

菜菜的scikit-learn课堂02

随机森林在sklearn中的实现

小伙伴们晚上好~o(￣▽￣)ブ

我是菜菜，这里是我的sklearn课堂第二期，今晚的直播内容是随机森林在sklearn中的实现和调参~

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 0.20.0 （你的版本至少要0.19

Numpy 1.15.3, **Pandas** 0.23.4, **Matplotlib** 3.0.1, **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~



CDA 数据分析师[®]
CERTIFIED DATA ANALYST

菜菜的scikit-learn课堂02

随机森林在sklearn中的实现

随机森林

1 概述

1.1 集成算法概述

1.2 sklearn中的集成算法

2 RandomForestClassifier

2.1 重要参数

2.1.1 控制基评估器的参数

2.1.2 n_estimators

2.1.3 random_state

2.1.4 bootstrap & oob_score

2.2 重要属性和接口

Bonus: Bagging的另一个必要条件

3 RandomForestRegressor

3.1 重要参数，属性与接口

criterion

重要属性和接口

3.2 实例：用随机森林回归填补缺失值

4 机器学习中调参的基本思想

5 实例：随机森林在乳腺癌数据上的调参

6 附录

6.1 Bagging vs Boosting

6.2 RFC的参数列表

6.3 RFC的属性列表

6.4 RFC的接口列表

随机森林

1 概述

1.1 集成算法概述

集成学习（ensemble learning）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多个模型，集成所有模型的建模结果。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。

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。**

多个模型集成成为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——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。

1.2 sklearn中的集成算法

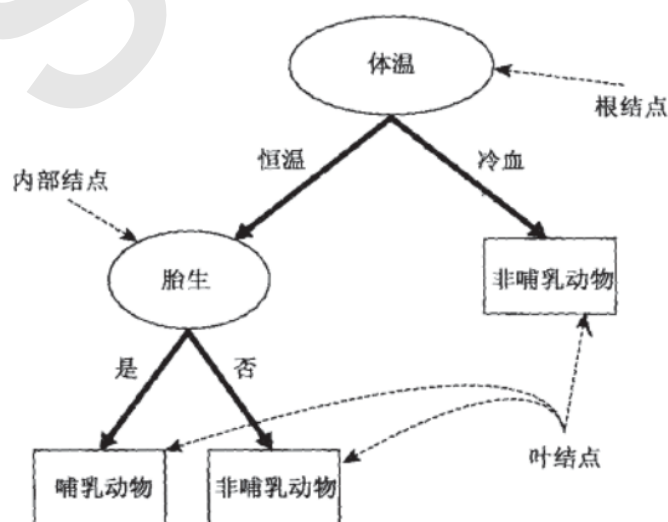
- sklearn中的集成算法模块ensemble

类	类的功能
ensemble.AdaBoostClassifier	AdaBoost分类
ensemble.AdaBoostRegressor	Adaboost回归
ensemble.BaggingClassifier	装袋分类器
ensemble.BaggingRegressor	装袋回归器
ensemble.ExtraTreesClassifier	Extra-trees分类（超树，极端随机树）
ensemble.ExtraTreesRegressor	Extra-trees回归
ensemble.GradientBoostingClassifier	梯度提升分类
ensemble.GradientBoostingRegressor	梯度提升回归
ensemble.IsolationForest	隔离森林
ensemble.RandomForestClassifier	随机森林分类
ensemble.RandomForestRegressor	随机森林回归
ensemble.RandomTreesEmbedding	完全随机树的集成
ensemble.VotingClassifier	用于不合适估算器的软投票/多数规则分类器

集成算法中，有一半以上都是树的集成模型，可以想见决策树在集成中必定是有很好的效果。在这堂课中，我们会以随机森林为例，慢慢为大家揭开集成算法的神秘面纱。

• 复习：sklearn中的决策树

在开始随机森林之前，我们先复习一下决策树。决策树是一种原理简单，应用广泛的模型，它可以同时被用于分类和回归问题。决策树的主要功能是从一张有特征和标签的表格中，通过对特定特征进行提问，为我们总结出一系列决策规则，并用树状图来呈现这些决策规则。



