

菜菜的scikit-learn课堂第一期 CDA 数据分析师[®]

sklearn入门 & 决策树在sklearn中的实现

小伙伴们大家好~o(—▽—)ブ

我是菜菜，这里是我的sklearn课堂，本章内容是sklearn入门 + 决策树在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)

Graphviz 0.8.4 (没有画不出决策树哦，安装代码conda install python-graphviz)

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~



菜菜的scikit-learn课堂第一期

sklearn入门 & 决策树在sklearn中的实现

sklearn入门

决策树

1 概述

1.1 决策树是如何工作的

1.2 sklearn中的决策树

2 DecisionTreeClassifier与红酒数据集

2.1 重要参数

2.1.1 criterion

2.1.2 random_state & splitter

2.1.3 剪枝参数

2.1.4 目标权重参数

2.2 重要属性和接口

3 DecisionTreeRegressor

3.1 重要参数, 属性及接口

criterion

3.2 实例：一维回归的图像绘制

4 实例：泰坦尼克号幸存者的预测

5 决策树的优缺点

6 附录

6.1 分类树参数列表

6.2 分类树属性列表

6.3 分类树接口列表

Bonus Chapter I 实例：分类树在合成数集上的表现

Bonus Chapter II: 配置开发环境&安装sklearn

sklearn入门

scikit-learn，又写作sklearn，是一个开源的基于python语言的机器学习工具包。它通过NumPy, SciPy和Matplotlib等python数值计算的库实现高效的算法应用，并且涵盖了几乎所有主流机器学习算法。

<http://scikit-learn.org/stable/index.html>

在工程应用中，用python手写代码来从头实现一个算法的可能性非常低，这样不仅耗时耗力，还不一定能够写出构架清晰，稳定性强的模型。更多情况下，是分析采集到的数据，根据数据特征选择适合的算法，在工具包中调用算法，调整算法的参数，获取需要的信息，从而实现算法效率和效果之间的平衡。而sklearn，正是这样一个可以帮助我们高效实现算法应用的工具包。

sklearn有一个完整而丰富的官网，里面讲解了基于sklearn对所有算法的实现和简单应用。然而，这个官网是全英文的，并且现在没有特别理想的中文接口，市面上也没有针对sklearn非常好的书。因此，这门课的目的就是由简向繁地向大家解析sklearn的全面应用，帮助大家了解不同的机器学习算法有哪些可调参数，有哪些可用接口，这些接口和参数对算法来说有什么含义，又会对算法的性能及准确性有什么影响。我们会讲解sklearn中对算法的说明，调参，属性，接口，以及实例应用。注意，本门课程的讲解不会涉及详细的算法原理，只会专注于算法在sklearn中的实现，如果希望详细了解算法的原理，建议阅读下面这两本书：

数据挖掘导论



作者: (美)Pang-Ning Tan / Michael Steinbach / Vipin Kumar
ar
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

决策树

1 概述

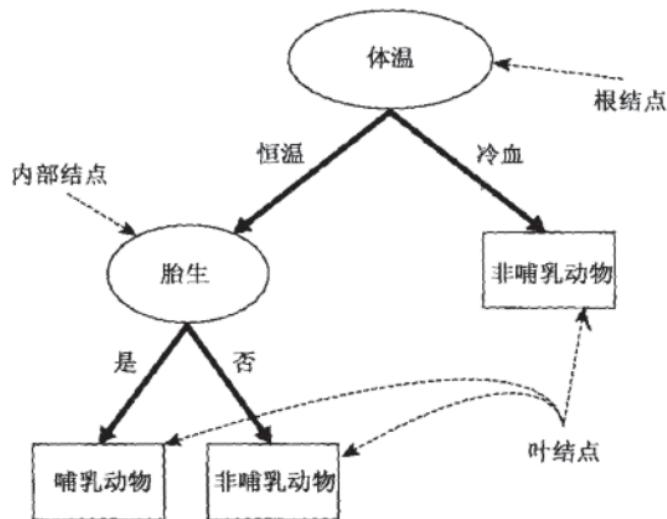
1.1 决策树是如何工作的

决策树（Decision Tree）是一种非参数的有监督学习方法，它能够从一系列有特征和标签的数据中总结出决策规则，并用树状图的结构来呈现这些规则，以解决分类和回归问题。决策树算法容易理解，适用各种数据，在解决各种问题时都有良好表现，尤其是以树模型为核心的各种集成算法，在各个行业和领域都有广泛的应用。

我们来简单了解一下决策树是如何工作的。决策树算法的本质是一种图结构，我们只需要问一系列问题就可以对数据进行分类了。比如说，来看看下面这组数据集，这是一系列已知物种以及所属类别的数据：

名字	体温	表皮覆盖	胎生	水生动物	飞行动物	有腿	冬眠	类标号
人类	恒温	毛发	是	否	否	是	否	哺乳类
鲑鱼	冷血	鳞片	否	是	否	否	否	鱼类
鲸	恒温	毛发	是	是	否	否	否	哺乳类
青蛙	冷血	无	否	半	否	是	是	两栖类
巨蜥	冷血	鳞片	否	否	是	是	否	爬行类
蝙蝠	恒温	毛发	是	否	是	是	是	哺乳类
鸽子	恒温	羽毛	否	否	是	是	否	鸟类
猫	恒温	软毛	是	否	否	是	否	哺乳类
豹纹鲨	冷血	鳞片	是	否	否	否	否	鱼类
海龟	冷血	鳞片	否	半	否	是	否	爬行类
企鹅	恒温	羽毛	否	半	否	是	是	鸟类
豪猪	恒温	刚毛	是	否	否	是	否	哺乳类
鳗	冷血	鳞片	否	是	否	否	是	鱼类
蝾螈	冷血	无	否	半	否	否	是	两栖类

我们现在的目标是，将动物们分为哺乳类和非哺乳类。那根据已经收集到的数据，决策树算法为我们算出了下面的这棵决策树：



假如我们现在发现了一种新物种Python，它是冷血动物，体表带鳞片，并且不是胎生，我们就可以通过这棵决策树来判断它的所属类别。

可以看出，在这个决策过程中，我们一直在对记录的特征进行提问。最初的问题所在的地方叫做**根节点**，在得到结论前的每一个问题都是**中间节点**，而得到的每一个结论（动物的类别）都叫做**叶子节点**。

关键概念：节点

根节点：没有进边，有出边。包含最初的，针对特征的提问。

中间节点：既有进边也有出边，进边只有一条，出边可以有很多条。都是针对特征的提问。

叶子节点：有进边，没有出边，**每个叶子节点都是一个类别标签**。

*子节点和父节点：在两个相连的节点中，更接近根节点的是父节点，另一个是子节点。

决策树算法的核心是要解决两个问题：

- 1) 如何从数据表中找出最佳节点和最佳分枝？
- 2) 如何让决策树停止生长，防止过拟合？

几乎所有决策树有关的模型调整方法，都围绕这两个问题展开。这两个问题背后的原理十分复杂，我们会在讲解模型参数和属性的时候为大家简单解释涉及到的部分。在这门课中，我会尽量避免让大家太过深入到决策树复杂的原理和数学公式中（尽管决策树的原理相比其他高级的算法来说是非常简单的），这门课会专注于实践和应用。如果大家希望理解更深入的细节，建议大家在听这门课之前还是先去阅读和学习一下决策树的原理。

1.2 sklearn中的决策树

- 模块sklearn.tree

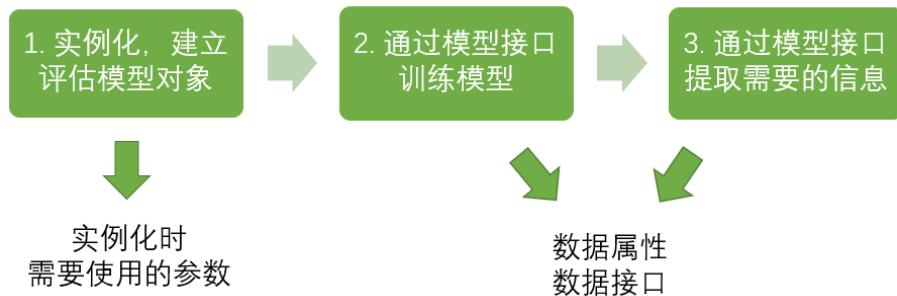
sklearn中决策树的类都在“tree”这个模块之下。这个模块总共包含五个类：

<code>tree.DecisionTreeClassifier</code>	分类树
<code>tree.DecisionTreeRegressor</code>	回归树
<code>tree.export_graphviz</code>	将生成的决策树导出为DOT格式，画图专用
<code>tree.ExtraTreeClassifier</code>	高随机版本的分类树
<code>tree.ExtraTreeRegressor</code>	高随机版本的回归树

我们会主要讲解分类树和回归树，并用图像呈现给大家。

- sklearn的基本建模流程

在那之前，我们先来了解一下sklearn建模的基本流程。



在这个流程下，分类树对应的代码是：

```

from sklearn import tree          # 导入需要的模块

clf = tree.DecisionTreeClassifier() # 实例化
clf = clf.fit(x_train,y_train)    # 用训练集数据训练模型
result = clf.score(x_test,y_test) # 导入测试集，从接口中调用需要的信息
  
```

2 DecisionTreeClassifier与红酒数据集

```

class sklearn.tree.DecisionTreeClassifier(criterion='gini', splitter='best', max_depth=None,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None,
random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None,
class_weight=None, presort=False)
  
```

2.1 重要参数

2.1.1 criterion

为了要将表格转化为一棵树，决策树需要找出最佳节点和最佳的分枝方法，对分类树来说，衡量这个“最佳”的指标叫做“不纯度”。通常来说，不纯度越低，决策树对训练集的拟合越好。现在使用的决策树算法在分枝方法上的核心大多是围绕在对某个不纯度相关指标的最优化上。

不纯度基于节点来计算，树中的每个节点都会有一个不纯度，并且子节点的不纯度一定是低于父节点的，也就是说，在同一棵决策树上，叶子节点的不纯度一定是最低的。

Criterion这个参数正是用来决定不纯度的计算方法的。sklearn提供了两种选择：

- 1) 输入"entropy"，使用**信息熵** (Entropy)
- 2) 输入"gini"，使用**基尼系数** (Gini Impurity)

$$\begin{aligned}
 Entropy(t) &= - \sum_{i=0}^{c-1} p(i|t) \log_2 p(i|t) \\
 Gini(t) &= 1 - \sum_{i=0}^{c-1} p(i|t)^2
 \end{aligned}$$

其中 t 代表给定的节点， i 代表标签的任意分类， $p(i|t)$ 代表标签分类*i*在节点 t 上所占的比例。注意，当使用信息熵时，sklearn实际计算的是基于信息熵的信息增益(Information Gain)，即父节点的信息熵和子节点的信息熵之差。

比起基尼系数，信息熵对不纯度更加敏感，对不纯度的惩罚最强。但是在实际使用中，信息熵和基尼系数的效果基本相同。信息熵的计算比基尼系数缓慢一些，因为基尼系数的计算不涉及对数。另外，因为信息熵对不纯度更加敏感，所以信息熵作为指标时，决策树的生长会更加“精细”，因此对于高维数据或者噪音很多的数据，信息熵很容易过拟合，基尼系数在这种情况下效果往往比较好。当模型拟合程度不足的时候，即当模型在训练集和测试集上都表现不太好的时候，使用信息熵。当然，这些不是绝对的。

参数	criterion
如何影响模型？	确定不纯度的计算方法，帮忙找出最佳节点和最佳分枝，不纯度越低，决策树对训练集的拟合越好
可能的输入有哪些？	不填默认基尼系数，填写gini使用基尼系数，填写entropy使用信息增益
怎样选取参数？	通常就使用基尼系数 数据维度很大，噪音很大时使用基尼系数 维度低，数据比较清晰的时候，信息熵和基尼系数没区别 当决策树的拟合程度不够的时候，使用信息熵 两个都试试，不好就换另外一个

到这里，决策树的基本流程其实可以简单概括如下：



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。

- 建立一棵树

- 导入需要的算法库和模块

```

from sklearn import tree
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split

```

- 探索数据

```
wine = load_wine()

wine.data.shape

wine.target

#如果wine是一张表，应该长这样：
import pandas as pd
pd.concat([pd.DataFrame(wine.data),pd.DataFrame(wine.target)],axis=1)

wine.feature_names
wine.target_names
```

3. 分训练集和测试集

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

Xtrain.shape
Xtest.shape
```

4. 建立模型

```
clf = tree.DecisionTreeClassifier(criterion="entropy")
clf = clf.fit(Xtrain, Ytrain)
score = clf.score(Xtest, Ytest) #返回预测的准确度

score
```

5. 画出一棵树吧

```
feature_name = ['酒精', '苹果酸', '灰', '灰的碱性', '镁', '总酚', '类黄酮', '非黄烷类酚类', '花青素', '颜色强度', '色调', 'od280/od315稀释葡萄酒', '脯氨酸']

import graphviz
dot_data = tree.export_graphviz(clf
                                ,out_file = None
                                ,feature_names= feature_name
                                ,class_names=["琴酒", "雪莉", "贝尔摩德"]
                                ,filled=True
                                ,rounded=True
                                )
graph = graphviz.Source(dot_data)
graph
```

6. 探索决策树

```
#特征重要性
clf.feature_importances_

[*zip(feature_name,clf.feature_importances_)]
```

我们已经在只了解一个参数的情况下，建立了一棵完整的决策树。但是回到步骤4建立模型，score会在某个值附近波动，引起步骤5中画出来的每一棵树都不一样。它为什么会不稳定呢？如果使用其他数据集，它还会不稳定吗？

我们之前提到过，无论决策树模型如何进化，在分枝上的本质都还是追求某个不纯度相关的指标的优化，而正如我们提到的，不纯度是基于节点来计算的，也就是说，决策树在建树时，是靠优化节点来追求一棵优化的树，但最优的节点能够保证最优的树吗？集成算法被用来解决这个问题：sklearn表示，既然一棵树不能保证最优，那就建更多的不同的树，然后从中取最好的。怎样从一组数据集中建不同的树？在每次分枝时，不从使用全部特征，而是随机选取一部分特征，从中选取不纯度相关指标最优的作为分枝用的节点。这样，每次生成的树也就不同了。

```
clf = tree.DecisionTreeClassifier(criterion="entropy", random_state=30)
clf = clf.fit(Xtrain, Ytrain)
score = clf.score(Xtest, Ytest) #返回预测的准确度

score
```

2.1.2 random_state & splitter

random_state用来设置分枝中的随机模式的参数，默认None，在高维度时随机性会表现更明显，低维度的数据（比如鸢尾花数据集），随机性几乎不会显现。输入任意整数，会一直长出同一棵树，让模型稳定下来。

splitter也是用来控制决策树中的随机选项的，有两种输入值，输入"best"，决策树在分枝时虽然随机，但是还是会优先选择更重要的特征进行分枝（重要性可以通过属性feature_importances_查看），输入"random"，决策树在分枝时会更加随机，树会因为含有更多的不必要信息而更深更大，并因这些不必要信息而降低对训练集的拟合。这也是防止过拟合的一种方式。当你预测到你的模型会过拟合，用这两个参数来帮助你降低树建成之后过拟合的可能性。当然，树一旦建成，我们依然是使用剪枝参数来防止过拟合。

```
clf = tree.DecisionTreeClassifier(criterion="entropy"
                                  ,random_state=30
                                  ,splitter="random")
clf = clf.fit(Xtrain, Ytrain)
score = clf.score(Xtest, Ytest)

score

import graphviz
dot_data = tree.export_graphviz(clf
                                ,feature_names= feature_name
                                ,class_names=["琴酒","雪莉","贝尔摩德"]
                                ,filled=True
                                ,rounded=True
)
graph = graphviz.Source(dot_data)
graph
```

2.1.3 剪枝参数

在不加限制的情况下，一棵决策树会生长到衡量不纯度的指标最优，或者没有更多的特征可用为止。这样的决策树往往回过拟合，这就是说，**它会在训练集上表现很好，在测试集上却表现糟糕**。我们收集的样本数据不可能和整体的状况完全一致，因此当一棵决策树对训练数据有了过于优秀的解释性，它找出的规则必然包含了训练样本中的噪声，并使它对未知数据的拟合程度不足。

```
#我们的树对训练集的拟合程度如何?
score_train = clf.score(Xtrain, Ytrain)
score_train
```

为了让决策树有更好的泛化性，我们要对决策树进行剪枝。**剪枝策略对决策树的影响巨大，正确的剪枝策略是优化决策树算法的核心**。sklearn为我们提供了不同的剪枝策略：

- **max_depth**

限制树的最大深度，超过设定深度的树枝全部剪掉

这是用得最广泛的剪枝参数，在高维度低样本量时非常有效。决策树多生长一层，对样本量的需求会增加一倍，所以限制树深度能够有效地限制过拟合。在集成算法中也非常实用。实际使用时，建议从=3开始尝试，看看拟合的效果再决定是否增加设定深度。

- **min_samples_leaf & min_samples_split**

`min_samples_leaf`限定，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，或者，分枝会朝着满足每个子节点都包含`min_samples_leaf`个样本的方向去发生

一般搭配`max_depth`使用，在回归树中有神奇的效果，可以让模型变得更加平滑。这个参数的数量设置得太小会引起过拟合，设置得太大就会阻止模型学习数据。一般来说，建议从=5开始使用。如果叶节点中含有的样本量变化很大，建议输入浮点数作为样本量的百分比来使用。同时，这个参数可以保证每个叶子的最小尺寸，可以在回归问题中避免低方差，过拟合的叶子节点出现。对于类别不多的分类问题，=1通常就是最佳选择。

`min_samples_split`限定，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。

```
clf = tree.DecisionTreeClassifier(criterion="entropy"
                                  , random_state=30
                                  , splitter="random"
                                  , max_depth=3
                                  , min_samples_leaf=10
                                  , min_samples_split=10
                                  )

clf = clf.fit(Xtrain, Ytrain)

dot_data = tree.export_graphviz(clf
                               , feature_names= feature_name
                               , class_names=[ "琴酒", "雪莉", "贝尔摩德"]
                               , filled=True
                               , rounded=True
                               )

graph = graphviz.Source(dot_data)
```

```
graph
clf.score(Xtrain,Ytrain)
clf.score(Xtest,Ytest)
```

- **max_features & min_impurity_decrease**

一般max_depth使用，用作树的“精修”

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。和max_depth异曲同工，max_features是用来限制高维度数据的过拟合的剪枝参数，但其方法比较暴力，是直接限制可以使用的特征数量而强行使决策树停下的参数，在不知道决策树中的各个特征的重要性的情况下，强行设定这个参数可能会导致模型学习不足。如果希望通过降维的方式防止过拟合，建议使用PCA，ICA或者特征选择模块中的降维算法。

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。这是在0.19版本中更新的功能，在0.19版本之前时使用min_impurity_split。

- **确认最优的剪枝参数**

那具体怎么来确定每个参数填写什么值呢？这时候，我们就要使用确定超参数的曲线来进行判断了，继续使用我们已经训练好的决策树模型clf。超参数的学习曲线，是一条以超参数的取值为横坐标，模型的度量指标为纵坐标的曲线，它是用来衡量不同超参数取值下模型的表现的线。在我们建好的决策树里，我们的模型度量指标就是score。

```
import matplotlib.pyplot as plt

test = []
for i in range(10):
    clf = tree.DecisionTreeClassifier(max_depth=i+1
                                       ,criterion="entropy"
                                       ,random_state=30
                                       ,splitter="random")
    clf = clf.fit(Xtrain, Ytrain)
    score = clf.score(Xtest, Ytest)
    test.append(score)
plt.plot(range(1,11),test,color="red",label="max_depth")
plt.legend()
plt.show()
```

思考：

1. 剪枝参数一定能够提升模型在测试集上的表现吗？ - 调参没有绝对的答案，一切都是看数据本身。
2. 这么多参数，一个个画学习曲线？ - 在泰坦尼克号的案例中，我们会解答这个问题。

无论如何，剪枝参数的默认值会让树无尽地生长，这些树在某些数据集上可能非常巨大，对内存的消耗也非常巨大。所以如果你手中的数据集非常巨大，你已经预测到无论如何你都是要剪枝的，那提前设定这些参数来控制树的复杂性和大小会比较好。

2.1.4 目标权重参数

- **class_weight & min_weight_fraction_leaf**

完成样本标签平衡的参数。样本不平衡是指在一组数据集中，标签的一类天生占有很大的比例。比如说，在银行要判断“一个办了信用卡的人是否会违约”，就是是vs否（1%: 99%）的比例。这种分类状况下，即便模型什么也不做，全把结果预测成“否”，正确率也能有99%。因此我们要使用class_weight参数对样本标签进行一定的均衡，给少量的标签更多的权重，让模型更偏向少数类，向捕获少数类的方向建模。该参数默认None，此模式表示自动给与数据集中的所有标签相同的权重。

有了权重之后，样本量就不再是单纯地记录数目，而是受输入的权重影响了，因此这时候剪枝，就需要搭配min_weight_fraction_leaf这个基于权重的剪枝参数来使用。另请注意，基于权重的剪枝参数（例如min_weight_fraction_leaf）将比不知道样本权重的标准（比如min_samples_leaf）更少偏向主导类。如果样本是加权的，则使用基于权重的预修剪标准来更容易优化树结构，这确保叶节点至少包含样本权重的总和的一小部分。

2.2 重要属性和接口

属性是在模型训练之后，能够调用查看的模型的各种性质。对决策树来说，最重要的是feature_importances_，能够查看各个特征对模型的重要性。

sklearn中许多算法的接口都是相似的，比如说我们之前已经用到的fit和score，几乎对每个算法都可以使用。除了这两个接口之外，决策树最常用的接口还有apply和predict。apply中输入测试集返回每个测试样本所在的叶子节点的索引，predict输入测试集返回每个测试样本的标签。返回的内容一目了然并且非常容易，大家感兴趣可以自己下去试试看。

在这里不得不提的是，**所有接口中要求输入X_train和X_test的部分，输入的特征矩阵必须至少是一个二维矩阵。**
sklearn不接受任何一维矩阵作为特征矩阵被输入。如果你的数据的确只有一个特征，那必须用reshape(-1,1)来给矩阵增维；如果你的数据只有一个特征和一个样本，使用reshape(1,-1)来给你的数据增维。

```
#apply返回每个测试样本所在的叶子节点的索引
clf.apply(Xtest)
```

```
#predict返回每个测试样本的分类/回归结果
clf.predict(Xtest)
```

至此，我们已经学完了分类树DecisionTreeClassifier和用决策树绘图（export_graphviz）的所有基础。我们讲解了决策树的基本流程，分类树的八个参数，一个属性，四个接口，以及绘图所用的代码。

八个参数：Criterion，两个随机性相关的参数（random_state, splitter），五个剪枝参数（max_depth, min_samples_split, min_samples_leaf, max_feature, min_impurity_decrease）

一个属性：feature_importances_

四个接口：fit, score, apply, predict

有了这些知识，基本上分类树的使用大家都能够掌握了，接下来再到实例中去磨练就好。

3 DecisionTreeRegressor

```
class sklearn.tree.DecisionTreeRegressor (criterion='mse', splitter='best', max_depth=None,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None,
random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, presort=False)
```

几乎所有参数，属性及接口都和分类树一模一样。需要注意的是，在回归树中，没有标签分布是否均衡的问题，因此没有class_weight这样的参数。

3.1 重要参数，属性及接口

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

属性中最重要的依然是feature_importances_，接口依然是apply, fit, predict, score最核心。

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

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：

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

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

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， 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。

简单看看回归树是怎样工作的

```

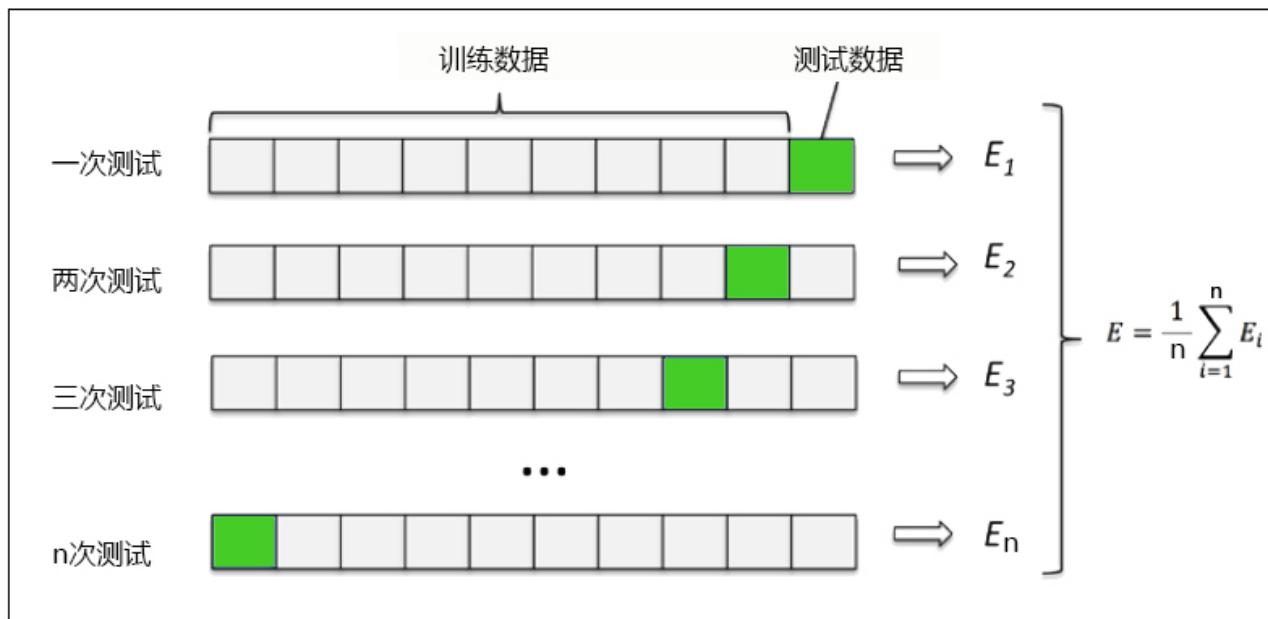
from sklearn.datasets import load_boston
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeRegressor

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

#交叉验证cross_val_score的用法

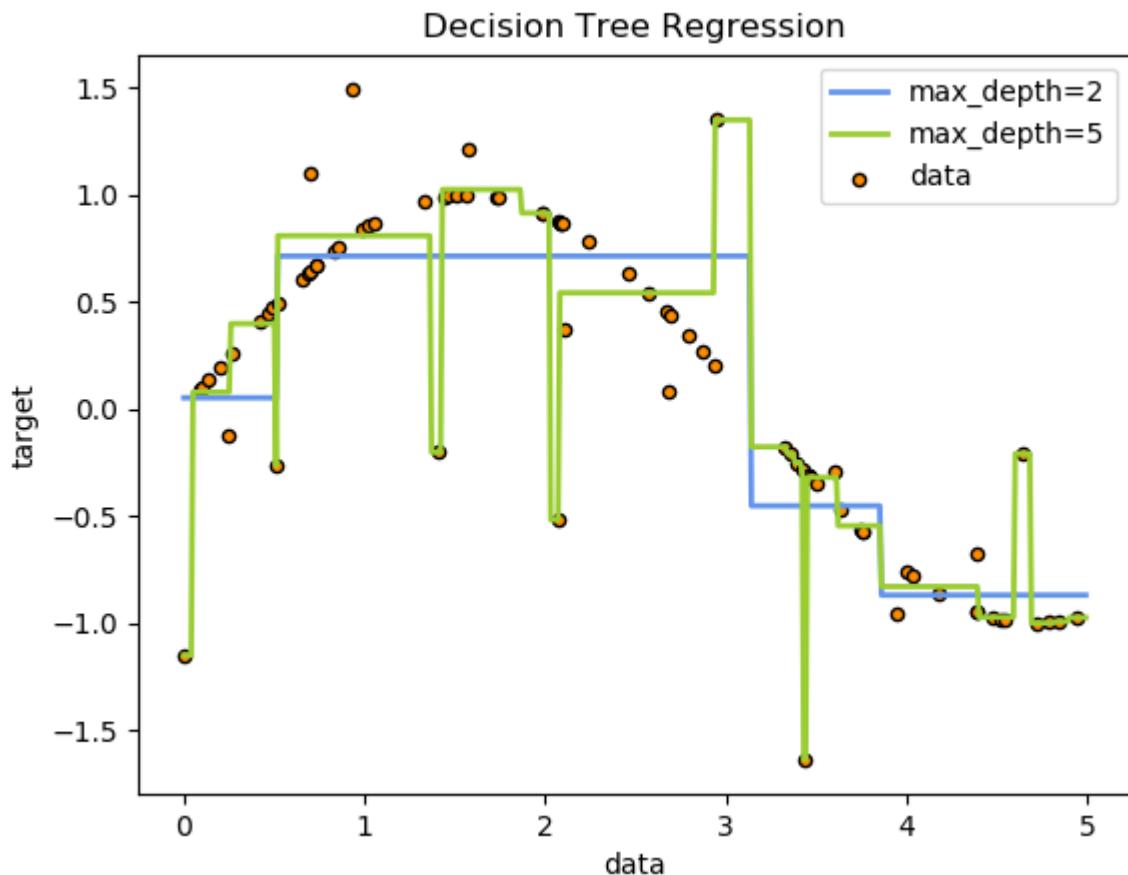
```

交叉验证是用来观察模型的稳定性的一种方法，我们将数据划分为n份，依次使用其中一份作为测试集，其他n-1份作为训练集，多次计算模型的精确性来评估模型的平均准确程度。训练集和测试集的划分会干扰模型的结果，因此用交叉验证n次的结果求出的平均值，是对模型效果的一个更好的度量。



3.2 实例：一维回归的图像绘制

接下来我们到二维平面上来观察决策树是怎样拟合一条曲线的。我们用回归树来拟合正弦曲线，并添加一些噪声来观察回归树的表现。



1. 导入需要的库

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor
import matplotlib.pyplot as plt
```

2. 创建一条含有噪声的正弦曲线

在这一步，我们的基本思路是，先创建一组随机的，分布在0~5上的横坐标轴的取值(x)，然后将这一组值放到sin函数中去生成纵坐标的值(y)，接着再到y上去添加噪声。全程我们会使用numpy库来为我们生成这个正弦曲线。

```
rng = np.random.RandomState(1)
x = np.sort(5 * rng.rand(80, 1), axis=0)
y = np.sin(x).ravel()
y[:5] += 3 * (0.5 - rng.rand(16))
```

#np.random.rand(数组结构), 生成随机数组的函数

```
#了解降维函数ravel()的用法
np.random.random((2,1))
np.random.random((2,1)).ravel()
np.random.random((2,1)).ravel().shape
```

3. 实例化&训练模型

```
regr_1 = DecisionTreeRegressor(max_depth=2)
regr_2 = DecisionTreeRegressor(max_depth=5)
regr_1.fit(X, y)
regr_2.fit(X, y)
```

4. 测试集导入模型，预测结果

```
x_test = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]
y_1 = regr_1.predict(x_test)
y_2 = regr_2.predict(x_test)

#np.arange(开始点, 结束点, 步长) 生成有序数组的函数

#了解增维切片np.newaxis的用法
l = np.array([1, 2, 3, 4])
l

l.shape

l[:, np.newaxis]

l[:, np.newaxis].shape

l[np.newaxis, :].shape
```

5. 绘制图像

```
plt.figure()
plt.scatter(X, y, s=20, edgecolor="black", c="darkorange", label="data")
plt.plot(x_test, y_1, color="cornflowerblue", label="max_depth=2", linewidth=2)
plt.plot(x_test, y_2, color="yellowgreen", label="max_depth=5", linewidth=2)
plt.xlabel("data")
plt.ylabel("target")
plt.title("Decision Tree Regression")
plt.legend()
plt.show()
```

可见，回归树学习了近似正弦曲线的局部线性回归。我们可以看到，如果树的最大深度（由max_depth参数控制）设置得太高，则决策树学习得太精细，它从训练数据中学了很多细节，包括噪声得呈现，从而使模型偏离真实的正弦曲线，形成过拟合。

4 实例：泰坦尼克号幸存者的预测

泰坦尼克号的沉没是世界上最严重的海难事故之一，今天我们通过分类树模型来预测一下哪些人可能成为幸存者。数据集来自<https://www.kaggle.com/c/titanic>，数据集会随着代码一起提供给大家，大家可以在下载页面拿到，或者到群中询问。数据集包含两个csv格式文件，data为我们接下来要使用的数据，test为kaggle提供的测试集。

接下来我们就来执行我们的代码。

1. 导入所需要的库

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
```

2. 导入数据集，探索数据

```
data = pd.read_csv(r"C:\work\Learnbetter\micro-class\week 1 DT\data\data.csv", index_col=0)

data.head()
data.info()
```

3. 对数据集进行预处理

```
#删除缺失值过多的列，和观察判断来说和预测的y没有关系的列
data.drop(["Cabin", "Name", "Ticket"], inplace=True, axis=1)

#处理缺失值，对缺失值较多的列进行填补，有一些特征只确实一两个值，可以采取直接删除记录的方法
data["Age"] = data["Age"].fillna(data["Age"].mean())
data = data.dropna()

#将分类变量转换为数值型变量

#将二分类变量转换为数值型变量
#astype能够将一个pandas对象转换为某种类型，和apply(int(x))不同，astype可以将文本类转换为数字，用这种方式可以很便捷地将二分类特征转换为0~1
data["Sex"] = (data["Sex"]=="male").astype("int")

#将三分类变量转换为数值型变量
labels = data["Embarked"].unique().tolist()
data["Embarked"] = data["Embarked"].apply(lambda x: labels.index(x))

#查看处理后的数据集
data.head()
```

4. 提取标签和特征矩阵，分测试集和训练集

```
x = data.iloc[:,data.columns != "Survived"]
y = data.iloc[:,data.columns == "Survived"]

from sklearn.model_selection import train_test_split
Xtrain, Xtest, Ytrain, Ytest = train_test_split(x,y,test_size=0.3)

#修正测试集和训练集的索引
for i in [Xtrain, Xtest, Ytrain, Ytest]:
    i.index = range(i.shape[0])

#查看分好的训练集和测试集
Xtrain.head()
```

5. 导入模型，粗略跑一下查看结果

```
clf = DecisionTreeClassifier(random_state=25)
clf = clf.fit(Xtrain, Ytrain)
score_ = clf.score(Xtest, Ytest)

score_

score = cross_val_score(clf,X,y,cv=10).mean()

score
```

6. 在不同max_depth下观察模型的拟合状况

```
tr = []
te = []
for i in range(10):
    clf = DecisionTreeClassifier(random_state=25
                                ,max_depth=i+1
                                ,criterion="entropy"
                                )
    clf = clf.fit(Xtrain, Ytrain)
    score_tr = clf.score(Xtrain,Ytrain)
    score_te = cross_val_score(clf,X,y,cv=10).mean()
    tr.append(score_tr)
    te.append(score_te)
print(max(te))
plt.plot(range(1,11),tr,color="red",label="train")
plt.plot(range(1,11),te,color="blue",label="test")
plt.xticks(range(1,11))
plt.legend()
plt.show()
```

#这里为什么使用“entropy”？因为我们注意到，在最大深度=3的时候，模型拟合不足，在训练集和测试集上的表现接近，但却都不是非常理想，只能达到83%左右，所以我们要使用entropy。

7. 用网格搜索调整参数

```
import numpy as np
gini_thresholds = np.linspace(0,0.5,20)

parameters = {'splitter':('best','random')
              , 'criterion':("gini","entropy")
              , "max_depth":[*range(1,10)]
              , 'min_samples_leaf':[*range(1,50,5)]
              , 'min_impurity_decrease':[*np.linspace(0,0.5,20)]}
}

clf = DecisionTreeClassifier(random_state=25)
GS = GridSearchCV(clf, parameters, cv=10)
GS.fit(Xtrain,Ytrain)

GS.best_params_
GS.best_score_
```

5 决策树的优缺点

决策树优点

1. 易于理解和解释，因为树木可以画出来被看见
2. 需要很少的数据准备。其他很多算法通常都需要数据规范化，需要创建虚拟变量并删除空值等。但请注意，sklearn中的决策树模块不支持对缺失值的处理。
3. 使用树的成本（比如说，在预测数据的时候）是用于训练树的数据点的数量的对数，相比于其他算法，这是一个很低的成本。
4. 能够同时处理数字和分类数据，既可以做回归又可以做分类。其他技术通常专门用于分析仅具有一种变量类型的数据集。
5. 能够处理多输出问题，即含有多个标签的问题，注意与一个标签中含有多种标签分类的问题区别开
6. 是一个白盒模型，结果很容易能够被解释。如果在模型中可以观察到给定的情况，则可以通过布尔逻辑轻松解释条件。相反，在黑盒模型中（例如，在人工神经网络中），结果可能更难以解释。
7. 可以使用统计测试验证模型，这让我们可以考虑模型的可靠性。
8. 即使其假设在某种程度上违反了生成数据的真实模型，也能够表现良好。

决策树的缺点

1. 决策树学习者可能创建过于复杂的树，这些树不能很好地推广数据。这称为过度拟合。修剪，设置叶节点所需的最小样本数或设置树的最大深度等机制是避免此问题所必需的，而这些参数的整合和调整对初学者来说会比较晦涩
2. 决策树可能不稳定，数据中微小的变化可能导致生成完全不同的树，这个问题需要通过集成算法来解决。
3. 决策树的学习是基于贪婪算法，它靠优化局部最优（每个节点的最优）来试图达到整体的最优，但这种做法不能保证返回全局最优决策树。这个问题也可以由集成算法来解决，在随机森林中，特征和样本会在分枝过程中被随机采样。
4. 有些概念很难学习，因为决策树不容易表达它们，例如XOR，奇偶校验或多路复用器问题。
5. 如果标签中的某些类占主导地位，决策树学习者会创建偏向主导类的树。因此，建议在拟合决策树之前平衡数据集。

6 附录

6.1 分类树参数列表

criterion	字符型, 可不填, 默认基尼系数 ("gini") 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)
splitter	字符型, 可不填, 默认最佳分枝 ("best") 确定每个节点的分枝策略 输入"best"使用最佳分枝, 或输入"random"使用最佳随机分枝
max_depth	整数或None, 可不填, 默认None 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
min_samples_split	整数或浮点数, 可不填, 默认=2 一个中间节点要分枝所需要的最小样本量。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 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 浮点数功能是0.18版本以上的sklearn才可以使用
min_sample_leaf	整数或浮点数, 可不填, 默认=1 一个叶节点要存在所需要的最小样本量。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 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_leaf	浮点数, 可不填, 默认=0. 一个叶节点要存在所需要的权重占输入模型的数据集的总权重的比例。 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
max_features	整数, 浮点数, 字符型或None, 可不填, 默认None 在做最佳分枝的时候, 考虑的特征个数 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 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
random_state	整数, sklearn中设定好的RandomState实例, 或None, 可不填, 默认None 1) 输入整数, random_state是由随机数生成器生成的随机数种子

	<p>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</p>
max_leaf_nodes	<p>整数或None，可不填，默认None 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. 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。 带权重的不纯度下降可以表示为：</p> $\frac{N_t}{N} * (\text{不纯度} - \frac{N_{tR}}{N_t} * \text{右测树枝的不纯度} - \frac{N_{tL}}{N_t} * \text{左侧树枝的不纯度})$ <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 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>
class_weight	<p>字典，字典的列表，“balanced”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> <pre>[{0:1,1:1}, {0:1,1:5}, {0:1, 1:1}, {0:1,1:1}]</pre> <p>而不是：</p> <pre>[{1:1}, {2:5}, {3:1}, {4:1}]</pre> <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>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>
presort	<p>布尔值，可不填，默认False 是否预先分配数据以加快拟合中最佳分枝的发现。在大型数据集上使用默认设置决策树时，</p>

将这个参数设置为true可能会延长训练过程，降低训练速度。当使用较小的数据集或限制树的深度时，设置这个参数为true可能会加快训练速度。

6.2 分类树属性列表

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)
max_features_	输出整数 参数max_features的推断值
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
tree_	输出一个可以导出建好的树结构的端口，通过这个端口，可以访问树的结构和低级属性，包括但不仅限于查看： 1) 二叉树的结构 2) 每个节点的深度以及它是否是叶子 3) 使用decision_path方法的示例到达的节点 4) 用apply这个接口取样出的叶子 5) 用于预测样本的规则 6) 一组样本共享的决策路径

tree_的更多内容可以参考：

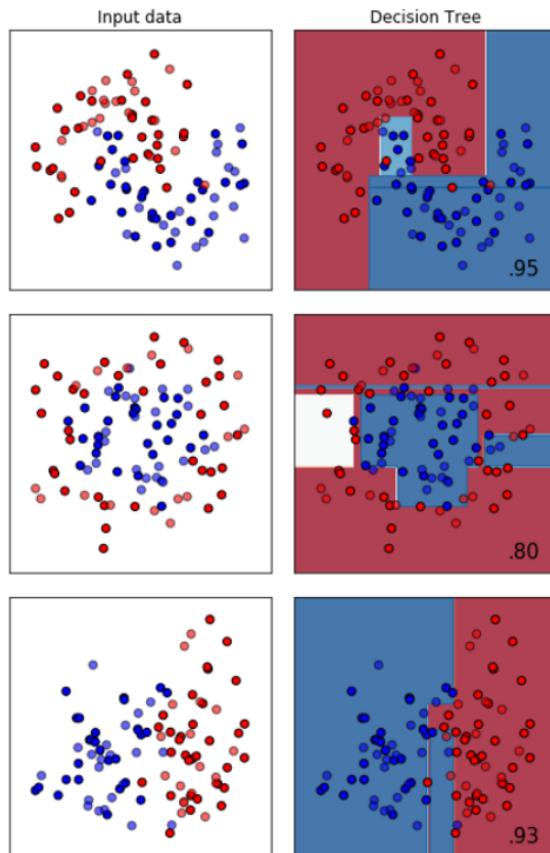
http://scikit-learn.org/stable/auto_examples/tree/plot_unveil_tree_structure.html#sphx-glr-auto-examples-tree-plot-unveil-tree-structure-py

6.3 分类树接口列表

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

Bonus Chapter I 实例：分类树在合成数集上的表现

我们在红酒数据集上画出了一棵树，并且展示了多个参数会对树形成这样的影响，接下来，我们将在不同结构的数据集上测试一下决策树的效果，让大家更好地理解决策树。



1. 导入需要的库

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.tree import DecisionTreeClassifier
```

2. 生成三种数据集

我们先从sklearn自带的数据库中生成三种类型的数据集：1) 月亮型数据，2) 环形数据，3) 二分型数据

```
#make_classification库生成随机的二分型数据
x, y = make_classification(n_samples=100, #生成100个样本
                           n_features=2, #包含2个特征，即生成二维数据
                           n_redundant=0, #添加冗余特征0个
                           n_informative=2, #包含信息的特征是2个
```

```

        random_state=1, #随机模式1
        n_clusters_per_class=1 #每个簇内包含的标签类别有1个
    )

#在这里可以查看一下x和y, 其中x是100行带有两个2特征的数据, y是二分类标签
#也可以画出散点图来观察一下x中特征的分布
#plt.scatter(x[:,0],x[:,1])

#从图上可以看出, 生成的二分型数据的两个簇离彼此很远, 这样不利于我们测试分类器的效果, 因此我们使用np生成
#随机数组, 通过让已经生成的二分型数据点加减0~1之间的随机数, 使数据分布变得更散更稀疏
#注意, 这个过程只能够运行一次, 因为多次运行之后x会变得非常稀疏, 两个簇的数据会混合在一起, 分类器的效应会
#继续下降
rng = np.random.RandomState(2) #生成一种随机模式
x += 2 * rng.uniform(size=x.shape) #加减0~1之间的随机数
linearly_separable = (x, y) #生成了新的x, 依然可以画散点图来观察一下特征的分布
#plt.scatter(x[:,0],x[:,1])

#用make_moons创建月亮型数据, make_circles创建环形数据, 并将三组数据打包起来放在列表datasets中
datasets = [make_moons(noise=0.3, random_state=0),
            make_circles(noise=0.2, factor=0.5, random_state=1),
            linearly_separable]

```

3. 画出三种数据集和三棵决策树的分类效应图像

```

#创建画布, 宽高比为6*9
figure = plt.figure(figsize=(6, 9))
#设置用来安排图像显示位置的全局变量i
i = 1

#开始迭代数据, 对datasets中的数据进行for循环

for ds_index, ds in enumerate(datasets):
    #对x中的数据进行标准化处理, 然后分训练集和测试集
    X, y = ds
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.4,
random_state=42)

    #找出数据集中两个特征的最大值和最小值, 让最大值+0.5, 最小值-0.5, 创造一个比两个特征的区间本身更大
    #一点的区间
    x1_min, x1_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    x2_min, x2_max = X[:, 1].min() - .5, X[:, 1].max() + .5

    #用特征向量生成网格数据, 网格数据, 其实就相当于坐标轴上无数个点
    #函数np.arange在给定的两个数之间返回均匀间隔的值, 0.2为步长
    #函数meshgrid用以生成网格数据, 能够将两个一维数组生成两个二维矩阵。
    #如果第一个数组是narray, 维度是n, 第二个参数是marray, 维度是m。那么生成的第一个二维数组是以
    narray为行, m行的矩阵, 而第二个二维数组是以marray的转置为列, n列的矩阵
    #生成的网格数据, 是用来绘制决策边界的, 因为绘制决策边界的函数contourf要求输入的两个特征都必须是二
    维的
    array1, array2 = np.meshgrid(np.arange(x1_min, x1_max, 0.2),
                                 np.arange(x2_min, x2_max, 0.2))

```

```

#接下来生成彩色画布
#用ListedColormap为画布创建颜色, #FF0000正红, #0000FF正蓝
cm = plt.cm.RdBu
cm_bright = ListedColormap(['#FF0000', '#0000FF'])

#在画布上加上一个子图, 数据为len(datasets)行, 2列, 放在位置i上
ax = plt.subplot(len(datasets), 2, i)

#到这里为止, 已经生成了0~1之间的坐标系3个了, 接下来为我们的坐标系放上标题
#我们有三个坐标系, 但我们只需要在第一个坐标系上有标题, 因此设定if ds_index==0这个条件
if ds_index == 0:
    ax.set_title("Input data")

#将数据集的分布放到我们的坐标系上
#先放训练集
ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train,
           cmap=cm_bright, edgecolors='k')
#放测试集
ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test,
           cmap=cm_bright, alpha=0.6, edgecolors='k')

#为图设置坐标轴的最大值和最小值, 并设定没有坐标轴
ax.set_xlim(array1.min(), array1.max())
ax.set_ylim(array2.min(), array2.max())
ax.set_xticks(())
ax.set_yticks(())

#每次循环之后, 改变i的取值让图每次位列不同的位置
i += 1

#至此为止, 数据集本身的图像已经布置完毕, 运行以上的代码, 可以看见三个已经处理好的数据集
#####
#####从这里开始是决策树模型#####

#迭代决策树, 首先用subplot增加子图, subplot(行, 列, 索引)这样的结构, 并使用索引i定义图的位置
#在这里, len(datasets)其实就是3, 2是两列
#在函数最开始, 我们定义了i=1, 并且在上边建立数据集的图像的时候, 已经让i+1, 所以i在每次循环中的取值
#是2, 4, 6
ax = plt.subplot(len(datasets), 2, i)

#决策树的建模过程: 实例化 → fit训练 → score接口得到预测的准确率
clf = DecisionTreeClassifier(max_depth=5)
clf.fit(X_train, y_train)
score = clf.score(X_test, y_test)

#绘制决策边界, 为此, 我们将为网格中的每个点指定一种颜色[x1_min, x1_max] × [x2_min, x2_max]
#分类树的接口, predict_proba, 返回每一个输入的数据点所对应的标签类概率
#类概率是数据点所在的叶节点中相同类的样本数量/叶节点中的样本总数量
#由于决策树在训练的时候导入的训练集X_train里面包含两个特征, 所以我们在计算类概率的时候, 也必须导入
#结构相同的数组, 即是说, 必须有两个特征
#ravel()能够将一个多维数组转换成一维数组
#np.c_是能够将两个数组组合起来的函数

```

```

#在这里，我们先将两个网格数据降维降维成一维数组，再将两个数组链接变成含有两个特征的数据，再带入决策
树模型，生成的Z包含数据的索引和每个样本点对应的类概率，再切片，切出类概率
z = clf.predict_proba(np.c_[array1.ravel(), array2.ravel()])[:, 1]

#np.c_[np.array([1,2,3]), np.array([4,5,6])]

#将返回的类概率作为数据，放到contourf里面绘制去绘制轮廓
z = z.reshape(array1.shape)
ax.contourf(array1, array2, z, cmap=cm, alpha=.8)

#将数据集的分布放到我们的坐标系上
# 将训练集放到图中去
ax.scatter(x_train[:, 0], x_train[:, 1], c=y_train, cmap=cm_bright,
           edgecolors='k')
# 将测试集放到图中去
ax.scatter(x_test[:, 0], x_test[:, 1], c=y_test, cmap=cm_bright,
           edgecolors='k', alpha=0.6)

#为图设置坐标轴的最大值和最小值
ax.set_xlim(array1.min(), array1.max())
ax.set_ylim(array2.min(), array2.max())
#设定坐标轴不显示标尺也不显示数字
ax.set_xticks(())
ax.set_yticks(())

#我们有三个坐标系，但我们只需要在第一个坐标系上有标题，因此设定if ds_index==0这个条件
if ds_index == 0:
    ax.set_title("Decision Tree")

#写在右下角的数字
ax.text(array1.max() - .3, array2.min() + .3, ('{:.1f}%'.format(score*100)),
        size=15, horizontalalignment='right')

#让i继续加一
i += 1

plt.tight_layout()
plt.show()

```

从图上来看，每一条线都是决策树在二维平面上画出的一条决策边界，每当决策树分枝一次，就有一条线出现。当数据的维度更高的时候，这条决策边界就会由线变成面，甚至变成我们想象不出的多维图形。

同时，很容易看得出，分类树天生不擅长环形数据。每个模型都有自己的决策上限，所以一个怎样调整都无法提升表现的可能性也是有的。当一个模型怎么调整都不行的时候，我们可以选择换其他的模型使用，不要在一棵树上吊死。顺便一说，最擅长月亮型数据的是最近邻算法，RBF支持向量机和高斯过程；最擅长环形数据的是最近邻算法和高斯过程；最擅长对半分的数据的是朴素贝叶斯，神经网络和随机森林。

Bonus Chapter II: 配置开发环境&安装sklearn

在整堂课中，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)

Graphviz 0.8.4 (没有画不出决策树哦，安装代码`conda install python-graphviz`)

Numpy 1.15.3, **Pandas** 0.23.4, **Matplotlib** 3.0.1, **SciPy** 1.1.0

以上的库和模块都是必须的，但版本可以不用太过限制，只要python在3.4以上，保证sklearn可以使用就没问题了。接下来，大家可以根据自己计算机的开发环境，选读下面的章节，以完成课程需要的开发环境的配置。

- "我从没有使用过python"

若你是第一次使用python，还没有安装过，一个最简单的方式是直接安装Python针对科学计算而发布的开发环境Anaconda。

<https://www.anaconda.com/download/>

在这个网站里，根据你所使用的操作系统下载合适的Anaconda版本来安装即可。Anaconda里包含这门课所需要的所有工具包，包括Numpy, Scipy, Pandas和sklearn等等，可以在安装python的时候一并安装完毕，无需任何多余的操作。

- "我用过python，但没有配置所有需要的库"

如果已经安装了Python，但没有所需的库，也不想再通过Anaconda进行操作，那可以使用pip逐个安装这些工具包。对于版本在2.7.9以上的python 2，以及3.4以上的python 3而言，pip是自带的，同时，这两个版本也是sklearn库对于python的最低要求。如果发现自己的python中无法运行pip命令，那你的python也将无法运行sklearn，最迅速的方法是直接卸载重装版本更高的python。现在从官网下载，python的版本是带有pip命令的3.7.1。

<https://www.python.org/downloads/>

能够使用Anaconda，就不推荐使用pip。pip安装一部分库的时候可能会出现异常，原因是pip默认下载的一部分库的版本（如SciPy）可能只适用于linux系统，而Anaconda的安装一般不会有这个问题。对于能够自己排查出现的问题的学员，请随意选择任何安装方式。

注意，Anaconda的安装和pip的安装尽量不要混用，由Anaconda安装的库在使用pip卸载或是更新的时候，可能出现无法卸载干净，无法正常更新，或更新后一部分库变得无法运行的情况。

所有安装命令均在cmd中运行，pip的安装命令如下，**这些命令需要每一条分开运行**：

```
pip install numpy  
pip install pandas  
pip install scipy  
pip install matplotlib  
pip install -u scikit-learn
```

Anaconda的安装命令如下：

```
conda install numpy  
conda install pandas  
conda install scipy  
conda install matplotlib  
conda install scikit-learn
```

安装过程中任何的报错，都可以通过卸载重装来解决问题，这是最有效率的方式。

- "我用过python，并且熟悉numpy&pandas"

如果你使用过python，并且对NumPy, Pandas等库比较熟悉，那你可以直接运行pip或conda来安装sklearn，在cmd中运行pip或者conda命令就可安装scikit-learn。命令二选一，不要重复安装。

```
pip install -u scikit-learn
```

```
conda install scikit-learn
```

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~



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。

1.2 sklearn中的集成算法

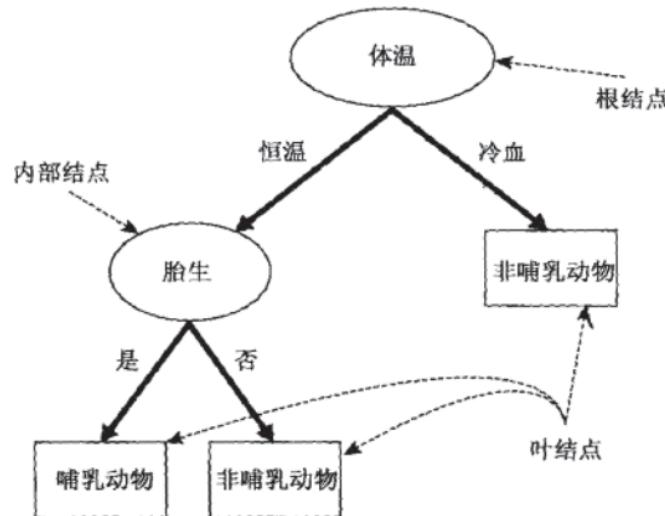
- **sklearn中的集成算法模块ensemble**

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	用于不合适的估算器的软投票/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:{}\n".format(score_c))
      , "Random Forest:{}\n".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 $0.2(\varepsilon)$ ，那20棵树以上都判断错误的可能性是：

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

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， $(1-\varepsilon)$ 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。

```
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。**

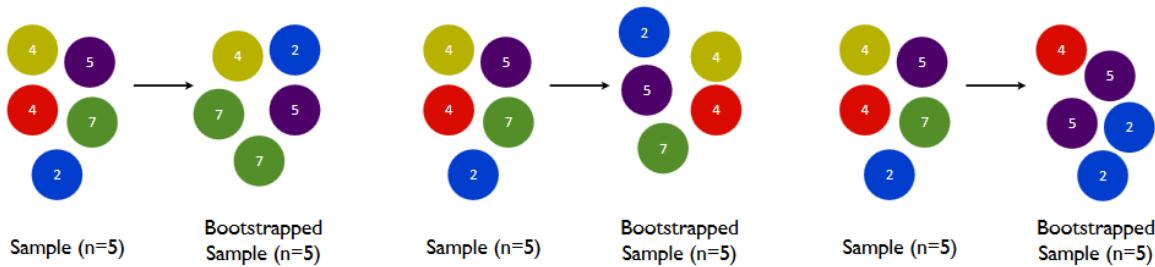
但这种做法的局限性是很强的，当我们需要成千上万棵树的时候，数据不一定能够提供成千上万的特征来让我们构筑尽量多尽量不同的树。因此，除了random_state。我们还需要其他的随机性。

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：

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

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， 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：

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

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

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， 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。

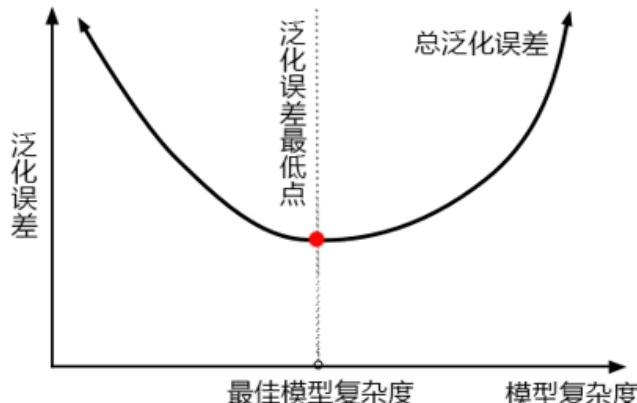
通过画学习曲线，或者网格搜索，我们能够探索到调参边缘（代价可能是训练一次模型要跑三天三夜），但是在现实中，高手调参恐怕还是多依赖于经验，而这些经验，来源于：1) 非常正确的调参思路和方法，2) 对模型评估指标的理解，3) 对数据的感觉和经验，4) 用洪荒之力去不断地尝试。

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 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。

• 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。

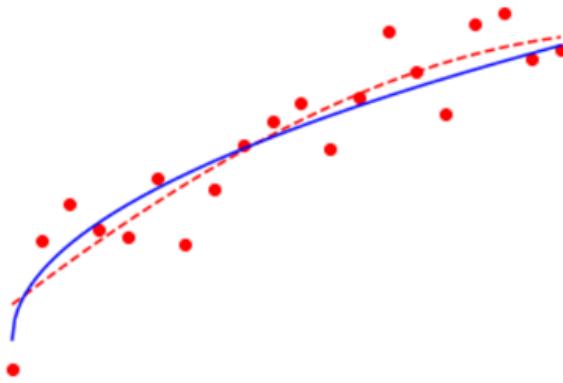
$$E(f; D) = bias^2(x) + var(x) + \varepsilon^2$$

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。

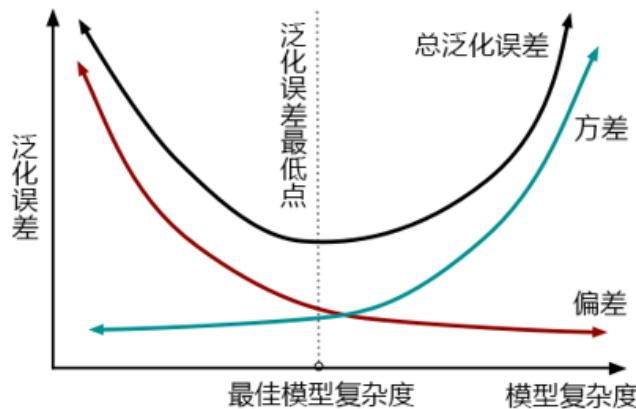
方差：反映的是模型每一次输出结果与模型预测值的平均水平之间的误差，即每一个红点到红色虚线的距离，衡量模型的稳定性。模型越稳定，方差越低。



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

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】 #####
```

```
score1 = []
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()
    score1.append(score)
print(max(score1),(score1.index(max(score1))*10)+1)
plt.figure(figsize=[20,5])
plt.plot(range(1,201,10),score1)
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

```
score1 = []
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()
    score1.append(score)
print(max(score1),[*range(35,45)][score1.index(max(score1))])
plt.figure(figsize=[20,5])
plt.plot(range(35,45),score1)
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

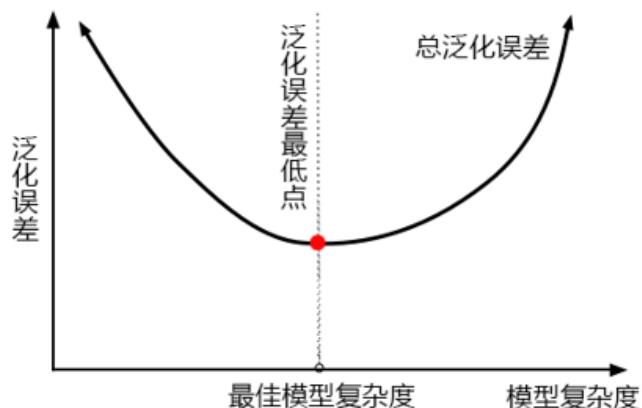
....

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)}
```

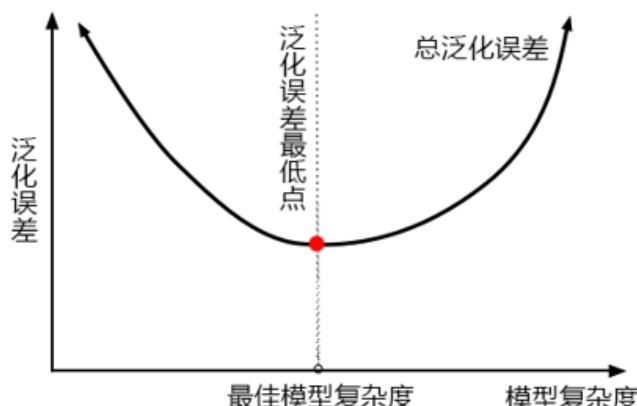
....

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，大家量力量时间而行，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	整数, 可不填, 默认10 随机森林中树模型的数量。 <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>
criterion	字符串型, 可不填, 默认基尼系数 ("gini") 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)
max_depth	整数或None, 可不填, 默认None 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
min_samples_split	整数或浮点数, 可不填, 默认=2 一个中间节点要分枝所需要的最小样本量。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 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 <i>浮点数功能是0.18版本以上的sklearn才可以使用</i>
min_sample_leaf	整数或浮点数, 可不填, 默认=1 一个叶节点要存在所需要的最小样本量。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 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	浮点数, 可不填, 默认=0. 一个叶节点要存在所需要的权重占输入模型的数据集的总权重的比例。 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
max_features	整数, 浮点数, 字符型或None, 可不填, 默认None 在做最佳分枝的时候, 考虑的特征个数 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 <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>

max_leaf_nodes	整数或None, 可不填, 默认None 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。
min_impurity_decrease	浮点数, 可以不填, 默认=0. 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。 带权重的不纯度下降可以表示为: $\frac{N_t}{N} * (\text{不纯度} - \frac{N_{t_R}}{N_t} * \text{右测树枝的不纯度} - \frac{N_{t_L}}{N_t} * \text{左侧树枝的不纯度})$ 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 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 仅在0.19以上版本中提供此功能
min_impurity_split	浮点数 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。 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
bootstrap	布尔值, 可不填, 默认True 在建树过程中, 是否使用自举样本抽样的方式。
oob_score	布尔值, 默认False 在建树时, 是否适用袋外样本来预测模型的泛化精确性
n_jobs	整数或None, 可不填, 默认None 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。 更多详细信息请参阅: https://scikit-learn.org/stable/glossary.html#term-n-jobs
random_state	整数, sklearn中设定好的RandomState实例, 或None, 可不填, 默认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	整数, 可不填, 默认0 在拟合和预测时控制树的复杂度。
warm_start	布尔值, 可不填, 默认False 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。

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><code>[{0:1,1:1}, {0:1,1:5}, {0:1, 1:1}, {0:1,1:1}]</code></p> <p>而不是：</p> <p><code>[{1:1}, {2:5}, {3:1}, {4:1}]</code></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

菜菜的scikit-learn课堂03

sklearn中的数据预处理和特征工程

小伙伴们晚上好~o(￣▽￣)ブ

我是菜菜，这里是我的sklearn课堂第三期，今晚的直播内容是数据预处理和特征工程~

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~



菜菜的scikit-learn课堂03

sklearn中的数据预处理和特征工程

1 概述

- 1.1 数据预处理与特征工程
- 1.2 sklearn中的数据预处理和特征工程

2 数据预处理 Preprocessing & Impute

- 2.1 数据无量纲化
- 2.2 缺失值
- 2.3 处理分类型特征：编码与哑变量
- 2.4 处理连续型特征：二值化与分段

3 特征选择 feature_selection

- 3.1 Filter过滤法
 - 3.1.1 方差过滤
 - 3.1.1.1 VarianceThreshold
 - 3.1.1.2 方差过滤对模型的影响
 - 3.1.1.3 选取超参数threshold
 - 3.1.2 相关性过滤
 - 3.1.2.1 卡方过滤
 - 3.1.2.2 选取超参数K
 - 3.1.2.3 F检验
 - 3.1.2.4 互信息法
 - 3.1.3 过滤法总结
- 3.2 Embedded嵌入法
- 3.3 Wrapper包装法
- 3.4 特征选择总结

4 sklearn课程12周提纲

1 概述

1.1 数据预处理与特征工程

想象一下未来美好的一天，你学完了菜菜的课程，成为一个精通各种算法和调参调库的数据挖掘工程师了。某一天你从你的同事，一位药物研究人员那里，得到了一份病人临床表现的数据。药物研究人员用前四列数据预测一下最后一数据，还说他要出差几天，可能没办法和你一起研究数据了，希望出差回来以后，可以有个初步分析结果。于是你就看了看数据，看着很普通，预测连续型变量，好说，导随机森林回归器调出来，调参调呀调，MSE很小，跑了个还不错的结果。

012	232	33.5	0	10.7
020	121	16.9	2	210.1
027	165	24.0	0	427.6

几天后，你同事出差回来了，准备要一起开会了，会上你碰见了和你同事在同一个项目里工作的统计学家。他问起你的分析结果，你说你已经小有成效了，统计学家很吃惊，他说：“不错呀，这组数据问题太多，我都分析不出什么来。”

你心里可能咯噔一下，忐忑地回答说：“我没听说数据有什么问题呀。”

统计学家：“第四列数据很坑爹，这个特征的取值范围是1~10，0是表示缺失值的。而且他们输入数据的时候出错，很多10都被录入成0了，现在分不出来了。”

你：“.....”

统计学家：“还有第二列和第三列数据基本是一样的，相关性太强了。”

你：“这个我发现了，不过这两个特征在预测中的重要性都不高，无论其他特征怎样出错，我这边结果里显示第一列的特征是最重要的，所以也无所谓啦。”

统计学家：“啥？第一列不就是编号吗？”

你：“不是吧。”

统计学家：“哦我想起来了！第一列就是编号，不过那个编号是我们根据第五列排序之后编上去的！这个第一列和第五列是由很强的联系，但是毫无意义啊！”

老血喷了一屏幕，数据挖掘工程师卒。

这个悲惨又可爱的故事来自《数据挖掘导论》，虽然这是故事里的状况十分极端，但我还是想把这段对话作为今天这章的开头，博大家一笑（虽然可能听完就泪流满面了）。在过去两周，我们已经讲了两个算法：决策树和随机森林，我们通过决策树带大家认识了sklearn，通过随机森林讲解了机器学习中调参的基本思想，现在可以说，只要上过前面两堂课的，人人都会调随机森林和决策树的分类器了，而我呢，也只需要跟着各大机器学习书籍的步伐，给大家一周一个算法带着讲解就是了。如果这样的话，结果可能就是，大家去工作了，遇到了一个不那么靠谱的同事，给了你一组有坑的数据，最后你就一屏幕老血吐过去，牺牲在数据行业的前线了。

数据不给力，再高级的算法都没有用。

我们在课堂中给大家提供的数据，都是经过层层筛选，适用于课堂教学的——运行时间短，预测效果好，没有严重缺失等等问题。尤其是sklearn中的数据，堪称完美。各大机器学习教材也是如此，都给大家提供处理好的数据，这就导致，很多人在学了很多算法之后，到了现实应用之中，发现模型经常就调不动了，因为现实中的数据，离平时上课使用的完美数据集，相差十万八千里。所以我决定，少讲一两个简单的算法，为大家专门拿一堂课来讲解建

模之前的流程，**数据预处理和特征工程**。这样大家即可以学到数据挖掘过程中很重要但是却经常被忽视的一些步骤，也可以不受课堂的限制，如果自己有时间，可以尝试在真实数据上建模。

数据挖掘的五大流程：

1. 获取数据

2. 数据预处理

数据预处理是从数据中检测，纠正或删除损坏，不准确或不适用于模型的记录的过程

可能面对的问题有：数据类型不同，比如有的是文字，有的是数字，有的含时间序列，有的连续，有的间断。也可能，数据的质量不行，有噪声，有异常，有缺失，数据出错，量纲不一，有重复，数据是偏态，数据量太大或太小

数据预处理的目的：让数据适应模型，匹配模型的需求

3. 特征工程：

特征工程是将原始数据转换为更能代表预测模型的潜在问题的特征的过程，可以通过挑选最相关的特征，提取特征以及创造特征来实现。其中创造特征又经常以降维算法的方式实现。

可能面对的问题有：特征之间有相关性，特征和标签无关，特征太多或太小，或者干脆就无法表现出应有的数据现象或无法展示数据的真实面貌

特征工程的目的：1) 降低计算成本，2) 提升模型上限

4. 建模，测试模型并预测出结果

5. 上线，验证模型效果

1.2 sklearn中的数据预处理和特征工程

sklearn中包含众多数据预处理和特征工程相关的模块，虽然刚接触sklearn时，大家都会为其中包含的各种算法的广度深度所震惊，但其实sklearn六大板块中有两块都是关于数据预处理和特征工程的，两个板块互相交互，为建模之前的全部工程打下基础。

Classification

Identifying to which category an object belongs to.

Applications: Spam detection, Image recognition.

Algorithms: SVM, nearest neighbors, random forest, ...

Regression

Predicting a continuous-valued attribute associated with an object.

Applications: Drug response, Stock prices.

Algorithms: SVR, ridge regression, Lasso, ...

Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, Grouping experiment outcomes

Algorithms: k-Means, spectral clustering, mean-shift, ...

Dimensionality reduction

Reducing the number of random variables to consider.

Applications: Visualization, Increased efficiency

Algorithms: PCA, feature selection, non-negative matrix factorization.

Model selection

Comparing, validating and choosing parameters and models.

Goal: Improved accuracy via parameter tuning

Modules: grid search, cross validation, metrics.

Preprocessing

Feature extraction and normalization.

Application: Transforming input data such as text for use with machine learning algorithms.

Modules: preprocessing, feature extraction.

— Examples

- 模块preprocessing：几乎包含数据预处理的所有内容
- 模块Impute：填补缺失值专用
- 模块feature_selection：包含特征选择的各种方法的实践

- 模块decomposition：包含降维算法

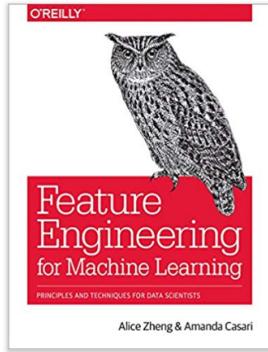
对于特征工程，来介绍O'Reilly Media出版社的新书：

Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists 1st Edition,

by Alice Zheng (Author), Amanda Casari (Author)

★★★★★ 6 customer reviews

[Look Inside](#)



Kindle \$29.99

Paperback from \$29.00

Other Sellers
See all 3 versions

Buy

\$29.99

Digital List Price: \$50.99 - Save \$21.00 (41%)



[Buy now with 1-Click](#)

[Send a free sample](#)

Deliver to your Kindle or other device

[Give as a Gift](#)

[Enter a promotion code or Gift Card](#)

ISBN-13: 978-1491953242

ISBN-10: 1491953241

[Why is ISBN important?](#)

Sold by: Amazon Digital Services LLC

2 数据预处理 Preprocessing & Impute

2.1 数据无量纲化

在机器学习算法实践中，我们往往有着将不同规格的数据转换到同一规格，或不同分布的数据转换到某个特定分布的需求，这种需求统称为将数据“无量纲化”。譬如梯度和矩阵为核心的算法中，譬如逻辑回归，支持向量机，神经网络，无量纲化可以加快求解速度；而在距离类模型，譬如K近邻，K-Means聚类中，无量纲化可以帮我们提升模型精度，避免某一个取值范围特别大的特征对距离计算造成影响。（一个特例是决策树和树的集成算法们，对决策树我们不需要无量纲化，决策树可以把任意数据都处理得很好。）

数据的无量纲化可以是线性的，也可以是非线性的。线性的无量纲化包括**中心化** (Zero-centered或者Mean-subtraction) 处理和**缩放处理** (Scale)。中心化的本质是让所有记录减去一个固定值，即让数据样本数据平移到某个位置。缩放的本质是通过除以一个固定值，将数据固定在某个范围之中，取对数也算是一种缩放处理。

- **preprocessing.MinMaxScaler**

当数据(x)按照最小值中心化后，再按极差（最大值 - 最小值）缩放，数据移动了最小值个单位，并且会被收敛到[0,1]之间，而这个过程，就叫做**数据归一化**(Normalization，又称Min-Max Scaling)。注意，Normalization是归一化，不是正则化，真正的正则化是regularization，不是数据预处理的一种手段。归一化之后的数据服从正态分布，公式如下：

$$x^* = \frac{x - \min(x)}{\max(x) - \min(x)}$$

在sklearn当中，我们使用**preprocessing.MinMaxScaler**来实现这个功能。MinMaxScaler有一个重要参数，feature_range，控制我们希望把数据压缩到的范围，默认是[0,1]。

```
from sklearn.preprocessing import MinMaxScaler

data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]

#不太熟悉numpy的小伙伴，能够判断data的结构吗？
#如果换成表是什么样子？
import pandas as pd
pd.DataFrame(data)

#实现归一化
scaler = MinMaxScaler()                      #实例化
scaler = scaler.fit(data)                     #fit，在这里本质是生成min(x)和max(x)
result = scaler.transform(data)                #通过接口导出结果
result

result_ = scaler.fit_transform(data)           #训练和导出结果一步达成

scaler.inverse_transform(result)               #将归一化后的结果逆转

#使用MinMaxScaler的参数feature_range实现将数据归一化到[0,1]以外的范围内

data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]
scaler = MinMaxScaler(feature_range=[5,10])    #依然实例化
```

```

result = scaler.fit_transform(data)           #fit_transform一步导出结果
result

#当x中的特征数量非常多的时候，fit会报错并表示，数据量太大了我计算不了
#此时使用partial_fit作为训练接口
#scaler = scaler.partial_fit(data)

```

BONUS: 使用numpy来实现归一化

```

import numpy as np
x = np.array([[[-1, 2], [-0.5, 6], [0, 10], [1, 18]]])

#归一化
x_nor = (x - x.min(axis=0)) / (x.max(axis=0) - x.min(axis=0))
x_nor

#逆转归一化
x_returned = x_nor * (x.max(axis=0) - x.min(axis=0)) + x.min(axis=0)
x_returned

```

- **preprocessing.StandardScaler**

当数据(x)按均值(μ)中心化后，再按标准差(σ)缩放，数据就会服从为均值为0，方差为1的正态分布（即标准正态分布），而这个过程，就叫做**数据标准化**(Standardization，又称Z-score normalization)，公式如下：

$$x^* = \frac{x - \mu}{\sigma}$$

```

from sklearn.preprocessing import StandardScaler
data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]

scaler = StandardScaler()                  #实例化
scaler.fit(data)                         #fit，本质是生成均值和方差

scaler.mean_                             #查看均值的属性mean_
scaler.var_                              #查看方差的属性var_

x_std = scaler.transform(data)            #通过接口导出结果

x_std.mean()                            #导出的结果是一个数组，用mean()查看均值
x_std.std()                             #用std()查看方差

scaler.fit_transform(data)               #使用fit_transform(data)一步达成结果

scaler.inverse_transform(x_std)          #使用inverse_transform逆转标准化

```

对于StandardScaler和MinMaxScaler来说，空值NaN会被当做是缺失值，在fit的时候忽略，在transform的时候保持缺失NaN的状态显示。并且，尽管去量纲化过程不是具体的算法，但在fit接口中，依然只允许导入至少二维数组，一维数组导入会报错。通常来说，我们输入的X会是我们的特征矩阵，现实案例中特征矩阵不太可能是一维所以不会存在这个问题。

- **StandardScaler和MinMaxScaler选哪个？**

看情况。大多数机器学习算法中，会选择StandardScaler来进行特征缩放，因为MinMaxScaler对异常值非常敏感。在PCA，聚类，逻辑回归，支持向量机，神经网络这些算法中，StandardScaler往往是最好的选择。

MinMaxScaler在不涉及距离度量、梯度、协方差计算以及数据需要被压缩到特定区间时使用广泛，比如数字图像处理中量化像素强度时，都会使用MinMaxScaler将数据压缩于[0,1]区间之中。

建议先试试看StandardScaler，效果不好换MinMaxScaler。

除了StandardScaler和MinMaxScaler之外，sklearn中也提供了各种其他缩放处理（中心化只需要一个pandas广播一下减去某个数就好了，因此sklearn不提供任何中心化功能）。比如，在希望压缩数据，却不影响数据的稀疏性时（不影响矩阵中取值为0的个数时），我们会使用MaxAbsScaler；在异常值多，噪声非常大时，我们可能会选用分位数来无量纲化，此时使用RobustScaler。更多详情请参考以下列表。

无量纲化	功能	中心化	缩放	详解
.StandardScaler	标准化	均值	方差	通过减掉均值并将数据缩放到单位方差来标准化特征，标准化完毕后的特征服从标准正态分布，即方差为1，均值为0
.MinMaxScaler	归一化	最小值	极差	通过最大值最小值将每个特征缩放到给定范围，默认[0,1]
.MaxAbsScaler	缩放	N/A	最大值	通过让每一个特征里的数据，除以该特征中绝对值最大的数值的绝对值，将数据压缩到[-1,1]之间，这种做法并没有中心化数据，因此不会破坏数据的稀疏性。数据的稀疏性是指，数据中包含0的比例，0越多，数据越稀疏。
.RobustScaler	无量纲化	中位数	四分位数范围	使用可以处理异常值，对异常值不敏感的统计量来缩放数据。 这个缩放器删除中位数并根据百分位数范围 (IQR: Interquartile Range) 缩放数据。IQR是第一分位数 (25%) 和第三分位数 (75%) 之间的范围。数据集的标准化是通过去除均值，缩放到单位方差来完成，但是异常值通常会对样本的均值和方差造成负面影响，当异常值很多噪音很大时，用中位数和四分位数范围通常会产生更好的效果。
.Normalizer	无量纲化	N/A	sklearn中未明确，依范数原理应当是： I1：样本向量的长度/样本中每个元素绝对值的和 I2：样本向量的长度/样本中每个元素的欧氏距离	将样本独立缩放到单位范数。每个至少带一个非0值的样本都回被独立缩放，使得整个样本（整个向量）的的长度都为I1范数或I2范数。这个类可以处理密集数组(numpy arrays)或scipy中的稀疏矩阵 (scipy.sparse)，如果你希望避免复制/转换过程中的负担，请使用CSR格式的矩阵。 将输入的数据缩放到单位范数是文本分类或聚类中的常见操作。例如，两个I2正则化后的TF-IDF向量的点积是向量的余弦相似度，并且是信息检索社区常用的向量空间模型的基本相似性度量。 使用参数norm来确定要正则化的范数方向，可以选择“l1”，“l2”以及“max”三种选项，默认l2范数。 这个评估器的fit接口什么也不做，但在管道中使用依然是很有用的。
.Power Transformer	非线性无量纲化	N/A	N/A	应用特征功率变换使数据更接近正态分布。 功率变换是一系列参数单调变换，用于使数据更像高斯。这对于建模与异方差性（非常数方差）或其他需要正态性的情况相关的问题非常有用。要求输入的数据严格为正，power_transform()通过最大似然估计来稳定方差和并确定最小化偏度的最佳参数。 默认情况下，标准化应用于转换后的数据。
.Quantile Transformer	非线性无量纲化	N/A	N/A	使用百分位数转换特征，通过缩小边缘异常值和非异常值之间的距离来提供特征的非线性变换。可以使用参数output_distribution = "normal"来将数据映射到标准正态分布。
.KernelCenterer	中心化	均值	N/A	将核矩阵中心化。设K(x, z)是由phi(x)^T phi(z)定义的核，其中phi是将x映射到希尔伯特空间的函数。 KernelCenterer在不明确计算phi(x)的情况下让数据中心化为0均值。它相当于使用sklearn.preprocessing.StandardScaler(with_std = False)来将phi(x)中心化。

2.2 缺失值

机器学习和数据挖掘中所使用的数据，永远不可能是完美的。很多特征，对于分析和建模来说意义非凡，但对于实际收集数据的人却不是如此，因此数据挖掘之中，常常会有重要的字段缺失值很多，但又不能舍弃字段的情况。因此，数据预处理中非常重要的一项就是处理缺失值。

```
import pandas as pd
data = pd.read_csv(r"C:\work\learnbetter\micro-class\week 3 Preprocessing\NarrativeData.csv", index_col=0)

data.head()
```

在这里，我们使用从泰坦尼克号提取出来的数据，这个数据有三个特征，一个数值型，两个字符型，标签也是字符型。从这里开始，我们就使用这个数据给大家作为例子，让大家慢慢熟悉sklearn中数据预处理的各种方式。

- **impute.SimpleImputer**

```
class sklearn.impute.SimpleImputer(missing_values=nan, strategy='mean', fill_value=None, verbose=0, copy=True)
```

在讲解随机森林的案例时，我们用这个类和随机森林回归填补了缺失值，对比了不同的缺失值填补方式对数据的影响。这个类是专门用来填补缺失值的。它包括四个重要参数：

参数	含义&输入
missing_values	告诉SimpleImputer，数据中的缺失值长什么样，默认空值np.nan
strategy	我们填补缺失值的策略，默认均值。 输入“mean”使用均值填补（仅对数值型特征可用） 输入“median”用中值填补（仅对数值型特征可用） 输入“most_frequent”用众数填补（对数值型和字符型特征都可用） 输入“constant”表示请参考参数“fill_value”中的值（对数值型和字符型特征都可用）
fill_value	当参数startegy为“constant”的时候可用，可输入字符串或数字表示要填充的值，常用0
copy	默认为True，将创建特征矩阵的副本，反之则会将缺失值填补到原本的特征矩阵中去。

```
data.info()
#填补年龄

Age = data.loc[:, "Age"].values.reshape(-1, 1) #sklearn当中特征矩阵必须是二维
Age[:20]

from sklearn.impute import SimpleImputer
imp_mean = SimpleImputer() #实例化， 默认均值填补
imp_median = SimpleImputer(strategy="median") #用中位数填补
imp_0 = SimpleImputer(strategy="constant", fill_value=0) #用0填补

imp_mean = imp_mean.fit_transform(Age) #fit_transform一步完成调取结果
imp_median = imp_median.fit_transform(Age)
imp_0 = imp_0.fit_transform(Age)

imp_mean[:20]
imp_median[:20]
imp_0[:20]

#在这里我们使用中位数填补Age
data.loc[:, "Age"] = imp_median

data.info()

#使用众数填补Embarked
Embarked = data.loc[:, "Embarked"].values.reshape(-1, 1)
```

```
imp_mode = SimpleImputer(strategy = "most_frequent")
data.loc[:, "Embarked"] = imp_mode.fit_transform(Embarked)

data.info()
```

BONUS: 用Pandas和Numpy进行填补其实更加简单

```
import pandas as pd
data = pd.read_csv(r"C:\work\Learnbetter\micro-class\week 3
Preprocessing\NarrativeData.csv", index_col=0)

data.head()

data.loc[:, "Age"] = data.loc[:, "Age"].fillna(data.loc[:, "Age"].median())
#.fillna 在DataFrame里面直接进行填补

data.dropna(axis=0, inplace=True)
#.dropna(axis=0)删除所有有缺失值的行, .dropna(axis=1)删除所有有缺失值的列
#参数inplace, 为True表示在原数据集上进行修改, 为False表示生成一个复制对象, 不修改原数据, 默认False
```

2.3 处理分类型特征：编码与哑变量

在机器学习中，大多数算法，譬如逻辑回归，支持向量机SVM，k近邻算法等都只能够处理数值型数据，不能处理文字，在sklearn当中，除了专用来处理文字的算法，其他算法在fit的时候全部要求输入数组或矩阵，也不能够导入文字型数据（其实手写决策树和普斯贝叶斯可以处理文字，但是sklearn中规定必须导入数值型）。

然而在现实中，许多标签和特征在数据收集完毕的时候，都不是以数字来表现的。比如说，学历的取值可以是["小学", "初中", "高中", "大学"]，付费方式可能包含["支付宝", "现金", "微信"]等等。在这种情况下，为了让数据适应算法和库，我们必须将数据进行**编码**，即是说，**将文字型数据转换为数值型**。

- **preprocessing.LabelEncoder**: 标签专用，能够将分类转换为分类数值

```
from sklearn.preprocessing import LabelEncoder

y = data.iloc[:, -1] #要输入的是标签, 不是特征矩阵, 所以允许一维

le = LabelEncoder() #实例化
le = le.fit(y) #导入数据
label = le.transform(y) #transform接口调取结果

le.classes_ #属性.classes_查看标签中究竟有多少类别
label #查看获取的结果label

le.fit_transform(y) #也可以直接fit_transform一步到位

le.inverse_transform(label) #使用inverse_transform可以逆转
```

```

data.iloc[:, -1] = label #让标签等于我们运行出来的结果

data.head()

#如果不需要教学展示的话我会这么写:
from sklearn.preprocessing import LabelEncoder
data.iloc[:, -1] = LabelEncoder().fit_transform(data.iloc[:, -1])

```

- **preprocessing.OrdinalEncoder**: 特征专用, 能够将分类特征转换为分类数值

```

from sklearn.preprocessing import OrdinalEncoder

#接口categories_对应LabelEncoder的接口classes_, 一模一样的功能
data_ = data.copy()

data_.head()

OrdinalEncoder().fit(data_.iloc[:, 1:-1]).categories_

data_.iloc[:, 1:-1] = OrdinalEncoder().fit_transform(data_.iloc[:, 1:-1])

data_.head()

```

- **preprocessing.OneHotEncoder**: 独热编码, 创建哑变量

我们刚刚已经用OrdinalEncoder把分类变量Sex和Embarked都转换成数字对应的类别了。在舱门Embarked这一列中, 我们使用[0,1,2]代表了三个不同的舱门, 然而这种转换是正确的吗?

我们来思考三种不同性质的分类数据:

1) 舱门 (S, C, Q)

三种取值S, C, Q是相互独立的, 彼此之间完全没有联系, 表达的是 $S \neq C \neq Q$ 的概念。这是名义变量。

2) 学历 (小学, 初中, 高中)

三种取值不是完全独立的, 我们可以明显看出, 在性质上可以有高中>初中>小学这样的联系, 学历有高低, 但是学历取值之间却不是可以计算的, 我们不能说小学 + 某个取值 = 初中。这是有序变量。

3) 体重 (>45kg, >90kg, >135kg)

各个取值之间有联系, 且是可以互相计算的, 比如 $120\text{kg} - 45\text{kg} = 90\text{kg}$, 分类之间可以通过数学计算互相转换。这是有距变量。

然而在对特征进行编码的时候, 这三种分类数据都会被我们转换为[0,1,2], 这三个数字在算法看来, 是连续且可以计算的, 这三个数字相互不等, 有大小, 并且有着可以相加相乘的联系。所以算法会把舱门, 学历这样的分类特征, 都误会成是体重这样的分类特征。这是说, 我们把分类转换成数字的时候, 忽略了数字中自带的数学性质, 所以给算法传达了一些不准确的信息, 而这会影响我们的建模。

类别OrdinalEncoder可以用来处理有序变量, 但对于名义变量, 我们只有使用哑变量的方式来处理, 才能够尽量向算法传达最准确的信息:



这样的变化，让算法能够彻底领悟，原来三个取值是没有可计算性质的，是“有你就没有我”的不等概念。在我们的数据中，性别和舱门，都是这样的名义变量。因此我们需要使用独热编码，将两个特征都转换为哑变量。

```
data.head()

from sklearn.preprocessing import OneHotEncoder
x = data.iloc[:,1:-1]

enc = OneHotEncoder(categories='auto').fit(x)
result = enc.transform(x).toarray()
result

#依然可以直接一步到位，但为了给大家展示模型属性，所以还是写成了三步
OneHotEncoder(categories='auto').fit_transform(x).toarray()

#依然可以还原
pd.DataFrame(enc.inverse_transform(result))

enc.get_feature_names()

result
result.shape

#axis=1,表示跨行进行合并，也就是将量表左右相连，如果是axis=0，就是将量表上下相连
newdata = pd.concat([data,pd.DataFrame(result)],axis=1)

newdata.head()

newdata.drop(['Sex','Embarked'],axis=1,inplace=True)

newdata.columns =
['Age','Survived','Female','Male','Embarked_C','Embarked_Q','Embarked_S']

newdata.head()
```

特征可以做哑变量，标签也可以吗？可以，使用类sklearn.preprocessing.LabelBinarizer可以对做哑变量，许多算法都可以处理多标签问题（比如说决策树），但是这样的做法在现实中不常见，因此我们在这里就不赘述了。

编码与哑变量	功能	重要参数	重要属性	重要接口
.LabelEncoder	分类标签编码	N/A	.classes_ : 查看 标签中究竟有多少类别	fit, transform, fit_transform, inverse_transform
.OrdinalEncoder	分类特征编码	N/A	.categories_ : 查看特征中究竟有多少类别	fit, transform, fit_transform, inverse_transform
.OneHotEncoder	独热编码，为 名义变量创建 哑变量	categories: 每个特征都有哪些类别，默认"auto"表示让算法自己判断，或者可以输入列表，每个元素都是一个列表，表示每个特征中的不同类别 handle_unknown: 当输入了categories，且算法遇见了categories中没有写明的特征或类别时，是否报错。默认"error"表示请报错，也可以选择"ignore"表示请无视。如果选择"ignore"，则未再categories中注明的特征或类别的哑变量会全部显示为0。在逆变(inverse transform)中，未知特征或类别会被返回为None。	.categories_ : 查看特征中究竟有多少类别，如果是自己输入的类别，那就不需要查看了	fit, transform, fit_transform, inverse_transform, get_feature_names: 查看生成的哑变量的每一列都是什么特征的什么取值

BONUS：数据类型以及常用的统计量

数据类型	数据名称	数学含义	描述	举例	可用操作
离散, 定性	名义	=, ≠	名义变量就是不同的名字，是用来告诉我们，这两个数据是否相同的	邮编, 性别, 眼睛的颜色, 职工号	众数, 信息熵 情形分析表或列联表, 相关性分析, 卡方检验
离散, 定性	有序	<, >	有序变量为数据的相对大小提供信息，告诉我们数据的顺序，但数据之间大小的间隔不是具有固定意义的，因此有序变量不能加减	材料的硬度, 学历	中位数, 分位数, 非参数相关分析(等级相关), 测量系统分析, 符号检验
连续, 定量	有距	+, -	有距变量之间的间隔是有固定意义的，可以加减，比如，一单位量纲	日期, 以摄氏度或华氏度为量纲的温度	均值, 标准差, 皮尔逊相关系数, t和F检验
连续, 定量	比率	*, /	比变量之间的间隔和比例本身都是有意义的，既可以加减又可以乘除	以开尔文为量纲的温度, 货币数量, 计数, 年龄, 质量, 长度, 电流	几何平均, 调和平均, 百分数, 变化量

2.4 处理连续型特征：二值化与分段

- `sklearn.preprocessing.Binarizer`

根据阈值将数据二值化（将特征值设置为0或1），用于处理连续型变量。大于阈值的值映射为1，而小于或等于阈值的值映射为0。默认阈值为0时，特征中所有的正值都映射到1。二值化是对文本计数数据的常见操作，分析人员可以决定仅考虑某种现象的存在与否。它还可以用作考虑布尔随机变量的估计器的预处理步骤（例如，使用贝叶斯设置中的伯努利分布建模）。

```
#将年龄二值化

data_2 = data.copy()

from sklearn.preprocessing import Binarizer
x = data_2.iloc[:,0].values.reshape(-1,1) #类为特征专用，所以不能使用一维数组
transformer = Binarizer(threshold=30).fit_transform(x)

transformer
```

- **preprocessing.KBinsDiscretizer**

这是将连续型变量划分为分类变量的类，能够将连续型变量排序后按顺序分箱后编码。总共包含三个重要参数：

参数	含义&输入
n_bins	每个特征中分箱的个数，默认5，一次会被运用到所有导入的特征
encode	编码的方式，默认“onehot” “onehot”: 做哑变量，之后返回一个稀疏矩阵，每一列是一个特征中的一个类别，含有该类别的样本表示为1，不含的表示为0 “ordinal”: 每个特征的每个箱都被编码为一个整数，返回每一列是一个特征，每个特征下含有不同整数编码的箱的矩阵 “onehot-dense”: 做哑变量，之后返回一个密集数组。
strategy	用来定义箱宽的方式，默认“quantile” “uniform”: 表示等宽分箱，即每个特征中的每个箱的最大值之间的差为 (特征.max() - 特征.min())/(n_bins) “quantile”: 表示等位分箱，即每个特征中的每个箱内的样本数量都相同 “kmeans”: 表示按聚类分箱，每个箱中的值到最近的一维k均值聚类的簇心得距离都相同

```
from sklearn.preprocessing import KBinsDiscretizer

X = data.iloc[:,0].values.reshape(-1,1)
est = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='uniform')
est.fit_transform(X)

#查看转换后分的箱：变成了一列中的三箱
set(est.fit_transform(X).ravel())

est = KBinsDiscretizer(n_bins=3, encode='onehot', strategy='uniform')
#查看转换后分的箱：变成了哑变量
est.fit_transform(X).toarray()
```

3 特征选择 feature_selection

当数据预处理完成后，我们就要开始进行特征工程了。

特征提取 (feature extraction)	特征创造 (feature creation)	特征选择 (feature selection)
从文字，图像，声音等其他非结构化数据中提取新信息作为特征。比如说，从淘宝宝贝的名称中提取出产品类别，产品颜色，是否是网红产品等等。	把现有特征进行组合，或互相计算，得到新的特征。比如说，我们有一列特征是速度，一列特征是距离，我们就可以通过让两列相乘，创造新的特征：通过距离所花的时间。	从所有的特征中，选择出有意义，对模型有帮助的特征，以避免必须将所有特征都导入模型去训练的情况。

在做特征选择之前，有三件非常重要的事：**跟数据提供者开会！跟数据提供者开会！跟数据提供者开会！**

一定要抓住给你提供数据的人，尤其是理解业务和数据含义的人，跟他们聊一段时间。技术能够让模型起飞，前提是你要和业务人员一样理解数据。所以特征选择的第一步，其实是根据我们的目标，用业务常识来选择特征。来看完整版泰坦尼克号数据中的这些特征：

	乘客 编号	存 活	舱位等 级	姓名	性别	年 龄	同船的兄弟姐 妹数量	同船的父辈 的数量	票号	乘客的体热 指标	船舱 编号	乘客登船的 港口
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...)	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

其中是否存活是我们的标签。很明显，以判断“是否存活”为目的，票号，登船的舱门，乘客编号明显是无关特征，可以直接删除。姓名，舱位等级，船舱编号，也基本可以判断是相关性比较低的特征。性别，年龄，船上的亲人数量，这些应该是相关性比较高的特征。

所以，特征工程的第一步是：理解业务。

当然了，在真正的数据应用领域，比如金融，医疗，电商，我们的数据不可能像泰坦尼克号数据的特征这样少，这样明显，那如果遇见极端情况，我们无法依赖对业务的理解来选择特征，该怎么办呢？我们有四种方法可以用来选择特征：过滤法，嵌入法，包装法，和降维算法。

```
#导入数据，让我们使用digit recognizer数据来一展身手
```

```
import pandas as pd
data = pd.read_csv(r"C:\work\learnbetter\micro-class\week 3 Preprocessing\digit
recognizer.csv")

X = data.iloc[:,1:]
y = data.iloc[:,0]

X.shape
```

这个数据量相对夸张，如果使用支持向量机和神经网络，很可能会直接跑不出来。使用KNN跑一次大概需要半个小时。
用这个数据举例，能更够体现特征工程的重要性。

3.1 Filter过滤法

过滤方法通常用作预处理步骤，特征选择完全独立于任何机器学习算法。它是根据各种统计检验中的分数以及相关性的各项指标来选择特征。



3.1.1 方差过滤

3.1.1.1 VarianceThreshold

这是通过特征本身的方差来筛选特征的类。比如一个特征本身的方差很小，就表示样本在这个特征上基本没有差异，可能特征中的大多数值都一样，甚至整个特征的取值都相同，那这个特征对于样本区分没有什么作用。**所以无论接下来的特征工程要做什么，都要优先消除方差为0的特征。** VarianceThreshold有重要参数**threshold**，表示方差的阈值，表示舍弃所有方差小于threshold的特征，不填默认为0，即删除所有的记录都相同的特征。

