
```

12. **调整完毕，总结出模型的最佳参数**

```
rfc = RandomForestClassifier(n_estimators=39,random_state=90)
score = cross_val_score(rfc,data.data,data.target,cv=10).mean()
score

score - score_pre
```

在整个调参过程之中，我们首先调整了n_estimators（无论如何都请先走这一步），然后调整max_depth，通过max_depth产生的结果，来判断模型位于复杂度-泛化误差图像的哪一边，从而选择我们应该调整的参数和调参的方向。如果感到困惑，也可以画很多学习曲线来观察参数会如何影响我们的准确率，选取学习曲线中单调的部分来放大研究（如同我们对n_estimators做的）。学习曲线的拐点也许就是我们一直在追求的，最佳复杂度对应的泛化误差最低点（也是方差和偏差的平衡点）。

网格搜索也可以一起调整多个参数，大家只要有时间，可以自己跑一下，看看网格搜索会给我们怎样的结果，有时候，它的结果比我们的好，有时候，我们手动调整的结果会比较好。当然了，我们的乳腺癌数据集非常完美，所以只需要调n_estimators一个参数就达到了随机森林在这个数据集上表现得极限。在泰坦尼克号案例的数据中，我们使用同样的方法调出了如下的参数组合。

```
rfc = RandomForestClassifier(n_estimators=68
                             ,random_state=90
                             ,criterion="gini"
                             ,min_samples_split=8
                             ,min_samples_leaf=1
                             ,max_depth=12
                             ,max_features=2
                             ,max_leaf_nodes=36
                             )
```