决策树的核心问题有两个，一个是如何找出正确的特征来进行提问，即如何分枝，二是树生长到什么时候应该停下。

对于第一个问题，我们定义了用来衡量分枝质量的指标不纯度，分类树的不纯度用基尼系数或信息熵来衡量，回归树的不纯度用MSE均方误差来衡量。每次分枝时，决策树对所有的特征进行不纯度计算，选取不纯度最低的特征进行分枝，分枝后，又再对被分枝的不同取值下，计算每个特征的不纯度，继续选取不纯度最低的特征进行分枝。



每分枝一层，树整体的不纯度会越来越小，决策树追求的是最小不纯度。因此，决策树会一致分枝，直到没有更多的特征可用，或整体的不纯度指标已经最优，决策树就会停止生长。

决策树非常容易过拟合，这是说，它很容易在训练集上表现优秀，却在测试集上表现很糟糕。为了防止决策树的过拟合，我们要对决策树进行剪枝，sklearn中提供了大量的剪枝参数，我们一会儿会带大家复习一下。

2 RandomForestClassifier

```
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)
```

随机森林是非常具有代表性的Bagging集成算法，它的所有基评估器都是决策树，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。

2.1 重要参数

2.1.1 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

11月7日进行的直播sklearn中的决策树中，有对以上所有参数的详细解释，大家可以进群领取课件，阅读课件中的内容，也可以回看直播，或直接在附录中查看这些参数的解释。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。

2.1.2 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

```
%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.data
wine.target
```

3. 复习:sklearn建模的基本流程

```
from sklearn.model_selection import train_test_split

Xtrain, Xtest, Ytrain, Ytest = train_test_split(wine.data, wine.target, test_size=0.3)

clf = DecisionTreeClassifier(random_state=0)
rfc = RandomForestClassifier(random_state=0)
clf = clf.fit(Xtrain, Ytrain)
rfc = rfc.fit(Xtrain, Ytrain)
score_c = clf.score(Xtest, Ytest)
score_r = rfc.score(Xtest, Ytest)

print("Single Tree:{}".format(score_c)
      , "Random Forest:{}".format(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 = "Decision Tree")
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)

plt.plot(range(1,11),rfc_l,label = "Random Forest")
plt.plot(range(1,11),clf_l,label = "Decision Tree")
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: 2mins 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？

2.1.3 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20棵树以上都判断错误的可能性是：

$$e_{random_forest} = \sum_{i=13}^{25} C_{25}^i \epsilon^i (1 - \epsilon)^{25-i} = 0.000369$$

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， $(1-\epsilon)$ 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，很多小伙伴可能就会想到了：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。

```
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

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。**

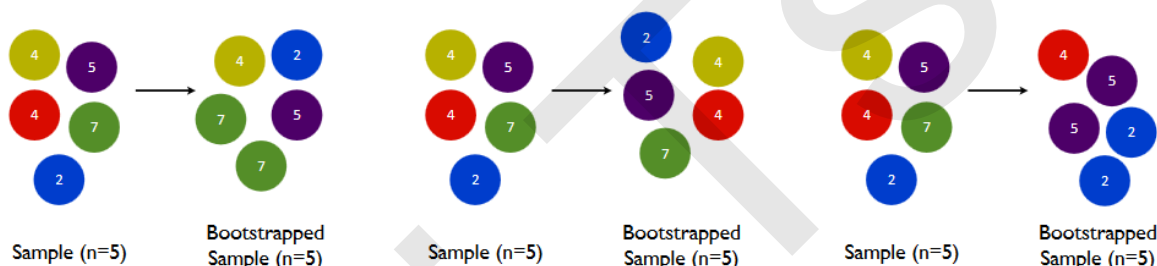
但这种做法的局限性是很强的，当我们需要成千上万棵树的时候，数据不一定能够提供成千上万的特征来让我们构筑尽量多尽量不同的树。因此，除了random_state。我们还需要其他的随机性。