```
from sklearn.feature_selection import VarianceThreshold
selector = VarianceThreshold()                      #实例化，不填参数默认方差为0
X_var0 = selector.fit_transform(X)                  #获取删除不合格特征之后的新特征矩阵

#也可以直接写成 X = VarianceThreshold().fit_transform(X)

X_var0.shape
```

可以看见，我们已经删除了方差为0的特征，但是依然剩下了708多个特征，明显还需要进一步的特征选择。然而，如果我们知道我们需要多少个特征，方差也可以帮助我们将特征选择一步到位。比如说，我们希望留下一半的特征，那可以设定一个让特征总数减半的方差阈值，只要找到特征方差的中位数，再将这个中位数作为参数**threshold**的值输入就好了：

```

import numpy as np
x_fsvr = VarianceThreshold(np.median(x.var().values)).fit_transform(x)

x.var().values

np.median(x.var().values)

x_fsvr.shape

```

当特征是二分类时，特征的取值就是伯努利随机变量，这些变量的方差可以计算为：

$$Var[X] = p(1 - p)$$

其中X是特征矩阵，p是二分类特征中的一类在这个特征中所占的概率。

```

#若特征是伯努利随机变量，假设p=0.8，即二分类特征中某种分类占到80%以上的时候删除特征
x_bvar = VarianceThreshold(.8 * (1 - .8)).fit_transform(x)
x_bvar.shape

```

3.1.1.2 方差过滤对模型的影响

我们这样做了以后，对模型效果会有怎样的影响呢？在这里，我为大家准备了KNN和随机森林分别在方差过滤前和方差过滤后运行的效果和运行时间的对比。KNN是K近邻算法中的分类算法，其原理非常简单，是利用每个样本到其他样本点的距离来判断每个样本点的相似度，然后对样本进行分类。KNN必须遍历每个特征和每个样本，因而特征越多，KNN的计算也就会越缓慢。由于这一段代码对比运行时间过长，所以我为大家贴出了代码和结果。

1. 导入模块并准备数据

```

#KNN vs 随机森林在不同方差过滤效果下的对比
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.neighbors import KNeighborsClassifier as KNN
from sklearn.model_selection import cross_val_score
import numpy as np

x = data.iloc[:,1:]
y = data.iloc[:,0]

x_fsvr = VarianceThreshold(np.median(x.var().values)).fit_transform(x)

```

我们从模块neighbors导入KNeighborsClassifier缩写为KNN，导入随机森林缩写为RFC，然后导入交叉验证模块和numpy。其中未过滤的数据是X和y，使用中位数过滤后的数据是x_fsvr，都是我们之前已经运行过的代码。

2. KNN方差过滤前

```
#=====【TIME WARNING: 35mins +】=====#
cross_val_score(KNN(), X, y, cv=5).mean()

#python中的魔法命令，可以直接使用%%timeit来计算运行这个cell中的代码所需的时间
#为了计算所需的时间，需要将这个cell中的代码运行很多次（通常是7次）后求平均值，因此运行%%timeit的时间会
远远超过cell中的代码单独运行的时间

#=====【TIME WARNING: 4 hours】=====#
%%timeit
cross_val_score(KNN(), X, y, cv=5).mean()
```

[55]: cross_val_score(KNN(), X, y, cv=5).mean()

[55]: 0.9658569700264943

[56]: %%timeit
cross_val_score(KNN(), X, y, cv=5).mean()

33min 58s ± 43.9 s per loop (mean ± std. dev. of 7 runs, 1 loop each)

3. KNN方差过滤后

```
#=====【TIME WARNING: 20 mins+】=====#
cross_val_score(KNN(), X_fsvar, y, cv=5).mean()

#=====【TIME WARNING: 2 hours】=====#
%%timeit
cross_val_score(KNN(), X_fsvar, y, cv=5).mean()
```

[57]: cross_val_score(KNN(), X_fsvar, y, cv=5).mean()

[57]: 0.9659997478150573

[58]: %%timeit
cross_val_score(KNN(), X_fsvar, y, cv=5).mean()

20min ± 4min 55s per loop (mean ± std. dev. of 7 runs, 1 loop each)

可以看出，对于KNN，过滤后的效果十分明显：准确率稍有提升，但平均运行时间减少了10分钟，特征选择过后算法的效果上升了1/3。那随机森林又如何呢？

4. 随机森林方差过滤前

```
cross_val_score(RFC(n_estimators=10, random_state=0), X, y, cv=5).mean()
```

```
[52]: #随机森林 - 方差过滤前
cross_val_score(RFC(n_estimators=10,random_state=0),X,y,cv=5).mean()
```

[52]: 0.9380003861799541

```
[53]: %%timeit
#查看一下模型运行的时间
cross_val_score(RFC(n_estimators=10,random_state=0),X,y,cv=5).mean()

11.5 s ± 305 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

5. 随机森林方差过滤后

```
cross_val_score(RFC(n_estimators=10,random_state=0),X_fsvar,y,cv=5).mean()
```

```
[50]: #随机森林 - 方差过滤后
cross_val_score(RFC(n_estimators=10,random_state=0),X_fsvar,y,cv=5).mean()
```

[50]: 0.9388098166696807

```
[51]: %%timeit
cross_val_score(RFC(n_estimators=10,random_state=0),X_fsvar,y,cv=5).mean()

11.1 s ± 72 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

首先可以观察到的是，随机森林的准确率略逊于KNN，但运行时间却连KNN的1%都不到，只需要十几秒钟。其次，方差过滤后，随机森林的准确率也微弱上升，但运行时间却几乎是没什么变化，依然是11秒钟。

为什么随机森林运行如此之快？为什么方差过滤对随机森林没很大的影响？这是由于两种算法的原理中涉及到的计算量不同。最近邻算法KNN，单棵决策树，支持向量机SVM，神经网络，回归算法，都需要遍历特征或升维来进行运算，所以他们本身的运算量就很大，需要的时间就很长，因此方差过滤这样的特征选择对他们来说就尤为重要。但对于不需要遍历特征的算法，比如随机森林，它随机选取特征进行分枝，本身运算就非常快速，因此特征选择对它来说效果平平。这其实很容易理解，无论过滤法如何降低特征的数量，随机森林也只会选取固定数量的特征来建模；而最近邻算法就不同了，特征越少，距离计算的维度就越少，模型明显会随着特征的减少变得轻量。因此，过滤法的主要对象是：**需要遍历特征或升维的算法们**，而过滤法的主要目的是：**在维持算法表现的前提下，帮助算法们降低计算成本**。

思考：过滤法对随机森林无效，却对树模型有效？

从算法原理上来说，传统决策树需要遍历所有特征，计算不纯度后进行分枝，而随机森林却是随机选择特征进行计算和分枝，因此随机森林的运算更快，过滤法对随机森林无用，对决策树却有用

在sklearn中，决策树和随机森林都是随机选择特征进行分枝（不记得的小伙伴可以去复习第一章：决策树，参数random_state），但决策树在建模过程中随机抽取的特征数目却远远超过随机森林当中每棵树随机抽取的特征数目（比如说对于这个780维的数据，随机森林每棵树只会抽取10~20个特征，而决策树可能会抽取300~400个特征），因此，过滤法对随机森林无用，却对决策树有用

也因此，在sklearn中，随机森林中的每棵树都比单独的一棵决策树简单得多，高维数据下的随机森林的计算比决策树快很多。

对受影响的算法来说，我们可以将方差过滤的影响总结如下：

	阈值很小 被过滤掉的特征比较少	阈值比较大 被过滤掉的特征有很多
模型表现	不会有太大影响	可能变更好，代表被滤掉的特征大部分是噪音 也可能变糟糕，代表被滤掉的特征中很多都是有效特征
运行时间	可能降低模型的运行时间 基于方差很小的特征有多少 当方差很小的特征不多时 对模型没有太大影响	一定能够降低模型的运行时间 算法在遍历特征时的计算越复杂，运行时间下降得越多

在我们的对比当中，我们使用的方差阈值是特征方差的中位数，因此属于阈值比较大，过滤掉的特征比较多的情况。我们可以观察到，无论是KNN还是随机森林，在过滤掉一半特征之后，模型的精确度都上升了。这说明被我们过滤掉的特征在当前随机模式(random_state = 0)下大部分是噪音。那我们就可以保留这个去掉了一半特征的数据，来为之后的特征选择做准备。当然，如果过滤之后模型的效果反而变差了，我们就可以认为，被我们过滤掉的特征中有很多都有有效特征，那我们就放弃过滤，使用其他手段来进行特征选择。

思考：虽然随机森林算得快，但KNN的效果比随机森林更好？

调整一下n_estimators试试看吧O(∩_∩)O，随机森林是个非常强大的模型哦~

3.1.1.3 选取超参数threshold

我们怎样知道，方差过滤掉的到底时噪音还是有效特征呢？过滤后模型到底会变好还是会变坏呢？答案是：每个数据集不一样，只能自己去尝试。这里的方差阈值，其实相当于是一个超参数，要选定最优的超参数，我们可以画学习曲线，找模型效果最好的点。但现实中，我们往往不会这样做，因为这样会耗费大量的时间。我们只会使用阈值为0或者阈值很小的方差过滤，来为我们优先消除一些明显用不到的特征，然后我们会选择更优的特征选择方法继续削减特征数量。

3.1.2 相关性过滤

方差挑选完毕之后，我们就要考虑下一个问题：相关性了。我们希望选出与标签相关且有意义的特征，因为这样的特征能够为我们提供大量信息。如果特征与标签无关，那只会白白浪费我们的计算内存，可能还会给模型带来噪音。在sklearn当中，我们有三种常用的方法来评判特征与标签之间的相关性：卡方，F检验，互信息。

3.1.2.1 卡方过滤

卡方过滤是专门针对离散型标签（即分类问题）的相关性过滤。卡方检验类**feature_selection.chi2**计算每个非负特征和标签之间的卡方统计量，并依照卡方统计量由高到低为特征排名。再结合**feature_selection.SelectKBest**这个可以输入“评分标准”来选出前K个分数最高的特征的类，我们可以借此除去最可能独立于标签，与我们分类目的无关的特征。

另外，如果卡方检验检测到某个特征中所有的值都相同，会提示我们使用方差先进行方差过滤。并且，刚才我们已经验证过，当我们使用方差过滤筛选掉一半的特征后，模型的表现时提升的。因此在这里，我们使用**threshold=中位数**时完成的方差过滤的数据来做卡方检验（如果方差过滤后模型的表现反而降低了，那我们就不会使用方差过滤后的数据，而是使用原数据）：

```
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.model_selection import cross_val_score
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

#假设在这里我一直我需要300个特征
x_fschi = SelectKBest(chi2, k=300).fit_transform(x_fvar, y)
x_fschi.shape
```

验证一下模型的效果如何：

```
cross_val_score(RFC(n_estimators=10, random_state=0), x_fschi, y, cv=5).mean()
```

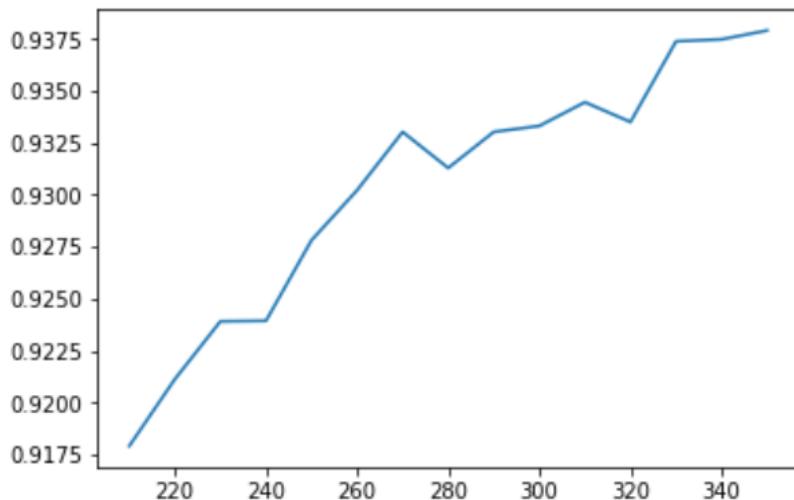
可以看出，模型的效果降低了，这说明我们在设定k=300的时候删除了与模型相关且有效的特征，我们的K值设置得过小，要么我们需要调整K值，要么我们必须放弃相关性过滤。当然，如果模型的表现提升，则说明我们的相关性过滤是有效的，是过滤掉了模型的噪音的，这时候我们就保留相关性过滤的结果。

3.1.2.2 选取超参数K

那如何设置一个最佳的K值呢？在现实数据中，数据量很大，模型很复杂的时候，我们也许不能先去跑一遍模型看看效果，而是希望最开始就能够选择一个最优的超参数k。那第一个方法，就是我们之前提过的学习曲线：

```
#=====【TIME WARNING: 5 mins】=====#
%matplotlib inline
import matplotlib.pyplot as plt

score = []
for i in range(390, 200, -10):
    x_fsch = SelectKBest(chi2, k=i).fit_transform(x_fvar, y)
    once = cross_val_score(RFC(n_estimators=10, random_state=0), x_fsch, y, cv=5).mean()
    score.append(once)
plt.plot(range(350, 200, -10), score)
plt.show()
```



通过这条曲线，我们可以观察到，随着K值的不断增加，模型的表现不断上升，这说明，K越大越好，数据中所有的特征都是与标签相关的。但是运行这条曲线的时间同样也是非常地长，接下来我们就来介绍一种更好的选择k的方法：看p值选择k。

卡方检验的本质是推测两组数据之间的差异，其检验的原假设是“两组数据是相互独立的”。卡方检验返回卡方值和P值两个统计量，其中卡方值很难界定有效的范围，而p值，我们一般使用0.01或0.05作为显著性水平，即p值判断的边界，具体我们可以这样来看：

P值	<=0.05或0.01	>0.05或0.01
数据差异	差异不是自然形成的	这些差异是很自然的样本误差
相关性	两组数据是相关的	两组数据是相互独立的
原假设	拒绝原假设，接受备择假设	接受原假设

从特征工程的角度，我们希望选取卡方值很大，p值小于0.05的特征，即和标签是相关联的特征。而调用SelectKBest之前，我们可以直接从chi2实例化后的模型中获得各个特征所对应的卡方值和P值。

```

chivalue, pvalues_chi = chi2(x_fsvar,y)

chivalue

pvalues_chi

#k取多少? 我们想要消除所有p值大于设定值, 比如0.05或0.01的特征:
k = chivalue.shape[0] - (pvalues_chi > 0.05).sum()

#x_fschi = SelectKBest(chi2, k=填写具体的k).fit_transform(x_fsvar, y)
#cross_val_score(RFC(n_estimators=10, random_state=0),x_fschi,y,cv=5).mean()

```

可以观察到, 所有特征的p值都是0, 这说明对于digit recognizer这个数据集来说, 方差验证已经把所有和标签无关的特征都剔除了, 或者这个数据集本身就不含与标签无关的特征。在这种情况下, 舍弃任何一个特征, 都会舍弃对模型有用的信息, 而使模型表现下降, 因此在我们对计算速度感到满意时, 我们不需要使用相关性过滤来过滤我们的数据。如果我们认为运算速度太缓慢, 那我们可以酌情删除一些特征, 但前提是, 我们必须牺牲模型的表现。接下来, 我们试试看用其他的相关性过滤方法验证一下我们在这个数据集上的结论。

3.1.2.3 F检验

F检验, 又称ANOVA, 方差齐性检验, 是用来捕捉每个特征与标签之间的线性关系的过滤方法。它即可以做回归也可以做分类, 因此包含**feature_selection.f_classif** (F检验分类) 和**feature_selection.f_regression** (F检验回归) 两个类。其中F检验分类用于标签是离散型变量的数据, 而F检验回归用于标签是连续型变量的数据。

和卡方检验一样, 这两个类需要和类**SelectKBest**连用, 并且我们也可以直接通过输出的统计量来判断我们到底要设置一个什么样的K。需要注意的是, F检验在数据服从正态分布时效果会非常稳定, 因此如果使用F检验过滤, 我们会先将数据转换成服从正态分布的方式。

F检验的本质是寻找两组数据之间的线性关系, 其原假设是“数据不存在显著的线性关系”。它返回F值和p值两个统计量。和卡方过滤一样, **我们希望选取p值小于0.05或0.01的特征, 这些特征与标签时显著线性相关的**, 而p值大于0.05或0.01的特征则被我们认为是和标签没有显著线性关系的特征, 应该被删除。以F检验的分类为例, 我们继续在数字数据集上来进行特征选择:

```

from sklearn.feature_selection import f_classif

F, pvalues_f = f_classif(x_fsvar,y)

F

pvalues_f

k = F.shape[0] - (pvalues_f > 0.05).sum()

#x_fsF = SelectKBest(f_classif, k=填写具体的k).fit_transform(x_fsvar, y)
#cross_val_score(RFC(n_estimators=10, random_state=0),x_fsF,y,cv=5).mean()

```

得到的结论和我们用卡方过滤得到的结论一模一样: 没有任何特征的p值大于0.01, 所有的特征都是和标签相关的, 因此我们不需要相关性过滤。

3.1.2.4 互信息法

互信息法是用来捕捉每个特征与标签之间的任意关系（包括线性和非线性关系）的过滤方法。和F检验相似，它既可以做回归也可以做分类，并且包含两个类**feature_selection.mutual_info_classif**（互信息分类）和**feature_selection.mutual_info_regression**（互信息回归）。这两个类的用法和参数都和F检验一模一样，不过互信息法比F检验更加强大，F检验只能够找出线性关系，而互信息法可以找出任意关系。

互信息法不返回p值或F值类似的统计量，它返回“每个特征与目标之间的互信息量的估计”，这个估计量在[0,1]之间取值，为0则表示两个变量独立，为1则表示两个变量完全相关。以互信息分类为例的代码如下：

```
from sklearn.feature_selection import mutual_info_classif as MIC

result = MIC(X_fsvr,y)

k = result.shape[0] - sum(result <= 0)

#X_fsmic = SelectKBest(MIC, k=填写具体的k).fit_transform(X_fsvr, y)
#cross_val_score(RFC(n_estimators=10, random_state=0), X_fsmic, y, cv=5).mean()
```

所有特征的互信息量估计都大于0，因此所有特征都与标签相关。

当然了，无论是F检验还是互信息法，大家也都可以使用学习曲线，只是使用统计量的方法会更加高效。当统计量判断已经没有特征可以删除时，无论用学习曲线如何跑，删除特征都只会降低模型的表现。当然了，如果数据量太庞大，模型太复杂，我们还是可以牺牲模型表现来提升模型速度，一切都看大家的具体需求。

3.1.3 过滤法总结

到这里我们学习了常用的基于过滤法的特征选择，包括方差过滤，基于卡方，F检验和互信息的相关性过滤，讲解了各个过滤的原理和面临的问题，以及怎样调这些过滤类的超参数。通常来说，我会建议，先使用方差过滤，然后使用互信息法来捕捉相关性，不过了解各种各样的过滤方式也是必要的。所有信息被总结在下表，大家自取：

类	说明	超参数的选择
VarianceThreshold	方差过滤，可输入方差阈值，返回方差大于阈值的新特征矩阵	看具体数据究竟是含有更多噪声还是更多有效特征 一般就使用0或1来筛选 也可以画学习曲线或取中位数跑模型来帮助确认
SelectKBest	用来选取K个统计量结果最佳的特征，生成符合统计量要求的新特征矩阵	看配合使用的统计量
chi2	卡方检验，专用于分类算法，捕捉相关性	追求p小于显著性水平的特征
f_classif	F检验分类，只能捕捉线性相关性 要求数据服从正态分布	追求p小于显著性水平的特征
f_regression	F检验回归，只能捕捉线性相关性 要求数据服从正态分布	追求p小于显著性水平的特征
mutual_info_classif	互信息分类，可以捕捉任何相关性 不能用于稀疏矩阵	追求互信息估计大于0的特征
mutual_info_regression	互信息回归，可以捕捉任何相关性 不能用于稀疏矩阵	追求互信息估计大于0的特征

3.2 Embedded嵌入法

嵌入法是一种让算法自己决定使用哪些特征的方法，即特征选择和算法训练同时进行。在使用嵌入法时，我们先使用某些机器学习的算法和模型进行训练，得到各个特征的权值系数，根据权值系数从大到小选择特征。这些权值系数往往代表了特征对于模型的某种贡献或某种重要性，比如决策树和树的集成模型中的feature_importances_属性，可以列出各个特征对树的建立的贡献，我们就可以基于这种贡献的评估，找出对模型建立最有用的特征。因此相比于过滤法，嵌入法的结果会更加精确到模型的效用本身，对于提高模型效力有更好的效果。并且，由于考虑特征对模型的贡献，因此无关的特征（需要相关性过滤的特征）和无区分度的特征（需要方差过滤的特征）都会因为缺乏对模型的贡献而被删除掉，可谓是过滤法的进化版。

算法依赖于模型评估完成特征子集选择



然而，嵌入法也不是没有缺点。

过滤法中使用的统计量可以使用统计知识和常识来查找范围（如p值应当低于显著性水平0.05），而嵌入法中使用的权值系数却没有这样的范围可找——我们可以说，权值系数为0的特征对模型丝毫没有作用，但当大量特征都对模型有贡献且贡献不一时，我们就很难去界定一个有效的临界值。这种情况下，模型权值系数就是我们的超参数，我们或许需要学习曲线，或者根据模型本身的某些性质去判断这个超参数的最佳值究竟应该是多少。在我们之后的学习当中，每次讲解新的算法，我都会为大家提到这个算法中的特征工程是如何处理，包括具体到每个算法的嵌入法如何使用。在这堂课中，我们会为大家讲解随机森林和决策树模型的嵌入法。

另外，嵌入法引入了算法来挑选特征，因此其计算速度也会和应用的算法有很大的关系。如果采用计算量很大，计算缓慢的算法，嵌入法本身也会非常耗时耗力。并且，在选择完毕之后，我们还是需要自己来评估模型。

- **feature_selection.SelectFromModel**

```
class sklearn.feature_selection.SelectFromModel(estimator, threshold=None, prefit=False, norm_order=1, max_features=None)
```

SelectFromModel是一个元变换器，可以与任何在拟合后具有coef_，feature_importances_属性或参数中可选惩罚项的评估器一起使用（比如随机森林和树模型就具有属性feature_importances_，逻辑回归就带有l1和l2惩罚项，线性支持向量机也支持l2惩罚项）。

对于有feature_importances_的模型来说，若重要性低于提供的阈值参数，则认为这些特征不重要并被移除。feature_importances_的取值范围是[0,1]，如果设置阈值很小，比如0.001，就可以删除那些对标签预测完全没贡献的特征。如果设置得很接近1，可能只有一两个特征能够被留下。

选读：使用惩罚项的模型的嵌入法

而对于使用惩罚项的模型来说，正则化惩罚项越大，特征在模型中对应的系数就会越小。当正则化惩罚项大到一定的程度的时候，部分特征系数会变成0，当正则化惩罚项继续增大到一定程度时，所有的特征系数都会趋于0。但是我们会发现一部分特征系数会更容易先变成0，这部分系数就是可以筛掉的。也就是说，我们选择特征系数较大的特征。另外，支持向量机和逻辑回归使用参数C来控制返回的特征矩阵的稀疏性，参数C越小，返回的特征越少。Lasso回归，用alpha参数来控制返回的特征矩阵，alpha的值越大，返回的特征越少。

参数	说明
estimator	使用的模型评估器，只要是带feature_importances_或者coef_属性，或带有l1和l2惩罚项的模型都可以使用
threshold	特征重要性的阈值，重要性低于这个阈值的特征都将被删除
prefit	默认False，判断是否将实例化后的模型直接传递给构造函数。如果为True，则必须直接调用fit和transform，不能使用fit_transform，并且SelectFromModel不能与cross_val_score，GridSearchCV和克隆估计器的类似实用程序一起使用。
norm_order	k可输入非零整数，正无穷，负无穷，默认值为1 在评估器的coef_属性高于一维的情况下，用于过滤低于阈值的系数的向量的范数的阶数。
max_features	在阈值设定下，要选择的最大特征数。要禁用阈值并仅根据max_features选择，请设置threshold = -np.inf

我们重点要考虑的是前两个参数。在这里，我们使用随机森林为例，则需要学习曲线来帮助我们寻找最佳特征值。

```

from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier as RFC

RFC_ = RFC(n_estimators = 10, random_state=0)

X_embedded = SelectFromModel(RFC_, threshold=0.005).fit_transform(X,y)

#在这里我只想取出来有限的特征。0.005这个阈值对于有780个特征的数据来说，是非常高的阈值，因为平均每个特征
#只能分到大约0.001的feature_importances_

X_embedded.shape

#模型的维度明显被降低了
#同样的，我们也可以画学习曲线来找最佳阈值

=====【TIME WARNING: 10 mins】=====

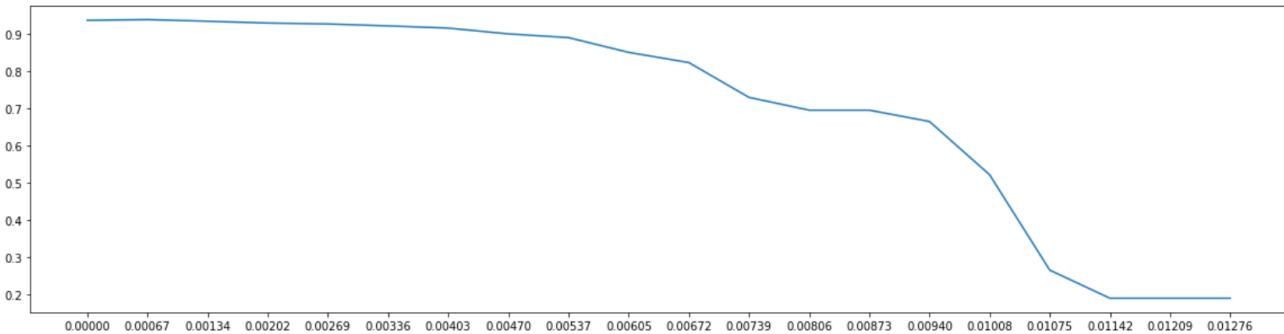
import numpy as np
import matplotlib.pyplot as plt

RFC_.fit(X,y).feature_importances_

threshold = np.linspace(0,(RFC_.fit(X,y).feature_importances_).max(),20)

score = []
for i in threshold:
    X_embedded = SelectFromModel(RFC_, threshold=i).fit_transform(X,y)
    once = cross_val_score(RFC_,X_embedded,y,cv=5).mean()
    score.append(once)
plt.plot(threshold,score)
plt.show()

```



从图像上来看，随着阈值越来越高，模型的效果逐渐变差，被删除的特征越来越多，信息损失也逐渐变大。但是在0.00134之前，模型的效果都可以维持在0.93以上，因此我们可以从中挑选一个数值来验证一下模型的效果。

```

X_embedded = SelectFromModel(RFC_, threshold=0.00067).fit_transform(X,y)
X_embedded.shape

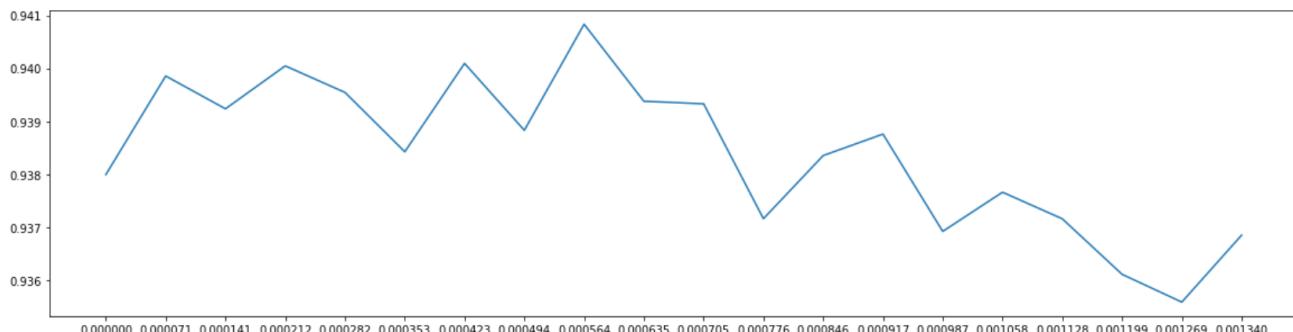
cross_val_score(RFC_,X_embedded,y,cv=5).mean()

```

可以看出，特征个数瞬间缩小到324多，这比我们在方差过滤的时候选择中位数过滤出来的结果392列要小，并且交叉验证分数0.9399高于方差过滤后的结果0.9388，这是由于嵌入法比方差过滤更具体到模型的表现的缘故，换一个算法，使用同样的阈值，效果可能就没有这么好了。

和其他调参一样，我们可以在第一条学习曲线后选定一个范围，使用细化的学习曲线来找到最佳值：

```
=====【TIME WARNING: 10 mins】=====#
score2 = []
for i in np.linspace(0,0.00134,20):
    X_embedded = SelectFromModel(RFC_, threshold=i).fit_transform(X,y)
    once = cross_val_score(RFC_,X_embedded,y,cv=5).mean()
    score2.append(once)
plt.figure(figsize=[20,5])
plt.plot(np.linspace(0,0.00134,20),score2)
plt.xticks(np.linspace(0,0.00134,20))
plt.show()
```



查看结果，果然0.00067并不是最高点，真正的最高点0.000564已经将模型效果提升到了94%以上。我们使用0.000564来跑一跑我们的SelectFromModel：

```
X_embedded = SelectFromModel(RFC_, threshold=0.000564).fit_transform(X,y)
X_embedded.shape

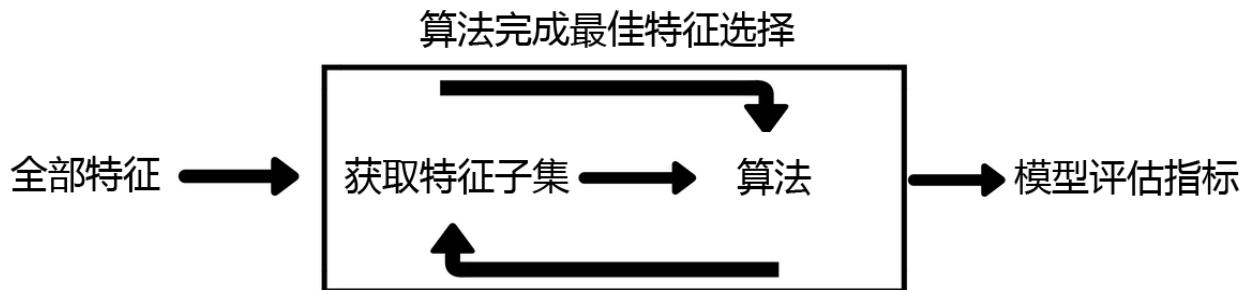
cross_val_score(RFC_,X_embedded,y,cv=5).mean()

=====【TIME WARNING: 2 min】=====
#我们可能已经找到了现有模型下的最佳结果，如果我们调整一下随机森林的参数呢？
cross_val_score(RFC(n_estimators=100,random_state=0),X_embedded,y,cv=5).mean()
```

得出的特征数目依然小于方差筛选，并且模型的表现也比没有筛选之前更高，已经完全可以和计算一次半小时的KNN相匹敌（KNN的准确率是96.58%），接下来再对随机森林进行调参，准确率应该还可以再升高不少。可见，在嵌入法下，我们很容易就能够实现特征选择的目标：减少计算量，提升模型表现。因此，比起要思考很多统计量的过滤法来说，嵌入法可能是更有效的一种方法。然而，在算法本身很复杂的时候，过滤法的计算远远比嵌入法要快，所以大型数据中，我们还是会优先考虑过滤法。

3.3 Wrapper包装法

包装法也是一个特征选择和算法训练同时进行的方法，与嵌入法十分相似，它也是依赖于算法自身的选择，比如 `coef_` 属性或 `feature_importances_` 属性来完成特征选择。但不同的是，我们往往使用一个目标函数作为黑盒来帮助我们选取特征，而不是自己输入某个评估指标或统计量的阈值。包装法在初始特征集上训练评估器，并且通过 `coef_` 属性或通过 `feature_importances_` 属性获得每个特征的重要性。然后，从当前的一组特征中修剪最不重要的特征。在修剪的集合上递归地重复该过程，直到最终到达所需数量的要选择的特征。区别于过滤法和嵌入法的一次训练解决所有问题，包装法要使用特征子集进行多次训练，因此它所需要的计算成本是最高的。



注意，在这个图中的“算法”，指的不是我们最终用来导入数据的分类或回归算法（即不是随机森林），而是专业的数据挖掘算法，即我们的目标函数。这些数据挖掘算法的核心功能就是选取最佳特征子集。

最典型的目标函数是递归特征消除法（Recursive feature elimination, 简写为RFE）。它是一种贪婪的优化算法，旨在找到性能最佳的特征子集。它反复创建模型，并在每次迭代时保留最佳特征或剔除最差特征，下一次迭代时，它会使用上一次建模中没有被选中的特征来构建下一个模型，直到所有特征都耗尽为止。然后，它根据自己保留或剔除特征的顺序来对特征进行排名，最终选出一个最佳子集。包装法的效果是所有特征选择方法中最利于提升模型表现的，它可以使用很少的特征达到很优秀的效果。除此之外，在特征数目相同时，包装法和嵌入法的效果能够匹敌，不过它比嵌入法算得更慢，所以也不适用于太大型的数据。相比之下，包装法是最能保证模型效果的特征选择方法。

- `feature_selection.RFE`

```
class sklearn.feature_selection.RFE (estimator, n_features_to_select=None, step=1, verbose=0)
```

参数 `estimator` 是需要填写的实例化后的评估器，`n_features_to_select` 是想要选择的特征个数，`step` 表示每次迭代中希望移除的特征个数。除此之外，RFE类有两个很重要的属性，`.support_`：返回所有的特征是否最后被选中的布尔矩阵，以及`.ranking_` 返回特征的按次数迭代中综合重要性的排名。类 `feature_selection.RFECV` 会在交叉验证循环中执行RFE以找到最佳数量的特征，增加参数 `cv`，其他用法都和RFE一模一样。

```

from sklearn.feature_selection import RFE
RFC_ = RFC(n_estimators =10, random_state=0)
selector = RFE(RFC_, n_features_to_select=340, step=50).fit(x, y)

selector.support_.sum()

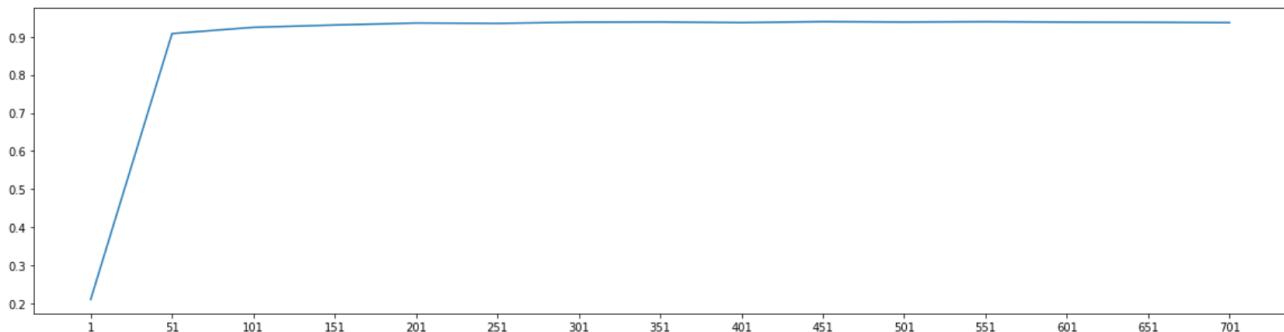
selector.ranking_

X_wrapper = selector.transform(x)

cross_val_score(RFC_, X_wrapper, y, cv=5).mean()
  
```

我们也可以对包装法画学习曲线：

```
#=====【TIME WARNING: 15 mins】=====#
score = []
for i in range(1,751,50):
    X_wrapper = RFE(RFC_,n_features_to_select=i, step=50).fit_transform(x,y)
    once = cross_val_score(RFC_,X_wrapper,y,cv=5).mean()
    score.append(once)
plt.figure(figsize=[20,5])
plt.plot(range(1,751,50),score)
plt.xticks(range(1,751,50))
plt.show()
```



明显能够看出，在包装法下面，应用50个特征时，模型的表现就已经达到了90%以上，比嵌入法和过滤法都高效很多。我们可以放大图像，寻找模型变得非常稳定的点来画进一步的学习曲线（就像我们在嵌入法中做的那样）。如果我们此时追求的是最大化降低模型的运行时间，我们甚至可以直接选择50作为特征的数目，这是一个在缩减了94%的特征的基础上，还能保证模型表现在90%以上的特征组合，不可谓不高效。

同时，我们提到过，在特征数目相同时，包装法能够在效果上匹敌嵌入法。试试看如果我们也使用340作为特征数目，运行一下，可以感受一下包装法和嵌入法哪一个的速度更加快。由于包装法效果和嵌入法相差不多，在更小的范围内使用学习曲线，我们也可以将包装法的效果调得很好，大家可以去试试看。

3.4 特征选择总结

至此，我们讲完了降维之外的所有特征选择的方法。这些方法的代码都不难，但是每种方法的原理都不同，并且都涉及到不同调整方法的超参数。经验来说，过滤法更快速，但更粗糙。包装法和嵌入法更精确，比较适合具体到算法去调整，但计算量比较大，运行时间长。当数据量很大的时候，优先使用方差过滤和互信息法调整，再上其他特征选择方法。使用逻辑回归时，优先使用嵌入法。使用支持向量机时，优先使用包装法。迷茫的时候，从过滤法走起，看具体数据具体分析。

其实特征选择只是特征工程中的第一步。真正的高手，往往使用特征创造或特征提取来寻找高级特征。在Kaggle之类的算法竞赛中，很多高分团队都是在高级特征上做文章，而这是比调参和特征选择更难的，提升算法表现的高深方法。特征工程非常深奥，虽然我们日常可能用到不多，但其实它非常美妙。若大家感兴趣，也可以自己去网上搜一搜，多读多看多试多想，技术逐渐会成为你的囊中之物。

4 sklearn课程12周提纲

菜菜的sklearn课堂

日期	期数	主题	涉及的sklearn模块
11月7日	01期	决策树	tree
11月14日	02期	随机森林	ensemble
11月21日	03期	数据预处理和特征工程	preprocessing, impute, feature_selection
11月28日	04期	主成分分析	decomposition
12月5日	05期	逻辑回归	linear_model
12月12日	06期	K-Means	cluster
12月19日	07期	SVM (1)	svm
12月26日	08期	SVM (2)	svm
1月2日	09期	线性回归	linear_model
1月9日	10期	朴素贝叶斯	naive_bayes
1月16日	11期	sklearn中的数据产生	datasets
1月23日	12期	神经网络	neural_network

菜菜的scikit-learn课堂04



sklearn中的降维算法PCA和SVD

小伙伴们晚上好~o(—▽—)ブ

我是菜菜，这里是我的sklearn课堂第四期，今晚的直播内容是降维算法PCA和SVD~

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~



菜菜的scikit-learn课堂04

sklearn中的降维算法PCA和SVD

1 概述

1.1 从什么叫“维度”说开来

1.2 sklearn中的降维算法

2 PCA与SVD

2.1 降维究竟是怎样实现?

2.2 重要参数n_components

2.2.1 迷你案例：高维数据的可视化

2.2.2 最大似然估计自选超参数

2.2.3 按信息量占比选超参数

2.3 PCA中的SVD

2.3.1 PCA中的SVD哪里来?

2.3.2 重要参数svd_solver 与 random_state

2.3.3 重要属性components_

2.4 重要接口inverse_transform

2.4.1 迷你案例：用人脸识别看PCA降维后的信息保存量

2.4.2 迷你案例：用PCA做噪音过滤

2.5 重要接口，参数和属性总结

3 案例：PCA对手写数字数据集的降维

4 附录

4.1 PCA参数列表

4.2 PCA属性列表

4.3 PCA接口列表

1 概述

1.1 从什么叫“维度”说开来

在过去的三周里，我们已经带大家认识了两个算法和数据预处理过程。期间，我们不断提到一些语言，比如说：随机森林是通过随机抽取特征来建树，以避免高维计算；再比如说，sklearn中导入特征矩阵，必须是至少二维；上周我们讲解特征工程，还特地提到了，特征选择的目的是通过降维来降低算法的计算成本……这些语言都很正常地被我用来使用，直到有一天，一个小伙伴问了我，“维度”到底是什么？

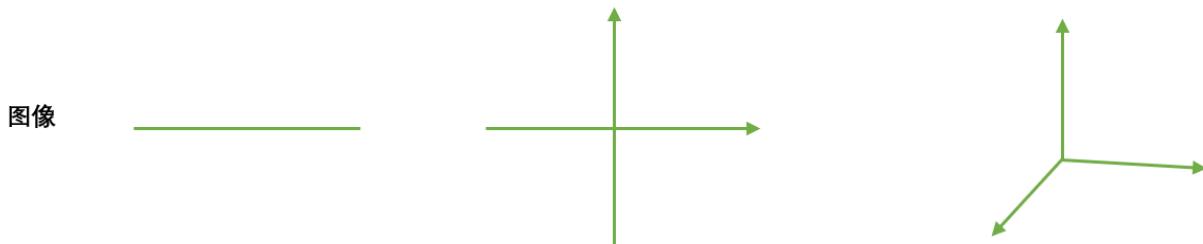
对于数组和Series来说，维度就是功能shape返回的结果，shape中返回了几个数字，就是几维。索引以外的数据，不分行列的叫一维（此时shape返回唯一的维度上的数据个数），有行列之分叫二维（shape返回行x列），也称为表。一张表最多二维，复数的表构成了更高的维度。当一个数组中存在2张3行4列的表时，shape返回的是（更高维，行，列）。当数组中存在2组2张3行4列的表时，数据就是4维，shape返回(2,2,3,4)。

	一维	二维	三维
数组	array([0., 0.])	array([[0., 0., 0., 0.], [0., 0., 0., 0.], [0., 0., 0., 0.]])	array([[[0., 0., 0., 0.], [0., 0., 0., 0.], [0., 0., 0., 0.]]], [[0., 0., 0., 0.], [0., 0., 0., 0.], [0., 0., 0., 0.]]])
Series	shape (2,)	shape (3,4)	shape (2,3,4)

数组中的每一张表，都可以是一个特征矩阵或一个DataFrame，这些结构永远只有一张表，所以一定有行列，其中行是样本，列是特征。针对每一张表，维度指的是样本的数量或特征的数量，一般无特别说明，指的都是特征的数量。除了索引之外，一个特征是一维，两个特征是二维，n个特征是n维。

	特征1		特征1 特征2		特征1 特征2 特征3		
DataFrame	0	0.0	0	0.0	0.0	0.0	0.0
特征矩阵	1	0.0	1	0.0	0.0	1	0.0

对图像来说，维度就是图像中特征向量的数量。特征向量可以理解为是坐标轴，一个特征向量定义一条直线，是一维，两个相互垂直的特征向量定义一个平面，即一个直角坐标系，就是二维，三个相互垂直的特征向量定义一个空间，即一个立体直角坐标系，就是三维。三个以上的特征向量相互垂直，定义人眼无法看见，也无法想象的高维空间。



降维算法中的“降维”，指的是降低特征矩阵中特征的数量。上周的课中我们说过，降维的目的是为了让算法运算更快，效果更好，但其实还有另一种需求：数据可视化。从上面的图我们其实可以看得出，图像和特征矩阵的维度是可以相互对应的，即一个特征对应一个特征向量，对应一条坐标轴。所以，三维及以下的特征矩阵，是可以被可视化的，这可以帮助我们很快地理解数据的分布，而三维以上特征矩阵的则不能被可视化，数据的性质也就比较难理解。

1.2 sklearn中的降维算法

sklearn中降维算法都被包括在模块decomposition中，这个模块本质是一个矩阵分解模块。在过去的十年中，如果要讨论算法进步的先锋，矩阵分解可以说是独树一帜。矩阵分解可以用在降维，深度学习，聚类分析，数据预处理，低纬度特征学习，推荐系统，大数据分析等领域。在2006年，Netflix曾经举办了一个奖金为100万美元的推荐系统算法比赛，最后的获奖者就使用了矩阵分解中的明星：奇异值分解SVD。（~o—3—）~菊安酱会讲SVD在推荐系统中的应用，大家不要错过！

类	说明
主成分分析	
decomposition.PCA	主成分分析 (PCA)
decomposition.IncrementalPCA	增量主成分分析 (IPCA)
decomposition.KernelPCA	核主成分分析 (KPCA)
decomposition.MiniBatchSparsePCA	小批量稀疏主成分分析
decomposition.SparsePCA	稀疏主成分分析 (SparsePCA)
decomposition.TruncatedSVD	截断的SVD (aka LSA)
因子分析	
decomposition.FactorAnalysis	因子分析 (FA)
独立成分分析	
decomposition.FastICA	独立成分分析的快速算法
字典学习	
decomposition.DictionaryLearning	字典学习
decomposition.MiniBatchDictionaryLearning	小批量字典学习
decomposition.dict_learning	字典学习用于矩阵分解
decomposition.dict_learning_online	在线字典学习用于矩阵分解
高级矩阵分解	
decomposition.LatentDirichletAllocation	具有在线变分贝叶斯算法的隐含狄利克雷分布
decomposition.NMF	非负矩阵分解 (NMF)
其他矩阵分解	
decomposition.SparseCoder	稀疏编码

SVD和主成分分析PCA都属于矩阵分解算法中的入门算法，都是通过分解特征矩阵来进行降维，它们也是我们今天要讲解的重点。虽然是入门算法，却不代表PCA和SVD简单：下面两张图是我在一篇SVD的论文中随意截取的两页，可以看到满满的数学公式（基本是线性代数）。要想在短短的一个小时内，给大家讲明白这些公式，我讲完不吐血大家听完也吐血了。所以今天，我会用最简单的方式为大家呈现降维算法的原理，但这注定意味着大家无法看到这个算法的全貌，在机器学习中逃避数学是邪道，所以更多原理大家自己去阅读。

6.2. Matrices for which matrix–vector products can be rapidly evaluated. In many problems in data mining and scientific computing, the cost T_{mult} of performing the matrix–vector multiplication $x \mapsto Ax$ is substantially smaller than the nominal cost $O(mn)$ for the dense case. It is not uncommon that $O(m+n)$ flops suffice. Standard examples include (i) very sparse matrices; (ii) structured matrices, such as Toeplitz operators, that can be applied using the FFT or other means; and (iii) matrices that arise from physical problems, such as discretized integral operators, that can be applied via, e.g., the fast multipole method [66].

Suppose that both A and A^* admit fast multiplies. The appropriate randomized approach for this scenario completes Stage A using Algorithm 4.1 with p constant (for the fixed-rank problem) or Algorithm 4.2 (for the fixed-precision problem) at a cost of $(k+p)T_{\text{mult}} + O(k^2m)$ flops. For Stage B, we invoke Algorithm 5.1, which requires $(k+p)T_{\text{mult}} + O(k^2(m+n))$ flops. The total cost T_{sparse} satisfies

$$T_{\text{sparse}} = 2(k+p)T_{\text{mult}} + O(k^2(m+n)). \quad (6.2)$$

As a rule of thumb, the approximation error of this procedure satisfies

$$\|A - U\Sigma V^*\| \lesssim \sqrt{kn} \cdot \sigma_{k+1}. \quad (6.3)$$

The estimate (6.3) follows from Corollary 10.9 and the discussion in §5.1. Actual errors are usually smaller.

When the singular spectrum of A decays slowly, we can incorporate q iterations of the power method (Algorithm 4.3) to obtain superior solutions to the fixed-rank problem. The computational cost increases to, cf. (6.2),

$$T_{\text{sparse}} = (2q+2)(k+p)T_{\text{mult}} + O(k^2(m+n)), \quad (6.4)$$

while the error (6.3) improves to

$$\|A - U\Sigma V^*\| \lesssim (kn)^{1/2(2q+1)} \cdot \sigma_{k+1}. \quad (6.5)$$

The estimate (6.5) takes into account the discussion in §10.4. The power scheme can also be adapted for the fixed-precision problem (§4.5).

In this setting, the classical prescription for obtaining a partial SVD is some variation of a Krylov-subspace method; see §3.3.4. These methods exhibit great diversity, so it is hard to specify a “typical” computational cost. To a first approximation, it is fair to say that in order to obtain an approximate SVD of rank k , the cost of a numerically stable implementation of a Krylov method is no less than the cost (6.2) with p set to zero. At this price, the Krylov method often obtains better accuracy than the basic randomized method obtained by combining Algorithms 4.1 and 5.1, especially for matrices whose singular values decay slowly. On the other hand, the randomized schemes are inherently more robust and allow much more freedom in organizing the computation to suit a particular application or a particular hardware architecture. The latter point is in practice of crucial importance because it is usually much faster to apply a matrix to k vectors simultaneously than it is to execute k matrix–vector multiplications consecutively. In practice, blocking and parallelism can lead to enough gain that a few steps of the power method (Algorithm 4.3) can be performed more quickly than k steps of a Krylov method.

REMARK 6.2. Any comparison between randomized sampling schemes and Krylov variants becomes complicated because of the fact that “basic” Krylov schemes such

PROPOSITION 8.2 (Perturbation of Inverses). *Suppose that $M \succcurlyeq 0$. Then*

$$I - (I + M)^{-1} \preccurlyeq M$$

Proof. Define $R = M^{1/2}$, the psd square root of M promised by [72, Thm. 7.2.6]. We have the chain of relations

$$I - (I + R^2)^{-1} = (I + R^2)^{-1}R^2 = R(I + R^2)^{-1}R \preccurlyeq R^2.$$

The first equality can be verified algebraically. The second holds because rational functions of a diagonalizable matrix, such as R , commute. The last relation follows from the conjugation rule because $(I + R^2)^{-1} \preccurlyeq I$. \blacksquare

Next, we present a generalization of the fact that the spectral norm of a psd matrix is controlled by its trace.

PROPOSITION 8.3. *We have $\|M\| \leq \|A\| + \|C\|$ for each partitioned psd matrix*

$$M = \begin{bmatrix} A & B \\ B^* & C \end{bmatrix}.$$

Proof. The variational characterization (8.1) of the spectral norm implies that

$$\begin{aligned} \|M\| &= \sup_{\|x\|^2 + \|y\|^2 = 1} \begin{bmatrix} x \\ y \end{bmatrix}^* \begin{bmatrix} A & B \\ B^* & C \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ &\leq \sup_{\|x\|^2 + \|y\|^2 = 1} (\|A\| \|x\|^2 + 2\|B\| \|x\| \|y\| + \|C\| \|y\|^2). \end{aligned}$$

The block generalization of Hadamard’s psd criterion [72, Thm. 7.7.7] states that $\|B\|^2 \leq \|A\| \|C\|$. Thus,

$$\|M\| \leq \sup_{\|x\|^2 + \|y\|^2 = 1} (\|A\|^{1/2} \|x\| + \|C\|^{1/2} \|y\|)^2 = \|A\| + \|C\|.$$

This point completes the argument. \blacksquare

8.2. Orthogonal projectors. An *orthogonal projector* is an Hermitian matrix P that satisfies the polynomial $P^2 = P$. This identity implies $0 \preccurlyeq P \preccurlyeq I$. An orthogonal projector is completely determined by its range. For a given matrix M , we write P_M for the unique orthogonal projector with $\text{range}(P_M) = \text{range}(M)$. When M has full column rank, we can express this projector explicitly:

$$P_M = M(M^*M)^{-1}M^*. \quad (8.3)$$

The orthogonal projector onto the complementary subspace, $\text{range}(P)^\perp$, is the matrix $I - P$. Our argument hinges on several other facts about orthogonal projectors.

PROPOSITION 8.4. *Suppose U is unitary. Then $U^*P_MU = P_{U^*M}$.*

Proof. Abbreviate $P = U^*P_MU$. It is clear that P is an orthogonal projector since it is Hermitian and $P^2 = P$. Evidently,

$$\text{range}(P) = U^* \text{range}(M) = \text{range}(U^*M).$$

2 PCA与SVD

在降维过程中，我们会减少特征的数量，这意味着删除数据，数据量变少则表示模型可以获取的信息会变少，模型的表现可能会因此受影响。同时，在高维数据中，必然有一些特征是不带有有效信息的（比如噪音），或者有一些特征带有的信息和其他一些特征是重复的（比如一些特征可能会线性相关）。我们希望能够找出一种办法来帮助我们衡量特征上所带的信息量，让我们在降维的过程中，能够即减少特征的数量，又保留大部分有效信息——将那些带有重复信息的特征合并，并删除那些带无效信息的特征等等——逐渐创造出能够代表原特征矩阵大部分信息的，特征更少的，新特征矩阵。

上周的特征工程课中，我们提到过一种重要的特征选择方法：方差过滤。如果一个特征的方差很小，则意味着这个特征上很可能有大量取值都相同（比如90%都是1，只有10%是0，甚至100%是1），那这一个特征的取值对样本而言就没有区分度，这种特征就不带有有效信息。从方差的这种应用就可以推断出，如果一个特征的方差很大，则说明这个特征上带有大量的信息。因此，在降维中，**PCA使用的信息量衡量指标，就是样本方差，又称可解释性方差，方差越大，特征所带的信息量越多。**

$$Var = \frac{1}{n-1} \sum_{i=1}^n (x_i - \hat{x})^2$$

Var代表一个特征的方差，n代表样本量，xi代表一个特征中的每个样本取值，xhat代表这一列样本的均值。

面试高危问题

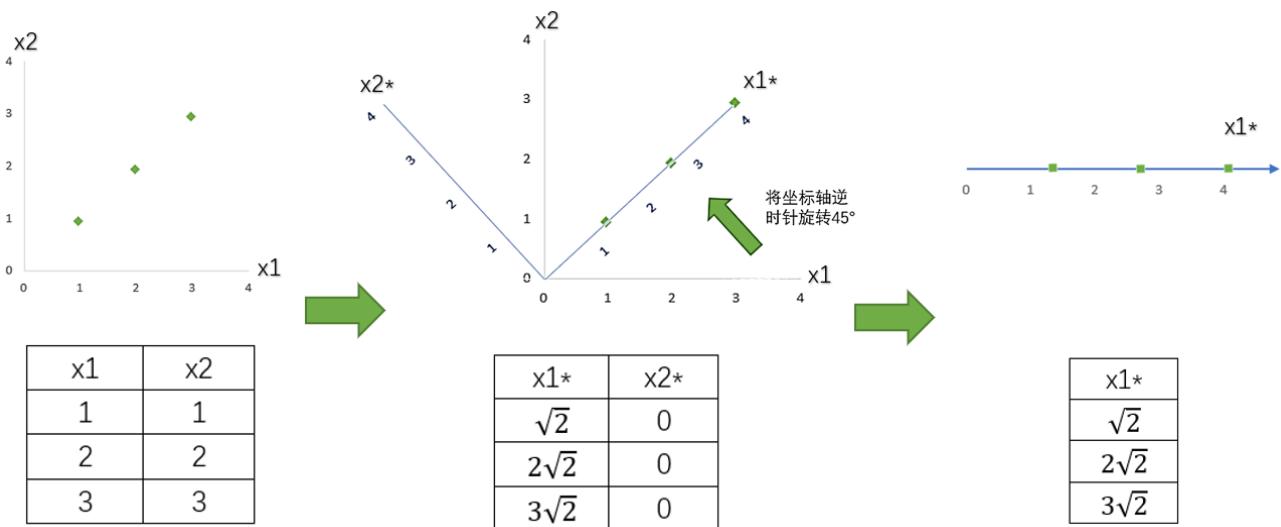
方差计算公式中为什么除数是n-1?

这是为了得到样本方差的无偏估计，更多大家可以自己去探索~

2.1 降维究竟是怎样实现？

```
class sklearn.decomposition.PCA (n_components=None, copy=True, whiten=False, svd_solver='auto', tol=0.0,
iterated_power='auto', random_state=None)
```

PCA作为矩阵分解算法的核心算法，其实没有太多参数，但不幸的是每个参数的意义和运用都很难，因为几乎每个参数都涉及到高深的数学原理。为了参数的运用和意义变得明朗，我们来看一组简单的二维数据的降维。



我们现在有一组简单的数据，有特征 x_1 和 x_2 ，三个样本数据的坐标点分别为 $(1,1)$, $(2,2)$, $(3,3)$ 。我们可以让 x_1 和 x_2 分别作为两个特征向量，很轻松地用一个二维平面来描述这组数据。这组数据现在每个特征的均值都为2，方差则等于：

$$x1_var = x2_var = \frac{(1-2)^2 + (2-2)^2 + (3-2)^2}{2} = 1$$

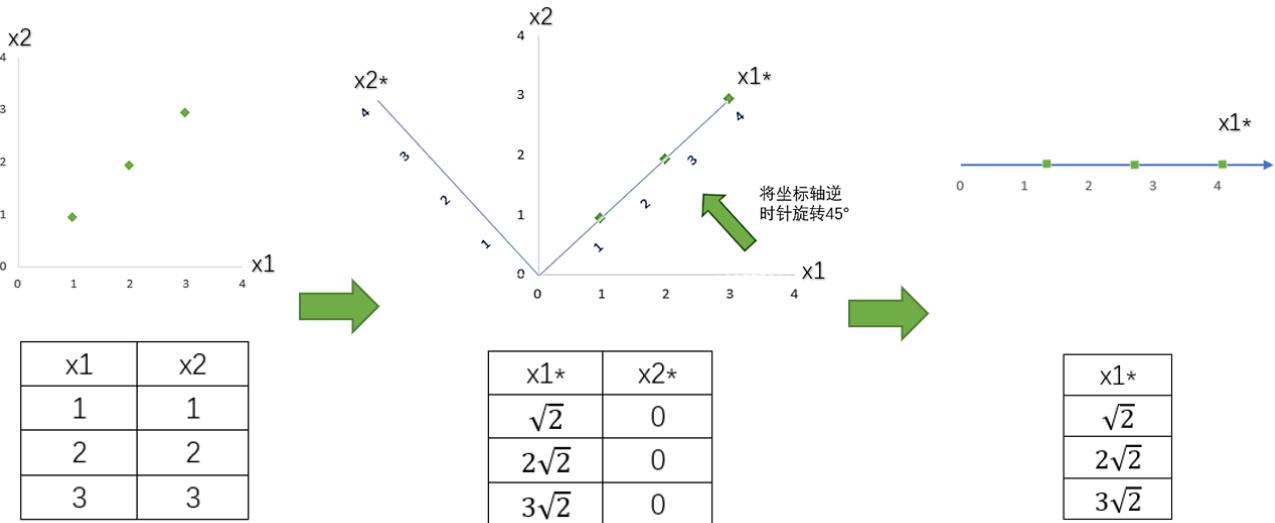
每个特征的数据一模一样，因此方差也都为1，数据的方差总和是2。

现在我们的目标是：只用一个特征向量来描述这组数据，即将二维数据降为一维数据，并且尽可能地保留信息量，即让数据的总方差尽量靠近2。于是，我们将原本的直角坐标系逆时针旋转45°，形成了新的特征向量 x_1^* 和 x_2^* 组成的新平面，在这个新平面中，三个样本数据的坐标点可以表示为 $(\sqrt{2}, 0)$, $(2\sqrt{2}, 0)$, $(3\sqrt{2}, 0)$ 。可以注意到， x_2^* 上的数值此时都变成了0，因此 x_2^* 明显不带有任何有效信息了（此时 x_2^* 的方差也为0了）。此时， x_1^* 特征上的数据均值是 $2\sqrt{2}$ ，而方差则可表示成：

$$x2^{\star}_var = \frac{(\sqrt{2}-2\sqrt{2})^2 + (2\sqrt{2}-2\sqrt{2})^2 + (3\sqrt{2}-2\sqrt{2})^2}{2} = 2$$

x_1^* 上的数据均值为0，方差也为0。

此时，我们根据信息含量的排序，取信息含量最大的一个特征，因为我们想要的是一维数据。所以我们可以将 x_2^* 删除，同时也删除图中的 x_2^* 特征向量，剩下的 x_1^* 就代表了曾经需要两个特征来代表的三个样本点。通过旋转原有特征向量组成的坐标轴来找到新特征向量和新坐标平面，我们将三个样本点的信息压缩到了一条直线上，实现了二维变一维，并且尽量保留原始数据的信息。一个成功的降维，就实现了。



不难注意到，在这个降维过程中，有几个重要的步骤：

过程	二维特征矩阵	n维特征矩阵
1	输入原数据，结构为 (3,2) 找出原本的2个特征对应的直角坐标系，本质是找出这2个特征构成的2维平面	输入原数据，结构为 (m,n) 找出原本的n个特征向量构成的n维空间V
2	决定降维后的特征数量：1	决定降维后的特征数量：k
3	旋转，找出一个新坐标系 本质是找出2个新的特征向量，以及它们构成的新2维平面 新特征向量让数据能够被压缩到少数特征上，并且总信息量不损失太多	通过某种变化，找出n个新的特征向量，以及它们构成的新n维空间V
4	找出数据点在新坐标系上，2个新坐标轴上的坐标	找出原始数据在新特征空间V中的n个新特征向量上对应的值，即“将数据映射到新空间中”
5	选取第1个方差最大的特征向量，删掉没有被选中的特征，成功将2维平面降为1维	选取前k个信息量最大的特征，删掉没有被选中的特征，成功将n维空间V降为k维

在步骤3当中，我们用来找出n个新特征向量，让数据能够被压缩到少数特征上并且总信息量不损失太多的技术就是**矩阵分解**。PCA和SVD是两种不同的降维算法，但他们遵从上面的过程来实现降维，只是两种算法中矩阵分解的方法不同，信息量的衡量指标不同罢了。PCA使用方差作为信息量的衡量指标，并且特征值分解来找出空间V。降维时，它会通过一系列数学的神秘操作（比如说，产生协方差矩阵 $\frac{1}{n}XX^T$ ）将特征矩阵X分解为以下三个矩阵，其中Q和 Q^{-1} 是辅助的矩阵， Σ 是一个对角矩阵（即除了对角线上有值，其他位置都是0的矩阵），其对角线上的元素就是方差。降维完成之后，PCA找到的每个新特征向量就叫做“主成分”，而被丢弃的特征向量被认为信息量很少，这些信息很可能就是噪音。

$$X \rightarrow \text{数学神秘的宇宙} \rightarrow Q\Sigma Q^{-1}$$

而SVD使用奇异值分解来找出空间V，其中Σ也是一个对角矩阵，不过它对角线上的元素是奇异值，这也是SVD中用来衡量特征上的信息量的指标。U和V^T分别是左奇异矩阵和右奇异矩阵，也都是辅助矩阵。

$$X \rightarrow \text{另一个数学神秘的宇宙} \rightarrow U\Sigma V^T$$

在数学原理中，无论是PCA和SVD都需要遍历所有的特征和样本来计算信息量指标。并且在矩阵分解的过程之中，会产生比原来的特征矩阵更大的矩阵，比如原数据的结构是(m,n)，在矩阵分解中为了找出最佳新特征空间V，可能需要产生(n,n), (m,m)大小的矩阵，还需要产生协方差矩阵去计算更多的信息。而现在无论是Python还是R，或者其他任何语言，在大型矩阵运算上都不是特别擅长，无论代码如何简化，我们不可避免地要等待计算机去完成这个非常庞大的数学计算过程。因此，降维算法的计算量很大，运行比较缓慢，但无论如何，它们的功能无可替代，它们依然是机器学习领域的宠儿。

思考：PCA和特征选择技术都是特征工程的一部分，它们有什么不同？

特征工程中有三种方式：特征提取，特征创造和特征选择。仔细观察上面的降维例子和上周我们讲解过的特征选择，你发现有什么不同了吗？

特征选择是从已存在的特征中选取携带信息最多的，选完之后的特征依然具有可解释性，我们依然知道这个特征在原数据的哪个位置，代表着原数据上的什么含义。

而PCA，是将已存在的特征进行压缩，降维完毕后的特征不是原本的特征矩阵中的任何一个特征，而是通过某些方式组合起来的新特征。通常来说，**在新的特征矩阵生成之前，我们无法知晓PCA都建立了怎样的新特征向量，新特征矩阵生成之后也不具有可读性**，我们无法判断新特征矩阵的特征是从原数据中的什么特征组合而来，新特征虽然带有原始数据的信息，却已经不是原数据上代表着的含义了。以PCA为代表的降维算法因此是特征创造 (feature creation, 或feature construction) 的一种。

可以想见，PCA一般不适用于探索特征和标签之间的关系的模型（如线性回归），因为无法解释的新特征和标签之间的关系不具有意义。在线性回归模型中，我们使用特征选择。

2.2 重要参数n_components

n_components是我们降维后需要的维度，即降维后需要保留的特征数量，降维流程中第二步里需要确认的k值，一般输入[0, min(X.shape)]范围中的整数。一说到K，大家可能都会想到，类似于KNN中的K和随机森林中的n_estimators，这是一个需要我们人为去确认的超参数，并且我们设定的数字会影响到模型的表现。如果留下的特征太多，就达不到降维的效果，如果留下的特征太少，那新特征向量可能无法容纳原始数据集中的大部分信息，因此，n_components既不能太大也不能太小。那怎么办呢？

可以先从我们的降维目标说起：如果我们希望可视化一组数据来观察数据分布，我们往往将数据降到三维以下，很多时候是二维，即n_components的取值为2。

2.2.1 迷你案例：高维数据的可视化

1. 调用库和模块

```
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
```

2. 提取数据集

```
iris = load_iris()
y = iris.target
X = iris.data
#作为数组，X是几维？
X.shape
#作为数据表或特征矩阵，X是几维？
import pandas as pd
pd.DataFrame(X)
```

3. 建模

```
#调用PCA
pca = PCA(n_components=2) #实例化
pca = pca.fit(X) #拟合模型
X_dr = pca.transform(X) #获取新矩阵

X_dr
#也可以fit_transform一步到位
#X_dr = PCA(2).fit_transform(X)
```

4. 可视化

#要将三种鸢尾花的数据分布显示在二维平面坐标系中，对应的两个坐标（两个特征向量）应该是三种鸢尾花降维后的x1和x2，怎样才能取出三种鸢尾花下不同的x1和x2呢？

X_dr[y == 0, 0] #这里是布尔索引，看出来了么？

#要展示三中分类的分布，需要对三种鸢尾花分别绘图
#可以写成三行代码，也可以写成for循环

```

"""
plt.figure()
plt.scatter(X_dr[y==0, 0], X_dr[y==0, 1], c="red", label=iris.target_names[0])
plt.scatter(X_dr[y==1, 0], X_dr[y==1, 1], c="black", label=iris.target_names[1])
plt.scatter(X_dr[y==2, 0], X_dr[y==2, 1], c="orange", label=iris.target_names[2])
plt.legend()
plt.title('PCA of IRIS dataset')
plt.show()
"""

colors = ['red', 'black', 'orange']
iris.target_names

plt.figure()
for i in [0, 1, 2]:
    plt.scatter(X_dr[y == i, 0]
                ,X_dr[y == i, 1]
                ,alpha=.7
                ,c=colors[i]
                ,label=iris.target_names[i]
                )
plt.legend()
plt.title('PCA of IRIS dataset')
plt.show()

```

鸢尾花的分布被展现在我们眼前了，明显这是一个分簇的分布，并且每个簇之间的分布相对比较明显，也许versicolor和virginica这两种花之间会有一些分类错误，但setosa肯定不会被分错。这样的数据很容易分类，可以遇见，KNN，随机森林，神经网络，朴素贝叶斯，Adaboost这些分类器在鸢尾花数据集上，未调整的时候都可以有95%上下的准确率。

6. 探索降维后的数据

```

#属性explained_variance_，查看降维后每个新特征向量上所带的信息量大小（可解释性方差的大小）
pca.explained_variance_

#属性explained_variance_ratio_，查看降维后每个新特征向量所占的信息量占原始数据总信息量的百分比
#又叫做可解释方差贡献率
pca.explained_variance_ratio_
#大部分信息都被有效地集中在了第一个特征上

pca.explained_variance_ratio_.sum()

```

7. 选择最好的n_components：累积可解释方差贡献率曲线

当参数n_components中不填写任何值，则默认返回min(X.shape)个特征，一般来说，样本量都会大于特征数目，所以什么都不填就相当于转换了新特征空间，但没有减少特征的个数。一般来说，不会使用这种输入方式。但我们可以使用这种输入方式来画出累计可解释方差贡献率曲线，以此选择最好的n_components的整数取值。

累积可解释方差贡献率曲线是一条以降维后保留的特征个数为横坐标，降维后新特征矩阵捕捉到的可解释方差贡献率为纵坐标的曲线，能够帮助我们决定n_components最好的取值。

```

import numpy as np
pca_line = PCA().fit(x)
plt.plot([1,2,3,4],np.cumsum(pca_line.explained_variance_ratio_))
plt.xticks([1,2,3,4]) #这是为了限制坐标轴显示为整数
plt.xlabel("number of components after dimension reduction")
plt.ylabel("cumulative explained variance ratio")
plt.show()

```

2.2.2 最大似然估计自选超参数

除了输入整数，n_components还有哪些选择呢？之前我们提到过，矩阵分解的理论发展在业界独树一帜，勤奋智慧的数学大神Minka, T.P.在麻省理工学院媒体实验室做研究时找出了让PCA用最大似然估计(maximum likelihood estimation)自选超参数的方法，输入“mle”作为n_components的参数输入，就可以调用这种方法。

```

pca_mle = PCA(n_components="mle")
pca_mle = pca_mle.fit(x)
x_mle = pca_mle.transform(x)

x_mle
#可以发现，mle为我们自动选择了3个特征

pca_mle.explained_variance_ratio_.sum()
#得到了比设定2个特征时更高的信息含量，对于鸢尾花这个很小的数据集来说，3个特征对应这么高的信息含量，并不需要去纠结于只保留2个特征，毕竟三个特征也可以可视化

```

2.2.3 按信息量占比选超参数

输入[0,1]之间的浮点数，并且让参数svd_solver =='full'，表示希望降维后的总解释性方差占比大于n_components指定的百分比，即是说，希望保留百分之多少的信息量。比如说，如果我们希望保留97%的信息量，就可以输入n_components = 0.97，PCA会自动选出能够让保留的信息量超过97%的特征数量。

```

pca_f = PCA(n_components=0.97,svd_solver="full")
pca_f = pca_f.fit(x)
x_f = pca_f.transform(x)

pca_f.explained_variance_ratio_

```

2.3 PCA中的SVD

2.3.1 PCA中的SVD哪里来?

细心的小伙伴可能注意到了，`svd_solver`是奇异值分解器的意思，为什么PCA算法下面会有有关奇异值分解的参数？不是两种算法么？我们之前曾经提到过，PCA和SVD涉及了大量的矩阵计算，两者都是运算量很大的模型，但其实，SVD有一种惊人的数学性质，即是**它可以跳过数学神秘的宇宙，不计算协方差矩阵，直接找出一个新特征向量组成的新维空间**，而这个n维空间就是奇异值分解后的右矩阵 V^T （所以一开始在讲解降维过程时，我们说“生成新特征向量组成的空间V”，并非巧合，而是特指奇异值分解中的矩阵 V^T ）。

传统印象中的SVD: $X \rightarrow$ 数学神秘的宇宙 $\rightarrow U\Sigma V^T$

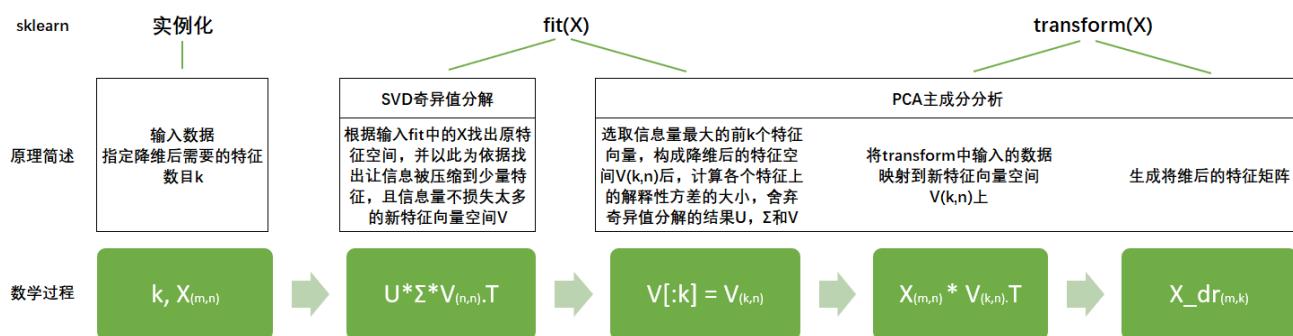
其实会开挂的SVD: $X \rightarrow$ 一个比起PCA简化非常多的数学过程 $\rightarrow V^T$

右奇异矩阵 V^T 有着如下性质：

$$X_{dr} = X * V[:k]^T$$

k就是`n_components`，是我们降维后希望得到的维度。若 X 为 (m,n) 的特征矩阵， V^T 就是结构为 (n,n) 的矩阵，取这个矩阵的前k行（进行切片），即将 V 转换为结构为 (k,n) 的矩阵。而 $V_{(k,n)}^T$ 与原特征矩阵 X 相乘，即可得到降维后的特征矩阵 X_{dr} 。这是说，**奇异值分解可以不计算协方差矩阵等等结构复杂计算冗长的矩阵，就直接求出新特征空间和降维后的特征矩阵**。

简而言之，SVD在矩阵分解中的过程比PCA简单快速，虽然两个算法都走一样的分解流程，但SVD可以作弊耍赖直接算出 V 。但是遗憾的是，SVD的信息量衡量指标比较复杂，要理解“奇异值”远不如理解“方差”来得容易，因此，sklearn将降维流程拆成了两部分：一部分是计算特征空间 V ，由奇异值分解完成，另一部分是映射数据和求解新特征矩阵，由主成分分析完成，实现了用SVD的性质减少计算量，却让信息量的评估指标是方差，具体流程如下图：



讲到这里，相信大家就能够理解，为什么PCA的类里会包含控制SVD分解器的参数了。通过SVD和PCA的合作，sklearn实现了一种计算更快更简单的“合作降维”。很多人理解SVD，是把SVD当作PCA的一种求解方法，其实指的就是在矩阵分解时不使用PCA本身的特征值分解，而使用奇异值分解来减少计算量。这种方法确实存在，但在sklearn中，矩阵 U 和 Σ 虽然会被计算出来（同样也是一种比起PCA来说简化非常多的数学过程，不产生协方差矩阵），但完全不会被用到，也无法调取查看或者使用，因此我们可以认为， U 和 Σ 在`fit`过后就被遗弃了。奇异值分解追求的仅仅是 V ，只要有了 V ，就可以计算出降维后的特征矩阵。在`transform`过程之后，`fit`中奇异值分解的结果除了 $V(k,n)$ 以外，就会被舍弃，而 $V(k,n)$ 会被保存在属性`components_`当中，可以调用查看。

```
PCA(2).fit(X).components_
```

```
PCA(2).fit(X).components_.shape
```

2.3.2 重要参数svd_solver 与 random_state

参数svd_solver是在降维过程中，用来控制矩阵分解的一些细节的参数。有四种模式可选："auto", "full", "arpack", "randomized"，默认"auto"。

- "**auto**"：基于X.shape和n_components的默认策略来选择分解器：如果输入数据的尺寸大于500x500且要提取的特征数小于数据最小维度min(X.shape)的80%，就启用效率更高的"randomized"方法。否则，精确完整的SVD将被计算，截断将会在矩阵被分解完成后有选择地发生
- "**full**"：从scipy.linalg.svd中调用标准的LAPACK分解器来生成精确完整的SVD，**适合数据量比较适中，计算时间充足的情况**，生成的精确完整的SVD的结构为：

$$U_{(m,m)}, \Sigma_{(m,n)}, V_{(n,n)}^T$$

- "**arpack**"：从scipy.sparse.linalg.svds调用ARPACK分解器来运行截断奇异值分解(SVD truncated)，分解时就将特征数量降到n_components中输入的数值k，**可以加快运算速度，适合特征矩阵很大的时候，但一般用于特征矩阵为稀疏矩阵的情况**，此过程包含一定的随机性。截断后的SVD分解出的结构为：

$$U_{(m,k)}, \Sigma_{(k,k)}, V_{(n,n)}^T$$

- "**randomized**"，通过Halko等人的随机方法进行随机SVD。在"full"方法中，分解器会根据原始数据和输入的n_components值去计算和寻找符合需求的新特征向量，但是在"randomized"方法中，分解器会先生成多个随机向量，然后一一去检测这些随机向量中是否有任何一个符合我们的分解需求，如果符合，就保留这个随机向量，并基于这个随机向量来构建后续的向量空间。这个方法已经被Halko等人证明，比"full"模式下计算快很多，并且还能够保证模型运行效果。**适合特征矩阵巨大，计算量庞大的情况**。

而参数random_state在参数svd_solver的值为"arpack" or "randomized"的时候生效，可以控制这两种SVD模式中的随机模式。通常我们就选用"auto"，不必对这个参数纠结太多。

2.3.3 重要属性components_

现在我们了解了，V(k,n)是新特征空间，是我们要将原始数据进行映射的那些新特征向量组成的矩阵。我们用它来计算新的特征矩阵，但我们希望获取的毕竟是X_dr，为什么我们要把V(k,n)这个矩阵保存在n_components这个属性当中来让大家调取查看呢？

我们之前谈到过PCA与特征选择的区别，即特征选择后的特征矩阵是可解读的，而PCA降维后的特征矩阵式不可解读的：PCA是将已存在的特征进行压缩，降维完毕后的特征不是原本的特征矩阵中的任何一个特征，而是通过某些方式组合起来的新特征。通常来说，在新的特征矩阵生成之前，我们无法知晓PCA都建立了怎样的新特征向量，新特征矩阵生成之后也不具有可读性，我们无法判断新特征矩阵的特征是从原数据中的什么特征组合而来，新特征虽然带有原始数据的信息，却已经不是原数据上代表着的含义了。

但是其实，在矩阵分解时，PCA是有目标的：在原有特征的基础上，找出能够让信息尽量聚集的新特征向量。在sklearn使用的PCA和SVD联合的降维方法中，这些新特征向量组成的新特征空间其实就是V(k,n)。当V(k,n)是数字时，我们无法判断V(k,n)和原有的特征究竟有着怎样千丝万缕的数学联系。但是，如果原特征矩阵是图像，V(k,n)这个空间矩阵也可以被可视化的话，我们就可以通过两张图来比较，就可以看出新特征空间究竟从原始数据里提取了什么重要的信息。

让我们来看一个，人脸识别中属性components_的运用。

1. 导入需要的库和模块

```
from sklearn.datasets import fetch_lfw_people
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import numpy as np
```

2. 实例化数据集，探索数据

```
faces = fetch_lfw_people(min_faces_per_person=60)
faces.images.shape
#怎样理解这个数据的维度？
faces.data.shape
#换成特征矩阵之后，这个矩阵是什么样？
X = faces.data
```

3. 看看图像什么样？将原特征矩阵进行可视化

```
#数据本身是图像，和数据本身只是数字，使用的可视化方法不同

#创建画布和子图对象
fig, axes = plt.subplots(4, 5
                        , figsize=(8, 4)
                        , subplot_kw = {"xticks":[], "yticks":[]} #不显示坐标轴
)
fig
axes

#不难发现，axes中的一个对象对应fig中的一个空格
#我们希望，在每一个子图对象中填充图像（共24张图），因此我们需要写一个在子图对象中遍历的循环

axes.shape

#二维结构，可以有两种循环方式，一种是使用索引，循环一次同时生成一列上的三个图
#另一种是把数据拉成一维，循环一次只生成一个图
#在这里，究竟使用哪一种循环方式，是要看我们要画的图的信息，储存在一个怎样的结构里
#我们使用 子图对象.imshow 来将图像填充到空白画布上
#而imshow要求的数据格式必须是一个(m, n)格式的矩阵，即每个数据都是一张单独的图
#因此我们需要遍历的是faces.images，其结构是(1277, 62, 47)
#要从一个数据集中取出24个图，明显是一次性的循环切片[i,:,:]来得便利
#因此我们要把axes的结构拉成一维来循环

axes.flat

enumerate(axes.flat)

#填充图像
for i, ax in enumerate(axes.flat):
    ax.imshow(faces.images[i,:,:]
              , cmap="gray" #选择色彩的模式
```

)

```
#https://matplotlib.org/tutorials/colors/colormaps.html
```

4. 建模降维，提取新特征空间矩阵

```
#原本有2900维，我们现在来降到150维
pca = PCA(150).fit(X)

V = pca.components_
V.shape
```

5. 将新特征空间矩阵可视化

```
fig, axes = plt.subplots(3,8, figsize=(8,4), subplot_kw = {"xticks":[],"yticks":[]})

for i, ax in enumerate(axes.flat):
    ax.imshow(V[i,:].reshape(62,47), cmap="gray")
```

这张图稍稍有一些恐怖，但可以看出，比起降维前的数据，新特征空间可视化后的人脸非常模糊，这是因为原始数据还没有被映射到特征空间中。但是可以看出，整体比较亮的图片，获取的信息较多，整体比较暗的图片，却只能看见黑漆漆的一块。在比较亮的图片中，眼睛，鼻子，嘴巴，都相对清晰，脸的轮廓，头发之类的比较模糊。

这说明，新特征空间里的特征向量们，大部分是“五官”和“亮度”相关的向量，所以新特征向量上的信息肯定大部分是由原数据中和“五官”和“亮度”相关的特征中提取出来的。到这里，我们通过可视化新特征空间V，解释了一部分降维后的特征：虽然显示出来的数字看着不知所云，但画出来的图表示，这些特征是和“五官”以及“亮度”有关的。这也再次证明了，PCA能够将原始数据集中重要的数据进行聚集。

2.4 重要接口inverse_transform

在上周的特征工程课中，我们学到了神奇的接口inverse_transform，可以将我们归一化，标准化，甚至做过哑变量的特征矩阵还原回原始数据中的特征矩阵，这几乎在向我们暗示，任何有inverse_transform这个接口的过程都是可逆的。PCA应该也是如此。在sklearn中，我们通过让原特征矩阵 X 右乘新特征空间矩阵 $V(k,n)$ 来生成新特征矩阵 X_{dr} ，那理论上来说，让新特征矩阵 X_{dr} 右乘 $V(k,n)$ 的逆矩阵 $V_{(k,n)}^{-1}$ ，就可以将新特征矩阵 X_{dr} 还原为 X 。那sklearn是否这样做了呢？让我们来看看下面的案例。

2.4.1 迷你案例：用人脸识别看PCA降维后的信息保存量

人脸识别是最容易的，用来探索inverse_transform功能的数据。我们先调用一组人脸数据 $X(m,n)$ ，对人脸图像进行绘制，然后我们对人脸数据进行降维得到 X_{dr} ，之后再使用inverse_transform(X_{dr})返回一个 $X_{inverse}(m,n)$ ，并对这个新矩阵中的人脸图像也进行绘制。如果PCA的降维过程是可逆的，我们应当期待 $X(m,n)$ 和 $X_{inverse}(m,n)$ 返回一模一样的图像，即携带一模一样的信息。

1. 导入需要的库和模块(与2.3.3节中步骤一致)

```
from sklearn.datasets import fetch_lfw_people
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import numpy as np
```

2. 导入数据，探索数据(与2.3.3节中步骤一致)

```
faces = fetch_lfw_people(min_faces_per_person=60)
faces.images.shape
#怎样理解这个数据的维度？
faces.data.shape
#换成特征矩阵之后，这个矩阵是什么样？
X = faces.data
```

3. 建模降维，获取降维后的特征矩阵 X_{dr}

```
pca = PCA(150)
X_dr = pca.fit_transform(X)
X_dr.shape
```

4. 将降维后矩阵用inverse_transform返回原空间

```
X_inverse = pca.inverse_transform(X_dr)
X_inverse.shape
```

5. 将特征矩阵X和X_inverse可视化

```

fig, ax = plt.subplots(2,10, figsize=(10,2.5)
                      , subplot_kw={"xticks":[], "yticks":[]})
#和2.3.3节中的案例一样，我们需要对子图对象进行遍历的循环，来将图像填入子图中
#那在这里，我们使用怎样的循环？
#现在我们的ax中是2行10列，第一行是原数据，第二行是inverse_transform后返回的数据
#所以我们需要同时循环两份数据，即一次循环画一列上的两张图，而不是把ax拉平

for i in range(10):
    ax[0,i].imshow(face.image[i,:,:], cmap="binary_r")
    ax[1,i].imshow(x_inverse[i].reshape(62,47), cmap="binary_r")

```

可以明显看出，这两组数据可视化后，由降维后再通过inverse_transform转换回原维度的数据画出的图像和原数据画的图像大致相似，但原数据的图像明显更加清晰。这说明，inverse_transform并没有实现数据的完全逆转。这是因为，在降维的时候，部分信息已经被舍弃了，X_dr中往往不会包含原数据100%的信息，所以在逆转的时候，即便维度升高，原数据中已经被舍弃的信息也不可能再回来了。所以，**降维不是完全可逆的**。

Inverse_transform的功能，是基于X_dr中的数据进行升维，将数据重新映射到原数据所在的特征空间中，而并非恢复所有原有的数据。但同时，我们也可以看出，降维到300以后的数据，的确保留了原数据的大部分信息，所以图像看起来，才会和原数据高度相似，只是稍稍模糊罢了。

2.4.2 迷你案例：用PCA做噪音过滤

降维的目的之一就是希望抛弃掉对模型带来负面影响的特征，而我们相信，带有效信息的特征的方差应该是远大于噪音的，所以相比噪音，有效的特征所带的信息应该不会在PCA过程中被大量抛弃。inverse_transform能够在不恢复原始数据的情况下，将降维后的数据返回到原本的高维空间，即是说能够实现“保证维度，但去掉方差很小特征所带的信息”。利用inverse_transform的这个性质，我们能够实现噪音过滤。

1. 导入所需要的库和模块

```

from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import numpy as np

```

2. 导入数据，探索数据

```

digits = load_digits()
digits.data.shape

```

3. 定义画图函数

```
def plot_digits(data):
    fig, axes = plt.subplots(4,10,figsize=(10,4)
                           ,subplot_kw = {"xticks":[],"yticks":[]})
    for i, ax in enumerate(axes.flat):
        ax.imshow(data[i].reshape(8,8),cmap="binary")

plot_digits(digits.data)
```

4. 为数据加上噪音

```
np.random.RandomState(42)

#在指定的数据集中，随机抽取服从正态分布的数据
#两个参数，分别是指定的数据集，和抽取出来的正太分布的方差
noisy = np.random.normal(digits.data,2)

plot_digits(noisy)
```

5. 降维

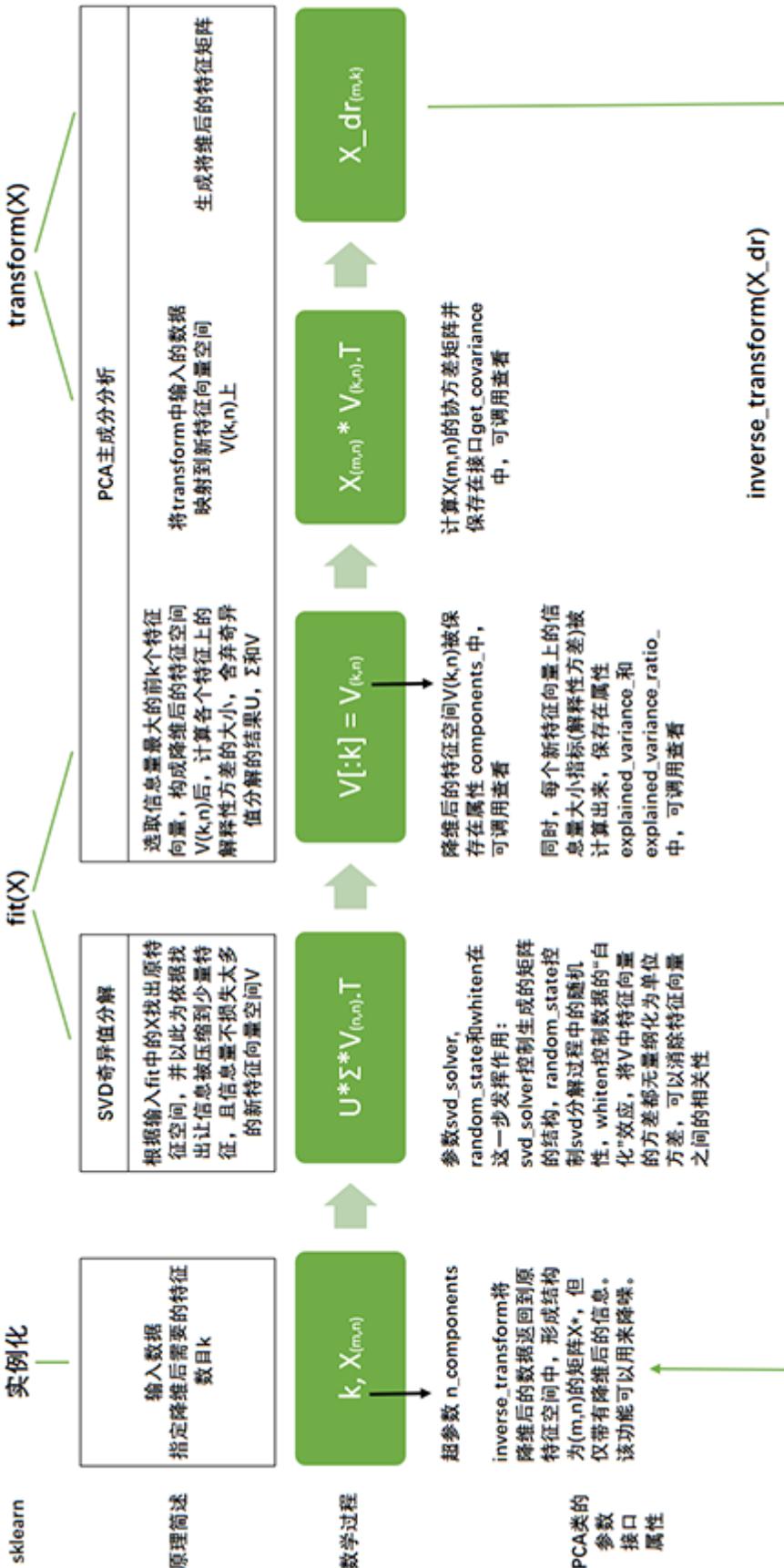
```
pca = PCA(0.5).fit(noisy)
X_dr = pca.transform(noisy)
X_dr.shape
```

6. 逆转降维结果，实现降噪

```
without_noise = pca.inverse_transform(X_dr)
plot_digits(without_noise)
```

2.5 重要接口，参数和属性总结

到现在，我们已经完成了对PCA的讲解。我们讲解了重要参数参数n_components, svd_solver, random_state, 讲解了三个重要属性: components_, explained_variance_以及explained_variance_ratio_，无数次用到了接口fit, transform, fit_transform, 还讲解了与众不同的重要接口inverse_transform。所有的这些内容都可以被总结在这张图中：



3 案例：PCA对手写数字数据集的降维

还记得我们上一周在讲特征工程时，使用的手写数字的数据集吗？数据集结构为(42000, 784)，用KNN跑一次半小时，得到准确率在96.6%上下，用随机森林跑一次12秒，准确率在93.8%，虽然KNN效果好，但由于数据量太大，KNN计算太缓慢，所以我们不得不选用随机森林。我们使用了各种技术对手写数据集进行特征选择，最后使用嵌入法SelectFromModel选出了324个特征，将随机森林的效果也调到了96%以上。但是，因为数据量依然巨大，还是有300多个特征。今天，我们就来试着用PCA处理一下这个数据，看看效果如何。

1. 导入需要的模块和库

```
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier as RFC
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，探索数据

```
data = pd.read_csv(r"C:\work\Learnbetter\micro-class\week 3 Preprocessing\digit
recognizer.csv")

X = data.iloc[:,1:]
y = data.iloc[:,0]

X.shape
```

3. 画累计方差贡献率曲线，找最佳降维后维度的范围

```
pca_line = PCA().fit(X)
plt.figure(figsize=[20,5])
plt.plot(np.cumsum(pca_line.explained_variance_ratio_))
plt.xlabel("number of components after dimension reduction")
plt.ylabel("cumulative explained variance ratio")
plt.show()
```

4. 降维后维度的学习曲线，继续缩小最佳维度的范围

```
=====【TIME WARNING: 2mins 30s】=====

score = []
for i in range(1,101,10):
    X_dr = PCA(i).fit_transform(X)
    once = cross_val_score(RFC(n_estimators=10,random_state=0),
                           X_dr,y, cv=5).mean()
    score.append(once)
plt.figure(figsize=[20,5])
plt.plot(range(1,101,10),score)
plt.show()
```

5. 细化学习曲线，找出降维后的最佳维度

```
#=====【TIME WARNING: 2mins 30s】=====#
score = []
for i in range(10,25):
    X_dr = PCA(i).fit_transform(X)
    once = cross_val_score(RFC(n_estimators=10,random_state=0),X_dr,y,cv=5).mean()
    score.append(once)
plt.figure(figsize=[20,5])
plt.plot(range(10,25),score)
plt.show()
```

6. 导入找出的最佳维度进行降维，查看模型效果

```
X_dr = PCA(23).fit_transform(X)

#=====【TIME WARNING:1mins 30s】=====#
cross_val_score(RFC(n_estimators=100,random_state=0),X_dr,y,cv=5).mean()
```

模型效果还好，跑出了94.49%的水平，但还是没有我们使用嵌入法特征选择过后的96%高，有没有什么办法能够提高模型的表现呢？

7. 突发奇想，特征数量已经不足原来的3%，换模型怎么样？

在之前的建模过程中，因为计算量太大，所以我们一直使用随机森林，但事实上，我们知道KNN的效果比随机森林更好，KNN在未调参的状况下已经达到96%的准确率，而随机森林在未调参前只能达到93%，这是模型本身的限制带来的，这个数据使用KNN效果就是会更好。现在我们的特征数量已经降到不足原来的3%，可以使用KNN了吗？

```
from sklearn.neighbors import KNeighborsClassifier as KNN
cross_val_score(KNN(),X_dr,y,cv=5).mean()
```

8. KNN的k值学习曲线

```
#=====【TIME WARNING: 】=====#
score = []
for i in range(10):
    X_dr = PCA(23).fit_transform(X)
    once = cross_val_score(KNN(i+1),X_dr,y,cv=5).mean()
    score.append(once)
plt.figure(figsize=[20,5])
plt.plot(range(10),score)
plt.show()
```

9. 定下超参数后，模型效果如何，模型运行时间如何？

```
cross_val_score(KNN(4), X_dr, y, cv=5).mean()
```

```
=====【TIME WARNING: 3mins】=====
```

```
%%timeit
```

```
cross_val_score(KNN(4), X_dr, y, cv=5).mean()
```

可以发现，原本785列的特征被我们缩减到23列之后，用KNN跑出了目前位置这个数据集上最好的结果。再进行更细致的调整，我们也许可以将KNN的效果调整到98%以上。PCA为我们提供了无限的可能，终于不用再因为数据量太庞大而被迫选择更加复杂的模型了！