2.1.4 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：

$$1 - \left(1 - \frac{1}{n}\right)^n \rightarrow \text{永远不会被抽到的概率}$$

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, 简称为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_
```

2.2 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`, `bootstrap`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。

#大家可以分别取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（比如上周完整课案例中的泰坦尼克号数据）。

Bonus: 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)

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。

3 RandomForestRegressor

```
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。

3.1 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，fi是模型回归出的数值，yi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：

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

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

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，fi是模型回归出的数值，yi是样本点i实际的数值标签。y帽是真实数值标签的平均数。R平方可以为正为负（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平方就会为负），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")

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平方，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。

3.2 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
#如果能够创建一个数组，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()
```

4 机器学习中调参的基本思想

在准备这一套课程的时候，我发现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。

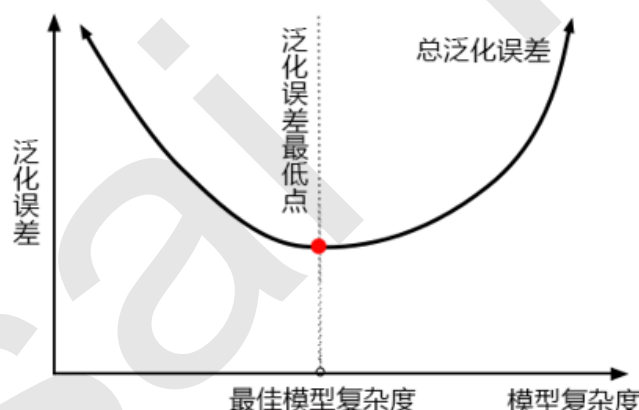
通过画学习曲线，或者网格搜索，我们能够探索到调参边缘（代价可能是训练一次模型要跑三天三夜），但是在现实中，高手调参恐怕还是多依赖于经验，而这些经验，来源于：1) 非常正确的调参思路和方法，2) 对模型评估指标的理解，3) 对数据的感觉和经验，4) 用洪荒之力去不断地尝试。

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（Generalization error）。

• 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”，原理十分复杂，无论你翻开哪一本书，你都会看见长篇的数学论证和每个字都能看懂但是连在一起就看不懂的文字解释。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 <i>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</i>	☆
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。

• 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(bias)和噪声(ϵ)共同决定。

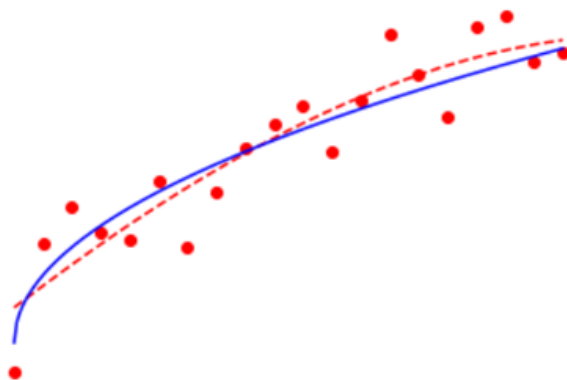
$$E(f; D) = bias^2(x) + var(x) + \epsilon^2$$

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。

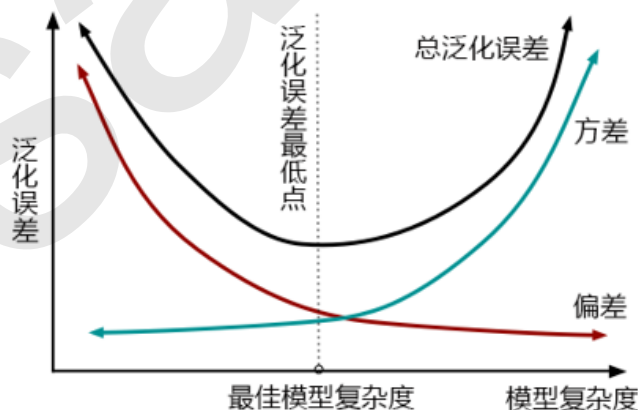
方差：反映的是模型每一次输出结果与模型预测值的平均水平之间的误差，即每一个红点到红色虚线的距离，衡量模型的稳定性。模型越稳定，方差越低。



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次太深的学习，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

5 实例：随机森林在乳腺癌数据上的调参

在这节课中，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。

案例中，往往使用真实数据，为什么我们要使用sklearn自带的的数据呢？因为真实数据在随机森林下的调参过程，往往非常缓慢。真实数据量大，维度高，在使用随机森林之前需要一系列的处理，因此不太适合用来做直播中的案例演示。在本章，我为大家准备了kaggle上下载的辨别手写数字的数据，有4W多条记录700多个左右的特征，随机森林在这个辨别手写数字的数据上有非常好的表现，其调参案例也是非常经典，但是由于数据的维度太高，太过复杂，运行一次完整的网格搜索需要四五个小时，因此不太可能拿来给大家进行演示。我们上周的案例中用的泰坦尼克号数据，用来调参的话也是需要很长时间，因此我才选择sklearn当中自带的，结构相对清晰简单的数据来为大家做这个案例。大家感兴趣的话，可以进群去下载数据，也可以直接到kaggle上进行下载，数据集名称是Digit Recognizer (<https://www.kaggle.com/c/digit-recognizer>)。

那我们接下来，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)
print(max(scorel),(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：1) 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。2) 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个叶子一个区间，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)}
```