4 附录

4.1 PCA参数列表

n_components	整数, 浮点数, None或输入字符串 要保留的特征数量。若不填写, 则保留的特征数量为数据最小维度min(X.shape)。 - 输入 "mle"并且参数svd_solver == "full", 表示使用Minka, T.P. 的自动选择PCA维度法来猜测维度, 此论文在NIPS上, 第598-604页。注意, 输入 "mle"参数后svd_solver == 'auto'也会被理解为svd_solver == 'full'。 - 输入 [0,1]之间的浮点数且svd_solver == 'full', 表示选择让"需要解释的总方差量"大于n_components指定的数目的那些特征 - 输入 "None", 则保留的特征数量为数据最小维度-1, 即min(X.shape) - 1 如果svd_solver == 'arpack', 则特征数必须严格小于数据最小维度min(X.shape)
copy	布尔值, 可不填, 默认True 如果为False, 则传递给fit的数据将被覆盖并且运行fit(X).transform(X)将不会产生预期结果, 请改用fit_transform(X)
whiten	布尔值, 可不填, 默认False 控制输入PCA的特征矩阵的白化。白化是数据预处理的一种, 其目的是去掉特征与特征之间的相关性, 并将所有特征的方差都归一化。当为True时, components_(即奇异值分解中分解出来的矩阵v)中的向量会被乘以样本数量的平方根sqrt(n_samples), 然后除以奇异值(奇异矩阵Σ的对角线上的元素Σi), 以确保特征向量之间不相关并且每个特征都具有单位方差。 白化将从变换后的信号中去除一些信息 (比如特征之间的相对方差量纲), 但有时可以通过使数据遵循一些硬连线假设来提高下游估计器的预测精度。
svd_solver	输入字符串, 可选"auto", "full", "arpack", "randomized" 控制SVD奇异值分解的分解模式, 默认 - 输入 "auto": 基于X.shape和n_components的默认策略来选择分解器: 如果输入数据的尺寸大于500x500且要提取的特征数小于数据最小维度min(X.shape)的80%, 就启用效率更高的'随机化'方法。否则, 精确完整的SVD将被计算, 截断将会在矩阵被分解完成后有选择地发生。 - 输入 "full": 运行精确完整的SVD, 从scipy.linalg.svd中调用标准的LAPACK分解器, 并通过后处理来选择特征。完整的SVD的结构为: U(m,m), S(m,n), V(n,n)。 - 输入 "arpack": 从scipy.sparse.linalg.svds调用ARPACK分解器来运行截断奇异值分解(SVD truncated), 以在分解时就将特征数量降到n_components中输入的数值, 严格要求n_components的取值在区间[0,数据最小维度min(X.shape)]之间。截断后的SVD分解出的结构为: U(m,n_components), S(n_components, n_componenst), V(n,n)。 - 输入 "randomized", 通过Halko等人的随机方法进行随机SVD。在"full"方法中, 分解器会根据原始数据和输入的n_components值去计算和寻找符合需求的新特征向量, 但是在"randomized"方法中, 分解器会先生成多个随机向量, 然后一一去检测这些随机向量中是否有任何一个符合我们的分解需求, 如果符合, 就保留这个随机向量, 并基于这个随机向量来构建后续的向量空间。这个方法已经被Halko等人证明, 比"full"模式下计算快很多, 并且还能够保证模型运行效果。 具体请参考Halko, N., Martinsson, P.G.以及Tropp, J.A.于2011年发表的论文《寻找具有随机性的结构: 用于构造近似矩阵分解的概率算法》 (<i>Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions</i>) , 来自学术期刊SIAM review, 53期第2册, 217-288页, 请同时参考Martinsson, P. G., Rokhlin, V., and Tygert, M. 于2011年发表的《用于矩阵分解的随机算法》 (<i>A randomized algorithm for the decomposition of matrices</i>) , 来自学术刊物Applied and Computational Harmonic Analysis, 30期第1册, 47-68页。

tol	大于等于0的浮点数，默认为0 当svd_solver为"arpack"的时候，计算奇异值所需要的容差。 <i>0.18.0版本中新增的功能。</i>
iterated_power	输入大于0的整数，或者"auto"，默认"auto" 当svd_solver的时候'randomized'，计算的幂方法的迭代次数。 <i>0.18.0版本中新增的功能。</i>
random_state	整数，sklearn中设定好的RandomState实例，或None，可不填，默认None 仅当参数svd_solver的值为"arpack" or "randomized"的时候才有效。 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 0.18.0版本中新增的功能。

4.2 PCA属性列表

components_	数组，结构为(n_components, n_features) 新特征空间V中的特征向量的方向，按explained_variance_的大小排序
explained_variance_	数组，结构为(n_components,) 每个所选特征的解释性方差 等于X的协方差矩阵的前n_components个最大特征值 <i>0.18.0版本中新增的功能</i>
explained_variance_ratio_	数组，结构为(n_components,) 每个所选特征的解释性方差占原数据方差综合的百分比 如果没有输入任何n_components，则保留所有的特征，并且比率综合为1
singular_values_	数组，结构为(n_components,) 返回每个所选特征的奇异值。奇异值等于低维空间中n_components变量的2范式
mean_	数组，结构为(n_features,) 从训练集估计出的每个特征的均值，数值上等于 X.mean(axis=0)
n_components_	整数 估计的特征数量。当参数n_components被设置为mle或0~1之间的数字时(并且svd_solver被设置为"full")，将根据输入的特征矩阵估算降维后的特征个数。否则，这个属性会等于输入参数n_components中的值。如果参数n_components被设置为None，这个属性会返回min(X.shape)，即数据的最小维度。
noise_variance_	浮点数 根据1999年Tipping和Bishop的Probabilistic PCA模型估计的噪声协方差。参见C.Bishop论文《模式识别和机器学习》，章节12.2.1 p.574或 http://www.miketipping.com/papers/met-mppca.pdf 。需要计算估计的数据协方差和分数样本。 等于特征矩阵的协方差矩阵的 (min(X.shape) - n_components) 的最小特征值的平均值。

4.3 PCA接口列表

接口	输入	含义	返回
fit	特征矩阵X	使用特征矩阵拟合模型	拟合好的模型本身
transform	特征矩阵X	将降维应用到特征矩阵	降维后的特征矩阵
fit_transform	特征矩阵X	使用特征矩阵拟合模型并且在特征矩阵上应用降维	降维后的特征矩阵
inverse_transform	特征矩阵X	将数据转换回原始空间	返回到原始空间的矩阵，该矩阵与原始特征矩阵结构相同，但数据不同
score	特征矩阵X	返回所有样本的平均对数似然	参见C.Bishop论文《模式识别和机器学习》，章节12.2.1 p.574或 http://www.miketipping.com/papers/met-mppca.pdf 。
score_samples	不需要输入任何对象	返回每个样本的对数似然	
get_covariance	不需要输入任何对象	在生成的模型上计算数据的协方差矩阵	数据的协方差 协方差的公式为： $cov = components_.T * S^2 * components_ + sigma2 * eye(n_features)$ 其中S^2包含解释的方差，sigma2包含噪声方差。
get_precision	不需要输入任何对象	在生成的模型上计算数据的精度矩阵	返回估计数据的精确度。 等于协方差的倒数，但用矩阵求逆引理来提升计算效率。
get_params	不需要输入任何对象	获取此评估器的参数	模型的参数
set_params	新参数组合	在建立好的模型上，重新设置此评估器的参数	用新参数组合重新实例化和训练的模型

菜菜的scikit-learn课堂05



sklearn中的逻辑回归

小伙伴们晚上好~o(—▽—)ブ

我是菜菜，这里是我的sklearn课堂第五期，今晚的直播内容是逻辑回归~

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scikit-learn课堂05

sklearn中的逻辑回归

1 概述

1.1 名为“回归”的分类器

1.2 为什么需要逻辑回归

1.3 sklearn中的逻辑回归

2 linear_model.LogisticRegression

2.1 二元逻辑回归的损失函数

2.1.1 损失函数的概念与解惑

2.1.2 【选学】二元逻辑回归损失函数的数学解释，公式推导与解惑

2.2 重要参数penalty & C

2.2.1 正则化

2.2.2 逻辑回归中的特征工程

2.3 梯度下降：重要参数max_iter

2.3.1 梯度下降求解逻辑回归

2.3.2 梯度下降的概念与解惑

2.3.3 步长的概念与解惑

2.4 二元回归与多元回归：重要参数solver & multi_class

2.5 样本不平衡与参数class_weight

3 案例：用逻辑回归制作评分卡

3.1 导库，获取数据

3.2 探索数据与数据预处理

3.2.1 去除重复值

3.2.2 填补缺失值

3.2.3 描述性统计处理异常值

3.2.4 为什么不统一量纲，也不标准化数据分布？

3.2.5 样本不均衡问题

3.2.6 分训练集和测试集

3.3 分箱

3.3.1 等频分箱

3.3.2 【选学】确保每个箱中都有0和1

3.3.3 定义WOE和IV函数

3.3.4 卡方检验，合并箱体，画出IV曲线

3.3.5 用最佳分箱个数分箱，并验证分箱结果

3.3.6 将选取最佳分箱个数的过程包装为函数

3.3.7 对所有特征进行分箱选择

3.4 计算各箱的WOE并映射到数据中

3.5 建模与模型验证

3.6 制作评分卡

4 附录：

4.1 逻辑回归的参数列表

4.2 逻辑回归的属性列表

4.3 逻辑回归的接口列表

1 概述

1.1 名为“回归”的分类器

在过去的四周中，我们接触了不少带“回归”二字的算法，回归树，随机森林的回归，无一例外他们都是区别于分类算法们，用来处理和预测连续型标签的算法。然而逻辑回归，是一种名为“回归”的线性分类器，其本质是由线性回归变化而来的，一种广泛使用于分类问题中的广义回归算法。要理解逻辑回归从何而来，得要先理解线性回归。线性回归是机器学习中最简单的的回归算法，它写作一个几乎人人熟悉的方程：

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

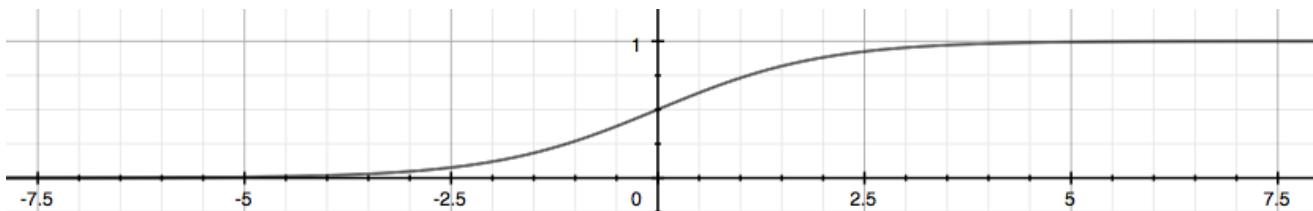
θ 被统称为模型的参数，其中 θ_0 被称为截距(intercept)， $\theta_1 \sim \theta_n$ 被称为系数(coefficient)，这个表达式，其实就和我们小学时就无比熟悉的 $y = ax + b$ 是同样的性质。我们可以使用矩阵来表示这个方程，其中x和 θ 都可以被看做是一个列矩阵，则有：

$$z = [\theta_0, \theta_1, \theta_2 \dots \theta_n] * \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \theta^T \mathbf{x} \quad (x_0 = 1)$$

线性回归的任务，就是构造一个预测函数 z 来映射输入的特征矩阵 x 和标签值 y 的线性关系，而**构造预测函数的核心就是找出模型的参数： θ^T 和 θ_0** ，著名的最小二乘法就是用来求解线性回归中参数的数学方法。

通过函数 z ，线性回归使用输入的特征矩阵 X 来输出一组连续型的标签值 y_{pred} ，以完成各种预测连续型变量的任务（比如预测产品销量，预测股价等等）。那如果我们的标签是离散型变量，尤其是，如果是满足0-1分布的离散型变量，我们要怎么办呢？我们可以通过引入联系函数(link function)，将线性回归方程 z 变换为 $g(z)$ ，并且令 $g(z)$ 的值分布在(0,1)之间，且当 $g(z)$ 接近0时样本的标签为类别0，当 $g(z)$ 接近1时样本的标签为类别1，这样就得到了一个分类模型。而这个联系函数对于逻辑回归来说，就是Sigmoid函数：

$$g(z) = \frac{1}{1 + e^{-z}}$$



面试高危问题：Sigmoid函数的公式和性质

Sigmoid函数是一个S型的函数，当自变量 z 趋近正无穷时，因变量 $g(z)$ 趋近于1，而当 z 趋近负无穷时， $g(z)$ 趋近于0，它能够将任何实数映射到(0,1)区间，使其可用于将任意值函数转换为更适合二分类的函数。

因为这个性质，Sigmoid函数也被当作是归一化的一种方法，与我们之前学过的MinMaxScaler同理，是属于数据预处理中的“缩放”功能，可以将数据压缩到[0,1]之内。区别在于，MinMaxScaler归一化之后，是可以取到0和1的（最大值归一化后就是1，最小值归一化后就是0），但Sigmoid函数只是无限趋近于0和1。

线性回归中 $z = \theta^T \mathbf{x}$ ，于是我们将 z 带入，就得到了二元逻辑回归模型的一般形式：

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

而 $y(x)$ 就是我们逻辑回归返回的标签值。此时， $y(x)$ 的取值都在[0,1]之间，因此 $y(x)$ 和 $1 - y(x)$ 相加必然为1。如果我们令 $y(x)$ 除以 $1 - y(x)$ 可以得到形似几率(odds)的 $\frac{y(x)}{1-y(x)}$ ，在此基础上取对数，可以很容易就得到：

$$\begin{aligned} \ln \frac{y(x)}{1 - y(x)} &= \ln \left(\frac{\frac{1}{1+e^{-\theta^T x}}}{1 - \frac{1}{1+e^{-\theta^T x}}} \right) \\ &= \ln \left(\frac{\frac{1}{1+e^{-\theta^T x}}}{\frac{e^{-\theta^T x}}{1+e^{-\theta^T x}}} \right) \\ &= \ln \left(\frac{1}{e^{-\theta^T x}} \right) \\ &= \ln(e^{\theta^T x}) \\ &= \theta^T x \end{aligned}$$

不难发现， $y(x)$ 的形似几率取对数的本质其实就是我们的线性回归 z ，我们实际上是在对线性回归模型的预测结果取对数几率来让其的结果无限逼近0和1。因此，其对应的模型被称为“对数几率回归”(logistic Regression)，也就是我们的逻辑回归，这个名为“回归”却是用来做分类工作的分类器。

之前我们提到过，线性回归的核心任务是通过求解 θ 构建 z 这个预测函数，并希望预测函数 z 能够尽量拟合数据，因此逻辑回归的核心任务也是类似的：求解 θ 来构建一个能够尽量拟合数据的预测函数 $y(x)$ ，并通过向预测函数中输入特征矩阵来获取相应的标签值 y 。

思考： $y(x)$ 代表了样本为某一类标签的概率吗？

$\ln \frac{y(x)}{1-y(x)}$ 是形似对数几率的一种变化。而几率odds的本质其实是 $\frac{p}{1-p}$ ，其中 p 是事件A发生的概率，而 $1-p$ 是事件A不会发生的概率，并且 $p+(1-p)=1$ 。因此，很多人在理解逻辑回归时，都对 $y(x)$ 做出如下的解释：

我们让线性回归结果逼近0和1，此时 $y(x)$ 和 $1 - y(x)$ 之和为1，因此它们可以被我们看作是一对正反例发生的概率，即 $y(x)$ 是某样本i的标签被预测为1的概率，而 $1 - y(x)$ 是i的标签被预测为0的概率， $\frac{y(x)}{1-y(x)}$ 就是样本i的标签被预测为1的相对概率。基于这种理解，我们使用最大似然法和概率分布函数推导出逻辑回归的损失函数，并且把返回样本在标签取值上的概率当成是逻辑回归的性质来使用，每当我们诉求概率的时候，我们都会使用逻辑回归。

然而这种理解是正确的吗？概率是度量偶然事件发生可能性的数值，尽管逻辑回归的取值在(0,1)之间，并且 $y(x)$ 和 $1 - y(x)$ 之和的确为1，但光凭这个性质，我们就可以认为 $y(x)$ 代表了样本x在标签上取值为1的概率吗？设想我们使用MaxMinScaler对特征进行归一化后，任意特征的取值也在[0,1]之间，并且任意特征的取值 x_0 和 $1 - x_0$ 也能够相加为1，但我们却不会认为0-1归一化后的特征是某种概率。**逻辑回归返回了概率**这个命题，这种说法严谨吗？

但无论如何，长年以来人们都是以“返回概率”的方式来理解逻辑回归，并且这样使用它的性质。可以说，逻辑回归返回的数字，即便本质上不是概率，却也有着概率的各种性质，可以被当成是概率来看待和使用。

1.2 为什么需要逻辑回归

线性回归对数据的要求很严格，比如标签必须满足正态分布，特征之间的多重共线性需要消除等等，而现实中很多真实情景的数据无法满足这些要求，因此线性回归在很多现实情境的应用效果有限。逻辑回归是由线性回归变化而来，因此它对数据也有一些要求，而我们之前已经学过了强大的分类模型决策树和随机森林，它们的分类效力很强，并且不需要对数据做任何预处理。

何况，逻辑回归的原理其实并不简单。一个人要理解逻辑回归，必须要有一定的数学基础，必须理解损失函数，正则化，梯度下降，海森矩阵等等这些复杂的概念，才能够对逻辑回归进行调优。其涉及到的数学理念，不比支持向量机少多少。况且，要计算概率，朴素贝叶斯可以计算出真正意义上的概率，要进行分类，机器学习中能够完成二分类功能的模型简直多如牛毛。因此，在数据挖掘，人工智能所涉及到的医疗，教育，人脸识别，语音识别这些领域，逻辑回归没有太多的出场机会。

甚至，在我们的各种机器学习经典书目中，周志华的《机器学习》400页仅有一页纸是关于逻辑回归的（还是一页数学公式），《数据挖掘导论》和《Python数据科学手册》中完全没有逻辑回归相关的内容，sklearn中对比各种分类器的效应也不带逻辑回归玩，可见业界地位。

但是，无论机器学习领域如何折腾，逻辑回归依然一个受工业商业热爱，使用广泛的模型，因为它有着不可替代的优点：

1. **逻辑回归对线性关系的拟合效果好到丧心病狂**，特征与标签之间的线性关系极强的数据，比如金融领域中的信用卡欺诈，评分卡制作，电商中的营销预测等等相关的数据，都是逻辑回归的强项。虽然现在有了梯度提升树GDBT，比逻辑回归效果更好，也被许多数据咨询公司启用，但逻辑回归在金融领域，尤其是银行业中的统治地位依然不可动摇（相对的，逻辑回归在非线性数据的效果很多时候比瞎猜还不如，所以如果你已经知道数据之间的联系是非线性的，千万不要迷信逻辑回归）
2. **逻辑回归计算快**：对于线性数据，逻辑回归的拟合和计算都非常快，计算效率优于SVM和随机森林，亲测表示在大型数据上尤其能够看得出区别
3. **逻辑回归返回的分类结果不是固定的0, 1，而是以小数形式呈现的类概率数字**：我们因此可以把逻辑回归返回的结果当成连续型数据来利用。比如在评分卡制作时，我们不仅需要判断客户是否会违约，还需要给出确定的“信用分”，而这个信用分的计算就需要使用类概率计算出的对数几率，而决策树和随机森林这样的分类器，可以产出分类结果，却无法帮助我们计算分数（当然，在sklearn中，决策树也可以产生概率，使用接口predict_proba调用就好，但一般来说，正常的决策树没有这个功能）。

另外，逻辑回归还有抗噪能力强的优点。福布斯杂志在讨论逻辑回归的优点时，甚至有着“技术上来说，最佳模型的AUC面积低于0.8时，逻辑回归非常明显优于树模型”的说法。并且，逻辑回归在小数据集上表现更好，在大型的数据集上，树模型有着更好的表现。

由此，我们已经了解了逻辑回归的本质，它是一个返回对数几率的，在线性数据上表现优异的分类器，它主要被应用在金融领域。**其数学目的是求解能够让模型对数据拟合程度最高的参数 θ 的值，以此构建预测函数 $y(x)$ ，然后将特征矩阵输入预测函数来计算出逻辑回归的结果y。**注意，虽然我们熟悉的逻辑回归通常被用于处理二分类问题，但逻辑回归也可以做多分类。

1.3 sklearn中的逻辑回归

逻辑回归相关的类	说明
linear_model.LogisticRegression	逻辑回归分类器（又叫logit回归，最大熵分类器）
linear_model.LogisticRegressionCV	带交叉验证的逻辑回归分类器
linear_model.logistic_regression_path	计算Logistic回归模型以获得正则化参数的列表
linear_model.SGDClassifier	利用梯度下降求解的线性分类器（SVM，逻辑回归等等）
linear_model.SGDRegressor	利用梯度下降最小化正则化后的损失函数的线性回归模型
metrics.log_loss	对数损失，又称逻辑损失或交叉熵损失
【在sklearn0.21版本中即将被移除】	
linear_model.RandomizedLogisticRegression	随机的逻辑回归
其他会涉及的类	说明
metrics.confusion_matrix	混淆矩阵，模型评估指标之一
metrics.roc_auc_score	ROC曲线，模型评估指标之一
metrics.accuracy_score	精确性，模型评估指标之一

2 linear_model.LogisticRegression

```
class sklearn.linear_model.LogisticRegression(penalty='l2', dual=False, tol=0.0001, C=1.0,
fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='warn', max_iter=100,
multi_class='warn', verbose=0, warm_start=False, n_jobs=None)
```

2.1 二元逻辑回归的损失函数

2.1.1 损失函数的概念与解惑

在学习决策树和随机森林时，我们曾经提到过两种模型表现：在训练集上的表现，和在测试集上的表现。我们建模，是追求模型在测试集上的表现最优，因此模型的评估指标往往是用来衡量模型在测试集上的表现的。然而，逻辑回归有着基于训练数据求解参数 θ 的需求，并且希望训练出来的模型能够尽可能地拟合训练数据，即模型在训练集上的预测准确率越靠近100%越好。

因此，我们使用“**损失函数**”这个评估指标，来衡量参数为 θ 的模型拟合训练集时产生的信息损失的大小，并以此衡量参数 θ 的优劣。如果用一组参数建模后，模型在训练集上表现良好，那我们就说模型拟合过程中的损失很小，损失函数的值很小，这一组参数就优秀；相反，如果模型在训练集上表现糟糕，损失函数就会很大，模型就训练不足，效果较差，这一组参数也就比较差。即是说，我们在求解参数 θ 时，追求损失函数最小，让模型在训练数据上的拟合效果最优，即预测准确率尽量靠近100%。

关键概念：损失函数

衡量参数 θ 的优劣的评估指标，用来求解最优参数的工具

损失函数小，模型在训练集上表现优异，拟合充分，参数优秀

损失函数大，模型在训练集上表现差劲，拟合不足，参数糟糕

我们追求，能够让损失函数最小化的参数组合

注意：没有“求解参数”需求的模型没有损失函数，比如KNN，决策树

逻辑回归的损失函数是由极大似然估计推导出来的，具体结果可以写作：

$$J(\theta) = - \sum_{i=1}^m (y_i * \log(y_\theta(x_i)) + (1 - y_i) * \log(1 - y_\theta(x_i)))$$

其中， θ 表示求解出来的一组参数， m 是样本的个数， y_i 是样本*i*上真实的标签， $y_\theta(x_i)$ 是样本*i*上，基于参数 θ 计算出来的逻辑回归返回值， x_i 是样本*i*各个特征的取值。我们的目标，就是求解出使 $J(\theta)$ 最小的 θ 取值。注意，在逻辑回归的本质函数 $y(x)$ 里，特征矩阵 x 是自变量，参数是 θ 。但在损失函数中，参数 θ 是损失函数的自变量， x 和 y 都是已知的特征矩阵和标签，相当于是损失函数的参数。不同的函数中，自变量和参数各有不同，因此大家需要在数学计算中，尤其是求导的时候避免混淆。

由于我们追求损失函数的最小值，让模型在训练集上表现最优，可能会引发另一个问题：如果模型在训练集上表示优秀，却在测试集上表现糟糕，模型就会过拟合。虽然逻辑回归和线性回归是天生欠拟合的模型，但我们还是需要控制过拟合的技术来帮助我们调整模型，**对逻辑回归中过拟合的控制，通过正则化来实现。**

2.1.2【选学】二元逻辑回归损失函数的数学解释，公式推导与解惑

虽然我们质疑过“逻辑回归返回概率”这样的说法，但不可否认逻辑回归的整个理论基础都是建立在这样的理解上的。在这里，我们基于极大似然法来推导二元逻辑回归的损失函数，这个推导过程能够帮助我们了解损失函数怎么得来的，以及为什么 $J(\theta)$ 的最小化能够实现模型在训练集上的拟合最好。

请时刻记得我们的目标：让模型对训练数据的效果好，追求损失最小。

二元逻辑回归的标签服从伯努利分布(即0-1分布)，因此我们可以将一个特征向量为 x ，参数为 θ 的模型中的一个样本*i*的预测情况表现为如下形式：

样本*i*在由特征向量 x_i 和参数 θ 组成的预测函数中，样本标签被预测为1的概率为：

$$P_1 = P(\hat{y}_i = 1 | x_i, \theta) = y_\theta(x_i)$$

样本*i*在由特征向量 x_i 和参数 θ 组成的预测函数中，样本标签被预测为0的概率为：

$$P_0 = P(\hat{y}_i = 0 | x_i, \theta) = 1 - y_\theta(x_i)$$

当 P_1 的值为1的时候，代表样本*i*的标签被预测为1，当 P_0 的值为1的时候，代表样本*i*的标签被预测为0。

我们假设样本*i*的真实标签 y_i 为1，此时如果 P_1 为1， P_0 为0，就代表样本*i*的标签被预测为1，与真实值一致。此时对于单样本*i*来说，模型的预测就是完全准确的，拟合程度很优秀，没有任何信息损失。相反，如果 P_1 此时为0， P_0 为1，就代表样本*i*的标签被预测为0，与真实情况完全相反。对于单样本*i*来说，模型的预测就是完全错误的，拟合程度很差，所有的信息都损失了。当 y_i 为0时，也是同样的道理。所以，当 y_i 为1的时候，我们希望 P_1 非常接近1，当 y_i 为0的时候，我们希望 P_0 非常接近1，这样，模型的效果就很好，信息损失就很少。

真实标签 y_i	被预测为1 的概率 P_1	被预测为0 的概率 P_0	样本被预 测为？	与真实值 一致吗？	拟合状况	信息损失
1	0	1	0	否	坏	大
1	1	0	1	是	好	小
0	0	1	0	是	好	小
0	1	0	1	否	坏	大

将两种取值的概率整合，我们可以定义如下等式：

$$P(\hat{y}_i | x_i, \theta) = P_1^{y_i} * P_0^{1-y_i}$$

这个等式代表同时代表了 P_1 和 P_0 。当样本*i*的真实标签 y_i 为1的时候， $1 - y_i$ 就等于0， P_0 的0次方就是1，所以 $P(\hat{y}_i | x_i, \theta)$ 就等于 P_1 ，这个时候，如果 P_1 为1，模型的效果就很好，损失就很小。同理，当 y_i 为0的时候， $P(\hat{y}_i | x_i, \theta)$ 就等于 P_0 ，此时如果 P_0 非常接近1，模型的效果就很好，损失就很小。**所以，为了达成让模型拟合好，损失小的目的，我们每时每刻都希望 $P(\hat{y}_i | x_i, \theta)$ 的值等于1。**而 $P(\hat{y}_i | x_i, \theta)$ 的本质是样本*i*由特征向量 x_i 和参数 θ 组成的预测函数中，预测出所有可能的 \hat{y}_i 的概率，因此1是它的最大值。**也就是说，每时每刻，我们都在追求 $P(\hat{y}_i | x_i, \theta)$ 的最大值。**这就将模型拟合中的“最小化损失”问题，转换成了对函数求解极值的问题。

$P(\hat{y}_i | x_i, \theta)$ 是对单个样本*i*而言的函数，对一个训练集的m个样本来说，我们可以定义如下等式来表达所有样本在特征矩阵X和参数 θ 组成的预测函数中，预测出所有可能的 \hat{y} 的概率 P 为：

$$\begin{aligned}
 P &= \prod_{i=1}^m P(\hat{y}_i | x_i, \theta) \\
 &= \prod_{i=1}^m (P_1^{y_i} * P_0^{1-y_i}) \\
 &= \prod_{i=1}^m (y_\theta(x_i)^{y_i} * (1 - y_\theta(x_i))^{1-y_i})
 \end{aligned}$$

对该概率P取对数，再由 $\log(A * B) = \log A + \log B$ 和 $\log A^B = B \log A$ 可得到：

$$\begin{aligned}
 \log P &= \log \prod_{i=1}^m (y_\theta(x_i)^{y_i} * (1 - y_\theta(x_i))^{1-y_i}) \\
 &= \sum_{i=1}^m \log(y_\theta(x_i)^{y_i} * (1 - y_\theta(x_i))^{1-y_i}) \\
 &= \sum_{i=1}^m (\log y_\theta(x_i)^{y_i} + \log(1 - y_\theta(x_i))^{1-y_i}) \\
 &= \sum_{i=1}^m (y_i * \log(y_\theta(x_i)) + (1 - y_i) * \log(1 - y_\theta(x_i)))
 \end{aligned}$$

这就是我们的交叉熵函数。为了数学上的便利以及更好地定义“损失”的含义，我们希望将极大值问题转换为极小值问题，因此我们对 $\log P$ 取负，并且让参数 θ 作为函数的自变量，就得到了我们的损失函数 $J(\theta)$ ：

$$J(\theta) = - \sum_{i=1}^m (y_i * \log(y_\theta(x_i)) + (1 - y_i) * \log(1 - y_\theta(x_i)))$$

这就是一个，基于逻辑回归的返回值 $y_\theta(x_i)$ 的概率性质得出的损失函数。**在这个函数上，我们只要追求最小值，就能让模型在训练数据上的拟合效果最好，损失最低。**这个推导过程，其实就是“极大似然法”的推导过程。

关键概念：似然与概率

似然与概率是一组非常相似的概念，它们都代表着某件事发生的可能性，但它们在统计学和机器学习中有着微妙的不同。以样本*i*为例，我们有表达式：

$$P(\hat{y}_i | x_i, \theta)$$

对这个表达式而言，如果参数 θ 是已知的，特征向量 x_i 是未知的，我们便称P是在探索不同特征取值下获取所有可能的 \hat{y} 的可能性，这种可能性就被称为**概率**，研究的是自变量和因变量之间的关系。

如果特征向量 x_i 是已知的，参数 θ 是未知的，我们便称P是在探索不同参数下获取所有可能的 \hat{y} 的可能性，这种可能性就被称为**似然**，研究的是参数取值与因变量之间的关系。

在逻辑回归的建模过程中，我们的特征矩阵是已知的，参数是未知的，因此我们讨论的所有“概率”其实严格来说都应该是“似然”。我们追求 $P(\hat{y}_i | x_i, \theta)$ 的最大值（换算成损失函数之后取负了，所以是最小值），就是在追求“极大似然”，所以逻辑回归的损失函数的推导方法叫做“极大似然法”。也因此，以下式子又被称为“极大似然函数”：

$$P(\hat{y}_i | x_i, \theta) = y_\theta(x_i)^{y_i} * (1 - y_\theta(x_i))^{1-y_i}$$

2.2 重要参数penalty & C

2.2.1 正则化

正则化是用来防止模型过拟合的过程，常用的有L1正则化和L2正则化两种选项，分别通过在损失函数后加上参数向量 θ 的L1范式和L2范式的倍数来实现。这个增加的范式，被称为“正则项”，也被称为“惩罚项”。损失函数改变，基于损失函数的最优化来求解的参数取值必然改变，我们以此来调节模型拟合的程度。其中L1范式表现为参数向量中的每个参数的绝对值之和，L2范数表现为参数向量中的每个参数的平方和的开方值。

$$J(\theta)_{L1} = C * J(\theta) + \sum_{j=1}^n |\theta_j| \quad (j \geq 1)$$

$$J(\theta)_{L2} = C * J(\theta) + \sqrt{\sum_{j=1}^n (\theta_j)^2} \quad (j \geq 1)$$

其中 $J(\theta)$ 是我们之前提过的损失函数，C是用来控制正则化程度的超参数，n是方程中特征的总数，也是方程中参数的总数，j代表每个参数。在这里，j要大于等于1，是因为我们的参数向量 θ 中，第一个参数是 θ_0 ，是我们的截距，它通常是不参与正则化的。

在许多书籍和博客中，大家可能也会见到如下的写法：

$$J(\theta)_{L1} = J(\theta) + \frac{1}{2b^2} \sum_j |\theta_j|$$

$$J(\theta)_{L2} = J(\theta) + \frac{\theta^T \theta}{2\sigma^2}$$

其实和上面我们展示的式子的本质是一模一样的。不过在大多数教材和博客中，常数项是乘以正则项，通过调控正则项来调节对模型的惩罚。而sklearn当中，常数项C是在损失函数的前面，通过调控损失函数本身的大小，来调节对模型的惩罚。

参数	说明
penalty	可以输入"l1"或"l2"来指定使用哪一种正则化方式，不填写默认"l2"。 注意，若选择"l1"正则化，参数solver仅能够使用求解方式"liblinear"和"saga"，若使用"l2"正则化，参数solver中所有的求解方式都可以使用。
C	C正则化强度的倒数，必须是一个大于0的浮点数，不填写默认1.0，即默认正则项与损失函数的比值是1:1。C越小，损失函数会越小，模型对损失函数的惩罚越重，正则化的效力越强，参数 θ 会逐渐被压缩得越来越小。

L1正则化和L2正则化虽然都可以控制过拟合，但它们的效果并不相同。当正则化强度逐渐增大（即C逐渐变小），参数 θ 的取值会逐渐变小，但**L1正则化会将参数压缩为0，L2正则化只会让参数尽量小，不会取到0。**

在L1正则化在逐渐加强的过程中，携带信息量小的、对模型贡献不大的特征的参数，会比携带大量信息的、对模型有巨大贡献的特征的参数更快地变成0，所以L1正则化本质是一个特征选择的过程，掌管了参数的“稀疏性”。L1正则化越强，参数向量中就越多的参数为0，参数就越稀疏，选出来的特征就越少，以此来防止过拟合。因此，如果特征量很大，数据维度很高，我们会倾向于使用L1正则化。由于L1正则化的这个性质，逻辑回归的特征选择可以由Embedded嵌入法来完成。

相对的，L2正则化在加强的过程中，会尽量让每个特征对模型都有一些小的贡献，但携带信息少，对模型贡献不大的特征的参数会非常接近于0。通常来说，如果我们的主要目的只是为了防止过拟合，选择L2正则化就足够了。但是如果选择L2正则化后还是过拟合，模型在未知数据集上的效果表现很差，就可以考虑L1正则化。

而两种正则化下C的取值，都可以通过学习曲线来进行调整。

建立两个逻辑回归，L1正则化和L2正则化的差别就一目了然了：

```
from sklearn.linear_model import LogisticRegression as LR
from sklearn.datasets import load_breast_cancer
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

data = load_breast_cancer()
X = data.data
y = data.target

data.data.shape

lrl1 = LR(penalty="l1", solver="liblinear", C=0.5, max_iter=1000)

lrl2 = LR(penalty="l2", solver="liblinear", C=0.5, max_iter=1000)

#逻辑回归的重要属性coef_，查看每个特征所对应的参数
lrl1 = lrl1.fit(X, y)
lrl1.coef_

(lrl1.coef_ != 0).sum(axis=1)

lrl2 = lrl2.fit(X, y)
lrl2.coef_
```

可以看见，当我们选择L1正则化的时候，许多特征的参数都被设置为了0，这些特征在真正建模的时候，就不会出现在我们的模型当中了，而L2正则化则是对所有的特征都给出了参数。

究竟哪个正则化的效果更好呢？还是都差不多？

```
l1 = []
l2 = []
l1test = []
l2test = []

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

for i in np.linspace(0.05, 1, 19):
    lrl1 = LR(penalty="l1", solver="liblinear", C=i, max_iter=1000)
    lrl2 = LR(penalty="l2", solver="liblinear", C=i, max_iter=1000)

    lrl1 = lrl1.fit(Xtrain, Ytrain)
    l1.append(accuracy_score(lrl1.predict(Xtrain), Ytrain))
    l1test.append(accuracy_score(lrl1.predict(Xtest), Ytest))

    lrl2 = lrl2.fit(Xtrain, Ytrain)
    l2.append(accuracy_score(lrl2.predict(Xtrain), Ytrain))
    l2test.append(accuracy_score(lrl2.predict(Xtest), Ytest))
```

```

lrl2 = lrl2.fit(Xtrain,Ytrain)
l2.append(accuracy_score(lrl2.predict(Xtrain),Ytrain))
l2test.append(accuracy_score(lrl2.predict(Xtest),Ytest))

graph = [l1,l2,l1test,l2test]
color = ["green","black","lightgreen","gray"]
label = ["L1","L2","L1test","L2test"]

plt.figure(figsize=(6,6))
for i in range(len(graph)):
    plt.plot(np.linspace(0.05,1,19),graph[i],color[i],label=label[i])
plt.legend(loc=4) #图例的位置在哪里?4表示, 右下角
plt.show()

```

可见，至少在我们的乳腺癌数据集下，两种正则化的结果区别不大。但随着C的逐渐变大，正则化的强度越来越小，模型在训练集和测试集上的表现都呈上升趋势，直到C=0.8左右，训练集上的表现依然在走高，但模型在未知数据集上的表现开始下跌，这时候就是出现了过拟合。我们可以认为，C设定为0.8会比较好。在实际使用时，基本就默认使用L2正则化，如果感觉到模型的效果不好，那就换L1试试看。

2.2.2 逻辑回归中的特征工程

当特征的数量很多的时候，我们出于业务考虑，也出于计算量的考虑，希望对逻辑回归进行特征选择来降维。比如，在判断一个人是否会患乳腺癌的时候，医生如果看5~8个指标来确诊，会比需要看30个指标来确诊容易得多。

- **业务选择**

说到降维和特征选择，首先要想到的是利用自己的业务能力进行选择，肉眼可见明显和标签有关的特征就是需要留下的。当然，如果我们并不了解业务，或者有成千上万的特征，那我们也可以使用算法来帮助我们。或者，可以让算法先帮助我们筛选过一遍特征，然后在少量的特征中，我们再根据业务常识来选择更少量的特征。

- **PCA和SVD一般不用**

说到降维，我们首先想到的是之前提过的高效降维算法，PCA和SVD，遗憾的是，这两种方法大多数时候不适用于逻辑回归。逻辑回归是由线性回归演变而来，线性回归的一个核心目的是通过求解参数来探究特征X与标签y之间的关系，而逻辑回归也传承了这个性质，我们常常希望通过逻辑回归的结果，来判断什么样的特征与分类结果相关，因此我们希望保留特征的原貌。PCA和SVD的降维结果是不可解释的，因此一旦降维后，我们就无法解释特征和标签之间的关系了。当然，在不需要探究特征与标签之间关系的线性数据上，降维算法PCA和SVD也是可以使用的。

- **统计方法可以使用，但不是非常必要**

既然降维算法不能使用，我们要用的就是特征选择方法。逻辑回归对数据的要求低于线性回归，由于我们不是使用最小二乘法来求解，所以逻辑回归对数据的总体分布和方差没有要求，也不需要排除特征之间的共线性，但如果我们确实希望使用一些统计方法，比如方差，卡方，互信息等方法来做特征选择，也并没有问题。过滤法中所有的方法，都可以用在逻辑回归上。

在一些博客中有这样的观点：多重共线性会影响线性模型的效果。对于线性回归来说，多重共线性会影响比较大，所以我们需要使用方差过滤和方差膨胀因子VIF(variance inflation factor)来消除共线性。但是对于逻辑回归，其实不是非常必要，甚至有时候，我们还需要多一些相互关联的特征来增强模型的表现。当然，如果我们无法通过其他方式提升模型表现，并且你感觉到模型中的共线性影响了模型效果，那懂得统计学的你可以试试看用VIF消除共线性的方法，遗憾的是现在sklearn中并没有提供VIF的功能。

轻松一刻

R vs Python, 统计学 vs 机器学习

有许多学过R，或者和python一起学习R的小伙伴，曾向我问起各种各样的统计学问题，因为R中有各种各样的统计功能，而python的统计学功能并不是那么全面。我也曾经被小伙伴们发给我的“R风格python代码”弄得晕头转向，也许R的代码希望看起来高大上，但python之美就是简单明了（P.S. Python开发者十分有情怀，在jupyter中输入import this可以查看python中隐含的彩蛋，python制作者所写的诗歌“python之禅”，通篇赞美了python代码的简单，明快，容易阅读之美，大家感兴趣的可以百度搜一搜看看翻译）。

回归正题，为什么python和R在统计学的功能上差异如此之大呢？也许大家听说过，R是学统计学的人开发的，因此整个思路都是统计学的思路，而python是学计算机的人开发的，因此整个思路都是计算机的思路，也难怪R在处理统计问题上比python强很多了，这两种学科不同的思路强烈反应在统计学和机器学习的各种建模流程当中。

统计学的思路是一种“先验”的思路，不管做什么都要先“检验”，先“满足条件”，事后也要各种“检验”，以确保各种数学假设被满足，不然的话，理论上就无法得出好结果。而机器学习是一种“后验”的思路，不管三七二十一，我先让模型跑一跑，效果不好我再想办法，如果模型效果好，我完全不在意什么共线性，残差不满足正态分布，没有哑变量之类的细节，模型效果好大过天！

作为一个非数学，非统计出身，从金融半路出家来敲python代码的人，我完全欣赏机器学习的这种“后验”思路：我们追求结果，不要过于在意那些需要满足的先决条件。对我而言，统计学是机器学习穷尽所有手段都无法解决问题后的“救星”，如果机器学习不能解决问题，我会向统计学寻求帮助，但我绝不会一开始就想带着要去满足各种统计要求。当然啦，如果大家是学统计学出身，写R出身，大家也可以把机器学习当成是统计学手段用尽后的“救星”。统计学和机器学习是相辅相成的，大家要了解两种思路的不同，以便在进入死胡同的时候，可以从另一个学科的思路中找到出路。只要能够解决问题的，都是好思路！

- 高效的嵌入法embedded

但是更有效的方法，毫无疑问会是我们的embedded嵌入法。我们已经说明了，由于L1正则化会使得部分特征对应的参数为0，因此L1正则化可以用来做特征选择，结合嵌入法的模块SelectFromModel，我们可以很容易就筛选出让模型十分高效的特征。注意，此时我们的目的是，尽量保留原数据上的信息，让模型在降维后的数据上的拟合效果保持优秀，因此我们不考虑训练集测试集的问题，把所有的数据都放入模型进行降维。

```
from sklearn.linear_model import LogisticRegression as LR
from sklearn.datasets import load_breast_cancer
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import cross_val_score
from sklearn.feature_selection import SelectFromModel

data = load_breast_cancer()
data.data.shape

LR_ = LR(solver="liblinear", C=0.9, random_state=420)
cross_val_score(LR_, data.data, data.target, cv=10).mean()

X_embedded = SelectFromModel(LR_, norm_order=1).fit_transform(data.data, data.target)

X_embedded.shape
```

```
cross_val_score(LR_, X_embedded, data.target, cv=10).mean()
```

看看结果，特征数量被减小到个位数，并且模型的效果却没有下降太多，如果我们要求不高，在这里其实就可以停下了。但是，能否让模型的拟合效果更好呢？在这里，我们有两种调整方式：

1) 调节SelectFromModel这个类中的参数threshold，这是嵌入法的阈值，表示删除所有参数的绝对值低于这个阈值的特征。现在threshold默认为None，所以SelectFromModel只根据L1正则化的结果来选择了特征，即选择了所有L1正则化后参数不为0的特征。我们此时，只要调整threshold的值（画出threshold的学习曲线），就可以观察不同的threshold下模型的效果如何变化。一旦调整threshold，就不是在使用L1正则化选择特征，而是使用模型的属性.coef_中生成的各个特征的系数来选择。coef_虽然返回的是特征的系数，但是系数的大小和决策树中的feature_importances_以及降维算法中的可解释性方差explained_vairance_概念相似，其实都是衡量特征的重要程度和贡献度的，因此SelectFromModel中的参数threshold可以设置为coef_的阈值，即可以剔除系数小于threshold中输入的数字的所有特征。

```
fullx = []
fsx = []

threshold = np.linspace(0, abs((LR_.fit(data.data, data.target).coef_)).max(), 20)

k=0
for i in threshold:
    X_embedded = SelectFromModel(LR_, threshold=i).fit_transform(data.data, data.target)
    fullx.append(cross_val_score(LR_, data.data, data.target, cv=5).mean())
    fsx.append(cross_val_score(LR_, X_embedded, data.target, cv=5).mean())
    print((threshold[k], X_embedded.shape[1]))
    k+=1

plt.figure(figsize=(20, 5))
plt.plot(threshold, fullx, label="full")
plt.plot(threshold, fsx, label="feature selection")
plt.xticks(threshold)
plt.legend()
plt.show()
```

然而，这种方法其实是比较无效的，大家可以用学习曲线来跑一跑：当threshold越来越大，被删除的特征越来越多，模型的效果也越来越差，模型效果最好的情况下需要保证有17个以上的特征。实际上我画了细化的学习曲线，如果要保证模型的效果比降维前更好，我们需要保留25个特征，这对于现实情况来说，是一种无效的降维：需要30个指标来判断病情，和需要25个指标来判断病情，对医生来说区别不大。

2) 第二种调整方法，是调逻辑回归的类LR_，通过画C的学习曲线来实现：

```
fullx = []
fsx = []

C=np.arange(0.01, 10.01, 0.5)

for i in C:
    LR_ = LR(solver="liblinear", C=i, random_state=420)

    fullx.append(cross_val_score(LR_, data.data, data.target, cv=10).mean())

    X_embedded = SelectFromModel(LR_, norm_order=1).fit_transform(data.data, data.target)
```

```

fsx.append(cross_val_score(LR_, X_embedded, data.target, cv=10).mean())

print(max(fsx), C[fsx.index(max(fsx))])

plt.figure(figsize=(20,5))
plt.plot(C, fullx, label="full")
plt.plot(C, fsx, label="feature selection")
plt.xticks(C)
plt.legend()
plt.show()

```

继续细化学习曲线:

```

fullx = []
fsx = []

C=np.arange(6.05,7.05,0.005)

for i in C:
    LR_ = LR(solver="liblinear", C=i, random_state=420)

    fullx.append(cross_val_score(LR_, data.data, data.target, cv=10).mean())

    X_embedded = SelectFromModel(LR_, norm_order=1).fit_transform(data.data, data.target)
    fsx.append(cross_val_score(LR_, X_embedded, data.target, cv=10).mean())

print(max(fsx), C[fsx.index(max(fsx))])

plt.figure(figsize=(20,5))
plt.plot(C, fullx, label="full")
plt.plot(C, fsx, label="feature selection")
plt.xticks(C)
plt.legend()
plt.show()

#验证模型效果：降维之前
LR_ = LR(solver="liblinear", C=6.069999999999999, random_state=420)
cross_val_score(LR_, data.data, data.target, cv=10).mean()

#验证模型效果：降维之后
LR_ = LR(solver="liblinear", C=6.069999999999999, random_state=420)
X_embedded = SelectFromModel(LR_, norm_order=1).fit_transform(data.data, data.target)
cross_val_score(LR_, X_embedded, data.target, cv=10).mean()

X_embedded.shape

```

这样我们就实现了在特征选择的前提下，保持模型拟合的高效，现在，如果有一位医生可以来为我们指点迷津，看看剩下的这些特征中，有哪些是对针对病情来说特别重要的，也许我们还可以继续降维。当然，除了嵌入法，系数累加法或者包装法也是可以使用的。

- 比较麻烦的系数累加法

系数累加法的原理非常简单。在PCA中，我们通过绘制累积可解释方差贡献率曲线来选择超参数，在逻辑回归中我们可以使用系数coef_来这样做，并且我们选择特征个数的逻辑也是类似的：找出曲线由锐利变平滑的转折点，转折点之前被累加的特征都是我们需要的，转折点之后的我们都不需要。不过这种方法相对比较麻烦，因为我们要先对特征系数进行从大到小的排序，还要确保我们知道排序后的每个系数对应的原始特征的位置，才能够正确找出那些重要的特征。如果要使用这样的方法，不如直接使用嵌入法来得方便。

- **简单快速的包装法**

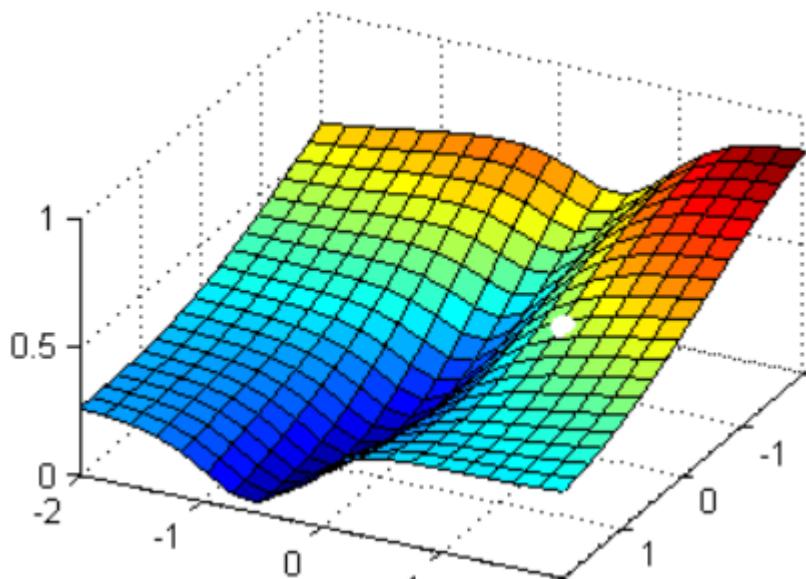
相对的，包装法可以直接设定我们需要的特征个数，逻辑回归在现实中运用时，可能会有“需要5~8个变量”这种需求，包装法此时就非常方便了。不过逻辑回归的包装法的使用和其他算法一样，并不具有特别之处，因此在这里就不在赘述，具体大家可以参考03期：数据预处理和特征工程中的代码。

2.3 梯度下降：重要参数max_iter

逻辑回归的数学目的是求解能够让模型最优化，拟合程度最好的参数 θ 的值，即求解能够让损失函数 $J(\theta)$ 最小化的 θ 值。对于二元逻辑回归来说，有多种方法可以用来求解参数 θ ，最常见的有梯度下降法(Gradient Descent)，坐标下降法(Coordinate Descent)，牛顿法(Newton-Raphson method)等，其中又以梯度下降法最为著名。每种方法都涉及复杂的数学原理，但这些计算在执行的任务其实是类似的。

2.3.1 梯度下降求解逻辑回归

我们以最著名也最常用的梯度下降法为例，来看看逻辑回归的参数求解过程究竟实在做什么。现在有一个带两个特征并且没有截距的逻辑回归 $y(x_1, x_2)$ ，两个特征所对应的参数分别为 $[\theta_1, \theta_2]$ 。下面这个华丽的平面就是我们的损失函数 $J(\theta_1, \theta_2)$ 在以 θ_1 ， θ_2 和 J 为坐标轴的三维立体坐标系上的图像。现在，我们寻求的是损失函数的最小值，也就是图像的最低点。



那我们怎么做呢？我在这个图像上随机放一个小球，当我松手，这个小球就会顺着这个华丽的平面滚落，直到滚到深蓝色的区域——损失函数的最低点。为了严格监控这个小球的行为，我让小球每次滚动的距离有限，不让他一次性滚到最低点，并且最多只允许它滚动100步，还要记下它每次滚动的方向，直到它滚到图像上的最低点。

可以看见，小球从高处滑落，在深蓝色的区域中来回震荡，最终停留在了图像凹陷处的某个点上。非常明显，我们可以观察到几个现象：

首先，小球并不是一开始就直向着最低点去的，它先一口气冲到了蓝色区域边缘，后来又折回来，我们已经规定了小球是多次滚动，所以可见，**小球每次滚动的方向都是不同的**。

另外，小球在进入深蓝色区域后，并没有直接找到某个点，而是在深蓝色区域中来回震荡了数次才停下。这有两种可能：1) 小球已经滚到了图像的最低点，所以停下了，2) 由于我设定的步数限制，小球还没有找到最低点，但也只好在100步的时候停下了。**也就是说，小球不一定滚到了图像的最低处**。

但无论如何，小球停下的就是我们在现有状况下可以获得的唯一点了。如果我们够幸运，这个点就是图像的最低点，那我们只要找到这个点的对应坐标 $(\theta_1^*, \theta_2^*, J_{min})$ ，就可以获取能够让损失函数最小的参数取值 $[\theta_1^*, \theta_2^*]$ 了。如此，梯度下降的过程就已经完成。

在这个过程中，**小球其实就是一组组的坐标点 (θ_1, θ_2, J) ；小球每次滚动的方向就是那一个坐标点的梯度向量的方向**，因为每滚动一步，小球所在的位置都发生变化，坐标点和坐标点对应的梯度向量都发生了变化，所以每次滚动的方向也都不一样；**人为设置的100次滚动限制，就是sklearn中逻辑回归的参数max_iter，代表着能走的最大步数，即最大迭代次数**。

所以梯度下降，其实就是在众多 $[\theta_1, \theta_2]$ 可能的值中遍历，一次次求解坐标点的梯度向量，不断让损失函数的取值 J 逐渐逼近最小值，再返回这个最小值对应的参数取值 $[\theta_1^*, \theta_2^*]$ 的过程。

2.3.2 梯度下降的概念与解惑

那梯度究竟如何定义呢？在多元函数上对各个自变量求 ∂ 偏导数，把求得的各个自变量的偏导数以向量的形式写出来，就是梯度。比如损失函数 $J(\theta_1, \theta_2)$ ，其自变量是逻辑回归预测函数 $y_\theta(x)$ 的参数 θ_1, θ_2 ，在损失函数上对 θ_1, θ_2 求偏导数，求得的梯度向量 d 就是 $[\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}]^T$ ，简称 $\text{grad } J(\theta_1, \theta_2)$ 或者 $\nabla J(\theta_1, \theta_2)$ 。在 θ_1, θ_2 和 J 的取值构成的坐标系上，点 $(\theta_1^*, \theta_2^*, J)$ 具体的梯度向量就是 $[\frac{\partial J}{\partial \theta_1^*}, \frac{\partial J}{\partial \theta_2^*}]^T$ ，或者 $\nabla J(\theta_1^*, \theta_2^*)$ 。如果是3个参数的梯度向量，就是 $[\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \frac{\partial J}{\partial \theta_3}]^T$ ，以此类推。

核心误区：到底在哪个函数上，求什么的偏导数？

注意，在一些博客或教材中，讲解梯度向量的定义时会写一些让人容易误解的句子，比如“对多元函数的参数求 ∂ 偏导数，把求得的各个参数的偏导数以向量的形式写出来，就是梯度”。注意，这种解释一眼看上去没错，却是不太严谨的。

一个多元函数的梯度，是对其自变量求偏导的结果，不是对其参数求偏导的结果。但是在逻辑回归的数学过程中，损失函数的自变量刚好是逻辑回归的预测函数 $y(x)$ 的参数，所以才造成了这种让人误解的，“对多元函数的参数求偏导”的写法。务必记住，正确的做法是：**在多元函数(损失函数)上对自变量(逻辑回归的预测函数 $y(x)$ 的参数)求偏导**，求解梯度的方式，和逻辑回归本身的预测函数 $y(x)$ 没有一丝联系。

以及，有一些博客会以 $f(x, y)$ 作为例子，解释说梯度向量是 $(\partial f / \partial x, \partial f / \partial y)^T$ 。这种举例方式又会造成误解：很多人看到这个式子，会特别自然地理解成：“ x 是逻辑回归中的特征呀，所以梯度向量是对模型的特征，即 x 求偏导数”。这个例子，其实是在表明，我们是对多元函数的自变量求偏导数，并不是代表我们在逻辑回归中要对特征求偏导数。

在这里我会不厌其烦地给大家强调：**求解梯度，是在损失函数 $J(\theta_1, \theta_2)$ 上对损失函数自身的自变量 θ_1 和 θ_2 求偏导，而这两个自变量，刚好是逻辑回归的预测函数 $y(x) = \frac{1}{1+e^{-\theta^T x}}$ 的参数。**

那梯度有什么含义呢？梯度是一个向量，因此它有大小也有方向。它的大小，就是偏导数组成的向量的大小，又叫做向量的模，记作 d 。它的方向，几何上来说，就是损失函数 $J(\theta)$ 的值增加最快的方向，就是小球每次滚动的方向的反方向。只要沿着梯度向量的反方向移动坐标，损失函数 $J(\theta)$ 的取值就会减少得最快，也就最容易找到损失函数的最小值。

在逻辑回归中，我们的损失函数如下所示：

$$J(\theta) = - \sum_{i=1}^m (y_i * \log(y_\theta(x_i)) + (1 - y_i) * \log(1 - y_\theta(x_i)))$$

我们对这个函数上的自变量 θ 求偏导，就可以得到梯度向量在第 j 组 θ 的坐标点上的表示形式：

$$\frac{\partial}{\partial \theta_j} J(\theta) = d_j = \sum_{i=1}^m (y_\theta(x_i) - y_i) x_{ij}$$

在这个公式下，只要给定一组 θ 的取值 θ_j ，再带入特征矩阵 x ，就可以求得这一组 θ 取值下的预测结果 $y_\theta(x_i)$ ，结合真实标签向量 y ，就可以获得这一组 θ_j 取值下的梯度向量，其大小表示为 d_j 。之前说过，我们的目的是在可能的 θ 取值上进行遍历，一次次计算梯度向量，并在梯度向量的反方向上让损失函数 J 下降至最小值。在这个过程中，我们的 θ 和梯度向量的大小 d 都会不断改变，而我们遍历 θ 的过程可以描述为：

$$\begin{aligned}\theta_{j+1} &= \theta_j - \alpha * d_j \\ &= \theta_j - \alpha * \sum_{i=1}^m (y_\theta(x_i) - y_i)x_{ij}\end{aligned}$$

其中 θ_{j+1} 是第 j 次迭代后的参数向量， θ_j 是第 j 次迭代时的参数向量， α 被称为步长，控制着每走一步（每迭代一次）后 θ 的变化，并以此来影响每次迭代后的梯度向量的大小和方向。

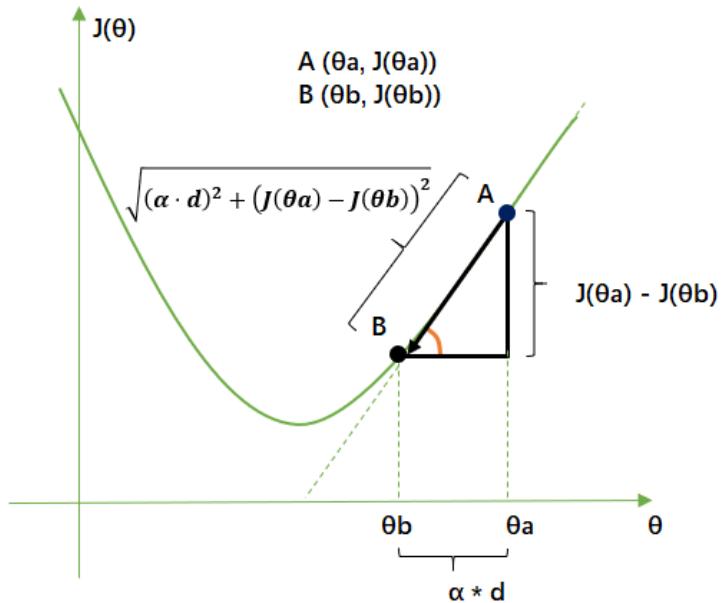
2.3.3 步长的概念与解惑

核心误区：步长到底是什么？

许多博客和教材在描述步长的时候，声称它是“梯度下降中每一步沿梯度的反方向前进的长度”，“沿着最陡峭最容易下山的位置走的那一步的长度”或者“梯度下降中每一步损失函数减小的量”，甚至有说，步长是二维平面著名的求导三角形中的“斜边”或者“对边”的。

这些说法都是错误的！

来看下面这一张二维平面的求导三角型图。类比到我们的损失函数和梯度概念上，图中的抛物线就是我们的损失函数 $J(\theta)$ ， $A(\theta_a, J(\theta_a))$ 就是小球最初在的位置， $B(\theta_b, J(\theta_b))$ 就是一次滚动后小球移动到的位置。从A到B的方向就是梯度向量的反方向，指向损失函数在A点下降最快的方向。而梯度向量的大小是点A在图像上对 θ 求导后的结果，也是点A切线方向的斜率，橙色角的tan结果，记作 d 。



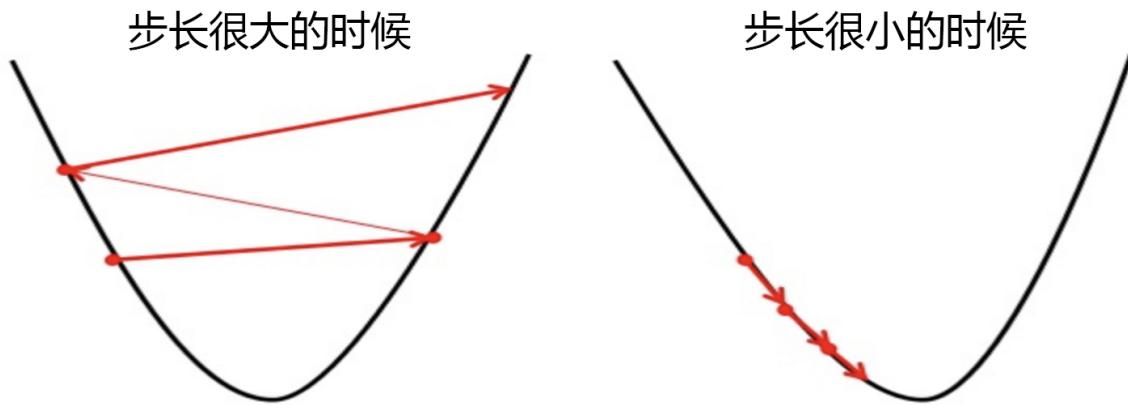
梯度下降每走一步，损失函数减小的量，是损失函数在 θ 变化之后的取值的变化，写作 $J(\theta_b) - J(\theta_a)$ ，这是二维平面的求导三角型中的“对边”。

梯度下降每走一步，参数向量的变化，写作 $\theta_a - \theta_b$ ，根据我们参数向量的迭代公式，就是我们的步长 * 梯度向量的大小，记作 $\alpha * d$ ，这是二维平面的求倒三角形中的“邻边”。

梯度下降中每走一步，下降的距离，是 $\sqrt{(\alpha * d)^2 + (J(\theta_a) - J(\theta_b))^2}$ ，是对边和邻边的根号下平方和，是二维平面的求导三角型中的“斜边”。

所以，步长不是任何物理距离，它甚至不是梯度下降过程中任何距离的直接变化，它是梯度向量的大小d上的一个比例，影响着参数向量 θ 每次迭代后改变的部分。

不难发现，既然参数迭代是靠梯度向量的大小 $d * \text{步长}\alpha$ 来实现的，而 $J(\theta)$ 的降低又是靠调节 θ 来实现的，所以步长可以调节损失函数下降的速率。在损失函数降低的方向上，步长越长， θ 的变动就越大。相对的，步长如果很短， θ 的每次变动就很小。具体地说，如果步长太大，损失函数下降得就非常快，需要的迭代次数就很少，但梯度下降过程可能跳过损失函数的最低点，无法获取最优值。而步长太小，虽然函数会逐渐逼近我们需要的最低点，但迭代的速度却很缓慢，迭代次数就需要很多。



记得我们在看小球运动时注意到，小球在进入深蓝色区域后，并没有直接找到某个点，而是在深蓝色区域中来回震荡了数次才停下，这种“震荡”其实就是因为设置的步长太大的缘故。但是在我们开始梯度下降之前，我们并不知道什么样的步长才合适，但梯度下降一定要在某个时候停止才可以，否则模型可能会无限地迭代下去。因此，在sklearn当中，我们设置参数max_iter最大迭代次数来代替步长，帮助我们控制模型的迭代速度并适时地让模型停下。max_iter越大，代表步长越小，模型迭代时间越长，反之，则代表步长设置很大，模型迭代时间很短。

迭代结束，获取到 $J(\theta)$ 的最小值后，我们就可以找出这个最小值对应的参数向量 θ ，逻辑回归的预测函数也就可以根据这个参数向量 θ 来建立了。

来看看乳腺癌数据集下，max_iter的学习曲线：

```

l2 = []
l2test = []

xtrain, xtest, Ytrain, Ytest = train_test_split(X,y,test_size=0.3,random_state=420)

for i in np.arange(1,201,10):
    lrL2 = LR(penalty="l2",solver="liblinear",C=0.9,max_iter=i)
    lrL2.fit(xtrain,Ytrain)
    l2.append(accuracy_score(lrL2.predict(xtrain),Ytrain))
    l2test.append(accuracy_score(lrL2.predict(xtest),Ytest))

graph = [l2,l2test]
color = ["black","gray"]
label = ["L2","L2test"]

plt.figure(figsize=(20,5))
for i in range(len(graph)):
    plt.plot(np.arange(1,201,10),graph[i],color[i],label=label[i])

```

```

plt.legend(loc=4)
plt.xticks(np.arange(1, 201, 10))
plt.show()

#我们可以使用属性.n_iter_来调用本次求解中真正实现的迭代次数

lr = LR(penalty="l2", solver="liblinear", C=0.9, max_iter=300).fit(xtrain, Ytrain)
lr.n_iter_

```

当max_iter中限制的步数已经走完了，逻辑回归却还没有找到损失函数的最小值，参数 θ 的值还没有被收敛，sklearn就会弹出这样的红色警告：

当参数solver="liblinear":

```
C:\Python\lib\site-packages\sklearn\svm\base.py:922: ConvergenceWarning: L
iblinear failed to converge, increase the number of iterations.
 "the number of iterations.", ConvergenceWarning)
```

当参数solver="sag":

```
C:\Python\lib\site-packages\sklearn\linear_model\sag.py:334: ConvergenceWa
rning: The max_iter was reached which means the coef_ did not converge
 "the coef_ did not converge", ConvergenceWarning)
```

虽然写法看起来略有不同，但其实都是一个含义，这是在提醒我们：参数没有收敛，请增大max_iter中输入的数字。但我们不一定听sklearn的。max_iter很大，意味着步长小，模型运行得会更加缓慢。虽然我们在梯度下降中追求的是损失函数的最小值，但这也可能意味着我们的模型会过拟合（在训练集上表现得太好，在测试集上却不一定），因此，如果在max_iter报红条的情况下，模型的训练和预测效果都已经不错了，那我们就不再增大max_iter中的数目了，毕竟一切都以模型的预测效果为基准——只要最终的预测效果好，运行又快，那就一切都好，无所谓是否报红色警告了。

2.4 二元回归与多元回归：重要参数solver & multi_class

之前我们对逻辑回归的讨论，都是针对二分类的逻辑回归展开，其实sklearn提供了多种可以使用逻辑回归处理多分类问题的选项。比如说，我们可以把某种分类类型都看作1，其余的分类类型都为0值，和“数据预处理”中的二值化的思维类似，这种方法被称为“一对多”(One-vs-rest)，简称OvR，在sklearn中表示为“ovr”。又或者，我们可以把好几个分类类型划为1，剩下的几个分类类型划为0值，这是一种“多对多”(Many-vs-Many)的方法，简称MvM，在sklearn中表示为“Multinomial”。每种方式都配合L1或L2正则项来使用。

在sklearn中，我们使用参数multi_class来告诉模型，我们的预测标签是什么样的类型。

multi_class

输入“ovr”，“multinomial”，“auto”来告知模型，我们要处理的分类问题的类型。默认是“ovr”。

‘ovr’：表示分类问题是二分类，或让模型使用“一对多”的形式来处理多分类问题。

‘multinomial’：表示处理多分类问题，这种输入在参数solver是‘liblinear’时不可用。

‘auto’：表示会根据数据的分类情况和其他参数来确定模型要处理的分类问题的类型。比如说，如果数据是二分类，或者solver的取值为“liblinear”，“auto”会默认选择“ovr”。反之，则会选择“multinomial”。

注意：默认值将在0.22版本中从“ovr”更改为“auto”。

我们之前提到的梯度下降法，只是求解逻辑回归参数 θ 的一种方法，并且我们只讲解了求解二分类变量的参数时的各种原理。sklearn为我们提供了多种选择，让我们可以使用不同的求解器来计算逻辑回归。求解器的选择，由参数"solver"控制，共有五种选择。其中"liblinear"是二分类专用，也是现在的默认求解器。

求解器	'liblinear'	'lbfgs'	'newton-cg'	'sag'	'saga'
求解器对应的求解方式	坐标下降法	拟牛顿法的一种，利用损失函数二阶导数矩阵(海森矩阵)来迭代优化损失函数	牛顿法的一种，利用损失函数二阶导数矩阵(海森矩阵)来迭代优化损失函数	随机平均梯度下降，与普通梯度下降法的区别是每次迭代仅仅用一部分的样本来计算梯度	随机平均梯度下降的进化，稀疏多项逻辑回归的首选
支持的惩罚项	L1, L2	L2	L2	L2	L1, L2
支持的回归类型					
Multinomial(MvM)	否	是	是	是	是
OvR	是	是	是	是	是
二分类	是	是	是	是	是
求解器的效果					
惩罚截距 (不要惩罚截距比较好)	是	否	否	否	否
在大型数据集上更快	否	否	否	是	是
对未标准化的数据集很有用	是	是	是	否	否

来看看鸢尾花数据集上，multinomial和ovr的区别怎么样：

```
from sklearn.datasets import load_iris
iris = load_iris()

for multi_class in ('multinomial', 'ovr'):
    clf = LogisticRegression(solver='sag', max_iter=100, random_state=42,
                             multi_class=multi_class).fit(iris.data, iris.target)

#打印两种multi_class模式下的训练分数
##%的用法，用%来代替打印的字符串中，想由变量替换的部分。%.3f表示，保留三位小数的浮点数。%s表示，字符串。
##字符串后的%后使用元祖来容纳变量，字符串中有几个%，元祖中就需要有几个变量

    print("training score : %.3f (%s)" % (clf.score(iris.data, iris.target),
    multi_class))
```

2.5 样本不平衡与参数class_weight

样本不平衡是指在一组数据集中，标签的一类天生占有很大的比例，或误分类的代价很高，即我们想要捕捉出某种特定的分类的时候的状况。

什么情况下误分类的代价很高？例如，我们现在要对潜在犯罪者和普通人进行分类，如果没有能够识别出潜在犯罪者，那么这些人就可能去危害社会，造成犯罪，识别失败的代价会非常高，但如果，我们将普通人错误地识别成了潜在犯罪者，代价却相对较小。所以我们宁愿将普通人分类为潜在犯罪者后再人工甄别，但是却不愿将潜在犯罪者分类为普通人，有种“宁愿错杀不能放过”的感觉。

再比如说，在银行要判断“一个新客户是否会违约”，通常不违约的人vs违约的人会是99: 1的比例，真正违约的人其实是非常少的。这种分类状况下，即便模型什么也不做，全把所有人都当成不会违约的人，正确率也能有99%，这使得模型评估指标变得毫无意义，根本无法达到我们的“要识别出会违约的人”的建模目的。

因此我们要使用参数class_weight对样本标签进行一定的均衡，给少量的标签更多的权重，让模型更偏向少数类，向捕获少数类的方向建模。该参数默认None，此模式表示自动给与数据集中的所有标签相同的权重，即自动1：1。当误分类的代价很高的时候，我们使用“balanced”模式，我们只是希望对标签进行均衡的时候，什么都不填就可以解决样本不均衡问题。

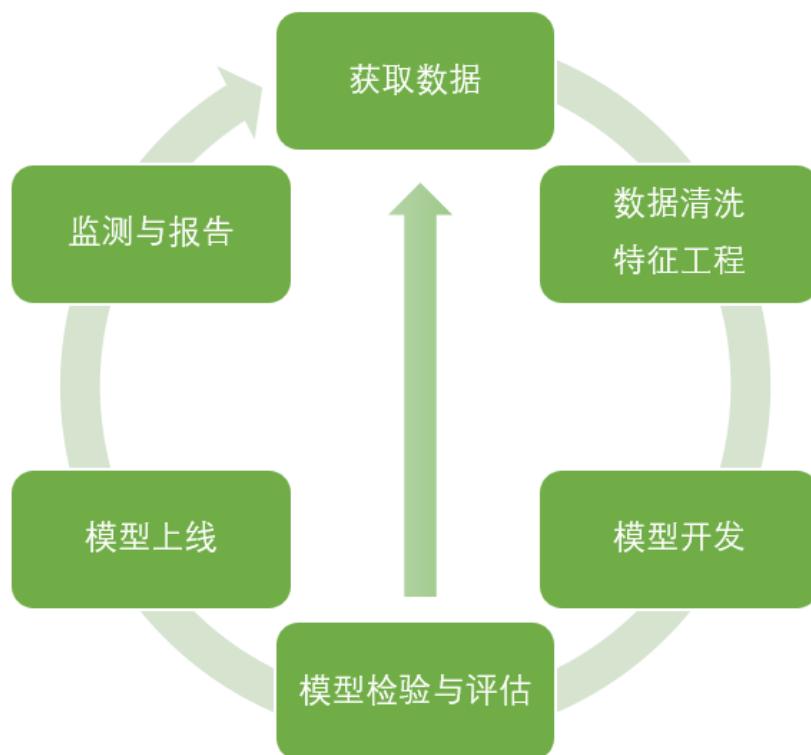
但是，sklearn当中的参数class_weight变幻莫测，大家用模型跑一跑就会发现，我们很难去找出这个参数引导的模型趋势，或者画出学习曲线来评估参数的效果，因此可以说是非常难用。我们有着处理样本不均衡的各种方法，其中主流的是采样法，是通过重复样本的方式来平衡标签，可以进行上采样（增加少数类的样本），比如SMOTE，或者下采样（减少多数类的样本）。对于逻辑回归来说，上采样是最好的办法。在案例中，会给大家详细来讲如何在逻辑回归中使用上采样。

3 案例：用逻辑回归制作评分卡

在银行借贷场景中，评分卡是一种以分数形式来衡量一个客户的信用风险大小的手段，它衡量向别人借钱的人（受信人，需要融资的公司）不能如期履行合同中的还本付息责任，并让借钱给别人的人（授信人，银行等金融机构）造成经济损失的可能性。一般来说，评分卡打出的分数越高，客户的信用越好，风险越小。

这些“借钱的人”，可能是个人，有可能是有需求的公司和企业。对于企业来说，我们按照融资主体的融资用途，分别使用企业融资模型，现金流融资模型，项目融资模型等模型。而对于个人来说，我们有“四张卡”来评判个人的信用程度：A卡，B卡，C卡和F卡。而众人常说的“评分卡”其实是指A卡，又称为申请者评级模型，主要应用于相关融资类业务中**新用户的主体评级**，即判断金融机构是否应该借钱给一个新用户，如果这个人的风险太高，我们可以拒绝贷款。

一个完整的模型开发，需要有以下流程：



今天我们以个人消费类贷款数据，来为大家简单介绍A卡的建模和制作流程，由于时间有限，我们的核心会在“数据清洗”和“模型开发”上。模型检验与评估也非常重要，但是在今天的课中，内容已经太多，我们就不再赘述了。

3.1 导库，获取数据

```
%matplotlib inline
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression as LR
```

#其实日常在导库的时候，并不是一次性能够知道我们要用的所有库的。通常都是在建模过程中逐渐导入需要的库。

在银行系统中，这个数据通常使来自于其他部门的同事的收集，因此千万别忘记抓住给你数据的人，问问她/他各个项都是什么含义。通常来说，当特征非常多的时候（比如几百个），都会有一个附带的excel或pdf文档给到你，备注了各个特征都是什么含义。这种情况下其实要一个个去看还是非常困难，所以如果特征很多，建议先做降维，具体参考“2.2.2 逻辑回归中的特征工程”。

```
data = pd.read_csv(r"C:\work\learnbetter\micro-class\week 5 logit regression\ranking
card\card\data\rankingcard.csv", index_col=0)
```

3.2 探索数据与数据预处理

在这一步我们要样本总体的大概情况，比如查看缺失值，量纲是否统一，是否需要做哑变量等等。其实数据的探索和数据的预处理并不是完全分开的，并不一定非要先做哪一个，因此这个顺序只是供大家参考。

```
#观察数据类型
data.head()
```

```
#观察数据结构
data.shape()
data.info()
```

特征/标签	含义
SeriousDlqin2yrs	出现 90 天或更长时间的逾期行为（即定义好坏客户）
RevolvingUtilizationOfUnsecuredLines	贷款以及信用卡可用额度与总额度比例
age	借款人借款年龄
NumberOfTime30-59DaysPastDueNotWorse	过去两年内出现35-59天逾期但是没有发展得更坏的次数
DebtRatio	每月偿还债务，赡养费，生活费用除以月总收入
MonthlyIncome	月收入
NumberOfOpenCreditLinesAndLoans	开放式贷款和信贷数量
NumberOfTimes90DaysLate	过去两年内出现90天逾期或更坏的次数
NumberRealEstateLoansOrLines	抵押贷款和房地产贷款数量，包括房屋净值信贷额度
NumberOfTime60-89DaysPastDueNotWorse	过去两年内出现60-89天逾期但是没有发展得更坏的次数
NumberOfDependents	家庭中不包括自身的家属人数（配偶，子女等）

3.2.1 去除重复值

现实数据，尤其是银行业数据，可能会存在的一个问题就是样本重复，即有超过一行的样本所显示的所有特征都一样。有时候可能人为输入重复，有时候可能是系统录入重复，总而言之我们必须对数据进行去重处理。可能有人说，难道不可能出现说两个样本的特征就是一模一样，但他们是两个样本吗？比如，两个人，一模一样的名字，年龄，性别，学历，工资……当特征量很少的时候，这的确是有可能的，但一些指标，比如说家属人数，月收入，

已借有的房地产贷款数量等等，几乎不可能都出现一样。尤其是银行业数据经常是几百个特征，所有特征都一样的可能性是微乎其微的。即便真的出现了如此极端的情况，我们也可以当作是少量信息损失，将这条记录当作重复值除去。

```
#去除重复值
data.drop_duplicates(inplace=True)

data.info()

#删除之后千万不要忘记，恢复索引
data.index = range(data.shape[0])

data.info()
```

3.2.2 填补缺失值

```
#探索缺失值
data.info()
data.isnull().sum()/data.shape[0]
#data.isnull().mean()
```

第二个要面临的问题，就是缺失值。在这里我们需要填补的特征是“收入”和“家属人数”。“家属人数”缺失很少，仅缺失了大约2.5%，可以考虑直接删除，或者使用均值来填补。“收入”缺失了几乎20%，并且我们知道，“收入”必然是一个对信用评分来说很重要的因素，因此这个特征必须要进行填补。在这里，我们使用均值填补“家属人数”。

```
data["NumberOfDependents"].fillna(int(data["NumberOfDependents"].mean()), inplace=True)

#如果你选择的是删除那些缺失了2.5%的特征，千万记得恢复索引哟~

data.info()
data.isnull().sum()/data.shape[0]
```

那字段“收入”怎么办呢？对于银行数据来说，我们甚至可以有这样的推断：一个来借钱的人应该是会知道，“高收入”或者“稳定收入”于他/她自己而言会是申请贷款过程中的一个助力，因此如果收入稳定良好的人，肯定会倾向于写上自己的收入情况，那么这些“收入”栏缺失的人，更可能是收入状况不稳定或收入比较低的人。基于这种判断，我们可以用比如说，四分位数来填补缺失值，把所有收入为空的客户都当成是低收入人群。当然了，也有可能这些缺失是银行数据收集过程中的失误，我们并无法判断为什么收入栏会有缺失，所以我们的推断也有可能是不正确的。具体采用什么样的手段填补缺失值，要和业务人员去沟通，观察缺失值是如何产生的。在这里，我们使用随机森林填补“收入”。

还记得我们用随机森林填补缺失值的案例么？随机森林利用“既然我可以使用A, B, C去预测Z, 那我也可以使用A, C, Z去预测B”的思想来填补缺失值。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_train，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。更具体地，大家可以回到随机森林地课中去复习。

之前我们所做的随机森林填补缺失值的案例中，我们面临整个数据集中多个特征都有缺失的情况，因此要先对特征排序，遍历所有特征来进行填补。这次我们只需要填补“收入”一个特征，就无需循环那么麻烦了，可以直接对这一列进行填补。我们来写一个能够填补任何列的函数：

```
def fill_missing_rf(x,y,to_fill):
    """
    使用随机森林填补一个特征的缺失值的函数
    参数:
    x: 要填补的特征矩阵
    y: 完整的, 没有缺失值的标签
    to_fill: 字符串, 要填补的那一列的名称
    """

    #构建我们的新特征矩阵和新标签
    df = x.copy()
    fill = df.loc[:,to_fill]
    df = pd.concat([df.loc[:,df.columns != to_fill],pd.DataFrame(y)],axis=1)

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

    #用随机森林回归来填补缺失值
    from sklearn.ensemble import RandomForestRegressor as rfr
    rfr = rfr(n_estimators=100)
    rfr = rfr.fit(Xtrain, Ytrain)
    Ypredict = rfr.predict(Xtest)

    return Ypredict
```

接下来，我们来创造函数需要的参数，将参数导入函数，产出结果：

```
X = data.iloc[:,1:]
y = data["SeriousDlqin2yrs"]
X.shape

=====【TIME WARNING: 1 min】=====
y_pred = fill_missing_rf(X,y,"MonthlyIncome")

#确认我们的结果合理之后，我们就可以将数据覆盖了
data.loc[data.loc[:, "MonthlyIncome"].isnull(),"MonthlyIncome"] = y_pred
```

3.2.3 描述性统计处理异常值

现实数据永远都会有一些异常值，首先我们要去把他们捕捉出来，然后观察他们的性质。注意，我们并不是要排除掉所有异常值，相反很多时候，异常值是我们的重点研究对象，比如说，双十一中购买量超高的品牌，或课堂上让很多学生都兴奋的课题，这些是我们要重点研究观察的。

日常处理异常值，我们使用箱线图或者 3σ 法则来找到异常值（千万不要说依赖于眼睛看，我们是数据挖掘工程师，除了业务理解，我们还要有方法）。但在银行数据中，我们希望排除的“异常值”不是一些超高或超低的数字，而是一些不符合常理的数据：比如，收入不能为负数，但是一个超高水平的收入却是合理的，可以存在的。所以在银行业务中，我们往往就使用普通的描述性统计来观察数据的异常与否与数据的分布情况。注意，这种方法只能在特征量有限的情况下进行，如果有几百个特征又无法成功降维或特征选择不管用，那还是用 3σ 比较好。

```
#描述性统计
data.describe([0.01, 0.1, 0.25, .5, .75, .9, .99]).T

#异常值也被我们观察到，年龄的最小值居然有0，这不符合银行的业务需求，即便是儿童账户也要至少8岁，我们可以
#查看一下年龄为0的人有多少
(data["age"] == 0).sum()

#发现只有一个人年龄为0，可以判断这肯定是录入失误造成的，可以当成是缺失值来处理，直接删除掉这个样本
data = data[data["age"] != 0]

....
```

另外，有三个指标看起来很奇怪：

```
"NumberOfTime30-59DaysPastDueNotWorse"
"NumberOfTime60-89DaysPastDueNotWorse"
"NumberOfTimes90DaysLate"
```

这三个指标分别是“过去两年内出现35~59天逾期但是没有发展的更坏的次数”，“过去两年内出现60~89天逾期但是没有发展的更坏的次数”，“过去两年内出现90天逾期的次数”。这三个指标，在99%的分布的时候依然是2，最大值却是98，看起来非常奇怪。一个人在过去两年内逾期35~59天98次，一年6个60天，两年内逾期98次这是怎么算出来的？

我们可以去咨询业务人员，请教他们这个逾期次数是如何计算的。如果这个指标是正常的，那这些两年内逾期了98次的客户，应该都是坏客户。在我们无法询问他们情况下，我们查看一下有多少个样本存在这种异常：

```
.....
data[data.loc[:, "NumberOfTimes90DaysLate"] > 90].count()

#有225个样本存在这样的情况，并且这些样本，我们观察一下，标签并不都是1，他们并不都是坏客户。因此，我们基本可以判断，这些样本是某种异常，应该把它们删除。

data = data[data.loc[:, "NumberOfTimes90DaysLate"] < 90]
#恢复索引
data.index = range(data.shape[0])
data.info()
```

3.2.4 为什么不统一量纲，也不标准化数据分布？

在描述性统计结果中，我们可以观察到数据量纲明显不统一，而且存在一部分极偏的分布，虽然逻辑回归对于数据没有分布要求，但是我们知道如果数据服从正态分布的话梯度下降可以收敛得更快。但在这里，我们不对数据进行标准化处理，也不进行量纲统一，为什么？

无论算法有什么样的规定，无论统计学有什么样的要求，我们的最终目的都是要为业务服务。现在我们要制作评分卡，评分卡是要给业务人员们使用的基于新客户填写的各种信息为客户打分的一张卡片，而为了制作这张卡片，我们需要对我们的数据进行一个“分档”，比如说，年龄20~30岁为一档，年龄30~50岁为一档，月收入1W以上为一档，5000~1W为一档，每档的分数不同。

一旦我们将数据统一量纲，或者标准化了之后，数据大小和范围都会改变，统计结果是漂亮的，但是对于业务人员来说，他们完全无法理解，标准化后的年龄在0.00328~0.00467之间为一档是什么含义。并且，新客户填写的信息，天生就是量纲不统一的，我们的确可以将所有的信息录入之后，统一进行标准化，然后导入算法计算，但是最终落到业务人员手上去判断的时候，他们会完全不理解为什么录入的信息变成了一串统计上很美但实际上根本看不懂的数字。由于业务要求，在制作评分卡的时候，我们要尽量保持数据的原貌，年龄就是8~110的数字，收入就是大于0，最大值可以无限的数字，即便量纲不统一，我们也不对数据进行标准化处理。

3.2.5 样本不均衡问题

```
#探索标签的分布
x = data.iloc[:, 1:]
y = data.iloc[:, 0]

y.value_counts()

n_sample = x.shape[0]

n_1_sample = y.value_counts()[1]
n_0_sample = y.value_counts()[0]

print('样本个数: {}; 1占 {:.2%}; 0占
 {:.2%}'.format(n_sample, n_1_sample/n_sample, n_0_sample/n_sample))
```

可以看出，样本严重不均衡。虽然大家都在努力防范信用风险，但实际违约的人并不多。并且，银行并不会真的一棒子打死所有会违约的人，很多人是会还钱的，只是忘记了还款日，很多人是不愿意欠人钱的，但是当时真的很困难，资金周转不过来，所以发生逾期，但一旦他有了钱，他就会把钱换上。对于银行来说，只要你最后能够把钱还上，我都愿意借钱给你，因为我借给你就有收入（利息）。所以，对于银行来说，真正想要被判别出来的其实是“恶意违约”的人，而这部分人数非常非常少，样本就会不均衡。这一直是银行业建模的一个痛点：我们永远希望捕捉少数类。

之前提到过，逻辑回归中使用最多的是上采样方法来平衡样本。

```
#如果报错，就在prompt安装: pip install imblearn
import imblearn
#imblearn是专门用来处理不平衡数据集的库，在处理样本不均衡问题中性能高过sklearn很多
#imblearn里面也是一个个的类，也需要进行实例化，fit拟合，和sklearn用法相似

from imblearn.over_sampling import SMOTE

sm = SMOTE(random_state=42) #实例化
x, y = sm.fit_sample(x, y)

n_sample_ = x.shape[0]

pd.Series(y).value_counts()

n_1_sample = pd.Series(y).value_counts()[1]
n_0_sample = pd.Series(y).value_counts()[0]

print('样本个数: {}; 1占 {:.2%}; 0占
 {:.2%}'.format(n_sample_, n_1_sample/n_sample_, n_0_sample/n_sample_))
```

如此，我们就实现了样本平衡，样本量也增加了。

3.2.6 分训练集和测试集

```
from sklearn.model_selection import train_test_split
X = pd.DataFrame(X)
y = pd.DataFrame(y)

X_train, X_vali, Y_train, Y_vali = train_test_split(X,y,test_size=0.3,random_state=420)
model_data = pd.concat([Y_train, X_train], axis=1)
model_data.index = range(model_data.shape[0])
model_data.columns = data.columns

vali_data = pd.concat([Y_vali, X_vali], axis=1)
vali_data.index = range(vali_data.shape[0])
vali_data.columns = data.columns

model_data.to_csv(r"C:\work\Learnbetter\micro-class\week 5 logit
regression\model_data.csv")

vali_data.to_csv(r"C:\work\Learnbetter\micro-class\week 5 logit
regression\vali_data.csv")
```

3.3 分箱

前面提到过，我们要制作评分卡，是要给各个特征进行分档，以便业务人员能够根据新客户填写的信息为客户打分。因此在评分卡制作过程中，一个重要的步骤就是分箱。可以说，分箱是评分卡最难，也是最核心的思路，分箱的本质，其实就是离散化连续变量，好让拥有不同属性的人被分成不同的类别（打上不同的分数），其实本质比较类似于聚类。那我们在分箱中要回答几个问题：

- 首先，要分多少个箱子才合适？

最开始我们并不知道，但是既然是将连续型变量离散化，想也知道箱子个数必然不能太多，最好控制在十个以下。而用来制作评分卡，最好能在4~5个为最佳。我们知道，离散化连续变量必然伴随着信息的损失，并且箱子越少，信息损失越大。为了衡量特征上的信息量以及特征对预测函数的贡献，银行业定义了概念Information value(IV)：

$$IV = \sum_{i=1}^N (good\% - bad\%) * WOE_i$$

其中N是这个特征上箱子的个数，i代表每个箱子，good%是这个箱内的优质客户（标签为0的客户）占整个特征中所有优质客户的比例，bad%是这个箱子里的坏客户（就是那些会违约，标签为1的那些客户）占整个特征中所有坏客户的比例，而WOE_i则写作：

$$WOE_i = \ln\left(\frac{good\%}{bad\%}\right)$$

这是我们在银行业中用来衡量违约概率的指标，中文叫做证据权重(weight of Evidence)，本质其实就是优质客户比上坏客户的比例的对数。WOE是对一个箱子来说的，WOE越大，代表了这个箱子里的优质客户越多。而IV是对整个特征来说的，IV代表的意义是我们特征上的信息量以及这个特征对模型的贡献，由下表来控制：

IV	特征对预测函数的贡献
< 0.03	特征几乎不带有效信息，对模型没有贡献，这种特征可以被删除
0.03 ~ 0.09	有效信息很少，对模型的贡献度低
0.1 ~ 0.29	有效信息一般，对模型的贡献度中等
0.3 ~ 0.49	有效信息较多，对模型的贡献度较高
>=0.5	有效信息非常多，对模型的贡献超高并且可疑

可见，IV并非越大越好，我们想要找到IV的大小和箱子个数的平衡点。箱子越多，IV必然越小，因为信息损失会非常多，所以，我们会对特征进行分箱，然后计算每个特征在每个箱子数目下的WOE值，利用IV值的曲线，找出合适的分箱个数。

- 其次，分箱要达成什么样的效果？

我们希望不同属性的人有不同的分数，因此我们希望在同一个箱子内的人的属性是尽量相似的，而不同箱子的人的属性是尽量不同的，即业界常说的“组间差异大，组内差异小”。对于评分卡来说，就是说我们希望一个箱子内的人违约概率是类似的，而不同箱子的人的违约概率差距很大，即WOE差距要大，并且每个箱子中坏客户所占的比重（bad%）也要不同。那我们，可以使用卡方检验来对比两个箱子之间的相似性，如果两个箱子之间卡方检验的P值很大，则说明他们非常相似，那我们就可以将这两个箱子合并为一个箱子。

基于这样的思想，我们总结出我们对一个特征进行分箱的步骤：

- 1) 我们首先把连续型变量分成一组数量较多的分类型变量，比如，将几万个样本分成100组，或50组
- 2) 确保每一组中都要包含两种类别的样本，否则IV值会无法计算
- 3) 我们对相邻的组进行卡方检验，卡方检验的P值很大的组进行合并，直到数据中的组数小于设定的N箱为止
- 4) 我们让一个特征分别分成[2,3,4,...,20]箱，观察每个分箱个数下的IV值如何变化，找出最适合的分箱个数
- 5) 分箱完毕后，我们计算每个箱的WOE值，bad%，观察分箱效果

这些步骤都完成后，我们可以对各个特征都进行分箱，然后观察每个特征的IV值，以此来挑选特征。

接下来，我们就以"age"为例子，来看看分箱如何完成。注意，分箱代码的版权属于Hsiaofei Tsien，我已获得授权在这门课中使用和讲解他的代码。

3.3.1 等频分箱

```
#按照等频对需要分箱的列进行分箱
```

```
model_data["qcut"] = pd.qcut(model_data["age"], retbins=True, q=20)
```

```
"""
```

pd.qcut，基于分位数的分箱函数，本质是将连续型变量离散化

只能处理一维数据。返回箱子的上限和下限

参数q：要分箱的个数

参数retbins=True来要求同时返回结构为索引为样本索引，元素为分到的箱子的series

现在返回两个值：每个样本属于哪个箱子，以及所有箱子的上限和下限

```
"""
```

```
#在这里时让model_data新添加一列叫做“分箱”，这一列其实就是每个样本所对应的箱子
model_data["qcut"]

#所有箱子的上限和下限
updown

# 统计每个分箱中0和1的数量
# 这里使用了数据透视表的功能groupby
coount_y0 = model_data[model_data["SeriousDlqin2yrs"] == 0].groupby(by="qcut").count()
["SeriousDlqin2yrs"]
coount_y1 = model_data[model_data["SeriousDlqin2yrs"] == 1].groupby(by="qcut").count()
["SeriousDlqin2yrs"]

#num_bins值分别为每个区间的上界，下界，0出现的次数，1出现的次数
num_bins = [*zip(updown, updown[1:], coount_y0, coount_y1)]
```

#注意zip会按照最短列来进行结合
num_bins

3.3.2【选学】确保每个箱中都有0和1

```
for i in range(20):
    #如果第一个组没有包含正样本或负样本，向后合并
    if 0 in num_bins[0][2:]:
        num_bins[0:2] = [((
            num_bins[0][0],
            num_bins[1][1],
            num_bins[0][2]+num_bins[1][2],
            num_bins[0][3]+num_bins[1][3]))]
        continue
```

.....

合并了之后，第一行的组是否一定有两种样本了呢？不一定

如果原本的第一组和第二组都没有包含正样本，或者都没有包含负样本，那即便合并之后，第一行的组也还是没有包含两种样本

所以我们在每次合并完毕之后，还需要再检查，第一组是否已经包含了两种样本

这里使用continue跳出了本次循环，开始下一次循环，所以回到了最开始的for i in range(20)，让i+1
这就跳过了下面的代码，又从头开始检查，第一组是否包含了两种样本

如果第一组中依然没有包含两种样本，则if通过，继续合并，每合并一次就会循环检查一次，最多合并20次

如果第一组中已经包含两种样本，则if不通过，就开始执行下面的代码

.....

```
#已经确认第一组中肯定包含两种样本了，如果其他组没有包含两种样本，就向前合并
#此时的num_bins已经被上面的代码处理过，可能被合并过，也可能没有被合并
#但无论如何，我们要在num_bins中遍历，所以写成in range(len(num_bins))
for i in range(len(num_bins)):
    if 0 in num_bins[i][2:]:
        num_bins[i-1:i+1] = [((
            num_bins[i-1][0],
            num_bins[i][1],
            num_bins[i-1][2]+num_bins[i][2],
```

```

        num_bins[i-1][3]+num_bins[i][3])
    break
#如果对第一组和对后面所有组的判断中，都没有进入if去合并，则提前结束所有的循环
else:
    break

"""

这个break，只有在if被满足的条件下才会被触发
也就是说，只有发生了合并，才会打断for i in range(len(num_bins))这个循环
为什么要打断这个循环？因为我们是在range(len(num_bins))中遍历
但合并发生后，len(num_bins)发生了改变，但循环却不会重新开始
举个例子，本来num_bins是5组，for i in range(len(num_bins))在第一次运行的时候就等于for i in
range(5)
range中输入的变量会被转换为数字，不会跟着num_bins的变化而变化，所以i会永远在[0,1,2,3,4]中遍历
进行合并后，num_bins变成了4组，已经不存在=4的索引了，但i却依然会取到4，循环就会报错
因此在这里，一旦if被触发，即一旦合并发生，我们就让循环被破坏，使用break跳出当前循环
循环就会回到最开始的for i in range(20)中
此时判断第一组是否有两种标签的代码不会被触发，但for i in range(len(num_bins))却会被重新运行
这样就更新了i的取值，循环就不会报错了
"""

```

3.3.3 定义WOE和IV函数

```

#计算WOE和BAD RATE
#BAD RATE与bad%不是一个东西
#BAD RATE是一个箱中，坏的样本所占的比例 (bad/total)
#而bad%是一个箱中的坏样本占整个特征中的坏样本的比例

def get_woe(num_bins):
    # 通过 num_bins 数据计算 woe
    columns = ["min", "max", "count_0", "count_1"]
    df = pd.DataFrame(num_bins, columns=columns)

    df["total"] = df.count_0 + df.count_1
    df["percentage"] = df.total / df.total.sum()
    df["bad_rate"] = df.count_1 / df.total
    df["good%"] = df.count_0 / df.count_0.sum()
    df["bad%"] = df.count_1 / df.count_1.sum()
    df["woe"] = np.log(df["good%"] / df["bad%"])
    return df

#计算IV值
def get_iv(df):
    rate = df["good%"] - df["bad%"]
    iv = np.sum(rate * df.woe)
    return iv

```

3.3.4 卡方检验，合并箱体，画出IV曲线

```

num_bins_ = num_bins_.copy()

import matplotlib.pyplot as plt
import scipy

IV = []
axisx = []

while len(num_bins_) > 2:
    pvs = []
    # 获取 num_bins_两两之间的卡方检验的置信度 (或卡方值)
    for i in range(len(num_bins_)-1):
        x1 = num_bins_[i][2:]
        x2 = num_bins_[i+1][2:]
        # 0 返回 chi2 值, 1 返回 p 值。
        pv = scipy.stats.chi2_contingency([x1,x2])[1]
        # chi2 = scipy.stats.chi2_contingency([x1,x2])[0]
        pvs.append(pv)

    # 通过 p 值进行处理。合并 p 值最大的两组
    i = pvs.index(max(pvs))
    num_bins_[i:i+2] = [(
        num_bins_[i][0],
        num_bins_[i+1][1],
        num_bins_[i][2]+num_bins_[i+1][2],
        num_bins_[i][3]+num_bins_[i+1][3])]

    bins_df = get_woe(num_bins_)
    axisx.append(len(num_bins_))
    IV.append(get_iv(bins_df))

plt.figure()
plt.plot(axisx,IV)
plt.xticks(axisx)
plt.xlabel("number of box")
plt.ylabel("IV")
plt.show()

```

3.3.5 用最佳分箱个数分箱，并验证分箱结果

将合并箱体的部分定义为函数，并实现分箱：

```

def get_bin(num_bins_,n):
    while len(num_bins_) > n:
        pvs = []
        for i in range(len(num_bins_)-1):
            x1 = num_bins_[i][2:]
            x2 = num_bins_[i+1][2:]

```

```

pv = scipy.stats.chi2_contingency([x1,x2])[1]
# chi2 = scipy.stats.chi2_contingency([x1,x2])[0]
pvs.append(pv)

i = pvs.index(max(pvs))
num_bins_[i:i+2] = [( 
    num_bins_[i][0],
    num_bins_[i+1][1],
    num_bins_[i][2]+num_bins_[i+1][2],
    num_bins_[i][3]+num_bins_[i+1][3])]

return num_bins_

afterbins = get_bin(num_bins,4)

afterbins

bins_df = get_woe(num_bins)

bins_df

```

3.3.6 将选取最佳分箱个数的过程包装为函数

```

def graphforbestbin(DF, X, Y, n=5,q=20,graph=True):
    """
    自动最优分箱函数，基于卡方检验的分箱

    参数：
    DF：需要输入的数据
    X：需要分箱的列名
    Y：分箱数据对应的标签 Y 列名
    n：保留分箱个数
    q：初始分箱的个数
    graph：是否要画出IV图像

    区间为前开后闭 []
    """

    DF = DF[[X,Y]].copy()

    DF["qcut"],bins = pd.qcut(DF[X], retbins=True, q=q,duplicates="drop")
    coount_y0 = DF.loc[DF[Y]==0].groupby(by="qcut").count()[Y]
    coount_y1 = DF.loc[DF[Y]==1].groupby(by="qcut").count()[Y]
    num_bins = [*zip(bins,bins[1:],coount_y0,coount_y1)]

    for i in range(q):
        if 0 in num_bins[0][2:]:
            num_bins[0:2] = [( 
                num_bins[0][0],
                num_bins[1][1],
                num_bins[0][2]+num_bins[1][2],

```

```

        num_bins[0][3]+num_bins[1][3]])
    continue

    for i in range(len(num_bins)):
        if 0 in num_bins[i][2:]:
            num_bins[i-1:i+1] = [((
                num_bins[i-1][0],
                num_bins[i][1],
                num_bins[i-1][2]+num_bins[i][2],
                num_bins[i-1][3]+num_bins[i][3]))]
            break
    else:
        break

def get_woe(num_bins):
    columns = ["min", "max", "count_0", "count_1"]
    df = pd.DataFrame(num_bins, columns=columns)
    df["total"] = df.count_0 + df.count_1
    df["percentage"] = df.total / df.total.sum()
    df["bad_rate"] = df.count_1 / df.total
    df["good%"] = df.count_0 / df.count_0.sum()
    df["bad%"] = df.count_1 / df.count_1.sum()
    df["woe"] = np.log(df["good%"] / df["bad%"])
    return df

def get_iv(df):
    rate = df["good%"] - df["bad%"]
    iv = np.sum(rate * df.woe)
    return iv

IV = []
axisx = []
while len(num_bins) > n:
    pvs = []
    for i in range(len(num_bins)-1):
        x1 = num_bins[i][2:]
        x2 = num_bins[i+1][2:]
        pv = scipy.stats.chi2_contingency([x1,x2])[1]
        pvs.append(pv)

    i = pvs.index(max(pvs))
    num_bins[i:i+2] = [((
        num_bins[i][0],
        num_bins[i+1][1],
        num_bins[i][2]+num_bins[i+1][2],
        num_bins[i][3]+num_bins[i+1][3]))]

    bins_df = pd.DataFrame(get_woe(num_bins))
    axisx.append(len(num_bins))
    IV.append(get_iv(bins_df))

if graph:
    plt.figure()

```

```

plt.plot(axisx,IV)
plt.xticks(axisx)
plt.xlabel("number of box")
plt.ylabel("IV")
plt.show()
return bins_df

```

3.3.7 对所有特征进行分箱选择

```

model_data.columns

for i in model_data.columns[1:-1]:
    print(i)
    graphforbestbin(model_data,i," SeriousDlqin2yrs",n=2,q=20)

```

我们发现，不是所有的特征都可以使用这个分箱函数，比如说有的特征，像家人数量，就无法分出20组。于是我们将可以分箱的特征放出来单独分组，不能自动分箱的变量自己观察然后手写：

```

auto_col_bins = {"RevolvingUtilizationOfUnsecuredLines":6,
                 "age":5,
                 "DebtRatio":4,
                 "MonthlyIncome":3,
                 "NumberOfOpenCreditLinesAndLoans":5}

#不能使用自动分箱的变量
hand_bins = {"NumberOfTime30-59DaysPastDueNotWorse":[0,1,2,13]
             , "NumberOfTimes90DaysLate": [0,1,2,17]
             , "NumberRealEstateLoansOrLines": [0,1,2,4,54]
             , "NumberOfTime60-89DaysPastDueNotWorse": [0,1,2,8]
             , "NumberOfDependents": [0,1,2,3]}

#保证区间覆盖使用 np.inf 替换最大值，用 -np.inf 替换最小值
hand_bins = {k:[-np.inf,*v[:-1],np.inf] for k,v in hand_bins.items()}

```

接下来对所有特征按照选择的箱体个数和手写的分箱范围进行分箱：

```

bins_of_col = {}

# 生成自动分箱的分箱区间和分箱后的 IV 值

for col in auto_col_bins:
    bins_df = graphforbestbin(model_data,col
                             , " SeriousDlqin2yrs"
                             , n=auto_col_bins[col]
                             # 使用字典的性质来取出每个特征所对应的箱的数量
                             , q=20
                             , graph=False)
    bins_list = sorted(set(bins_df["min"]).union(bins_df["max"]))
    # 保证区间覆盖使用 np.inf 替换最大值 -np.inf 替换最小值
    bins_list[0],bins_list[-1] = -np.inf,np.inf
    bins_of_col[col] = bins_list

```

```
#合并手动分箱数据
bins_of_col.update(hand_bins)

bins_of_col
```

3.4 计算各箱的WOE并映射到数据中

我们现在已经有了我们的箱子，接下来我们要做的是计算各箱的WOE，并且把WOE替换到我们的原始数据model_data中，因为我们将使用WOE覆盖后的数据来建模，我们希望获取的是“各个箱”的分类结果，即评分卡上各个评分项目的分类结果。

```
data = model_data.copy()

#函数pd.cut，可以根据已知的分箱间隔把数据分箱
#参数为 pd.cut(数据, 以列表表示的分箱间隔)
data = data[["age", "SeriousDlqin2yrs"]].copy()

data["cut"] = pd.cut(data["age"], [-np.inf, 48.49986200790144, 58.757170160044694, 64.0,
74.0, np.inf])

data

#将数据按分箱结果聚合，并取出其中的标签值
data.groupby("cut")["SeriousDlqin2yrs"].value_counts()

#使用unstack()来将树状结构变成表状结构
data.groupby("cut")["SeriousDlqin2yrs"].value_counts().unstack()

bins_df = data.groupby("cut")["SeriousDlqin2yrs"].value_counts().unstack()

bins_df["woe"] = np.log((bins_df[0]/bins_df[0].sum())/(bins_df[1]/bins_df[1].sum()))
```

把以上过程包装成函数：

```
def get_woe(df,col,y,bins):
    df = df[[col,y]].copy()
    df["cut"] = pd.cut(df[col],bins)
    bins_df = df.groupby("cut")[y].value_counts().unstack()
    woe = bins_df["woe"] =
    np.log((bins_df[0]/bins_df[0].sum())/(bins_df[1]/bins_df[1].sum()))
    return woe

#将所有特征的WOE存储到字典当中
woeall = []
for col in bins_of_col:
    woeall[col] = get_woe(model_data,col,"SeriousDlqin2yrs",bins_of_col[col])

woeall
```

接下来，把所有WOE映射到原始数据中：

```
#不希望覆盖掉原本的数据，创建一个新的DataFrame，索引和原始数据model_data一模一样
model_woe = pd.DataFrame(index=model_data.index)

#将原数据分箱后，按箱的结果把woe结构用map函数映射到数据中
model_woe["age"] = pd.cut(model_data["age"], bins_of_col["age"]).map(woeall["age"])

#对所有特征操作可以写成：
for col in bins_of_col:
    model_woe[col] = pd.cut(model_data[col], bins_of_col[col]).map(woeall[col])

#将标签补充到数据中
model_woe["SeriousDlqin2yrs"] = model_data["SeriousDlqin2yrs"]

#这就是我们的建模数据了
model_woe.head()
```

3.5 建模与模型验证

终于弄完了我们的训练集，接下来我们要处理测试集，在已经有分箱的情况下，测试集的处理就非常简单了，我们只需要将已经计算好的WOE映射到测试集中去就可以了：

```
#处理测试集

vali_woe = pd.DataFrame(index=vali_data.index)

for col in bins_of_col:
    vali_woe[col] = pd.cut(vali_data[col], bins_of_col[col]).map(woeall[col])
vali_woe["SeriousDlqin2yrs"] = vali_data["SeriousDlqin2yrs"]

vali_X = vali_woe.iloc[:, :-1]
vali_y = vali_woe.iloc[:, -1]
```

接下来，就可以开始顺利建模了：

```
x = model_woe.iloc[:, :-1]
y = model_woe.iloc[:, -1]

from sklearn.linear_model import LogisticRegression as LR

lr = LR().fit(x, y)
lr.score(vali_X, vali_y)
```

返回的结果一般，我们可以试着使用C和max_iter的学习曲线把逻辑回归的效果调上去。

```
c_1 = np.linspace(0.01, 1, 20)
c_2 = np.linspace(0.01, 0.2, 20)
```

```

score = []
for i in c_2:
    lr = LR(solver='liblinear', C=i).fit(x, y)
    score.append(lr.score(vali_X, vali_y))
plt.figure()
plt.plot(c_2, score)
plt.show()

lr.n_iter_

score = []
for i in [1, 2, 3, 4, 5, 6]:
    lr = LR(solver='liblinear', C=0.025, max_iter=i).fit(x, y)
    score.append(lr.score(vali_X, vali_y))
plt.figure()
plt.plot([1, 2, 3, 4, 5, 6], score)
plt.show()

```

尽管从准确率来看，我们的模型效果属于一般，但我们可以来看看ROC曲线上的结果。

```

import scikitplot as skplt

# %%cmd
# pip install scikit-plot

vali_proba_df = pd.DataFrame(lr.predict_proba(vali_X))
skplt.metrics.plot_roc(vali_y, vali_proba_df,
                      plot_micro=False, figsize=(6, 6),
                      plot_macro=False)

```

3.6 制作评分卡

建模完毕，我们使用准确率和ROC曲线验证了模型的预测能力。接下来就是要讲逻辑回归转换为标准评分卡了。评分卡中的分数，由以下公式计算：

$$Score = A - B * \log(odds)$$

其中A与B是常数，A叫做“补偿”，B叫做“刻度”， $\log(odds)$ 代表了一个人违约的可能性。其实逻辑回归的结果取对数几率形式会得到 $\theta^T x$ ，即我们的参数*特征矩阵，所以 $\log(odds)$ 其实就是我们的参数。两个常数可以通过两个假设的分值带入公式求出，这两个假设分别是：

1. 某个特定的违约概率下的预期分值
2. 指定的违约概率翻倍的分数 (PDO)

例如，假设对数几率为 $\frac{1}{60}$ 时设定的特定分数为600， PDO=20，那么对数几率为 $\frac{1}{30}$ 时的分数就是620。带入以上线性表达式，可以得到：

$$600 = A - B * \log\left(\frac{1}{60}\right)$$

$$620 = A - B * \log\left(\frac{1}{30}\right)$$

用numpy很容易求出A和B的值：

```
B = 20/np.log(2)
A = 600 + B*np.log(1/60)

B, A
```

有了A和B，分数就很容易得到了。其中不受评分卡中各特征影响的基础分，就是将截距作为 $\log(odds)$ 带入公式进行计算，而其他各个特征各个分档的分数，也是将系数带入进行计算：

```
base_score = A - B*lr.intercept_
base_score

score_age = woeall["age"] * (-B*lr.coef_[0][0])
score_age
```

我们可以通过循环，将所有特征的评分卡内容全部一次性写往一个本地文件ScoreData.csv：

```
file = "C:/work/learnbetter/micro-class/week 5 logit regression/ScoreData.csv"

#open是用来打开文件的python命令，第一个参数是文件的路径+文件名，如果你的文件是放在根目录下，则你只需要文件名就好
#第二个参数是打开文件后的用途，“w”表示用于写入，通常使用的是“r”，表示打开来阅读
#首先写入基准分数
#之后使用循环，每次生成一组score_age类似的分档和分数，不断写入文件之中

with open(file,"w") as fdata:
    fdata.write("base_score,{}\n".format(base_score))
    for i,col in enumerate(X.columns):
        score = woeall[col] * (-B*lr.coef_[0][i])
        score.name = "Score"
        score.index.name = col
        score.to_csv(file,header=True,mode="a")
```

至此，我们评分卡的内容就全部结束了。由于时间有限，我无法给大家面面俱到这个很难的模型，如果有时间，还会给大家补充更多关于模型验证和评估的内容。其实大家可以发现，真正建模的部分不多，更多是我们如何处理数据，如何利用统计和机器学习的方法将数据调整成我们希望的样子，所以除了算法，更加重要的是我们能够达成数据目的的工程能力。这份代码也还有很多细节可以改进，大家在使用的时候可以多找bug多修正，敢于挑战现有的内容，写出属于自己的分箱函数和评分卡模型。

4 附录:

4.1 逻辑回归的参数列表

penalty	可以输入 "l1" 或 "l2" 来指定使用哪一种正则化方式，不填写默认 "l2"。 注意，若选择 "l1" 正则化，参数 solver 仅能够使用求解方式 "liblinear" 和 "saga"，若使用 "l2" 正则化，参数 solver 中所有的求解方式都可以使用。0.19 版本中更新：l1 正则化与 solver "saga" 一起使用，并且允许 "multinomial" + L1 的组合。
dual	布尔值，默认 False。 使用对偶或原始计算方式。对偶方式只在求解器 "liblinear" 与 l2 正则项连用的情况下有效。如果样本量大于特征的数目，这个参数设置为 False 会更好。
tol	浮点数，默认 1.00E-04。 让迭代停下来的小值，数字越大，迭代越早停下来。
C	C 正则化强度的倒数，必须是一个大于 0 的浮点数，不填写默认 1.0，即默认正则项与损失函数的比值是 1:1。C 越小，损失函数会越小，模型对损失函数的惩罚越重，正则化的效力越强，参数 θ 会逐渐被压缩得越来越小。
fit_intercept	布尔值，默认 True。 指定是否应将常量（比如说，偏差或截距）添加到决策函数中。
intercept_scaling	浮点数，默认 1。 仅在使用求解器 "liblinear" 且 self.fit_intercept 设置为 True 时有用。 在这种情况下，x 变为 [x, self.intercept_scaling]，即具有等于 intercept_scaling 的常数值的“合成”特征会被附加到实例矢量。截距会变为 intercept_scaling * synthetic_feature_weight。 注意：合成特征权重 (synthetic_feature_weight) 与所有其他特征一样会经历 l1 和 l2 正则化。为了减小正则化对合成特征权重（并因此对截距）的影响，必须增加 intercept_scaling。
class_weight	字典，字典的列表，"balanced" 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。 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： [{0:1, 1:1}, {0:1, 1:5}, {0:1, 1:1}, {0:1, 1:1}] 而不是： [{1:1}, {2:5}, {3:1}, {4:1}] 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 n_samples / (n_classes * np.bincount(y))。 "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。 注意：如果指定了 sample_weight，这些权重将通过 fit 接口与 sample_weight 相乘。
random_state	整数，sklearn 中设定好的 RandomState 实例，或 None，默认 None。 当求解器是 "sag" 或 "liblinear" 时才有效 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

solver	<p>字符, 可输入{"newton-cg", "lbfgs", "liblinear", "sag", "saga"}, 默认"liblinear"</p> <p>用于求解使模型最优化的参数的算法, 即最优化问题的算法。</p> <p>对于小数据集, 'liblinear'是一个不错的选择, 而'sag'和'saga'对于大数据集来说更快。</p> <p>对于多分类问题, 只有'newton-cg', 'sag', 'saga'和'lbfgs'能够处理多分类的损失函数, 'liblinear'仅限于"一对多"(ovr)和普通二分类方案。</p> <p>'newton-cg', 'lbfgs'和'sag'只处理L2正则项, 而'liblinear'和'saga'可与L1, L2正则项都连用。</p> <p>请注意, "sag"和"saga"快速收敛仅在量纲大致相同的数据上得到保证。您可以使用sklearn.preprocessing中的缩放功能来预处理数据。</p> <p>注意: 默认值将在0.22中从"liblinear"更改为"lbfgs"。</p>
max_iter	<p>整数, 默认100</p> <p>仅适用于newton-cg, sag和lbfgs求解器。求解器收敛的最大迭代次数。</p>
multi_class	<p>字符, 可输入{"ovr", "multinomial", "auto"}, 默认"ovr"</p> <p>表示模型要处理的分类问题的类型。</p> <p>如果输入 'ovr', 表示分类问题是二分类, 或使用"一对多"的格式来处理多分类问题。</p> <p>如果输入 "multinomial", 最小化的损失函数是拟合在整个概率分布上的多项式损失函数, 即使数据是二分类数据。当参数solver的值是'liblinear'时, 'multinomial'不可用。</p> <p>如果输入 "auto", 则表示会根据数据的分类情况和其他参数来确定模型要处理的分类问题的类型。比如说, 如果数据是二分类, 或者solver的取值为"liblinear", "auto"会默认选择"ovr"。反之, 则会选择 "multinomial"。</p> <p>注意: 默认值将在0.22版本中从"ovr"更改为"auto"。</p>
verbose	<p>整数, 默认0</p> <p>对于liblinear和lbfgs求解器, 将verbose设置为任何正数可以表示需要的拟合详细程度。</p>
warm_start	<p>布尔值, 默认False</p> <p>设置为True时, 使用上一次的拟合结果, 否则, 重新实例化一个模型来进行训练。</p> <p>注意: 从0.17版本开始, warm_start支持lbfgs, newton-cg, sag, saga四种求解器。</p>
n_jobs	<p>整数或None, 默认None</p> <p>在multi_class ='ovr'中平行计算类别时使用的CPU线程数。无论是否指定了"multi_class", 当求解器设置为"liblinear"时, 都会忽略此参数。如果此参数在joblib.parallel_backend上下文中, 就表示-1, 否则表示1。-1表示使用处理器的所有线程来进行计算。</p> <p>更多可以参考: https://scikit-learn.org/stable/glossary.html#term-n-jobs</p>

4.2 逻辑回归的属性列表

属性	结构	含义
coef_	数组, 结构 (1,n_features)或 (n_classes,n_features)	决策函数, 即逻辑回归的预测函数中, 特征对应的系数。 当给定问题是二分类问题时, coef_具有形状(1, n_features)。特别地, 当multi_class = 'multinomial'时, coef_对应于结果1(True)并且-coef_对应于结果0(False)。
intercept_	数组, 结构(1,)或 (n_classes,)	逻辑回归的预测函数中的截距(或者偏差) 如果fit_intercept设置为False, 则这个属性返回0。当给定问题是二分类问题时, intercept_具有形状(1,)。特别是, 当multi_class = 'multinomial'时, intercept_对应于结果1(True), -intercept_对应于结果0(False)。
n_iter_	数组, 结构(n_classes,)或(1,)	所有分类的实际迭代次数。如果是二分类或multinomial的问题, 则只返回1个元素。对于liblinear求解器, 会给出了所有类的最大迭代次数。 注意: 如果SciPy版本 <= 1.0.0, lbfgs求解器的迭代次数可能超过max_iter, 这种状况下, n_iter_最多返回max_iter。

4.3 逻辑回归的接口列表

接口	输入	功能	返回
decision_function	测试集X	预测样本的置信度分数 样本的置信度得分是该样本与超平面的有符号距离	每个(样本, 类别)组合的置信度分数。在二分类的情况下, self.classes_[1]的置信度得分中大于0的部分表示将样本预测为此类。
fit	训练集X	使用特征矩阵X拟合模型	拟合好的模型本身
predict	测试集X	预测所提供的测试集X中样本点的标签	返回模型预测的测试样本的标签
predict_log_proba	测试集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	返回测试集中每个样本点对应的每个标签的对数概率
predict_proba	测试集X	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 对于multi_class问题, 如果multi_class参数设置为 "multinomial", 则使用softmax函数用于查找每个类的预测概率。否则就使用一对多ovr的方法, 即使用逻辑函数计算每个类假设为正的概率, 并在所有类中正则化这些值
score	测试集X, 测试集y	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	返回给定数据和标签的平均准确度, 分数越高越好 在多标签分类中返回子集的精度, 这是一个非常严格度量, 因为我们需要为每个样本正确预测每个标签
set_params	新参数组合	在建立好的模型上, 重新设置此评估器的参数	用新参数组合重新实例化和训练的模型
get_params	不需要输入任何对象	获取此评估器的参数	模型的参数
densify	不需要输入任何对象	将系数矩阵转换为密集矩阵	转换好的密集矩阵
sparsify	不需要输入任何对象	将系数矩阵转换为稀疏矩阵	转换好的稀疏矩阵

菜菜的scikit-learn课堂06



sklearn中的聚类算法K-Means

小伙伴们晚上好~o(—▽—)ブ

我是菜菜，这里是我的sklearn课堂第六期，今晚的直播内容是聚类算法K-Means~

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scikit-learn课堂06

sklearn中的聚类算法K-Means

1 概述

1.1 无监督学习与聚类算法

1.2 sklearn中的聚类算法

2 KMeans

2.1 KMeans是如何工作的

2.2 簇内误差平方和的定义和解惑

2.3 KMeans算法的时间复杂度

3 sklearn.cluster.KMeans

3.1 重要参数n_clusters

3.1.1 先进行一次聚类看看吧

3.1.2 聚类算法的模型评估指标

3.1.2.1 当真实标签已知的时候

3.1.2.2 当真实标签未知的时候：轮廓系数

3.1.2.3 当真实标签未知的时候：Calinski-Harabaz Index

3.1.3 案例：基于轮廓系数来选择n_clusters

3.2 重要参数init & random_state & n_init: 初始质心怎么放好?

3.3 重要参数max_iter & tol: 让迭代停下来

3.4 重要属性与重要接口

3.5 函数cluster.k_means

4 案例：聚类算法用于降维，KMeans的矢量量化应用

5 附录

5.1 KMeans参数列表

5.2 KMeans属性列表

5.3 KMeans接口列表

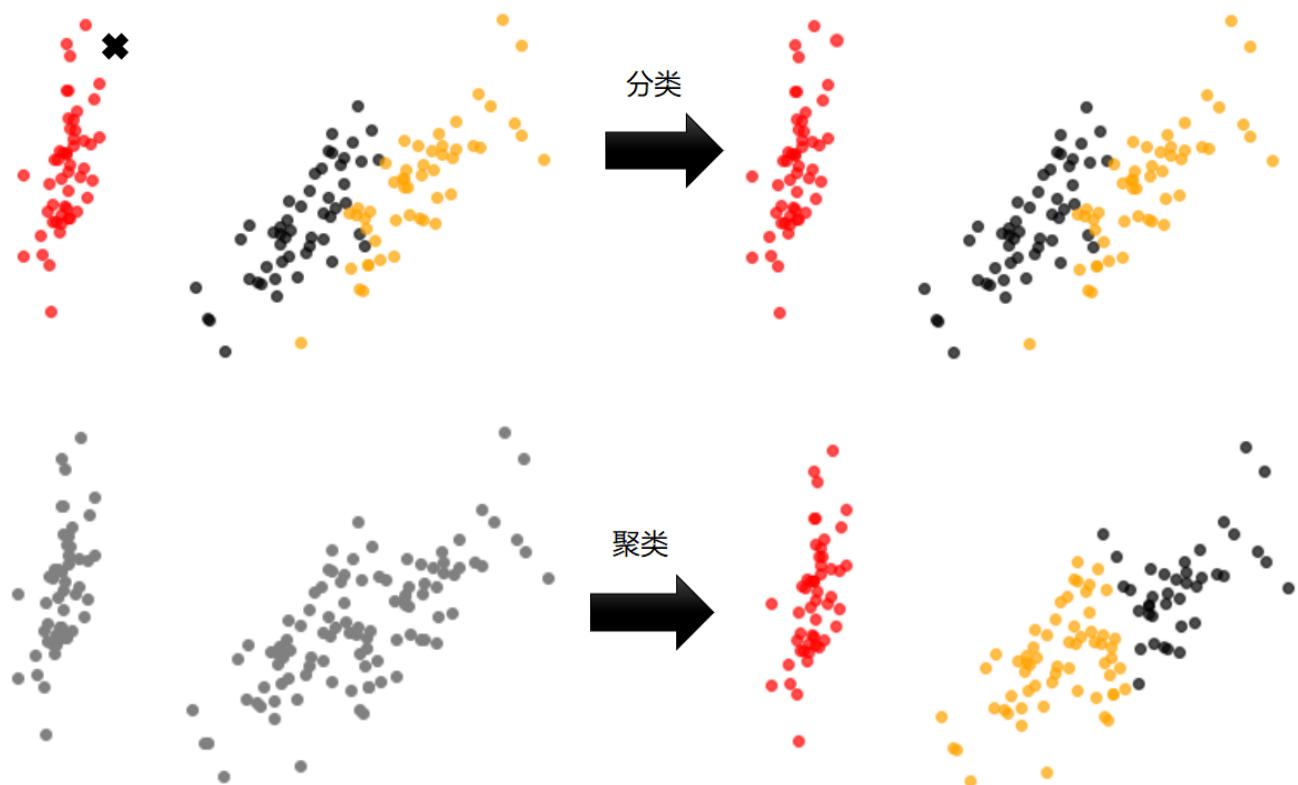
1 概述

1.1 无监督学习与聚类算法

在过去的五周之内，我们学习了决策树，随机森林，逻辑回归，他们虽然有着不同的功能，但却都属于“有监督学习”的一部分，即是说，模型在训练的时候，即需要特征矩阵 X ，也需要真实标签 y 。机器学习当中，还有相当一部分算法属于“无监督学习”，无监督的算法在训练的时候只需要特征矩阵 X ，不需要标签。我们曾经学过的PCA降维算法就是无监督学习中的一种，聚类算法，也是无监督学习的代表算法之一。

聚类算法又叫做“无监督分类”，其目的是将数据划分成有意义或有用的组（或簇）。这种划分可以基于我们的业务需求或建模需求来完成，也可以单纯地帮助我们探索数据的自然结构和分布。比如在商业中，如果我们手头有大量的当前和潜在客户的信息，我们可以使用聚类将客户划分为若干组，以便进一步分析和开展营销活动，最有名的客户价值判断模型RFM，就常常和聚类分析共同使用。再比如，聚类可以用于降维和矢量量化（vector quantization），可以将高维特征压缩到一列当中，常常用于图像、声音、视频等非结构化数据，可以大幅度压缩数据量。

- 聚类vs分类



	聚类	分类
核心	将数据分成多个组 探索每个组的数据是否有联系	从已经分组的数据中去学习 把新数据放到已经分好的组中去
学习类型	无监督，无需标签进行训练	有监督，需要标签进行训练
典型算法	K-Means, DBSCAN, 层次聚类, 光谱聚类	决策树, 贝叶斯, 逻辑回归
算法输出	聚类结果是不确定的 不一定总是能够反映数据的真实分类 同样的聚类，根据不同的业务需求 可能是一个好结果，也可能是一个坏结果	分类结果是确定的 分类的优劣是客观的 不是根据业务或算法需求决定

1.2 sklearn中的聚类算法

聚类算法在sklearn中有两种表现形式，一种是类（和我们目前为止学过的分类算法以及数据预处理方法们都一样），需要实例化，训练并使用接口和属性来调用结果。另一种是函数(function)，只需要输入特征矩阵和超参数，即可返回聚类的结果和各种指标。

类	含义	输入 []内代表可以选择输入, []外代表必须输入
cluster.AffinityPropagation	执行亲和传播数据聚类	[damping, ...]
cluster.AgglomerativeClustering	凝聚聚类	[...]
cluster.Birch	实现Birch聚类算法	[threshold, branching_factor, ...]
cluster.DBSCAN	从矢量数组或距离矩阵执行DBSCAN聚类	[eps, min_samples, metric, ...]
cluster.FeatureAgglomeration	凝聚特征	[n_clusters, ...]
cluster.KMeans	K均值聚类	[n_clusters, init, n_init, ...]
cluster.MiniBatchKMeans	小批量K均值聚类	[n_clusters, init, ...]
cluster.MeanShift	使用平坦核函数的平均移位聚类	[bandwidth, seeds, ...]
cluster.SpectralClustering	光谱聚类，将聚类应用于规范化拉普拉斯的投影	[n_clusters, ...]

函数	含义	输入
cluster.affinity_propagation	执行亲和传播数据聚类	S[, ...]
cluster.dbscan	从矢量数组或距离矩阵执行DBSCAN聚类	X[, eps, min_samples, ...]
cluster.estimate_bandwidth	估计要使用均值平移算法的带宽	X[, quantile, ...]
cluster.k_means	K均值聚类	X, n_clusters[, ...]
cluster.mean_shift	使用平坦核函数的平均移位聚类	X[, bandwidth, seeds, ...]
cluster.spectral_clustering	将聚类应用于规范化拉普拉斯的投影	affinity[, ...]
cluster.ward_tree	光谱聚类，将聚类应用于规范化拉普拉斯的投影	X[, connectivity, ...]

- 输入数据

需要注意的一件重要事情是，该模块中实现的算法可以采用不同类型的矩阵作为输入。所有方法都接受形状[n_samples, n_features]的标准特征矩阵，这些可以从sklearn.feature_extraction模块中的类中获得。对于亲和传播，光谱聚类和DBSCAN，还可以输入形状[n_samples, n_samples]的相似性矩阵，我们可以使用sklearn.metrics.pairwise模块中的函数来获取相似性矩阵。

2 KMeans

2.1 KMeans是如何工作的

作为聚类算法的典型代表，KMeans可以说是最简单的聚类算法没有之一，那它是怎么完成聚类的呢？

关键概念：簇与质心

KMeans算法将一组N个样本的特征矩阵X划分为K个无交集的簇，直观上来看是簇是一组一组聚集在一起的数据，在一个簇中的数据就认为是同一类。簇就是聚类的结果表现。

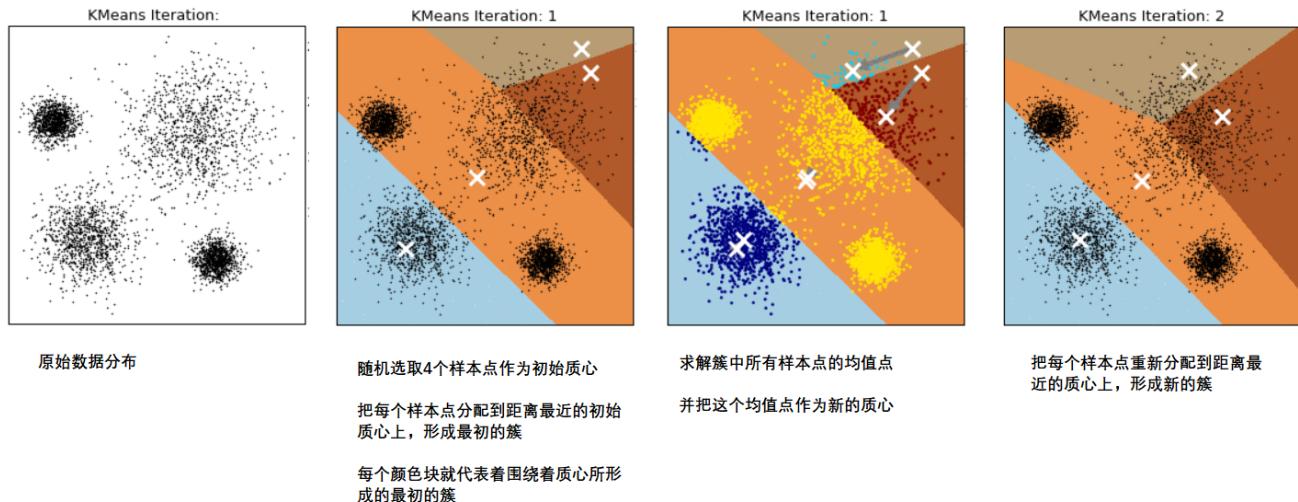
簇中所有数据的均值 μ_j 通常被称为这个簇的“质心”(centroids)。在一个二维平面中，一族数据点的质心的横坐标就是这一簇数据点的横坐标的均值，质心的纵坐标就是这一簇数据点的纵坐标的均值。同理可推广至高维空间。

在KMeans算法中，簇的个数K是一个超参数，需要我们人为输入来确定。KMeans的核心任务就是根据我们设定好的K，找出K个最优的质心，并将离这些质心最近的数据分别分配到这些质心代表的簇中去。具体过程可以总结如下：

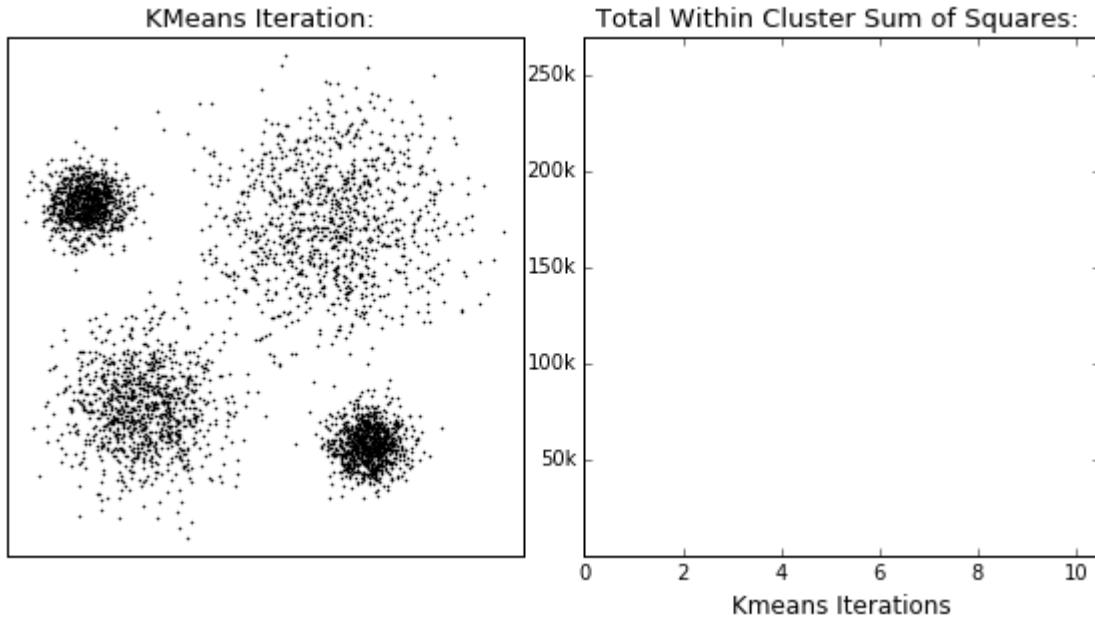
顺序	过程
1	随机抽取K个样本作为最初的质心
2	开始循环：
2.1	将每个样本点分配到离他们最近的质心，生成K个簇
2.2	对于每个簇，计算所有被分到该簇的样本点的平均值作为新的质心
3	当质心的位置不再发生变化，迭代停止，聚类完成

那什么情况下，质心的位置会不再变化呢？当我们找到一个质心，在每次迭代中被分配到这个质心上的样本都是一致的，即每次新生成的簇都是一致的，所有的样本点都不会再从一个簇转移到另一个簇，质心就不会变化了。

这个过程可以在下图来显示，我们规定，将数据分为4簇(K=4)，其中白色X代表质心的位置：



在数据集下多次迭代(iteration), 就会有:



可以看见，第六次迭代之后，基本上质心的位置就不再改变了，生成的簇也变得稳定。此时我们的聚类就完成了，我们可以明显看出，KMeans按照数据的分布，将数据聚集成了我们规定的4类，接下来我们就可以按照我们的业务需求或者算法需求，对这四类数据进行不同的处理。

2.2 簇内误差平方和的定义和解惑

聚类算法聚出的类有什么含义呢？这些类有什么样的性质？我们认为，**被分在同一个簇中的数据是有相似性的，而不同簇中的数据是不同的**，当聚类完毕之后，我们就要分别去研究每个簇中的样本都有什么样的性质，从而根据业务需求制定不同的商业或者科技策略。这个听上去和我们在上周的评分卡案例中讲解的“分箱”概念有些类似，即我们分箱的目的是希望，一个箱内的人有着相似的信用风险，而不同箱的人的信用风险差异巨大，以此来区别不同信用度的人，因此我们追求“组内差异小，组间差异大”。聚类算法也是同样的目的，我们追求“簇内差异小，簇外差异大”。而这个“差异”，由**样本点到其所在簇的质心的距离**来衡量。

对于一个簇来说，所有样本点到质心的距离之和越小，我们就认为这个簇中的样本越相似，簇内差异就越小。而距离的衡量方法有多种，令 x 表示簇中的一个样本点， μ 表示该簇中的质心， n 表示每个样本点中的特征数目， i 表示组成点 x 的每个特征，则该样本点到质心的距离可以由以下距离来度量：

$$\text{欧几里得距离: } d(x, \mu) = \sqrt{\sum_{i=1}^n (x_i - \mu_i)^2}$$

$$\text{曼哈顿距离: } d(x, \mu) = \sum_{i=1}^n (|x_i - \mu_i|)$$

$$\text{余弦距离: } \cos\theta = \frac{\sum_1^n (x_i * \mu_i)}{\sqrt{\sum_1^n (x_i)^2} * \sqrt{\sum_1^n (\mu_i)^2}}$$

如我们采用欧几里得距离，则一个簇中所有样本点到质心的距离的平方和为：

$$\begin{aligned} \text{Cluster Sum of Square (CSS)} &= \sum_{j=0}^m \sum_{i=1}^n (x_i - \mu_i)^2 \\ \text{Total Cluster Sum of Square} &= \sum_{l=1}^k CSS_l \end{aligned}$$

其中， m 为一个簇中样本的个数， j 是每个样本的编号。这个公式被称为**簇内平方和** (cluster Sum of Square) ，又叫做Inertia。而将一个数据集中的所有簇的簇内平方和相加，就得到了整体平方和 (Total Cluster Sum of Square) ，又叫做total inertia。Total Inertia越小，代表着每个簇内样本越相似，聚类的效果就越好。**因此KMeans追求的是，求解能够让Inertia最小化的质心。**实际上，在质心不断变化不断迭代的过程中，总体平方和是越来越小的。我们可以使用数学来证明，当整体平方和最小的时候，质心就不再发生变化了。如此，K-Means的求解过程，就变成了一个最优化问题。

这是我们在这个课程中第二次遇见最优化问题，即需要将某个指标最小化来求解模型中的一部分信息。记得我们在逻辑回归中是怎样的吗？我们在一个固定的方程 $y(x) = \frac{1}{1+e^{\theta^T x}}$ 中最小化损失函数来求解模型的参数向量 θ ，并且基于参数向量 θ 的存在去使用模型。而在KMeans中，我们在一个固定的簇数K下，最小化总体平方和来求解最佳质心，并基于质心的存在去进行聚类。两个过程十分相似，并且，整体距离平方和的最小值其实可以使用梯度下降来求解。因此，有许多博客和教材都这样写道：簇内平方和/整体平方和是KMeans的损失函数。

疑惑：Kmeans有损失函数吗？

记得我们在逻辑回归中曾有这样的结论：损失函数本质是用来衡量模型的拟合效果的，只有有着求解参数需求的算法，才会有损失函数。Kmeans不求解什么参数，它的模型本质也没有在拟合数据，而是在对数据进行一种探索。所以如果你去问大多数数据挖掘工程师，甚至是算法工程师，他们可能会告诉你说，K-Means不存在什么损失函数，Inertia更像是Kmeans的模型评估指标，而非损失函数。

但我们类比过了Kmeans中的Inertia和逻辑回归中的损失函数的功能，我们发现它们确实非常相似。所以，从“求解模型中的某种信息，用于后续模型的使用”这样的功能来看，我们可以认为Inertia是Kmeans中的损失函数，虽然这种说法并不严谨。

对比来看，在决策树中，我们有衡量分类效果的指标准确度accuracy，准确度所对应的损失叫做泛化误差，但我们不能通过最小化泛化误差来求解某个模型中需要的信息，我们只是希望模型的效果上表现出来的泛化误差很小。因此决策树，KNN等算法，是绝对没有损失函数的。

大家可以发现，我们的Inertia是基于欧几里得距离的计算公式得来的。实际上，我们也可以使用其他距离，每个距离都有自己对应的Inertia。在过去的经验中，我们总结出不同距离所对应的质心选择方法和Inertia，在Kmeans中，只要使用了正确的质心和距离组合，无论使用什么样的距离，都可以达到不错的聚类效果：

距离度量	质心	Inertia
欧几里得距离	均值	最小化每个样本点到质心的欧式距离之和
曼哈顿距离	中位数	最小化每个样本点到质心的曼哈顿距离之和
余弦距离	均值	最小化每个样本点到质心的余弦距离之和

而这些组合，都可以由严格的数学证明来推导。在sklearn当中，我们无法选择使用的距离，只能使用欧式距离。因此，我们也无需去担忧这些距离所搭配的质心选择是如何得来的了。

2.3 KMeans算法的时间复杂度

除了模型本身的效果之外，我们还使用另一种角度来度量算法：算法复杂度。算法的复杂度分为时间复杂度和空间复杂度，时间复杂度是指执行算法所需要的计算工作量，常用大O符号表述；而空间复杂度是指执行这个算法所需要的内存空间。如果一个算法的效果很好，但需要的时间复杂度和空间复杂度都很大，那我们将会权衡算法的效果和所需的计算成本之间，比如我们在降维算法和特征工程那两章中，我们尝试了一个很大的数据集下KNN和随机森林所需的运行时间，以此来表明我们降维的目的和决心。

和KNN一样，KMeans算法是一个计算成本很大的算法。在这里，我们介绍KMeans算法的时间和空间复杂度来加深对KMeans的理解。

KMeans算法的平均复杂度是 $O(k*n*T)$ ，其中k是我们的超参数，所需要输入的簇数，n是整个数据集中的样本量，T是所需要的迭代次数（相对的，KNN的平均复杂度是 $O(n)$ ）。在最坏的情况下，KMeans的复杂度可以写作 $O(n^{(k+2)/p})$ ，其中n是整个数据集中的样本量，p是特征总数。这个最高复杂度是由D. Arthur和S. Vassilvitskii在2006年发表的论文“k-means方法有多慢？”中提出的。

在实践中，比起其他聚类算法，k-means算法已经快了，但它一般找到Inertia的局部最小值。这就是为什么多次重启它会很有用。

3 sklearn.cluster.KMeans

```
class sklearn.cluster.KMeans (n_clusters=8, init='k-means++', n_init=10, max_iter=300, tol=0.0001,
precompute_distances='auto', verbose=0, random_state=None, copy_x=True, n_jobs=None, algorithm='auto')
```

3.1 重要参数n_clusters

n_clusters是KMeans中的k，表示着我们告诉模型我们要分几类。这是KMeans当中唯一一个必填的参数，默认为8类，但通常我们的聚类结果会是一个小于8的结果。通常，在开始聚类之前，我们并不知道n_clusters究竟是多少，因此我们要对它进行探索。

3.1.1 先进行一次聚类看看吧

当我们拿到一个数据集，如果可能的话，我们希望能够通过绘图先观察一下这个数据集的数据分布，以此来为我们聚类时输入的n_clusters做一个参考。

首先，我们来自己创建一个数据集。这样的数据集是我们自己创建，所以是有标签的。

```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

#自己创建数据集
x, y = make_blobs(n_samples=500, n_features=2, centers=4, random_state=1)

fig, ax1 = plt.subplots(1)
ax1.scatter(x[:, 0], x[:, 1],
            marker='o' #点的形状
            , s=8 #点的大小
            )
plt.show()

#如果我们想要看见这个点的分布，怎么办？
```

```

color = ["red", "pink", "orange", "gray"]
fig, ax1 = plt.subplots(1)

for i in range(4):
    ax1.scatter(x[y==i, 0], x[y==i, 1]
                ,marker='o' #点的形状
                ,s=8 #点的大小
                ,c=color[i])
)
plt.show()

```

基于这个分布，我们来使用Kmeans进行聚类。首先，我们要猜测一下，这个数据中有几簇？

```

from sklearn.cluster import KMeans

n_clusters = 3

cluster = KMeans(n_clusters=n_clusters, random_state=0).fit(x)

y_pred = cluster.labels_
y_pred

pre = cluster.fit_predict(x)
pre == y_pred

cluster_smallsub = KMeans(n_clusters=n_clusters, random_state=0).fit(x[:200])
y_pred_ = cluster_smallsub.predict(x)
y_pred == y_pred_

centroid = cluster.cluster_centers_
centroid

centroid.shape

inertia = cluster.inertia_
inertia

color = ["red", "pink", "orange", "gray"]
fig, ax1 = plt.subplots(1)

for i in range(n_clusters):
    ax1.scatter(x[y_pred==i, 0], x[y_pred==i, 1]
                ,marker='o'
                ,s=8
                ,c=color[i])
)
ax1.scatter(centroid[:,0], centroid[:,1]
            ,marker="x"
            ,s=15
            ,c="black")
plt.show()

n_clusters = 4

```

```

cluster_ = KMeans(n_clusters=n_clusters, random_state=0).fit(X)
inertia_ = cluster_.inertia_
inertia_

n_clusters = 5
cluster_ = KMeans(n_clusters=n_clusters, random_state=0).fit(X)
inertia_ = cluster_.inertia_
inertia_

n_clusters = 6
cluster_ = KMeans(n_clusters=n_clusters, random_state=0).fit(X)
inertia_ = cluster_.inertia_
inertia_

```

3.1.2 聚类算法的模型评估指标

不同于分类模型和回归，聚类算法的模型评估不是一件简单的事。在分类中，有直接结果（标签）的输出，并且分类的结果有正误之分，所以我们使用预测的准确度，混淆矩阵，ROC曲线等等指标来进行评估，但无论如何评估，都是在“模型找到正确答案”的能力。而回归中，由于要拟合数据，我们有SSE均方误差，有损失函数来衡量模型的拟合程度。但这些衡量指标都不能够使用于聚类。

面试高危问题：如何衡量聚类算法的效果？

聚类模型的结果不是某种标签输出，并且聚类的结果是不确定的，其优劣由业务需求或者算法需求来决定，并且没有永远的正确答案。那我们如何衡量聚类的效果呢？

记得我们说过，KMeans的目标是确保“簇内差异小，簇外差异大”，我们就可以通过衡量簇内差异来衡量聚类的效果。我们刚才说过，Inertia是用距离来衡量簇内差异的指标，因此，我们是否可以使用Inertia来作为聚类的衡量指标呢？Inertia越小模型越好嘛。

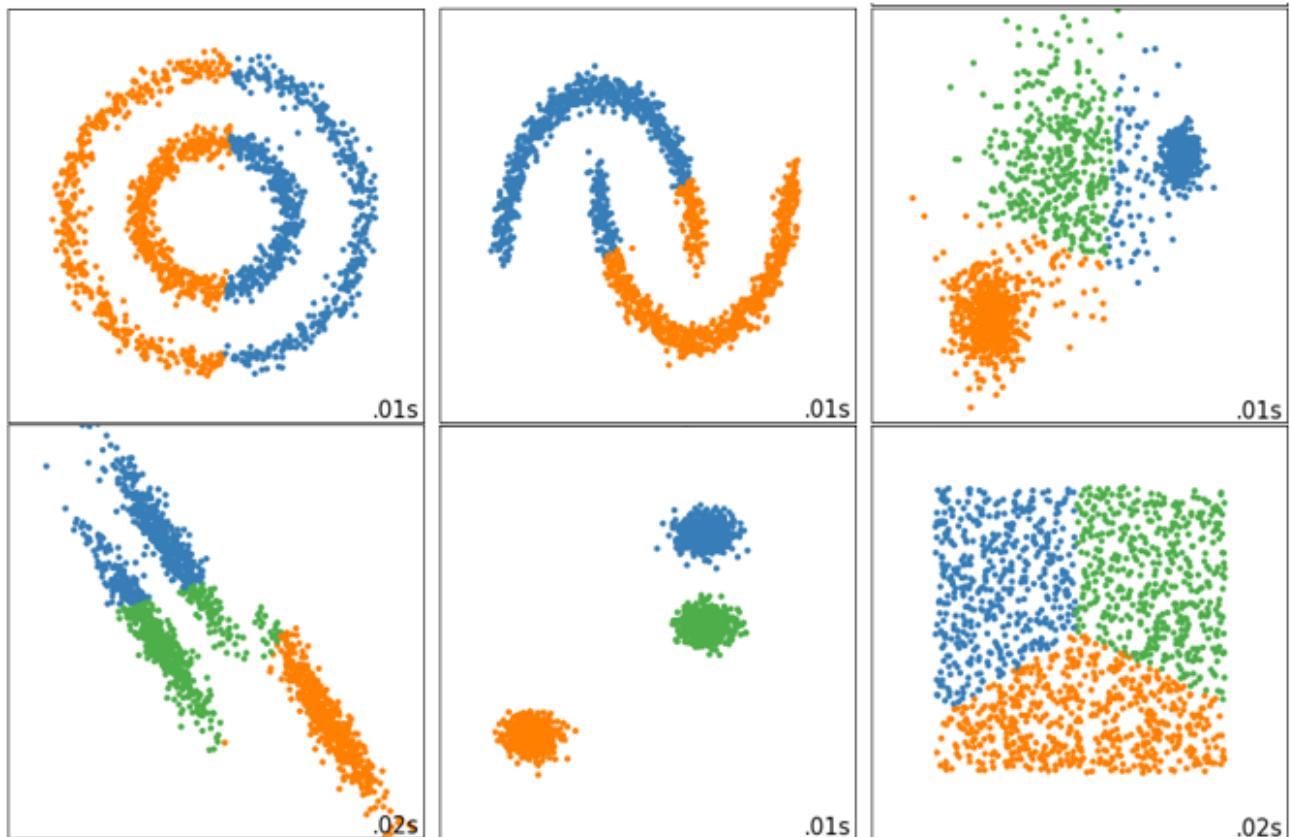
可以，但是这个指标的缺点和极限太大。

首先，它不是有界的。我们只知道，Inertia是越小越好，是0最好，但我们不知道，一个较小的Inertia究竟有没有达到模型的极限，能否继续提高。

第二，它的计算太容易受到特征数目的影响，数据维度很大的时候，Inertia的计算量会陷入维度诅咒之中，计算量会爆炸，不适合用来一次次评估模型。

第三，它会受到超参数K的影响，在我们之前的常识中其实我们已经发现，随着K越大，Inertia注定会越来越小，但这并不代表模型的效果越来越好。

第四，Inertia对数据的分布有假设，它假设数据满足凸分布（即数据在二维平面图像上看起来是一个凸函数的样子），并且它假设数据是各向同性的（isotropic），即是说数据的属性在不同方向上代表着相同的含义。但是现实中的数据往往不是这样。所以使用Inertia作为评估指标，会让聚类算法在一些细长簇，环形簇，或者不规则形状的流形时表现不佳：



那我们可以使用什么指标呢？分两种情况来看。

3.1.2.1 当真实标签已知的时候

虽然我们在聚类中不输入真实标签，但这不代表我们拥有的数据中一定不具有真实标签，或者一定没有任何参考信息。当然，在现实中，拥有真实标签的情况非常少见（几乎是不可能的）。如果拥有真实标签，我们更倾向于使用分类算法。但不排除我们依然可能使用聚类算法的可能性。如果我们有样本真实聚类情况的数据，我们可以对于聚类算法的结果和真实结果来衡量聚类的效果。常用的有以下三种方法：

模型评估指标	说明
互信息分 普通互信息分 <code>metrics.adjusted_mutual_info_score(y_pred, y_true)</code> 调整的互信息分 <code>metrics.mutual_info_score(y_pred, y_true)</code> 标准化互信息分 <code>metrics.normalized_mutual_info_score(y_pred, y_true)</code>	取值范围在(0,1)之中 越接近1, 聚类效果越好 在随机均匀聚类下产生0分
V-measure : 基于条件上分析的一系列直观度量 同质性: 是否每个簇仅包含单个类的样本 <code>metrics.homogeneity_score(y_true, y_pred)</code> 完整性: 是否给定类的所有样本都被分配给同一个簇中 <code>metrics.completeness_score(y_true, y_pred)</code> 同质性和完整性的调和平均, 叫做V-measure <code>metrics.v_measure_score(labels_true, labels_pred)</code> 三者可以被一次性计算出来: <code>metrics.homogeneity_completeness_v_measure(labels_true, labels_pred)</code>	取值范围在(0,1)之中 越接近1, 聚类效果越好 由于分为同质性和完整性两种度量, 可以更仔细地研究, 模型到底哪个任务做得不够好 对样本分布没有假设, 在任何分布上都可以有不错的表现 在随机均匀聚类下不会产生0分
调整兰德系数 <code>metrics.adjusted_rand_score(y_true, y_pred)</code>	取值在(-1,1)之间, 负值象征着簇内的点差异巨大, 甚至相互独立, 正类的兰德系数比较优秀, 越接近1越好 对样本分布没有假设, 在任何分布上都可以有不错的表现, 尤其是在具有"折叠"形状的数据上表现优秀 在随机均匀聚类下产生0分

3.1.2.2 当真实标签未知的时候: 轮廓系数

在99%的情况下, 我们是对没有真实标签的数据进行探索, 也就是对不知道真正答案的数据进行聚类。这样的聚类, 是完全依赖于评价簇内的稠密程度 (簇内差异小) 和簇间的离散程度 (簇外差异大) 来评估聚类的效果。其中轮廓系数是最常用的聚类算法的评价指标。它是对每个样本来定义的, 它能够同时衡量:

- 1) 样本与其自身所在的簇中的其他样本的相似度**a**, 等于样本与同一簇中所有其他点之间的平均距离
- 2) 样本与其他簇中的样本的相似度**b**, 等于样本与下一个最近的簇中的所有点之间的平均距离

根据聚类的要求“簇内差异小, 簇外差异大”, 我们希望b永远大于a, 并且大得越多越好。

单个样本的轮廓系数计算为:

$$s = \frac{b - a}{\max(a, b)}$$

这个公式可以被解析为:

$$s = \begin{cases} 1 - a/b, & \text{if } a < b \\ 0, & \text{if } a = b \\ b/a - 1, & \text{if } a > b \end{cases}$$

很容易理解轮廓系数范围是(-1,1), 其中值越接近1表示样本与自己所在的簇中的样本很相似, 并且与其他簇中的样本不相似, 当样本点与簇外的样本更相似的时候, 轮廓系数就为负。当轮廓系数为0时, 则代表两个簇中的样本相似度一致, 两个簇本应该是一个簇。可以总结为轮廓系数越接近于1越好, 负数则表示聚类效果非常差。

如果一个簇中的大多数样本具有比较高的轮廓系数, 则簇会有较高的总轮廓系数, 则整个数据集的平均轮廓系数越高, 则聚类是合适的。如果许多样本点具有低轮廓系数甚至负值, 则聚类是不合适的, 聚类的超参数K可能设定得太大或者太小。

在sklearn中，我们使用模块metrics中的类silhouette_score来计算轮廓系数，它返回的是一个数据集中，所有样本的轮廓系数的均值。但我们还有同在metrics模块中的silhouette_sample，它的参数与轮廓系数一致，但返回的是数据集中每个样本自己的轮廓系数。

我们来看看轮廓系数在我们自建的数据集上表现如何：

```
from sklearn.metrics import silhouette_score
from sklearn.metrics import silhouette_samples

X
y_pred

silhouette_score(X,y_pred)

silhouette_score(X,cluster_.labels_)

silhouette_samples(X,y_pred)
```

轮廓系数有很多优点，它在有限空间中取值，使得我们对模型的聚类效果有一个“参考”。并且，轮廓系数对数据的分布没有假设，因此在很多数据集上都表现良好。但它在每个簇的分割比较清洗时表现最好。但轮廓系数也有缺陷，它在凸型的类上表现会虚高，比如基于密度进行的聚类，或通过DBSCAN获得的聚类结果，如果使用轮廓系数来衡量，则会表现出比真实聚类效果更高的分数。

3.1.2.3 当真实标签未知的时候：Calinski-Harabaz Index

除了轮廓系数是最常用的，我们还有卡林斯基-哈拉巴斯指数（Calinski-Harabaz Index，简称CHI，也被称为方差比标准），戴维斯-布尔丁指数（Davies-Bouldin）以及权变矩阵（Contingency Matrix）可以使用。

标签未知时的评估指标

卡林斯基-哈拉巴斯指数

```
sklearn.metrics.calinski_harabaz_score (X, y_pred)
```

戴维斯-布尔丁指数

```
sklearn.metrics.davies_bouldin_score (X, y_pred)
```

权变矩阵

```
sklearn.metrics.cluster.contingency_matrix (X, y_pred)
```

在这里我们重点来了解一下卡林斯基-哈拉巴斯指数。Calinski-Harabaz指数越高越好。对于有k个簇的聚类而言，Calinski-Harabaz指数s(k)写作如下公式：

$$s(k) = \frac{Tr(B_k)}{Tr(W_k)} * \frac{N - k}{k - 1}$$

其中N为数据集中的样本量，k为簇的个数（即类别的个数）， B_k 是组间离散矩阵，即不同簇之间的协方差矩阵， W_k 是簇内离散矩阵，即一个簇内数据的协方差矩阵，而tr表示矩阵的迹。在线性代数中，一个 $n \times n$ 矩阵A的主对角线（从左上方至右下方的对角线）上各个元素的总和被称为矩阵A的迹（或迹数），一般记作 $tr(A)$ 。**数据之间的离散程度越高，协方差矩阵的迹就会越大**。组内离散程度低，协方差的迹就会越小， $Tr(W_k)$ 也就越小，同时，组间离散程度大，协方差的迹也会越大， $Tr(B_k)$ 就越大，这正是我们希望的，因此Calinski-harabaz指数越高越好。

```
from sklearn.metrics import calinski_harabaz_score

x
y_pred

calinski_harabaz_score(x, y_pred)
```

虽然calinski-Harabaz指数没有界，在凸型的数据上的聚类也会表现虚高。但是比起轮廓系数，它有一个巨大的优点，就是计算非常快速。之前我们使用过魔法命令%%timeit来计算一个命令的运算时间，今天我们来选择另一种方法：时间戳计算运行时间。

```
from time import time
t0 = time()
calinski_harabaz_score(x, y_pred)
time() - t0

t0 = time()
silhouette_score(x, y_pred)
time() - t0

import datetime
datetime.datetime.fromtimestamp(t0).strftime("%Y-%m-%d %H:%M:%S")
```

可以看得出，calinski-harabaz指数比轮廓系数的计算块了一倍不止。想想看我们使用的数据量，如果是一个以万计的数据，轮廓系数就会大大拖慢我们模型的运行速度了。

3.1.3 案例：基于轮廓系数来选择n_clusters

我们通常会绘制轮廓系数分布图和聚类后的数据分布图来选择我们的最佳n_clusters。

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score

import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np

n_clusters = 4
fig, (ax1, ax2) = plt.subplots(1, 2)
fig.set_size_inches(18, 7)
ax1.set_xlim([-0.1, 1])
ax1.set_ylim([0, x.shape[0] + (n_clusters + 1) * 10])
clusterer = KMeans(n_clusters=n_clusters, random_state=10).fit(x)
cluster_labels = clusterer.labels_

silhouette_avg = silhouette_score(x, cluster_labels)
print("For n_clusters =", n_clusters,
      "The average silhouette score is :", silhouette_avg)
sample_silhouette_values = silhouette_samples(x, cluster_labels)
```

```

y_lower = 10
for i in range(n_clusters):
    ith_cluster_silhouette_values = sample_silhouette_values[cluster_labels == i]
    ith_cluster_silhouette_values.sort()
    size_cluster_i = ith_cluster_silhouette_values.shape[0]
    y_upper = y_lower + size_cluster_i
    color = cm.nipy_spectral(float(i)/n_clusters)
    ax1.fill_betweenx(np.arange(y_lower, y_upper),
                      ith_cluster_silhouette_values,
                      facecolor=color,
                      alpha=0.7
                     )
    ax1.text(-0.05
             , y_lower + 0.5 * size_cluster_i
             , str(i))
    y_lower = y_upper + 10
ax1.set_title("The silhouette plot for the various clusters.")
ax1.set_xlabel("The silhouette coefficient values")
ax1.set_ylabel("Cluster label")

ax1.axvline(x=silhouette_avg, color="red", linestyle="--")
ax1.set_yticks([])
ax1.set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])

colors = cm.nipy_spectral(cluster_labels.astype(float) / n_clusters)

ax2.scatter(X[:, 0], X[:, 1]
            , marker='o'
            , s=8
            , c=colors
           )
centers = clusterer.cluster_centers_
# Draw white circles at cluster centers
ax2.scatter(centers[:, 0], centers[:, 1], marker='x',
            c="red", alpha=1, s=200)

ax2.set_title("The visualization of the clustered data.")
ax2.set_xlabel("Feature space for the 1st feature")
ax2.set_ylabel("Feature space for the 2nd feature")

plt.suptitle(("Silhouette analysis for KMeans clustering on sample data "
              "with n_clusters = %d" % n_clusters),
              fontsize=14, fontweight='bold')
plt.show()

```

将上述过程包装成一个循环，可以得到：

```

from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score

import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np

```

```

for n_clusters in [2,3,4,5,6,7]:
    n_clusters = n_clusters
    fig, (ax1, ax2) = plt.subplots(1, 2)
    fig.set_size_inches(18, 7)
    ax1.set_xlim([-0.1, 1])
    ax1.set_ylim([0, x.shape[0] + (n_clusters + 1) * 10])
    clusterer = KMeans(n_clusters=n_clusters, random_state=10).fit(x)
    cluster_labels = clusterer.labels_
    silhouette_avg = silhouette_score(x, cluster_labels)
    print("For n_clusters =", n_clusters,
          "The average silhouette_score is :", silhouette_avg)
    sample_silhouette_values = silhouette_samples(x, cluster_labels)
    y_lower = 10
    for i in range(n_clusters):
        ith_cluster_silhouette_values = sample_silhouette_values[cluster_labels == i]
        ith_cluster_silhouette_values.sort()
        size_cluster_i = ith_cluster_silhouette_values.shape[0]
        y_upper = y_lower + size_cluster_i
        color = cm.nipy_spectral(float(i)/n_clusters)
        ax1.fill_betweenx(np.arange(y_lower, y_upper),
                          ith_cluster_silhouette_values,
                          facecolor=color,
                          alpha=0.7
                          )
        ax1.text(-0.05
                 , y_lower + 0.5 * size_cluster_i
                 , str(i))
        y_lower = y_upper + 10

    ax1.set_title("The silhouette plot for the various clusters.")
    ax1.set_xlabel("The silhouette coefficient values")
    ax1.set_ylabel("Cluster label")
    ax1.axvline(x=silhouette_avg, color="red", linestyle="--")
    ax1.set_yticks([])
    ax1.set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])

    colors = cm.nipy_spectral(cluster_labels.astype(float) / n_clusters)
    ax2.scatter(x[:, 0], x[:, 1],
                marker='o',
                s=8
                ,c=colors
                )
    centers = clusterer.cluster_centers_
    # Draw white circles at cluster centers
    ax2.scatter(centers[:, 0], centers[:, 1], marker='x',
                c="red", alpha=1, s=200)

    ax2.set_title("The visualization of the clustered data.")
    ax2.set_xlabel("Feature space for the 1st feature")
    ax2.set_ylabel("Feature space for the 2nd feature")

plt.suptitle(("Silhouette analysis for KMeans clustering on sample data "))

```

```

    "with n_clusters = %d" % n_clusters),
    fontsize=14, fontweight='bold')
plt.show()

```

3.2 重要参数init & random_state & n_init：初始质心怎么放好？

在K-Means中有一个重要的环节，就是放置初始质心。如果有足够的时间，K-means一定会收敛，但Inertia可能收敛到局部最小值。是否能够收敛到真正的最小值很大程度上取决于质心的初始化。init就是用来帮助我们决定初始化方式的参数。

初始质心放置的位置不同，聚类的结果很可能也会不一样，一个好的质心选择可以让K-Means避免更多的计算，让算法收敛稳定且更快。在之前讲解初始质心的放置时，我们是使用“随机”的方法在样本点中抽取k个样本作为初始质心，这种方法显然不符合“稳定且更快”的需求。为此，我们可以使用random_state参数来控制每次生成的初始质心都在相同位置，甚至可以画学习曲线来确定最优的random_state是哪个整数。

一个random_state对应一个质心随机初始化的随机数种子。如果不指定随机数种子，则sklearn中的K-means并不会只选择一个随机模式扔出结果，而会在每个随机数种子下运行多次，并使用结果最好的一个随机数种子来作为初始质心。我们可以使用参数n_init来选择，每个随机数种子下运行的次数。这个参数不常用到，默认10次，如果我们希望运行的结果更加精确，那我们可以增加这个参数n_init的值来增加每个随机数种子下运行的次数。

然而这种方法依然是基于随机性的。

为了优化选择初始质心的方法，2007年Arthur, David, and Sergei Vassilvitskii三人发表了论文[“k-means++: The advantages of careful seeding”](#)，他们开发了“k-means ++”初始化方案，使得初始质心（通常）彼此远离，以此来引导出比随机初始化更可靠的结果。在sklearn中，我们使用参数init ='k-means ++'来选择使用k-means ++作为质心初始化的方案。通常来说，我建议保留默认的“k-means++”的方法。

init: 可输入"“k-means++”， "random"或者一个n维数组。这是初始化质心的方法，默认"“k-means++”"。输入"“k-means++”": 一种为K均值聚类选择初始聚类中心的聪明的办法，以加速收敛。如果输入了n维数组，数组的形状应该是(n_clusters, n_features)并给出初始质心。

random_state: 控制每次质心随机初始化的随机数种子

n_init: 整数，默认10，使用不同的质心随机初始化的种子来运行k-means算法的次数。最终结果会是基于Inertia来计算的n_init次连续运行后的最佳输出

```

x
y

plus = KMeans(n_clusters = 10).fit(x)
plus.n_iter_

random = KMeans(n_clusters = 10, init="random", random_state=420).fit(x)
random.n_iter_

```

3.3 重要参数max_iter & tol: 让迭代停下来

在之前描述K-Means的基本流程时我们提到过，当质心不再移动，Kmeans算法就会停下来。但在完全收敛之前，我们也可以使用max_iter，最大迭代次数，或者tol，两次迭代间Inertia下降的量，这两个参数来让迭代提前停下来。有时候，当我们的n_clusters选择不符合数据的自然分布，或者我们为了业务需求，必须要填入与数据的自然分布不合的n_clusters，提前让迭代停下来反而能够提升模型的表现。

max_iter: 整数，默认300，单次运行的k-means算法的最大迭代次数

tol: 浮点数，默认1e-4，两次迭代间Inertia下降的量，如果两次迭代之间Inertia下降的值小于tol所设定的值，迭代就会停下

```
random = KMeans(n_clusters = 10, init="random", max_iter=10, random_state=420).fit(x)
y_pred_max10 = random.labels_
silhouette_score(x,y_pred_max10)

random = KMeans(n_clusters = 10, init="random", max_iter=20, random_state=420).fit(x)
y_pred_max20 = random.labels_
silhouette_score(x,y_pred_max20)
```

3.4 重要属性与重要接口

到这里，所有的重要参数就讲完了。在使用模型的过程中，我也向大家呈现了各种重要的属性与接口，在这一小节来复习一下：

属性	含义
cluster_centers_	收敛到的质心。如果算法在完全收敛之前就已经停下了（受到参数max_iter和tol的控制），所返回的内容将与labels_属性中反应出来的聚类结果不一致。
labels_	每个样本点对应的标签
inertia_	每个样本点到距离他们最近的簇心的均方距离，又叫做“簇内平方和”
n_iter	实际的迭代次数

接口	输入	功能&返回
fit	训练特征矩阵X, [训练用标签, sample_weight]	拟合模型，计算K均值的聚类结果
fit_predict	训练特征矩阵X, [训练用标签, sample_weight]	返回每个样本所对应的簇的索引 计算质心并且为每个样本预测所在的簇的索引，功能相当于先fit再predict
fit_transform	训练特征矩阵X, [训练用标签, sample_weight]	返回新空间中的特征矩阵 进行聚类并且将特征矩阵X转换到簇距离空间当中，功能相当于先fit再transform
get_params	不需要任何输入	获取该类的参数
predict	测试特征矩阵X, [sample_weight]	预测每个测试集X中的样本的所在簇，并返回每个样本所对应的簇的索引 在矢量量化相关的文献中，cluster_centers_被称为代码簿，而predict返回的每个值是代码簿中最接近的代码的索引。
score	测试特征矩阵X, [训练用标签, sample_weight]	返回聚类后的Inertia，即簇内平方和的负数 簇内平方和是Kmeans常用的模型评价指标，簇内平方和越小越好，最佳值为0
set_params	需要新设定的参数	为建立好的类重设参数
transform	任意特征矩阵X	将X转换到簇距离空间中 在新空间中，每个维度（即每个坐标轴）是样本点到集群中心的距离。请注意，即使X是稀疏的，变换返回的数组通常也是密集的。

3.5 函数cluster.k_means

```
sklearn.cluster.k_means (X, n_clusters, sample_weight=None, init='k-means++', precompute_distances='auto',
n_init=10, max_iter=300, verbose=False, tol=0.0001, random_state=None, copy_x=True, n_jobs=None,
algorithm='auto', return_n_iter=False)
```

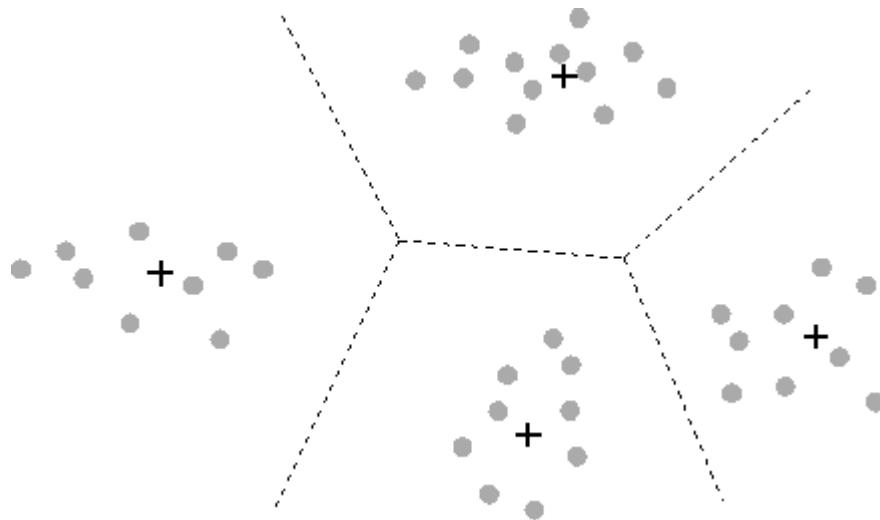
函数k_means的用法其实和类非常相似，不过函数是输入一系列值，而直接返回结果。一次性地，函数k_means会依次返回质心，每个样本对应的簇的标签，inertia以及最佳迭代次数。

```
from sklearn.cluster import k_means  
k_means(X, 4, return_n_iter=True)
```

4 案例：聚类算法用于降维，KMeans的矢量量化应用

K-Means聚类最重要的应用之一是非结构数据（图像，声音）上的矢量量化（VQ）。非结构化数据往往占用比较多的储存空间，文件本身也会比较大，运算非常缓慢，我们希望能够在保证数据质量的前提下，尽量地缩小非结构化数据的大小，或者简化非结构化数据的结构。矢量量化就可以帮助我们实现这个目的。KMeans聚类的矢量量化本质是一种降维运用，但它与我们之前学过的任何一种降维算法的思路都不相同。特征选择的降维是直接选取对模型贡献最大的特征，PCA的降维是聚合信息，而矢量量化的降维是在同等样本量上压缩信息的大小，即不改变特征的数目也不改变样本的数目，只改变在这些特征下的样本上的信息量。

对于图像来说，一张图片上的信息可以被聚类如下表示：



这是一组40个样本的数据，分别含有40组不同的信息(x_1, x_2)。我们将代表所有样本点聚成4类，找出四个质心，我们认为，这些点和他们所属的质心非常相似，因此他们所承载的信息就约等于他们所在的簇的质心所承载的信息。于是，我们可以使用每个样本所在的簇的质心来覆盖原有的样本，有点类似四舍五入的感觉，类似于用1来代替0.9和0.8。这样，40个样本带有的40种取值，就被我们压缩了4组取值，虽然样本量还是40个，但是这40个样本所带的取值其实只有4个，就是分出来的四个簇的质心。

用K-Means聚类中获得的质心来替代原有的数据，可以把数据上的信息量压缩到非常小，但又不损失太多信息。我们接下来就通过一张图图片的矢量量化来看一看K-Means如何实现压缩数据大小，却不损失太多信息量。

1. 导入需要的库

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import pairwise_distances_argmin
from sklearn.datasets import load_sample_image
from sklearn.utils import shuffle
```

2. 导入数据，探索数据

```
china = load_sample_image("china.jpg")
china
china.dtype
```

```

china.shape

china[0][0]

newimage = china.reshape((427 * 640, 3))

import pandas as pd
pd.DataFrame(newimage).drop_duplicates().shape

plt.figure(figsize=(15,15))
plt.imshow(china)

flower = load_sample_image("flower.jpg")
plt.figure(figsize=(15,15))
plt.imshow(flower)

```

图像探索完毕，我们了解了，图像现在有9W多种颜色。我们希望来试试看，能否使用K-Means将颜色压缩到64种，还不严重损耗图像的质量。为此，我们要使用K-Means来将9W种颜色聚类成64类，然后使用64个簇的质心来替代全部的9W种颜色，记得质心有着这样的性质：簇中的点都是离质心最近的样本点。

为了比较，我们还要画出随机压缩到64种颜色的矢量量化图像。我们需要随机选取64个样本点作为随机质心，计算原数据中每个样本到它们的距离来找出离每个样本最近的随机质心，然后用每个样本所对应的随机质心来替换原本的样本。两种状况下，我们观察图像可视化之后的状况，以查看图片信息的损失。

在这之前，我们需要把数据处理成sklearn中的K-Means类能够接受的数据。

3. 决定超参数，数据预处理

```

n_clusters = 64

china = np.array(china, dtype=np.float64) / china.max()
w, h, d = original_shape = tuple(china.shape)
assert d == 3
image_array = np.reshape(china, (w * h, d))

china = np.array(china, dtype=np.float64) / china.max()

w, h, d = original_shape = tuple(china.shape)

w
h
d

assert d == 3

d_ = 5
assert d_ == 3, "一个格子中的特征数目不等于3种"

image_array = np.reshape(china, (w * h, d))
image_array

```

```
image_array.shape  
  
a = np.random.random((2,4))  
  
a  
  
a.reshape((4,2))  
  
np.reshape(a,(4,2))  
  
np.reshape(a,(2,2,2))  
  
np.reshape(a,(3,2))
```

4. 对数据进行K-Means的矢量量化

```
image_array_sample = shuffle(image_array, random_state=0)[:1000]  
kmeans = KMeans(n_clusters=n_clusters, random_state=0).fit(image_array_sample)  
kmeans.cluster_centers_  
  
labels = kmeans.predict(image_array)  
labels.shape  
  
image_kmeans = image_array.copy()  
for i in range(w*h):  
    image_kmeans[i] = kmeans.cluster_centers_[labels[i]]  
  
image_kmeans  
pd.DataFrame(image_kmeans).drop_duplicates().shape  
  
image_kmeans = image_kmeans.reshape(w,h,d)  
image_kmeans.shape
```

5. 对数据进行随机的矢量量化

```
centroid_random = shuffle(image_array, random_state=0)[:n_clusters]  
labels_random = pairwise_distances_argmin(centroid_random,image_array, axis=0)  
  
labels_random.shape  
  
len(set(labels_random))  
  
image_random = image_array.copy()  
for i in range(w*h):  
    image_random[i] = centroid_random[labels_random[i]]  
  
image_random = image_random.reshape(w,h,d)  
image_random.shape
```

6. 将原图，按KMeans矢量量化和随机矢量量化的图像绘制出来

```
plt.figure(figsize=(10,10))
plt.axis('off')
plt.title('Original image (96,615 colors)')
plt.imshow(china)

plt.figure(figsize=(10,10))
plt.axis('off')
plt.title('Quantized image (64 colors, K-Means)')
plt.imshow(image_kmeans)

plt.figure(figsize=(10,10))
plt.axis('off')
plt.title('Quantized image (64 colors, Random)')
plt.imshow(image_random)
plt.show()
```

5 附录

5.1 KMeans参数列表

参数	含义
n_clusters	整数, 可不填, 默认8 要分成的簇数, 以及要生成的质心数
init	可输入 "k-means++", "random"或者一个n维数组 初始化质心的方法, 默认"k-means++" 输入 "k-means++": 一种为K均值聚类选择初始聚类中心的聪明的办法, 以加速收敛。详细信息请参阅 k_init中的注释部分。 如果输入了n维数组, 数组的形状应该是(n_clusters,n_features)并给出初始质心。
n_init	整数, 默认10 使用不同的质心随机初始化的种子来运行k-means算法的次数。最终结果会是基于Inertia来计算的n_init次连续运行后的最佳输出。
max_iter	整数, 默认300 单次运行的k-means算法的最大迭代次数
tol	浮点数, 默认1e-4 两次迭代间Inertia下降的量, 如果两次迭代之间Inertia下降的值小于tol所设定的值, 迭代就会停下
precompute_distances	"auto", True, False, 默认 "auto" 预算算距离 (更快但需要更多内存)。 'auto': 如果n_samples * n_clusters > 1200万, 请不要预先计算距离。这相当于使用双精度来学习, 每个作业大约需要100MB的内存开销。 True: 始终预先计算距离 False: 从不预先计算距离
verbose	整数, 默认0 计算中的详细模式
random_state	整数, RandomState或者None, 默认None 确定质心初始化的随机数生成。使用int可以使随机性具有确定性。
copy_x	布尔值, 可不填, 默认True 在预先计算距离时, 如果先中心化数据, 距离的预算算会更加精确。如果copy_x为True (默认值), 则不修改原始数据, 确保特征矩阵X是C-连续(C-contiguous)的。如果为False, 原始数据被修改, 并在函数返回之前放回, 但是可以通过减去或增加数据均值来引入微小的数值差异, 在这种情况下, False模式无法确保数据是C-连续的, 这可能导致K-Means的计算量显著变慢。
n_jobs	整数或None, 可不填, 默认1 用于计算的作业数。在计算每个n_init时并行运行的作业数。 这个参数允许K-means在多个作业线上并行运行。给这个参数一个正值n_jobs, 表示使用n_jobs条处理器中的线程。值-1表示使用所有可用的处理器, -2表示使用所有可用的处理器-1个处理器, 依此类推。并行化通常以内存为代价加速计算 (在这种情况下, 需要存储多个质心副本, 每个作业一个)。
algorithm	可输入 "auto" , "full" or "elkan", 默认为" auto" K-means算法使用。经典的EM风格算法是“完整的”。通过使用三角不等式, “elkan” 变体更有效, 但目前不支持稀疏数据。“auto” 为密集数据选择 “elkan”, 为稀疏数据选择 “full”。

5.2 KMeans属性列表

属性	含义
<code>cluster_centers_</code>	收敛到的质心。如果算法在完全收敛之前就已经停下了（受到参数 <code>max_iter</code> 和 <code>tol</code> 的控制），所返回的内容将与 <code>labels_</code> 属性中反应出来的聚类结果不一致。
<code>labels_</code>	每个样本点对应的标签
<code>inertia_</code>	每个样本点到距离他们最近的簇心的均方距离，又叫做“簇内平方和”
<code>n_iter</code>	实际的迭代次数

5.3 KMeans接口列表

接口	输入	功能&返回
<code>fit</code>	训练特征矩阵X, [训练用标签, <code>sample_weight</code>]	拟合模型, 计算K均值的聚类结果
<code>fit_predict</code>	训练特征矩阵X, [训练用标签, <code>sample_weight</code>]	返回每个样本所对应的簇的索引 计算质心并且为每个样本预测所在的簇的索引, 功能相当于先 <code>fit</code> 再 <code>predict</code>
<code>fit_transform</code>	训练特征矩阵X, [训练用标签, <code>sample_weight</code>]	返回新空间中的特征矩阵 进行聚类并且将特征矩阵X转换到簇距离空间当中, 功能相当于先 <code>fit</code> 再 <code>transform</code>
<code>get_params</code>	不需要任何输入	获取该类的参数
<code>predict</code>	测试特征矩阵X, [<code>sample_weight</code>]	预测每个测试集X中的样本的所在簇, 并返回每个样本所对应的簇的索引 在矢量量化的相关文献中, <code>cluster_centers_</code> 被称为代码簿, 而 <code>predict</code> 返回的每个值是代码簿中最接近的代码的索引。
<code>score</code>	测试特征矩阵X, [训练用标签, <code>sample_weight</code>]	返回聚类后的 <code>Inertia</code> , 即簇内平方和的负数 簇内平方和是Kmeans常用的模型评价指标, 簇内平方和越小越好, 最佳值为0
<code>set_params</code>	需要新设定的参数	为建立好的类重设参数
<code>transform</code>	任意特征矩阵X	将X转换到簇距离空间中 在新空间中, 每个维度(即每个坐标轴)是样本点到集群中心的距离。请注意, 即使X是稀疏的, 变换返回的数组通常也是密集的。

菜菜的scikit-learn课堂07



sklearn中的支持向量机SVM (上)

小伙伴们晚上好~o(—▽—)ブ

我是菜菜，这里是我的sklearn课堂第七期，今晚的直播内容是支持向量机（上），下周还有下篇哦~

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scikit-learn课堂07

sklearn中的支持向量机SVM（上）

1 概述

- 1.1 支持向量机分类器是如何工作的
- 1.2 支持向量机原理的三层理解
- 1.2 sklearn中的支持向量机

2 sklearn.svm.SVC

- 2.1 线性SVM用于分类的原理
 - 2.1.1 线性SVM的损失函数详解
 - 2.1.2 函数间隔与几何间隔
 - 2.1.3 线性SVM的拉格朗日对偶函数和决策函数
 - 2.1.3.1 将损失函数从最初形态转换为拉格朗日乘数形态
 - 2.1.3.2 将拉格朗日函数转换为拉格朗日对偶函数
 - 2.1.3.3 求解拉格朗日对偶函数极其后续过程
 - 2.1.4 线性SVM决策过程的可视化
- 2.2 非线性SVM与核函数
 - 2.2.1 SVC在非线性数据上的推广
 - 2.2.2 重要参数kernel
 - 2.2.3 探索核函数在不同数据集上的表现
 - 2.2.4 探索核函数的优势和缺陷
 - 2.2.5 选取与核函数相关的参数: degree & gamma & coef0
- 2.3 硬间隔与软间隔: 重要参数C
 - 2.3.1 SVM在软间隔数据上的推广
 - 2.3.2 重要参数C
- 2.4 总结

3 附录

- 3.1 SVC的参数列表
- 3.2 SVC的属性列表
- 3.3 SVC的接口列表

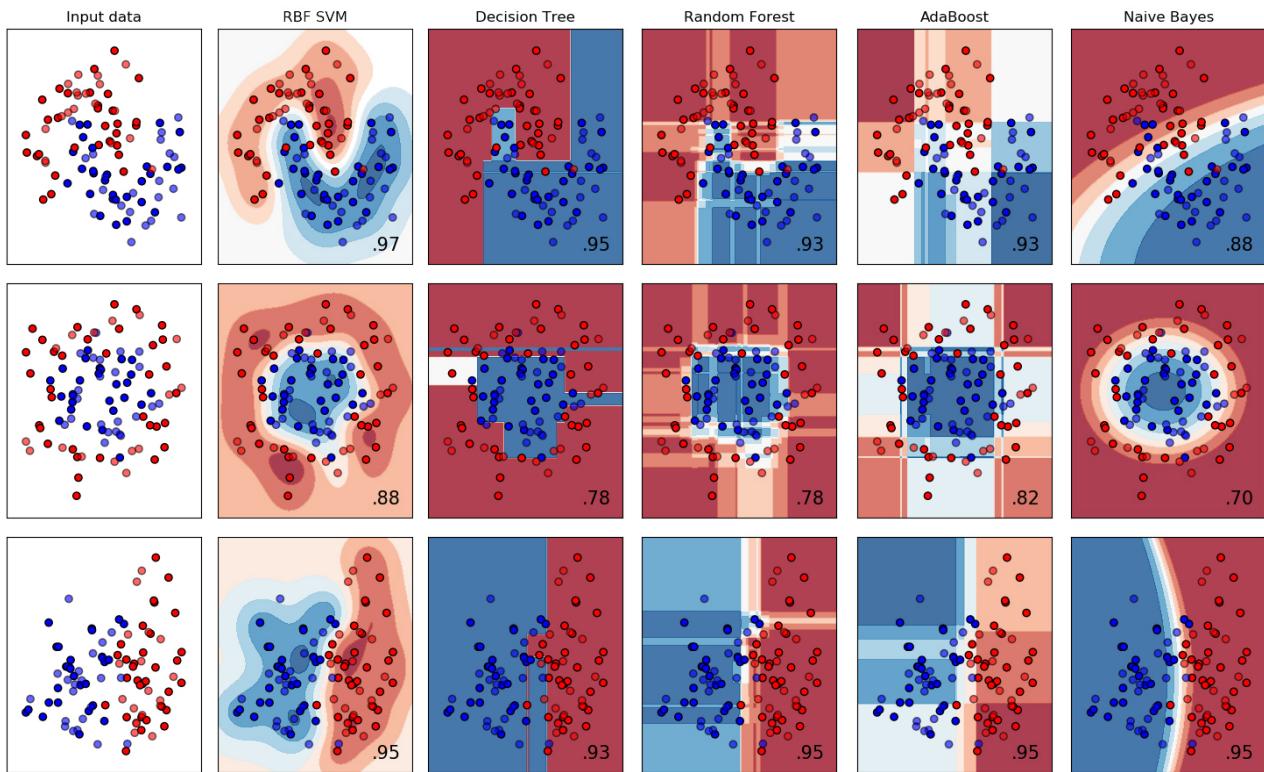
1 概述

支持向量机（SVM，也称为支持向量网络），是机器学习中获得关注最多的算法没有之一。它源于统计学习理论，是我们除了集成算法之外，接触的第一个强学习器。它有多强呢？

从算法的功能来看，SVM几乎囊括了我们前六周讲解的所有算法的功能：

	功能
有监督学习	线性二分类与多分类 (Linear Support Vector Classification) 非线性二分类与多分类 (Support Vector Classification, SVC) 普通连续型变量的回归 (Support Vector Regression) 概率型连续变量的回归 (Bayesian SVM)
无监督学习	支持向量聚类 (Support Vector Clustering, SVC) 异常值检测 (One-class SVM)
半监督学习	转导支持向量机 (Transductive Support Vector Machines, TSVM)

从分类效力来讲，SVM在无论线性还是非线性分类中，都是明星般的存在：



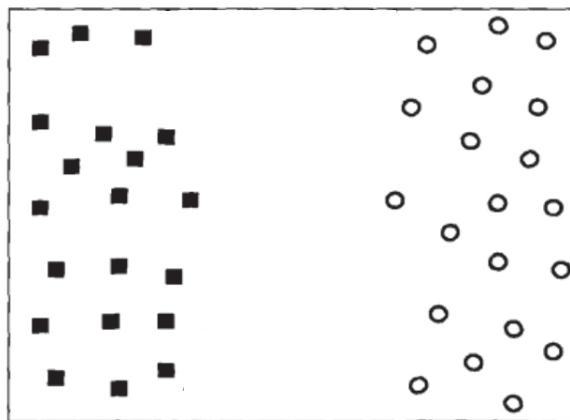
从实际应用来看，SVM在各种实际问题中都表现非常优秀。它在**手写识别数字和人脸识别**中应用广泛，在**文本和超文本的分类**中举足轻重，因为SVM可以大量减少标准归纳 (standard inductive) 和转换设置 (transductive settings) 中对标记训练实例的需求。同时，SVM也被用来执行**图像的分类，并用于图像分割系统**。实验结果表明，在仅仅三到四轮相关反馈之后，SVM就能实现比传统的查询细化方案 (query refinement schemes) 高出一大截的搜索精度。除此之外，生物学和许多其他科学都是SVM的青睐者，SVM现在已经广泛被用于**蛋白质分类**，现在化合物分类的业界平均水平可以达到90%以上的准确率。在生物科学的尖端研究中，人们还使用支持向量机来**识别用于模型预测的各种特征**，以找出各种基因表现结果的影响因素。

从学术的角度来看，SVM是最接近深度学习的机器学习算法。线性SVM可以看成是神经网络的单个神经元（虽然损失函数与神经网络不同），非线性的SVM则与两层的神经网络相当，非线性的SVM中如果添加多个核函数，则可以模仿多层的神经网络。而从数学的角度来看，SVM的数学原理是公认的对初学者来说难于上青天的水平，对于没有数学基础和数学逻辑熏陶的人来说，探究SVM的数学原理本身宛如在知识的荒原上跋涉。

当然了，没有算法是完美的，比SVM强大的算法在集成学习和深度学习中还有很多很多。但不可否认，它是我们目前为止接触到的最强大的算法。接下来的两周，我们将一起来探索SVM的神秘世界。

1.1 支持向量机分类器是如何工作的

支持向量机所作的事情其实非常容易理解。先来看看下面这一组数据的分布，这是一组两种标签的数据，两种标签分别由圆和方块代表。**支持向量机的分类方法，是在这组分布中找出一个超平面作为决策边界，使模型在数据上的分类误差尽量接近于小，尤其是在未知数据集上的分类误差（泛化误差）尽量小。**



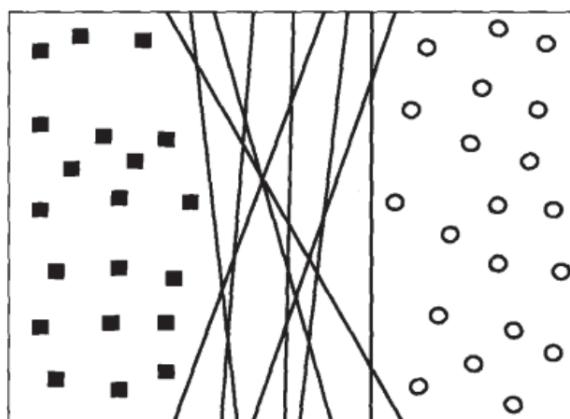
关键概念：超平面

在几何中，超平面是一个空间的子空间，它是维度比所在空间小一维的空间。如果数据空间本身是三维的，则其超平面是二维平面，而如果数据空间本身是二维的，则其超平面是一维的直线。

在二分类问题中，如果一个超平面能够将数据划分为两个集合，其中每个集合中包含单独的一个类别，我们就说这个超平面是数据的“决策边界”。

决策边界一侧的所有点在分类为属于一个类，而另一侧的所有点分类属于另一个类。如果我们能够找出决策边界，分类问题就可以变成探讨每个样本对于决策边界而言的相对位置。比如上面的数据分布，我们很容易就可以在方块和圆的中间画出一条线，并让所有落在直线左边的样本被分类为方块，在直线右边的样本被分类为圆。如果把数据当作我们的训练集，只要直线的一边只有一种类型的数据，就没有分类错误，我们的训练误差就会为0。

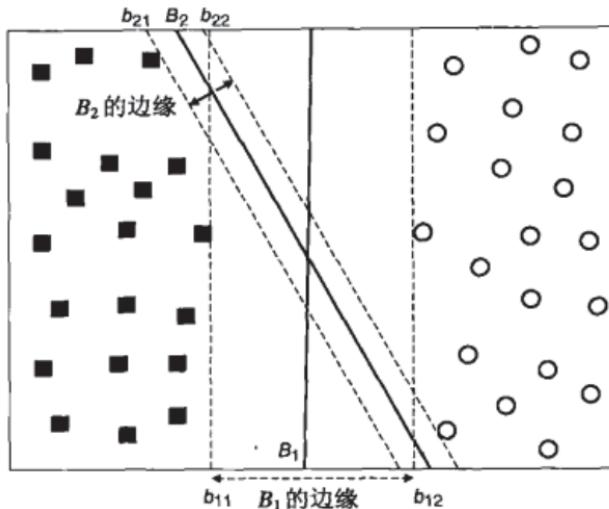
但是，对于一个数据集来说，让训练误差为0的决策边界可以有无数条。



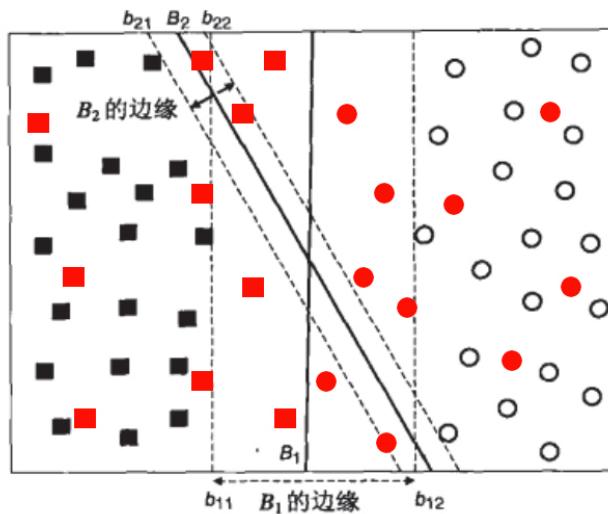
但在此基础上，我们无法保证这条决策边界在未知数据集（测试集）上的表现也会优秀。对于现有的数据集来说，我们有 B_1 和 B_2 两条可能的决策边界。我们可以把决策边界 B_1 向两边平移，直到碰到离这条决策边界最近的方块和圆圈后停下，形成两个新的超平面，分别是 b_{11} 和 b_{12} ，并且我们将原始的决策边界移动到 b_{11} 和 b_{12} 的中间，确保 B_1 到 b_{11} 和 b_{12} 的距离相等。在 b_{11} 和 b_{12} 之间的距离，叫做 B_1 这条决策边界的边际(margin)，通常记作 d 。