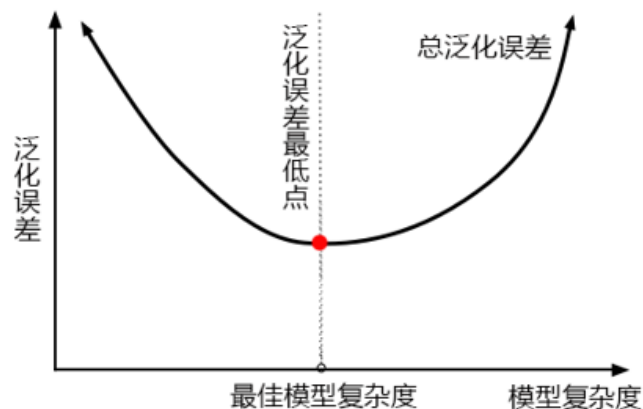
一般根据数据的大小来进行一个试探，乳腺癌数据很小，所以可以采用1~10，或者1~20这样的试探
但对于像digit recognition那样的大型数据来说，我们应该尝试30~50层深度（或许还不够）
更应该画出学习曲线，来观察深度对模型的影响

```
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。

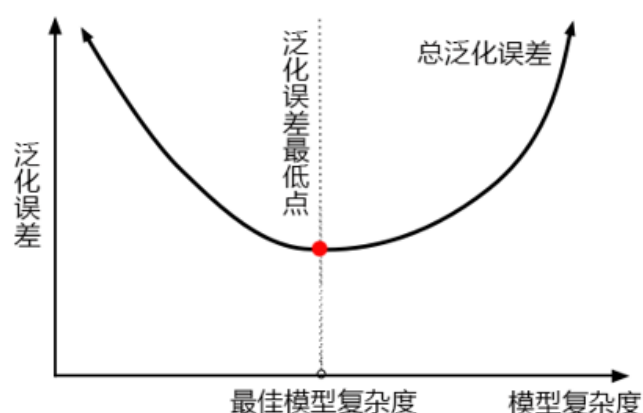
```
"""
```