为了简便，我们称 b_{11} 和 b_{12} 为“虚线超平面”，在其他博客或教材中可能有着其他的称呼，但大家知道是这两个超平面是由原来的决策边界向两边移动，直到碰到距离原来的决策边界最近的样本后停下而形成的超平面就可以了。

对 B_2 也执行同样的操作，然后我们来对比一下两个决策边界。现在两条决策边界右边的数据都被判断为圆，左边的数据都被判断为方块，两条决策边界在现在的数据集上的训练误差都是0，没有一个样本被分错。



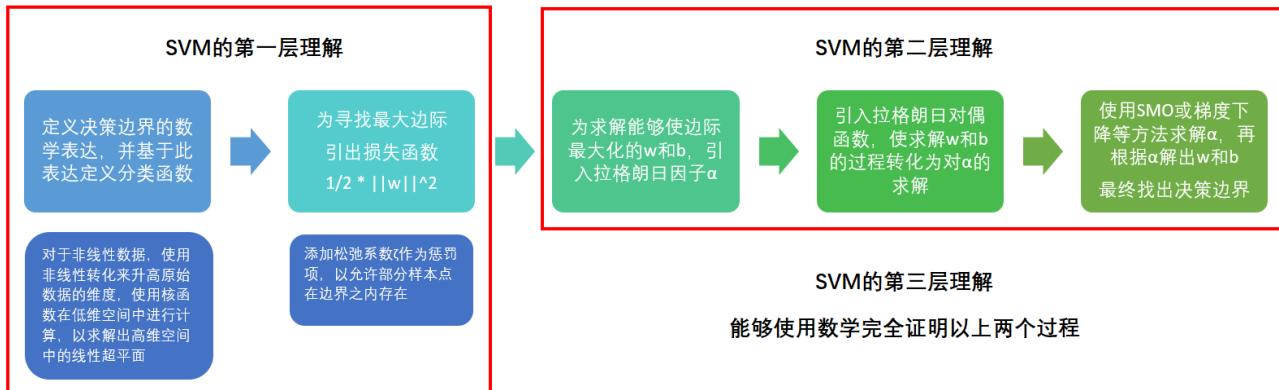
我们引入和原本的数据集相同分布的测试样本（红色所示），平面中的样本变多了，此时我们可以发现，对于 B_1 而言，依然没有一个样本被分错，这条决策边界上的泛化误差也是0。但是对于 B_2 而言，却有三个方块被误人类成了圆，二有两个圆被误分类成了方块，这条决策边界上的泛化误差就远远大于 B_1 了。这个例子表现出，**拥有更大边际的决策边界在分类中的泛化误差更小**，这一点可以由结构风险最小化定律来证明（SRM）。如果边际很小，则任何轻微扰动都会对决策边界的分类产生很大的影响。**边际很小的情况，是一种模型在训练集上表现很好，却在测试集上表现糟糕的情况，所以会“过拟合”**。所以我们在找寻决策边界的时候，希望边际越大越好。



支持向量机，就是通过找出边际最大的决策边界，来对数据进行分类的分类器。也因此，支持向量分类器又叫做最大边际分类器。这个过程在二维平面中看起来十分简单，但将上述过程使用数学表达出来，就不是一件简单的事情了。

1.2 支持向量机原理的三层理解

目标是“找出边际最大的决策边界”，听起来是一个十分熟悉的表达，这是一个最优化问题，而最优化问题往往和损失函数联系在一起。和逻辑回归中的过程一样，SVM也是通过最小化损失函数来求解一个用于后续模型使用的重要信息：决策边界。



1.2 sklearn中的支持向量机

类	含义	输入
svm.LinearSVC	线性支持向量分类	[penalty, loss, dual, tol, C, ...]
svm.LinearSVR	线性支持向量回归	[epsilon, tol, C, loss, ...]
svm.SVC	非线性多维支持向量分类	[C, kernel, degree, gamma, coef0, ...]
svm.SVR	非线性多维支持向量回归	[kernel, degree, gamma, coef0, tol, ...]
svm.NuSVC	Nu支持向量分类	[nu, kernel, degree, gamma, ...]
svm.NuSVR	Nu支持向量回归	[nu, C, kernel, degree, gamma, ...]
svm.OneClassSVM	无监督异常值检测	[kernel, degree, gamma, ...]
svm.l1_min_c	返回参数C的最低边界，使得对于C in (l1_min_C, infinity)，模型保证不为空	X, y[, loss, fit_intercept, ...]
直接使用libsvm的函数		
svm.libsvm.cross_validation	SVM专用的交叉验证	
svm.libsvm.decision_function	SVM专用的预测边际函数 (libsvm名称为predict_values)	
svm.libsvm.fit	使用libsvm训练模型	
svm.libsvm.predict	给定模型预测X的目标值	
svm.libsvm.predict_proba	预测概率	

注意，除了特别表明是线性的两个类LinearSVC和LinearSVR之外，其他的所有类都是同时支持线性和非线性的。NuSVC和NuSVR可以手动调节支持向量的数目，其他参数都与最常用的SVC和SVR一致。注意OneClassSVM是无监督的类。

除了本身所带的类之外，sklearn还提供了直接调用libsvm库的几个函数。Libsvm是台湾大学林智仁(Lin Chih-Jen)教授等人开发设计的一个简单、易于使用和快速有效的英文的SVM库，它提供了大量SVM的底层计算和参数选择，也是sklearn的众多类背后所调用的库。目前，LIBSVM拥有C、Java、Matlab、Python、R等数十种语言版本，每种语言版本都可以在libsvm的官网上进行下载：

<https://www.csie.ntu.edu.tw/~cjlin/libsvm/>

2 sklearn.svm.SVC

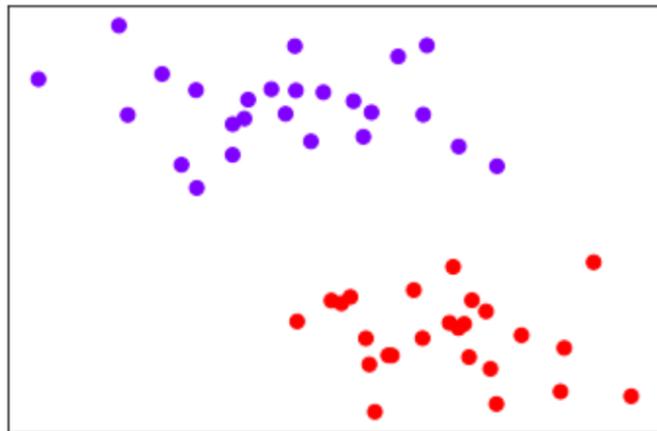
```
class sklearn.svm.SVC (C=1.0, kernel='rbf', degree=3, gamma='auto_deprecated', coef0=0.0, shrinking=True, probability=False, tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', random_state=None)
```

2.1 线性SVM用于分类的原理

2.1.1 线性SVM的损失函数详解

要理解SVM的损失函数，我们先来定义决策边界。假设现在数据中总计有 N 个训练样本，每个训练样本*i*可以被表示为 (\mathbf{x}_i, y_i) ($i = 1, 2, \dots, N$)，其中 \mathbf{x}_i 是 $(x_{1i}, x_{2i}, \dots, x_{ni})^T$ 这样的一个特征向量，每个样本总共含有 n 个特征。二分类标签 y_i 的取值是{-1, 1}。

如果n等于2，则有 $i = (x_{1i}, x_{2i}, y_i)^T$ ，分别由我们的特征向量和标签组成。此时我们可以在二维平面上，以 x_2 为横坐标， x_1 为纵坐标， y 为颜色，来可视化我们所有的N个样本：



我们让所有紫色点的标签为1，红色点的标签为-1。我们要在这个数据集上寻找一个决策边界，在二维平面上，决策边界（超平面）就是一条直线。二维平面上的任意一条线可以被表示为：

$$x_1 = ax_2 + b$$

我们将此表达式变换一下：

$$\begin{aligned} 0 &= ax_2 - x_1 + b \\ 0 &= [a, -1] * \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} + b \\ 0 &= \mathbf{w}^T \mathbf{x} + b \end{aligned}$$

其中 $[a, -1]$ 就是我们的参数向量 \mathbf{w} ， \mathbf{x} 就是我们的特征向量， b 是我们的截距。注意，这个表达式长得非常像我们线性回归的公式：

$$y(x) = \theta^T \mathbf{x} + \theta_0$$

线性回归中等号的一边是标签，回归过后会拟合出一个标签，而决策边界的表达式中却没有标签的存在，全部是由参数，特征和截距组成的一个式子，等号的一边是0。

在一组数据下，给定固定的 w 和 b ，这个式子就可以是一条固定直线，在 w 和 b 不确定的情况下，这个表达式 $w^T x + b = 0$ 就可以代表平面上的任意一条直线。如果在 w 和 b 固定时，给定一个唯一的 x 的取值，这个表达式就可以表示一个固定的点。**在SVM中，我们就使用这个表达式来表示我们的决策边界。**我们的目标是求解能够让边际最大化的决策边界，所以我们要求解参数向量 w 和截距 b 。

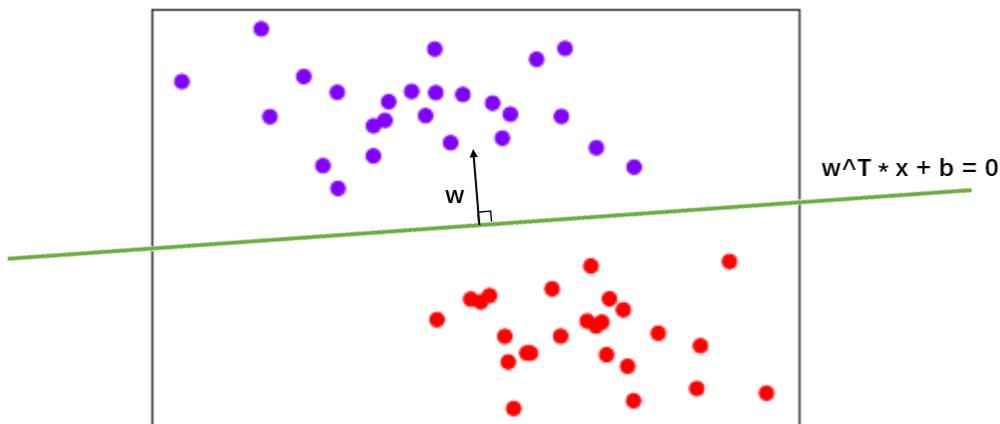
如果在决策边界上任意取两个点 x_a, x_b ，并带入决策边界的表达式，则有：

$$\begin{aligned} w^T x_a + b &= 0 \\ w^T x_b + b &= 0 \end{aligned}$$

将两式相减，可以得到：

$$w^T * (x_a - x_b) = 0$$

一个列向量的转置乘以另一个列向量，可以获得两个向量的点积(dot product)，表示为 $\langle w \cdot (x_a - x_b) \rangle$ 。两个向量的点击为0表示两个向量的方向式互相垂直的。 x_a 与 x_b 是一条直线上的两个点，相减后的得到的向量方向是由 x_b 指向 x_a ，所以 $x_a - x_b$ 的方向是平行于他们所在的直线——我们的决策边界的。而 w 与 $x_a - x_b$ 相互垂直，所以参数向量 w 的方向必然是垂直于我们的决策边界。



此时，我们有了我们的决策边界。任意一个紫色的点 x_p 就可以被表示为：

$$w \cdot x_p + b = p$$

由于紫色的点所代表的标签 y 是1，所以我们规定， $p>0$ 。同样的，对于任意一个红色的点 x_r 而言，我们可以将它表示为：

$$w \cdot x_r + b = r$$

由于红色点所表示的标签 y 是-1，所以我们规定， $r<0$ 。由此，如果我们有新的测试数据 x_t ，则的 x_t 标签就可以根据以下式子来判定：

$$y = \begin{cases} 1, & \text{if } w \cdot x_t + b > 0 \\ -1, & \text{if } w \cdot x_t + b < 0 \end{cases}$$

核心误区：p和r的符号

注意，在这里， p 和 r 的符号是我们人为规定的。在一些博客或教材中，会认为 p 和 r 的符号是由原本的决策边界上下移动得到。这是一种误解。

如果 k 和 k' 是由原本的决策边界平移得到的话，紫色的点在决策边界上方， $w \cdot x + b = 0$ 应该要向上平移，直线向上平移的话是增加截距，也就是说应该写作 $w \cdot x + b + \text{一个正数} = 0$ ，那 p 在等号的右边，怎么可能

核心误区：p和r的符号 (续)

一个大于0的数呢？同理，向下平移的话应该是截距减小，所以r也不可能是一个小于0的数。所以p和r的符号，不完全是平移的结果。

有人说，“直线以上的点带入直线为正，直线以下的点带入直线为负”是直线的性质，这又是另一种误解。假设我们有穿过圆点的直线 $y = x$ ，我们取点 $(x,y) = (0,1)$ 这个在直线上的点为例，如果直线的表达式写作 $y - x = 0$ ，则点 $(0,1)$ 带入后为正1，如果我们将直线的表达式写作 $x - y = 0$ ，则带入 $(0,1)$ 后结果为-1。所以，一个点在直线的上方，究竟会返回什么样的符号，是跟直线的表达式的写法有关的，不是直线上的点都为正，直线下的点都为负。

可能细心的小伙伴会发现，我们规定了p和r的符号与标签的符号一致，所以有人会说，p和r的符号，由所代表的点的标签的符号决定。这不是完全错误的，但这种说法无法解释，为什么我们就可以这样规定。并且，标签可以不是{-1,1}，可以是{0, 1}，可以是{1,2}，两个标签之间并不需要是彼此的负数，标签的取值其实也是我们规定的。

那p和r的符号，到底是依据什么来定的呢？数学中很多过程，都是可以取巧的，来看以下过程。记得我们的决策边界如果写成矩阵，可以表示为：

$$\begin{aligned}[a, -1] * \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} + b &= 0 \\ \mathbf{w} \cdot \mathbf{x} + b &= 0\end{aligned}$$

紫色点 x_p 毫无疑问是在决策边界的上方的，此时我将决策边界向上移动，形成一条过 x_p 的直线。根据我们平移的规则，直线向上平移，是在截距后加一个正数，则等号的右边是一个负数，假设这个数等于-3，则有：

$$[a, -1] * \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} + b = -3$$

另等式两边同时乘以 -1:

$$\begin{aligned}[-a, 1] * \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} + (-b) &= 3 \\ \mathbf{w} \cdot \mathbf{x} + b &= 3\end{aligned}$$

可以注意到，我们的参数向量由 $[a, -1]$ 变成了 $[-a, 1]$ ， b 变成了 $-b$ ，但参数向量依旧可以被表示成 \mathbf{w} ，只是它是原来的负数了，截距依旧可以被表示成 b ，只是如果它原来是正，它现在就是负数了，如果它原本就是负数，那它现在就是正数了。在这个调整中，我们通过将向上平移时产生的负号放入了参数向量和截距当中，这不影响我们求解，只不过我们求解出的参数向量和截距的符号变化了，但决策边界本身没有变化。所以我们依然可以使用原来的字母来表示这些更新后的参数向量和截距。通过这种方法，我们让 $\mathbf{w} \cdot \mathbf{x} + b = p$ 中的p大于0。我们让p大于0的目的，是为了它的符号能够与我们的标签的符号一致，都是为了后续计算和推导的简便。

为了推导和计算的简便，我们规定：

标签是{-1,1}

决策边界以上的点，标签都为正，并且通过调整 \mathbf{w} 和 b 的符号，让这个点在 $\mathbf{w} \cdot \mathbf{x} + b$ 上得出的结果为正。

决策边界以下的点，标签都为负，并且通过调整 \mathbf{w} 和 b 的符号，让这个点在 $\mathbf{w} \cdot \mathbf{x} + b$ 上得出的结果为负。

结论：决策边界以上的点都为正，以下的点都为负，是我们为了计算简便，而人为规定的。这种规定，不会影响对参数向量 \mathbf{w} 和截距 b 的求解。

有了这个理解，剩下的推导就简单多了。我们之前说过，决策边界的两边要有两个超平面，这两个超平面在二维空间中就是两条平行线（就是我们的虚线超平面），而他们之间的距离就是我们的边际 d 。而决策边界位于这两条线的中间，所以这两条平行线必然是对称的。我们另这两条平行线被表示为：

$$\mathbf{w} \cdot \mathbf{x} + b = k, \quad \mathbf{w} \cdot \mathbf{x} + b = -k$$

两个表达式同时除以 k ，则可以得到：

$$\mathbf{w} \cdot \mathbf{x} + b = 1, \quad \mathbf{w} \cdot \mathbf{x} + b = -1$$

这就是我们平行于决策边界的两条线的表达式，**表达式两边的1和-1分别表示了两条平行于决策边界的虚线到决策边界的相对距离**。此时，我们可以让这两条线分别过**两类数据中距离我们的决策边界最近的点**，这些点就被称为“支持向量”，而决策边界永远在这两条线的中间，所以可以被调整。我们令紫色类的点为 x_p ，红色类的点为 x_r ，则我们可以得到：

$$\mathbf{w} \cdot \mathbf{x}_p + b = 1, \quad \mathbf{w} \cdot \mathbf{x}_r + b = -1$$

两个式子相减，则有：

$$\mathbf{w} \cdot (\mathbf{x}_p - \mathbf{x}_r) = 2$$

如下图所示， $(\mathbf{x}_p - \mathbf{x}_r)$ 可表示为两点之间的连线，而我们的边际 d 是平行于 \mathbf{w} 的，所以我们现在，相当于是得到了三角型中的斜边，并且知道一条直角边的方向。在线性代数中，我们有如下数学性质：

线性代数中模长的运用

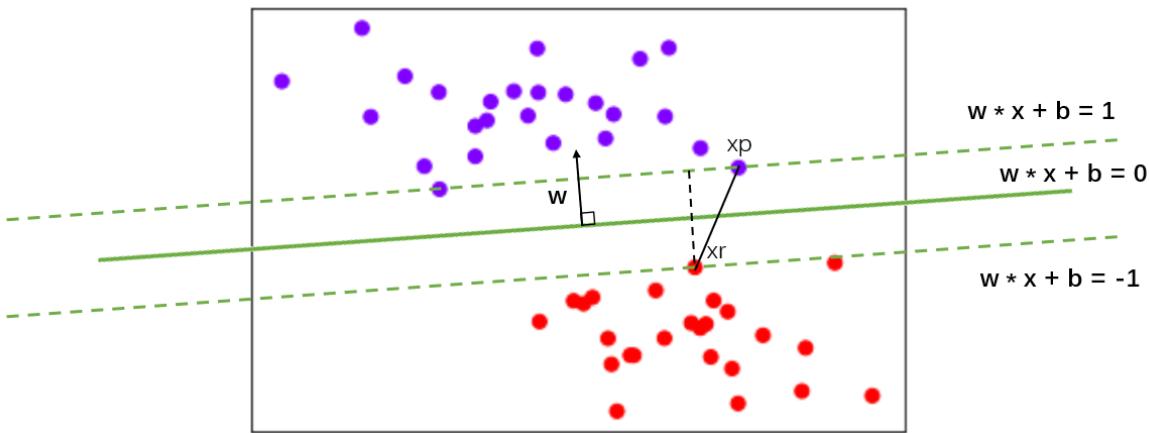
向量 b 除以自身的模长 $\|\mathbf{b}\|$ 可以得到 b 方向上的单位向量。

向量 a 乘以向量 b 方向上的单位向量，可以得到向量 a 在向量 b 方向上的投影的长度。

所以，我们另上述式子两边同时除以 $\|\mathbf{w}\|$ ，则可以得到：

$$\frac{\mathbf{w} \cdot (\mathbf{x}_p - \mathbf{x}_r)}{\|\mathbf{w}\|} = \frac{2}{\|\mathbf{w}\|}$$

$$\therefore d = \frac{2}{\|\mathbf{w}\|}$$



还记得我们想求什么吗？最大边界所对应的决策边界，那问题就简单了，**要最大化 d ，就求解 w 的最小值**。极值问题可以相互转化，我们可以把求解 w 的最小值转化为，求解以下函数的最小值：

$$f(w) = \frac{\|\mathbf{w}\|^2}{2}$$

之所以要在模长上加上平方，是因为模长的本质是一个距离，所以它是一个带根号的存在，我们对它取平方，是为了消除根号（其实模长的本质是向量 w 的l2范式，还记得l2范式公式如何写的小伙伴必定豁然开朗）。

我们的两条虚线表示的超平面，是数据边缘所在的点。所以对于任意样本*i*，我们可以把决策函数写作：

$$\begin{aligned} \mathbf{w} \cdot \mathbf{x}_i + b &\geq 1 \quad \text{if } y_i = 1 \\ \mathbf{w} \cdot \mathbf{x}_i + b &\leq -1 \quad \text{if } y_i = -1 \end{aligned}$$

整理一下，我们可以把两个式子整合成：

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1, i = 1, 2, \dots, N$$

在一部分教材中，这个式子被称为“函数间隔”。将函数间隔作为条件附加到我们的 $f(w)$ 上，我们就得到了SVM的损失函数最初形态：

$$\begin{aligned} \min_{w,b} \frac{\|\mathbf{w}\|^2}{2} \\ \text{subject to } y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1, \quad i = 1, 2, \dots, N. \end{aligned}$$

到这里，我们就完成了对SVM第一层理解的第一部分：线性SVM做二分类的损失函数。

2.1.2 函数间隔与几何间隔

重要定义：函数间隔与几何间隔

每一本机器学习的书或每一篇博客都可能有不同的原理讲解思路，在许多教材中，推导损失函数的过程与我们现在所说的不同。许多教材会先定义如下概念来辅助讲解：

对于给定的数据集T和超平面 (w, b) ，定义超平面 (w, b) 关于样本点 (x_i, y_i) 的函数间隔为：

$$\gamma_i = y_i(\mathbf{w} \cdot \mathbf{x}_i + b)$$

这其实是我们的虚线超平面的表达式整理过后得到的式子。函数间隔可以表示分类预测的正确性以及确信度。

再在这个函数间隔的基础上除以 w 的模长 $\|\mathbf{w}\|$ 来得到几何间隔：

$$\gamma_i = y_i \left(\frac{\mathbf{w}}{\|\mathbf{w}\|} \cdot \mathbf{x}_i + \frac{b}{\|\mathbf{w}\|} \right)$$

几何间隔的本质其实是点 x_i 到超平面 (w, b) ，即到我们的决策边界的带符号的距离(signed distance)。

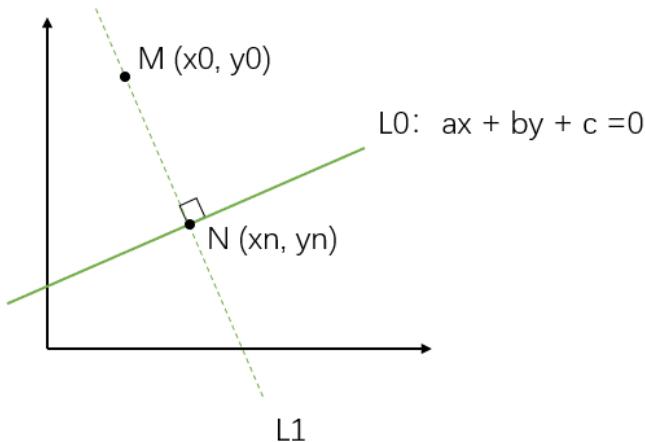
为什么几何间隔能够表示点到决策边界的距离？如果理解点到直线的距离公式，就可以很简单地理解这个式子。对于平面上的一个点 (x_0, y_0) 和一条直线 $ax + by + c = 0$ ，我们可以推导出点到直线的距离为：

$$\text{distance} = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

其中 $[a,b]$ 就是直线的参数向量 w ，而 $\sqrt{a^2 + b^2}$ 其实就是参数向量 w 的模长 $\|\mathbf{w}\|$ 。而我们的几何间隔中， y_i 的取值是 $\{-1, 1\}$ ，所以并不影响整个表达式的大小，只影响方向。而 $wx + b = 0$ 是决策边界，所以直线带入 x_i 后再除以参数向量的模长，就可以得到点 x_i 到决策边界的距离。

- 点到之间的距离的公式推导

现在有直线 $L_0 : ax + by + c = 0$ ，在直线上任意处选取点 $M(x_0, y_0)$ ，过点M画垂直于直线 L_0 的线 L_1 ，两条线的交点为 $N(x_n, y_n)$ 。现在MN线段的长度就是点M到直线 L_0 的距离，在求解距离过程中，M点的坐标和直线 L_0 都是已知的，未知数是交点的坐标 $N(x_n, y_n)$ ，于是我们将求解距离问题转化为求解交点的问题。



首先将 L_1 稍作变化，写作 $y = -\frac{a}{b}x - \frac{c}{b}$ 。开始证明：

$$\because L_0 \perp L_1$$

\therefore 两条直线的斜率相乘必为 -1

$$\text{又 } \because \beta_{L1} = -\frac{a}{b}, \therefore \beta_{L2} = \frac{b}{a}$$

$$\therefore L_1 : y = \frac{b}{a}x - c'$$

点 M 和点 N 都位于 L_1 上，将两点带入到直线中可得：

$$y_0 = \frac{b}{a}x_0 - c'$$

$$y_n = \frac{b}{a}x_n - c'$$

将两个方程相减，可得到：

$$y_0 - y_n = \frac{b}{a}(x_0 - x_n) \quad (1)$$

由于点 N 其实在直线 L_0 上，我们又可以得到：

$$y_n = -\frac{a}{b}x_n - \frac{c}{b} \quad (2)$$

将两个式子整合：

$$\begin{aligned} y_0 + \frac{a}{b}x_n + \frac{c}{b} &= \frac{b}{a}(x_0 - x_n) \\ aby_0 + a^2x_n + ac &= b^2x_0 - b^2x_n \\ a^2x_n + b^2x_n &= b^2x_0 - ac - aby_0 \\ x_n(a^2 + b^2) &= b^2x_0 - ac - aby_0 \\ x_n &= \frac{b^2x_0 - aby_0 - ac}{a^2 + b^2} \end{aligned}$$

将结果带入等式(2)中，可得到：

$$\begin{aligned} y_n &= -\frac{a}{b} * \frac{b^2x_0 - aby_0 - ac}{(a^2 + b^2)} - \frac{c}{b} \\ &= \frac{-ab^2x_0 + a^2by_0 + a^2c}{b(a^2 + b^2)} - \frac{c}{b} \\ &= \frac{-ab^2x_0 + a^2by_0 + a^2c}{b(a^2 + b^2)} - \frac{c(a^2 + b^2)}{b(a^2 + b^2)} \\ &= \frac{-ab^2x_0 + a^2by_0 + a^2c - (a^2c + b^2c)}{b(a^2 + b^2)} \\ &= \frac{-ab^2x_0 + a^2by_0 - b^2c}{b(a^2 + b^2)} \\ &= \frac{a^2y_0 - abx_0 - bc}{a^2 + b^2} \end{aligned}$$

$$\therefore N(x_n, y_n) = N\left(\frac{b^2x_0 - aby_0 - ac}{a^2 + b^2}, \frac{a^2y_0 - abx_0 - bc}{a^2 + b^2}\right)$$

求得点N的坐标后，线段MN的长度，即M到N的距离则可表示为：

$$\begin{aligned}
 MN &= \sqrt{(x_n - x_0)^2 + (y_n - y_0)^2} \\
 &= \sqrt{\left(\frac{b^2 x_0 - aby_0 - ac}{a^2 + b^2} - x_0\right)^2 + \left(\frac{a^2 y_0 - abx_0 - bc}{a^2 + b^2} - y_0\right)^2} \\
 &= \sqrt{\left(\frac{b^2 x_0 - aby_0 - ac - a^2 x_0 - b^2 x_0}{a^2 + b^2}\right)^2 + \left(\frac{a^2 y_0 - abx_0 - bc - a^2 y_0 - b^2 y_0}{a^2 + b^2}\right)^2} \\
 &= \sqrt{\left(\frac{-aby_0 - ac - a^2 x_0}{a^2 + b^2}\right)^2 + \left(\frac{-abx_0 - bc - b^2 y_0}{a^2 + b^2}\right)^2} \\
 &= \sqrt{\frac{a^2(ax_0 + c + by_0)^2 + b^2(ax_0 + c + by_0)^2}{(a^2 + b^2)^2}} \\
 &= \sqrt{\frac{(a^2 + b^2)(by_0 + c + ax_0)^2}{(a^2 + b^2)^2}} \\
 &= \sqrt{\frac{(by_0 + c + ax_0)^2}{(a^2 + b^2)}} \\
 &= \frac{|ax_0 + by_0 + c|}{\sqrt{(a^2 + b^2)}}
 \end{aligned}$$

证明完毕。

2.1.3 线性SVM的拉格朗日对偶函数和决策函数

有了我们的损失函数过后，我们就需要对损失函数进行求解。这个求解过程异常复杂，涉及到的数学的难度不是推导损失函数的部分可比。并且，在sklearn当中，我们作为使用者完全无法干涉这个求解的过程。因此作为使用sklearn的人，这部分属于进阶内容。如果实在对数学感到苦手，大家也可根据自己的需求选读。

我们之前得到了线性SVM损失函数的最初形态：

$$\begin{aligned}
 &\min_{w,b} \frac{\|w\|^2}{2} \\
 &\text{subject to } y_i(w \cdot x_i + b) \geq 1, \\
 &\quad i = 1, 2, \dots, N.
 \end{aligned}$$

这个损失函数分为两部分：需要最小化的函数，以及参数求解后必须满足的约束条件。这是一个最优化问题。

2.1.3.1 将损失函数从最初形态转换为拉格朗日乘数形态

- 为什么要进行转换？

我们的目标是求解让损失函数最小化的 w ，但其实很容易看得出来，如果 $\|w\|=0$ ， $f(w)$ 必然最小了。但是， $\|w\|=0$ 其实是一个无效的值，原因有简单：首先，我们的决策边界是 $w \cdot x + b = 0$ ，如果 w 为0，则这个向量里包含的所有元素都为0，那就有 $b=0$ 这个唯一值。然而，如果 b 和 w 都为0，决策边界就不再是一条直线了，函数间隔 $y_i(w \cdot x_i + b)$ 就会为0，条件中的 $y_i(w \cdot x_i + b) \geq 1$ 就不可能实现，所以 w 不可以是一个0向量。可见，单纯让 $f(w) = \frac{\|w\|^2}{2}$ 为0，是不能求解出合理的 w 的，我们希望能够找出一种方式，能够让我们的条件

$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$ 在计算中也被纳入考虑，一种业界认可的方法是使用拉格朗日乘数法(standard Lagrange multiplier method)。

- 为什么可以进行转换？

我们的损失函数是二次的(quadratic)，并且我们损失函数中的约束条件在参数 w 和 b 下是线性的，求解这样的损失函数被称为“凸优化问题”(convex optimization problem)。拉格朗日乘数法正好可以用来解决凸优化问题，这种方法也是业界常用的，用来解决带约束条件，尤其是带有不等式的约束条件的函数的数学方法。首先第一步，我们需要使用拉格朗日乘数来将损失函数改写为考虑了约束条件的形式：

$$L(w, b, \alpha) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^N \alpha_i (y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1) \quad (\alpha_i \geq 0)$$

这是一个非常聪明而且巧妙的表达式，它被称为拉格朗日函数，其中 α_i 就叫做拉格朗日乘数。此时此刻，我们要求解的就不只有参数向量 w 和截距 b 了，我们也要求解拉格朗日乘数 α ，而我们的 x_i 和 y_i 都是我们已知的特征矩阵和标签。

- 怎样进行转换？

拉格朗日函数也分为两部分。第一部分和我们原始的损失函数一样，第二部分呈现了我们带有不等式的约束条件。我们希望， $L(w, b, \alpha)$ 不仅能够代表我们原有的损失函数 $f(w)$ 和约束条件，还能够表示我们想要最小化损失函数来求解 w 和 b 的意图，所以我们要先以 α 为参数，求解 $L(w, b, \alpha)$ 的最大值，再以 w 和 b 为参数，求解 $L(w, b, \alpha)$ 的最小值。因此，我们的目标可以写作：

$$\min_{w,b} \max_{\alpha_i \geq 0} L(w, b, \alpha) \quad (\alpha_i \geq 0)$$

怎么理解这个式子呢？首先，我们第一步先执行 \max ，即最大化 $L(w, b, \alpha)$ ，那就有两种情况：

当 $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) > 1$ ，函数的第二部分 $\sum_{i=1}^N \alpha_i (y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1)$ 就一定为正，式子 $\frac{1}{2} \|\mathbf{w}\|^2$ 就要减去一个正数，此时若要最大化 $L(w, b, \alpha)$ ，则 α 必须取到0。

当 $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) < 1$ ，函数的第二部分 $\sum_{i=1}^N \alpha_i (y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1)$ 就一定为负，式子 $\frac{1}{2} \|\mathbf{w}\|^2$ 就要减去一个负数，相当于加上一个正数，此时若要最大化 $L(w, b, \alpha)$ ，则 α 必须取到正无穷。

若把函数第二部分当作一个惩罚项来看待，则 $y_i(\mathbf{w} \cdot \mathbf{x}_i + b)$ 大于1时函数没有受到惩罚，而 $y_i(\mathbf{w} \cdot \mathbf{x}_i + b)$ 小于1时函数受到了极致的惩罚，即加上了一个正无穷项，函数整体永远不可能取到最小值。所以第二步，我们执行 \min 的命令，求解函数整体的最小值，我们就永远不能让 α 必须取到正无穷的状况出现，即是说永远不让 $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) < 1$ 的状况出现，从而实现了求解最小值的同时让约束条件被满足。

现在， $L(w, b, \alpha)$ 就是我们新的损失函数了，我们的目标是要通过先最大化，在最小化它来求解参数向量 w 和截距 b 的值。

2.1.3.2 将拉格朗日函数转换为拉格朗日对偶函数

- 为什么要进行转换？

要求极值，最简单的方法还是对参数求导后让一阶导数等于0。我们先来试试看对拉格朗日函数求极值，在这里我们对参数向量 w 和截距 b 分别求偏导并且让他们等于0。这个求导过程比较简单：

$$\begin{aligned}
L(w, b, \alpha) &= \frac{1}{2} \|w\|^2 - \sum_{i=1}^N \alpha_i (y_i (\mathbf{w} \cdot \mathbf{x}_i + b) - 1) \\
&= \frac{1}{2} \|w\|^2 - \sum_{i=1}^N (\alpha_i y_i \mathbf{w} \cdot \mathbf{x}_i + \alpha_i y_i b - \alpha_i) \\
&= \frac{1}{2} \|w\|^2 - \sum_{i=1}^N (\alpha_i y_i \mathbf{w} \cdot \mathbf{x}_i) - \sum_{i=1}^N \alpha_i y_i b + \sum_{i=1}^N \alpha_i \\
&= \frac{1}{2} (\mathbf{w}^T \mathbf{w})^{\frac{1}{2}*2} - \sum_{i=1}^N (\alpha_i y_i \mathbf{w} \cdot \mathbf{x}_i) - \sum_{i=1}^N \alpha_i y_i b + \sum_{i=1}^N \alpha_i \\
&= \frac{1}{2} \mathbf{w}^T \mathbf{w} - \sum_{i=1}^N (\alpha_i y_i \mathbf{w} \cdot \mathbf{x}_i) - \sum_{i=1}^N \alpha_i y_i b + \sum_{i=1}^N \alpha_i \\
\frac{\partial L(\mathbf{w}, b, \alpha)}{\partial \mathbf{w}} &= \frac{1}{2} * 2\mathbf{w} - \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i \\
&= \mathbf{w} - \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i = 0 \rightarrow \mathbf{w} = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i \quad (1) \\
\frac{\partial L(\mathbf{w}, b, \alpha)}{\partial b} &= \sum_{i=1}^N \alpha_i y_i = 0 \rightarrow \sum_{i=1}^N \alpha_i y_i = 0 \quad (2)
\end{aligned}$$

由于两个求偏导结果中都带有未知的拉格朗日乘数 α_i ，因此我们还是无法求解出 w 和 b ，我们必须想出一种方法来求解拉格朗日乘数 α_i 。幸运地是，拉格朗日函数可以被转换成一种只带有 α_i ，而不带有 w 和 b 的形式，这种形式被称为拉格朗日对偶函数。在对偶函数下，我们就可以求解出拉格朗日乘数 α_i ，然后带入到上面推导出的(1)和(2)式中来求解 w 和 b 。

- 为什么能够进行转换？

对于任何一个拉格朗日函数 $L(x, \alpha) = f(x) + \sum_{i=1}^q \alpha_i h_i(x)$ ，都存在一个与它对应的对偶函数 $g(\alpha)$ ，只带有拉格朗日乘数 α 作为唯一的参数。如果 $L(x, \alpha)$ 的最优解存在并可以表示为 $\min_x L(x, \alpha)$ ，并且对偶函数的最优解也存在并可以表示为 $\max_{\alpha} g(\alpha)$ ，则我们可以定义**对偶差异(dual gap)**，即**拉格朗日函数的最优解与其对偶函数的最优解之间的差值**：

$$\Delta = \min_x L(x, \alpha) - \max_{\alpha} g(\alpha)$$

如果 $\Delta = 0$ ，则称 $L(x, \alpha)$ 与其对偶函数之间存在**强对偶关系(strong duality property)**，此时我们就可以通过求解其对偶函数的最优解来替代求解原始函数的最优解。那强对偶关系什么时候存在呢？则这个拉格朗日函数必须满足KKT(Karush-Kuhn-Tucker)条件：

$$\begin{aligned}
\frac{\partial L}{\partial x_i} &= 0, \forall i = 1, 2, \dots, d \\
h_i(x) &\leq 0, \forall i = 1, 2, \dots, q \\
\alpha_i &\geq 0, \forall i = 1, 2, \dots, q \\
\alpha_i h_i(x) &= 0, \forall i = 1, 2, \dots, q
\end{aligned}$$

这里的条件其实都比较好理解。首先是所有参数的一阶导数必须为0，然后约束条件中的函数本身需要小于等于0，拉格朗日乘数需要大于等于0，以及约束条件乘以拉格朗日乘数必须等于0，即不同 i 的取值下，两者之中至少有一个为0。当所有限制都被满足，则拉格朗日函数 $L(x, \alpha)$ 的最优解与其对偶函数的最优解相等，我们就可以将原始的最优化问题转换成为对偶函数的最优化问题。而不难注意到，对于我们的损失函数 $L(w, b, \alpha)$ 而言，KKT条件都是可以操作的。如果我们能够人为让KKT条件全部成立，我们就可以求解出 $L(w, b, \alpha)$ 的对偶函数来解出 α 。

之前我们已经让拉格朗日函数上对参数 w 和 b 的求导为0, 得到了式子:

$$\sum_{i=1}^N \alpha_i y_i \mathbf{x}_i = \mathbf{w} \quad (1)$$

$$\sum_{i=1}^N \alpha_i y_i = 0 \quad (2)$$

并且在我们的函数中, 我们通过先求解最大值再求解最小值的方法使得函数天然满足:

$$-(y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1) \leq 0 \quad (3), \quad \alpha_i \geq 0 \quad (4)$$

所以接下来, 我们只需要再满足一个条件:

$$\alpha_i(y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1) = 0 \quad (5)$$

这个条件其实很容易满足, 能够让 $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 = 0$ 的就是落在虚线的超平面上的样本点, 即我们的支持向量。所有不是支持向量的样本点则必须满足 $\alpha_i = 0$ 。满足这个式子说明了, 我们求解的参数 w 和 b 以及求解的超平面的存在, 只与支持向量相关, 与其他样本点都无关。现在KKT的五个条件都得到了满足, 我们就可以使用 $L(w, b, \alpha)$ 的对偶函数来求解 α 了。

• 怎样进行转换?

首先让拉格朗日函数对参数 w 和 b 求导后的结果为0, 本质是在探索拉格朗日函数的最小值。然后:

$$\begin{aligned} L(w, b, \alpha) &= \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^N \alpha_i (y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1) \\ &= \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^N (\alpha_i y_i \mathbf{w} \cdot \mathbf{x}_i) - \sum_{i=1}^N \alpha_i y_i b + \sum_{i=1}^N \alpha_i \\ &= \frac{1}{2} \|\mathbf{w}\|^2 - \mathbf{w} \sum_{i=1}^N (\alpha_i y_i \cdot \mathbf{x}_i) - b \sum_{i=1}^N \alpha_i y_i + \sum_{i=1}^N \alpha_i \end{aligned}$$

代入式(1)和式(2)有:

$$\begin{aligned} &= \frac{1}{2} \mathbf{w}^T \mathbf{w} - \mathbf{w}^T \mathbf{w} + \sum_{i=1}^N \alpha_i \\ &= -\frac{1}{2} \mathbf{w}^T \mathbf{w} + \sum_{i=1}^N \alpha_i \end{aligned}$$

再次带入式(1)则有:

$$-\frac{1}{2} \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i^T * \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i + \sum_{i=1}^N \alpha_i$$

令这两个 w 源于不同的特征矩阵和标签:

$$\begin{aligned} &= -\frac{1}{2} \sum_{i,j=1}^N \alpha_i y_i \mathbf{x}_i^T \alpha_j y_j \mathbf{x}_j + \sum_{i=1}^N \alpha_i \\ &= \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \end{aligned}$$

将矩阵相乘转换为内积形式:

$$L_d = \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j$$

函数 L_d 就是我们的对偶函数。对所有存在对偶函数的拉格朗日函数我们有对偶差异如下表示：

$$\Delta = \min_x L(x, \alpha) - \max_{\alpha} g(\alpha)$$

则对于我们的 $L(w, b, \alpha)$ 和 L_d ，我们则有：

$$\Delta = \min_{w,b} \max_{\alpha_i \geq 0} L(w, b, \alpha) - \max_{\alpha_i \geq 0} L_d$$

还记得我们推导 L_d 的第一步是什么吗？是对 $L(w, b, \alpha)$ 求偏导并让偏导数都为0，所以我们求解对偶函数的过程其实是在求解 $L(w, b, \alpha)$ 的最小值，所以我们又可以把公式写成：

$$\begin{aligned} \Delta &= \min_{w,b} \max_{\alpha_i \geq 0} L(w, b, \alpha) - \max_{\alpha_i \geq 0} \min_{w,b} L(w, b, \alpha) \\ &\because \text{所有 KKT 条件被满足} \\ \therefore \min_{w,b} \max_{\alpha_i \geq 0} L(w, b, \alpha) &= \max_{\alpha_i \geq 0} \min_{w,b} L(w, b, \alpha) \end{aligned}$$

这就是众多博客和教材上写的，对偶函数与原始函数的转化过程的由来。如此，我们只需要求解对偶函数的最大值，就可以求出 α 了。最终，我们的目标函数变化为：

$$\max_{\alpha_i \geq 0} \left(\sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \right)$$

2.1.3.3 求解拉格朗日对偶函数极其后续过程

到了这一步，我们就需要使用梯度下降，SMO或者二次规划(QP, quadratic programming)来求解我们的 α ，数学的难度又进一步上升。考虑到这一过程对数学的要求已经远远超出了我们需要的程度，更是远远超出我们在使用sklearn时需要掌握的程度，如何求解对偶函数中的 α 在这里就不做讲解了。

但大家需要知道，一旦我们求得了 α 值，我们就可以使用求导后得到的(1)式求解 w ，并可以使用(1)式和决策边界的表达式结合，得到下面的式子来求解 b ：

$$\sum_{i=1}^N \alpha_i y_i \mathbf{x}_i * x + b = 0$$

当求得特征向量 w 和 b ，我们就得到了我们的决策边界的表达式，也可以利用决策边界和其有关的超平面来进行分类了，我们的决策函数就可以被写作：

$$f(x_{test}) = sign(w \cdot x_{test} + b) = sign\left(\sum_{i=1}^N \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x}_{test} + b\right)$$

其中 x_{test} 是任意测试样本， $sign(h)$ 是 $h > 0$ 时返回1， $h < 0$ 时返回-1的符号函数。到这里，我们可以说我们完成了对SVM的第二层理解的大部分内容，我们了解了线性SVM的四种相关函数：损失函数的初始形态，拉格朗日函数，拉格朗日对偶函数以及最后的决策函数。熟练掌握以上的推导过程，对理解支持向量机会有极大的助益，也是对我们数学能力的一种完善。

2.1.4 线性SVM决策过程的可视化

我们可以使用sklearn中的式子来为可视化我们的决策边界，支持向量，以及决策边界平行的两个超平面。

1. 导入需要的模块

```
from sklearn.datasets import make_blobs
from sklearn.svm import SVC
import matplotlib.pyplot as plt
import numpy as np
```

2. 实例化数据集，可视化数据集

```
x,y = make_blobs(n_samples=50, centers=2, random_state=0, cluster_std=0.6)
plt.scatter(x[:,0],x[:,1],c=y,s=50,cmap="rainbow")
plt.xticks([])
plt.yticks([])
plt.show()
```

3. 画决策边界：理解函数contour

`matplotlib.axes.Axes.contour([X, Y], Z, [levels], **kwargs)`

Contour是我们专门用来绘制等高线的函数。等高线，本质上是在二维图像上表现三维图像的一种形式，其中两维X和Y是两条坐标轴上的取值，而Z表示高度。Contour就是将由X和Y构成平面上的所有点中，高度一致的点连接成线段的函数，在同一条等高线上的点一定具有相同的Z值。我们可以利用这个性质来绘制我们的决策边界。

参数	含义
X, Y	选填。两维平面上所有的点的横纵坐标取值，一般要求是二维结构并且形状需要与Z相同，往往通过numpy.meshgrid()这样的函数来创建。如果X和Y都是一维，则Z的结构必须为(len(Y), len(X))。如果不填写，则默认X = range(Z.shape[1]), Y = range(Z.shape[0])。
Z	必填。平面上所有的点所对应的高度。
levels	可不填，不填默认显示所有的等高线，填写用于确定等高线的数量和位置。如果填写整数n，则显示n个数据区间，即绘制n+1条等高线。水平高度自动选择。如果填写的是数组或列表，则在指定的高度级别绘制等高线。列表或数组中的值必须按递增顺序排列。

回忆一下，我们的决策边界是 $w \cdot x + b = 0$ ，并在决策边界的两边找出两个超平面，使得超平面到决策边界的相对距离为1。那其实，我们只需要在我们的样本构成的平面上，把所有到决策边界的距离为0的点相连，就是我们的决策边界，而把所有到决策边界的相对距离为1的点相连，就是我们的两个平行于决策边界的超平面了。此时，我们的Z就是平面上的任意点到达超平面的距离。

那首先，我们需要获取样本构成的平面，作为一个对象。

```
#首先要有散点图
plt.scatter(x[:,0],x[:,1],c=y,s=50,cmap="rainbow")
ax = plt.gca() #获取当前的子图，如果不存在，则创建新的子图
```

有了这个平面，我们需要在平面上制作一个足够细的网格，来代表我们“平面上的所有点”。

4. 画决策边界：制作网格，理解函数meshgrid

```
#获取平面上两条坐标轴的最大值和最小值
```

```

xlim = ax.get_xlim()
ylim = ax.get_ylim()

#在最大值和最小值之间形成30个规律的数据
axisx = np.linspace(xlim[0], xlim[1], 30)
axisy = np.linspace(ylim[0], ylim[1], 30)

axisy, axisx = np.meshgrid(axisy, axisx)
#我们将使用这里形成的二维数组作为我们contour函数中的X和Y
#使用meshgrid函数将两个一维向量转换为特征矩阵
#核心是将两个特征向量广播，以便获取y.shape * x.shape这么多个坐标点的横坐标和纵坐标

xy = np.vstack([axisx.ravel(), axisy.ravel()]).T
#其中ravel()是降维函数，vstack能够将多个结构一致的一维数组按行堆叠起来
#xy就是已经形成的网格，它是遍布在整个画布上的密集的点

plt.scatter(xy[:, 0], xy[:, 1], s=1, cmap="rainbow")

#理解函数meshgrid和vstack的作用
a = np.array([1, 2, 3])
b = np.array([7, 8])
#两两组合，会得到多少个坐标？
#答案是6个，分别是 (1,7), (2,7), (3,7), (1,8), (2,8), (3,8)

v1, v2 = np.meshgrid(a, b)

v1

v2

v = np.vstack([v1.ravel(), v2.ravel()]).T

```

有了网格后，我们需要计算网格所代表的“平面上所有的点”到我们的决策边界的距离。所以我们需要我们的模型和决策边界。

5. 建模，计算决策边界并找出网格上每个点到决策边界的距离

```

#建模，通过fit计算出对应的决策边界
clf = SVC(kernel = "linear").fit(X, y)
Z = clf.decision_function(xy).reshape(axisx.shape)
#重要接口decision_function，返回每个输入的样本所对应的到决策边界的距离
#然后再将这个距离转换为axisx的结构，这是由于画图的函数contour要求Z的结构必须与X和Y保持一致

#画决策边界和平行于决策边界的超平面
ax.contour(axisx, axisy, Z
            , colors="k"
            , levels=[-1, 0, 1] #画三条等高线，分别是z为-1, z为0和z为1的三条线
            , alpha=0.5
            , linestyles=["--", "-", "--"])
ax.set_xlim(xlim)
ax.set_ylim(ylim)

```

```
#记得Z的本质么？是输入的样本到决策边界的距离，而contour函数中的level其实是输入了这个距离
#让我们用一个点来试试看
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap="rainbow")
plt.scatter(X[10, 0], X[10, 1], c="black", s=50, cmap="rainbow")

clf.decision_function(X[10].reshape(1, 2))
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap="rainbow")
ax = plt.gca()
ax.contour(axisx, axisy, P
            , colors="k"
            , levels=[-3.33917354]
            , alpha=0.5
            , linestyles=["--"])
```

6. 将绘图过程包装成函数

```
#将上述过程包装成函数：
def plot_svc_decision_function(model, ax=None):
    if ax is None:
        ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    x = np.linspace(xlim[0], xlim[1], 30)
    y = np.linspace(ylim[0], ylim[1], 30)
    Y, X = np.meshgrid(y, x)
    xy = np.vstack([X.ravel(), Y.ravel()]).T
    P = model.decision_function(xy).reshape(X.shape)

    ax.contour(X, Y, P, colors="k", levels=[-1, 0, 1], alpha=0.5, linestyles=["--", "-", "--"])
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)

#则整个绘图过程可以写作：
clf = SVC(kernel = "linear").fit(X, y)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap="rainbow")
plot_svc_decision_function(clf)
```

7. 探索建好的模型

```
clf.predict(X)
#根据决策边界，对X中的样本进行分类，返回的结构为n_samples

clf.score(X, y)
#返回给定测试数据和标签的平均准确度

clf.support_vectors_
#返回支持向量

clf.n_support_
#返回每个类中支持向量的个数
```

8. 推广到非线性情况

我们之前所讲解的原理，以及绘图的过程，都是基于数据本身是线性可分的情况。如果把数据推广到非线性数据，比如说环形数据上呢？

```
from sklearn.datasets import make_circles
x,y = make_circles(100, factor=0.1, noise=.1)

x.shape

y.shape

plt.scatter(x[:,0],x[:,1],c=y,s=50,cmap="rainbow")
plt.show()
```

试试看用我们已经定义的函数来划分这个数据的决策边界：

```
clf = SVC(kernel = "linear").fit(x,y)
plt.scatter(x[:,0],x[:,1],c=y,s=50,cmap="rainbow")
plot_svc_decision_function(clf)
```

明显，现在线性SVM已经不适合于我们的状况了，我们无法找出一条直线来划分我们的数据集，让直线的两边分别是两种类别。这个时候，如果我们能够在原本的X和y的基础上，添加一个维度r，变成三维，我们可视化这个数据，来看看添加维度让我们的数据如何变化。

9. 为非线性数据增加维度并绘制3D图像

```
#定义一个由x计算出来的新维度r
r = np.exp(-(x**2).sum(1))

rlim = np.linspace(min(r),max(r),0.2)

from mpl_toolkits import mplot3d

#定义一个绘制三维图像的函数
#elev表示上下旋转的角度
#azim表示平行旋转的角度
def plot_3D(elev=30,azim=30,x=x,y=y):
    ax = plt.subplot(projection="3d")
    ax.scatter3D(x[:,0],x[:,1],r,c=y,s=50,cmap='rainbow')
    ax.view_init(elev=elev,azim=azim)
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("r")
    plt.show()

plot_3D()
```

可以看见，此时此刻我们的数据明显是线性可分的了：我们可以使用一个平面来将数据完全分开，并使平面的上方的所有数据点为一类，平面下方的所有数据点为另一类。

10. 将上述过程放到Jupyter Notebook中运行

```

#如果放到jupyter notebook中运行
from sklearn.svm import SVC
import matplotlib.pyplot as plt
import numpy as np

from sklearn.datasets import make_circles
x,y = make_circles(100, factor=0.1, noise=.1)
plt.scatter(x[:,0],x[:,1],c=y,s=50,cmap="rainbow")

def plot_svc_decision_function(model,ax=None):
    if ax is None:
        ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    x = np.linspace(xlim[0],xlim[1],30)
    y = np.linspace(ylim[0],ylim[1],30)
    X,Y = np.meshgrid(x,y)
    xy = np.vstack([X.ravel(), Y.ravel()]).T
    P = model.decision_function(xy).reshape(X.shape)

    ax.contour(X, Y, P,colors="k",levels=[-1,0,1],alpha=0.5,linestyles=["--","-","-"])
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)

clf = SVC(kernel = "linear").fit(x,y)
plt.scatter(x[:,0],x[:,1],c=y,s=50,cmap="rainbow")
plot_svc_decision_function(clf)

r = np.exp(-(x**2).sum(1))

rlim = np.linspace(min(r),max(r),0.2)

from mpl_toolkits import mplot3d

def plot_3D(elev=30,azim=30,X=X,y=y):
    ax = plt.subplot(projection="3d")
    ax.scatter3D(x[:,0],x[:,1],r,c=y,s=50,cmap='rainbow')
    ax.view_init(elev=elev,azim=azim)
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("r")
    plt.show()

from ipywidgets import interact,fixed
interact(plot_3D,elev=[0,30],azim=(-180,180),x=fixed(X),y=fixed(y))
plt.show()

```

此时我们的数据在三维空间中，我们的超平面就是一个二维平面。明显我们可以用一个平面将两类数据隔开，这个平面就是我们的决策边界了。我们刚才做的，计算r，并将r作为数据的第三维度来将数据升维的过程，被称为“核变换”，即是将数据投影到高维空间中，以寻找能够将数据完美分割的超平面，即是说寻找能够让数据线性可分的高维空间。为了详细解释这个过程，我们需要引入SVM中的核心概念：核函数。

2.2 非线性SVM与核函数

2.2.1 SVC在非线性数据上的推广

为了能够找出非线性数据的线性决策边界，我们需要将数据从原始的空间 x 投射到新空间 $\Phi(x)$ 中。 Φ 是一个映射函数，它代表了某种非线性的变换，如同我们之前所做过的使用 r 来升维一样，这种非线性变换看起来是一种非常有效的方式。使用这种变换，线性SVM的原理可以被很容易推广到非线性情况下，其推导过程和逻辑都与线性SVM一模一样，只不过在定义决策边界之前，我们必须先对数据进行升维度，即将原始的 x 转换成 $\Phi(x)$ 。

如此，非线性SVM的损失函数的初始形态为：

$$\begin{aligned} & \min_{w,b} \frac{\|w\|^2}{2} \\ & \text{subject to } y_i(w \cdot \Phi(x_i) + b) \geq 1, \\ & \quad i = 1, 2, \dots, N \end{aligned}$$

同理，非线性SVM的拉格朗日函数和拉格朗日对偶函数也可得：

$$\begin{aligned} L(w, b, \alpha) &= \frac{1}{2}\|w\|^2 - \sum_{i=1}^N \alpha_i(y_i(w \cdot \Phi(x_i) + b) - 1) \\ L_d &= \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \Phi(x_i) \Phi(x_j) \end{aligned}$$

使用同样的推导方式，让拉格朗日函数满足KKT条件，并在拉格朗日函数上对每个参数求导，经过和线性SVM相同的变换后，就可以得到拉格朗日对偶函数。同样使用梯度下降或SMO等方式对 α 进行求解，最后可以求得决策边界，并得到最终的决策函数：

$$f(x_{test}) = \text{sign}(w \cdot \Phi(x_{test}) + b) = \text{sign}\left(\sum_{i=1}^N \alpha_i y_i \Phi(x_i) \cdot \Phi(x_{test}) + b\right)$$

2.2.2 重要参数kernel

这种变换非常巧妙，但也带有一些实现问题。首先，我们可能不清楚应该什么样的数据应该使用什么类型的映射函数来确保可以在变换空间中找出线性决策边界。极端情况下，数据可能会被映射到无限维度的空间中，这种高维空间可能不是那么友好，维度越多，推导和计算的难度都会随之暴增。其次，即使已知适当的映射函数，我们想要计算类似于 $\Phi(x_i) \cdot \Phi(x_{test})$ 这样的点积，计算量可能会无比巨大，要找出超平面所付出的代价是非常昂贵的。

关键概念：核函数

而解决这些问题的数学方式，叫做“核技巧”(Kernel Trick)，是一种能够使用数据原始空间中的向量计算来表示升维后的空间中的点积结果的数学方式。具体表现为， $K(u, v) = \Phi(u) \cdot \Phi(v)$ 。而这个原始空间中的点积函数 $K(u, v)$ ，就被叫做“核函数”(Kernel Function)。

核函数能够帮助我们解决三个问题：

第一，有了核函数之后，我们无需去担心 Φ 究竟应该是什么样，因为非线性SVM中的核函数都是正定核函数(positive definite kernel functions)，他们都满足美世定律(Mercer's theorem)，确保了高维空间中任意两个向量的点积一定可以被低维空间中的这两个向量的某种计算来表示（多数时候是点积的某种变换）。

第二，使用核函数计算低维度中的向量关系比计算原本的 $\Phi(x_i) \cdot \Phi(x_{test})$ 要简单太多了。

第三，因为计算是在原始空间中进行，所以避免了维度诅咒的问题。

选用不同的核函数，就可以解决不同数据分布下的寻找超平面问题。在SVC中，这个功能由参数“kernel”和一系列与核函数相关的参数来进行控制。之前的代码中我们一直使用这个参数并输入“linear”，但却没有给大家详细讲解，也是因为如果不先理解核函数本身，很难说明这个参数到底在做什么。参数“kernel”在sklearn中可选以下几种选项：

输入	含义	解决问题	核函数的表达式	参数 gamma	参数 degree	参数 coef0
“linear”	线性核	线性	$K(x, y) = x^T y = x \cdot y$	No	No	No
“poly”	多项式核	偏线性	$K(x, y) = (\gamma(x \cdot y) + r)^d$	Yes	Yes	Yes
“sigmoid”	双曲正切核	非线性	$K(x, y) = \tanh(\gamma(x \cdot y) + r)$	Yes	No	Yes
“rbf”	高斯径向基	偏非线性	$K(x, y) = e^{-\gamma \ x-y\ ^2}, \gamma > 0$	Yes	No	No

可以看出，除了选项“linear”之外，其他核函数都可以处理非线性问题。多项式核函数有次数d，当d为1的时候它就是再处理线性问题，当d为更高次项的时候它就是在处理非线性问题。我们之前画图时使用的是选项“linear”，自然不能处理环形数据这样非线性的状况。而刚才我们使用的计算r的方法，其实是高斯径向基核函数所对应的功能，在参数“kernel”中输入“rbf”就可以使用这种核函数。我们来看看模型找出的决策边界时什么样：

```
clf = SVC(kernel = "rbf").fit(X,y)
plt.scatter(X[:,0],X[:,1],c=y,s=50,cmap="rainbow")
plot_svc_decision_function(clf)
```

可以看到，决策边界被完美地找了出来。

2.2.3 探索核函数在不同数据集上的表现

除了“linear”以外的核函数都能够处理非线性情况，那究竟什么时候选择哪一个核函数呢？遗憾的是，关于核函数在不同数据集上的研究甚少，谷歌学术上的论文中也没有几篇是研究核函数在SVM中的运用的，更多的是关于核函数在深度学习，神经网络中如何使用。在sklearn中，也没有提供任何关于如何选取核函数的信息。

但无论如何，我们还是可以通过在不同的核函数中循环去找寻最佳的核函数来对核函数进行一个选取。接下来我们就通过一个例子，来探索一下不同数据集上核函数的表现。我们现在有一系列线性或非线性可分的数据，我们希望通过绘制SVC在不同核函数下的决策边界并计算SVC在不同核函数下分类准确率来观察核函数的效用。

1. 导入所需要的库和模块

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import svm
from sklearn.datasets import make_circles, make_moons, make_blobs, make_classification
```

2. 创建数据集，定义核函数的选择

```

n_samples = 100

datasets = [
    make_moons(n_samples=n_samples, noise=0.2, random_state=0),
    make_circles(n_samples=n_samples, noise=0.2, factor=0.5, random_state=1),
    make_blobs(n_samples=n_samples, centers=2, random_state=5),
    make_classification(n_samples=n_samples, n_features =
2, n_informative=2, n_redundant=0, random_state=5)
]

Kernel = ["linear", "poly", "rbf", "sigmoid"]

#四个数据集分别是什么样子呢？
for X, Y in datasets:
    plt.figure(figsize=(5, 4))
    plt.scatter(X[:, 0], X[:, 1], c=Y, s=50, cmap="rainbow")

```

我们总共有四个数据集，四种核函数，我们希望观察每种数据集下每个核函数的表现。以核函数为列，以图像分布为行，我们总共需要16个子图来展示分类结果。而同时，我们还希望观察图像本身的状况，所以我们总共需要20个子图，其中第一列是原始图像分布，后面四列分别是这种分布下不同核函数的表现。

3. 构建子图

```

nrows=len(datasets)
ncols=len(Kernel) + 1

fig, axes = plt.subplots(nrows, ncols, figsize=(20, 16))

```

4. 开始进行子图循环

```

#第一层循环：在不同的数据集中循环
for ds_cnt, (X, Y) in enumerate(datasets):

    #在图像中的第一列，放置原数据的分布
    ax = axes[ds_cnt, 0]
    if ds_cnt == 0:
        ax.set_title("Input data")
    ax.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=plt.cm.Paired, edgecolors='k')
    ax.set_xticks(())
    ax.set_yticks(())

    #第二层循环：在不同的核函数中循环
    #从图像的第二列开始，一个个填充分类结果
    for est_idx, kernel in enumerate(Kernel):

        #定义子图位置
        ax = axes[ds_cnt, est_idx + 1]

        #建模
        clf = svm.SVC(kernel=kernel, gamma=2).fit(X, Y)

```

```

score = clf.score(X, Y)

#绘制图像本身分布的散点图
ax.scatter(X[:, 0], X[:, 1], c=Y
           , zorder=10
           , cmap=plt.cm.Paired, edgecolors='k')
#绘制支持向量
ax.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=50,
           facecolors='none', zorder=10, edgecolors='k')

#绘制决策边界
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

#np.mgrid, 合并了我们之前使用的np.linspace和np.meshgrid的用法
#一次性使用最大值和最小值来生成网格
#表示为[起始值: 结束值: 步长]
#如果步长是复数, 则其整数部分就是起始值和结束值之间创建的点的数量, 并且结束值被包含在内
XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
#np.c_, 类似于np.vstack的功能
Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()]).reshape(XX.shape)
#填充等高线不同区域的颜色
ax.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)
#绘制等高线
ax.contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '--', '--'],
           levels=[-1, 0, 1])

#设定坐标轴为不显示
ax.set_xticks(())
ax.set_yticks(())

#将标题放在第一行的顶上
if ds_cnt == 0:
    ax.set_title(kernel)

#为每张图添加分类的分数
ax.text(0.95, 0.06, ('%.2f' % score).lstrip('0')
        , size=15
        , bbox=dict(boxstyle='round', alpha=0.8, facecolor='white')
        #为分数添加一个白色的格子作为底色
        , transform=ax.transAxes #确定文字所对应的坐标轴, 就是ax子图的坐标轴本身
        , horizontalalignment='right' #位于坐标轴的什么方向
        )

plt.tight_layout()
plt.show()

```

可以观察到, 线性核函数和多项式核函数在非线性数据上表现会浮动, 如果数据相对线性可分, 则表现不错, 如果是像环形数据那样彻底不可分的, 则表现糟糕。在线性数据集上, 线性核函数和多项式核函数即便有扰动项也可以表现不错, 可见多项式核函数是虽然也可以处理非线性情况, 但更偏向于线性的功能。

Sigmoid核函数就比较尴尬了, 它在非线性数据上强于两个线性核函数, 但效果明显不如rbf, 它在线性数据上完全比不上线性的核函数们, 对扰动项的抵抗也比较弱, 所以它功能比较弱小, 很少被用到。

rbf，高斯径向基核函数基本在任何数据集上都表现不错，属于比较万能的核函数。我个人的经验是，无论如何先试试看高斯径向基核函数，它适用于核转换到很高的空间的情况，在各种情况下往往效果都很不错，如果rbf效果不好，那我们再试试看其他的核函数。另外，多项式核函数多被用于图像处理之中。

2.2.4 探索核函数的优势和缺陷

看起来，除了Sigmoid核函数，其他核函数效果都还不错。但其实rbf和poly都有自己的弊端，我们使用乳腺癌数据集作为例子来展示一下：

```
from sklearn.datasets import load_breast_cancer
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
from time import time
import datetime

data = load_breast_cancer()
X = data.data
y = data.target

X.shape
np.unique(y)

plt.scatter(X[:, 0], X[:, 1], c=y)
plt.show()

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

Kernel = ["linear", "poly", "rbf", "sigmoid"]

for kernel in Kernel:
    time0 = time()
    clf = SVC(kernel=kernel,
               gamma="auto",
               degree=1,
               cache_size=5000
               ).fit(Xtrain, Ytrain)
    print("The accuracy under kernel %s is %f" % (kernel, clf.score(Xtest, Ytest)))
    print(datetime.datetime.fromtimestamp(time() - time0).strftime("%M:%S:%f"))
```

然后我们发现，怎么跑都跑不出来。模型一直停留在线性核函数之后，就没有再打印结果了。这证明，多项式核函数此时此刻要消耗大量的时间，运算非常的缓慢。让我们在循环中去掉多项式核函数，再试试看能否跑出结果：

```

Kernel = ["linear", "rbf", "sigmoid"]

for kernel in Kernel:
    time0 = time()
    clf = SVC(kernel = kernel
               , gamma="auto"
               # , degree = 1
               , cache_size=5000
               ).fit(Xtrain, Ytrain)
    print("The accuracy under kernel %s is %f" % (kernel,clf.score(Xtest,Ytest)))
    print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S:%f"))

```

我们可以有两个发现。首先，乳腺癌数据集是一个线性数据集，线性核函数跑出来的效果很好。rbf和sigmoid两个擅长非线性的数据从效果上来看完全不可用。其次，线性核函数的运行速度远远不如非线性的两个核函数。

如果数据是线性的，那如果我们把degree参数调整为1，多项式核函数应该也可以得到不错的结果：

```

Kernel = ["linear", "poly", "rbf", "sigmoid"]

for kernel in Kernel:
    time0 = time()
    clf = SVC(kernel = kernel
               , gamma="auto"
               , degree = 1
               , cache_size=5000
               ).fit(Xtrain, Ytrain)
    print("The accuracy under kernel %s is %f" % (kernel,clf.score(Xtest,Ytest)))
    print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S:%f"))

```

多项式核函数的运行速度立刻加快了，并且精度也提升到了接近线性核函数的水平，可喜可贺。但是，我们之前的实验中，我们了解说，rbf在线性数据上也可以表现得非常好，那在这里，为什么跑出来的结果如此糟糕呢？

其实，这里真正的问题是数据的量纲问题。回忆一下我们如何求解决策边界，如何判断点是否在决策边界的一边？是靠计算“距离”，虽然我们不能说SVM是完全的距离类模型，但是它严重受到数据量纲的影响。让我们来探索一下乳腺癌数据集的量纲：

```

import pandas as pd
data = pd.DataFrame(X)
data.describe([0.01,0.05,0.1,0.25,0.5,0.75,0.9,0.99]).T

```

一眼望去，果然数据存在严重的量纲不一的问题。我们来使用数据预处理中的标准化的类，对数据进行标准化：

```

from sklearn.preprocessing import StandardScaler
X = StandardScaler().fit_transform(X)
data = pd.DataFrame(X)
data.describe([0.01,0.05,0.1,0.25,0.5,0.75,0.9,0.99]).T

```

标准化完毕后，再次让SVC在核函数中遍历，此时我们把degree的数值设定为1，观察各个核函数在去量纲后的数据上的表现：

```

xtrain, xtest, Ytrain, Ytest = train_test_split(x,y,test_size=0.3,random_state=420)

kernel = ["linear","poly","rbf","sigmoid"]

for kernel in Kernel:
    time0 = time()
    clf= SVC(kernel = kernel
              , gamma="auto"
              , degree = 1
              , cache_size=5000
              ).fit(Xtrain,Ytrain)
    print("The accuracy under kernel %s is %f" % (kernel,clf.score(Xtest,Ytest)))
    print(datetime.datetime.fromtimestamp(time()-time0).strftime("%M:%S:%f"))

```

量纲统一之后，可以观察到，所有核函数的运算时间都大大地减少了，尤其是对于线性核来说，而多项式核函数居然变成了计算最快的。其次，rbf表现出了非常优秀的结果。经过我们的探索，我们可以得到的结论是：

1. 线性核，尤其是多项式核函数在高次项时计算非常缓慢
2. rbf和多项式核函数都不擅长处理量纲不统一的数据集

幸运的是，这两个缺点都可以由数据无量纲化来解决。因此，**SVM执行之前，非常推荐先进行数据的无量纲化！**到了这一步，我们是否已经完成建模了呢？虽然线性核函数的效果是最好的，但它是没有核函数相关参数可以调整的，rbf和多项式却还有着可以调整的相关参数，接下来我们就来看看这些参数。

2.2.5 选取与核函数相关的参数：degree & gamma & coef0

输入	含义	解决问题	核函数的表达式	参数 gamma	参数 degree	参数 coef0
"linear"	线性核	线性	$K(x, y) = x^T y = x \cdot y$	No	No	No
"poly"	多项式核	偏线性	$K(x, y) = (\gamma(x \cdot y) + r)^d$	Yes	Yes	Yes
"sigmoid"	双曲正切核	非线性	$K(x, y) = \tanh(\gamma(x \cdot y) + r)$	Yes	No	Yes
"rbf"	高斯径向基	偏非线性	$K(x, y) = e^{-\gamma \ x-y\ ^2}, \gamma > 0$	Yes	No	No

在知道如何选取核函数后，我们还要观察一下除了kernel之外的核函数相关的参数。对于线性核函数，"kernel"是唯一能够影响它的参数，但是对于其他三种非线性核函数，他们还受到参数gamma, degree以及coef0的影响。参数gamma就是表达式中的 γ , degree就是多项式核函数的次数 d , 参数coef0就是常数项 r 。其中，高斯径向基核函数受到gamma的影响，而多项式核函数受到全部三个参数的影响。

参数	含义
degree	整数, 可不填, 默认3 多项式核函数的次数 ('poly') , 如果核函数没有选择"poly", 这个参数会被忽略
gamma	浮点数, 可不填, 默认"auto" 核函数的系数, 仅在参数Kernel的选项为"rbf","poly"和"sigmoid"的时候有效 输入"auto", 自动使用 $1/(n_features)$ 作为gamma的取值 输入"scale", 则使用 $1/(n_features * X.std())$ 作为gamma的取值 输入"auto_deprecated", 则表示没有传递明确的gamma值 (不推荐使用)
coef0	浮点数, 可不填, 默认=0.0 核函数中的常数项, 它只在参数kernel为'poly'和'sigmoid'的时候有效。

但从核函数的公式来看, 我们其实很难去界定具体每个参数如何影响了SVM的表现。当gamma的符号变化, 或者degree的大小变化时, 核函数本身甚至都不是永远单调的。所以如果我们想要彻底地理解这三个参数, 我们要先推导出它们如何影响核函数地变化, 再找出核函数的变化如何影响了我们的预测函数 (可能改变我们的核变化所在的维度), 再判断出决策边界随着预测函数的改变发生了怎样的变化。无论是从数学的角度来说还是从实践的角度来说, 这个过程太复杂也太低效。所以, 我们往往避免去真正探究这些参数如何影响了我们的核函数, 而直接使用学习曲线或者网格搜索来帮助我们查找最佳的参数组合。

对于高斯径向基核函数, 调整gamma的方式其实比较容易, 那就是画学习曲线。我们来试试看高斯径向基核函数rbf的参数gamma在乳腺癌数据集上的表现:

```
score = []
gamma_range = np.logspace(-10, 1, 50) #返回在对数刻度上均匀间隔的数字
for i in gamma_range:
    clf = SVC(kernel="rbf", gamma = i, cache_size=5000).fit(Xtrain, Ytrain)
    score.append(clf.score(Xtest, Ytest))

print(max(score), gamma_range[score.index(max(score))])
plt.plot(gamma_range, score)
plt.show()
```

通过学习曲线, 很容就找出了rbf的最佳gamma值。但我们观察到, 这其实与线性核函数的准确率一模一样之前的准确率。我们可以多次调整gamma_range来观察结果, 可以发现97.6608应该是rbf核函数的极限了。

但对于多项式核函数来说, 一切就没有那么容易了, 因为三个参数共同作用在一个数学公式上影响它的效果, 因此我们往往使用网格搜索来共同调整三个对多项式核函数有影响的参数。依然使用乳腺癌数据集。

```
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.model_selection import GridSearchCV

time0 = time()

gamma_range = np.logspace(-10, 1, 20)
coef0_range = np.linspace(0, 5, 10)

param_grid = dict(gamma = gamma_range
                  , coef0 = coef0_range)
```

```

cv = StratifiedShuffleSplit(n_splits=5, test_size=0.3, random_state=420)
grid = GridSearchCV(SVC(kernel = "poly", degree=1, cache_size=5000),
param_grid=param_grid, cv=cv)
grid.fit(X, y)

print("The best parameters are %s with a score of %0.5f" % (grid.best_params_,
grid.best_score_))
print(datetime.datetime.fromtimestamp(time() - time0).strftime("%M:%S:%f"))

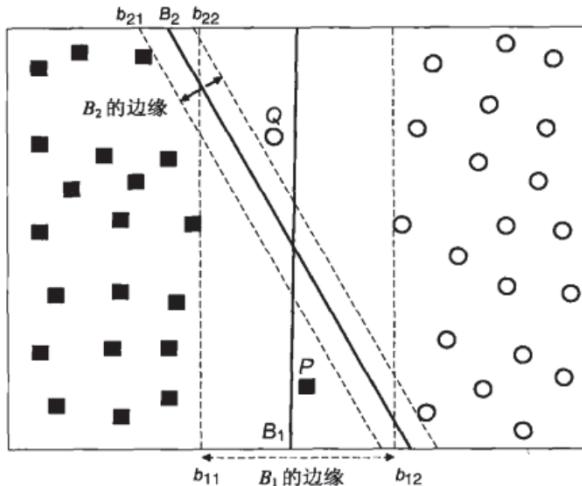
```

可以发现，网格搜索为我们返回了参数 $\text{coef}_0=0$, $\text{gamma}=0.18329807108324375$, 但整体的分数是0.96959, 虽然比调参前略有提高, 但依然没有超过线性核函数核rbf的结果。可见, 如果最初选择核函数的时候, 你就发现多项式的结果不如rbf和线性核函数, 那就不要挣扎了, 试试看调整rbf或者直接使用线性。

2.3 硬间隔与软间隔：重要参数C

2.3.1 SVM在软间隔数据上的推广

到这里, 我们已经了解了线性SVC的基本原理, 以及SVM如何被推广到非线性情况下, 还了解了核函数的选择和应用。但实际上, 我们依然没有完全了解sklearn当中的SVM用于二分类的全貌。我们之前在理论推导中使用的数据都有一个特点, 那就是他们或是完全线性可分, 或者是非线性的数据。在我们对比核函数时, 实际上用到了一种不同的数据, 那就是不完全线性可分的数据集。比如说如下数据集:

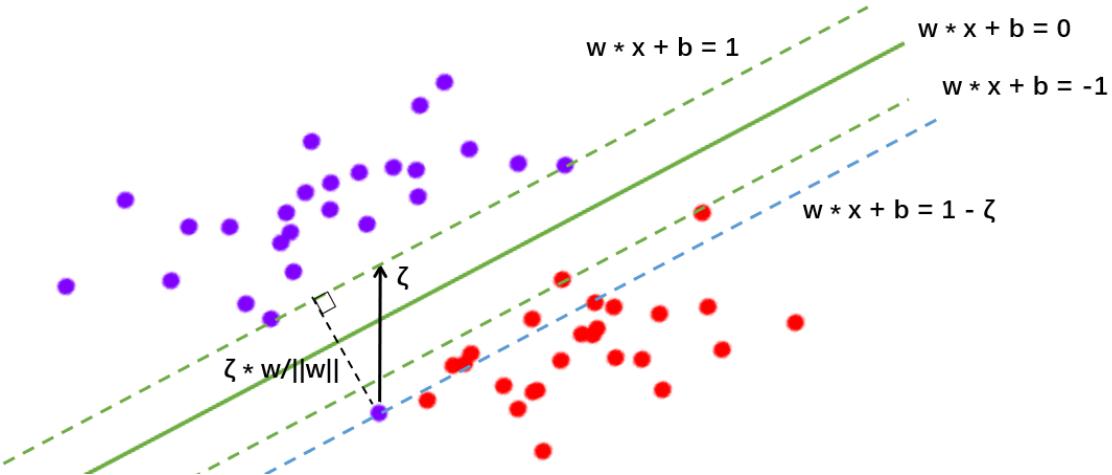


这个数据集和我们最开始介绍SVM如何工作的时候的数据集一模一样, 除了多了P和Q两个点。我们注意到, 虽然决策边界 B_1 的间隔已经非常宽了, 然而点P和Q依然被分错了类别, 相反, 边界比较小的 B_2 却正确地分出了点P和Q的类别。这里并不是说 B_2 此时此刻就是一条更好的边界了, 与之前的论述中一致, 如果我们引入更多的训练数据, 或引入测试数据, B_1 更加宽敞的边界可以帮助它有更好的表现。但是, 和之前不一样, 现在即便是让边际最大的决策边界 B_1 的训练误差也不可能为0了。此时, 我们就需要引入“软间隔”的概念:

关键概念：硬间隔与软间隔

当两组数据是完全线性可分, 我们可以找出一个决策边界使得训练集上的分类误差为0, 这两种数据就被称为是存在“硬间隔”的。当两组数据几乎是完全线性可分的, 但决策边界在训练集上存在较小的训练误差, 这两种数据就被称为是存在“软间隔”。

我们可以通过调整我们对决策边界的定义，将硬间隔时得出的数学结论推广到软间隔的情况下，让决策边界能够忍受一小部分训练误差。这个时候，我们的决策边界就不是单纯地寻求最大边际了，因为对于软间隔地数据来说，边际越大被分错的样本也就会越多，因此我们需要找出一个“最大边际”与“被分错的样本数量”之间的平衡。



看上图，原始的决策边界 $w \cdot x + b = 0$ ，原本的平行于决策边界的两个虚线超平面 $w \cdot x + b = 1$ 和 $w \cdot x + b = -1$ 都依然有效。我们的原始判别函数为：

$$\begin{aligned} w \cdot x_i + b &\geq 1 \quad \text{if } y_i = 1 \\ w \cdot x_i + b &\leq -1 \quad \text{if } y_i = -1 \end{aligned}$$

不过，这些超平面现在无法让数据上的训练误差等于0了，因为此时存在了一个混杂在红色点中的紫色点 x_p 。于是，我们需要放松我们原始判别函数中的不等条件，来让决策边界能够适用于我们的异常点，于是我们引入松弛系数 ζ 来帮助我们优化原始的判别函数：

$$\begin{aligned} w \cdot x_i + b &\geq 1 - \zeta_i \quad \text{if } y_i = 1 \\ w \cdot x_i + b &\leq -1 + \zeta_i \quad \text{if } y_i = -1 \end{aligned}$$

其中 $\zeta_i > 0$ 。可以看得出，这其实是将原本的虚线面向图像的上方和下方平移，其符号的处理方式和我们原本讲解过的把符号放入 w 是一模一样的方式。松弛系数其实很好理解，来看上面的图像。位于红色点附近的紫色点 x_p 在原本的判别函数中必定会被分为红色，所以一定会被判断错。现在我们作一条与决策边界平行，但是过点 x_p 的直线 $w \cdot x_i + b = 1 - \zeta_i$ （图中的蓝色虚线）。这条直线是由 $w \cdot x_i + b = 1$ 平移得到，所以两条直线在纵坐标上的差异就是 ζ （竖直的黑色箭头）。而点 x_p 到 $w \cdot x_i + b = 1$ 的距离就可以表示为 $\frac{\zeta \cdot w}{\|w\|}$ ，即 ζ 在 w 方向上的投影。由于单位向量是固定的，所以 ζ 可以作为点 x_p 在原始的决策边界上的分类错误的程度的表示，隔得越远，分得越错。但注意， ζ 并不是点到决策超平面的距离本身。

不难注意到，我们让 $w \cdot x_i + b \geq 1 - \zeta_i$ 作为我们的新决策超平面，是由一定的问题的，虽然我们把异常的紫色点分类正确了，但我们同时也分错了一系列红色的点。所以我们必须在我们求解最大边际的损失函数中加上一个惩罚项，用来惩罚我们具有巨大松弛系数的决策超平面。我们的拉格朗日函数，拉格朗日对偶函数，也因此都被松弛系数改变。现在，我们的损失函数为：

$$\begin{aligned} \min_{w,b,\zeta} \quad & \frac{\|w\|^2}{2} + C \sum_{i=1}^n \zeta_i \\ \text{subject to} \quad & y_i(w \cdot \Phi(x_i) + b) \geq 1 - \zeta_i, \\ & \zeta_i \geq 0, \\ & i = 1, 2, \dots, N \end{aligned}$$

其中C是用来控制惩罚项的惩罚力度的系数。

我们的拉格朗日函数为（其中 μ 是第二个拉格朗日乘数）：

$$L(w, b, \alpha, \zeta) = \frac{1}{2} \|w\|^2 + C \sum_{i=1}^N \zeta_i - \sum_{i=1}^N \alpha_i (y_i (\mathbf{w} \cdot \Phi(\mathbf{x}_i) + b) - 1 + \zeta_i) - \sum_{i=1}^N \mu_i \zeta_i$$

需要满足的KKT条件为：

$$\begin{aligned}\frac{\partial L(w, b, \alpha, \zeta)}{\partial w} &= \frac{\partial L(w, b, \alpha, \zeta)}{\partial b} = \frac{\partial L(w, b, \alpha, \zeta)}{\partial \zeta} = 0 \\ \zeta_i &\geq 0, \alpha_i \geq 0, \mu_i \geq 0 \\ \alpha_i (y_i (\mathbf{w} \cdot \Phi(\mathbf{x}_i) + b) - 1 + \zeta_i) &= 0 \\ \mu_i \zeta_i &= 0\end{aligned}$$

拉格朗日对偶函数为：

$$\begin{aligned}L_D &= \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \Phi(\mathbf{x}_i) \Phi(\mathbf{x}_j) \\ \text{subject to } C &\geq \alpha_i \geq 0\end{aligned}$$

这种状况下的拉格朗日对偶函数看起来和线性可分状况下的对偶函数一模一样，但是需要注意的是，在这个函数中，拉格朗日乘数 α 的取值的限制改变了。在硬间隔的状况下，拉格朗日乘数值需要大于等于0，而现在它被要求不能够大于用来控制惩罚项的惩罚力度的系数C。有了对偶函数之后，我们的求解过程和硬间隔下的步骤一致。以上所有的公式，是以线性硬间隔数据为基础，考虑了软间隔存在的情况和数据是非线性的状况而得来的。而这些公式，就是sklearn类SVC背后使用的最终公式。公式中现在唯一的新变量，松弛系数的惩罚力度C，由我们的参数C来进行控制。

2.3.2 重要参数C

参数C用于权衡“训练样本的正确分类”与“决策函数的边际最大化”两个不可同时完成的目标，希望找出一个平衡点来让模型的效果最佳。

参数	含义
C	浮点数，默认1，必须大于等于0，可不填 松弛系数的惩罚项系数。如果C值设定比较大，那SVC可能会选择边际较小的，能够更好地分类所有训练点的决策边界，不过模型的训练时间也会更长。如果C的设定值较小，那SVC会尽量最大化边界，决策功能会更简单，但代价是训练的准确度。换句话说，C在SVM中的影响就像正则化参数对逻辑回归的影响。

在实际使用中，C和核函数的相关参数（gamma, degree等等）们搭配，往往是SVM调参的重点。与gamma不同，C没有在对偶函数中出现，并且是明确了调参目标的，所以我们可以明确我们究竟是否需要训练集上的高精确度来调整C的方向。默认情况下C为1，通常来说这都是一个合理的参数。如果我们的数据很嘈杂，那我们往往减小C。当然，我们也可以使用网格搜索或者学习曲线来调整C的值。

```
#调线性核函数
score = []
C_range = np.linspace(0.01, 30, 50)
for i in C_range:
    clf = SVC(kernel="linear", C=i, cache_size=5000).fit(xtrain, ytrain)
    score.append(clf.score(xtest, ytest))
```

```

print(max(score), C_range[score.index(max(score))])
plt.plot(C_range, score)
plt.show()

#换rbf
score = []
C_range = np.linspace(0.01, 30, 50)
for i in C_range:
    clf = SVC(kernel="rbf", C=i, gamma =
0.012742749857031322, cache_size=5000).fit(Xtrain, Ytrain)
    score.append(clf.score(Xtest, Ytest))

print(max(score), C_range[score.index(max(score))])
plt.plot(C_range, score)
plt.show()