```
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。在我们上周使用的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
                             )
```

基于泰坦尼克号数据调整出来的参数，数据的处理过程请参考上一期的完整视频。这个组合的准确率达到了83.915%，比单棵决策树提升了大约7%，比调参前的随机森林提升了2.02%，这对于调参来说其实是一个非常巨大的进步。不过，泰坦尼克号数据的运行缓慢，大家量力而行，可以试试看用复杂度-泛化误差方法（方差-偏差方法）来解读一下这个调参结果和过程。

6 附录

6.1 Bagging vs Boosting

	装袋法 Bagging	提升法 Boosting
评估器	相互独立，同时运行	相互关联，按顺序依次构建，后建的模型会在先建模型预测失败的样本上有更多的权重
抽样数集	有放回抽样	有放回抽样，但会确认数据的权重，每次抽样都会给容易预测失败的样本更多的权重
决定集成的结果	平均或少数服从多数原则	加权平均，在训练集上表现更好的模型会有更大的权重
目标	降低方差，提高模型整体的稳定性	降低偏差，提高模型整体的精确度
单个评估器存在过拟合问题的时候	能够一定程度上解决过拟合问题	可能会加剧过拟合问题
单个评估器的效力比较弱的时候	不是非常有帮助	很可能会提升模型表现
代表算法	随机森林	梯度提升树，Adaboost

6.2 RFC的参数列表

n_estimators	<p>整数，可不填，默认10</p> <p>随机森林中树模型的数量。</p> <p><i>注意：在0.22版本中，n_estimators的默认值即将被改为100，在0.20版本中依旧为10</i></p>
criterion	<p>字符型，可不填，默认基尼系数 ("gini")</p> <p>用来衡量分枝质量的指标，即衡量不纯度的指标 输入"gini"使用基尼系数，或输入"entropy"使用信息增益 (Information Gain)</p>
max_depth	<p>整数或None，可不填，默认None</p> <p>树的最大深度。如果是None，树会持续生长直到所有叶子节点的不纯度为0，或者直到每个叶子节点所含的样本量都小于参数min_samples_split中输入的数字</p>
min_samples_split	<p>整数或浮点数，可不填，默认=2</p> <p>一个中间节点要分枝所需的最小样本量。如果一个节点包含的样本量小于min_samples_split中填写的数字，这个节点的分枝就不会发生，也就是说，这个节点一定会成为一个叶子节点</p> <ol style="list-style-type: none"> 1) 如果输入整数，则认为输入的数字是分枝所需的最小样本量 2) 如果输入浮点数，则认为输入的浮点数是比例，输入的浮点数*输入模型的数据集的样本量 (n_samples) 是分枝所需的最小样本量 <p><i>浮点数功能是0.18版本以上的sklearn才可以使用</i></p>
min_sample_leaf	<p>整数或浮点数，可不填，默认=1</p> <p>一个叶节点要存在所需的最小样本量。一个节点在分枝后的每个子节点中，必须要包含至少min_sample_leaf个训练样本，否则分枝就不会发生。这个参数可能会有着使模型更平滑的效果，尤其是在回归中</p> <ol style="list-style-type: none"> 1) 如果输入整数，则认为输入的数字是叶节点存在所需的最小样本量 2) 如果输入浮点数，则认为输入的浮点数是比例，输入的浮点数*输入模型的数据集的样本量 (n_samples) 是叶节点存在所需的最小样本量
min_weight_fraction_leaf	<p>浮点数，可不填，默认=0.</p> <p>一个叶节点要存在所需的权重占输入模型的数据集的总权重的比例。</p> <p>总权重由fit接口中的sample_weight参数确定，当sample_weight是None时，默认所有样本的权重相同</p>