#进一步细化
score = []
C_range = np.linspace(5, 7, 50)
for i in C_range:
    clf = SVC(kernel="rbf", C=i, gamma =
0.012742749857031322, cache_size=5000).fit(Xtrain, Ytrain)
    score.append(clf.score(Xtest, Ytest))

print(max(score), C_range[score.index(max(score))])
plt.plot(C_range, score)
plt.show()

```

此时，我们找到了乳腺癌数据集上的最优解：rbf核函数下的98.24%的准确率。当然，我们还可以使用交叉验证来改进我们的模型，获得不同测试集和训练集上的交叉验证结果。但上述过程，为大家展现了如何选择正确的核函数，以及如何调整核函数的参数，过程虽然简单，但是希望可以对大家有所启发。

2.4 总结

本周，我们讲解了支持向量机的原理，向大家介绍了支持向量机的损失函数，拉格朗日函数，拉格朗日对偶函数，预测函数以及这些函数在非线性，软间隔这些情况上的推广。我们介绍了四种核函数，包括它们的特点，适合什么样的数据，有什么相关参数，优缺点，以及什么时候使用。最后我们还讲解了核函数在相关参数上的调参。本节课的内容非常多，但我们对SVM的探索还远远没有结束。下一周我们会继续讲解SVM，包括SVC的模型评价指标，SVC使用中的其他重要参数，重要属性和接口，以及其他重要问题，当然还有案例。希望大家尽力消化这一周的内容，SVC的探索道路依然任重而道远。

3 附录

3.1 SVC的参数列表

C	浮点数, 可不填, 默认1.0 松弛系数的惩罚项系数。如果C值设定比较大, 那SVC可能会选择边际较小的, 能够更好地分类所有训练点的决策边界。如果C的设定值较小, 那SVC会尽量最大化边界, 决策功能会更简单, 但代价是训练的准确度。换句话说, C在SVM中的影响就像正则化参数对逻辑回归的影响。
kernel	字符, 可不填, 默认 "rbf" 指定要在算法中使用的核函数类型, 可以输入 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'或者可调用对象 (如函数, 类等)。如果给出可调用对象, 则这个对象将被用于从特征矩阵X预先计算内核矩阵; 该矩阵应该是是一个 (n_samples, n_samples) 结构的数组。
degree	整数, 可不填, 默认3 多项式核函数的次数 ('poly') , 如果核函数没有选择"poly", 这个参数会被忽略。
gamma	浮点数, 可不填, 默认 "auto" 核函数的系数, 仅在参数Kernel的选项为" rbf", "poly"和"sigmoid" 的时候有效。 当输入 "auto", 自动使用 $1/(n_features)$ 作为gamma的取值。 在sklearn0.22版本中, 将可输入 "scale", 则使用 $1/(n_features * X.std())$ 作为gamma的取值。 若输入 "auto_deprecated", 则表示没有传递明确的gamma值 (不推荐使用) 。
coef0	浮点数, 可不填, 默认=0.0 核函数中的独立项, 它只在参数kernel为'poly'和'sigmoid'的时候有效。
shrinking	布尔值, 可不填, 默认True 是否使用收缩启发式计算(shrinking heuristics), 如果使用, 有时可以加速最优化的计算进程, 加速迭代速度。 P.S. 此参数的相关信息和讨论甚少, 参数背后的真实面貌必须要阅读源码才能够解读, 在解读之前大家谨慎使用。
probability	布尔值, 可不填, 默认False 是否启用概率估计。必须在调用fit之前启用它, 启用此功能会减慢SVM的运算速度。
tol	浮点数, 可不填, 默认 $1e-3$ 停止迭代的容差。
cache_size	浮点数, 可不填, 默认200 指定核函数占用的缓存的大小 (以MB为单位)
class_weight	字典, "balanced" , 可不填 对SVC, 将类i的参数C设置为class_weight [i] * C。如果没有给出具体的class_weight, 则所有类都占用相同的权重1。 "Balanced" 模式使用y的值自动调整与输入数据中的类频率成反比的权重为 $n_samples/(n_classes * np.bincount(y))$
verbose	布尔值, 默认False 启用详细输出。请注意, 此设置利用libsvm中的 "进程前运行时间设置" 。如果启用, 则可能无法在多线程上下文中正常运行。
max_iter	整数, 可不填, 默认=-1 最大迭代次数, 输入 "-1" 表示没有限制。

decision_function_shape	可输入 "ovo", "ovr", 默认" ovr" 对所有分类器，是否返回结构为(n_samples, n_classes)的one-rest-rest('ovr')决策函数，或者返回libsvm中原始的，结构为(n_samples), n_classes * (n_classes - 1) / 2)的one-vs-one('ovo')决策函数。但是，一对一('ovo')总是在多分类问题中才使用。 在版本0.19中更改：decision_function_shape默认为'ovr'。 版本0.17中的新功能：建议使用decision_function_shape = 'ovr'。 更改版本0.17：已弃用decision_function_shape = 'ovo'和None。
random_state	整数，随机数种子，None，可不填，默认None 在对数据进行混洗以用于概率估计时使用的伪随机数种子生成器。如果输入整数，则random_state是随机数生成器使用的随机数种子；如果是RandomState实例，则random_state是随机数生成器；如果为None，则随机数生成器是np.random使用的RandomState实例。

3.2 SVC的属性列表

属性	形态	含义
<code>support_</code>	类数组, 结构 = [n_SV]	支持向量的索引
<code>support_vectors_</code>	类数组, 结构 = [n_SV, n_features]	支持向量本身
<code>n_support_</code>	类数组, 类型为int32, 结构 = [n_class]	每个类的支持向量数
<code>dual_coef_</code>	数组, 结构 = [n_class-1, n_SV]	决策函数中支持向量的系数。对于多分类问题而言, 是所有ovo模式分类器中的系数。多类情况下系数的布局不是非常重要。
<code>coef_</code>	数组, 结构 = [n_class * (n_class-1) / 2, n_features]	赋予特征的权重 (原始问题中的系数), 这个属性仅适用于线性内核。 <code>coef_</code> 是从 <code>dual_coef_</code> 和 <code>support_vectors_</code> 派生的只读属性。
<code>intercept_</code>	数组, 结构 = [n_class * (n_class-1) / 2]	决策函数中的常量, 在二维平面上是截距。
<code>fit_status_</code>	整数	如果正确拟合, 则显示0, 否则为1 (将发出警告)

3.3 SVC的接口列表

接口	输入	含义
<code>decision_function</code>	特征矩阵X	返回X中每个样本到划分数据集的超平面的距离, 结构为(n_samples,)。如果 <code>decision_function_shape = 'ovr'</code> , 则返回的矩阵结构为(n_samples, n_classes)。
<code>fit</code>	特征矩阵X, 真实标签[, sample_weight]	根据给定的训练数据拟合SVM模型 如果X和y不是C顺序的, 并且np.float64和X的连续数组不是scipy.sparse.csr_matrix, 则X和/或y可能会被复制。 如果X是密集数组, 那么其他接口都将不支持稀疏矩阵作为输入。
<code>predict</code>	测试集的矩阵X	对X中的样本进行分类, 返回的结构为(n_samples,)
<code>score</code>	特征矩阵X, 真实标签[, sample_weight]	返回给定测试数据和标签的平均准确度 在多标签分类中, 这是子集精度。子集精度是一个非常严格的度量, 因为子集精度要求为每个样本预测的每个标签都必须正确。
<code>set_params</code>	想要替换的新参数	设置此估算器的参数
<code>get_params</code>	无需输入	获取此估算工具的参数
<code>predict_log_proba</code>	特征矩阵X	计算X中样本在所有可能分到的类下的对数概率。每个列按所有类出现在属性classes_中的顺序排列。 模型需要在训练时计算概率信息: probability参数必须被设置为True。 概率模型是使用交叉验证创建的, 因此结果可能与通过预测获得的结果略有不同。此外, 它可能在非常小的数据集上产生无意义的结果。
<code>predict_proba</code>	特征矩阵X	计算X中样本在所有可能分到的类下的概率。每个列按所有类出现在属性classes_中的顺序排列。 模型需要在训练时计算概率信息: probability参数必须被设置为True。 概率模型是使用交叉验证创建的, 因此结果可能与通过预测获得的结果略有不同。此外, 它可能在非常小的数据集上产生无意义的结果。

菜菜的scikit-learn课堂08



sklearn中的支持向量机SVM（下）

小伙伴们晚上好~o(—▽—)ブ

我是菜菜，这里是我的sklearn课堂第八期，今晚的直播内容是支持向量机（下）！

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scikit-learn课堂08

sklearn中的支持向量机SVM（下）

1 二分类SVC的进阶

- 1.1 SVC用于二分类的原理复习
- 1.2 参数C的理解进阶
- 1.3 二分类SVC中的样本不均衡问题：重要参数class_weight

2 SVC的模型评估指标

- 2.1 混淆矩阵（Confusion Matrix）
 - 2.1.1 模型整体效果：准确率
 - 2.1.2 捕捉少数类的艺术：精确度，召回率和F1 score
 - 2.1.3 判错多数类的考量：特异度与假正率
 - 2.1.4 sklearn中的混淆矩阵
- 2.2 ROC曲线以及其相关问题
 - 2.2.1 概率(probability)与阈值(threshold)
 - 2.2.2 SVM实现概率预测：重要参数probability，接口predict_proba以及decision_function
 - 2.2.3 绘制SVM的ROC曲线
 - 2.2.4 sklearn中的ROC曲线和AUC面积
 - 2.2.5 利用ROC曲线找出最佳阈值

3 使用SVC时的其他考虑

- 3.1 SVC处理多分类问题：重要参数decision_function_shape
- 3.2 SVM的模型复杂度
- 3.3 SVM中的随机性：参数random_state
- 3.4 SVC的重要属性补充
- 3.5 一窥线性支持向量机类LinearSVC

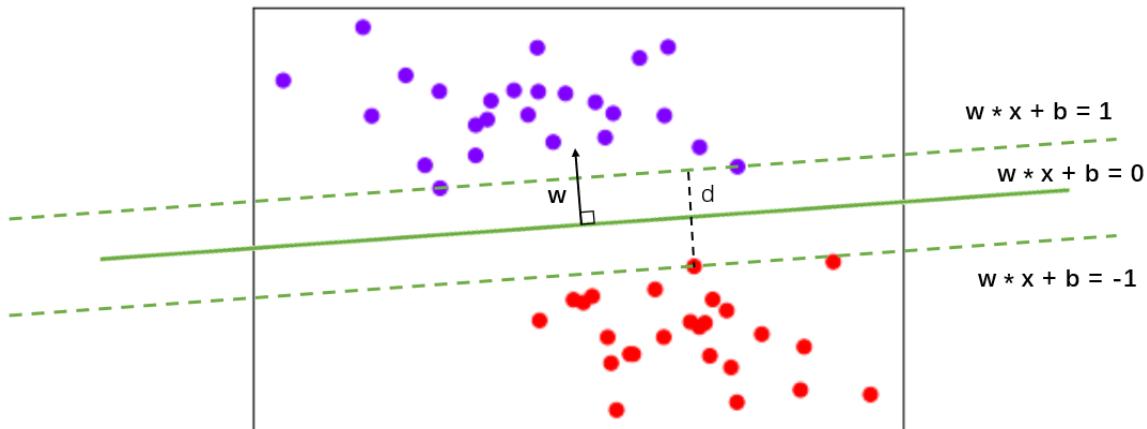
4 SVC真实数据案例：预测明天是否会下雨

- 4.1 导入库导数据，探索特征
- 4.2 分集，优先探索标签
- 4.3 探索特征，开始处理特征矩阵
 - 4.3.1 描述性统计与异常值
 - 4.3.2 处理困难特征：日期
 - 4.3.3 处理困难特征：地点
 - 4.3.4 处理分类型变量：缺失值
 - 4.3.5 处理分类型变量：将分类型变量编码
 - 4.3.6 处理连续型变量：填补缺失值
 - 4.3.7 处理连续型变量：无量纲化
- 4.4 建模与模型评估
- 4.5 模型调参
 - 4.5.1 最求最高Recall
 - 4.5.2 追求最高准确率
 - 4.5.3 追求平衡
- 4.6 SVM总结&结语

1 二分类SVC的进阶

1.1 SVC用于二分类的原理复习

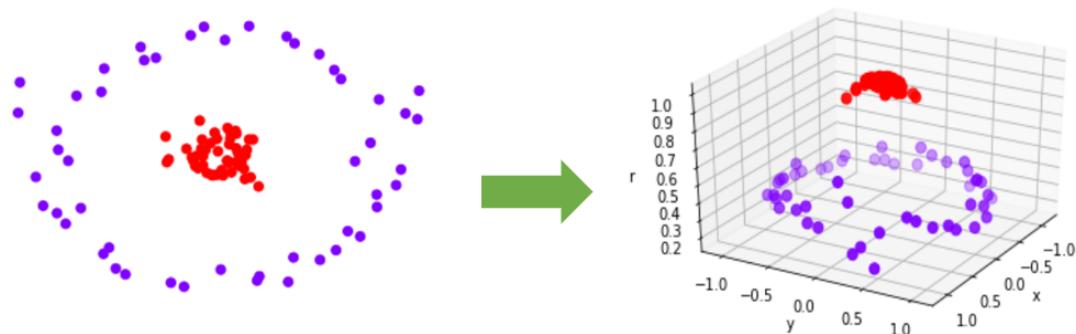
在上周的支持向量SVM（上）中，我们学习了二分类SVC的所有基本知识，包括SVM的原理，二分类SVC的损失函数，拉格朗日函数，拉格朗日对偶函数，预测函数以及这些函数在非线性，软间隔这些情况上的推广，并且引出了核函数这个关键概念。今天，基于我们已经学过的理论，我们继续探索支持向量机的其他性质，并在真实数据集上运用SVM。开始今天的探索之前，我们先来简单回忆一下支持向量机是如何工作的。



支持向量机分类器，是在数据空间中找出一个超平面作为决策边界，利用这个决策边界来对数据进行分类，并使分类误差尽量小的模型。决策边界是比所在数据空间小一维的空间，在三维数据空间中就是一个平面，在二维数据空间中就是一条直线。以二维数据为例，图中的数据集有两个特征，标签有两类，一类为紫色，一类为红色。对于这组数据，我们找出的决策边界被表达为 $w \cdot x + b = 0$ ，决策边界把平面分成了上下两部分，决策边界以上的样本都分为一类，决策边界以下的样本被分为另一类。以我们的图像为例，绿色实线上部分为一类（全部都是紫色点），下部分为另一类（全部都是红色点）。

平行于决策边界的两条虚线是距离决策边界相对距离为1的超平面，他们分别压过两类样本中距离决策边界最近的样本点，这些样本点就成为支持向量。两条虚线超平面之间的距离叫做边际，简写为 d 。支持向量机分类器，就是以找出**最大化的边际 d** 为目标来求解损失函数，以求解出参数 w 和 b ，以构建决策边界，然后用决策边界来分类的分类器。

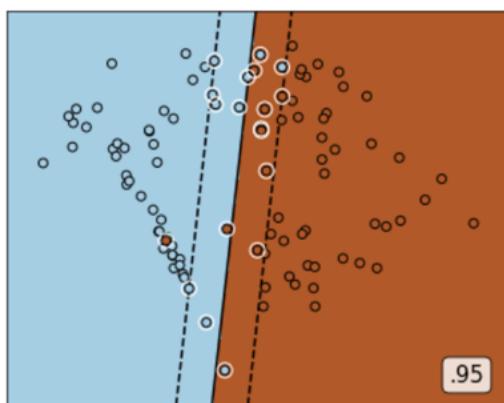
当然，不是所有数据都是线性可分的，不是所有数据我们都能够一眼看出，有一条直线，或一个平面，甚至一个超平面可以将数据完全分开。比如下面的环形数据。对于这样的数据，我们需要对它进行一个升维变化，来数据从原始的空间 x 投射到新空间 $\Phi(x)$ 中。升维之后，我们明显可以找出一个平面，能够将数据切分开来。 Φ 是一个映射函数，它代表了某种能够将数据升维的非线性的变换，我们对数据进行这样的变换，确保数据在自己的空间中一定能够线性可分。



但这种手段是有问题的，我们很难去找出一个函数 $\Phi(x)$ 来满足我们的需求，并且我们并不知道数据究竟被映射到了一个多少维度的空间当中，有可能数据被映射到了无限空间中，陷入“维度诅咒”，让我们的计算和预测都变得无比艰难。为了避免这些问题，我们使用核函数来帮助我们。核函数 $K(x, x_{test})$ 能够用原始数据空间中向量计算来表示升维后地空间中的点积 $\Phi(x) \cdot \Phi(x_{test})$ ，以帮助我们避免寻找 $\Phi(x)$ 。选用不同的核函数，就可以解决不同数据分布下的寻找超平面问题。在sklearn的SVC中，这个功能由参数“kernel”('kernel')和一系列与核函数相关的参数来进行控制，包括gamma, coef0和degree。同时，我们还讲解了软间隔和硬间隔中涉及到的参数C。今天我们就从参数C的进阶理解开始继续探索我们的支持向量机。

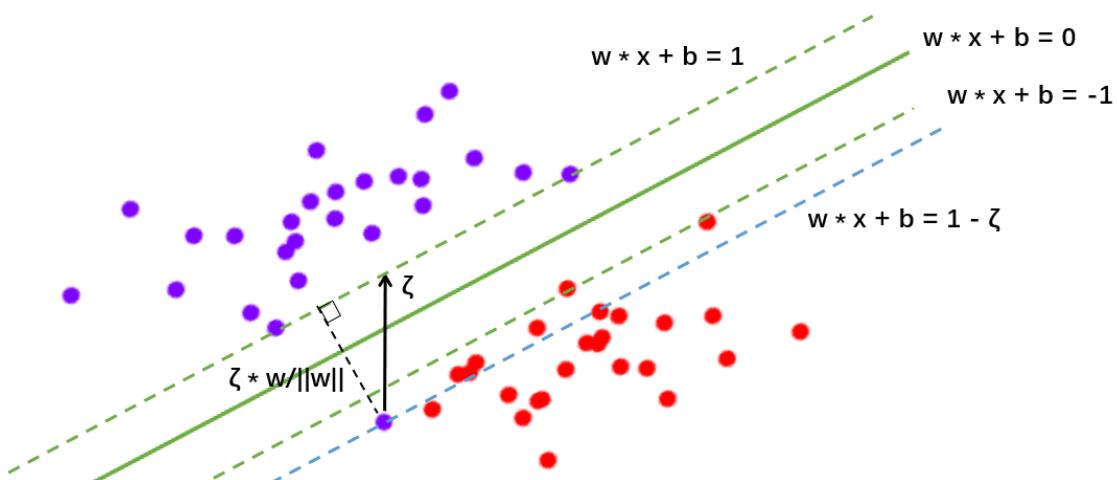
1.2 参数C的理解进阶

有一些数据，可能是线性可分，但在线性可分状况下训练准确率不能达到100%，即无法让训练误差为0，这样的数据被我们称为“存在软间隔的数据”。此时此刻，我们需要让我们决策边界能够忍受一小部分训练误差，我们就不能单纯地寻求最大边际了。



因为对于软间隔地数据来说，边际越大被分错的样本也就会越多，因此我们需要找出一个“最大边际”与“被分错的样本数量”之间的平衡。因此，我们引入松弛系数 ζ 和松弛系数的系数C作为一个惩罚项，来惩罚我们对最大边际的追求。

那我们的参数C如何影响我们的决策边界呢？在硬间隔的时候，我们的决策边界完全由两个支持向量和最小化损失函数（最大化边际）来决定，而我们的支持向量是两个标签类别不一致的点，即分别是正样本和负样本。然而在软间隔情况下我们的边际依然由支持向量决定，但此时此刻的支持向量可能就不再是来自两种标签类别的点了，而是分布在决策边界两边的，同类别的点。回忆一下我们的图像：



此时我们的虚线超平面 $\mathbf{w} \cdot \mathbf{x}_i + b = 1 - \zeta_i$ 是由混杂在红色点中间的紫色点来决定的，所以此时此刻，这个紫色点就是我们的支持向量了。所以软间隔让决定两条虚线超平面的支持向量可能是来自于同一个类别的样本点，而硬间隔的时候两条虚线超平面必须是由来自两个不同类别的支持向量决定的。而C值会决定我们究竟是依赖红色点作为支持向量（只追求最大边界），还是我们要依赖软间隔中，混杂在红色点中的紫色点来作为支持向量（追求最大边界和判断正确的平衡）。如果C值设定比较大，那SVC可能会选择边际较小的，能够更好地分类所有训练点的决策边界，不过模型的训练时间也会更长。如果C的设定值较小，那SVC会尽量最大化边界，尽量将掉落在决策边界另一方的样本点预测正确，决策功能会更简单，但代价是训练的准确度，因为此时会有更多红色的点被分类错误。换句话说，C在SVM中的影响就像正则化参数对逻辑回归的影响。

此时此刻，所有可能影响我们的超平面的样本可能都会被定义为支持向量，所以支持向量就不再是所有压在虚线超平面上的点，而是所有可能影响我们的超平面的位置的那些混杂在彼此的类别中的点了。观察一下我们对不同数据集分类时，支持向量都有哪些？软间隔如何影响了超平面和支持向量，就一目了然了。

```

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import svm
from sklearn.datasets import make_circles, make_moons, make_blobs, make_classification

n_samples = 100

datasets = [
    make_moons(n_samples=n_samples, noise=0.2, random_state=0),
    make_circles(n_samples=n_samples, noise=0.2, factor=0.5, random_state=1),
    make_blobs(n_samples=n_samples, centers=2, random_state=5),
    make_classification(n_samples=n_samples, n_features =
2, n_informative=2, n_redundant=0, random_state=5)
]

Kernel = ["linear"]

#四个数据集分别是什么样子呢?
for X,Y in datasets:
    plt.figure(figsize=(5,4))
    plt.scatter(X[:,0],X[:,1],c=Y,s=50,cmap="rainbow")

nrows=len(datasets)
ncols=len(Kernel) + 1

fig, axes = plt.subplots(nrows, ncols, figsize=(10,16))

#第一层循环: 在不同的数据集中循环
for ds_cnt, (X,Y) in enumerate(datasets):

    ax = axes[ds_cnt, 0]
    if ds_cnt == 0:
        ax.set_title("Input data")
    ax.scatter(X[:, 0], X[:, 1], c=Y, zorder=10, cmap=plt.cm.Paired, edgecolors='k')
    ax.set_xticks(())
    ax.set_yticks(())

    for est_idx, kernel in enumerate(Kernel):

```

```

ax = axes[ds_cnt, est_idx + 1]

clf = svm.SVC(kernel=kernel, gamma=2).fit(X, Y)
score = clf.score(X, Y)

ax.scatter(X[:, 0], X[:, 1], c=Y
           , zorder=10
           , cmap=plt.cm.Paired, edgecolors='k')
ax.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=100,
           facecolors='none', zorder=10, edgecolors='white')

x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()]).reshape(XX.shape)
ax.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)
ax.contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '--', '--'],
           levels=[-1, 0, 1])

ax.set_xticks(())
ax.set_yticks(())

if ds_cnt == 0:
    ax.set_title(kernel)

ax.text(0.95, 0.06, ('%.2f' % score).lstrip('0')
       , size=15
       , bbox=dict(boxstyle='round', alpha=0.8, facecolor='white')
         #为分数添加一个白色的格子作为底色
       , transform=ax.transAxes #确定文字所对应的坐标轴, 就是ax子图的坐标轴本身
       , horizontalalignment='right' #位于坐标轴的什么方向
       )

plt.tight_layout()
plt.show()

```

白色圈圈出的就是我们的支持向量，大家可以看到，所有在两条虚线超平面之间的点，和虚线超平面外，但属于另一个类别的点，都被我们认为是支持向量。并不是因为这些点都在我们的超平面上，而是因为我们的超平面由所有的这些点来决定，我们可以通过调节C来移动我们的超平面，让超平面过任何一个白色圈圈出的点。参数C就是这样影响了我们的决策，可以说是彻底改变了支持向量机的决策过程。

1.3 二分类SVC中的样本不均衡问题：重要参数class_weight

对于分类问题，永远都逃不过的一个痛点就是样本不均衡问题。样本不均衡是指在一组数据集中，标签的一类天生占有很大的比例，但我们有着捕捉出某种特定的分类的需求的状况。比如，我们现在要对潜在犯罪者和普通人进行分类，潜在犯罪者占总人口的比例是相当低的，也许只有2%左右，98%的人都是普通人，而我们的目标是要捕获出潜在犯罪者。这样的标签分布会带来许多问题。

首先，分类模型天生会倾向于多数的类，让多数类更容易被判断正确，少数类被牺牲掉。因为对于模型而言，样本量越大的标签可以学习的信息越多，算法就会更加依赖于从多数类中学到的信息来进行判断。如果我们希望捕获少数类，模型就会失败。**其次，模型评估指标会失去意义。**这种分类状况下，即便模型什么也不做，全把所有人都当成不会犯罪的人，准确率也能非常高，这使得模型评估指标accuracy变得毫无意义，根本无法达到我们的“要识别出会犯罪的人”的建模目的。

所以现在，我们首先要让算法意识到数据的标签是不均衡的，通过施加一些惩罚或者改变样本本身，来让模型向着捕获少数类的方向建模。然后，我们要改进我们的模型评估指标，使用更加针对于少数类的指标来优化模型。

要解决第一个问题，我们在逻辑回归中已经介绍了一些基本方法，比如上采样下采样。但这些采样方法会增加样本的总数，对于支持向量机这个样本总是对计算速度影响巨大的算法来说，我们完全不想轻易地增加样本数量。况且，支持向量机中决策仅仅决策边界的影响，而决策边界又仅仅受到参数C和支持向量的影响，单纯地增加样本数量不仅会增加计算时间，可能还会增加无数对决策边界无影响的样本点。因此在支持向量机中，我们要大力依赖我们调节样本均衡的参数：SVC类中的class_weight和接口fit中可以设定的sample_weight。

在逻辑回归中，参数class_weight默认None，此模式表示假设数据集中的所有标签是均衡的，即自动认为标签的比例是1:1。所以当样本不均衡的时候，我们可以使用形如{"标签的值1": 权重1, "标签的值2": 权重2}的字典来输入真实的样本标签比例，来让算法意识到样本是不平衡的。或者使用"balanced"模式，直接使用n_samples/(n_classes * np.bincount(y))作为权重，可以比较好地修正我们的样本不均衡情况。

但在SVM中，我们的分类判断是基于决策边界的，而最终决定究竟使用怎样的支持向量和决策边界的参数是参数C，所以所有的样本均衡都是通过参数C来调整的。

SVC的参数：class_weight

可输入字典或者"balanced"，可不填，默认None 对SVC，将类i的参数C设置为class_weight [i] * C。如果没有给出具体的class_weight，则所有类都被假设为占有相同的权重1，模型会根据数据原本的状况去训练。如果希望改善样本不均衡状况，请输入形如{"标签的值1": 权重1, "标签的值2": 权重2}的字典，则参数C将会自动被设为：

标签的值1的C：权重1 * C，标签的值2的C：权重2*C

或者，可以使用"balanced"模式，这个模式使用y的值自动调整与输入数据中的类频率成反比的权重为n_samples/(n_classes * np.bincount(y))

SVC的接口fit的参数：sample_weight

数组，结构为(n_samples,)，必须对应输入fit中的特征矩阵的每个样本

每个样本在fit时的权重，让权重 * 每个样本对应的C值来迫使分类器强调设定的权重更大的样本。通常，较大的权重加在少数类的样本上，以迫使模型向着少数类的方向建模

通常来说，这两个参数我们只选取一个来设置。如果我们同时设置了两个参数，则C会同时受到两个参数的影响，即 class_weight中设定的权重 * sample_weight中设定的权重 * C。

我们接下来就来看看如何使用这个参数。

首先，我们来自建一组样本不平衡的数据集。我们在这组数据集上建两个SVC模型，一个设置有class_weight参数，一个不设置class_weight参数。我们对两个模型分别进行评估并画出他们的决策边界，以此来观察class_weight带来的效果。

1. 导入需要的库和模块

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.datasets import make_blobs
```

2. 创建样本不均衡的数据集

```
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)

#看看数据集长什么样
plt.scatter(X[:, 0], X[:, 1], c=y, cmap="rainbow", s=10)
#其中红色点是少数类, 紫色点是多数类
```

3. 在数据集上分别建模

```
#不设定class_weight
clf = svm.SVC(kernel='linear', C=1.0)
clf.fit(X, y)

#设定class_weight
wclf = svm.SVC(kernel='linear', class_weight={1: 10})
wclf.fit(X, y)

#给两个模型分别打分看看, 这个分数是accuracy准确度
clf.score(X, y)

wclf.score(X, y)
```

4. 绘制两个模型下数据的决策边界

还记得决策边界如何绘制的么？我们利用Contour函数来帮助我们。Contour是专门用来绘制等高线的函数。等高线，本质上是在二维图像上表现三维图像的一种形式，其中两维X和Y是两条坐标轴上的取值，而Z表示高度。

Contour就是将由X和Y构成平面上的所有点中，高度一致的点连接成线段的函数，在同一条等高线上的点一定具有相同的Z值。回忆一下，我们的决策边界是 $w \cdot x + b = 0$ ，并在决策边界的两边找出两个超平面，使得超平面到决策边界的相对距离为1。那其实，我们只需要在我们的样本构成的平面上，把所有到决策边界的距离为0的点相连，就是我们的决策边界。而到决策边界的距离可以使用我们说明过的接口decision_function来调用。

```
#首先要有数据分布
plt.figure(figsize=(6, 5))
plt.scatter(X[:, 0], X[:, 1], c=y, cmap="rainbow", s=10)
ax = plt.gca() #获取当前的子图, 如果不存在, 则创建新的子图

#绘制决策边界的第一步: 要有网格
```

```

xlim = ax.get_xlim()
ylim = ax.get_ylim()

xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T

#第二步：找出我们的样本点到决策边界的距离
z_clf = clf.decision_function(xy).reshape(xx.shape)
a = ax.contour(XX, YY, z_clf, colors='black', levels=[0], alpha=0.5, linestyles=['-'])

z_wclf = wclf.decision_function(xy).reshape(xx.shape)
b = ax.contour(XX, YY, z_wclf, colors='red', levels=[0], alpha=0.5, linestyles=['-'])

#第三步：画图例
plt.legend([a.collections[0], b.collections[0]], ["non weighted", "weighted"],
           loc="upper right")
plt.show()

```

图例这一步是怎么做到的？

```

a.collections #调用这个等高线对象中画的所有线，返回一个惰性对象

#用[*]把它打开试试看
[*a.collections] #返回了一个linecollection对象，其实是我们等高线里所有的线的列表

#现在我们只有一条线，所以我们可以使用索引0来锁定这个对象
a.collections[0]

#plt.legend([对象列表],[图例列表],loc)
#只要对象列表和图例列表相对应，就可以显示出图例

```

从图像上可以看出，灰色是我们做样本平衡之前的决策边界。灰色线上方的点被分为一类，下方的点被分为另一类。可以看到，大约有一半少数类（红色）被分错，多数类（紫色点）几乎都被分类正确了。红色是我们做样本平衡之后的决策边界，同样是红色线上方一类，红色线下方一类。可以看到，做了样本平衡后，少数类几乎全部都被分类正确了，但是多数类有许多被分错了。我们来看看两种情况下模型的准确率如何表现：

```

#给两个模型分别打分看看，这个分数是accuracy准确度
#做样本均衡之后，我们的准确率下降了，没有样本均衡的准确率更高
clf.score(X,y)

wclf.score(X,y)

```

可以看出，从准确率的角度来看，不做样本平衡的时候准确率反而更高，做了样本平衡准确率反而变低了，**这是因为做了样本平衡后，为了要更有效地捕捉出少数类，模型误伤了许多多数类样本，而多数类被分错的样本数量 > 少数类被分类正确的样本数量，使得模型整体的精确性下降**。现在，如果我们的目的是模型整体的准确率，那我们就要拒绝样本平衡，使用class_weight被设置之前的模型。

然而在现实中，我们往往都在追求捕捉少数类，因为在很多情况下，将少数类判断错的代价是巨大的。比如我们之前提到的，判断潜在犯罪者和普通人的例子，如果我们没有能够识别出潜在犯罪者，那么这些人就可能去危害社会，造成恶劣影响，但如果我们把普通人错认为是潜在犯罪者，我们也许只是需要增加一些监控和人为甄别的成本。所以对我们来说，我们宁愿把普通人判错，也不想放过任何一个潜在犯罪者。我们希望不惜一切代价来捕获少数类，或者希望捕捉出尽量多的少数类，那我们就必须使用class_weight设置后的模型。

2 SVC的模型评估指标

从上一节的例子中可以看出，如果我们的目标是希望尽量捕获少数类，那准确率这个模型评估逐渐失效，所以我们需要新的模型评估指标来帮助我们。如果简单来看，其实我们只需要查看模型在少数类上的准确率就好了，只要能够将少数类尽量捕捉出来，就能够达到我们的目的。

但此时，新问题又出现了，我们对多数类判断错误后，会需要人工甄别或者更多的业务上的措施来一一排除我们判断错误的多数类，这种行为往往伴随着很高的成本。比如银行在判断“一个申请信用卡的客户是否会出现违约行为”的时候，如果一个客户被判断为“会违约”，这个客户的信用卡申请就会被驳回，如果为了捕捉出“会违约”的人，大量地将“不会违约”的客户判断为“会违约”的客户，就会有许多无辜的客户的申请被驳回。信用卡对银行来说意味着利息收入，而拒绝了许多本来不会违约的客户，对银行来说就是巨大的损失。同理，大众在召回不符合欧盟标准的汽车时，如果为了找到所有不符合标准的汽车，而将一堆本来符合标准了的汽车召回，这个成本是不可估量的。

也就是说，单纯地追求捕捉出少数类，就会成本太高，而不顾及少数类，又会无法达成模型的效果。所以在现实中，我们往往在寻找**捕获少数类的能力**和**将多数类判错后需要付出的成本**的平衡。如果一个模型在能够尽量捕获少数类的情况下，还能够尽量对多数类判断正确，则这个模型就非常优秀了。为了评估这样的能力，我们将引入新的模型评估指标：混淆矩阵和ROC曲线来帮助我们。

2.1 混淆矩阵 (Confusion Matrix)

混淆矩阵是二分类问题的多维衡量指标体系，在样本不平衡时极其有用。在混淆矩阵中，我们将少数类认为是正例，多数类认为是负例。在决策树，随机森林这些普通的分类算法里，即是说少数类是1，多数类是0。在SVM里，就是说少数类是1，多数类是-1。普通的混淆矩阵，一般使用{0,1}来表示。混淆矩阵如其名，十分容易让人混淆，在许多教材中，混淆矩阵中各种各样的名称和定义让大家难以理解难以记忆。我为大家找出了一种简化的方式来显示标准二分类的混淆矩阵，如图所示：

		预测值		
		1	0	
真实值	1	11	10	真实为1: 11 + 10
	0	01	00	真实为0: 01 + 00
		判断为1: 11 + 01	判断为0: 10 + 00	全部样本之和: 11 + 10 + 01 + 00

混淆矩阵中，永远是真实值在前，预测值在后。其实可以很容易看出，11和00的对角线就是全部预测正确的，01和10的对角线就是全部预测错误的。基于混淆矩阵，我们有六个不同的模型评估指标，这些评估指标的范围都在[0,1]之间，所有以11和00为分子的指标都是越接近1越好，所以以01和10为分子的指标都是越接近0越好。对于所有的指标，我们用橙色表示分母，用绿色表示分子，则我们有：

2.1.1 模型整体效果：准确率

		预测值		准确率 Accuracy
		1	0	
真实值	1	11	10	
	0	01	00	
		11 + 01	10 + 00	11 + 10 + 01 + 00

$$Accuracy = \frac{11 + 00}{11 + 10 + 01 + 00}$$

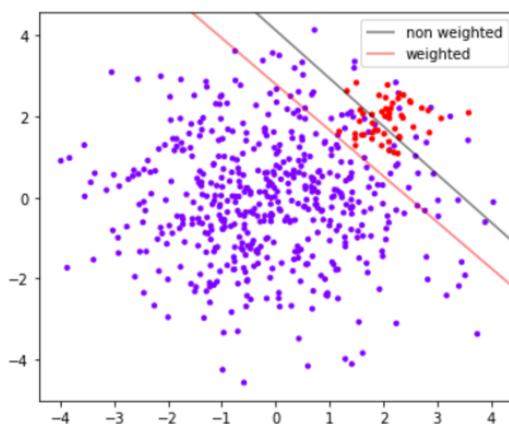
准确率Accuracy就是所有预测正确的所有样本除以总样本，通常来说越接近1越好。

2.1.2 捕捉少数类的艺术：精确度，召回率和F1 score

		预测值		精确度 Precision
		1	0	
真实值	1	11	10	
	0	01	00	
		11 + 01	10 + 00	11 + 10 + 01 + 00

$$Precision = \frac{11}{11 + 01}$$

精确度Precision，又叫查准率，表示所有被我们预测为是少数类的样本中，真正的少数类所占的比例。在支持向量机中，精确度可以被形象地表示为决策边界上方的所有点中，红色点所占的比例。精确度越高，代表我们捕捉正确的红色点越多，对少数类的预测越精确。精确度越低，则代表我们误伤了过多的多数类。**精确度是“将多数类判错后所需付出成本”的衡量。**



```
#所有判断正确并确实为1的样本 / 所有被判断为1的样本
#对于没有class_weight，没有做样本平衡的灰色决策边界来说：
(y[y == clf.predict(X)] == 1).sum() / (clf.predict(X) == 1).sum()

#对于有class_weight，做了样本平衡的红色决策边界来说：
(y[y == wclf.predict(X)] == 1).sum() / (wclf.predict(X) == 1).sum()
```

可以看出，做了样本平衡之后，精确度是下降的。因为很明显，样本平衡之后，有更多的多数类紫色点被我们误伤了。精确度可以帮助我们判断，是否每一次对少数类的预测都精确，所以又被称为“查准率”。在现实的样本不平衡例子中，**当每一次将多数类判断错误的成本非常高昂的时候（比如大众召回车辆的例子），我们会追求高精确度。**精确度越低，我们对多数类的判断就会越错误。当然了，如果我们的目标是不计一切代价捕获少数类，那我们并不在意精确度。

		预测值		召回率 (敏感度, 真正率) Recall (sensitivity, true positive rate)
		1	0	
真实值	1	11	10	11 + 10
	0	01	00	01 + 00
		11 + 01	10 + 00	11 + 10 + 01 + 00

$$Recall = \frac{11}{11 + 10}$$

召回率Recall，又被称为敏感度(sensitivity)，真正率，查全率，表示所有真实为1的样本中，被我们预测正确的样本所占的比例。在支持向量机中，召回率可以被表示为，决策边界上方的所有红色点占全部样本中的红色点的比例。召回率越高，代表我们尽量捕捉出了越多的少数类，召回率越低，代表我们没有捕捉出足够的少数类。

```
#所有predict为1的点 / 全部为1的点的比例
#对于没有class_weight，没有做样本平衡的灰色决策边界来说：
(y[y == clf.predict(x)] == 1).sum() / (y == 1).sum()

#对于有class_weight，做了样本平衡的红色决策边界来说：
(y[y == wclf.predict(x)] == 1).sum() / (y == 1).sum()
```

可以看出，做样本平衡之前，我们只成功捕获了60%左右的少数类点，而做了样本平衡之后的模型，捕捉到了100%的少数类点，从图像上来看，我们的红色决策边界的确捕捉到了全部的少数类，而灰色决策边界只捕捉到了一半左右。召回率可以帮助我们判断，我们是否捕捉除了全部的少数类，所以又叫做查全率。

如果我们希望不计一切代价，找出少数类（比如找出潜在犯罪者的例子），那我们就会追求高召回率，相反如果我们的目标不是尽量捕获少数类，那我们就不需要在意召回率。

注意召回率和精确度的分子是相同的（都是11），只是分母不同。而**召回率和精确度是此消彼长的，两者之间的平衡代表了捕捉少数类的需求和尽量不要误伤多数类的需求的平衡**。究竟要偏向于哪一方，取决于我们的业务需求：究竟是误伤多数类的成本更高，还是无法捕捉少数类的代价更高。

为了同时兼顾精确度和召回率，我们创造了两者的调和平均数作为考量两者平衡的综合性指标，称之为**F1 measure**。两个数之间的调和平均倾向于靠近两个数中比较小的那个数，因此我们追求尽量高的F1 measure，能够保证我们的精确度和召回率都比较高。F1 measure在[0,1]之间分布，越接近1越好。

$$F - measure = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}} = \frac{2 * Precision * Recall}{Precision + Recall}$$

从Recall延伸出来的另一个评估指标叫做**假负率 (False Negative Rate)**，它等于 $1 - Recall$ ，用于衡量所有真实为1的样本中，被我们错误判断为0的，通常用得不多。

$$FNR = \frac{10}{11 + 10}$$

2.1.3 判错多数类的考量：特异度与假正率

		预测值		
		1	0	
真实值	1	11	10	$11 + 10$
	0	01	00	$01 + 00$
		$11 + 01$	$10 + 00$	$11 + 10 + 01 + 00$

特异度 (真负率)
Specificity, true negative rate

$$Specificity = \frac{00}{01 + 00}$$

特异度(Specificity)表示所有真实为0的样本中，被正确预测为0的样本所占的比例。在支持向量机中，可以形象地表示为，决策边界下方的点占所有紫色点的比例。

```
#所有被正确预测为0的样本 / 所有的0样本
#对于没有class_weight, 没有做样本平衡的灰色决策边界来说:
(y[y == clf.predict(x)] == 0).sum() / (y == 0).sum()

#对于有class_weight, 做了样本平衡的红色决策边界来说:
(y[y == wclf.predict(x)] == 0).sum() / (y == 0).sum()
```

特异度衡量了一个模型将多数类判断正确的能力，而 $1 - specificity$ 就是一个模型将多数类判断错误的能力，这种能力被计算如下，并叫做**假正率 (False Positive Rate)**：

		预测值		
		1	0	
真实值	1	11	10	$11 + 10$
	0	01	00	$01 + 00$
		$11 + 01$	$10 + 00$	$11 + 10 + 01 + 00$

假正率
False positive rate

$$FPR = \frac{01}{01 + 00}$$

在支持向量机中，假正率就是决策边界上方的紫色点（所有被判断错误的多数类）占所有紫色点的比例。根据我们之前在precision处的分析，其实可以看得出来，当样本均衡过后，假正率会更高，因为有更多紫色点被判断错误，而样本均衡之前，假正率比较低，被判错的紫色点比较少。所以假正率其实类似于Precision的反向指标，Precision衡量有多少少数点被判断正确，而假正率FPR衡量有多少多数点被判断错误，性质是十分类似的。

2.1.4 sklearn中的混淆矩阵

sklearn当中提供了大量的类来帮助我们了解和使用混淆矩阵。

类	含义
sklearn.metrics.confusion_matrix	混淆矩阵
sklearn.metrics.accuracy_score	准确率accuracy
sklearn.metrics.precision_score	精确度precision
sklearn.metrics.recall_score	召回率recall
sklearn.metrics.precision_recall_curve	精确度-召回率平衡曲线
sklearn.metrics.f1_score	F1 measure

2.2 ROC曲线以及其相关问题

基于混淆矩阵，我们学习了总共六个指标：准确率Accuracy，精确度Precision，召回率Recall，精确度和召回度的平衡指标F measure，特异度Specificity，以及假正率FPR。

其中，假正率有一个非常重要的应用：我们在追求较高的Recall的时候，Precision会下降，就是说随着更多的少数类被捕捉出来，会有更多的多数类被判断错误，但我们很好奇，随着Recall的逐渐增加，模型将多数类判断错误的能力如何变化呢？我们希望理解，我每判断正确一个少数类，就有多少个多数类会被判断错误。假正率正好可以帮助我们衡量这个能力的变化。相对的，Precision无法判断这些判断错误的多数类在全部多数类中究竟占多大的比例，所以无法在提升Recall的过程中也顾及到模型整体的Accuracy。因此，我们可以使用Recall和FPR之间的平衡，来替代Recall和Precision之间的平衡，让我们衡量模型在尽量捕捉少数类的时候，误伤多数类的情况如何变化，这就是我们的ROC曲线衡量的平衡。

ROC曲线，全称The Receiver Operating Characteristic Curve，译为受试者操作特性曲线。这是一条以不同阈值下的假正率FPR为横坐标，不同阈值下的召回率Recall为纵坐标的曲线。让我们先从概率和阈值开始讲起。

2.2.1 概率(probability)与阈值(threshold)

要理解概率与阈值，最容易的状况是来回忆一下我们用逻辑回归做分类的时候的状况。逻辑回归的predict_proba接口对每个样本生成每个标签类别下的似然（类概率）。对于这些似然，逻辑回归天然规定，当一个样本所对应的这个标签类别下的似然大于0.5的时候，这个样本就被分为这一类。比如说，一个样本在标签1下的似然是0.6，在标签0下的似然是0.4，则这个样本的标签自然就被分为1。逻辑回归的回归值本身，其实也就是标签1下的似然。在这个过程中，0.5就被称为阈值。来看看下面的例子：

1. 自建数据集

```
class_1_ = 7
class_2_ = 4
centers_ = [[0.0, 0.0], [1, 1]]
clusters_std = [0.5, 1]
X_, y_ = make_blobs(n_samples=[class_1_, class_2_],
                     centers=centers_,
                     cluster_std=clusters_std,
                     random_state=0, shuffle=False)
plt.scatter(X_[:, 0], X_[:, 1], c=y_, cmap="rainbow", s=30)
```

2. 建模，调用概率

```
from sklearn.linear_model import LogisticRegression as LogiR

clf_lo = LogiR().fit(x_,y_)

prob = clf_lo.predict_proba(x_)

#将样本和概率放到一个DataFrame中
import pandas as pd
prob = pd.DataFrame(prob)

prob.columns = ["0","1"]

prob
```

3. 使用阈值0.5，大于0.5的样本被预测为1，小于0.5的样本被预测为0

```
#手动调节阈值，来改变我们的模型效果
for i in range(prob.shape[0]):
    if prob.loc[i,"1"] > 0.5:
        prob.loc[i,"pred"] = 1
    else:
        prob.loc[i,"pred"] = 0

prob["y_true"] = y_

prob = prob.sort_values(by="1",ascending=False)

prob
```

4. 使用混淆矩阵查看结果

```
from sklearn.metrics import confusion_matrix as CM, precision_score as P, recall_score as R

CM(prob.loc[:, "y_true"],prob.loc[:, "pred"],labels=[1,0])

#试试看手动计算Precision和Recall?
P(prob.loc[:, "y_true"],prob.loc[:, "pred"],labels=[1,0])

R(prob.loc[:, "y_true"],prob.loc[:, "pred"],labels=[1,0])
```

5. 假如我们使用0.4作为阈值呢？

```
for i in range(prob.shape[0]):
    if prob.loc[i,"1"] > 0.4:
        prob.loc[i,"pred"] = 1
    else:
        prob.loc[i,"pred"] = 0
```

```
prob
```

```
CM(prob.loc[:, "y_true"], prob.loc[:, "pred"], labels=[1, 0])  
P(prob.loc[:, "y_true"], prob.loc[:, "pred"], labels=[1, 0])  
R(prob.loc[:, "y_true"], prob.loc[:, "pred"], labels=[1, 0])  
  
#注意，降低或者升高阈值并不一定能够让模型的效果变好，一切都基于我们要追求怎样的模型效果
```

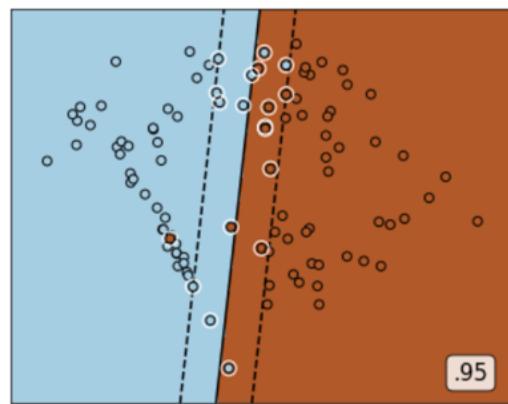
可见，在不同阈值下，我们的模型评估指标会发生变化，我们正利用这一点来观察Recall和FPR之间如何互相影响。**但是注意，并不是升高阈值，就一定能够增加或者减少Recall，一切要根据数据的实际分布来进行判断。**而要体现阈值的影响，首先必须的得到分类器在少数类下的预测概率。对于逻辑回归这样天生成似然的算法和朴素贝叶斯这样就是在计算概率的算法，自然非常容易得到概率，但对于一些其他的分类算法，比如决策树，比如SVM，他们的分类方式和概率并不相关。那在他们身上，我们就无法画ROC曲线了吗？并非如此。

决策树有叶子节点，一个叶子节点上可能包含着不同类的样本。假设一个样本被包含在叶子节点a中，节点a包含10个样本，其中6个为1，4个为0，则1这个正类在这个叶子节点中的出现概率就是60%，类别0在这个叶子节点中的出现概率就是40%。对于所有在这个叶子节点中的样本而言，节点上的1和0出现的概率，就是这个样本对应的取到1和0的概率，大家可以去自己验证一下。但是思考一个问题，由于决策树可以被画得很深，在足够深的情况下，决策树的每个叶子节点上可能都不包含多个类别的标签了，可能一片叶子中只有唯一的一个标签，即叶子节点的不纯度为0，此时此刻，对于每个样本而言，他们所对应的“概率”就是0或者1了。这个时候，我们就无法调节阈值来调节我们的Recall和FPR了。对于随机森林，也是如此。

所以，如果我们有概率需求，我们还是会优先追求逻辑回归或者朴素贝叶斯。不过其实，SVM也可以生成概率，我们一起来看看，它是怎么做的。

2.2.2 SVM实现概率预测：重要参数**probability**，接口**predict_proba**以及**decision_function**

我们在画等高线，也就是决策边界的时候曾经使用SVC的接口**decision_function**，它返回我们输入的特征矩阵中每个样本到划分数据集的超平面的距离。我们在SVM中利用超平面来判断我们的样本，本质上来说，当两个点的距离是相同的符号的时候，越远离超平面的样本点归属于某个标签类的概率就很大。比如说，一个距离超平面0.1的点，和一个距离超平面100的点，明显是距离为0.1的点更有可能是负类别的点混入了边界。同理，一个距离超平面距离为-0.1的点，和一个离超平面距离为-100的点，明显是-100的点的标签更有可能是负类。所以，到超平面的距离一定程度上反应了样本归属于某个标签类的可能性。**接口**decision_function**返回的值也因此被我们认为是SVM中的置信度（confidence）。**



不过，置信度始终不是概率，它没有边界，可以无限大，大部分时候也不是以百分比或者小数的形式呈现，而SVC的判断过程又不像决策树一样可以求解出一个比例。为了解决这个矛盾，SVC有重要参数probability。

| 参数 | 含义 | | ----- | | probability | 布尔值，可不填，默认 False

是否启用概率估计。进行必须在调用fit之前启用它，启用此功能会减慢SVM的运算速度。

设置为True则会启动，启用之后，SVC的接口predict_proba和predict_log_proba将生效。

在二分类情况下，SVC将使用Platt缩放来生成概率，即在decision_function生成的距离上进行Sigmoid压缩，并附加训练数据的交叉验证拟合，来生成类逻辑回归的SVM分数。

在多分类状况下，参考Wu et al. (2004)发表的文章来将二分类情况推广到多分类。 |

- Wu, Lin and Weng, "[Probability estimates for multi-class classification by pairwise coupling](#)", JMLR 5:975-1005, 2004.

来实现一下我们的概率预测吧：

```
#使用最初的x和y，样本不均衡的这个模型

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)

#看看数据集长什么样
plt.scatter(x[:, 0], x[:, 1], c=y, cmap="rainbow", s=10)
#其中红色点是少数类，紫色点是多数类

clf_proba = svm.SVC(kernel="linear", C=1.0, probability=True).fit(x,y)

clf_proba.predict_proba(x)

clf_proba.predict_proba(x).shape

clf_proba.decision_function(x)
```

```
c1f_proba.decision_function(x).shape
```

值得注意的是，在二分类过程中，`decision_function`只会生成一列距离，样本的类别由距离的符号来判断，但是`predict_proba`会生成两个类别分别对应的概率。SVM也可以生成概率，所以我们可以使用和逻辑回归同样的方式来在SVM上设定和调节我们的阈值。

毋庸置疑，Platt缩放中涉及的交叉验证对于大型数据集来说非常昂贵，计算会非常缓慢。另外，由于Platt缩放的理论原因，在二分类过程中，有可能出现`predict_proba`返回的概率小于0.5，但样本依旧被标记为正类的情况出现，毕竟支持向量机本身并不依赖于概率来完成自己的分类。如果我们的的确需要置信度分数，但不一定非要是概率形式的话，那建议可以将`probability`设置为`False`，使用`decision_function`这个接口而不是`predict_proba`。

2.2.3 绘制SVM的ROC曲线

现在，我们理解了什么是阈值（threshold），了解了不同阈值会让混淆矩阵产生变化，也了解了如何从我们的分类算法中获取概率。现在，我们就可以开始画我们的ROC曲线了。ROC是一条以不同阈值下的假正率FPR为横坐标，不同阈值下的召回率Recall为纵坐标的曲线。简单地来说，只要我们有数据和模型，我们就可以在python中绘制出我们的ROC曲线。思考一下，我们要绘制ROC曲线，就必须在我们的数据中去不断调节阈值，不断求解混淆矩阵，然后不断获得我们的横坐标和纵坐标，最后才能够将曲线绘制出来。接下来，我们就来执行这个过程：

```
#首先来看看如何从混淆矩阵中获取FPR和Recall
cm = CM(prob.loc[:, "y_true"], prob.loc[:, "pred"], labels=[1, 0])

cm

#FPR
cm[1, 0] / cm[1, :].sum()

#Recall
cm[0, 0] / cm[0, :].sum()

#开始绘图
recall = []
FPR = []

probrange = np.linspace(c1f_proba.predict_proba(x)
[:, 1].min(), c1f_proba.predict_proba(x)[:, 1].max(), num=50, endpoint=False)

from sklearn.metrics import confusion_matrix as CM, recall_score as R
import matplotlib.pyplot as plot

for i in probrange:
    y_predict = []
    for j in range(x.shape[0]):
        if c1f_proba.predict_proba(x)[j, 1] > i:
            y_predict.append(1)
        else:
            y_predict.append(0)
    cm = CM(y, y_predict, labels=[1, 0])
    recall.append(cm[0, 0] / cm[0, :].sum())
```

```
FPR.append(cm[1,0]/cm[1,:].sum())

recall.sort()
FPR.sort()

plt.plot(FPR, recall, c="red")
plt.plot(probrange+0.05, probrange+0.05, c="black", linestyle="--")
plt.show()
```

现在我们就画出了ROC曲线了，那我们如何理解这条曲线呢？先来回忆一下，我们建立ROC曲线的根本目的是找寻Recall和FPR之间的平衡，让我们能够衡量模型在尽量捕捉少数类的时候，误伤多数类的情况会如何变化。横坐标是FPR，代表着模型将多数类判断错误的能力，纵坐标Recall，代表着模型捕捉少数类的能力，所以ROC曲线代表着，随着Recall的不断增加，FPR如何增加。我们希望随着Recall的不断提升，FPR增加得越慢越好，这说明我们可以尽量高效地捕捉出少数类，而不会将很多地多数类判断错误。所以，我们希望看到的图像是，纵坐标急速上升，横坐标缓慢增长，也就是在整个图像左上方的一条弧线。这代表模型的效果很不错，拥有较好的捕获少数类的能力。

中间的虚线代表着，当recall增加1%，我们的FPR也增加1%，也就是说，我们每捕捉出一个少数类，就会有一个多数类被判错，这种情况下，模型的效果就不好，这种模型捕获少数类的结果，会让许多多数类被误伤，从而增加我们的成本。ROC曲线通常都是凸型的。对于一条凸型ROC曲线来说，曲线越靠近左上角越好，越往下越糟糕，曲线如果在虚线的下方，则证明模型完全无法使用。但是它也有可能是一条凹形的ROC曲线。对于一条凹型ROC曲线来说，应该越靠近右下角越好，凹形曲线代表模型的预测结果与真实情况完全相反，那也不算非常糟糕，只要我们手动将模型的结果逆转，就可以得到一条左上方的弧线了。最糟糕的就是，无论曲线是凹形还是凸型，曲线位于图像中间，和虚线非常靠近，那我们拿它无能为力。

好了，现在我们有了这条曲线，我们的确知道模型的效果还算是不错了。但依然非常摸棱两可，有没有具体的数字来帮助我们理解ROC曲线和模型的效果呢？的确存在，这个数字就叫做AUC面积，它代表了ROC曲线下方的面积，这个面积越大，代表ROC曲线越接近左上角，模型就越好。AUC面积的计算比较繁琐，因此，我们使用sklearn来帮助我们。接下来我们来看看，在sklearn当中，如何绘制我们的ROC曲线，找出我们的的AUC面积。

2.2.4 sklearn中的ROC曲线和AUC面积

在sklearn中，我们有帮助我们计算ROC曲线的横坐标假正率FPR，纵坐标Recall和对应的阈值的类sklearn.metrics.roc_curve。同时，我们还有帮助我们计算AUC面积的类sklearn.metrics.roc_auc_score。在一些比较老旧的sklearn版本中，我们使用sklearn.metrics.auc这个类来计算AUC面积，但这个类即将在0.22版本中被放弃，因此建议大家都使用roc_auc_score。来看看我们的这两个类：

```
sklearn.metrics.roc_curve(y_true, y_score, pos_label=None, sample_weight=None, drop_intermediate=True)
```

y_true : 数组，形状 = [n_samples]，真实标签

y_score : 数组，形状 = [n_samples]，置信度分数，可以是正类样本的概率值，或置信度分数，或者decision_function返回的距离

pos_label : 整数或者字符串，默认None，表示被认为是正类样本的类别

sample_weight : 形如 [n_samples]的类数组结构，可不填，表示样本的权重

drop_intermediate : 布尔值，默认True，如果设置为True，表示会舍弃一些ROC曲线上不显示的阈值点，这对于计算一个比较轻量的ROC曲线来说非常有用

这个类以此返回：FPR，Recall以及阈值。

来看看我们如何使用：

```
from sklearn.metrics import roc_curve

FPR, recall, thresholds = roc_curve(y, clf_proba.decision_function(X), pos_label=1)

FPR
recall
thresholds #此时的threshold就不是一个概率值，而是距离值中的阈值了，所以它可以大于1，也可以为负
```

```
sklearn.metrics.roc_auc_score(y_true, y_score, average='macro', sample_weight=None, max_fpr=None)
```

AUC面积的分数使用以上类来进行计算，输入的参数也比较简单，就是真实标签，和与roc_curve中一致的置信度分数或者概率值。

```
from sklearn.metrics import roc_auc_score as AUC

area = AUC(y, clf_proba.decision_function(X))
```

接下来就可以开始画图了：

```
plt.figure()
plt.plot(FPR, recall, color='red',
          label='ROC curve (area = %0.2f)' % area)
plt.plot([0, 1], [0, 1], color='black', linestyle='--')
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('Recall')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()
```

如此就得到了我们的ROC曲线和AUC面积，可以看到，SVM在这个简单数据集上的效果还是非常好的。并且大家可以通过观察我们使用decision_function画出的ROC曲线，对比一下我们之前强行使用概率画出来的曲线，两者非常相似，所以在无法获取模型概率的情况下，其实不必强行使用概率，如果有置信度，那也使可以完成我们的ROC曲线的。感兴趣的小伙伴可以画一下如果带上class_weight这个参数，模型的效果会变得如何。

2.2.5 利用ROC曲线找出最佳阈值

现在，有了ROC曲线，了解了模型的分类效力，以及面对样本不均衡问题时的效力，那我们如何求解我们最佳的阈值呢？我们想要了解，什么样的状况下我们的模型的效果才是最好的。回到我们对ROC曲线的理解来：ROC曲线反应的是recall增加的时候FPR如何变化，也就是当模型捕获少数类的能力变强的时候，会误伤多数类的情况是否严重。我们的希望是，模型在捕获少数类的能力变强的时候，尽量不误伤多数类，也就是说，随着recall的变大，FPR的大小越小越好。所以我们希望找到的最有意义，其实是Recall和FPR差距最大的点。这个点，又叫做**约登指数**。

```
maxindex = (recall - FPR).tolist().index(max(recall - FPR))
```

```

thresholds[maxindex]

#我们在图像上来看看这个点在哪里
plt.scatter(FPR[maxindex], recall[maxindex], c="black", s=30)

#把上述代码放入这段代码中:
plt.figure()
plt.plot(FPR, recall, color='red',
          label='ROC curve (area = %0.2f)' % area)
plt.plot([0, 1], [0, 1], color='black', linestyle='--')
plt.scatter(FPR[maxindex], recall[maxindex], c="black", s=30)
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('Recall')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()

```

最佳阈值就这样选取出来了，由于现在我们是使用decision_function来画ROC曲线，所以我们选择出来的最佳阈值其实是最佳距离。如果我们使用的是概率，我们选取的最佳阈值就会使一个概率值了。只要我们让这个距离/概率以上的点，都为正类，让这个距离/概率以下的点都为负类，模型就是最好的：即能够捕捉出少数类，又能够尽量不误伤多数类，整体的精确性和对少数类的捕捉都得到了保证。

而从找出的最优阈值点来看，这个点，其实是图像上离左上角最近的点，离中间的虚线最远的点，也是ROC曲线的转折点。如果没有时间进行计算，或者横坐标比较清晰的时候，我们就可以观察转折点来找到我们的最佳阈值。

到这里为止，SVC的模型评估指标就介绍完毕了。但是，SVC的样本不均衡问题还可以有很多的探索。另外，我们还可以使用KS曲线，或者收益曲线(profit chart)来选择我们的阈值，都是和ROC曲线类似的用法。大家若有余力，可以自己深入研究一下。模型评估指标，还有很多深奥的地方。

3 使用SVC时的其他考虑

3.1 SVC处理多分类问题：重要参数decision_function_shape

之前所有的SVM内容，全部是基于二分类的情况来说明的，因为**支持向量机是天生二分类的模型**。不过，它也可以做多分类，但是SVC在多分类情况上的推广，属于恶魔级别的难度，要从数学角度去理解几乎是不可能的，因为要研究透彻多分类状况下的SVC，就必须研究透彻多分类时所需要的决策边界个数，每个决策边界所需要的支持向量的个数，以及这些支持向量如何组合起来计算我们的拉格朗日乘数，要求我们必须对SMO或者梯度下降求解SVC的拉格朗日乘数的过程十分熟悉。这些内容推广到多分类之后，即便在线性可分的二维数据上都已经复杂，要再推广到非线性可分的高维情况，就远远超出了我们这个课程的要求。

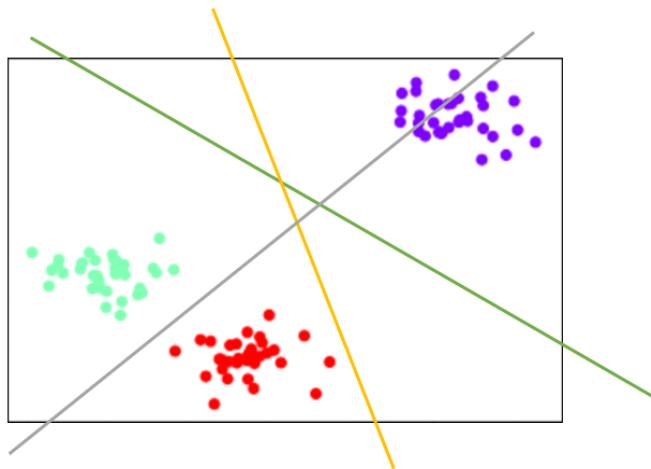
sklearn中用了许多巧妙的方法来为我们呈现结果，在这里，这一小节会为大家简单介绍sklearn当中是如何SVC的多分类问题的，但需要注意，这一节的内容只是一个简介，并不能带大家深入理解多分类中各种深奥的情况。大家可根据自己的需求酌情选读。

支持向量机是天生二分类的模型，所以支持向量机在处理多分类问题的时候，是把多分类问题转换成了二分类问题来解决。这种转换有两种模式，一种叫做“一对一”模式 (one vs one)，一种叫做“一对多”模式(one vs rest)。

在ovo模式下，标签中的所有类别会被两两组合，每两个类别之间建一个SVC模型，每个模型生成一个决策边界，分别进行二分类。这种模式下，对于含有 n_{class} 个标签类别的数据来说，SVC会生成总共 $C_{n_{class}}^2$ 个模型，即会生成总共 $C_{n_{class}}^2$ 个超平面，其中：

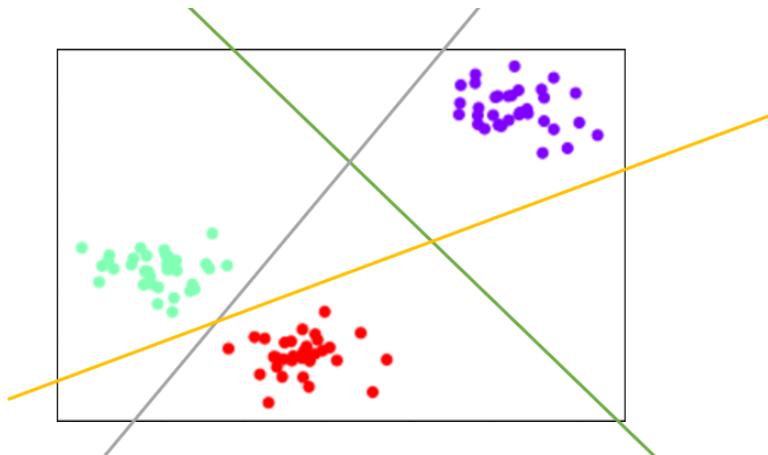
$$C_{n_{class}}^2 = \frac{n_{class} * (n_{class} - 1)}{2}$$

比如说，来看ovo模式下，二维空间中的三分类情况。



首先让提出紫色点和红色点作为一组，然后求解出两个类之间的SVC和绿色决策边界。然后让绿色点和红色点作为一组，求解出两个类之间的SVC和灰色边界。最后让绿色和紫色组成一组，组成两个类之间的SVC和黄色边界。然后基于三个边界，分别对三个类别进行分类。

在ovr模式下，标签中所有的类别会分别与其他类别进行组合，建立 n_{class} 个模型，每个模型生成一个决策边界，分别进行二分类。同样的数据集，如果是ovr模式，则会生成如下的决策边界：



紫色类 vs 剩下的类，生成绿色的决策边界。红色类 vs 剩下的类，生成黄色的决策边界。绿色类 vs 剩下的类，生成灰色的决策边界，当类别更多的时候，如此类推下去，我们永远需要 n_{class} 个模型。

当类别更多的时候，无论是ovr还是ovo模式需要的决策边界都会越来越多，模型也会越来越复杂，不过ovo模式下的模型计算会更加复杂，因为ovo模式中的决策边界数量增加更快，但相对的，ovo模型也会更加精确。ovr模型计算更快，但是效果往往不是很好。在硬件可以支持的情况下，还是建议选择ovo模式。

一旦模型和超平面的数量变化了，SVC的很多计算过程，还有接口和属性都会发生变化：

1. 在二分类中，所有的支持向量都服务于唯一的超平面，在多分类问题中，每个支持向量都会被用来服务于多个超平面
2. 在生成一个超平面的二分类过程中，我们计算一个超平面上的支持向量对应的拉格朗日乘数 α ，现在，由于有多个超平面，所以需要的支持向量的个数增长了，因而求解拉格朗日乘数的需求也变得更多。在二分类问题中，每一个支持向量求解出一个拉格朗日乘数，因此拉格朗日乘数的数目和支持向量的数一致。但在多分类问题中，两个不同超平面的支持向量被用来决定一个拉格朗日乘数的取值，并且规定一个支持向量至少要被两个超平面使用。假设一个多分类问题中分别有三个超平面，超平面A上有3个支持向量，超平面B和C上分别有2个支持向量，则总共7个支持向量就要求解14个对应的拉格朗日乘数。以这样的考虑来看，拉格朗日乘数的计算也会变得异常复杂。
3. 在简单二分类中，`decision_function`只返回每个样本点到唯一的超平面的距离，而在多分类问题中这个接口将根据选择的多分类模式不同而返回不同的结构。
4. 同理，在二分类中只生成一条直线，所以属性`coef_`和`intercept_`返回的结构都很单纯，但在多分类问题，尤其是ovo类型下，两个属性都受到不同程度的影响。

参数`decision_function_shape`决定我们究竟使用哪一种分类模式。

`decision_function_shape`

可输入“ovo”，“ovr”，默认“ovr”，对所有分类器，选择使用ovo或者ovr模式。

选择ovr模式，则返回的`decision_function`结构为 $(n_{\text{samples}}, n_{\text{class}})$ 。二分类时，尽管选用ovr模式，却会返回 $(n_{\text{samples}},)$ 的结构。

选择ovo模式，则使用libsvm中原始的，结构为 $(n_{\text{samples}}, \frac{n_{\text{class}}(n_{\text{class}}-1)}{2})$ 的`decision_function`接口。在ovo模式并且核函数为线性核的情况下，属性`coef_`和`intercept_`会分别返回 $(\frac{n_{\text{class}}(n_{\text{class}}-1)}{2}, n_{\text{features}})$ 和 $(\frac{n_{\text{class}}(n_{\text{class}}-1)}{2},)$ 的结构，每行对应一个生成的二元分类器。

ovo模式只在多分类的状况下使用。

3.2 SVM的模型复杂度

支持向量机是强大的工具，但随着训练向量的数量的增大，它对的计算和存储的需求迅速增加。SVM的核心是二次规划问题 (QP)，将支持向量与其余训练数据分开。这个基于libsvm的实现使用的QP解算器可以在实践中实现 $O(n_{features} * n_{samples}^2)$ 到 $O(n_{features} * n_{samples}^3)$ 之间的复杂度，一切基于libsvm的缓存在实践中的效率有多高。这个效率由数据集确定。如果数据集非常稀疏，在应该将时间复杂度中的 $n_{features}$ 替换为样本向量中非零特征的平均个数。注意，如果数据是线性的，使用类LinearSVC比使用SVC中的核函数"linear"要更有效，而且LinearSVC可以几乎线性地拓展到数百万个样本或特征的数据集上。

3.3 SVM中的随机性：参数random_state

虽然不常用，但是SVC中包含参数random_state，这个参数受到probability参数的影响，仅在生成高概率估计的时候才会生效。在概率估计中，SVC使用随机数生成器来混合数据。如果概率设置为False，则random_state对结果没有影响。如果不实现概率估计，SVM中不存在有随机性的过程。

3.4 SVC的重要属性补充

到目前为止，SVC的几乎所有重要参数，属性和接口我们都已经介绍完毕了。在这里，给大家做一个查缺补漏：

```
#属性n_support_: 调用每个类别下的支持向量的数目
clf_proba.n_support_

#属性coef_: 每个特征的重要性，这个系数仅仅适合于线性核
clf_proba.coef_

#属性intercept_: 查看生成的决策边界的截距
clf_proba.intercept_

#属性dual_coef_: 查看生成的拉格朗日乘数
clf_proba.dual_coef_

clf_proba.dual_coef_.shape

#注意到这个属性的结构了吗？来看看查看支持向量的属性
clf_proba.support_vectors_

clf_proba.support_vectors_.shape

#注意到dual_coef_中生成的拉格朗日乘数的数目和我们的支持向量的数目一致
#注意到KKT条件的条件中的第五条，所有非支持向量会让拉格朗日乘数为0
#所以拉格朗日乘数的数目和支持向量的数目是一致的
#注意，此情况仅仅在二分类中适用！
```

3.5 一窥线性支持向量机类LinearSVC

到这里，我们基本上已经了解完毕了SVC在sklearn中的使用状况。当然，还有很多可以深入的东西，大家如果感兴趣可以自己深入研究。除了最常见的SVC类之外，还有一个重要的类可以使用：线性支持向量机linearSVC。

```
class sklearn.svm.LinearSVC(penalty='l2', loss='squared_hinge', dual=True, tol=0.0001, C=1.0, multi_class='ovr',
fit_intercept=True, intercept_scaling=1, class_weight=None, verbose=0, random_state=None, max_iter=1000)
```

线性支持向量机其实与SVC类中选择"linear"作为核函数的功能类似，但是其背后的实现库是liblinear而不是libsvm，这使得在线性数据上，linearSVC的运行速度比SVC中的"linear"核函数要快，不过两者的运行结果相似。在现实中，许多数据都是线性的，因此我们可以依赖计算得更快得LinearSVC类。除此之外，线性支持向量可以很容易地推广到大样本上，还可以支持稀疏矩阵，多分类中也支持ovr方案。

线性支持向量机的许多参数看起来和逻辑回归非常类似，比如可以选择惩罚项，可以选择损失函数等等，这让它在线性数据上表现更加灵活。

参数	含义
penalty	在求解决策边界过程中使用的正则惩罚项，可以输入"l1"或者"l2"，默认"l2"和逻辑回归中地正则惩罚项非常类似，"l1"会让决策边界中部分特征的系数w被压缩到0，而"l2"会让每个特征都被分配到一个不为0的系数
loss	在求解决策边界过程中使用的损失函数，可以输入"hinge"或者"squared_hinge"，默认为"square_hinge"当输入"hinge"，表示默认使用和类SVC中一致的损失函数，使用"squared_hinge"表示使用SVC中损失函数的平方作为损失函数。
dual	布尔值，默认为True 选择让算法直接求解原始的拉格朗日函数，或者求解对偶函数。当选择为True的时候，表示求解对偶函数，如果样本量大于特征数目，建议求解原始拉格朗日函数，设定dual = False。

和SVC一样，LinearSVC也有C这个惩罚参数，但LinearSVC在C变大时对C不太敏感，并且在某个阈值之后就不能再改善结果了。同时，较大的C值将需要更多的时间进行训练，2008年时有人做过实验，LinearSVC在C很大的时候训练时间可以比原来长10倍。

4 SVC真实数据案例：预测明天是否会下雨

SVC在现实中的应用十分广泛，尤其在图像和文字识别方面。然而，这些数据不仅非常难以获取，还难以在课程中完整呈现出来，但SVC真实应用的代码其实就在sklearn中的三行，真正能够展现出SVM强大之处的，反而很少是案例本身，而是我们之前所作的各种探索。

我们在学习算法的时候，会使用各种各样的数据集来进行演示，但这些数据往往非常干净并且规整，不需要做太多的数据预处理。在我们讲解第三章：数据预处理与特征工程时，用了自制的文字数据和kaggle上的高维数据来为大家讲解，然而这些数据依然不能够和现实中采集到的数据的复杂程度相比。因此大家学习了这门课程，却依然会对究竟怎样做预处理感到困惑。

在实际工作中，数据预处理往往比建模难得多，耗时多得多，因此合理的数据预处理是非常必要的。考虑到大家渴望学习真实数据上的预处理的需求，以及SVM需要在比较规则的数据集上来表现的特性，我为大家准备了这个Kaggle上下载的，未经过预处理的澳大利亚天气数据集。我们的目标是在这个数据集上来预测明天是否会下雨。

这个案例的核心目的，是通过巧妙的预处理和特征工程来向大家展示，在现实数据集上我们往往如何做数据预处理，或者我们都有哪些预处理的方式和思路。预测天气是一个非常非常困难的主题，因为影响天气的因素太多，而Kaggle的这份数据也丝毫不让我们失望，是一份非常难的数据集，难到我们目前学过的所有算法在这个数据集上都不会有太好的结果，尤其是召回率recall，异常地低。在这里，我为大家抛砖引玉，在这个15W行数据的数据集上，随机抽样5000个样本来为大家演示我的数据预处理和特征工程的过程，为大家提供一些数据预处理和特征工程的思路。不过，特征工程没有标准答案，因此大家应当多尝试，希望使用原数据集的小伙伴们可以到Kaggle下载最原始版本，或者直接从我们的课件打包下载的数据中获取：

Kaggle下载链接走这里：<https://www.kaggle.com/jsphyg/weather-dataset-rattle-package>

对于使用Kaggle原数据集的小伙伴的温馨提示：

记得好好阅读Kaggle上的各种数据集说明哦~！有一些特征是不能够使用的！

那就让我们开始我们的案例吧。

4.1 导库导数据，探索特征

导入需要的库

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
```

导入数据，探索数据

```
weather = pd.read_csv(r"C:\work\learnbetter\micro-class\week 8 SVM
(2)\data\weatherAUS5000.csv", index_col=0)

weather.head()
```

来查看一下各个特征都代表了什么：

特征/标签	含义
Date	观察日期
Location	获取该信息的气象站的名称
MinTemp	以摄氏度为单位的最低温度
MaxTemp	以摄氏度为单位的最高温度
Rainfall	当天记录的降雨量, 单位为mm
Evaporation	到早上9点之前的24小时的A级蒸发量 (mm)
Sunshine	白日受到日照的完整小时
WindGustDir	在到午夜12点前的24小时中的最强风的风向
WindGustSpeed	在到午夜12点前的24小时中的最强风速 (km / h)
WindDir9am	上午9点时的风向
WindDir3pm	下午3点时的风向
WindSpeed9am	上午9点之前每个十分钟的风速的平均值 (km / h)
WindSpeed3pm	下午3点之前每个十分钟的风速的平均值 (km / h)
Humidity9am	上午9点的湿度 (百分比)
Humidity3am	下午3点的湿度 (百分比)
Pressure9am	上午9点平均海平面上的大气压 (hpa)
Pressure3pm	下午3点平均海平面上的大气压 (hpa)
Cloud9am	上午9点的天空被云层遮蔽的程度, 这是以“oktas”来衡量的, 这个单位记录了云层遮挡天空的程度。0表示完全晴朗的天空, 而8表示它完全是阴天。
Cloud3pm	下午3点的天空被云层遮蔽的程度
Temp9am	上午9点的摄氏度温度
Temp3pm	下午3点的摄氏度温度
RainTomorrow	目标变量, 我们的标签: 明天下雨了吗?

```
#将特征矩阵和标签Y分开
X = weather.iloc[:, :-1]
Y = weather.iloc[:, -1]
```

```
X.shape
```

```
#探索数据类型
X.info()
```

```
#探索缺失值
x.isnull().mean()

#探索标签的分类
np.unique(Y)
```

粗略观察可以发现，这个特征矩阵由一部分分类变量和一部分连续变量组成，其中云层遮蔽程度虽然是以数字表示，但是本质却是分类变量。大多数特征都是采集的自然数据，比如蒸发量，日照时间，湿度等等，而少部分特征是人为构成的。还有一些是单纯表示样本信息的变量，比如采集信息的地点，以及采集的时间。

4.2 分集，优先探索标签

分训练集和测试集，并做描述性统计

```
#分训练集和测试集
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, test_size=0.3, random_state=420)

#恢复索引
for i in [Xtrain, Xtest, Ytrain, Ytest]:
    i.index = range(i.shape[0])
```

在现实中，我们会先分训练集和测试集，再开始进行数据预处理。这是由于，测试集在现实中往往是不可获得的，或者被假设为是不可获得的，我们不希望我们建模的任何过程受到测试集数据的影响，否则的话，就相当于提前告诉了模型一部分预测的答案。在之前的课中，为了简便操作，都给大家忽略了这个过程，一律先进行预处理，再分训练集和测试集，这是一种不规范的做法。在这里，为了让案例尽量接近真实的样貌，所以采取了现实中所使用的这种方式：先分训练集和测试集，再一步步进行预处理。这样导致的结果是，我们对训练集执行的所有操作，都必须对测试集执行一次，工作量是翻倍的。

```
#是否有样本不平衡问题？
Ytrain.value_counts()
Ytest.value_counts()

#将标签编码
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder().fit(Ytrain)
Ytrain = pd.DataFrame(encoder.transform(Ytrain))
Ytest = pd.DataFrame(encoder.transform(Ytest))
```

4.3 探索特征，开始处理特征矩阵

4.3.1 描述性统计与异常值

```
#描述性统计
Xtrain.describe([0.01,0.05,0.1,0.25,0.5,0.75,0.9,0.99]).T
Xtest.describe([0.01,0.05,0.1,0.25,0.5,0.75,0.9,0.99]).T
.....
对于去kaggle上下载了数据的小伙伴们，以及坚持要使用完整版数据的（15w行）小伙伴们
如果你发现了异常值，首先你要观察，这个异常值出现的频率
如果异常值只出现了一次，多半是输入错误，直接把异常值删除
如果异常值出现了多次，去跟业务人员沟通，可能这是某种特殊表示，如果是人为造成的错误，异常值留着是没有用的，只要数据量不是太大，都可以删除
如果异常值占到你总数据量的10%以上了，不能轻易删除。可以考虑把异常值替换成非异常但是非干扰的项，比如说用0来进行替换，或者把异常当缺失值，用均值或者众数来进行替换
.....

#先查看原始的数据结构
Xtrain.shape
Xtest.shape

#观察异常值是大量存在，还是少数存在
Xtrain.loc[Xtrain.loc[:, "Cloud9am"] == 9, "Cloud9am"]
Xtest.loc[Xtest.loc[:, "Cloud9am"] == 9, "Cloud9am"]
Xtest.loc[Xtest.loc[:, "Cloud3pm"] == 9, "Cloud3pm"]

#少数存在，于是采取删除的策略
#注意如果删除特征矩阵，则必须连对应的标签一起删除，特征矩阵的行和标签的行必须要一一对应
Xtrain = Xtrain.drop(index = 71737)
Ytrain = Ytrain.drop(index = 71737)

#删除完毕之后，观察原始的数据结构，确认删除正确
Xtrain.shape

Xtest = Xtest.drop(index = [19646, 29632])
Ytest = Ytest.drop(index = [19646, 29632])
Xtest.shape

#进行任何行删除之后，千万记得要恢复索引
for i in [Xtrain, Xtest, Ytrain, Ytest]:
    i.index = range(i.shape[0])

Xtrain.head()
Xtest.head()
```

4.3.2 处理困难特征：日期

我们采集数据的日期是否和我们的天气有关系呢？我们可以探索一下我们的采集日期有什么样的性质：

```
Xtrainc = Xtrain.copy()
```

```
xtrainc.sort_values(by="Location")

xtrain.iloc[:,0].value_counts()
#首先，日期不是独一无二的，日期有重复
#其次，在我们分训练集和测试集之后，日期也不是连续的，而是分散的
#某一年的某一天倾向于会下雨？或者倾向于不会下雨吗？
#不是日期影响了下雨与否，反而更多的是这一天的日照时间，湿度，温度等等这些因素影响了是否会下雨
#光看日期，其实感觉它对我们的判断并无直接影响
#如果我们把它当作连续型变量处理，那算法会认为它是一系列1~3000左右的数字，不会意识到这是日期

xtrain.iloc[:,0].value_counts().count()
#如果我们把它当作分类型变量处理，类别太多，有2141类，如果换成数值型，会被直接当成连续型变量，如果做成哑
变量，我们特征的维度会爆炸
```

如果我们的思考简单一些，我们可以直接删除日期这个特征。首先它不是一个直接影响我们标签的特征，并且要处理日期其实是非常困难的。如果大家认可这种思路，那可以直接运行下面的代码来删除我们的日期：

```
xtrain = xtrain.drop(["Date"],axis=1)
xtest = xtest.drop(["Date"],axis=1)
```

但在这里，很多人可能会持不同意见，怎么能够随便删除一个特征（哪怕我们已经觉得它可能无关）？如果我们要删除，我们可能需要一些统计过程，来判断说这个特征确实是和标签无关的，那我们可以先将“日期”这个特征编码后对它和标签做方差齐性检验（ANOVA），如果检验结果表示日期这个特征的确和我们的标签无关，那我们就可以愉快地删除这个特征了。但要编码“日期”这个特征，就又回到了它到底是否会被算法当成是分类变量的问题上。

其实我们可以想到，日期必然是和我们的结果有关的，它会从两个角度来影响我们的标签：

首先，我们可以想到，昨天的天气可能会影响今天的天气，而今天的天气又可能会影响明天的天气。也就是说，随着日期的逐渐改变，样本是会受到上一个样本的影响的。但是对于算法来说，普通的算法是无法捕捉到样本与样本之间的联系的，我们的算法捕捉的是样本的每个特征与标签之间的联系（即列与列之间的联系），而无法捕捉样本与样本之间的联系（行与行的联系）。

要让算法理解上一个样本的标签可能会影响下一个样本的标签，我们必须使用时间序列分析。时间序列分析是指将同一统计指标的数值按其发生的时间先后顺序排列而成的数列。时间序列分析的主要目的是根据已有的历史数据对未来进行预测。然而，（据我所知）时间序列只能在单调的，唯一的时间上运行，即一次只能够对一个地点进行预测，不能够实现一次性预测多个地点，除非进行循环。而我们的时间数据本身，不是单调的，也不是唯一的，经过抽样之后，甚至连连续的都不是了，我们的时间是每个混杂在多个地点中，每个地点上的一小段时间。如何使用时间序列来处理这个问题，就会变得复杂。

那我们可以换一种思路，既然算法处理的是列与列之间的关系，我是否可以把“今天的天气会影响明天的天气”这个指标转换成一个特征呢？我们就这样来操作。

我们观察到，我们的特征中有一列叫做“Rainfall”，这是表示当前日期当前地区下的降雨量，换句话说，也就是“今天的降雨量”。凭常识我们认为，今天是否下雨，应该会影响明天是否下雨，比如有的地方可能就有这样的气候，一旦下雨就连着下很多天，也有可能有的地方的气候就是一场暴雨来得快去得快。因此，我们可以将时间对气候的连续影响，转换为“今天是否下雨”这个特征，巧妙地将样本对应标签之间的联系，转换成是特征与标签之间的联系了。

```
xtrain["Rainfall"].head(20)

xtrain.loc[xtrain["Rainfall"] >= 1, "RainToday"] = "Yes"
xtrain.loc[xtrain["Rainfall"] < 1, "RainToday"] = "No"
xtrain.loc[xtrain["Rainfall"] == np.nan, "RainToday"] = np.nan

xtest.loc[xtest["Rainfall"] >= 1, "RainToday"] = "Yes"
xtest.loc[xtest["Rainfall"] < 1, "RainToday"] = "No"
xtest.loc[xtest["Rainfall"] == np.nan, "RainToday"] = np.nan

xtrain.head()

xtest.head()
```

如此，我们就创造了一个特征，今天是否下雨“RainToday”。

那现在，我们是否就可以将日期删除了呢？对于我们而言，日期本身并不影响天气，但是日期所在的月份和季节其实是影响天气的，如果任选梅雨季节的某一天，那明天下雨的可能性必然比非梅雨季节的那一天要大。虽然我们无法让机器学习体会不同月份是什么季节，但是我们可以对不同月份进行分组，算法可以通过训练感受到，“这个月或者这个季节更容易下雨”。因此，我们可以将月份或者季节提取出来，作为一个特征使用，而舍弃掉具体的日期。如此，我们又可以创造第二个特征，月份“Month”。

```
int(xtrain.loc[0, "Date"].split("-")[1]) #提取出月份

xtrain["Date"] = xtrain["Date"].apply(lambda x:int(x.split("-")[1]))
#替换完毕后，我们需要修改列的名称
#rename是比较少有的，可以用来修改单个列名的函数
#我们通常都直接使用 df.columns = 某个列表 这样的形式来一次修改所有的列名
#但rename允许我们只修改某个单独的列
xtrain = xtrain.rename(columns={"Date":"Month"})

xtrain.head()

xtest["Date"] = Xtest["Date"].apply(lambda x:int(x.split("-")[1]))
Xtest = Xtest.rename(columns={"Date":"Month"})

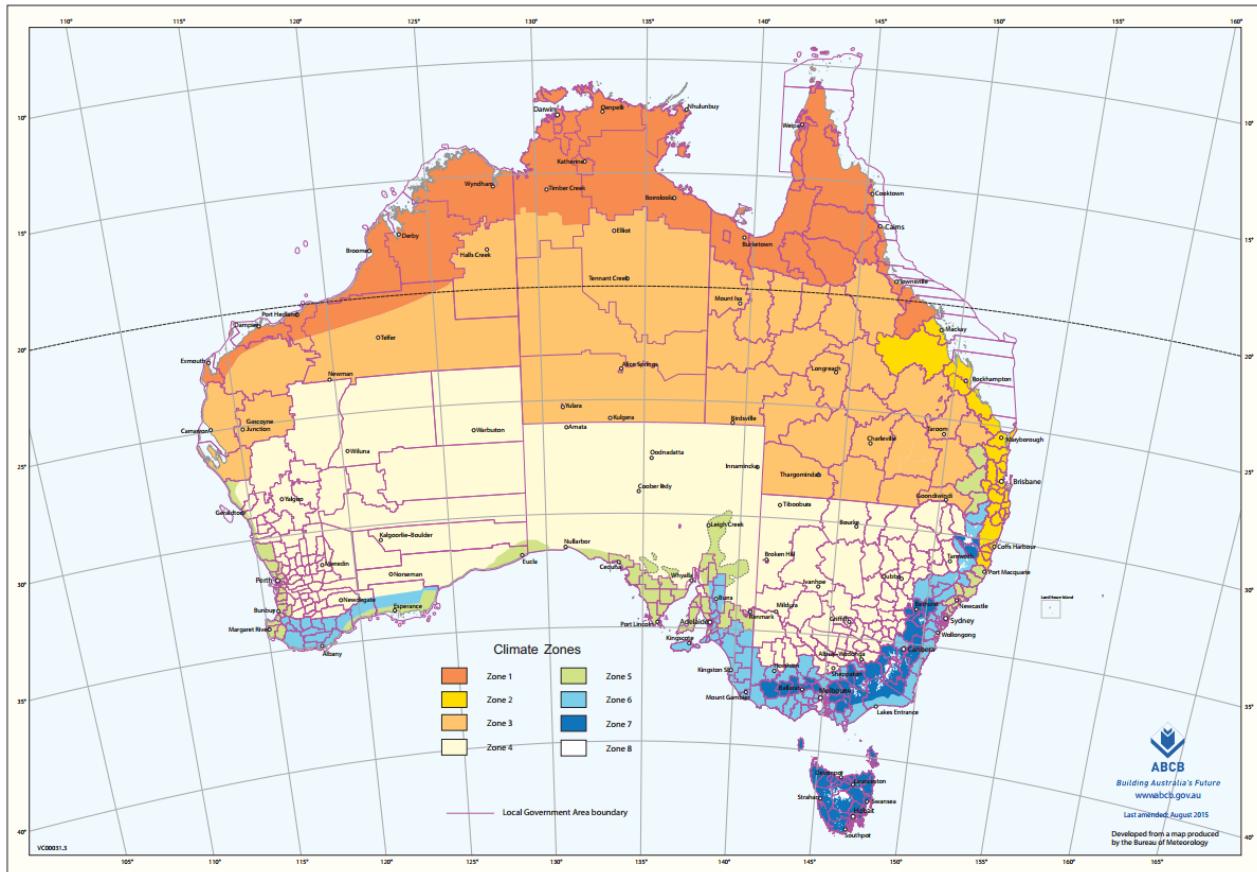
Xtest.head()
```

通过时间，我们处理出两个新特征，“今天是否下雨”和“月份”。接下来，让我们来看看如何处理另一个更加困难的特征，地点。

4.3.3 处理困难特征：地点

地点，又是一个非常tricky的特征。常识上来说，我们认为地点肯定是对明天是否会下雨存在影响的。比如说，如果其他信息都不给出，我们只猜测，“伦敦明天是否会下雨”和“北京明天是否会下雨”，我一定会猜测伦敦会下雨，而北京不会，因为伦敦是常年下雨的城市，而北京的气候非常干燥。对澳大利亚这样面积巨大的国家来说，必然存在着不同的城市有着不同的下雨倾向的情况。但尴尬的是，和时间一样，我们输入地点的名字对于算法来说，就是一串字符，“London”和“Beijing”对算法来说，和0, 1没有区别。同样，我们的样本中含有49个不同地点，如果做成分类型变量，算法就无法辨别它究竟是否是分类变量。也就是说，我们需要让算法意识到，不同的地点因为气候不同，所以对“明天是否会下雨”有着不同的影响。如果我们能够将地点转换为这个地方的气候的话，我们就可以将不同城市打包到同一个气候中，而同一个气候下反应的降雨情况应该是相似的。

那我们如何将城市转换为气候呢？我在google找到了如下地图：



这是由澳大利亚气象局和澳大利亚建筑规范委员会（ABCB）制作统计的，澳大利亚不同地区不同城市的所在的气候区域划分。总共划分为八个区域，非常适合我们用来做分类。如果能够把49个地点转换成八种不同的气候，这个信息应该会对是否下雨的判断比较有用。基于气象局和ABCB的数据，我为大家制作了澳大利亚主要城市所对应的气候类型数据，并保存在csv文件city_climate.csv当中。然后，我使用以下代码，在google上进行爬虫，爬出了每个城市所对应的经纬度，并保存在数据cityll.csv当中，大家可以自行导入，来查看这个数据。

爬虫的过程，我录制成了短视频，详细的解释和操作大家可以在视频里看到：

<https://www.bilibili.com/video/av39338080/>

爬虫的代码如下所示，大家可以把谷歌的主页换成百度，修改一下爬虫的命令，就可以自己试试看这段代码。**注意要先定义你需要爬取的城市名称的列表cityname哦。**

```
import time
from selenium import webdriver #导入需要的模块，其中爬虫使用的是selenium
import pandas as pd
import numpy as np

df = pd.DataFrame(index=range(len(cityname))) #创建新dataframe用于存储爬取的数据

driver = webdriver.Chrome() #调用谷歌浏览器

time0 = time.time() #计时开始

#循环开始
for num, city in enumerate(cityname): #在城市名称中进行遍历
```

```

driver.get('https://www.google.co.uk/webhp?
hl=en&sa=X&ved=0ahUKEwimtcX24cTfAhUJE7wKHVkB5AQPAgH')
    #首先打开谷歌主页
time.sleep(0.3)
    #停留0.3秒让我们知道发生了什么
search_box = driver.find_element_by_name('q') #锁定谷歌的搜索输入框
search_box.send_keys('%s Australia Latitude and longitude' % (city)) #在输入框中输入
“城市” 澳大利亚 纬度
search_box.submit() #enter, 确认开始搜索
result = driver.find_element_by_xpath('//div[@class="z0Lcw"]').text #? 爬取需要的经纬
度, 就是这里, 怎么获取的呢?
resultsplit = result.split(" ") #将爬取的结果用split进行分割
df.loc[num, "City"] = city #向提前创建好的df中输入爬取的数据, 第一列是城市名
df.loc[num, "Latitude"] = resultsplit[0] #第二列是纬度
df.loc[num, "Longitude"] = resultsplit[2] #第三列是经度
df.loc[num, "Latitudedir"] = resultsplit[1] #第四列是纬度的方向
df.loc[num, "Longitudedir"] = resultsplit[3] #第五列是经度的方向
print("%i webcrawler successful for city %s" % (num,city)) #每次爬虫成功之后, 就打印“城
市”成功了

time.sleep(1) #全部爬取完毕后, 停留1秒钟
driver.quit() #关闭浏览器
print(time.time() - time0) #打印所需的时间

```

为什么我们会需要城市的经纬度呢？我曾经尝试过直接使用样本中的城市来爬取城市本身的气候，然而由于样本中的地点名称，其实是气候站的名称，而不是城市本身的名称，因此不是每一个城市都能够直接获取到城市的气候。比如说，如果我们搜索“海淀区气候”，搜索引擎返回的可能是海淀区现在的气温，而不是整个北京的气候类型。因此，我们需要澳大利亚气象局的数据，来找到这些气候站所对应的城市。

我们有了澳大利亚全国主要城市的气候，也有了澳大利亚主要城市的经纬度（地点），我们就可以通过计算我们样本中的每个气候站到各个主要城市的地理距离，来找出一个离这个气象站最近的主要城市，而这个主要城市的气候就是我们样本点所在的地点的气候。

让我们把cityll.csv和cityclimate.csv来导入，来看看它们是什么样子：

```

cityll = pd.read_csv(r"C:\work\Learnbetter\micro-class\week 8 SVM
(2)\cityll.csv",index_col=0)
city_climate = pd.read_csv(r"C:\work\Learnbetter\micro-class\week 8 SVM
(2)\cityclimate.csv")

cityll.head()
city_climate.head()

```

接下来，我们来将这两张表处理成可以使用的样子，首先要去掉cityll中经纬度上带有的度数符号，然后要将两张表合并起来。

```
#去掉度数符号
cityll[\"Latitudenum\"] = cityll[\"Latitude\"].apply(lambda x:float(x[:-1]))
cityll[\"Longitudenum\"] = cityll[\"Longitude\"].apply(lambda x:float(x[:-1]))

#观察一下所有的经纬度方向都是一致的，全部是南纬，东经，因为澳大利亚在南半球，东半球
#所以经纬度的方向我们可以舍弃了
citylld = cityll.iloc[:,[0,5,6]]

#将city_climate中的气候添加到我们的citylld中
citylld[\"climate\"] = city_climate.iloc[:, -1]

citylld.head()
```

接下来，我们如果想要计算距离，我们就会需要所有样本数据中的城市。我们认为，只有出现在训练集中的地点才会出现在测试集中，基于这样的假设，我们来爬取训练集中所有的地点所对应的经纬度，并且保存在一个csv文件samplecity.csv中：

```
#训练集中所有的地点
cityname = Xtrain.iloc[:,1].value_counts().index.tolist()

cityname

import time
from selenium import webdriver #导入需要的模块，其中爬虫使用的是selenium
import pandas as pd
import numpy as np

df = pd.DataFrame(index=range(len(cityname))) #创建新dataframe用于存储爬取的数据

driver = webdriver.Chrome() #调用谷歌浏览器

time0 = time.time() #计时开始

#循环开始
for num, city in enumerate(cityname): #在城市名称中进行遍历
    driver.get('https://www.google.co.uk/webhp?hl=en&sa=X&ved=0ahUKEwimtcX24cTfAhUJE7wKHVkB5AQPAgH')
        #首先打开谷歌主页
    time.sleep(0.3)
        #停留0.3秒让我们知道发生了什么
    search_box = driver.find_element_by_name('q') #锁定谷歌的搜索输入框
    search_box.send_keys('%s Australia Latitude and longitude' % (city)) #在输入框中输入“城市” 澳大利亚 经纬度
    search_box.submit() #enter, 确认开始搜索
    result = driver.find_element_by_xpath('//div[@class="zOLcw"]').text #? 爬取需要的经纬度，就是这里，怎么获取的呢？
    resultsplit = result.split(" ") #将爬取的结果用split进行分割
    df.loc[num, "City"] = city #向提前创建好的df中输入爬取的数据，第一列是城市名
    df.loc[num, "Latitude"] = resultsplit[0] #第二列是经度
    df.loc[num, "Longitude"] = resultsplit[2] #第三列是纬度
    df.loc[num, "Latitudedir"] = resultsplit[1] #第四列是经度的方向
    df.loc[num, "Longitudedir"] = resultsplit[3] #第五列是纬度的方向
```

```

print("%i webcrawler successful for city %s" % (num,city)) #每次爬虫成功之后，就打印“城市”成功了

time.sleep(1) #全部爬取完毕后，停留1秒钟
driver.quit() #关闭浏览器
print(time.time() - time0) #打印所需的时间

df.to_csv(r"C:\work\learnbetter\micro-class\week 8 SVM (2)\samplecity.csv")

```

来查看一下我们爬取出的内容是什么样子：

```

samplecity = pd.read_csv(r"C:\work\learnbetter\micro-class\week 8 SVM (2)\samplecity.csv",index_col=0)

#我们对samplecity也执行同样的处理：去掉经纬度中度数的符号，并且舍弃我们的经纬度的方向

samplecity["Latitudenum"] = samplecity["Latitude"].apply(lambda x:float(x[:-1]))
samplecity["Longitudenum"] = samplecity["Longitude"].apply(lambda x:float(x[:-1]))

samplecityd = samplecity.iloc[:,[0,5,6]]

samplecityd.head()

```

好了，我们现在有了澳大利亚主要城市的经纬度和对应的气候，也有了我们的样本的地点所对应的经纬度，接下来我们要开始计算我们样本上的地点到每个澳大利亚主要城市的距离，而离我们的样本点最近的那个澳大利亚主要城市的气候，就是我们样本点的气候。

在地理上，两个地点之间的距离，由如下公式来进行计算：

$$dist = R * \arccos(\sin(slat) * \sin(elat) + \cos(slat) * \cos(elat) * \cos(slon - elon))$$

其中R是地球的半径，6371.01km，arccos是三角反余弦函数，slat是起始地点的纬度，slon是起始地点的经度，elat是结束地点的纬度，elon是结束地点的经度。本质还是计算两点之间的距离。而我们爬取的经纬度，本质其实是角度，所以需要用各种三角函数和弧度公式将角度转换成距离。由于我们不是地理专业，拿到公式可以使用就okay了，不需要去纠结这个公式究竟怎么来的。

```

#首先使用radians将角度转换成弧度

from math import radians, sin, cos, acos
city11d.loc[:, "slat"] = city11d.iloc[:, 1].apply(lambda x : radians(x))
city11d.loc[:, "slon"] = city11d.iloc[:, 2].apply(lambda x : radians(x))
samplecityd.loc[:, "elat"] = samplecityd.iloc[:, 1].apply(lambda x : radians(x))
samplecityd.loc[:, "elon"] = samplecityd.iloc[:, 2].apply(lambda x : radians(x))

import sys
for i in range(samplecityd.shape[0]):
    slat = city11d.loc[:, "slat"]
    slon = city11d.loc[:, "slon"]
    elat = samplecityd.loc[i, "elat"]
    elon = samplecityd.loc[i, "elon"]
    dist = 6371.01 * np.arccos(np.sin(slat)*np.sin(elat) +
                               np.cos(slat)*np.cos(elat)*np.cos(slon.values - elon))

```

```

city_index = np.argsort(dist)[0]
#每次计算后，取距离最近的城市，然后将最近的城市和城市对应的气候都匹配到samplecityd中
samplecityd.loc[i,"closest_city"] = city11d.loc[city_index,"City"]
samplecityd.loc[i,"climate"] = city11d.loc[city_index,"climate"]

#查看最后的结果，需要检查城市匹配是否基本正确
samplecityd.head()

#查看气候的分布
samplecityd["climate"].value_counts()

#确认无误后，取出样本城市所对应的气候，并保存
locfinal = samplecityd.iloc[:,[0,-1]]

locfinal.head()

locfinal.columns = ["Location","Climate"]

#在这里设定locfinal的索引为地点，是为了之后进行map的匹配
locfinal = locfinal.set_index(keys="Location")

locfinal.to_csv(r"C:\work\learnbetter\micro-class\week 8 SVM (2)\samplelocation.csv")

locfinal.head()

```

有了每个样本城市所对应的气候，我们接下来就使用气候来替掉原本的城市，原本的气象站的名称。在这里，我们可以使用map功能，map能够将特征中的值一一对应到我们设定的字典中，并且用字典中的值来替换样本中原本的值，我们在评分卡中曾经使用这个功能来用WOE替换我们原本的特征的值。

```

#是否还记得训练集长什么样呢？
xtrain.head()

#将location中的内容替换，并且确保匹配进入的气候字符串中不含有逗号，气候两边不含有空格
#我们使用re这个模块来消除逗号
#re.sub(希望替换的值，希望被替换成的值，要操作的字符串)
#x.strip()是去掉空格的函数
import re
xtrain["Location"] = xtrain["Location"].map(locfinal.iloc[:,0]).apply(lambda
x:re.sub(",","",x.strip()))
xtest["Location"] = xtest["Location"].map(locfinal.iloc[:,0]).apply(lambda
x:re.sub(",","",x.strip()))

#修改特征内容之后，我们使用新列名“Climate”来替换之前的列名“Location”
#注意这个命令一旦执行之后，就再没有列"Location"了，使用索引时要特别注意
xtrain = xtrain.rename(columns={"Location":"Climate"})
xtest = xtest.rename(columns={"Location":"Climate"})

xtrain.head()
xtest.head()

```

到这里，地点就处理完毕了。其实，我们还没有将这个特征转化为数字，即还没有对它进行编码。我们稍后和其他的分类型变量一起来编码。

4.3.4 处理分类型变量：缺失值

接下来，我们总算可以开始处理我们的缺失值了。首先我们要注意到，由于我们的特征矩阵由两种类型的数据组成：分类型和连续型，因此我们必须对两种数据采用不同的填补缺失值策略。传统地，如果是分类型特征，我们则采用众数进行填补。如果是连续型特征，我们则采用均值来填补。

此时，由于我们已经分了训练集和测试集，我们需要考虑一件事：究竟使用哪一部分的数据进行众数填补呢？答案是，使用训练集上的众数对训练集和测试集都进行填补。为什么会这样呢？按道理说就算用测试集上的众数对测试集进行填补，也不会使测试集数据进入我们建好的模型，不会给模型透露一些信息。然而，在现实中，我们的测试集未必是很多条数据，也许我们的测试集只有一条数据，而某个特征上是空值，此时此刻测试集本身的众数根本不存在，要如何利用测试集本身的众数去进行填补呢？因此为了避免这种尴尬的情况发生，我们假设测试集和训练集的数据分布和性质都是相似的，因此我们统一使用训练集的众数和均值来对测试集进行填补。

在sklearn当中，即便是我们的填补缺失值的类也需要由实例化，fit和接口调用执行填补三个步骤来进行，而这种分割其实一部分也是为了满足我们使用训练集的建模结果来填补测试集的需求。我们只需要实例化后，使用训练集进行fit，然后在调用接口执行填补时用训练集fit后的结果分别来填补测试集和训练集就可以了。