max_features	<p>整数，浮点数，字符型或None，可不填，默认None</p> <p>在做最佳分枝的时候，考虑的特征个数</p> <ol style="list-style-type: none"> 1) 输入整数，则每一次分枝都考虑max_features个特征 2) 输入浮点数，则认为输入的浮点数是比例，每次分枝考虑的特征数目是max_features输入模型的数据集的特征个数(n_features) 3) 输入 "auto"，采用n_features的平方根作为分枝时考虑的特征数目 4) 输入 "sqrt"，采用n_features的平方根作为分枝时考虑的特征数目 5) 输入 "log2"，采用$\log_2(n_features)$作为分枝时考虑的特征数目 6) 输入 "None"，n_features就是分枝时考虑的特征数目 <p><i>注意：如果在限制的max_features中，决策树无法找到节点样本上至少一个有效的分枝，那对分枝的搜索不会停止，决策树将会检查比限制的max_features数目更多的特征</i></p>

max_leaf_nodes	<p>整数或None，可不填，默认None</p> <p>最大叶节点数量。在最佳分枝方式下，以max_leaf_nodes为限制来生长树。如果是None，则没有叶节点数量的限制。</p>
min_impurity_decrease	<p>浮点数，可以不填，默认=0.</p> <p>当一个节点的分枝后引起的不纯度的降低大于或等于min_impurity_decrease中输入的数值，则这个分枝则会被保留，不会被剪枝。</p> <p>带权重的不纯度下降可以表示为：</p> $\frac{N_t}{N} * (\text{不纯度} - \frac{N_{tR}}{N_t} * \text{右侧树枝的不纯度} - \frac{N_{tL}}{N_t} * \text{左侧树枝的不纯度})$ <p>中N是样本总量，N_t是节点t中的样本量，N_t_L是左侧子节点的样本量，N_t_R是右侧子节点的样本量</p> <p>注意：如果sample_weight在fit接口中有值，则N，N_t，N_t_R，N_t_L都是指样本量的权重，而非单纯的样本数量</p> <p>仅在0.19以上版本中提供此功能</p>
min_impurity_split	<p>浮点数</p> <p>防止树生长的阈值之一。如果一个节点的不纯度高于min_impurity_split，这个节点就会被分枝，否则的话这个节点就只能是叶子节点。</p> <p>在0.19以上版本中，这个参数的功能已经被min_impurity_decrease取代，在0.21版本中这个参数将会被删除，请使用min_impurity_decrease</p>
bootstrap	<p>布尔值，可不填，默认True</p> <p>在建树过程中，是否使用自举样本抽样的方式。</p>
oob_score	<p>布尔值，默认False</p> <p>在建树时，是否适用袋外样本来预测模型的泛化精确性</p>
n_jobs	<p>整数或None，可不填，默认None</p> <p>训练(fit)和预测(predict)并行运行的作业数。None表示1除非None是标注在参数joblib.parallel_backend context中，-1表示使用整个处理器来运行。</p> <p>更多详细信息请参阅： https://scikit-learn.org/stable/glossary.html#term-n-jobs </p>
random_state	<p>整数，sklearn中设定好的RandomState实例，或None，可不填，默认None</p> <ol style="list-style-type: none"> 1) 输入整数，random_state是由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模块中的一个RandomState实例
verbose	<p>整数，可不填，默认0</p> <p>在拟合和预测时控制树的复杂度。</p>
warm_start	<p>布尔值，可不填，默认False</p> <p>设置为True时，使用上一次实例化中得到的树模型来fit并以此向整体添加更多估算器，否则，重新建立一棵树来进行训练。</p>