```
#查看缺失值的缺失情况
xtrain.isnull().mean()

#首先找出，分类型特征都有哪些
cate = xtrain.columns[xtrain.dtypes == "object"].tolist()

#除了特征类型为"object"的特征们，还有虽然用数字表示，但是本质为分类型特征的云层遮蔽程度
cloud = ["Cloud9am", "Cloud3pm"]
cate = cate + cloud
cate

#对于分类型特征，我们使用众数来进行填补
from sklearn.impute import SimpleImputer

si = SimpleImputer(missing_values=np.nan, strategy="most_frequent")
#注意，我们使用训练集数据来训练我们的填补器，本质是在生成训练集中的众数
si.fit(xtrain.loc[:,cate])

#然后我们用训练集中的众数来同时填补训练集和测试集
xtrain.loc[:,cate] = si.transform(xtrain.loc[:,cate])
xtest.loc[:,cate] = si.transform(xtest.loc[:,cate])

xtrain.head()
xtest.head()

#查看分类型特征是否依然存在缺失值
xtrain.loc[:,cate].isnull().mean()
xtest.loc[:,cate].isnull().mean()
```

4.3.5 处理分类型变量：将分类型变量编码

在编码中，和我们的填补缺失值一样，我们也是需要先用训练集fit模型，本质是将训练集中已经存在的类别转换成是数字，然后我们再使用接口transform分别在测试集和训练集上来编码我们的特征矩阵。当我们使用接口在测试集上进行编码的时候，如果测试集上出现了训练集中从未出现过的类别，那代码就会报错，表示说“我没有见过这个类别，我无法对这个类别进行编码”，此时此刻你就要思考，你的测试集上或许存在异常值，错误值，或者的确

有一个新的类别出现了，而你曾经的训练数据中并没有这个类别。以此为基础，你需要调整你的模型。

```
#将所有的分类型变量编码为数字，一个类别是一个数字
from sklearn.preprocessing import OrdinalEncoder
oe = OrdinalEncoder()

#利用训练集进行fit
oe = oe.fit(xtrain.loc[:,cate])

#用训练集的编码结果来编码训练和测试特征矩阵
#在这里如果测试特征矩阵报错，就说明测试集中出现了训练集中从未见过的类别
xtrain.loc[:,cate] = oe.transform(xtrain.loc[:,cate])
xtest.loc[:,cate] = oe.transform(xtest.loc[:,cate])

xtrain.loc[:,cate].head()
xtest.loc[:,cate].head()
```

4.3.6 处理连续型变量：填补缺失值

连续型变量的缺失值由均值来进行填补。连续型变量往往已经是数字，无需进行编码转换。与分类型变量中一样，我们也是使用训练集上的均值对测试集进行填补。如果学过随机森林填补缺失值的小伙伴，可能此时会问，为什么不使用算法来进行填补呢？使用算法进行填补也是没有问题的，但在现实中，其实我们非常少用到算法来进行填补，有以下几个理由：

1. 算法是黑箱，解释性不强。如果你是一个数据挖掘工程师，你使用算法来填补缺失值后，你不懂机器学习的老板或者同事问你的缺失值是怎么来的，你可能需要从头到尾帮他/她把随机森林解释一遍，这种效率过低的事情是不可能做的，而许多老板和上级不会接受他们无法理解的东西。
2. 算法填补太过缓慢，运行一次森林需要有至少100棵树才能够基本保证森林的稳定性，而填补一个列就需要很长的时间。在我们并不知道森林的填补结果是好是坏的情况下，填补一个很大的数据集风险非常高，有可能需要跑好几个小时，但填补出来的结果却不怎么优秀，这明显是一个低效的方法。

因此在现实工作时，我们往往使用易于理解的均值或者中位数来进行填补。当然了，在算法比赛中，我们可以穷尽一切我们能够想到的办法来填补缺失值以追求让模型的效果更好，不过现实中，除了模型效果之外，我们还要追求可解释性。

```
col = xtrain.columns.tolist()

for i in cate:
    col.remove(i)

col

#实例化模型，填补策略为"mean"表示均值
impmean = SimpleImputer(missing_values=np.nan,strategy = "mean")
#用训练集来fit模型
impmean = impmean.fit(xtrain.loc[:,col])
#分别在训练集和测试集上进行均值填补
xtrain.loc[:,col] = impmean.transform(xtrain.loc[:,col])
xtest.loc[:,col] = impmean.transform(xtest.loc[:,col])

xtrain.head()
xtest.head()
```

4.3.7 处理连续型变量：无量纲化

数据的无量纲化是SVM执行前的重要步骤，因此我们需要对数据进行无量纲化。但注意，这个操作我们不对分类型变量进行。

```
col.remove("Month")

col

from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
ss = ss.fit(Xtrain.loc[:, col])
Xtrain.loc[:, col] = ss.transform(Xtrain.loc[:, col])
Xtest.loc[:, col] = ss.transform(Xtest.loc[:, col])

Xtrain.head()
Xtest.head()

Ytrain.head()
Ytest.head()
```

特征工程到这里就全部结束了。大家可以分别查看一下我们的Ytrain, Ytest, Xtrain, Xtest, 确保我们熟悉他们的结构并且确保我们的确已经处理完毕全部的内容。将数据处理完毕之后，建议大家都使用to_csv来保存我们已经处理好的数据集，避免我们在后续建模过程中出现覆盖了原有的数据集的失误后，需要从头开始做数据预处理。在开始建模之前，无比保存好处理好的数据，然后在建模的时候，重新将数据导入。

4.4 建模与模型评估

```
from time import time
import datetime
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
from sklearn.metrics import roc_auc_score, recall_score

Ytrain = Ytrain.iloc[:, 0].ravel()
Ytest = Ytest.iloc[:, 0].ravel()

#建模选择自然是我们的支持向量机SVC，首先用核函数的学习曲线来选择核函数
#我们希望同时观察，精确性，recall以及AUC分数
times = time() #因为SVM是计算量很大的模型，所以我们需要时刻监控我们的模型运行时间

for kernel in ["linear", "poly", "rbf", "sigmoid"]:
    clf = SVC(kernel=kernel
               , gamma="auto"
               , degree=1
               , cache_size=5000
               ).fit(Xtrain, Ytrain)
    result = clf.predict(Xtest)
    score = clf.score(Xtest, Ytest)
```

```

recall = recall_score(Ytest, result)
auc = roc_auc_score(Ytest,clf.decision_function(Xtest))
print("%s 's testing accuracy %f, recall is %f, auc is %f" %
(kernel,score,recall,auc))
print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))

```

我们注意到，模型的准确度和auc面积还是勉勉强强，但是每个核函数下的recall都不太高。相比之下，其实线性模型的效果是最好的。那现在我们可以开始考虑了，在这种状况下，我们要向着什么方向进行调参呢？我们最想要的是什么？

我们可以有不同的目标：

- 一，我希望不计一切代价判断出少数类，得到最高的recall。
- 二，我们希望追求最高的预测准确率，一切目的都是为了让accuracy更高，我们不在意recall或者AUC。
- 三，我们希望达到recall，ROC和accuracy之间的平衡，不追求任何一个也不牺牲任何一个。

4.5 模型调参

4.5.1 最求最高Recall

如果我们想要的是最高的recall，可以牺牲我们准确度，希望不计一切代价来捕获少数类，那我们首先可以打开我们的class_weight参数，使用balanced模式来调节我们的recall：

```

times = time()
for kernel in ["linear", "poly", "rbf", "sigmoid"]:
    clf = SVC(kernel = kernel
               ,gamma="auto"
               ,degree = 1
               ,cache_size = 5000
               ,class_weight = "balanced"
               ).fit(Xtrain, Ytrain)
    result = clf.predict(Xtest)
    score = clf.score(Xtest,Ytest)
    recall = recall_score(Ytest, result)
    auc = roc_auc_score(Ytest,clf.decision_function(Xtest))
    print("%s 's testing accuracy %f, recall is %f, auc is %f" %
(kernel,score,recall,auc))
    print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))

```

在锁定了线性核函数之后，我甚至可以将class_weight调节得更加倾向于少数类，来不计代价提升recall。

```

times = time()
clf = SVC(kernel = "linear"
           ,gamma="auto"
           ,cache_size = 5000
           ,class_weight = {1:10} #注意，这里写的是类别1: 10，隐藏了类别0: 1这个比例
           ).fit(Xtrain, Ytrain)
result = clf.predict(Xtest)
score = clf.score(Xtest,Ytest)
recall = recall_score(Ytest, result)
auc = roc_auc_score(Ytest,clf.decision_function(Xtest))
print("testing accuracy %f, recall is %f", auc is %f" %(score,recall,auc))
print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))

```

随着recall地无节制上升，我们的精确度下降得十分厉害，不过看起来AUC面积却还好，稳定保持在0.86左右。如果此时我们的目的就是追求一个比较高的AUC分数和比较好的recall，那我们的模型此时就算是很不错了。虽然现在，我们的精确度很低，但是我们的确精准地捕捉出了每一个雨天。

4.5.2 追求最高准确率

在我们现有的目标（判断明天是否会下雨）下，追求最高准确率而不顾recall其实意义不大，但出于练习的目的，我们来看看我们能够有怎样的思路。此时此刻我们不在意我们的Recall了，那我们首先要观察一下，我们的样本不均衡状况。如果我们的样本非常不均衡，但是此时却有很多多数类被判错的话，那我们可以让模型任性地把所有地样本都判断为0，完全不顾少数类。

```

valuec = pd.Series(Ytest).value_counts()

valuec

valuec[0]/valuec.sum()

```

初步判断，可以认为我们其实已经将大部分的多数类判断正确了，所以才能够得到现在的正确率。为了证明我们的判断，我们可以使用混淆矩阵来计算我们的特异度，如果特异度非常高，则证明多数类上已经很难被操作了。

```

#查看模型的特异度

from sklearn.metrics import confusion_matrix as CM
clf = SVC(kernel = "linear"
           ,gamma="auto"
           ,cache_size = 5000
           ).fit(Xtrain, Ytrain)
result = clf.predict(Xtest)

cm = CM(Ytest,result,labels=(1,0))

cm

specificity = cm[1,1]/cm[1,:].sum()

specificity #几乎所有的0都被判断正确了，还有不少1也被判断正确了

```

可以看到，特异度非常高，此时此刻如果要求模型将所有的类都判断为0，则已经被判断正确的少数类会被误伤，整体的准确率一定会下降。而如果我们希望通过让模型捕捉更多少数类来提升精确率的话，却无法实现，因为一旦我们让模型更加倾向于少数类，就会有更多的多数类被判错。

可以试试看使用class_weight将模型向少数类的方向稍微调整，已查看我们是否有更多的空间来提升我们的准确率。如果在轻微向少数类方向调整过程中，出现了更高的准确率，则说明模型还没有到极限。

```
irange = np.linspace(0.01, 0.05, 10)

for i in irange:
    times = time()
    clf = SVC(kernel = "linear"
               ,gamma="auto"
               ,cache_size = 5000
               ,class_weight = {1:1+i}
               ).fit(Xtrain, Ytrain)
    result = clf.predict(Xtest)
    score = clf.score(Xtest, Ytest)
    recall = recall_score(Ytest, result)
    auc = roc_auc_score(Ytest, clf.decision_function(Xtest))
    print("under ratio 1:%f testing accuracy %f, recall is %f', auc is %f" %
(1+i,score,recall,auc))
    print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))
```

惊喜出现了，我们的最高准确度是84.53%，超过了我们之前什么都不做的时候得到的84.40%。可见，模型还是有潜力的。我们可以继续细化我们的学习曲线来进行调整：

```
irange_ = np.linspace(0.018889, 0.027778, 10)

for i in irange_:
    times = time()
    clf = SVC(kernel = "linear"
               ,gamma="auto"
               ,cache_size = 5000
               ,class_weight = {1:1+i}
               ).fit(Xtrain, Ytrain)
    result = clf.predict(Xtest)
    score = clf.score(Xtest, Ytest)
    recall = recall_score(Ytest, result)
    auc = roc_auc_score(Ytest, clf.decision_function(Xtest))
    print("under ratio 1:%f testing accuracy %f, recall is %f', auc is %f" %
(1+i,score,recall,auc))
    print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))
```

模型的效果没有太好，并没有再出现比我们的84.53%精确度更高的取值。可见，模型在不做样本平衡的情况下，准确度其实已经非常接近极限了，让模型向着少数类的方向调节，不能够达到质变。如果我们真的希望再提升准确度，只能选择更换模型的方式，调整参数已经不能够帮助我们了。想想看什么模型在线性数据上表现最好呢？

```

from sklearn.linear_model import LogisticRegression as LR

logclf = LR(solver="liblinear").fit(Xtrain, Ytrain)
logclf.score(Xtest, Ytest)

C_range = np.linspace(3, 5, 10)

for C in C_range:
    logclf = LR(solver="liblinear", C=C).fit(Xtrain, Ytrain)
    print(C, logclf.score(Xtest, Ytest))

```

尽管我们实现了非常小的提升，但可以看出来，模型的精确度还是没有能够实现质变。也许，要将模型的精确度提升到90%以上，我们需要集成算法：比如，梯度提升树。大家如果感兴趣，可以自己下去试试看。

4.5.3 追求平衡

我们前面经历了多种尝试，选定了线性核，并发现调节class_weight并不能够使我们模型有较大的改善。现在我们来试试看调节线性核函数的C值能否有效果：

```

##### 【TIME WARNING: 10mins】 #####
import matplotlib.pyplot as plt
C_range = np.linspace(0.01, 20, 20)

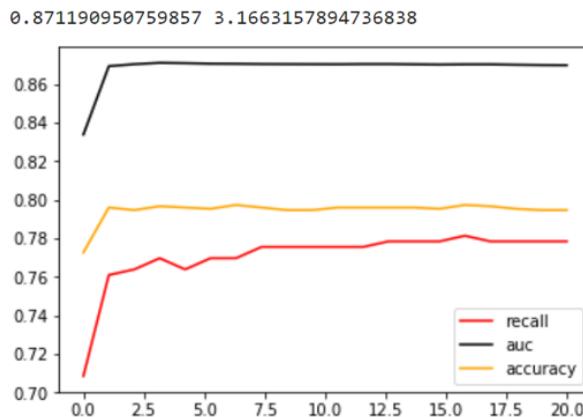
recallall = []
aucall = []
scoreall = []
for C in C_range:
    times = time()
    clf = SVC(kernel = "linear", C=C, cache_size = 5000
               , class_weight = "balanced"
               ).fit(Xtrain, Ytrain)
    result = clf.predict(Xtest)
    score = clf.score(Xtest, Ytest)
    recall = recall_score(Ytest, result)
    auc = roc_auc_score(Ytest, clf.decision_function(Xtest))
    recallall.append(recall)
    aucall.append(auc)
    scoreall.append(score)
    print("under C %f, testing accuracy is %f, recall is %f", auc is %f" %
(c, score, recall, auc))
    print(datetime.datetime.fromtimestamp(time() - times).strftime("%M:%S:%f"))

print(max(aucall), C_range[aucall.index(max(aucall))])
plt.figure()
plt.plot(C_range, recallall, c="red", label="recall")
plt.plot(C_range, aucall, c="black", label="auc")
plt.plot(C_range, scoreall, c="orange", label="accuracy")
plt.legend()
plt.show()

```

这段代码运行大致需要10分钟时间，因此我给大家我展现一下我运行出来的结果：

```
testing accuracy 0.772667,recall is 0.708455', auc is 0.833897
00:00:849718
testing accuracy 0.796000,recall is 0.760933', auc is 0.869442
00:03:936478
testing accuracy 0.794667,recall is 0.763848', auc is 0.870473
00:06:983325
testing accuracy 0.796667,recall is 0.769679', auc is 0.871191
00:09:971297
testing accuracy 0.796000,recall is 0.763848', auc is 0.871010
00:11:635886
testing accuracy 0.795333,recall is 0.769679', auc is 0.870740
00:13:774190
testing accuracy 0.797333,recall is 0.769679', auc is 0.870677
00:17:560051
testing accuracy 0.796000,recall is 0.775510', auc is 0.870548
00:18:792739
testing accuracy 0.794667,recall is 0.775510', auc is 0.870501
00:21:729467
testing accuracy 0.794667,recall is 0.775510', auc is 0.870473
00:24:741792
testing accuracy 0.796000,recall is 0.775510', auc is 0.870432
00:28:398084
testing accuracy 0.796000,recall is 0.775510', auc is 0.870543
00:32:642712
testing accuracy 0.796000,recall is 0.778426', auc is 0.870538
00:32:388473
testing accuracy 0.796000,recall is 0.778426', auc is 0.870407
00:33:395677
testing accuracy 0.795333,recall is 0.778426', auc is 0.870281
00:37:999388
testing accuracy 0.797333,recall is 0.781341', auc is 0.870390
00:37:636358
testing accuracy 0.796667,recall is 0.778426', auc is 0.870392
00:47:516797
testing accuracy 0.795333,recall is 0.778426', auc is 0.870198
00:47:939767
testing accuracy 0.794667,recall is 0.778426', auc is 0.870024
00:52:270030
testing accuracy 0.794667,recall is 0.778426', auc is 0.869944
00:53:789153
```



可以观察到几个现象。

首先，我们注意到，随着C值逐渐增大，模型的运行速度变得越来越慢。对于SVM这个本来运行就不快的模型来说，巨大的C值会是一个比较危险的消耗。所以正常来说，我们应该设定一个较小的C值范围来进行调整。

其次，C很小的时候，模型的各项指标都很低，但当C到1以上之后，模型的表现开始逐渐稳定，在C逐渐变大之后，模型的效果并没有显著地提高。可以认为我们设定的C值范围太大了，然而再继续增大或者缩小C值的范围，AUC面积也只能在0.86上下进行变化了，调节C值不能够让模型的任何指标实现质变。

我们把目前为止最佳的C值带入模型，看看我们的准确率，Recall的具体值：

```
times = time()
clf = SVC(kernel = "linear", C=3.1663157894736838, cache_size = 5000
           , class_weight = "balanced"
           ).fit(Xtrain, Ytrain)
result = clf.predict(Xtest)
score = clf.score(Xtest, Ytest)
recall = recall_score(Ytest, result)
auc = roc_auc_score(Ytest, clf.decision_function(Xtest))
print("testing accuracy %f, recall is %f', auc is %f" % (score, recall, auc))
print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))
```

可以看到，这种情况下模型的准确率，Recall和AUC都没有太差，但是也没有太好，这也许就是模型平衡后的一种结果。现在，光是调整支持向量机本身的参数，已经不能够满足我们的需求了，要想让AUC面积更进一步，我们需要绘制ROC曲线，查看我们是否可以通过调整阈值来对这个模型进行改进。

```
from sklearn.metrics import roc_curve as ROC
import matplotlib.pyplot as plt

FPR, Recall, thresholds = ROC(Ytest, clf.decision_function(Xtest), pos_label=1)

area = roc_auc_score(Ytest, clf.decision_function(Xtest))

plt.figure()
plt.plot(FPR, Recall, color='red',
         label='ROC curve (area = %0.2f)' % area)
plt.plot([0, 1], [0, 1], color='black', linestyle='--')
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False Positive Rate')
```

```
plt.ylabel('Recall')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()
```

以此模型作为基础，我们来求解最佳阈值：

```
maxindex = (Recall - FPR).tolist().index(max(Recall - FPR))
thresholds[maxindex]
```

基于我们选出的最佳阈值，我们来认为确定y_predict，并确定在这个阈值下的recall和准确度的值：

```
from sklearn.metrics import accuracy_score as AC

times = time()
clf = SVC(kernel = "linear", C=3.1663157894736838, cache_size = 5000
           , class_weight = "balanced"
           ).fit(Xtrain, Ytrain)

prob = pd.DataFrame(clf.decision_function(Xtest))

prob.loc[prob.iloc[:, 0] >= thresholds[maxindex], "y_pred"] = 1
prob.loc[prob.iloc[:, 0] < thresholds[maxindex], "y_pred"] = 0

prob.loc[:, "y_pred"].isnull().sum()

#检查模型本身的准确度
score = AC(Ytest, prob.loc[:, "y_pred"].values)
recall = recall_score(Ytest, prob.loc[:, "y_pred"])
print("testing accuracy %f, recall is %f" % (score, recall))
print(datetime.datetime.fromtimestamp(time() - times).strftime("%M:%S:%f"))
```

反而还不如我们不调整时的效果好。可见，如果我们追求平衡，那SVC本身的结果就已经非常接近最优结果了。调节阈值，调节参数C和调节class_weight都不一定有效果。但整体来看，我们的模型不是一个糟糕的模型，但这个结果如果提交到kaggle参加比赛是绝对不够的。如果大家感兴趣，还可以更加深入地探索模型，或者换别的方法来处理特征，以达到AUC面积0.9以上，或是准确度或recall都提升到90%以上。

4.6 SVM总结&结语

在两周的学习中，我们逐渐探索了SVC在sklearn中的全貌，我们学习了SVM原理，包括决策边界，损失函数，拉格朗日函数，拉格朗日对偶函数，软间隔硬间隔，核函数以及核函数的各种应用。我们了解了SVC类的各种重要参数，属性和接口，其中参数包括软间隔的惩罚系数C，核函数kernel，核函数的相关参数gamma，coef0和degree，解决样本不均衡的参数class_weight，解决多分类问题的参数decision_function_shape，控制概率的参数probability，控制计算内存的参数cache_size，属性主要包括调用支持向量的属性support_vectors_和查看特征重要性的属性coef_。接口中，我们学习了最核心的decision_function。除此之外，我们介绍了分类模型的模型评估指标：混淆矩阵和ROC曲线，还介绍了部分特征工程和数据预处理的思路。

支持向量机是深奥并且强大的模型，我们还可以在很多地方继续进行探索，能够学到这里的大家都非常棒！希望大家再接再厉，掌握好这个强大的算法，后面的学习中继续加油。

菜菜的scikit-learn课堂09

sklearn中的线性回归大家族

小伙伴们晚上好~o(￣▽￣)ゞ

我是菜菜，这里是我的sklearn课堂第九期，本周的内容是线性回归大家族~

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scikit-learn课堂09

sklearn中的线性回归大家族

1 概述

1.1 线性回归大家族

1.2 sklearn中的线性回归

2 多元线性回归LinearRegression

2.1 多元线性回归的基本原理

2.2 最小二乘法求解多元线性回归的参数

2.3 linear_model.LinearRegression

3 回归类的模型评估指标

3.1 是否预测了正确的数值

3.2 是否拟合了足够的信息

4 多重共线性：岭回归与Lasso

4.1 最熟悉的陌生人：多重共线性

4.2 岭回归

4.2.1 岭回归解决多重共线性问题

4.2.2 linear_model.Ridge

4.2.3 选取最佳的正则化参数取值

4.3 Lasso

4.3.1 Lasso与多重共线性

4.3.2 Lasso的核心作用：特征选择

4.3.3 选取最佳的正则化参数取值

5 非线性问题：多项式回归

5.1 重塑我们心中的“线性”概念

5.1.1 变量之间的线性关系

5.1.2 数据的线性与非线性

5.1.3 线性模型与非线性模型

5.2 使用分箱处理非线性问题

5.3 多项式回归PolynomialFeatures

5.3.1 多项式对数据做了什么

5.3.2 多项式回归处理非线性问题

5.3.3 多项式回归的可解释性

5.3.4 线性还是非线性模型？

6 结语

1 概述

1.1 线性回归大家族

回归是一种应用广泛的预测建模技术，这种技术的核心在于预测的结果是连续型变量。决策树，随机森林，支持向量机的分类器等分类算法的预测标签是分类变量，多以{0, 1}来表示，而无监督学习算法比如PCA, KMeans并不求解标签，注意加以区别。回归算法源于统计学理论，它可能是机器学习算法中产生最早的算法之一，其在现实中的应用非常广泛，包括使用其他经济指标预测股票市场指数，根据喷射流的特征预测区域内的降水量，根据公司的广告花费预测总销售额，或者根据有机物质中残留的碳-14的量来估计化石的年龄等等，只要一切基于特征预测连续型变量的需求，我们都使用回归技术。

既然线性回归是源于统计分析，是结合机器学习与统计学的重要算法。通常来说，我们认为统计学注重先验，而机器学习看重结果，因此机器学习中不会提前为线性回归排除共线性等可能会影响模型的因素，反而会先建立模型以查看效果。模型确立之后，如果效果不好，我们就根据统计学的指导来排除可能影响模型的因素。我们的课程会从机器学习的角度来为大家讲解回归类算法，如果希望理解统计学角度的小伙伴们，各种统计学教材都可以满足你的需求。

回归需求在现实中非常多，所以我们自然也有各种各样的回归类算法。最著名的就是我们的线性回归和逻辑回归，从他们衍生出了岭回归，Lasso，弹性网，除此之外，还有众多分类算法改进后的回归，比如回归树，随机森林的回归，支持向量回归，贝叶斯回归等等。除此之外，我们还有各种鲁棒的回归：比如RANSAC, Theil-Sen估计，胡贝尔回归等等。考虑到回归问题在现实中的泛用性，回归家族可以说是非常繁荣昌盛，家大业大了。

回归类算法的数学相对简单，相信在经历了逻辑回归，主成分分析与奇异值分解，支持向量机这三个章节之后，大家不会感觉到线性回归中的数学有多么困难。通常，理解线性回归可以有两种角度：**矩阵的角度和代数的角度**。几乎所有机器学习的教材都是从代数的角度来理解线性回归的，类似于我们在逻辑回归和支持向量机中做的那样，将求解参数的问题转化为一个带条件的最优化问题，然后使用三维图像让大家理解求极值的过程。如果大家掌握了逻辑回归和支持向量机，这个过程可以说是相对简单的，因此我们在本章节中就不进行赘述了。相对的，在我们的课程中一直都缺乏比较系统地使用矩阵来解读算法的角度，**因此在本堂课中，我将全程使用矩阵方式（线性代数的方式）为大家展现回归大家族的面貌。**

学完这堂课之后，大家需要对线性模型有个相对全面的了解，尤其是需要掌握线性模型究竟存在什么样的优点和问题，并且如何解决这些问题。

1.2 sklearn中的线性回归

sklearn中的线性模型模块是linear_model，我们曾经在学习逻辑回归的时候提到过这个模块。linear_model包含了多种多样的类和函数，其中逻辑回归相关的类和函数在这里就不给大家列举了。今天的课中我将会为大家来讲解：普通线性回归，多项式回归，岭回归，LASSO，以及弹性网。

类/函数	含义
普通线性回归	
linear_model.LinearRegression	使用普通最小二乘法的线性回归
岭回归	
linear_model.Ridge	岭回归，一种将L2作为正则化工具的线性最小二乘回归

类/函数	含义
linear_model.RidgeCV	带交叉验证的岭回归
linear_model.RidgeClassifier	岭回归的分类器
linear_model.RidgeClassifierCV	带交叉验证的岭回归的分类器
linear_model.ridge_regression	【函数】用正太方程法求解岭回归
LASSO	
linear_model.Lasso	Lasso, 使用L1作为正则化工具来训练的线性回归模型
linear_model.LassoCV	带交叉验证和正则化迭代路径的Lasso
linear_model.LassoLars	使用最小角度回归求解的Lasso
linear_model.LassoLarsCV	带交叉验证的使用最小角度回归求解的Lasso
linear_model.LassoLarsIC	使用BIC或AIC进行模型选择的, 使用最小角度回归求解的Lasso
linear_model.MultiTaskLasso	使用L1 / L2混合范数作为正则化工具训练的多标签Lasso
linear_model.MultiTaskLassoCV	使用L1 / L2混合范数作为正则化工具训练的, 带交叉验证的多标签Lasso
linear_model.lasso_path	【函数】用坐标下降计算Lasso路径
弹性网	
linear_model.ElasticNet	弹性网, 一种将L1和L2组合作为正则化工具的线性回归
linear_model.ElasticNetCV	带交叉验证和正则化迭代路径的弹性网
linear_model.MultiTaskElasticNet	多标签弹性网
linear_model.MultiTaskElasticNetCV	带交叉验证的多标签弹性网
linear_model.enet_path	【函数】用坐标下降法计算弹性网的路径
最小角度回归	
linear_model.Lars	最小角度回归 (Least Angle Regression, LAR)
linear_model.LarsCV	带交叉验证的最小角度回归模型
linear_model.lars_path	【函数】使用LARS算法计算最小角度回归路径或Lasso的路径
正交匹配追踪	
linear_model.OrthogonalMatchingPursuit	正交匹配追踪模型 (OMP)
linear_model.OrthogonalMatchingPursuitCV	交叉验证的正交匹配追踪模型 (OMP)
linear_model.orthogonal_mp	【函数】正交匹配追踪 (OMP)
linear_model.orthogonal_mp_gram	【函数】Gram正交匹配追踪 (OMP)
贝叶斯回归	
linear_model.ARDRidge	贝叶斯ARD回归。ARD是自动相关性确定回归 (Automatic Relevance Determination Regression) , 是一种类似于最小二乘的, 用来计算参数向量的数学方法。
linear_model.BayesianRidge	贝叶斯岭回归

类/函数	含义
其他回归	
linear_model.PassiveAggressiveClassifier	被动攻击性分类器
linear_model.PassiveAggressiveRegressor	被动攻击性回归
linear_model.Perceptron	感知机
linear_model.RANSACRegressor	RANSAC (RANdom SAmple Consensus) 算法。
linear_model.HuberRegressor	胡博回归，对异常值具有鲁棒性的一种线性回归模型
linear_model.SGDRegressor	通过最小化SGD的正则化损失函数来拟合线性模型
linear_model.TheilSenRegressor	Theil-Sen估计器，一种鲁棒的多元回归模型

2 多元线性回归LinearRegression

2.1 多元线性回归的基本原理

线性回归是机器学习中最简单的回归算法，多元线性回归指的就是一个样本有多个特征的线性回归问题。对于一个有 n 个特征的样本*i*而言，它的回归结果可以写成一个几乎人人熟悉的方程：

$$\hat{y}_i = w_0 + w_1 x_{i1} + w_2 x_{i2} + \dots + w_n x_{in}$$

w 被统称为模型的参数，其中 w_0 被称为截距(intercept)， $w_1 \sim w_n$ 被称为回归系数(regression coefficient)，有时也是使用 θ 或者 β 来表示。这个表达式，其实就和我们小学时就无比熟悉的 $y = ax + b$ 是同样的性质。其中 y 是我们的目标变量，也就是标签。 $x_{i1} \sim x_{in}$ 是样本*i*上的特征不同特征。如果考虑我们有m个样本，则回归结果可以被写成：

$$\hat{\mathbf{y}} = w_0 + w_1 \mathbf{x}_1 + w_2 \mathbf{x}_2 + \dots + w_n \mathbf{x}_n$$

其中 \mathbf{y} 是包含了m个全部的样本的回归结果的列向量。注意，我们通常使用粗体的小写字母来表示列向量，粗体的大写字母表示矩阵或者行列式。我们可以使用矩阵来表示这个方程，其中 w 可以被看做是一个结构为($n+1, 1$)的列矩阵， \mathbf{X} 是一个结构为($m, n+1$)的特征矩阵，则有：

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \\ \vdots \\ \hat{y}_m \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ 1 & x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ 1 & x_{31} & x_{32} & x_{33} & \dots & x_{3n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix} * \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$$

线性回归的任务，就是构造一个预测函数来映射输入的特征矩阵 \mathbf{X} 和标签值 y 的线性关系，这个预测函数在不同的教材上写法不同，可能写作 $f(x)$ ， $y_w(x)$ ，或者 $h(x)$ 等形式，但无论如何，**这个预测函数的本质就是我们需要构建的模型，而构造预测函数的核心就是找出模型的参数向量 w** 。但我们怎样才能够求解出参数向量呢？

记得在逻辑回归和SVM中，我们都是先定义了损失函数，然后通过最小化损失函数或损失函数的某种变化来将求解参数向量，以此将单纯的求解问题转化为一个最优化问题。在多元线性回归中，我们的损失函数如下定义：

$$\sum_{i=1}^m (y_i - \hat{y}_i)^2 = \sum_{i=1}^m (y_i - \mathbf{X}_i \mathbf{w})^2$$

其中 y_i 是样本*i*对应的真实标签， \hat{y}_i ，也就是 $\mathbf{X}_i \mathbf{w}$ 是样本*i*在一组参数 \mathbf{w} 下的预测标签。

首先，这个损失函数代表了向量 $y - \hat{y}$ 的L2范式的平方结果，L2范式的本质是就是欧式距离，即是两个向量上的每个点对应相减后的平方和再开平方，我们现在只实现了向量上每个点对应相减后的平方和，并没有开方，所以我们的损失函数是L2范式，即欧式距离的平方结果。

在这个平方结果下，我们的 y 和 \hat{y} 分别是我们的真实标签和预测值，也就是说，这个损失函数实在计算我们的真实标签和预测值之间的距离。因此，我们认为这个损失函数衡量了我们构造的模型的预测结果和真实标签的差异，因此我们固然希望我们的预测结果和真实值差异越小越好。所以我们的求解目标就可以转化成：

$$\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2$$

其中右下角的2表示向量 $\mathbf{y} - \mathbf{X}\mathbf{w}$ 的L2范式，也就是我们的损失函数所代表的含义。在L2范式上开平方，就是我们的损失函数。这个式子，也正是sklearn当中，用在类Linear_model.LinerRegression背后使用的损失函数。我们往往称呼这个式子为SSE (Sum of Squared Error, 误差平方和) 或者RSS (Residual Sum of Squares 残差平方和)。**在sklearn所有官方文档和网页上，我们都称之为RSS残差平方和**，因此在我们的课件中我们也这样称呼。

2.2 最小二乘法求解多元线性回归的参数

现在问题转换成了求解让RSS最小化的参数向量 \mathbf{w} ，**这种通过最小化真实值和预测值之间的RSS来求解参数的方法叫做最小二乘法**。求解极值的第一步往往是求解一阶导数并让一阶导数等于0，最小二乘法也不能免俗。因此，我们现在残差平方和RSS上对参数向量 \mathbf{w} 求导。这里的过程涉及到少数矩阵求导的内容，需要查表来确定，感兴趣可以去维基百科去查看矩阵求导的详细公式的表格：

https://en.wikipedia.org/wiki/Matrix_calculus

接下来，我们就来对 \mathbf{w} 求导：

$$\begin{aligned} \frac{\partial \text{RSS}}{\partial \mathbf{w}} &= \frac{\partial \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2}{\partial \mathbf{w}} \\ &= \frac{\partial (\mathbf{y} - \mathbf{X}\mathbf{w})^T (\mathbf{y} - \mathbf{X}\mathbf{w})}{\partial \mathbf{w}} \end{aligned}$$

$$\because (A - B)^T = A^T - B^T \text{ 并且 } (AB)^T = B^T * A^T$$

$$\begin{aligned} \therefore &= \frac{\partial (\mathbf{y}^T - \mathbf{w}^T \mathbf{X}^T)(\mathbf{y} - \mathbf{X}\mathbf{w})}{\partial \mathbf{w}} \\ &= \frac{\partial (\mathbf{y}^T \mathbf{y} - \mathbf{w}^T \mathbf{X}^T \mathbf{y} - \mathbf{y}^T \mathbf{X}\mathbf{w} + \mathbf{w}^T \mathbf{X}^T \mathbf{X}\mathbf{w})}{\partial \mathbf{w}} \end{aligned}$$

\because 矩阵求导中， a 为常数，有如下规则：

$$\frac{\partial a}{\partial A} = 0, \quad \frac{\partial A^T B^T C}{\partial A} = B^T C, \quad \frac{\partial C^T B A}{\partial A} = B^T C, \quad \frac{\partial A^T B A}{\partial A} = (B + B^T)A$$

$$\begin{aligned}
 &= 0 - \mathbf{X}^T \mathbf{y} - \mathbf{X}^T \mathbf{y} + 2\mathbf{X}^T \mathbf{X}\mathbf{w} \\
 &= \mathbf{X}^T \mathbf{X}\mathbf{w} - \mathbf{X}^T \mathbf{y}
 \end{aligned}$$

我们让求导后的一阶导数为0：

$$\begin{aligned}
 \mathbf{X}^T \mathbf{X}\mathbf{w} - \mathbf{X}^T \mathbf{y} &= 0 \\
 \mathbf{X}^T \mathbf{X}\mathbf{w} &= \mathbf{X}^T \mathbf{y} \\
 \text{左乘一个 } (\mathbf{X}^T \mathbf{X})^{-1} \text{ 则有:} \\
 \mathbf{w} &= (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}
 \end{aligned}$$

到了这里，我们希望能够将 \mathbf{w} 留在等式的左边，其他与特征矩阵有关的部分都放到等式的右边，如此就可以求出 \mathbf{w} 的最优解了。这个功能非常容易实现，只需要我们左乘 $\mathbf{X}^T \mathbf{X}$ 的逆矩阵就可以。**在这里，逆矩阵存在的充分必要条件是特征矩阵不存在多重共线性。**我们将会在第四节详细讲解多重共线性这个主题。

假设矩阵的逆是存在的，此时我们的 \mathbf{w} 就是我们参数的最优解。求解出这个参数向量，我们就解出了我们的 $\mathbf{X}\mathbf{w}$ ，也就能够计算出我们的预测值 $\hat{\mathbf{y}}$ 了。对于多元线性回归的理解，到这里就足够了，如果大家还希望继续深入，那我可以给大家一个方向：你们知道矩阵 $\mathbf{X}^T \mathbf{X}$ 其实是使用奇异值分解来进行求解的吗？可以仔细去研究一下。以算法工程师和数据挖掘工程师为目标的大家，能够手动推导上面的求解过程是基本要求。

除了多元线性回归的推导之外，这里还需要提到一些在上面的推导过程中不曾被体现出来的问题。在统计学中，使用最小二乘法来求解线性回归的方法是一种“无偏估计”的方法，这种无偏估计要求因变量，也就是标签的分布必须服从正态分布。这是说，我们的 \mathbf{y} 必须经由正太化处理（比如说取对数，或者使用在第三章《数据预处理与特征工程》中提到的类QuantileTransformer或者PowerTransformer）。在机器学习中，我们会先考虑模型的效果，如果模型效果不好，那我们可能考虑改变因变量的分布。

2.3 linear_model.LinearRegression

```
class sklearn.linear_model.LinearRegression(fit_intercept=True, normalize=False, copy_X=True, n_jobs=None)
```

参数	含义
<code>fit_intercept</code>	布尔值，可不填，默认为True 是否计算此模型的截距。如果设置为False，则不会计算截距
<code>normalize</code>	布尔值，可不填，默认为False 当 <code>fit_intercept</code> 设置为False时，将忽略此参数。如果为True，则特征矩阵 \mathbf{X} 在进入回归之前将被减去均值（中心化）并除以L2范式（缩放）。如果你希望进行标准化，请在 <code>fit</code> 数据之前使用 <code>preprocessing</code> 模块中的标准化专用类 <code>StandardScaler</code>
<code>copy_X</code>	布尔值，可不填，默认为True 如果为真，将在 <code>X.copy()</code> 上进行操作，否则的话原本的特征矩阵 \mathbf{X} 可能被线性回归影响并覆盖
<code>n_jobs</code>	整数或者None，可不填，默认为None 用于计算的作业数。只在多标签的回归和数据量足够大的时候才生效。除非None在 <code>joblib.parallel_backend</code> 上下文中，否则None统一表示为1。如果输入-1，则表示使用全部的CPU来进行计算。

线性回归的类可能是我们目前为止学到的最简单的类，仅有四个参数就可以完成一个完整的算法。并且看得出，这些参数中并没有一个是必填的，更没有对我们的模型有不可替代作用的参数。这说明，线性回归的性能，往往取决于数据本身，而并非是我们的调参能力，线性回归也因此对数据有着很高的要求。幸运的是，现实中大部分连续型变量之间，都存在着或多或少的线性联系。所以线性回归虽然简单，却很强大。

顺便一提，sklearn中的线性回归可以处理多标签问题，只需要在fit的时候输入多维度标签就可以了。

- 来做一次回归试试看吧

1. 导入需要的模块和库

```
from sklearn.linear_model import LinearRegression as LR
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.datasets import fetch_california_housing as fch #加利福尼亚房屋价值数据集
import pandas as pd
```

2. 导入数据，探索数据

```
housevalue = fch() #会需要下载，大家可以提前运行试试看

X = pd.DataFrame(housevalue.data) #放入DataFrame中便于查看

y = housevalue.target

X.shape

y.shape

X.head()

housevalue.feature_names

X.columns = housevalue.feature_names

"""

MedInc: 该街区住户的收入中位数
HouseAge: 该街区房屋使用年代的中位数
AveRooms: 该街区平均的房间数目
AveBedrms: 该街区平均的卧室数目
Population: 街区人口
AveOccup: 平均入住率
Latitude: 街区的纬度
Longitude: 街区的经度
"""


```

3. 分训练集和测试集

```
xtrain, Xtest, Ytrain, Ytest = train_test_split(X,y,test_size=0.3,random_state=420)

for i in [Xtrain, Xtest]:
    i.index = range(i.shape[0])

Xtrain.shape

#如果希望进行数据标准化，还记得应该怎么做吗？
#先用训练集训练标准化的类，然后用训练好的类分别转化训练集和测试集
```

4. 建模

```
reg = LR().fit(Xtrain, Ytrain)
yhat = reg.predict(Xtest)
yhat
```

5. 探索建好的模型

```
reg.coef_
[*zip(Xtrain.columns, reg.coef_)]
.....
MedInc: 该街区住户的收入中位数
HouseAge: 该街区房屋使用年代的中位数
AveRooms: 该街区平均的房间数目
AveBedrms: 该街区平均的卧室数目
Population: 街区人口
AveOccup: 平均入住率
Latitude: 街区的纬度
Longitude: 街区的经度
.....
reg.intercept_
```

属性	含义
coef_	数组，形状为 (n_features,) 或者 (n_targets, n_features) 线性回归方程中估计出的系数。如果在fit中传递多个标签（当y为二维或以上的时候），则返回的系数是形状为 (n_targets, n_features) 的二维数组，而如果仅传递一个标签，则返回的系数是长度为n_features的一维数组
intercept_	数组，线性回归中的截距项。

建模的过程在sklearn当中其实非常简单，但模型的效果如何呢？接下来我们来看看多元线性回归的模型评估指标。

3 回归类的模型评估指标

回归类算法的模型评估一直都是回归算法中的一个难点，但不像我们曾经讲过的无监督学习算法中的轮廓系数等等评估指标，回归类与分类型算法的模型评估其实是相似的法则——找真实标签和预测值的差异。只不过在分类型算法中，这个差异只有一种角度来评判，那就是是否预测到了正确的分类，而在我们的回归类算法中，我们有两种不同的角度来看待回归的效果：

第一，我们是否预测到了正确的数值。

第二，我们是否拟合到了足够的信息。

这两种角度，分别对应着不同的模型评估指标。

3.1 是否预测了正确的数值

回忆一下我们的RSS残差平方和，它的本质是我们的预测值与真实值之间的差异，也就是从第一种角度来评估我们回归的效力，所以RSS既是我们的损失函数，也是我们回归类模型的模型评估指标之一。但是，RSS有着致命的缺点：它是一个无界的和，可以无限地大。我们只知道，我们想要求解最小的RSS，从RSS的公式来看，它不能为负，所以RSS越接近0越好，但我们没有一个概念，究竟多小才算好，多接近0才算好？为了应对这种状况，sklearn中使用RSS的变体，均方误差MSE (mean squared error) 来衡量我们的预测值和真实值的差异：

$$MSE = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$$

均方误差，本质是在RSS的基础上除以了样本总量，得到了每个样本量上的平均误差。有了平均误差，我们就可以将平均误差和我们的标签的取值范围在一起比较，以此获得一个较为可靠的评估依据。在sklearn当中，我们有两种方式调用这个评估指标，一种是使用sklearn专用的模型评估模块metrics里的类mean_squared_error，另一种是调用交叉验证的类cross_val_score并使用里面的scoring参数来设置使用均方误差。

```
from sklearn.metrics import mean_squared_error as MSE
MSE(yhat, Ytest)

y.max()
y.min()

cross_val_score(reg, X, y, cv=10, scoring="mean_squared_error")

#为什么报错了？来试试看！
import sklearn
sorted(sklearn.metrics.SCORERS.keys())

cross_val_score(reg, X, y, cv=10, scoring="neg_mean_squared_error")
```

欢迎来的线性回归的大坑一号：均方误差为负。

我们在决策树和随机森林中都提到过，虽然均方误差永远为正，但是sklearn中的参数scoring下，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)。在sklearn当中，所有的损失都使用负数表示，因此均方误差也被显示为负数了。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。

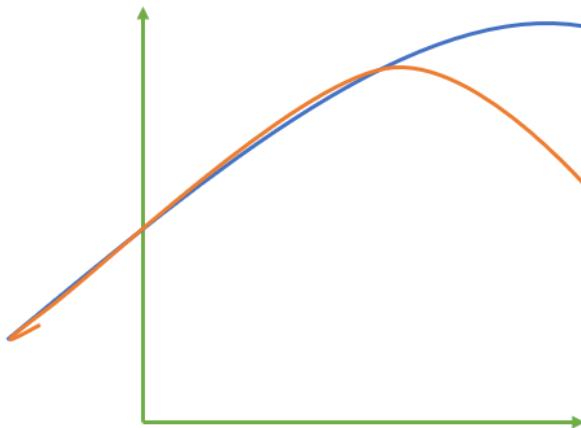
除了MSE，我们还有与MSE类似的MAE（Mean absolute error，绝对均值误差）：

$$MAE = \frac{1}{m} \sum_{i=0}^{m-1} |y_i - \hat{y}_i|$$

其表达的概念与均方误差完全一致，不过在真实标签和预测值之间的差异外我们使用的是L1范式（绝对值）。**现实使用中，MSE和MAE选一个来使用就好了。**在sklearn当中，我们使用命令`from sklearn.metrics import mean_absolute_error`来调用MAE，同时，我们也可以使用交叉验证中的`scoring = "neg_mean_absolute_error"`，以此在交叉验证时调用MAE。

3.2 是否拟合了足够的信息

对于回归类算法而言，只探索数据预测是否准确是不够的。除了数据本身的数值大小之外，我们还希望我们的模型能够捕捉到数据的“规律”，比如数据的分布规律，单调性等等，而是否捕获了这些信息并无法使用MSE来衡量。



来看这张图，其中红色线是我们的真实标签，而蓝色线是我们的拟合模型。这是一种比较极端，但的确可能发生的情况。这张图像上，前半部分的拟合非常成功，看上去我们的真实标签和我们的预测结果几乎重合，但后半部分的拟合却非常糟糕，模型向着与真实标签完全相反的方向去了。对于这样的一个拟合模型，如果我们使用MSE来对它进行判断，它的MSE会很小，因为大部分样本其实都被完美拟合了，少数样本的真实值和预测值的巨大差异在被均分到每个样本上之后，MSE就会很小。但这样的拟合结果必然不是一个好结果，因为一旦我的新样本是处于拟合曲线的后半段的，我的预测结果必然会有巨大的偏差，而这不是我们希望看到的。所以，我们希望找到新的指标，除了判断预测的数值是否正确之外，还能够判断我们的模型是否拟合了足够多的，数值之外的信息。

在我们学习降维算法PCA的时候，我们提到我们使用方差来衡量数据上的信息量。如果方差越大，代表数据上的信息量越多，而这个信息量不仅包括了数值的大小，还包括了我们希望模型捕捉的那些规律。为了衡量模型对数据上的信息量的捕捉，我们定义了 R^2 来帮助我们：

$$R^2 = 1 - \frac{\sum_{i=0}^m (y_i - \hat{y}_i)^2}{\sum_{i=0}^m (y_i - \bar{y})^2} = 1 - \frac{RSS}{\sum_{i=0}^m (y_i - \bar{y})^2}$$

其中 y 是我们的真实标签， \hat{y} 是我们的预测结果， \bar{y} 是我们的均值， $y_i - \bar{y}$ 如果除以样本量 m 就是我们的方差。方差的本质是任意一个 y 值和样本均值的差异，差异越大，这些值所带的信息越多。在 R^2 中，分子是真实值和预测值之差的差值，也就是我们的模型没有捕获到的信息总量，分母是真实标签所带的信息量，所以其衡量的是**1 - 我们的模型没有捕获到的信息量占真实标签中所带的信息量的比例**，所以， R^2 越接近1越好。

R^2 可以使用三种方式来调用，一种是直接从metrics中导入r2_score，输入预测值和真实值后打分。第二种是直接从线性回归LinearRegression的接口score来进行调用。第三种是在交叉验证中，输入"r2"来调用。

```
#调用R2
from sklearn.metrics import r2_score
r2_score(yhat,Ytest)

r2 = reg.score(Xtest,Ytest)
r2
```

我们现在踩到了线性回归的大坑二号：相同的评估指标不同的结果。

为什么结果会不一致呢？这就是回归和分类算法的不同带来的坑。

在我们的分类模型的评价指标当中，我们进行的是一种 if $a == b$ 的对比，这种判断和if $b == a$ 其实完全是一种概念，所以我们在进行模型评估的时候，从未踩到我们现在在的这个坑里。然而看 R^2 的计算公式， R^2 明显和分类模型的指标中的accuracy或者precision不一样， R^2 涉及到的计算中对预测值和真实值有极大的区别，必须是预测值在分子，真实值在分母，所以我们在调用metrcis模块中的模型评估指标的时候，必须要检查清楚，指标的参数中，究竟是要求我们先输入真实值还是先输入预测值。

```
#使用shift tab键来检查究竟哪个值先进行输入
r2_score(Ytest,yhat)

#或者你也可以指定参数，就不必在意顺序了
r2_score(y_true = Ytest,y_pred = yhat)

cross_val_score(reg,X,y,cv=10,scoring="r2").mean()
```

我们观察到，我们在加利福尼亚房屋价值数据集上的MSE其实不是一个很大的数（0.5），但我们的 R^2 不高，这证明我们的模型比较好地拟合了一部分数据的数值，却没有能正确拟合数据的分布。让我们与绘图来看看，究竟是不是这样一回事。我们可以绘制一张图上的两条曲线，一条曲线是我们的真实标签Ytest，另一条曲线是我们的预测结果yhat，两条曲线的交叠越多，我们的模型拟合就越好。

```
import matplotlib.pyplot as plt
sorted(Ytest)

plt.plot(range(len(Ytest)),sorted(Ytest),c="black",label= "Data")
plt.plot(range(len(yhat)),sorted(yhat),c="red",label = "Predict")
plt.legend()
plt.show()
```

可见，虽然我们的大部分数据被拟合得比较好，但是图像的开头和结尾处却又着较大的拟合误差。如果我们在图像右侧分布着更多的数据，我们的模型就会越来越偏离我们真正的标签。这种结果类似于我们前面提到的，虽然在有限的数据集上将数值预测正确了，但却没有正确拟合数据的分布，如果有更多的数据进入我们的模型，那数据标签被预测错误的可能性是非常大的。

思考

在sklearn中，一个与 R^2 非常相似的指标叫做可解释性方差分数 (explained_variance_score, EVS) ，它也是衡量 1 - 没有捕获到的信息占总信息的比例，但它和 R^2 略有不同。虽然在实践应用中EVS应用不多，但感兴趣的小伙伴可以探索一下这个回归类模型衡量指标。

现在，来看一组有趣的情况：

```
import numpy as np
rng = np.random.RandomState(42)
x = rng.randn(100, 80)
y = rng.randn(100)
cross_val_score(LR(), x, y, cv=5, scoring='r2')
```

好了，现在我们跋山涉水来到了线性回归的三号大坑：负的 R^2 。

许多学习过统计理论的小伙伴此时可能会感觉到，太坑了！sklearn真的是设置了太多障碍，均方误差是负的， R^2 也是负的，不能忍了。还有的小伙伴可能觉得，这个 R^2 名字里都带平方了，居然是负的，好气哦！无论如何，我们再来看看 R^2 的计算公式：

$$R^2 = 1 - \frac{\sum_{i=0}^m (y_i - \hat{y}_i)^2}{\sum_{i=0}^m (y_i - \bar{y})^2} = 1 - \frac{RSS}{TSS}$$

第一次学习机器学习或者统计学的小伙伴，可能会感觉没什么问题了， R^2 是1减一个数，后面的部分只要大于1的话 R^2 完全可以小于0。但是学过机器学习，尤其是在统计学上有基础的小伙伴可能会坐不住了：这不对啊！

一直以来，众多的机器学习教材中都有这样的解读：

除了RSS之外，我们还有解释平方和ESS (Explained Sum of Squares, 也叫做SSR回归平方和) 以及总离差平方和TSS (Total Sum of Squares, 也叫做SST总离差平方和)。解释平方和ESS定义了我们的预测值和样本均值之间的差异，而总离差平方和定义了真实值和样本均值之间的差异（就是 R^2 中的分母），两个指标分别写作：

$$\begin{aligned} TSS &= \sum_{i=0}^m (y_i - \bar{y})^2 \\ ESS &= \sum_{i=0}^m (\hat{y}_i - \bar{y})^2 \end{aligned}$$

而我们有公式：

$$TSS = RSS + ESS \tag{1}$$

看我们的 R^2 的公式，如果带入我们的TSS和ESS，那就有：

$$R^2 = 1 - \frac{RSS}{TSS} = \frac{TSS - RSS}{TSS} = \frac{ESS}{TSS}$$

而ESS和TSS都带平方，所以必然都是正数，那 R^2 怎么可能是负的呢？

好了，颠覆认知的时刻到了——**公式 $TSS = RSS + ESS$ 不是永远成立的！**就算所有的教材和许多博客里都理所当然这样写了大家也请抱着怀疑精神研究一下，你很快就会发现很多新世界。我们来看一看我们是如何证明(1)这个公式的：

$$\begin{aligned}
 TSS &= \sum_{i=0}^m (y_i - \bar{y})^2 \\
 &= \sum_{i=0}^m (y_i - \hat{y}_i + \hat{y}_i - \bar{y})^2 \\
 &= \sum_{i=0}^m (y_i - \hat{y}_i)^2 + \sum_{i=0}^m (\hat{y}_i - \bar{y})^2 + 2 \sum_{i=0}^m (y_i - \hat{y}_i)(\hat{y}_i - \bar{y}) \\
 &= RSS + ESS + 2 \sum_{i=0}^m (y_i - \hat{y}_i)(\hat{y}_i - \bar{y})
 \end{aligned}$$

两边同时除以 TSS , 则有:

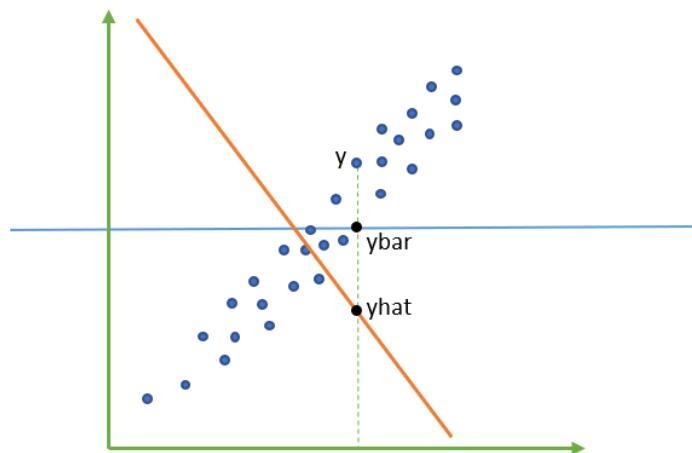
$$1 = \frac{RSS}{TSS} + \frac{ESS}{TSS} + \frac{2 \sum_{i=0}^m (y_i - \hat{y}_i)(\hat{y}_i - \bar{y})}{TSS}$$

$$1 - \frac{RSS}{TSS} = \frac{ESS + 2 \sum_{i=0}^m (y_i - \hat{y}_i)(\hat{y}_i - \bar{y})}{TSS}$$

$$R^2 = \frac{ESS + 2 \sum_{i=0}^m (y_i - \hat{y}_i)(\hat{y}_i - \bar{y})}{TSS}$$

许多教材和博客中让 $2 \sum_{i=0}^m (y_i - \hat{y}_i)(\hat{y}_i - \bar{y})$ 这个式子为0, 公式 $TSS = RSS + ESS$ 自然就成立了, 但要让这个式子成立是有条件的。现在有了这个式子的存在, R^2 就可以是一个负数了。只要我们的 $(y_i - \hat{y}_i)$ 衡量的是真实值到预测值的距离, 而 $(\hat{y}_i - \bar{y})$ 衡量的是预测值到均值的距离, 只要当这两个部分的符号不同的时候, 我们的式子 $2 \sum_{i=0}^m (y_i - \hat{y}_i)(\hat{y}_i - \bar{y})$ 就为负, 而 R^2 就有机会是一个负数。

看下面这张图, 蓝色的横线是我们的均值线 \bar{y} , 橙色的线是我们的模型 \hat{y} , 蓝色的点是我们的样本点。现在对于 x_i 来说, 我们的真实标签减预测值的值 $(y_i - \hat{y}_i)$ 为正, 但我们的预测值 $(\hat{y}_i - \bar{y})$ 却是一个负数, 这说明, 数据本身的均值, 比我们对数据的拟合模型本身更接近数据的真实值, 那我们的模型就是废的, 完全没有作用, 类似于分类模型中的分类准确率为50%, 不如瞎猜。



也就是说, 当我们的 R^2 显示为负的时候, 这证明我们的模型对我们的数据的拟合非常糟糕, 模型完全不能使用。所有, 一个负的 R^2 是合理的。当然了, 现实应用中, 如果你发现你的线性回归模型出现了负的 R^2 , 不代表你就要接受他了, 首先检查你的建模过程和数据处理过程是否正确, 也许你已经伤害了数据本身, 也许你的建模过程是存在bug的。如果是集成模型的回归, 检查你的弱评估器的数量是否不足, 随机森林, 提升树这些模型在只有两三棵树的时候

很容易出现负的 R^2 。如果你检查了所有的代码，也确定了你的预处理没有问题，但你的 R^2 也还是负的，那这就证明，线性回归模型不适合你的数据，试试看其他的算法吧。

4 多重共线性：岭回归与Lasso

4.1 最熟悉的陌生人：多重共线性

在第二节中我们曾推导了多元线性回归使用最小二乘法的求解原理，我们对多元线性回归的损失函数求导，并得出求解系数 w 的式子和过程：

$$\begin{aligned} \mathbf{X}^T \mathbf{X} \mathbf{w} - \mathbf{X}^T \mathbf{y} &= 0 \\ \mathbf{X}^T \mathbf{X} \mathbf{w} &= \mathbf{X}^T \mathbf{y} \\ \text{左乘一个 } (\mathbf{X}^T \mathbf{X})^{-1} \text{ 则有:} \\ \mathbf{w} &= (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \end{aligned}$$

在最后一步中我们需要左乘 $\mathbf{X}^T \mathbf{X}$ 的逆矩阵，而逆矩阵存在的充分必要条件是特征矩阵不存在多重共线性。多重共线性这个词对于许多人来说都不陌生，然而却很少有人能够透彻理解这个性质究竟是什么含义，会有怎样的影响。这一节我们会来深入讲解，什么是多重共线性，我们是如何一步步从逆矩阵必须存在推导到多重共线性不能存在的。本节需要大量的线性代数知识作为支撑，讲解会比较细致，如果依然感觉对其中的数学原理理解困难，建议以专门讲线性代数的数学教材为辅学习本节。

- 逆矩阵存在的充分必要条件

首先，我们需要先理解逆矩阵存在与否的意义和影响。一个矩阵什么情况下才可以有逆矩阵呢？来看逆矩阵的计算公式：

$$\mathbf{A}^{-1} = \frac{1}{|\mathbf{A}|} \mathbf{A}^*$$

分子上 \mathbf{A}^* 是伴随矩阵，任何矩阵都可以有伴随矩阵，因此这一部分不影响逆矩阵的存在性。而分母上的行列式 $|\mathbf{A}|$ 就不同了，位于分母的变量不能为0，一旦为0则无法计算出逆矩阵。因此逆矩阵存在的充分必要条件是：矩阵的行列式不能为0，对于线性回归而言，即是说 $|\mathbf{X}^T \mathbf{X}|$ 不能为0。这是使用最小二乘法来求解线性回归的核心条件之一。

- 行列式不为0的充分必要条件

那行列式要不为0，需要满足什么条件呢？在这里，我们来复习一下线性代数中的基本知识。假设我们的特征矩阵 \mathbf{X} 结构为 (m, n) ，则 $\mathbf{X}^T \mathbf{X}$ 就是结构为 (n, m) 的矩阵乘以结构为 (m, n) 的矩阵，从而得到结果为 (n, n) 的方阵。

$$\mathbf{X}^T \mathbf{X} = (n, m) * (m, n) = (n, n)$$

因此以下所有的例子都将以方阵进行举例，方便大家理解。首先区别一下矩阵和行列式：

$$\text{矩阵 } A = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

$$\text{矩阵 } A \text{ 的行列式} = \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} = |A|$$

重要定义：矩阵和行列式

矩阵是一组数按一定方式排列成的数表，一般记作 A

行列式是这一组数按某种运算法则最后计算出来的一个数，通常记作 $|A|$ 或者 $\det A$

任何矩阵都可以有行列式。以一个 3×3 的行列式为例，我们来看看行列式是如何计算的：

$$|A| = \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix}$$

$$= x_{11}x_{22}x_{33} + x_{12}x_{23}x_{31} + x_{13}x_{21}x_{32} - x_{11}x_{23}x_{32} - x_{12}x_{21}x_{33} - x_{13}x_{22}x_{31}$$

这个式子乍一看非常混乱，其实并非如此，我们把行列式 $|A|$ 按照下面的方式排列一下，就很容易看出这个式子实际上是怎么回事了：

$$\begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} \quad \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} \quad \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix}$$

$$\begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} \quad \begin{array}{c} x_{11} \\ x_{21} \\ x_{31} \end{array} \quad \begin{array}{c} x_{12} \\ x_{22} \\ x_{32} \end{array} \quad \begin{array}{c} x_{13} \\ x_{23} \\ x_{33} \end{array} \quad \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} \quad \begin{array}{c} x_{11} \\ x_{21} \\ x_{31} \end{array} \quad \begin{array}{c} x_{12} \\ x_{22} \\ x_{32} \end{array} \quad \begin{array}{c} x_{13} \\ x_{23} \\ x_{33} \end{array}$$

三个特征的特征矩阵的行列式就有六个交互项，在现实中我们的特征矩阵不可能是如此低维度的数据，因此使用这样的方式计算行列式就变得异常困难。在线性代数中，我们可以通过行列式的计算将一个行列式整合成一个梯形的行列式：

$$|A| = \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} \rightarrow \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{vmatrix}$$

梯形的行列式表现为，所有的数字都被整合到对角线的上方或下方（通常是上方），虽然具体的数字发生了变化（比如由 x_{11} 变成了 a_{11} ），但是行列式的大小在初等行变换/列变换的过程中是不变的。对于梯形行列式，行列式的计算要容易很多：

$$|A| = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{vmatrix}$$

$$\begin{aligned} &= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31} \\ &= a_{11}a_{22}a_{33} + a_{12}a_{23} * 0 + a_{13} * 0 * 0 - a_{11}a_{23} * 0 - a_{12} * 0 * a_{33} - a_{13}a_{22} * 0 \\ &= a_{11}a_{22}a_{33} \end{aligned}$$

不难发现，由于梯形行列式下半部分为0，整个矩阵的行列式其实就是梯形行列式对角线上的元素相乘。并且此时此刻，只要对角线上的任意元素为0，整个行列式都会为0。那只要对角线上没有一个元素为0，行列式就不会为0了。在这里，我们来引入一个重要的概念：满秩矩阵。

重要定义：满秩矩阵

满秩矩阵：A是一个n行n列的矩阵，若A转换为梯形矩阵后，没有任何全为0的行或者全为0的列，则称A为满秩矩阵。简单来说，只要对角线上没有一个元素为0，则这个矩阵中绝对不可能存在全为0的行或列。

举例来说，下面的矩阵就不是满秩矩阵，因为它的对角线上有一个0，因此它存在全为0的行。

不是 满 秩 矩 阵 :

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & 0 \end{bmatrix}$$

即是说，矩阵满秩（即转换为梯形矩阵后对角线上没有0）是矩阵的行列式不为0的充分必要条件。

• 矩阵满秩的充分必要条件

一个矩阵要满秩，则转换为梯形矩阵后的对角线上没有0，那什么样的矩阵在转换为梯形矩阵后对角线上才没有0呢？来看下面的三个矩阵：

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 4.002 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 17 \end{bmatrix}$$

我们可以对矩阵做初等行变换和列变换，包括交换行/列顺序，将一列/一行乘以一个常数后加减到另一列/一行上，来将矩阵化为梯形矩阵。对于上面的两个矩阵我们可以有如下变换：

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 4.002 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 17 \end{bmatrix} \quad \text{--- 第一行 * 2}$$

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 0.002 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 13 \end{bmatrix}$$

继续进行变换：

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 0.002 \end{bmatrix}, C = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 13 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 0 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 1 & 1 & 2 \\ 0 & -2 & 1 \\ 0 & 0 & 0.002 \end{bmatrix}, C = \begin{bmatrix} 1 & 1 & 2 \\ 0 & -2 & 1 \\ 0 & 0 & 13 \end{bmatrix}$$

如此就转换成了梯形矩阵。我们可以看到，矩阵A明显不是满秩的，它有全零行所以行列式会为0。而矩阵B和C没有全零行所以满秩。而矩阵A和矩阵B的区别在于，A中存在着完全具有线性关系的两行（1, 1, 2和2, 2, 4），而B和C中则没有这样的两行。而矩阵B虽然对角线上每个元素都不为0，但具有非常接近于0的元素0.02，而矩阵C的对角线上没有任何元素特别接近于0。

矩阵A中第一行和第三行的关系，被称为“**精确相关关系**”，即完全相关，一行可使另一行为0。在这种精确相关关系下，矩阵A的行列式为0，则矩阵A的逆不可能存在。在我们的最小二乘法中，如果矩阵 $\mathbf{X}^T \mathbf{X}$ 中存在这种精确相关关系，则逆不存在，最小二乘法完全无法使用，**线性回归会无法求出结果**。

$$(\mathbf{X}^T \mathbf{X})^{-1} = \frac{1}{|\mathbf{X}^T \mathbf{X}|} (\mathbf{X}^T \mathbf{X})^* \rightarrow \frac{1}{0} (\mathbf{X}^T \mathbf{X})^* \rightarrow \text{除零错误}$$

矩阵B中第一行和第三行的关系不太一样，他们之间非常接近于“精确相关关系”，但又不是完全相关，一行不能使另一行为0，这种关系被称为“**高度相关关系**”。在这种高度相关关系下，矩阵的行列式不为0，但是一个非常接近0数，矩阵A的逆存在，不过接近于无限大。在这种情况下，最小二乘法可以使用，不过得到的逆会很大，直接影响我们对参数向量 w 的求解：

$$(\mathbf{X}^T \mathbf{X})^{-1} = \frac{1}{|\mathbf{X}^T \mathbf{X}|} (\mathbf{X}^T \mathbf{X})^* \rightarrow \frac{1}{\text{非常接近 } 0 \text{ 的数}} (\mathbf{X}^T \mathbf{X})^* \rightarrow \infty$$

$$\mathbf{w} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \rightarrow \infty$$

这样求解出来的参数向量 w 会很大，因此会影响建模的结果，造成模型有偏差或者不可用。**精确相关关系和高度相关关系并称为“多重共线性”**。在多重共线性下，模型无法建立，或者模型不可用。

相对的，矩阵C的行之间结果相互独立，梯形矩阵看起来非常正常，它的对角线上没有任何元素特别接近于0，因此其行列式也就不会接近0或者为0，因此矩阵C得出的参数向量 w 就不会有太大偏差，对于我们拟合而言是比较理想的。

$$(\mathbf{X}^T \mathbf{X})^{-1} = \frac{1}{|\mathbf{X}^T \mathbf{X}|} (\mathbf{X}^T \mathbf{X})^* \rightarrow \frac{1}{\text{不是非常接近于 } 0 \text{ 的常数}} (\mathbf{X}^T \mathbf{X})^* \rightarrow \text{逆矩阵的大小正常}$$

$$\mathbf{w} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \rightarrow \text{拟合出适合的 } w$$

从上面的所有过程我们可以看得出来，**一个矩阵如果要满秩，则要求矩阵中每个向量之间不能存在多重共线性**，这也构成了线性回归算法对于特征矩阵的要求。



• 多重共线性与相关性

多重共线性如果存在，则线性回归就无法使用最小二乘法来进行求解，或者求解就会出现偏差。幸运的是，不能存在多重共线性，不代表不能存在相关性——机器学习不要求特征之间必须独立，必须不相关，只要不是高度相关或者精确相关就好。

多重共线性 Multicollinearity 与 相关性 Correlation

多重共线性是一种统计现象，是指线性模型中的特征（解释变量）之间由于存在精确相关关系或高度相关关系，多重共线性的存在会使模型无法建立，或者估计失真。多重共线性使用指标方差膨胀因子（variance inflation factor, VIF）来进行衡量（from statsmodels.stats.outliers_influence import variance_inflation_factor），通常当我们提到“共线性”，都特指多重共线性。

相关性是衡量两个或多个变量一起波动的程度的指标，它可以是正的，负的或者0。当我们说变量之间具有相关性，通常是指线性相关性，线性相关一般由皮尔逊相关系数进行衡量，非线性相关可以使用斯皮尔曼相关系数或者互信息法进行衡量。

在现实中特征之间完全独立的情况其实非常少，因为大部分数据统计手段或者收集者并不考虑统计学或者机器学习建模时的需求，现实数据多多少少都会存在一些相关性，极端情况下，甚至还可能出现收集的特征数量比样本数量多的情况。通常来说，这些相关性在机器学习中通常无伤大雅（在统计学中他们可能是比较严重的问题），即便有一些偏差，只要最小二乘法能够求解，我们都可能会无视掉它。毕竟，想要消除特征的相关性，无论使用怎样的手段，都无法避免进行特征选择，这意味着可用的信息变得更加少，对于机器学习来说，很有可能尽量排除相关性后，模型的整体效果会受到巨大的打击。这种情况下，我们选择不处理相关性，只要结果好，一切万事大吉。

然而多重共线性就不是这样一回事了，它的存在会造成模型极大地偏移，无法模拟数据的全貌，因此这是必须解决的问题。为了保留线性模型计算快速，理解容易的优点，我们并不希望更换成非线性模型，这促使统计学家和机器学习研究者们钻研出了多种能够处理多重共线性的方法，其中有三种比较常见的：

使用统计学的先验思路	使用向前逐步回归	改进线性回归
在开始建模之前先对数据进行各种相关性检验，如果存在多重共线性则可考虑对数据的特征进行删减筛查，或者使用降维算法对其进行处理，最终获得一个完全不存在相关性的数据集	逐步归回能够筛选对标签解释力度最强的特征，同时对于存在相关性的特征们加上一个惩罚项，削弱其对标签的贡献，以绕过最小二乘法对共线性较为敏感的缺陷	在原有的线性回归算法基础上进行修改，使其能够容忍特征列存在多重共线性的情况，并且能够顺利建模，且尽可能的保证RSS取得最小值

这三种手段中，第一种相对耗时耗力，需要较多的人工操作，并且会需要混合各种统计学中的知识和检验来进行使用。在机器学习中，能够使用一种模型解决的问题，我们尽量不用多个模型来解决，如果能够追求结果，我们会尽量避免进行一系列检验。况且，统计学中的检验往往以“让特征独立”为目标，与机器学习中的“稍微有点相关性也无妨”不太一致。

第二种手段在现实中应用较多，不过由于理论复杂，效果也不是非常高效，因此向前逐步回归不是机器学习的首选。在本周的课程中，我们的核心会是使用第三种方法：改进线性回归来处理多重共线性。为此，一系列算法，岭回归，Lasso，弹性网就被研究出来了。接下来，我们就来看看这些改善多重共线性问题的算法。

4.2 岭回归

4.2.1 岭回归解决多重共线性问题

在线性模型之中，除了线性回归之外，最知名的就是岭回归与Lasso了。这两个算法非常神秘，他们的原理和应用都不像其他算法那样高调，学习资料也很少。这可能是因为这两个算法不是为了提升模型表现，而是为了修复漏洞而设计的（实际上，我们使用岭回归或者Lasso，模型的效果往往会下降一些，因为我们删除了一小部分信息），因此在结果为上的机器学习领域颇有些被冷落的意味。这一节我们就来了解一下岭回归。

岭回归，又称为吉洪诺夫正则化 (Tikhonov regularization)。通常来说，大部分的机器学习教材会使用代数的形式来展现岭回归的原理，这个原理和逻辑回归及支持向量机非常相似，都是将求解 w 的过程转化为一个带条件的最优化问题，然后用最小二乘法求解。然而，岭回归可以做到的事其实可以用矩阵非常简单地表达出来。

岭回归在多元线性回归的损失函数上加上了正则项，表达为系数 w 的L2范式（即系数 w 的平方项）乘以正则化系数 α 。如果你们看其他教材中的代数推导，正则化系数会写作 λ ，用以和Lasso区别，不过在sklearn中由于是两个不同的算法，因此正则项系数都使用 α 来代表。岭回归的损失函数的完整表达式写作：

$$\min_w \|\mathbf{X}w - \mathbf{y}\|_2^2 + \alpha\|w\|_2^2$$

这个操作看起来简单，其实带来了巨大的变化。在线性回归中我们通过在损失函数上对 w 求导来求解极值，在这里虽然加上了正则项，我们依然使用最小二乘法来求解。假设我们的特征矩阵结构为 (m, n) ，系数 w 的结构是 $(1, n)$ ，则可以有：

$$\begin{aligned} \frac{\partial(RSS + \alpha\|w\|_2^2)}{\partial w} &= \frac{\partial(\|\mathbf{y} - \mathbf{X}w\|_2^2 + \alpha\|w\|_2^2)}{\partial w} \\ &= \frac{\partial(\mathbf{y} - \mathbf{X}w)^T(\mathbf{y} - \mathbf{X}w)}{\partial w} + \frac{\partial\alpha\|w\|_2^2}{\partial w} \end{aligned}$$

前半部分我们推导过，后半部分对 w 求导非常简单：

$$= 0 - 2\mathbf{X}^T \mathbf{y} + 2\mathbf{X}^T \mathbf{X}w + 2\alpha w$$

将含有 w 的项合并，其中 α 为常数

为了实现矩阵相加，让它乘以一个结构为 $n * n$ 的单位矩阵 I ：

$$\begin{aligned} &= (\mathbf{X}^T \mathbf{X} + \alpha I)w - \mathbf{X}^T \mathbf{y} \\ (\mathbf{X}^T \mathbf{X} + \alpha I)w &= \mathbf{X}^T \mathbf{y} \end{aligned}$$

现在，只要 $(\mathbf{X}^T \mathbf{X} + \alpha I)$ 存在逆矩阵，我们就可以求解出 w 。一个矩阵存在逆矩阵的充分必要条件是这个矩阵的行列式不为0。假设原本的特征矩阵中存在共线性，则我们的方阵 $\mathbf{X}^T \mathbf{X}$ 就会不满秩（存在全为零的行）：

$$\mathbf{X}^T \mathbf{X} = \begin{vmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{3n} \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & 0 \end{vmatrix}$$

此时方阵 $\mathbf{X}^T \mathbf{X}$ 就是没有逆的，最小二乘法就无法使用。然而，加上了 $\alpha \mathbf{I}$ 之后，我们的矩阵就大不一样了：

$$\alpha \mathbf{I} = \alpha * \begin{vmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & 1 \end{vmatrix} = \begin{vmatrix} \alpha & 0 & 0 & \dots & 0 \\ 0 & \alpha & 0 & \dots & 0 \\ 0 & 0 & \alpha & \dots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & \alpha \end{vmatrix}$$

$$\begin{aligned} \mathbf{X}^T \mathbf{X} + \alpha \mathbf{I} &= \begin{vmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{3n} \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & 0 \end{vmatrix} + \begin{vmatrix} \alpha & 0 & 0 & \dots & 0 \\ 0 & \alpha & 0 & \dots & 0 \\ 0 & 0 & \alpha & \dots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & \alpha \end{vmatrix} \\ &= \begin{vmatrix} a_{11} + \alpha & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} + \alpha & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} + \alpha & \dots & a_{3n} \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & \alpha \end{vmatrix} \end{aligned}$$

最后得到的这个行列式还是一个梯形行列式，然而它的已经不存在全0行或者全0列了，除非：

(1) α 等于0，或者

(2) 原本的矩阵 $\mathbf{X}^T \mathbf{X}$ 中存在对角线上元素为 $-\alpha$ ，其他元素都为0的行或者列

否则矩阵 $\mathbf{X}^T \mathbf{X} + \alpha \mathbf{I}$ 永远都是满秩。在sklearn中， α 的值我们可以自由控制，因此我们可以让它不为0，以避免第一种情况。而第二种情况，如果我们发现某个 α 的取值下模型无法求解，那我们只需要换一个 α 的取值就好了，也可以顺利避免。也就是说，矩阵的逆是永远存在的！有利这个保障，我们的 \mathbf{w} 就可以写作：

$$\begin{aligned} (\mathbf{X}^T \mathbf{X} + \alpha \mathbf{I}) \mathbf{w} - \mathbf{X}^T \mathbf{y} &= 0 \\ (\mathbf{X}^T \mathbf{X} + \alpha \mathbf{I}) \mathbf{w} &= \mathbf{X}^T \mathbf{y} \end{aligned}$$

左乘一个 $(\mathbf{X}^T \mathbf{X} + \alpha \mathbf{I})^{-1}$ 则有：

$$\mathbf{w} = (\mathbf{X}^T \mathbf{X} + \alpha \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y}$$

如此，正则化系数 α 就非常爽快地避免了“精确相关关系”带来的影响，至少最小二乘法在 α 存在的情况下是一定可以使用了。对于存在“高度相关关系”的矩阵，我们也可以通过调大 α ，来让 $\mathbf{X}^T \mathbf{X} + \alpha \mathbf{I}$ 矩阵的行列式变大，从而让逆矩阵变小，以此控制参数向量 w 的偏移。当 α 越大，模型越不容易受到共线性的影响。

$$(\mathbf{X}^T \mathbf{X} + \alpha \mathbf{I})^{-1} = \frac{1}{|\mathbf{X}^T \mathbf{X} + \alpha \mathbf{I}|} (\mathbf{X}^T \mathbf{X} + \alpha \mathbf{I})^*$$

如此，多重共线性就被控制住了：最小二乘法一定有解，并且这个解可以通过 α 来进行调节，以确保不会偏离太多。当然了， α 挤占了 w 中由原始的特征矩阵贡献的空间，因此 α 如果太大，也会导致 w 的估计出现较大的偏移，无法正确拟合数据的真实面貌。我们在使用中，需要找出 α 让模型效果变好的最佳取值。

4.2.2 linear_model.Ridge

在sklearn中，岭回归由线性模型库中的Ridge类来调用：

```
class sklearn.linear_model.Ridge(alpha=1.0, fit_intercept=True, normalize=False, copy_X=True, max_iter=None, tol=0.001, solver='auto', random_state=None)
```

和线性回归相比，岭回归的参数稍微多了那么一点点，但是真正核心的参数就是我们的**正则项的系数 α** ，其他的参数是当我们希望使用最小二乘法之外的求解方法求解岭回归的时候才需要的，通常我们完全不会去触碰这些参数。所以大家只需要了解 α 的用法就可以了。

之前我们在加利佛尼亚房屋价值数据集上使用线性回归，得出的结果大概是训练集上的拟合程度是60%，测试集上的拟合程度也是60%左右，那这个很低的拟合程度是不是由多重共线性造成的呢？在统计学中，我们会通过VIF或者各种检验来判断数据是否存在共线性，然而在机器学习中，我们可以使用模型来判断——如果一个数据集在岭回归中使用各种正则化参数取值下模型表现没有明显上升（比如出现持平或者下降），则说明数据没有多重共线性，顶多是特征之间有一些相关性。反之，如果一个数据集在岭回归的各种正则化参数取值下表现出明显的上升趋势，则说明数据存在多重共线性。

接下来，我们就在加利佛尼亚房屋价值数据集上来验证一下这个说法：

```
import numpy as np
import pandas as pd
from sklearn.linear_model import Ridge, LinearRegression, Lasso
from sklearn.model_selection import train_test_split as TTS
from sklearn.datasets import fetch_california_housing as fch
import matplotlib.pyplot as plt

housevalue = fch()

X = pd.DataFrame(housevalue.data)
y = housevalue.target
X.columns = ["住户收入中位数", "房屋使用年代中位数", "平均房间数目",
             "平均卧室数目", "街区人口", "平均入住率", "街区的纬度", "街区的经度"]

X.head()

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

#数据集索引恢复
for i in [Xtrain,Xtest]:
    i.index = range(i.shape[0])

#使用岭回归来进行建模
reg = Ridge(alpha=1).fit(Xtrain,Ytrain)
```

```

reg.score(Xtest, Ytest)

#交叉验证下，与线性回归相比，岭回归的结果如何变化？
alpharange = np.arange(1, 1001, 100)
ridge, lr = [], []
for alpha in alpharange:
    reg = Ridge(alpha=alpha)
    linear = LinearRegression()
    regs = cross_val_score(reg, X, y, cv=5, scoring = "r2").mean()
    linears = cross_val_score(linear, X, y, cv=5, scoring = "r2").mean()
    ridge.append(regs)
    lr.append(linears)
plt.plot(alpharange, ridge, color="red", label="Ridge")
plt.plot(alpharange, lr, color="orange", label="LR")
plt.title("Mean")
plt.legend()
plt.show()

#细化一下学习曲线
alpharange = np.arange(1, 201, 10)

```

可以看出，加利佛尼亚数据集上，岭回归的结果轻微上升，随后骤降。可以说，加利佛尼亚房屋价值数据集带有很轻微的一部分共线性，这种共线性被正则化参数 α 消除后，模型的效果提升了一点点，但是对于整个模型而言是杯水车薪。在过了控制多重共线性的点后，模型的效果飞速下降，显然是正则化的程度太重，挤占了参数 w 本来的估计空间。从这个结果可以看出，加利佛尼亚数据集的核心问题不在于多重共线性，岭回归不能够提升模型表现。

另外，在正则化参数逐渐增大的过程中，我们可以观察一下模型的方差如何变化：

```

#模型方差如何变化？
alpharange = np.arange(1, 1001, 100)
ridge, lr = [], []
for alpha in alpharange:
    reg = Ridge(alpha=alpha)
    linear = LinearRegression()
    varR = cross_val_score(reg, X, y, cv=5, scoring="r2").var()
    varLR = cross_val_score(linear, X, y, cv=5, scoring="r2").var()
    ridge.append(varR)
    lr.append(varLR)
plt.plot(alpharange, ridge, color="red", label="Ridge")
plt.plot(alpharange, lr, color="orange", label="LR")
plt.title("variance")
plt.legend()
plt.show()

```

可以发现，模型的方差上升快速，不过方差的值本身很小，其变化不超过 R^2 上升部分的1/3，因此只要噪声的状况维持恒定，模型的泛化误差可能还是一定程度上降低了的。虽然岭回归和Lasso不是设计来提升模型表现，而是专注于解决多重共线性问题的，但当 α 在一定范围内变动的时候，消除多重共线性也许能够一定程度上提高模型的泛化能力。

但是泛化能力毕竟没有直接衡量的指标，因此我们往往只能通过观察模型的准确性指标和方差来大致评判模型的泛化能力是否提高。来看看多重共线性更为明显一些的情况：

```

from sklearn.datasets import load_boston
from sklearn.model_selection import cross_val_score

X = load_boston().data
y = load_boston().target

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

#先查看方差的变化
alpharange = np.arange(1,1001,100)
ridge, lr = [], []
for alpha in alpharange:
    reg = Ridge(alpha=alpha)
    linear = LinearRegression()
    varR = cross_val_score(reg,X,y,cv=5,scoring="r2").var()
    varLR = cross_val_score(linear,X,y,cv=5,scoring="r2").var()
    ridge.append(varR)
    lr.append(varLR)
plt.plot(alpharange,ridge,color="red",label="Ridge")
plt.plot(alpharange,lr,color="orange",label="LR")
plt.title("variance")
plt.legend()
plt.show()

#查看R2的变化
alpharange = np.arange(1,1001,100)
ridge, lr = [], []
for alpha in alpharange:
    reg = Ridge(alpha=alpha)
    linear = LinearRegression()
    regs = cross_val_score(reg,X,y,cv=5,scoring = "r2").mean()
    linears = cross_val_score(linear,X,y,cv=5,scoring = "r2").mean()
    ridge.append(regs)
    lr.append(linears)
plt.plot(alpharange,ridge,color="red",label="Ridge")
plt.plot(alpharange,lr,color="orange",label="LR")
plt.title("Mean")
plt.legend()
plt.show()

#细化学习曲线
alpharange = np.arange(100,300,10)
ridge, lr = [], []
for alpha in alpharange:
    reg = Ridge(alpha=alpha)
    #linear = LinearRegression()
    regs = cross_val_score(reg,X,y,cv=5,scoring = "r2").mean()
    #linears = cross_val_score(linear,X,y,cv=5,scoring = "r2").mean()
    ridge.append(regs)
    lr.append(linears)
plt.plot(alpharange,ridge,color="red",label="Ridge")
#plt.plot(alpharange,lr,color="orange",label="LR")
plt.title("Mean")

```

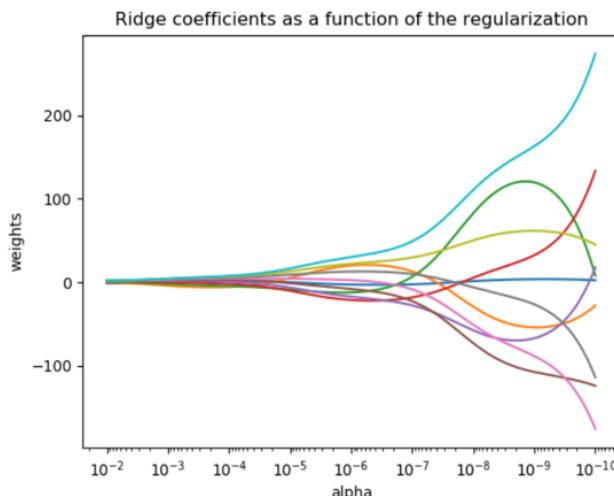
```
plt.legend()
plt.show()
```

可以发现，比起加利佛尼亚房屋价值数据集，波士顿房价数据集的方差降低明显，偏差也降低明显，可见使用岭回归还是起到了一定的作用，模型的泛化能力是有可能会上升的。

遗憾的是，没有人会希望自己获取的数据集中存在多重共线性，因此发布到scikit-learn或者kaggle上的数据基本都经过一定的多重共线性的处理的，要找出绝对具有多重共线性的数据非常困难，也就无法给大家展示岭回归在实际数据中大显身手的模样。我们也许可以找出具有一些相关性的数据，但是大家如果去尝试就会发现，基本上如果我们使用岭回归或者Lasso，那模型的效果都是会降低的，很难升高，这恐怕也是岭回归和Lasso一定程度上被机器学习领域冷遇的原因。

4.2.3 选取最佳的正则化参数取值

既然要选择 α 的范围，我们就不可避免地要进行最优参数的选择。在各种机器学习教材中，总是教导使用岭迹图来判断正则项参数的最佳取值。传统的岭迹图长这样，形似一个开口的喇叭图（根据横坐标的正负，喇叭有可能朝右或者朝左）：



这一个以正则化参数为横坐标，线性模型求解的系数 w 为纵坐标的图像，其中每一条彩色的线都是一个系数 w 。其目标是建立正则化参数与系数 w 之间的直接关系，以此来观察正则化参数的变化如何影响了系数 w 的拟合。岭迹图认为，线条交叉越多，则说明特征之间的多重共线性越高。我们应该选择系数较为平稳的喇叭口所对应的 α 取值作为最佳的正则化参数的取值。绘制岭迹图的方法非常简单，代码如下：

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model

#创造10*10的希尔伯特矩阵
X = 1. / (np.arange(1, 11) + np.arange(0, 10)[:, np.newaxis])
y = np.ones(10)

#计算横坐标
n_alphas = 200
alphas = np.logspace(-10, -2, n_alphas)

#建模，获取每一个正则化取值下的系数组合
model = linear_model.Ridge(fit_intercept=False)
```

```

coefs = []
for a in alphas:
    ridge = linear_model.Ridge(alpha=a, fit_intercept=False)
    ridge.fit(x, y)
    coefs.append(ridge.coef_)

#绘图展示结果
ax = plt.gca()
ax.plot(alphas, coefs)
ax.set_xscale('log')
ax.set_xlim(ax.get_xlim()[::-1]) #将横坐标逆转
plt.xlabel('正则化参数alpha')
plt.ylabel('系数w')
plt.title('岭回归下的岭迹图')
plt.axis('tight')
plt.show()

```

其中涉及到的希尔伯特矩阵长这样：

$$H = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}.$$

然而，我非常不建议大家使用岭迹图来作为寻找最佳参数的标准。

有这样的两个理由：

1. 岭迹图的很多细节，很难以解释。比如为什么多重共线性存在会使得线与线之间有很多交点？当 α 很大了之后看上去所有的系数都很接近于0，难道不是那时候线之间的交点最多吗？
2. 岭迹图的评判标准，非常模糊。哪里才是最佳的喇叭口？哪里才是所谓的系数开始变得“平稳”的时候？一千个读者一千个哈姆雷特的画像？未免也太不严谨了。

难得一提的机器学习发展历史：过时的岭迹图

其实，岭迹图会有这样的问题不难理解。岭回归最初由Hoerl和Kennard在1970提出来用来改进多重共线性问题的模型，在这片1970年的论文中，两位作者提出了岭迹图并且向广大学者推荐这种方法，然而遭到了许多人的批评和反抗。大家接受了岭回归，却鲜少接受岭迹图，这使得岭回归被发明了50年之后，市面上关于岭迹图的教材依然只有1970年的论文中写的那几句话。

1974年，Stone M发表论文，表示应当在统计学和机器学习中使用交叉验证。1980年代，机器学习技术迎来第一次全面爆发（1979年ID3决策树被发明出来，1980年之后CART树，adaboost，带软间隔的支持向量，梯度提升树逐渐诞生），从那之后，除了统计学家们，几乎没有人再使用岭迹图了。在2000年以后，岭迹图只是教学中会被略微提到的一个知识点（还会被强调是过时的技术），在现实中，真正应用来选择正则化系数的技术是交叉验证，并且选择的标准非常明确——我们选择让交叉验证下的均方误差最小的正则化系数 α 。

所以我们应该使用交叉验证来选择最佳的正则化系数。在sklearn中，我们有带交叉验证的岭回归可以使用，我们来看一看：

```
class sklearn.linear_model.RidgeCV(alphas=(0.1, 1.0, 10.0), fit_intercept=True, normalize=False, scoring=None, cv=None, gcv_mode=None, store_cv_values=False)
```

可以看到，这个类于普通的岭回归类Ridge非常相似，不过在输入正则化系数 α 的时候我们可以传入元祖作为正则化系数的备选，非常类似于我们在画学习曲线前设定的for i in 的列表对象。来看RidgeCV的重要参数，属性和接口：

重要参数	含义
alphas	需要测试的正则化参数的取值的元祖
scoring	用来进行交叉验证的模型评估指标，默认是 R^2 ，可自行调整
store_cv_values	是否保存每次交叉验证的结果，默认False
cv	交叉验证的模式，默认是None，表示默认进行留一交叉验证 可以输入Kfold对象和StratifiedKFold对象来进行交叉验证 注意，仅仅当为None时，每次交叉验证的结果才可以被保存下来 当cv有值存在（不是None）时，store_cv_values无法被设定为True
重要属性	含义
alpha_	查看交叉验证选中的alpha
cv_values_	调用所有交叉验证的结果，只有当store_cv_values=True的时候才能够调用，因此返回的结构是(n_samples, n_alphas)
重要接口	含义
score	调用Ridge类不进行交叉验证的情况下返回的R平方

这个类的使用也非常容易，依然使用我们之前建立的加利佛尼亚房屋价值数据集：

```
import numpy as np
import pandas as pd
from sklearn.linear_model import RidgeCV, LinearRegression
from sklearn.model_selection import train_test_split as TTS
from sklearn.datasets import fetch_california_housing as fch
import matplotlib.pyplot as plt

housevalue = fch()

X = pd.DataFrame(housevalue.data)
y = housevalue.target
X.columns = ["住户收入中位数", "房屋使用年代中位数", "平均房间数目",
             "平均卧室数目", "街区人口", "平均入住率", "街区的纬度", "街区的经度"]

Ridge_ = RidgeCV(alphas=np.arange(1, 1001, 100)
                  #, scoring="neg_mean_squared_error"
                  , store_cv_values=True
                  #, cv=5
                  ).fit(X, y)
```

```
#无关交叉验证的岭回归结果
Ridge_.score(X,y)

#调用所有交叉验证的结果
Ridge_.cv_values_.shape

#进行平均后可以查看每个正则化系数取值下的交叉验证结果
Ridge_.cv_values_.mean(axis=0)

#查看被选择出来的最佳正则化系数
Ridge_.alpha_
```

4.3 Lasso

4.3.1 Lasso与多重共线性

除了岭回归之外，最常被人们提到还有模型Lasso。Lasso全称最小绝对收缩和选择算子 (least absolute shrinkage and selection operator)，由于这个名字过于复杂所以简称为Lasso。和岭回归一样，Lasso是被创造来作用于多重共线性问题的算法，不过Lasso使用的是系数 w 的L1范式 (L1范式则是系数 w 的绝对值) 乘以正则化系数 α ，所以Lasso的损失函数表达式为：

$$\min_w \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \alpha\|\mathbf{w}\|_1$$

许多博客和机器学习教材会说，Lasso与岭回归非常相似，都是利用正则项来对原本的损失函数形成一个惩罚，以此来防止多重共线性。这种说法不是非常严谨，我们来看看Lasso的数学过程。当我们使用最小二乘法来求解Lasso中的参数 w ，我们依然对损失函数进行求导：

$$\begin{aligned} \frac{\partial(RSS + \|\mathbf{w}\|_1)}{\partial \mathbf{w}} &= \frac{\partial (\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \alpha\|\mathbf{w}\|_1)}{\partial \mathbf{w}} \\ &= \frac{\partial(\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w})}{\partial \mathbf{w}} + \frac{\partial\alpha\|\mathbf{w}\|_1}{\partial \mathbf{w}} \end{aligned}$$

前半部分我们推导过，后半部分对 w 求导和岭回归有巨大的不同

假设所有的系数都为正：

$$= 0 - 2\mathbf{X}^T \mathbf{y} + 2\mathbf{X}^T \mathbf{X}\mathbf{w} + \alpha$$

将含有 w 的项合并，其中 α 为常数

为了实现矩阵相加，让它乘以一个结构为 $n * n$ 的单位矩阵 I ：

$$\begin{aligned} &= \mathbf{X}^T \mathbf{X}\mathbf{w} - \mathbf{X}^T \mathbf{y} + \frac{\alpha I}{2} \\ \mathbf{X}^T \mathbf{X}\mathbf{w} &= \mathbf{X}^T \mathbf{y} - \frac{\alpha I}{2} \end{aligned}$$

大家可能已经注意到了，现在问题又回到了要求 $\mathbf{X}^T \mathbf{X}$ 的逆必须存在。在岭回归中，我们通过正则化系数 α 能够向方阵 $\mathbf{X}^T \mathbf{X}$ 加上一个单位矩阵，以此来防止方阵 $\mathbf{X}^T \mathbf{X}$ 的行列式为0，而现在L1范式所带的正则项 α 在求导之后并不带有 w 这个项，因此它无法对 $\mathbf{X}^T \mathbf{X}$ 造成任何影响。也就是说，**Lasso无法解决特征之间“精确相关”的问题**。当我们使用最小二乘法求解线性回归时，如果线性回归无解或者报除零错误，换Lasso不能解决任何问题。

岭回归 vs Lasso

岭回归可以解决特征间的精确相关关系导致的最小二乘法无法使用的问题，而Lasso不行。

幸运的是，在现实中我们其实会比较少遇到“精确相关”的多重共线性问题，大部分多重共线性问题是“高度相关”，而如果我们假设方阵 $\mathbf{X}^T \mathbf{X}$ 的逆是一定存在的，那我们可以有：

$$\mathbf{w} = (\mathbf{X}^T \mathbf{X})^{-1} (\mathbf{X}^T \mathbf{y} - \frac{\alpha \mathbf{I}}{2})$$

通过增大 α ，我们可以为 w 的计算增加一个负项，从而限制参数估计中 w 的大小，而防止多重共线性引起的参数 w 被估计过大导致模型失准的问题。**Lasso不是从根本上解决多重共线性问题，而是限制多重共线性带来的影响**。何况，这还是在我们假设所有的系数都为正的情况下，假设系数 w 无法为正，则很有可能我们需要将我们的正则项参数 α 设定为负，因此 α 可以取负数，并且负数越大，对共线性的限制也越大。

所有这些让Lasso成为了一个神奇的算法，尽管它是为了限制多重共线性被创造出来的，然而世人其实并不使用它来抑制多重共线性，反而接受了它在其他方面的优势。我们在讲解逻辑回归时曾提到过，L1和L2正则化一个核心差异就是他们对系数 w 的影响：两个正则化都会压缩系数 w 的大小，对标签贡献更少的特征的系数会更小，也会更容易被压缩。不过，L2正则化只会将系数压缩到尽量接近0，但L1正则化主导稀疏性，因此会将系数压缩到0。这个性质，让Lasso成为了线性模型中的特征选择工具首选，接下来，我们就来看看如何使用Lasso来选择特征。

4.3.2 Lasso的核心作用：特征选择

```
class sklearn.linear_model.Lasso(alpha=1.0, fit_intercept=True, normalize=False, precompute=False,
copy_X=True, max_iter=1000, tol=0.0001, warm_start=False, positive=False, random_state=None, selection='cyclic')
```

sklearn中我们使用类Lasso来调用lasso回归，众多参数中我们需要比较在意的就是参数 α ，正则化系数。另外需要注意的就是参数positive。当这个参数为"True"的时候，是我们要求Lasso回归出的系数必须为正数，以此来保证我们的 α 一定以增大来控制正则化的程度。

需要注意的是，在sklearn中我们的Lasso使用的损失函数是：

$$\min_w \frac{1}{2n_{samples}} \| \mathbf{X} \mathbf{w} - \mathbf{y} \|_2^2 + \alpha \| \mathbf{w} \|_1$$

其中 $\frac{1}{2n_{samples}}$ 只是作为系数存在，用来消除我们对损失函数求导后多出来的那个2的（求解 w 时所带的1/2），然后对整体的RSS求了一个平均而已，无论时从损失函数的意义来看还是从Lasso的性质和功能来看，这个变化没有造成任何影响，只不过计算上会更加简便一些。

接下来，我们就来看看lasso如何做特征选择：

```
import numpy as np
import pandas as pd
from sklearn.linear_model import Ridge, LinearRegression, Lasso
from sklearn.model_selection import train_test_split as TTS
```

```

from sklearn.datasets import fetch_california_housing as fch
import matplotlib.pyplot as plt

housevalue = fch()

X = pd.DataFrame(housevalue.data)
y = housevalue.target
X.columns = ["住户收入中位数", "房屋使用年代中位数", "平均房间数目",
             "平均卧室数目", "街区人口", "平均入住率", "街区的纬度", "街区的经度"]

X.head()

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

#恢复索引
for i in [Xtrain,Xtest]:
    i.index = range(i.shape[0])

#线性回归进行拟合
reg = LinearRegression().fit(Xtrain,Ytrain)
(reg.coef_*100).tolist()

#岭回归进行拟合
Ridge_ = Ridge(alpha=0).fit(Xtrain,Ytrain)
(Ridge_.coef_*100).tolist()

#Lasso进行拟合
Lasso_ = Lasso(alpha=0).fit(Xtrain,Ytrain)
(Lasso_.coef_*100).tolist()

```

可以看到，岭回归没有报出错误，但Lasso就不一样了，虽然依然对系数进行了计算，但是报出了整整三个红条：

```

C:\Python\lib\site-packages\ipykernel_launcher.py:2: UserWarning: With alpha=0, this algorithm does not converge well. You are advised to use the LinearRegression estimator

C:\Python\lib\site-packages\sklearn\linear_model\coordinate_descent.py:478: UserWarning: Coordinate descent with no regularization may lead to unexpected results and is discouraged.
  positive)

C:\Python\lib\site-packages\sklearn\linear_model\coordinate_descent.py:492: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations. Fitting data with very small alpha may cause precision problems.
  ConvergenceWarning)

```

这三条分别是这样的内容：

1. 正则化系数为0，这样算法不可收敛！如果你想让正则化系数为0，请使用线性回归吧
2. 没有正则项的坐标下降法可能会导致意外的结果，不鼓励这样做！
3. 目标函数没有收敛，你也许想要增加迭代次数，使用一个非常小的alpha来拟合模型可能会造成精确度问题！

看到这三条内容，大家可能会比较懵——怎么出现了坐标下降？这是由于sklearn中的Lasso类不是使用最小二乘法来进行求解，而是使用坐标下降。考虑一下，Lasso既然不能够从根本解决多重共线性引起的最小二乘法无法使用的问题，那我们为什么要坚持最小二乘法呢？明明有其他更快更好的求解方法，比如坐标下降就很好呀。下面两篇论文解释了scikit-learn坐标下降求解器中使用的迭代方式，以及用于收敛控制的对偶间隙计算方式，感兴趣的大家可以进行阅读。

使用坐标下降法求解Lasso

"Regularization Path For Generalized linear Models by Coordinate Descent", Friedman, Hastie & Tibshirani, J Stat Softw, 2010 ([Paper](#)).