class_weight	<p>字典，字典的列表，“balanced”，“balanced_subsample”或者“None”，默认None与标签相关联的权重，表现方式是{标签的值：权重}。如果为None，则默认所有的标签持有相同的权重。对于多输出问题，字典中权重的顺序需要与各个y在标签数据集中的排列顺序相同</p> <p>注意，对于多输出问题（包括多标签问题），定义的权重必须具体到每个标签下的每个类，其中类是字典键值对中的键，权重是键值对中的值。比如说，对于有四个标签，且每个标签是二分类（0和1）的分类问题而言，权重应该被表示为：</p> <p style="text-align: center;">[[0:1,1:1], {0:1,1:5}, {0:1, 1:1}, {0:1,1:1}]</p> <p>而不是：</p> <p style="text-align: center;">[{1:1}, {2:5}, {3:1}, {4:1}]</p> <p>如果使用“balanced”模式，将会使用y的值自动调整与输入数据中的类频率成反比的权重，比如 $\frac{n_{samples}}{n_{classes} * np.bincount(y)}$</p> <p>“balanced_subsample”模式与“balanced”相同，只是基于每个生长的树的随机放回抽样样本计算权重。</p> <p>对于多输出问题，每一列y的权重将被相乘</p> <p>注意：如果指定了sample_weight，这些权重将通过fit接口与sample_weight相乘</p>
---------------------	--

6.3 RFC的属性列表

estimators_	输出包含单个决策树分类器的列表，是所有训练过的基分类器的集合
classes_	输出一个数组(array)或者一个数组的列表(list)，结构为标签的数目(n_classes) 输出所有标签
feature_importances_	输出一个数组，结构为特征的数目(n_features) 返回每个特征的重要性，一般是这个特征在多次分枝中产生的信息增益的综合，也被称为“基尼重要性”(Gini Importance)
n_classes_	输出整数或列表 标签类别的数据
n_features_	在训练模型(fit)时使用的特征个数
n_outputs_	在训练模型(fit)时输出的结果的个数
oob_score_	输出浮点数，使用袋外数据来验证模型效益的分数
oob_decision_function_	根据袋外验证结果计算的决策函数。如果n_estimators非常小，那有可能在进行随机放回抽样的过程中没有数据掉落在袋外，在这种情况下，oob_decision_function_的结果会是NaN。

6.4 RFC的接口列表

apply (X[, check_input])	输入测试集或样本点，返回每个样本被分到的叶节点的索引 check_input是接口apply的参数，输入布尔值，默认True，通常不使用
decision_path(X[, check_input])	输入测试集或样本点，返回树中的决策树结构 Check_input同样是参数
fit(X, y[, sample_weight, check_input, ...])	训练模型的接口，其中X代表训练样本的特征，y代表目标数据，即标签，X和y都必须是类数组结构，一般我们都使用ndarray来导入 sample_weight是fit的参数，用来为样本标签设置权重，输入的格式是一个和测试集样本量一致长度的数字数组，数组中所带有的数字表示每个样本量所占的权重，数组中数字的综合代表整个测试集的权重总数 返回训练完毕的模型
get_params([deep])	布尔值，获取这个模型评估对象的参数。接口本身的参数deep，默认为True，表示返回此估计器的参数并包含作为估算器的子对象。 返回模型评估对象在实例化时的参数设置
predict(X[, check_input])	预测所提供的测试集X中样本点的标签，这里的测试集X必须和fit中提供的训练集结构一致 返回模型预测的测试样本的标签或回归值
predict_log_proba(X)	预测所提供的测试集X中样本点归属于各个标签的对数概率
predict_proba(X[, check_input])	预测所提供的测试集X中样本点归属于各个标签的概率 返回测试集中每个样本点对应的每个标签的概率，各个标签按词典顺序排序。预测的类概率是叶中相同类的样本的分数。
score(X, y[, sample_weight])	用给定测试数据和标签的平均准确度作为模型的评分标准，分数越高模型越好。其中X是测试集，y是测试集的真实标签。sample_weight是score的参数，用法与fit的参数一致 返回给定策树数据和标签的平均准确度，在多标签分类中，这个指标是子集精度。
set_params(**params)	可以为已经建立的评估器重设参数 返回重新设置的评估器本身