"An Interior-Point Method for Large-Scale L1-Regularized Least Squares," S. J. Kim, K. Koh, M. Lustig, S. Boyd and D. Gorinevsky, in IEEE Journal of Selected Topics in Signal Processing, 2007 ([Paper](#))

有了坐标下降，就有迭代和收敛的问题，因此sklearn不推荐我们使用0这样的正则化系数。如果我们的的确希望取到0，那我们可以使用一个比较很小的数，比如 0.01 ，或者 $10 * e^{-3}$ 这样的值：

```
#岭回归进行拟合
Ridge_ = Ridge(alpha=0.01).fit(Xtrain,Ytrain)
(Ridge_.coef_*100).tolist()

#Lasso进行拟合
lasso_ = Lasso(alpha=0.01).fit(Xtrain,Ytrain)
(lasso_.coef_*100).tolist()
```

这样就不会报任何错误了。

```
#加大正则项系数，观察模型的系数发生了什么变化
Ridge_ = Ridge(alpha=10**4).fit(Xtrain,Ytrain)
(Ridge_.coef_*100).tolist()

lasso_ = Lasso(alpha=10**4).fit(Xtrain,Ytrain)
(lasso_.coef_*100).tolist()

#看来10**4对于Lasso来说是一个过于大的取值
lasso_ = Lasso(alpha=1).fit(Xtrain,Ytrain)
(lasso_.coef_*100).tolist()

#将系数进行绘图
plt.plot(range(1,9),(reg.coef_*100).tolist(),color="red",label="LR")
plt.plot(range(1,9),(Ridge_.coef_*100).tolist(),color="orange",label="Ridge")
plt.plot(range(1,9),(lasso_.coef_*100).tolist(),color="k",label="Lasso")
plt.plot(range(1,9),[0]*8,color="grey",linestyle="--")
plt.xlabel('w') #横坐标是每一个特征所对应的系数
plt.legend()
plt.show()
```

可见，比起岭回归，Lasso所带的L1正则项对于系数的惩罚要重得多，并且它会将系数压缩至0，因此可以被用来做特征选择。也因此，我们往往让Lasso的正则化系数 α 在很小的空间中变动，以此来寻找最佳的正则化系数。

4.3.3 选取最佳的正则化参数取值

```
class sklearn.linear_model.LassoCV(eps=0.001, n_alphas=100, alphas=None, fit_intercept=True,
normalize=False, precompute='auto', max_iter=1000, tol=0.0001, copy_X=True, cv='warn', verbose=False,
n_jobs=None, positive=False, random_state=None, selection='cyclic')
```

使用交叉验证的Lasso类的参数看起来与岭回归略有不同，这是由于Lasso对于alpha的取值更加敏感的性质决定的。之前提到过，由于Lasso对正则化系数的变动过于敏感，因此我们往往让 α 在很小的空间中变动。这个小空间小到超乎人们的想象（不是0.01到0.02之间这样的空间，这个空间对lasso而言还是太大了），因此我们设定了一个重要概念“**正则化路径**”，用来设定正则化系数的变动：

重要概念：正则化路径 regularization path

假设我们的特征矩阵中有 n 个特征，则我们就有特征向量 x_1, x_2, \dots, x_n 。对于每一个 α 的取值，我们都可以得出一组对应这个特征向量的参数向量 w ，其中包含了 $n+1$ 个参数，分别是 $w_0, w_1, w_2, \dots, w_n$ 。这些参数可以被看作是一个 $n+1$ 维空间中的一个点（想想我们在主成分分析和奇异值分解中讲解的 n 维空间）。对于不同的 α 取值，我们就将得到许多个在 $n+1$ 维空间中的点，所有的这些点形成的序列，就被我们称之为是正则化路径。

我们把形成这个正则化路径的 α 的最小值除以 α 的最大值得到的量 $\frac{\alpha_{min}}{\alpha_{max}}$ 称为正则化路径的长度 (length of the path)。在sklearn中，我们可以通过规定正则化路径的长度（即限制 α 的最小值和最大值之间的比例），以及路径中 α 的个数，来让sklearn为我们自动生成 α 的取值，这就避免了我们需要自己生成非常非常小的 α 的取值列表来让交叉验证类使用，类LassoCV自己就可以计算了。

和岭回归的交叉验证类相似，除了进行交叉验证之外，LassoCV也会单独建立模型。它会先找出最佳的正则化参数，然后在这个参数下按照模型评估指标进行建模。需要注意的是，LassoCV的模型评估指标选用的是均方误差，而岭回归的模型评估指标是可以自己设定的，并且默认是 R^2 。

参数	含义
eps	正则化路径的长度，默认0.001
n_alphas	正则化路径中 α 的个数，默认100
alphas	需要测试的正则化参数的取值的元祖，默认None。当不输入的时候，自动使用eps和n_alphas来自动生成带入交叉验证的正则化参数
cv	交叉验证的次数，默认3折交叉验证，将在0.22版本中改为5折交叉验证
属性	含义
alpha_	调用交叉验证选出来的最佳正则化参数
alphas_	使用正则化路径的长度和路径中 α 的个数来自动生成的，来进行交叉验证的正则化参数
mse_path	返回所以交叉验证的结果细节
coef_	调用最佳正则化参数下建立的模型的系数

来看看将这些参数和属性付诸实践的代码：

```
from sklearn.linear_model import LassoCV

#自己建立Lasso进行alpha选择的范围
alpharange = np.logspace(-10, -2, 200, base=10)

#其实是形成10为底的指数函数
```

```
#10**(-10)到10**(-2)次方

alpharange.shape

xtrain.head()

lasso_ = LassoCV(alphas=alpharange #自行输入的alpha的取值范围
                  ,cv=5 #交叉验证的折数
                  ).fit(Xtrain, Ytrain)

#查看被选择出来的最佳正则化系数
lasso_.alpha_

#调用所有交叉验证的结果
lasso_.mse_path_

lasso_.mse_path_.shape #返回每个alpha下的五折交叉验证结果

lasso_.mse_path_.mean(axis=1) #有注意到在岭回归中我们的轴向是axis=0吗?

#在岭回归当中，我们是留一验证，因此我们的交叉验证结果返回的是，每一个样本在每个alpha下的交叉验证结果
#因此我们要求每个alpha下的交叉验证均值，就是axis=0，跨行求均值
#而在这里，我们返回的是，每一个alpha取值下，每一折交叉验证的结果
#因此我们要求每个alpha下的交叉验证均值，就是axis=1，跨列求均值

#最佳正则化系数下获得的模型的系数结果
lasso_.coef_

lasso_.score(Xtest,Ytest)

#与线性回归相比如何？
reg = LinearRegression().fit(Xtrain,Ytrain)
reg.score(Xtest,Ytest)

#使用LassoCV自带的正则化路径长度和路径中的alpha个数来自动建立alpha选择的范围
ls_ = LassoCV(eps=0.00001
              ,n_alphas=300
              ,cv=5
              ).fit(Xtrain, Ytrain)

ls_.alpha_

ls_.alphas_ #查看所有自动生成的alpha取值

ls_.alphas_.shape

ls_.score(Xtest,Ytest)

ls_.coef_
```

到这里，岭回归和Lasso的核心作用就为大家讲解完毕了。时间缘故无法为大家将坐标下降法展开来解释，Lasso作为线性回归家族中在改良上走得最远的算法，还有许多领域等待我们去探讨。比如说，在现实中，我们不仅可以适用交叉验证来选择最佳正则化系数，我们也可以使用BIC（贝叶斯信息准则）或者AIC（Akaike information criterion，艾凯克信息准则）来做模型选择。同时，我们可以不使用坐标下降法，还可以使用最小角度回归来对lasso进行计算。

当然了，这些方法下做的模型选择和模型计算，其实在模型效果上表现和普通的Lasso没有太大的区别，不过他们在各个方面对原有的Lasso做了一些相应的改进（比如说提升了本来就已经很快的计算速度，增加了模型选择的维度，因为均方误差作为损失函数只考虑了偏差，不考虑方差的存在）。除了解决多重共线性这个核心问题之外，线性模型还有更重要的事情要做：提升模型表现。这才是机器学习最核心的需求，而Lasso和岭回归不是为此而设计的。下一节，让我们来认识一下为了提升模型表现而做出的改进：多项式回归。

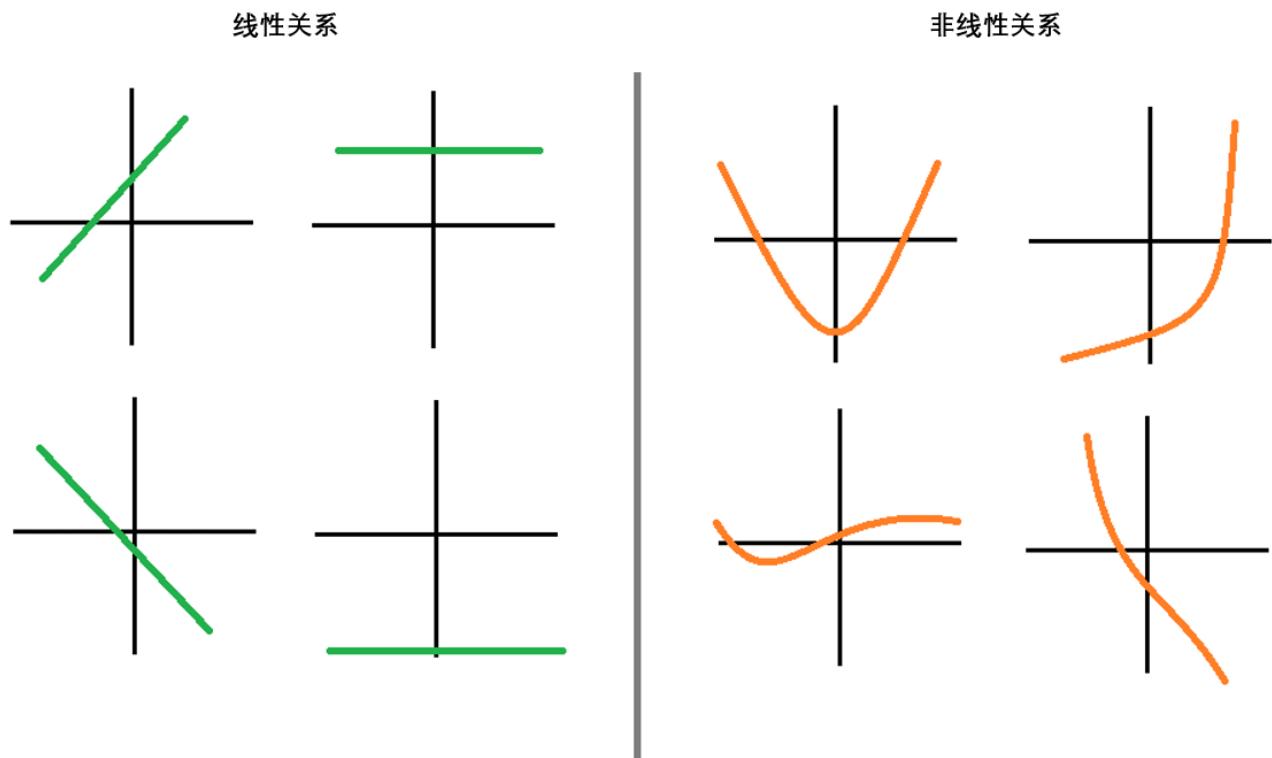
5 非线性问题：多项式回归

5.1 重塑我们心中的“线性”概念

在机器学习和统计学中，甚至在我们之前的课程中，我们无数次提到“线性”这个名词。首先我们本周的算法就叫做“线性回归”，而在支持向量机中，我们也曾经提到最初的支持向量机只能够分割线性可分的数据，然后引入了“核函数”来帮助我们分类那些非线性可分的数据。我们也曾经说起过，比如说决策树，支持向量机是“非线性”模型。所有的这些概念，让我们对“线性”这个词非常熟悉，却又非常陌生——因为我们并不知道它的真实含义。在这一小节，我将来为大家重塑线性的概念，并且为大家解决线性回归模型改进的核心之一：帮助线性回归解决非线性问题。

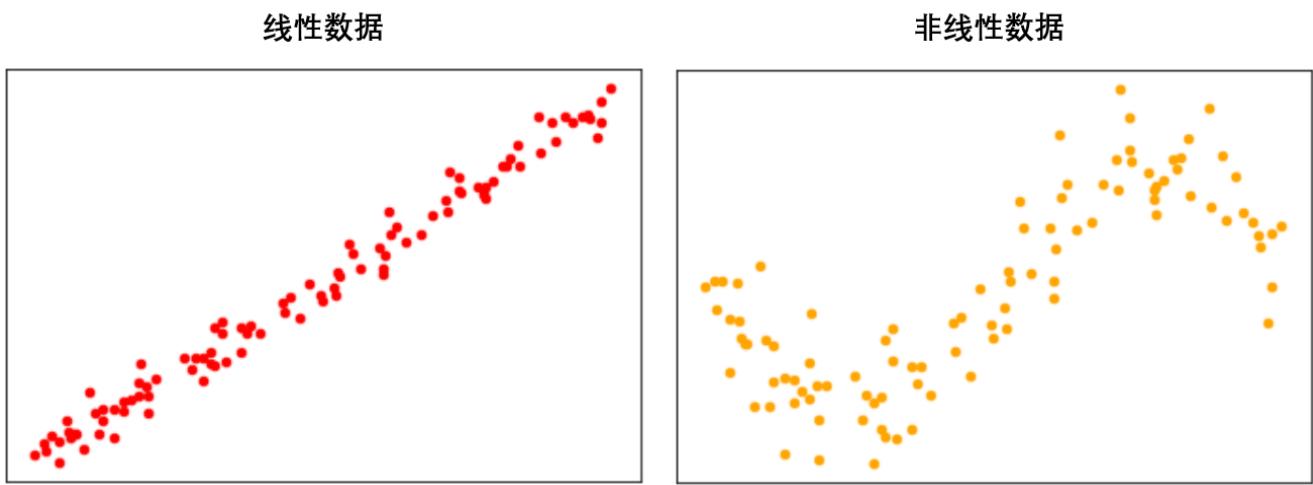
5.1.1 变量之间的线性关系

首先，“线性”这个词用于描述不同事物时有着不同的含义。我们最常使用的线性是指“**变量之间的线性关系** (linear relationship)”，它表示两个变量之间的关系可以展示为一条直线，即可以使用方程 $y = ax + b$ 来进行拟合。要探索两个变量之间的关系是否是线性的，最简单的方式就是绘制散点图，如果散点图能够相对均匀地分布在一条直线的两端，则说明这两个变量之间的关系是线性的。因此，三角函数(如 $\sin(x)$)，高次函数($y = ax^3 + b, (a \neq 0)$)，指数函数($y = e^x$)等等图像不为直线的函数所对应的自变量和因变量之间是非线性关系 (non-linear relationship)。
 $y = ax + b$ 也因此被称为线性方程或线性函数 (linear function)，三角函数，高次函数等也因此被称为非线性函数 (non-linear function)。

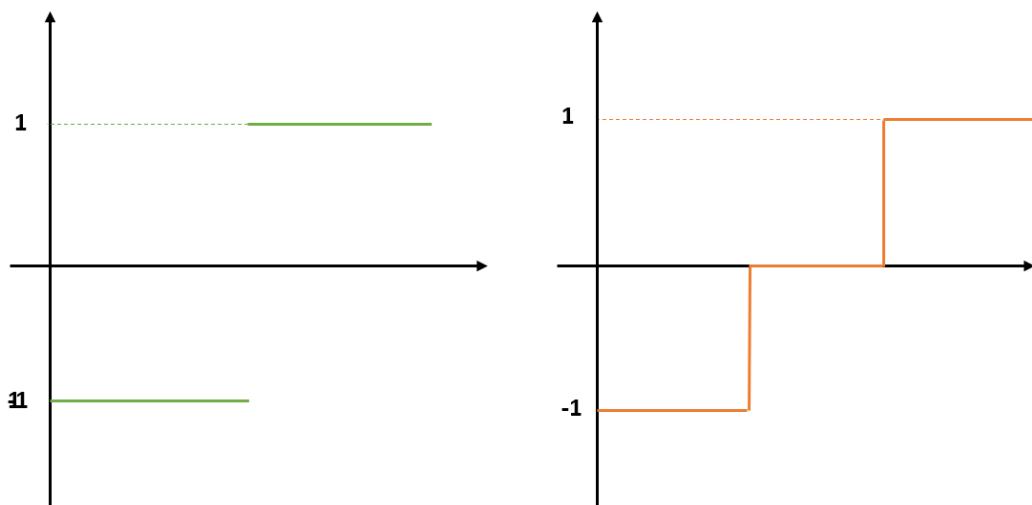


5.1.2 数据的线性与非线性

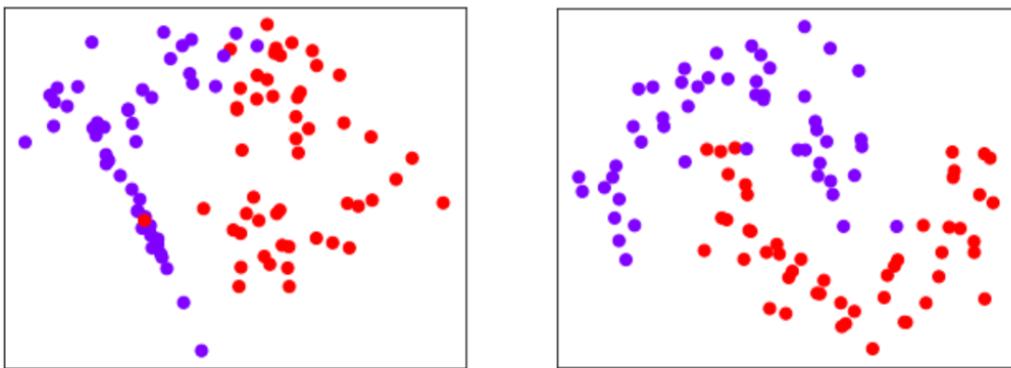
从线性关系这个概念出发，我们有了一种说法叫做“**线性数据**”。通常来说，一组数据由多个特征和标签组成。当这些特征分别与标签存在线性关系的时候，我们就说这一组数据是线性数据。当特征矩阵中任意一个特征与标签之间的关系需要使用三角函数，指数函数等函数来定义，则我们就说这种数据叫做“**非线性数据**”。对于线性和非线性数据，最简单的判别方法就是利用模型来帮助我们——如果是做分类则使用逻辑回归，如果做回归则使用线性回归，如果效果好那数据是线性的，效果不好则数据不是线性的。当然，也可以降维后进行绘图，绘制出的图像分布接近一条直线，则数据就是线性的。



不难发现，都这里为止我们为大家展示的都是或多或少能够连成线的数据分布，他们之间只不过是直线与曲线的区别罢了。然而考虑一下，当我们在进行分类的时候，我们的决策函数往往是一个分段函数，比如二分类下的决策函数可以是符号函数 $sign(x)$ ，符号函数的图像可以表示为取值为1和-1的两条直线。这个函数明显不符合我们所说的可以用一条直线来进行表示的属性，因此**分类问题中特征与标签[0,1]或者[-1,1]之间关系明显是非线性的关系**。除非我们在拟合分类的概率，否则不存在例外。



同时我们还注意到，当我们在进行分类的时候，我们的数据分布往往是这样的：



可以看得出，这些数据都不能由一条直线来进行拟合，他们也没有均匀分布在某一条线的周围，那我们怎么判断，这些数据是线性数据还是非线性数据呢？在这里就要注意了，当我们在回归中绘制图像时，绘制的是特征与标签的关系图，横坐标是特征，纵坐标是标签，我们的标签是连续型的，所以我们可以使用是否能够使用一条直线来拟合图像判断数据究竟属于线性还是非线性。然而在分类中，我们绘制的是数据分布图，横坐标是其中一个特征，纵坐标是另一个特征，标签则是数据点的颜色。因此在分类数据中，我们使用“**是否线性可分**”(linearly separable)这个概念来划分分类数据集。当分类数据的分布上可以使用一条直线来将两类数据分开时，我们就说数据是线性可分的。反之，数据不是线性可分的。

总结一下，对于回归问题，数据若能分布为一条直线，则是线性的，否则是非线性。对于分类问题，数据分布若能使用一条直线来划分类别，则是线性可分的，否则数据则是线性不可分的。

5.1.3 线性模型与非线性模型

在回归中，线性数据可以使用如下的方程来进行拟合：

$$y = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 \dots w_n x_n$$

也就是我们的线性回归的方程。根据线性回归的方程，我们可以拟合出一组参数 w ，在这一组固定的参数下我们可以建立一个模型，而这个模型就被我们称之为**线性回归模型**。所以建模的过程就是寻找参数的过程。此时此刻我们建立的线性回归模型，是一个**用于拟合线性数据的线性模型**。作为线性模型的典型代表，我们可以从线性回归的方程中总结出线性模型的特点：**其自变量都是一次项**。

那线性回归在非线性数据上的表现如何呢？我们来建立一个明显是非线性的数据集，并观察线性回归和决策树的而回归在拟合非线性数据集时的表现：

1. 导入所需要的库

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
```

2. 创建需要拟合的数据集

```
rnd = np.random.RandomState(42) #设置随机数种子
x = rnd.uniform(-3, 3, size=100) #random.uniform, 从输入的任意两个整数中取出size个随机数

#生成y的思路：先使用NumPy中的函数生成一个sin函数图像，然后再人为添加噪音
y = np.sin(x) + rnd.normal(size=len(x)) / 3 #random.normal, 生成size个服从正态分布的随机数

#使用散点图观察建立的数据集是什么样子
plt.scatter(x, y, marker='o', c='k', s=20)
plt.show()

#为后续建模做准备：sklearn只接受二维以上数组作为特征矩阵的输入
x.shape

x = x.reshape(-1, 1)
```

3. 使用原始数据进行建模

```
#使用原始数据进行建模
LinearR = LinearRegression().fit(x, y)
TreeR = DecisionTreeRegressor(random_state=0).fit(x, y)

#放置画布
fig, ax1 = plt.subplots(1)

#创建测试数据：一系列分布在横坐标上的点
line = np.linspace(-3, 3, 1000, endpoint=False).reshape(-1, 1)

#将测试数据带入predict接口，获得模型的拟合效果并进行绘制
ax1.plot(line, LinearR.predict(line), linewidth=2, color='green',
          label="linear regression")
ax1.plot(line, TreeR.predict(line), linewidth=2, color='red',
          label="decision tree")

#将原数据上的拟合绘制在图像上
ax1.plot(x[:, 0], y, 'o', c='k')

#其他图形选项
ax1.legend(loc="best")
ax1.set_ylabel("Regression output")
ax1.set_xlabel("Input feature")
```

```
ax1.set_title("Result before discretization")
plt.tight_layout()
plt.show()
```

#从这个图像来看，可以得出什么结果？

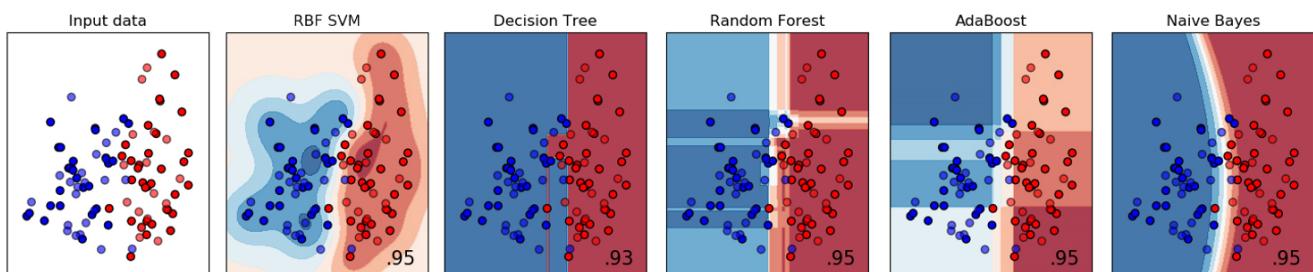
从图像上可以看出，线性回归无法拟合出这条带噪音的正弦曲线的真实面貌，只能够模拟出大概的趋势，而决策树却通过建立复杂的模型将几乎每个点都拟合出来了。可见，使用线性回归模型来拟合非线性数据的效果并不好，而决策树这样的模型却拟合得太细致，但是相比之下，还是决策树的拟合效果更好一些。

决策树无法写出一个方程（我们在XGBoost章节中会详细讲解如何将决策树定义成一个方程，但它绝对不是一个形似 $y = ax + b$ 的方程），它是一个典型的非线性模型，当它被用于拟合非线性数据，可以发挥奇效。其他典型的非线性模型还包括使用高斯核的支持向量机，树的集成算法，以及一切通过三角函数，指数函数等非线性方程来建立的模型。

根据这个思路，我们也许可以这样推断：线性模型用于拟合线性数据，非线性模型用于拟合非线性数据。但事实上机器学习远远比我们想象的灵活得多，**线性模型可以用来拟合非线性数据，而非线性模型也可以用来拟合线性数据，更神奇的是，有的算法没有模型也可以处理各类数据，而有的模型既可以是线性，也可以是非线性模型！**接下来，我们就来——讨论这些问题。

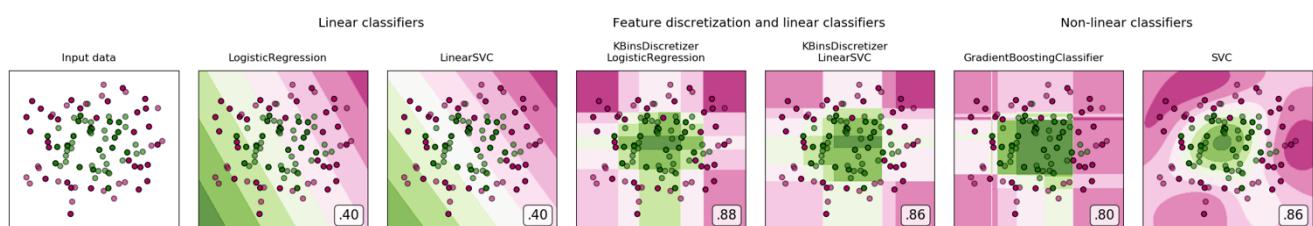
• 非线性模型拟合线性数据

非线性模型能够拟合或处理线性数据的例子非常多，我们在之前的课程中多次为大家展示了非线性模型诸如决策树，随机森林等算法在分类中处理线性可分的数据的效果。无一例外的，非线性模型们几乎都可以在线性可分数据上有不逊于线性模型的表现。同样的，如果我们使用随机森林来拟合一条直线，那随机森林毫无疑问会过拟合，因为线性数据对于非线性模型来说太过简单，很容易就把训练集上的 R^2 训练得很高，MSE训练的很低。



• 线性模型拟合非线性数据

但是相反的，线性模型若用来拟合非线性数据或者对非线性可分的数据进行分类，那通常都会表现糟糕。通常如果我们已经发现数据属于非线性数据，或者数据非线性可分的数据，则我们不会选择使用线性模型来进行建模。改善线性模型在非线性数据上的效果的方法之一时进行分箱，并且从下图来看分箱的效果不是一般的好，甚至高过一些非线性模型。在下一节中我们会详细来讲解分箱的效果，但很容易注意到，在没有其他算法或者预处理帮忙的情况下，线性模型在非线性数据上的表现时很糟糕的。

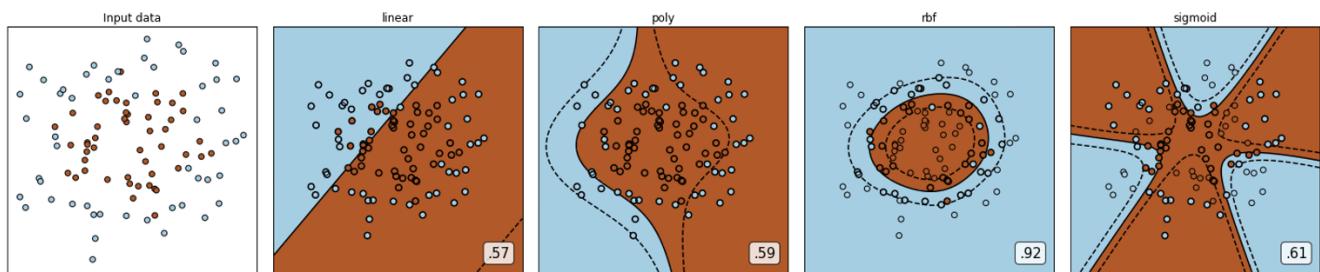


从上面的图中，我们可以观察出一个特性：线性模型们的决策边界都是一条条平行的直线，而非线性模型们的决策边界是交互的直线（格子），曲线，环形等等。对于分类模型来说，这是我们判断模型是线性还是非线性的重要评判因素：**线性模型的决策边界是平行的直线，非线性模型的决策边界是曲线或者交叉的直线。**之前我们提到，模型上如果自变量上的最高次方为1，则模型是线性的，但这种方式只适用于回归问题。分类模型中，我们很少讨论模型是否线性，因为我们很少使用线性模型来执行分类任务（逻辑回归是一个特例）。但从上面我们总结出的结果来看，我们可以认为**对分类问题而言，如果一个分类模型的决策边界上自变量的最高次方为1，则我们称这个模型是线性模型。**

- **既是线性，也是非线性的模型**

对于有一些模型来说，他们既可以处理线性模型又可以处理非线性模型，比如说强大的支持向量机。支持向量机的前身是感知机模型，朴实的感知机模型是实打实的线性模型（其决策边界是直线），在线性可分数据上表现优秀，但在非线性可分的数据上基本属于无法使用状态。

但支持向量机就不一样了。支持向量机本身也是处理线性可分数据的，但却可以通过对数据进行升维（将数据 x 转移到高维空间 Φ 中），将非线性可分数据变成高维空间中的线性可分数据，然后使用相应的“核函数”来求解。当我们选用线性核函数“linear”的时候，数据没有进行变换，支持向量机中就是线性模型，此时它的决策边界是直线。而当我们选用非线性核函数比如高斯径向基核函数的时候，数据进行了升维变化，此时支持向量机就是非线性模型，此时它的决策边界在二维空间中是曲线。所以这个模型可以在线性和非线性之间自由切换，一切取决于它的核函数。



还有更加特殊的，没有模型的算法，比如最近邻算法KNN，这些都是不建模，但是能够直接预测出标签或做出判断的算法。而这些算法，并没有线性非线性之分，单纯的是不建模的算法们。

讨论到这里，相信大家对于线性和非线性模型的概念就比较清楚了。来看看下面这张表的总结：

	线性模型	非线性模型
代表模型	线性回归，逻辑回归，弹性网，感知机	决策树，树的集成模型，使用高斯核的SVM
模型特点	模型简单，运行速度快	模型复杂，效果好，但速度慢
数学特征：回归	自变量是一次项	自变量不都是一次项
分类	决策边界上的自变量都是一次项	决策边界上的自变量不都是一次项
可视化：回归	拟合出的图像是一条直线	拟合出的图像不是一条直线
分类	决策边界在二维平面是一条直线	决策边界在二维平面不是一条直线
擅长数据类型	主要是线性数据，线性可分数据	所有数据

模型在线性和非线性数据集上的表现为我们选择模型提供了一个思路：当我们获取数据时，我们往往希望使用线性模型来对数据进行最初的拟合（线性回归用于回归，逻辑回归用于分类），如果线性模型表现良好，则说明数据本身很可能是线性的或者线性可分的，如果线性模型表现糟糕，那毫无疑问我们会投入决策树，随机森林这些模型的怀抱，就不必浪费时间在线性模型上了。

不过这并不代表着我们就完全不能使用线性模型来处理非线性数据了。在现实中，线性模型有着不可替代的优势：计算速度异常快速，所以也还是存在着我们无论如何也希望使用线性回归的情况。因此，我们有多种手段来处理线性回归无法拟合非线性问题的问题，接下来我们就来看一看。

5.2 使用分箱处理非线性问题

让线性回归在非线性数据上表现提升的核心方法之一是对数据进行分箱，也就是离散化。与线性回归相比，我们常用的一种回归是决策树的回归。我们之前拟合过一条带有噪音的正弦曲线以展示多元线性回归与决策树的效用差异，我们来分析一下这张图，然后再使用采取措施帮助我们的线性回归。

1. 导入所需要的库

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
```

2. 创建需要拟合的数据集

```
rnd = np.random.RandomState(42) #设置随机数种子
x = rnd.uniform(-3, 3, size=100) #random.uniform, 从输入的任意两个整数中取出size个随机数

#生成y的思路：先使用NumPy中的函数生成一个sin函数图像，然后再人为添加噪音
y = np.sin(x) + rnd.normal(size=len(x)) / 3 #random.normal, 生成size个服从正态分布的随机数

#使用散点图观察建立的数据集是什么样子
plt.scatter(x, y, marker='o', c='k', s=20)
plt.show()

#为后续建模做准备：sklearn只接受二维以上数组作为特征矩阵的输入
x.shape

x = x.reshape(-1, 1)
```

3. 使用原始数据进行建模

```
#使用原始数据进行建模
LinearR = LinearRegression().fit(x, y)
TreeR = DecisionTreeRegressor(random_state=0).fit(x, y)

#放置画布
fig, ax1 = plt.subplots(1)

#创建测试数据：一系列分布在横坐标上的点
line = np.linspace(-3, 3, 1000, endpoint=False).reshape(-1, 1)
```

```

#将测试数据带入predict接口，获得模型的拟合效果并进行绘制
ax1.plot(line, LinearR.predict(line), linewidth=2, color='green',
         label="linear regression")
ax1.plot(line, TreeR.predict(line), linewidth=2, color='red',
         label="decision tree")

#将原数据上的拟合绘制在图像上
ax1.plot(x[:, 0], y, 'o', c='k')

#其他图形选项
ax1.legend(loc="best")
ax1.set_ylabel("Regression output")
ax1.set_xlabel("Input feature")
ax1.set_title("Result before discretization")
plt.tight_layout()
plt.show()

#从这个图像来看，可以得出什么结果？

```

从图像上可以看出，线性回归无法拟合出这条带噪音的正弦曲线的真实面貌，只能够模拟出大概的趋势，而决策树却通过建立复杂的模型将几乎每个点都拟合出来了。此时此刻，决策树正处于过拟合的状态，对数据的学习过于细致，而线性回归处于拟合不足的状态，这是由于模型本身只能在线性关系间进行拟合的性质决定的。为了让线性回归在类似的数据上变得更加强大，我们可以使用分箱，也就是离散化连续型变量的方法来处理原始数据，以此来提升线性回归的表现。来看看我们如何实现：

4. 分箱及分箱的相关问题

```

from sklearn.preprocessing import KBinsDiscretizer

#将数据分箱
enc = KBinsDiscretizer(n_bins=10 #分几类?
                       , encode="onehot") #ordinal
x_binned = enc.fit_transform(x)
#encode模式"onehot": 使用哑变量方式做离散化
#之后返回一个稀疏矩阵(m, n_bins)，每一列是一个分好的类别
#对每一个样本而言，它包含的分类（箱子）中它表示为1，其余分类中它表示为0

x.shape

x_binned

#使用pandas打开稀疏矩阵
import pandas as pd
pd.DataFrame(x_binned.toarray()).head()

#我们将使用分箱后的数据来训练模型，在sklearn中，测试集和训练集的结构必须保持一致，否则报错
LinearR_ = LinearRegression().fit(x_binned, y)

LinearR_.predict(line) #line作为测试集

line.shape #测试

```

```
x_binned.shape #训练

#因此我们需要创建分箱后的测试集：按照已经建好的分箱模型将line分箱
line_binned = enc.transform(line)

line_binned.shape #分箱后的数据是无法进行绘图的

line_binned

LinearR_.predict(line_binned).shape
```

5. 使用分箱数据进行建模和绘图

```
#准备数据
enc = KBinsDiscretizer(n_bins=10, encode="onehot")
x_binned = enc.fit_transform(x)
line_binned = enc.transform(line)

#将两张图像绘制在一起，布置画布
fig, (ax1, ax2) = plt.subplots(ncols=2
                               , sharey=True #让两张图共享y轴上的刻度
                               , figsize=(10, 4))

#在图1中布置在原始数据上建模的结果
ax1.plot(line, LinearR_.predict(line), linewidth=2, color='green',
          label="linear regression")
ax1.plot(line, TreeR_.predict(line), linewidth=2, color='red',
          label="decision tree")
ax1.plot(x[:, 0], y, 'o', c='k')
ax1.legend(loc="best")
ax1.set_ylabel("Regression output")
ax1.set_xlabel("Input feature")
ax1.set_title("Result before discretization")

#使用分箱数据进行建模
LinearR_ = LinearRegression().fit(x_binned, y)
TreeR_ = DecisionTreeRegressor(random_state=0).fit(x_binned, y)

#进行预测，在图2中布置在分箱数据上进行预测的结果
ax2.plot(line #横坐标
          , LinearR_.predict(line_binned) #分箱后的特征矩阵的结果
          , linewidth=2
          , color='green'
          , linestyle='--'
          , label='linear regression')

ax2.plot(line, TreeR_.predict(line_binned), linewidth=2, color='red',
          linestyle=':', label='decision tree')

#绘制和箱宽一致的竖线
ax2.vlines(enc.bin_edges_[0] #x轴
            , *plt.gca().get_ylim() #y轴的上限和下限
            , linewidth=1
```

```

    , alpha=.2)

#将原始数据分布放置在图像上
ax2.plot(x[:, 0], y, 'o', c='k')

#其他绘图设定
ax2.legend(loc="best")
ax2.set_xlabel("Input feature")
ax2.set_title("Result after discretization")
plt.tight_layout()
plt.show()

```

从图像上可以看出，离散化后线性回归和决策树上的预测结果完全相同了——线性回归比较成功地拟合了数据的分布，而决策树的过拟合效应也减轻了。由于特征矩阵被分箱，因此特征矩阵在每个区域内获得的值是恒定的，因此所有模型对同一个箱中所有的样本都会获得相同的预测值。与分箱前的结果相比，线性回归明显变得更加灵活，而决策树的过拟合问题也得到了改善。但注意，一般来说我们是不使用分箱来改善决策树的过拟合问题的，因为树模型带有丰富而有效的剪枝功能来防止过拟合。

在这个例子中，我们设置的分箱箱数为10，不难想到这个箱数的设定肯定会影响模型最后的预测结果，我们来看看不同的箱数会如何影响回归的结果：

6. 箱子数如何影响模型的结果

```

enc = KBinsDiscretizer(n_bins=5, encode="onehot")
x_binned = enc.fit_transform(x)
line_binned = enc.transform(line)

fig, ax2 = plt.subplots(1, figsize=(5,4))

LinearR_ = LinearRegression().fit(x_binned, y)
print(LinearR_.score(line_binned, np.sin(line)))
TreeR_ = DecisionTreeRegressor(random_state=0).fit(x_binned, y)

ax2.plot(line #横坐标
         , LinearR_.predict(line_binned) #分箱后的特征矩阵的结果
         , linewidth=2
         , color='green'
         , linestyle='--'
         , label='linear regression')
ax2.plot(line, TreeR_.predict(line_binned), linewidth=2, color='red',
         linestyle=':', label='decision tree')
ax2.vlines(enc.bin_edges_[0], *plt.gca().get_ylim(), linewidth=1, alpha=.2)
ax2.plot(x[:, 0], y, 'o', c='k')
ax2.legend(loc="best")
ax2.set_xlabel("Input feature")
ax2.set_title("Result after discretization")
plt.tight_layout()
plt.show()

```

7. 如何选取最优的箱数

```
#怎样选取最优的箱子?
```

```

from sklearn.model_selection import cross_val_score as CVS
import numpy as np

pred,score,var = [],[],[]
binsrange = [2,5,10,15,20,30]
for i in binsrange:
    #实例化分箱类
    enc = KBinsDiscretizer(n_bins=i, encode="onehot")
    #转换数据
    X_binned = enc.fit_transform(X)
    line_binned = enc.transform(line)
    #建立模型
    LinearR_ = LinearRegression()
    #全数据集上的交叉验证
    cvresult = CVS(LinearR_,X_binned,y, cv=5)
    score.append(cvresult.mean())
    var.append(cvresult.var())
    #测试数据集上的打分结果
    pred.append(LinearR_.fit(X_binned,y).score(line_binned,np.sin(line)))
#绘制图像
plt.figure(figsize=(6,5))
plt.plot(binsrange,pred,c="orange",label="test")
plt.plot(binsrange, score, c="k", label="full data")
plt.plot(binsrange, score+np.array(var)*0.5, c="red", linestyle="--", label = "var")
plt.plot(binsrange, score-np.array(var)*0.5, c="red", linestyle="--")
plt.legend()
plt.show()

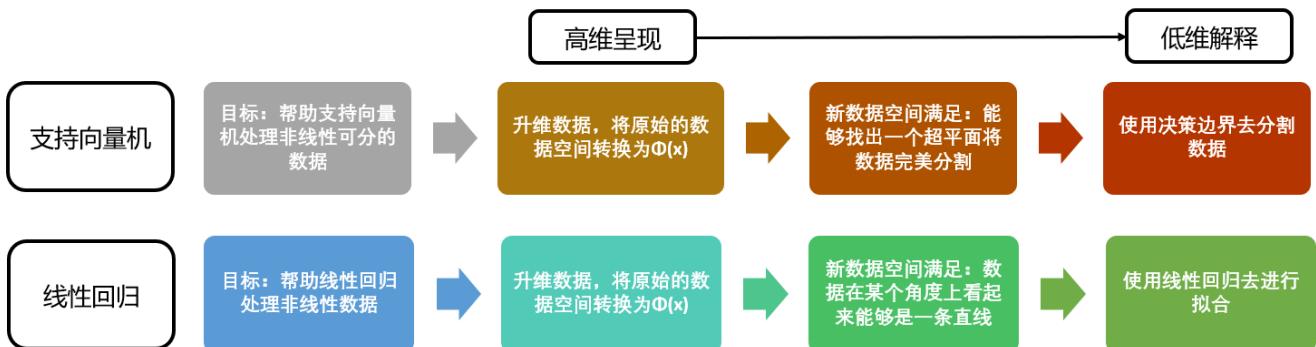
```

在工业中，大量离散化变量与线性模型连用的实例很多，在深度学习出现之前，这种模式甚至一度统治一些工业中的机器学习应用场景，可见效果优秀，应用广泛。对于现在的很多工业场景而言，大量离散化特征的情况可能已经不是那么多了，不过大家依然需要对“分箱能够解决线性模型无法处理非线性数据的问题”有所了解。

5.3 多项式回归PolynomialFeatures

5.3.1 多项式对数据做了什么

除了分箱之外，另一种更普遍的用于解决“线性回归只能处理线性数据”问题的手段，就是使用多项式回归对线性回归进行改进。这样的手法是机器学习研究者们从支持向量机中获得的：支持向量机通过升维可以将非线性可分数据转化为线性可分，然后使用核函数在低维空间中进行计算，这是一种“高维呈现，低维解释”的思维。那我们为什么不能让线性回归使用类似于升维的转换，将数据由非线性转换为线性，从而为线性回归赋予处理非线性数据的能力呢？当然可以。



接下来，我们就来看看线性模型中的升维工具：**多项式变化**。这是一种通过增加自变量上的次数，而将数据映射到高维空间的方法，只要我们设定一个自变量上的次数（大于1），就可以相应地获得数据投影在高次方的空间中的结果。这种方法可以非常容易地通过sklearn中的类PolynomialFeatures来实现。我们先来简单看看这个类是如何使用的。

```
class sklearn.preprocessing.PolynomialFeatures(degree=2, interaction_only=False, include_bias=True)
```

参数	含义
degree	多项式中的次数，默认为2
interaction_only	布尔值是否只产生交互项，默认为False
include_bias	布尔值，是否产出与截距项相乘的 x_0 ，默认True

```

from sklearn.preprocessing import PolynomialFeatures
import numpy as np

#如果原始数据是一维的
X = np.arange(1,4).reshape(-1,1)
X

#二次多项式，参数degree控制多项式的次方
poly = PolynomialFeatures(degree=2)

#接口transform直接调用
X_ = poly.fit_transform(X)

X_
X_.shape

#三次多项式
PolynomialFeatures(degree=3).fit_transform(X)
  
```

不难注意到，多项式变化后数据看起来不太一样了：首先，数据的特征（维度）增加了，这正符合我们希望的将数据转换到高维空间的愿望。其次，维度的增加是有一定的规律的。不难发现，如果我们本来的特征矩阵中只有一个特征 x ，而转换后我们得到：

```


$$\begin{array}{cccc} x_0 & x & x^2 & x^3 \end{array}$$

array([[ 1.,  1.,  1.,  1.],
       [ 1.,  2.,  4.,  8.],
       [ 1.,  3.,  9., 27.]])

```

这个规律在转换为二次多项式的时候同样适用。原本，我们的模型应该是形似 $y = ax + b$ 的结构，而转换后我们的特征变化导致了模型的变化。根据我们在支持向量机中的经验，现在这个被投影到更高维空间中的数据在某个角度上看起来已经是一条直线了，于是我们可以**继续使用线性回归来进行拟合**。线性回归是会对每个特征拟合出权重 w 的，所以当我们拟合高维数据的时候，我们会得到下面的模型：

$$y = w_0 x_0 + w_1 x + w_2 x^2 + w_3 x^3, \quad (x_0 = 1)$$

由此推断，假设多项式转化的次数是 n ，则数据会被转化成形如：

$$[1, x, x^2, x^3 \dots x^n]$$

而拟合出的方程也可以被改写成：

$$y = w_0 x_0 + w_1 x + w_2 x^2 + w_3 x^3 \dots w_n x^n, \quad (x_0 = 1)$$

这就是大家会在大多数数学和机器学习教材中会看到的“**多项式回归**”的表达式。这个过程看起来非常简单，只不过是将原始的 x 上的次方增加，并且为这些次方项都加上权重 w ，然后增加一列所有次方为0的列作为截距系数的 x_0 ，参数`include_bias`就是用来控制 x_0 的生成的。

```
#三次多项式，不带与截距项相乘的x0
PolynomialFeatures(degree=3, include_bias=False).fit_transform(x)
```

```
#为什么我们会希望不生成与截距相乘的x0呢？
#对于多项式回来说，我们已经为线性回归准备好了x0，但是线性回归并不知道
xxx = PolynomialFeatures(degree=3).fit_transform(x)
```

```
xxx.shape
```

```
rnd = np.random.RandomState(42) #设置随机数种子
y = rnd.randn(3)
```

```
y
```

```
#生成了多少个系数？
LinearRegression().fit(xxx,y).coef_
```

```
#查看截距
LinearRegression().fit(xxx,y).intercept_
```

```
#发现问题了吗？线性回归并没有把多项式生成的x0当作是截距项
#所以我们可以选择：关闭多项式回归中的include_bias
#也可以选择：关闭线性回归中的fit_intercept
```

```
#生成了多少个系数？
LinearRegression(fit_intercept=False).fit(xxx,y).coef_
```

#查看截距

`LinearRegression(fit_intercept=False).fit(xxx,y).intercept_`

不过，这是一维状况的表达，大多数时候我们的原始特征矩阵不可能会是一维的，至少也是二维以上，很多时候还可能存在上千个特征或者维度。现在我们来看看原始特征矩阵是二维的状况：

```
x = np.arange(6).reshape(3, 2)
x

#尝试二次多项式
PolynomialFeatures(degree=2).fit_transform(x)
```

很明显，上面一维的转换公式已经不适用了，但如果我们仔细看，是可以看出这样的规律的：

$$\begin{array}{cccccc} x_0 & x_1 & x_2 & x_1^2 & x_1x_2 & x_2^2 \\ \text{array}([[1., 0., 1., 0., 0., 1.], \\ [1., 2., 3., 4., 6., 9.], \\ [1., 4., 5., 16., 20., 25.]]) \end{array}$$

当原始特征为二维的时候，多项式的二次变化突然将特征增加到了六维，其中一维是常量（也就是截距）。当我们继续适用线性回归去拟合的时候，我们会得到的方程如下：

$$\begin{aligned} [x_1, x_2] &\rightarrow [x_0, x_1, x_2, x_1^2, x_1x_2, x_2^2] \\ y = w_0x_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_1x_2 + w_5x_2^2 \end{aligned}$$

这个时候大家可能就会感觉到比较困惑了，怎么会出现这样的变化？如果想要总结这个规律，可以继续来尝试三次多项式：

#尝试三次多项式

`PolynomialFeatures(degree=3).fit_transform(x)`

很明显，我们可以看出这次生成的数据有这样的规律：

$$\begin{array}{cccccccccc} x_0 & x_1 & x_2 & x_1^2 & x_1x_2 & x_2^2 & x_1^3 & x_1^2x_2 & x_1x_2^2 & x_2^3 \\ \text{array}([[1., 0., 1., 0., 0., 1., 0., 0., 0., 1.], \\ [1., 2., 3., 4., 6., 9., 8., 12., 18., 27.], \\ [1., 4., 5., 16., 20., 25., 64., 80., 100., 125.]]) \end{array}$$

不难发现：当我们进行多项式转换的时候，多项式会产出到最高次数为止的所有低高次项。比如如果我们规定多项式的次数为2，多项式就会产出所有次数为1和次数为2的项反馈给我们，相应的如果我们规定多项式的次数为n，则多项式会产出所有从次数为1到次数为n的项。注意， x_1x_2 和 x_1^2 一样都是二次项，一个自变量的平方其实也就相当于 x_1x_1 ，所以在三次多项式中 $x_1^2x_2$ 就是三次项。

在多项式回归中，我们可以规定是否产生平方或者立方项，其实如果我们只要求高次项的话， x_1x_2 会是一个比 x_1^2 更好的高次项，因为 x_1x_2 和 x_1 之间的共线性会比 x_1^2 与 x_1 之间的共线性好那么一点点（只是一点点），而我们多项式转化之后是需要使用线性回归模型来进行拟合的，就算机器学习中不是那么在意数据上的基本假设，但是太过分的共线性还是会影响到模型的拟合。因此sklearn中存在着控制是否要生成平方和立方项的参数interaction_only，默认为False，以减少共线性。来看这个参数是如何工作的：

```
PolynomialFeatures(degree=2).fit_transform(X)

PolynomialFeatures(degree=2, interaction_only=True).fit_transform(X)

#对比之下，当interaction_only为True的时候，只生成交互项
```

从之前的许多次尝试中我们可以看出，随着多项式的次数逐渐变高，特征矩阵会被转化得越来越复杂。不仅是次数，当特征矩阵中的维度数（特征数）增加的时候，多项式同样会变得更加复杂：

```
#更高维度的原始特征矩阵
X = np.arange(9).reshape(3, 3)
X

PolynomialFeatures(degree=2).fit_transform(X)

PolynomialFeatures(degree=3).fit_transform(X)

X_ = PolynomialFeatures(degree=20).fit_transform(X)

X_.shape
```

如此，多项式变化对于数据会有怎样的影响就一目了然了：随着原特征矩阵的维度上升，随着我们规定的最高次数的上升，数据会变得越来越复杂，维度越来越多，并且这种维度的增加并不能用太简单的数学公式表达出来。**因此，多项式回归没有固定的模型表达式**，多项式回归的模型最终长什么样子是由数据和最高次数决定的，因此我们无法断言说某个数学表达式“就是多项式回归的数学表达”，因此要求解多项式回归不是一件容易的事儿，感兴趣的大家可以自己去尝试看看用最小二乘法求解多项式回归。接下来，我们就来看看多项式回归的根本作用：处理非线性问题。

5.3.2 多项式回归处理非线性问题

之前我们说过，是希望通过这种将数据投影到高维的方式来帮助我们解决非线性问题。那我们现在就来看一看多项式转化对模型造成了什么样的影响：

```
from sklearn.preprocessing import PolynomialFeatures as PF
from sklearn.linear_model import LinearRegression
import numpy as np

rnd = np.random.RandomState(42) #设置随机数种子
X = rnd.uniform(-3, 3, size=100)
y = np.sin(X) + rnd.normal(size=len(X)) / 3

#将X升维，准备好放入sklearn中
X = X.reshape(-1, 1)

#创建测试数据，均匀分布在训练集X的取值范围内的一千个点
line = np.linspace(-3, 3, 1000, endpoint=False).reshape(-1, 1)

#原始特征矩阵的拟合结果
LinearR = LinearRegression().fit(X, y)
#对训练数据的拟合
```

```

LinearR.score(x,y)

#对测试数据的拟合
LinearR.score(line,np.sin(line))

#多项式拟合，设定高次项
d=5

#进行高此项转换
poly = PF(degree=d)
X_ = poly.fit_transform(x)
line_ = PF(degree=d).fit_transform(line)

#训练数据的拟合
LinearR_ = LinearRegression().fit(X_, y)
LinearR_.score(X_,y)

#测试数据的拟合
LinearR_.score(line_,np.sin(line))

```

如果我们将这个过程可视化：

```

import matplotlib.pyplot as plt

d=5
#和上面展示一致的建模流程
LinearR = LinearRegression().fit(x, y)
X_ = PF(degree=d).fit_transform(x)
LinearR_ = LinearRegression().fit(X_, y)
line = np.linspace(-3, 3, 1000, endpoint=False).reshape(-1, 1)
line_ = PF(degree=d).fit_transform(line)

#放置画布
fig, ax1 = plt.subplots(1)

#将测试数据带入predict接口，获得模型的拟合效果并进行绘制
ax1.plot(line, LinearR.predict(line), linewidth=2, color='green'
          ,label="linear regression")
ax1.plot(line, LinearR_.predict(line_), linewidth=2, color='red'
          ,label="Polynomial regression")

#将原数据上的拟合绘制在图像上
ax1.plot(X[:, 0], y, 'o', c='k')

#其他图形选项
ax1.legend(loc="best")
ax1.set_ylabel("Regression output")
ax1.set_xlabel("Input feature")
ax1.set_title("Linear Regression ordinary vs poly")
plt.tight_layout()
plt.show()

#来一起鼓掌，感叹多项式回归的神奇

```

```
#随后可以试试看较低和较高的次方会发生什么变化
#d=2
#d=20
```

从这里大家可以看出，多项式回归能够较好地拟合非线性数据，还不容易发生过拟合，可以说是保留了线性回归作为线性模型所带的“不容易过拟合”和“计算快速”的性质，同时又实现了优秀地拟合非线性数据。到了这里，相信大家对于多项式回归的效果已经不再怀疑了。多项式回归非常迷人也非常神奇，因此一直以来都有各种各样围绕着多项式回归进行的讨论。在这里，为大家梳理几个常见问题和讨论，供大家参考。

5.3.3 多项式回归的可解释性

线性回归是一个具有高解释性的模型，它能够对每个特征拟合出参数 w 以帮助我们理解每个特征对于标签的作用。当我们进行了多项式转换后，尽管我们还是形成形如线性回归的方程，但随着数据维度和多项式次数的上升，方程也变得异常复杂，我们可能无法一眼看出增维后的特征是由之前的什么特征组成的（之前我们都是肉眼看肉眼判断）。不过，多项式回归的可解释性依然是存在的，我们可以使用接口`get_feature_names`来调用生成的新特征矩阵的各个特征上的名称，以便帮助我们解释模型。来看下面的例子：

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

X = np.arange(9).reshape(3, 3)
X

poly = PolynomialFeatures(degree=5).fit(X)

#重要接口get_feature_names
poly.get_feature_names()
```

使用加利佛尼亚房价数据集给大家作为例子，当我们有标签名称的时候，可以直接在接口`get_feature_names()`中输入标签名称来查看新特征究竟是由原特征矩阵中的什么特征组成的：

```
from sklearn.datasets import fetch_california_housing as fch
import pandas as pd

housevalue = fch()

X = pd.DataFrame(housevalue.data)

y = housevalue.target

housevalue.feature_names

X.columns = ["住户收入中位数", "房屋使用年代中位数", "平均房间数目",
             "平均卧室数目", "街区人口", "平均入住率", "街区的纬度", "街区的经度"]

poly = PolynomialFeatures(degree=2).fit(X,y)

poly.get_feature_names(X.columns)
```

```

x_ = poly.transform(x)

#在这之后，我们依然可以直接建立模型，然后使用线性回归的coef_属性来查看什么特征对标签的影响最大
reg = LinearRegression().fit(x_,y)

coef = reg.coef_

[*zip(poly.get_feature_names(x_.columns),reg.coef_)]

#放到DataFrame中进行排序
coeff = pd.DataFrame([poly.get_feature_names(x_.columns),reg.coef_.tolist()]).T
coeff.columns = ["feature","coef"]

coeff.sort_values(by="coef")

```

可以发现，不仅数据的可解释性还存在，我们还可以通过这样的手段做特征工程——特征创造。多项式帮助我们进行了一系列特征之间相乘的组合，若能够找出组合起来后对标签贡献巨大的特征，那我们就是创造了新的有效特征，对于任何学科而言发现新特征都是非常有价值的。

在加利佛尼亚房屋价值数据集上来再次确认多项式回归提升模型表现的能力：

```

#顺便可以查看一下多项式变化之后，模型的拟合效果如何了
poly = PolynomialFeatures(degree=4).fit(x,y)
x_ = poly.transform(x)

reg = LinearRegression().fit(x,y)
reg.score(x,y)

from time import time
time0 = time()
reg_ = LinearRegression().fit(x_,y)
print("R2:{}".format(reg_.score(x_,y)))
print("time:{}".format(time()-time0))

#假设使用其他模型？
from sklearn.ensemble import RandomForestRegressor as RFR

time0 = time()
print("R2:{}".format(RFR(n_estimators=100).fit(x,y).score(x,y)))
print("time:{}".format(time()-time0))

```

5.3.4 线性还是非线性模型？

另一个围绕多项式回归的核心问题是，多项式回归是一个线性模型还是非线性模型呢？从我们之前对线性模型的定义来看，自变量上需要没有高次项才是线性模型，按照这个定义，在 x 上填上高次方的多项式回归肯定不是线性模型了。然而事情却没有这么简单。来看原始特征为二维，多项式次数为二次的多项式回归表达式：

$$y = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_1 x_2 + w_5 x_2^2$$

经过变化后的数据有六个特征，分别是：

$$[x_0, x_1, x_2, x_1^2, x_1 x_2, x_2^2]$$

我们能够一眼看出从第四个特征开始，都是高次特征，而这些高次特征与 y 之间的关系必然不是线性的。但是我们也可以换一种方式来思考这个问题：假设我们不知道这些特征是由多项式变化改变来的，我们只是拿到了含有六个特征的任意数据，于是现在对于我们来说这六个特征就是：

$$[z_0, z_1, z_2, z_3, z_4, z_5]$$

我们通过检验发现， z_1 和 z_4 , z_5 之间存在一定的共线性， z_2 也是如此，但是现实中的数据不太可能完全不相关，因此一部分的共线性是合理的，我们没有任何理由去联想到说，这个数据其实是由多项式变化来生成的。所以我们了线性回归来对数据进行拟合，然后得到了方程：

$$y = w_0 z_0 + w_1 z_1 + w_2 z_2 + w_3 z_3 + w_4 z_4 + w_5 z_5 \quad (2)$$

这妥妥的是一个线性方程没错：并不存在任何高次项，只要拟合结果不错，大概也没有人会在意这个六个特征的原始数据究竟是怎么来的，那多项式回归不就变成一个含有部分共线性的线性方程了么？在许多教材中，多项式回归被称为“线性回归的一种特殊情况”，这就是为什么许多人会产生混淆，多项式回归究竟是线性模型还是非线性模型呢？这就需要聊到我们对“线性模型”的狭义和广义定义了。

狭义线性模型 vs 广义线性模型

狭义线性模型：自变量上不能有高此项，自变量与标签之间不能存在非线性关系。

广义线性模型：只要标签与模型拟合出的**参数之间的关系是线性的**，模型就是线性的。这是说，只要生成的一系列 w 之间没有相乘或者相除的关系，我们就认为模型是线性的。

就多项式回归本身的性质来说，如果我们考虑狭义线性模型的定义，那它肯定是一种非线性模型没有错——否则如何能够处理非线性数据呢，并且在统计学中我们认为，特征之间若存在精确相关关系或高度相关关系，线性模型的估计就会被“扭曲”，从而失真或难以估计准确。多项式正是利用线性回归的这种“扭曲”，为线性模型赋予了处理非线性数据的能力。但如果我们考虑广义线性模型的定义，多项式回归就是一种线性模型，毕竟它的系数 w 之间也没有相乘或者相除。

另外，当Python在处理数据时，它并不知道这些特征是由多项式变化来的，它只注意到这些特征之间高度相关，然而既然你让我使用线性回归，那我就忠实执行命令，因此Python看待数据的方式是我们提到的第二种：并不了解数据之间的真实关系的建模。于是Python会为我们建立形如式子(2)的模型。这也证明了多项式回归是广义线性模型。所以，我们认为多项式回归是一种特殊的线性模型，不过要记得，它中间的特征包含了相当的共线性，如果处理线性数据，是会严重失误的。

关于究竟是线性还是非线性的更多讨论，大家可以参考这一篇非常优秀的stackexchange问答：

<https://stats.stackexchange.com/questions/92065/why-is-polynomial-regression-considered-a-special-case-of-multiple-linear-regres>

总结一下，**多项式回归通常被认为是非线性模型，但广义上它是一种特殊的线性模型**，它能够帮助我们处理非线性数据，是线性回归的一种进化。大家要能够理解多项式回归的争议从哪里来，并且能够解释清楚观察多项式回归的不同角度，以避免混淆。

另外一个需要注意的点是，线性回归进行多项式变化后被称为多项式回归，但这并不代表多项式变化只能与线性回归连用。在现实中，多项式变化疯狂增加数据维度的同时，也增加了过拟合的可能性，因此多项式变化多与能够处理过拟合的线性模型如岭回归，Lasso等来连用，与在线性回归上使用的效果是一致的，感兴趣的话大家可以自己尝试一下。

到这里，多项式回归就全部讲解完毕了。多项式回归主要是通过对自变量上的次方进行调整，来为线性回归赋予更多的学习能力，它的核心表现在提升模型在现有数据集上的表现。

6 结语

本章之中，大家学习了多元线性回归，岭回归，Lasso和多项式回归总计四个算法，他们都是围绕着原始的线性回归进行的拓展和改进。其中岭回归和Lasso是为了解决多元线性回归中使用最小二乘法的各种限制，主要用途是消除多重共线性带来的影响并且做特征选择，而多项式回归解决了线性回归无法拟合非线性数据的明显缺点，核心作用是提升模型的表现。除此之外，本章还定义了多重共线性和各种线性相关的概念，并为大家补充了一些线性代数知识。回归算法属于原理简单，但操作困难的机器学习算法，在实践和理论上都还有很长的路可以走，希望大家继续探索，让线性回归大家族中的算法真正称为大家的武器。



sklearn中的朴素贝叶斯

小伙伴们晚上好~o(￣▽￣)ゞ

我是菜菜，这里是我的sklearn课堂第十期，本期的内容是朴素贝叶斯~

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第十期

sklearn中的朴素贝叶斯

1 概述

1.1 真正的概率分类器

1.2 朴素贝叶斯是如何工作的

1.2.1瓢虫冬眠：理解 $P(Y|X)$

1.2.2 贝叶斯的性质与最大后验估计

1.2.3 汉堡称重：连续型变量的概率估计

1.3 sklearn中的朴素贝叶斯

2 不同分布下的贝叶斯

2.1 高斯朴素贝叶斯GaussianNB

2.1.1 认识高斯朴素贝叶斯

2.1.2 探索贝叶斯：高斯朴素贝叶斯擅长的数据集

2.1.3 探索贝叶斯：高斯朴素贝叶斯的拟合效果与运算速度

2.2 概率类模型的评估指标

2.2.1 布里尔分数Brier Score

2.2.2 对数似然函数Log Loss

2.2.3 可靠性曲线Reliability Curve

2.2.4 预测概率的直方图

2.2.5 校准可靠性曲线

2.3 多项式朴素贝叶斯以及其变化

2.3.1 多项式朴素贝叶斯MultinomialNB

2.3.2 伯努利朴素贝叶斯BernoulliNB

2.3.3 探索贝叶斯：贝叶斯的样本不均衡问题

2.3.4 改进多项式朴素贝叶斯：补集朴素贝叶斯ComplementNB

3 案例：贝叶斯分类器做文本分类

3.1 文本编码技术简介

3.1.1 单词计数向量

3.1.2 TF-IDF

3.2 探索文本数据

3.3 使用TF-IDF将文本数据编码

3.4 在贝叶斯上分别建模，查看结果

1 概述

1.1 真正的概率分类器

在许多分类算法应用中，特征和标签之间的关系并非是决定性的。比如说，我们想预测一个人究竟是否会在泰坦尼克号海难中生存下来，那我们可以建一棵决策树来学习我们的训练集。在训练中，其中一个人的特征为：30岁，男，普通舱，他最后在泰坦尼克号海难中去世了。当我们测试的时候，我们发现有另一个人的特征也为：30岁，男，普通舱。基于在训练集中的学习，我们的决策树必然会给这个人打上标签：去世。然而这个人的真实情况一定是去世了吗？并非如此。

也许这个人是心脏病患者，得到了上救生艇的优先权。又有可能，这个人就是挤上了救生艇，活了下来。对分类算法来说，基于训练的经验，这个人“很有可能”是没有活下来，但算法永远也无法确定“这个人一定没有活下来”。即便这个人最后真的没有活下来，算法也无法确定基于训练数据给出的判断，是否真的解释了这个人没有存活下来的真实情况。这就是说，**算法得出的结论，永远不是100%确定的，更多的是判断出了一种“样本的标签更可能是某类的可能性”，而非一种“确定”**。我们通过某些规定，比如说，在决策树的叶子节点上占比较多的标签，就是叶子节点上所有样本的标签，来强行让算法为我们返回一个固定结果。但许多时候，我们也希望能够理解算法判断出的可能性本身。

每种算法使用不同的指标来衡量这种可能性。比如说，决策树使用的就是叶子节点上占比较多的标签所占的比例（接口predict_proba调用），逻辑回归使用的是sigmoid函数压缩后的似然（接口predict_proba调用），而SVM使用的是样本点到决策边界的距离（接口decision_function调用）。但这些指标的本质，其实都是一种“类概率”的表示，我们可以通过归一化或sigmoid函数将这些指标压缩到0~1之间，让他们表示我们的模型对预测的结果究竟有多大的把握（置信度）。但无论如何，我们都希望使用真正的概率来衡量可能性，因此就有了真正的概率算法：朴素贝叶斯。

朴素贝叶斯是一种直接衡量标签和特征之间的概率关系的有监督学习算法，是一种专注分类的算法。朴素贝叶斯的算法根源就是基于概率论和数理统计的贝叶斯理论，因此它是根正苗红的概率模型。接下来，我们就来认识一下这个简单快速的概率算法。

1.2 朴素贝叶斯是如何工作的

朴素贝叶斯被认为是最简单的分类算法之一。首先，我们需要了解一些概率论的基本理论。假设有两个随机变量X和Y，他们分别可以取值为x和y。有这两个随机变量，我们可以定义两种概率：

关键概念：联合概率与条件概率

联合概率：“X取值为x”和“Y取值为y”两个事件同时发生的概率，表示为 $P(X = x, Y = y)$

条件概率：在“X取值为x”的前提下，“Y取值为y”的概率，表示为 $P(Y = y | X = x)$

举个例子，我们让X为“气温”，Y为“七星瓢虫冬眠”，则X和Y可能的取值分别为x和y，其中 $x = \{0, 1\}$ ，0表示没有下降到0度以下，1表示下降到了0度以下。 $y = \{0, 1\}$ ，其中0表示否，1表示是。

两个事件分别发生的概率就为：

$P(X = 1) = 50\%$ ，则是说明，气温下降到0度以下的可能性为50%，则 $P(X = 0) = 1 - P(X = 1) = 50\%$ 。

$P(Y = 1) = 70\%$ ，则是说明，七星瓢虫会冬眠的可能性为70%，则 $P(Y = 0) = 1 - P(Y = 1) = 30\%$ 。

则这两个事件的联合概率为 $P(X = 1, Y = 1)$ ，这个概率代表了气温下降到0度以下和七星瓢虫去冬眠这两件事情同时，独立发生的概率。

而两个事件之间的条件概率为 $P(Y = 1|X = 1)$, 这个概率代表了, 当气温下降到0度以下这个条件被满足之后, 七星瓢虫会去冬眠的概率。也就是说, 气温下降到0度以下, 一定程度上影响了七星瓢虫去冬眠这个事件。

在概率论中, 我们可以证明, 两个事件的联合概率等于这两个事件任意条件概率 * 这个条件事件本身的概率。

$$P(X = 1, Y = 1) = P(Y = 1|X = 1) * P(X = 1) = P(X = 1|Y = 1) * P(Y = 1)$$

简单一些, 则可以将上面的式子写成:

$$P(X, Y) = P(Y|X) * P(X) = P(X|Y) * P(Y)$$

由上面的式子, 我们可以得到贝叶斯理论等式:

$$P(Y|X) = \frac{P(X|Y) * P(Y)}{P(X)}$$

而这个式子, 就是我们一切贝叶斯算法的根源理论。我们可以把我们的特征 X 当成是我们的条件事件, 而我们要求解的标签 Y 当成是我们被满足条件后会被影响的结果, 而两者之间的概率关系就是 $P(Y|X)$, 这个概率在机器学习中, 被我们称之为是标签的后验概率 (posterior probability), 即是说我们先知道了条件, 再去求解结果。而标签 Y 在没有任何条件限制下取值为某个值的概率, 被我们写作 $P(Y)$, 与后验概率相反, 这是完全没有任何条件限制的, 标签的先验概率 (prior probability)。而我们的 $P(X|Y)$ 被称为“类的条件概率”, 表示当 Y 的取值固定的时候, X 为某个值的概率。那现在, 有趣的事情就出现了。

1.2.1 瓢虫冬眠: 理解 $P(Y|X)$

假设, 我们依然让 X 是“气温”, 这就是我们的特征, Y 是“七星瓢虫冬眠”, 就是我们的标签。现在, 我们建模的目的是, 预测七星瓢虫是否会冬眠。在许多教材和博客里, 大家会非常自然地开始说, 我们现在求的就是我们的 $P(Y|X)$, 然后根据贝叶斯理论等式开始做各种计算和分析。现在请问大家, 我写作 $P(Y|X)$ 的这个概率, 代表了什么呢? 更具体一点, 这个表达, 可以代表多少种概率呢?

$P(Y|X)$ 代表了多少种情况的概率?

$P(Y = 1|X = 1)$ 气温0度以下的条件下, 七星瓢虫冬眠的概率

$P(Y = 1|X = 0)$ 气温0度以上的条件下, 七星瓢虫冬眠的概率

$P(Y = 0|X = 1)$ 气温0度以下的条件下, 七星瓢虫没有冬眠的概率

$P(Y = 0|X = 0)$ 气温0度以上的条件下, 七星瓢虫没有冬眠的概率

数学中的第一个步骤, 也就是最重要的事情, 就是定义清晰。其实在数学中, $P(Y|X)$ 还真的就代表了全部的可能性, 而不是单一的概率本身。现在我们的 Y 有两种取值, 而 X 也有两种取值, 就让概率 $P(Y|X)$ 的定义变得很模糊, 排列组合之后竟然有4种可能。在机器学习当中, 一个特征 X 下取值可能远远不止两种, 标签也可能是多分类的, 还会有多个特征, 排列组合一下, 到底求解的 $P(Y|X)$ 是什么东西, 是一个太让人感到混淆的点。同理, $P(Y)$ 随着标签中分类的个数, 可以有不同的取值。 $P(X|Y)$ 也是一样。在这里, 我们来为大家澄清:

机器学习中的简写 $P(Y)$, 通常表示标签取到少数类的概率, 少数类往往使用正样本表示, 也就是 $P(Y = 1)$, 本质就是所有样本中标签为1的样本所占的比例。如果没有样本不均衡问题, 则必须在求解的时候明确, 你的 Y 的取值是什么。

而 $P(Y|X)$ 是对于任意一个样本而言，如果这个样本的特征 X 的取值为1，则表示求解 $P(Y = 1|X = 1)$ 。如果这个样本下的特征 X 取值为0，则表示求解 $P(Y = 1|X = 0)$ 。也就是说， $P(Y|X)$ 是具体到每一个样本上的，究竟求什么概率，由样本本身的特征的取值决定。每个样本的 $P(Y|X)$ 如果大于阈值0.5，则认为样本是少数类（正样本，1），如果这个样本的 $P(Y|X)$ 小于阈值0.5，则认为样本是多数类（负样本，0或者-1）。如果没有具体的样本，只是说明例子，则必须明确 $P(Y|X)$ 中 X 的取值。

在机器学习当中，对每一个样本，我们不可能只有一个特征 X ，而是会存在着包含n个特征的取值的特征向量 \mathbf{X} 。因此机器学习中的后验概率，被写作 $P(Y|\mathbf{X})$ ，其中 \mathbf{X} 中包含样本在n个特征 X_i 上的分别的取值 x_i ，由此可以表示为 $\mathbf{X} = \{X_1 = x_1, X_2 = x_2, \dots, X_n = x_n\}$ 。

因此，我们有：

一个样本上，所有特征取值下的概率：

每一个向量都是一个样本上的特征向量

$$\begin{aligned} &P(\mathbf{X}) \\ &P(\mathbf{x}) \\ &P(X_1 = x_1, X_2 = x_2, \dots, X_n = x_n) \\ &P(X_1, X_2, \dots, X_n) \\ &P(x_1, x_2, \dots, x_n) \end{aligned}$$

一个样本上，一个特征所取值下的概率：

是对于单独元素来说，没有向量存在

$$\begin{aligned} &P(X_i = x_i) \\ &P(X_i) \\ &P(x_i) \end{aligned}$$

一个特征上，所有样本下所取得的概率：

每一个向量都是一个特征下的样本向量

$$\begin{aligned} &P(\mathbf{X}_i) \\ &P(i) \end{aligned}$$

虽然写法不同，但其实都包含折同样的含义。以此为基础，机器学习中，**对每一个样本**我们有：

$$P(Y = 1|\mathbf{X}) = \frac{P(\mathbf{X}|Y = 1) * P(Y = 1)}{P(\mathbf{X})} = \frac{P(x_1, x_2, \dots, x_n|Y = 1) * P(Y = 1)}{P(x_1, x_2, \dots, x_n)}$$

对于分子而言， $P(Y = 1)$ 就是少数类占总样本量的比例， $P(\mathbf{X}|Y = 1)$ 则需要稍微复杂一点的过程来求解。假设我们只有两个特征 X_1, X_2 ，由联合概率公式，我们可以有如下证明：

$$\begin{aligned} P(X_1, X_2|Y = 1) &= \frac{P(X_1, X_2, Y = 1)}{P(Y = 1)} \\ &= \frac{P(X_1, X_2, Y = 1)}{P(X_2, Y = 1)} * \frac{P(X_2, Y = 1)}{P(Y = 1)} \\ &= P(X_1|X_2, Y = 1) * P(X_2|Y = 1) \end{aligned}$$

假设 X_1 和 X_2 之是有条件独立的：

$$= P(X_1|Y = 1) * P(X_2|Y = 1)$$

是一种若推广到n个 X 上，则有：

$$P(\mathbf{X}|Y=1) = \prod_{i=1}^n P(X_i = x_i|Y=1)$$

这个式子证明，在 $Y=1$ 的条件下，多个特征的取值被同时取到的概率，就等于 $Y=1$ 的条件下，多个特征的取值被分别取到的概率相乘。其中，假设 X_1 与 X_2 是有条件独立则可以让公式 $P(X_1|X_2, Y=1) = P(X_1|Y=1)$ ，这是在假设 X_2 是一个对 X_1 在某个条件下的取值完全无影响的变量。

比如说，温度(X_1)与观察到的瓢虫的数目(X_2)之间的关系。有人可能会说，温度在零下的时候，观察到的瓢虫数目往往很少。这种关系的存在可以通过一个中间因素：瓢虫会冬眠(Y)来解释。冬天瓢虫都冬眠了，自然观察不到很多瓢虫出来活动。也就是说，如果瓢虫冬眠的属性是固定的($Y=1$)，那么观察到的温度和瓢虫出没数目之间关系就会消失，因此无论是否还存在着“观察到的瓢虫的数目”这样的因素，我们都可以判断这只瓢虫到底会不会冬眠。这种情况下，我们就说，温度与观察到的瓢虫的数目，是条件独立的。

假设特征之间是有条件独立的，可以解决众多问题，也简化了很多计算过程，这是朴素贝叶斯被称为“朴素”的理由。因此，贝叶斯在特征之间有较多相关性的数据集上表现不佳，而现实中的数据多多少少都会有一些相关性，所以贝叶斯的分类效力在分类算法中不算特别强大。同时，一些影响特征本身的相关性的降维算法，比如PCA和SVD，和贝叶斯连用效果也会不佳。但无论如何，有了这个式子，我们就可以求解出我们的分子了。

接下来，来看看我们贝叶斯理论等式的分母 $P(\mathbf{X})$ 。我们可以使用全概率公式来求解 $P(\mathbf{X})$:

$$P(\mathbf{X}) = \sum_{i=1}^m P(y_i) * P(\mathbf{X}|Y_i)$$

其中 m 代表标签的种类，也就是说，对于二分类而言我们有：

$$P(\mathbf{X}) = P(Y=1) * P(\mathbf{X}|Y=1) + P(Y=0) * P(\mathbf{X}|Y=0)$$

基于这个方程，我们可以来求解一个非常简单的例子下的后验概率。

索引	温度 (X_1)	瓢虫的年龄 (X_2)	瓢虫冬眠 (Y)
0	零下	10天	是
1	零下	20天	是
2	零上	10天	否
3	零下	一个月	是
4	零下	20天	否
5	零上	两个月	否
6	零下	一个月	否
7	零下	两个月	是
8	零上	一个月	否
9	零上	10天	否
10	零下	20天	否

现在，我们希望预测零下的时候，年龄为20天的瓢虫，是否会冬眠。

$$P(Y = 1 | X_1 = \text{零下}, X_2 = 20\text{天}) = \frac{P(x_1, x_2 | Y = 1) * P(Y = 1)}{P(x_1, x_2)}$$

$$\begin{aligned} P(\text{冬眠} | \text{零下}, 20\text{天}) &= \frac{P(\text{零下}, 20\text{天} | \text{冬眠}) * P(\text{冬眠})}{P(\text{零下}, 20\text{天})} \\ &= \frac{P(\text{零下} | \text{冬眠}) * P(20\text{天} | \text{冬眠}) * P(\text{冬眠})}{P(\text{冬眠}) * P(\text{零下}, 20\text{天} | \text{冬眠}) + P(\text{不冬眠}) * P(\text{零下}, 20\text{天} | \text{不冬眠})} \end{aligned}$$

对于分子我们可以求得：

$$P(\text{冬眠}) = \frac{4}{11}, \quad P(\text{零下} | \text{冬眠}) = \frac{4}{4} = 1, \quad P(20\text{天} | \text{冬眠}) = \frac{1}{4}$$

对于分母我们可以求得：

$$P(\text{零下}, 20\text{天} | \text{冬眠}) = \frac{1}{4}, \quad P(\text{不冬眠}) = \frac{7}{11}, \quad P(\text{零下}, 20\text{天} | \text{不冬眠}) = \frac{2}{7}$$

所以我们的，温度为零下的时候，生活了20天的瓢虫，会冬眠的概率为：

$$\frac{4/11 * 1 * 1/4}{4/11 * 1/4 + 7/11 * 2/7} = \frac{1/11}{3/11} = 0.33$$

设定阈值为0.5，假设大于0.5的就是会冬眠，小于0.5的就是不会冬眠。根据我们的计算，我们认为一个在零下条件下，年龄为20天的瓢虫，是不会冬眠的。这就完成了一次预测。但是这样，有趣的地方又来了。刚才的预测过程是没有问题的。但我们总是好奇，这个过程如何对应sklearn当中的fit和predict呢？这个决策过程中，我们的训练集和我的测试集分别在哪里？以及，算法建模建模，我的模型在哪里呢？

1.2.2 贝叶斯的性质与最大后验估计

在过去的许多个星期内，我们学习的分类算法总是有一个特点：这些算法先从训练集中学习，获取某种信息来建立模型，然后用模型去对测试集进行预测。比如逻辑回归，我们要先从训练集中获取让损失函数最小的参数，然后用参数建立模型，再对测试集进行预测。在比如支持向量机，我们要先从训练集中获取让边际最大的决策边界，然后用决策边界对测试集进行预测。相同的流程在决策树，随机森林中也出现，我们在fit的时候必然已经构造好了能够让对测试集进行判断的模型。而朴素贝叶斯，似乎没有这个过程。

我给大家一张有标签的表，然后提出说，我要预测零下的时候，年龄为20天的瓢虫，会冬眠的概率，然后我们就顺利成章地算了出来。没有利用训练集求解某个模型的过程，也没有训练完毕了我们来做测试的过程，而是直接对有标签的数据提出要求，就可以得到预测结果了。

这说明，**朴素贝叶斯是一个不建模的算法**。以往我们学的不建模算法，比如KMeans，比如PCA，都是无监督学习，而朴素贝叶斯是第一个有监督的，不建模的分类算法。在我们刚才举的例子中，有标签的表格就是我们的训练集，而我提出的要求“零下的时候，年龄为20天的瓢虫”就是没有标签的测试集。我们认为，训练集和测试集都来自于同一个不可获得的大样本下，并且这个大样本下的各种属性所表现出来的规律应当是一致的，因此训练集上计算出来的各种概率，可以直接放到测试集上来使用。即便不建模，也可以完成分类。

但实际中，贝叶斯的决策过程并没有我们给出的例子这么简单。

$$P(Y=1|\mathbf{X}) = \frac{P(Y=1) * \prod_{i=1}^n P(x_i|Y=1)}{P(\mathbf{X})}$$

对于这个式子来说，从训练集中求解 $P(Y=1)$ 很容易，但 $P(\mathbf{X})$ 和 $P(x_i|Y=1)$ 这一部分就没有这么容易了。在我们的例子中，我们通过全概率公式来求解分母，两个特征就求解了四项概率。随着特征数目的逐渐变多，分母上的计算两会成指数级增长，而分子中的 $P(x_i|Y=1)$ 也越来越难计算。

不过幸运的是，对于同一个样本来说，在二分类状况下我们可以有：

$$P(Y=1|\mathbf{X}) = \frac{P(Y=1) * \prod_{i=1}^n P(x_i|Y=1)}{P(\mathbf{X})}$$

$$P(Y=0|\mathbf{X}) = \frac{P(Y=0) * \prod_{i=1}^n P(x_i|Y=0)}{P(\mathbf{X})}$$

并且：

$$P(Y=1|\mathbf{X}) + P(Y=0|\mathbf{X}) = 1$$

在分类的时候，我们选择 $P(Y=1|\mathbf{X})$ 和 $P(Y=0|\mathbf{X})$ 中较大的一个所对应的Y的取值，作为这个样本的分类。在比较两个类别的时候，两个概率计算的分母是一致的，因此我们可以不用计算分母，只考虑分子的大小。当我们分别计算出分子的大小之后，就可以通过让两个分子相加，来获得分母的值，以此来避免计算一个样本上所有特征下的概率 $P(\mathbf{X})$ 。这个过程，被我们称为“最大后验估计”(MAP)。在最大后验估计中，我们只需要求解分子，主要是求解一个样本下每个特征取值下的概率 $P(x_i|Y=y_i)$ ，再求连乘便能够获得相应的概率。

在现实中，要求解分子也会有各种各样的问题。比如说，测试集中出现的某种概率组合，是训练集中从未出现的状况，这种时候就会出现某一个概率为0的情况，贝叶斯概率的分子就会为0。还有，现实中的大多数标签还是连续型变量，要处理连续型变量的概率，就不是单纯的数样本个数的占比的问题了。接下来我们就来看看，如何对连续型特征求解概率。

1.2.3 汉堡称重：连续型变量的概率估计

要处理连续型变量，我们可以有两种方法。第一种是把连续型变量分成 j 个箱，把连续型强行变成分类型变量。我们分箱后，将每个箱中的均值 \bar{x}_i 当作一个特征 X_i 上的取值，然后我们计算箱 j 中 $Y=1$ 所占的比例，就是我们的 $P(x_i|Y=1)$ 。这个过程的主要问题是，箱子不能太大也不能太小，如果箱子太大，就失去了分箱的基本意义，如果箱子太小，可能每个箱子里就没有足够的样本来帮助我们计算 $P(x_i|Y)$ ，因此我们必须要适当地衡量我们的分箱效果。

但其实，我们没有必要这样做，因为我们可以直接通过概率论中来计算连续型变量的概率分布。在分类型变量的情况下，比如掷骰子的情况，我们有且仅有六种可能的结果1~6，并且每种结果的可能性为 $1/6$ 。此时每个基本的随机事件发生的概率都是相等的，所以我们可以使用 $\frac{1}{N}$ 来表示有 N 个基本随机事件可以发生的情况。

基于此，我们来思考一个简单的问题：汉堡王向客户承诺说他们的汉堡至少是100g一个，但如果我们要去汉堡王买个汉堡，我们可以预料到它肯定不是标准的100g。设我们的汉堡重量为特征 X_i ，100g就是我们的取值 x_i ，那我买到一个汉堡是100g的概率 $P(100g|Y)$ 是多少呢？如果我们买 n 个汉堡，很可能 n 个汉堡都不一样重，只要我们称重足够精确，100.000001g和100.000002g就可以是不一致的。这种情况下我们可以买无限个汉堡，可能得到无限个重量，可以有无限个基本随机事件的发生，所以我们有：

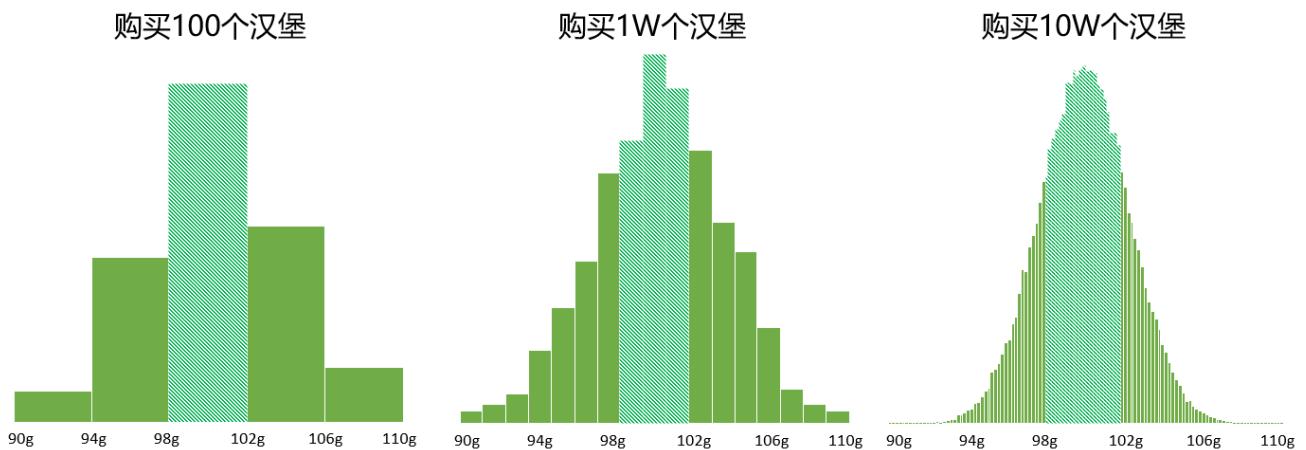
$$P(100g|Y) = \lim_{N \rightarrow \infty} \frac{1}{N} = 0$$

即买到的汉堡刚好是100g概率为0。当一个特征下有无数种可能发生的事件时，这个特征的取值就是连续型的，比如我们现在的特征“汉堡的重量”。从上面的例子可以看得出，**当特征为连续型时，随机取到某一个事件发生的概率就为0。**

那换一个问题，我们随机买一个汉堡，汉堡的重量在98g~102g之间的概率是多少？即是说，我们现在求解概率 $P(98g < x < 102g)$ 。那我们现在随机购买100个汉堡，称重后记下所有重量在98g~102g之间的汉堡个数，假设为m，则就有：

$$P(98g < x < 102g) = \frac{m}{100}$$

如果我们基于100个汉堡绘制直方图，并规定每4g为一个区间，横坐标为汉堡的重量的分布，纵坐标为这个区间上汉堡的个数。



那我们可以看到最左边的图。m就是中间的浅绿色区间中所对应的纵坐标轴。则对于我们的概率，我们可以变换为：

$$P(98g < x < 102g) = \frac{m * 4}{100 * 4} = \frac{\text{浅绿色区间的面积}}{\text{直方图中所有区间的面积}}$$

如果我们购买一万个汉堡并绘制直方图（如中间的图），将直方图上的区间缩小， $P(98g < x < 102g)$ 依然是所有浅绿色区域的面积除以所有柱状图的面积，可以看见现在我们的直方图变得更加平滑，看起来更像一座山了。那假设我们购买10W个，或者无限个汉堡，则我们可以想象我们的直方图最终会变成仿佛一条曲线，而我们的汉堡重量的概率 $P(98g < x < 102g)$ 依然是所有浅绿色柱子的面积除以曲线下所有柱状图的总面积。当购买无数个汉堡的时候形成的则条曲线就叫做概率密度曲线 (probability density function, PDF)。

一条曲线下的面积，就是这条曲线所代表的函数的积分。如果我们定义曲线可以用函数 $f(x)$ 来表示的话，我们整条曲线下的面积就是：

$$\int_{-\infty}^{+\infty} f(x) dx$$

其中 dx 是 $f(x)$ 在 x 上的微分。在某些特定的 $f(x)$ 下，我们可以证明，上述积分等于1。总面积是1，这说明一个连续型特征 X 的取值 x 取到某个区间 $[x_i, x_i + \epsilon]$ 之内的概率就为这个区间上概率密度曲线下的面积，所以我们的特征 X_i 在区间 $[x_i, x_i + \epsilon]$ 中取值的概率可以表示为：

$$\begin{aligned} P(x_i < x < x_i + \epsilon) &= \int_{x_i}^{x_i + \epsilon} f(x) dx \\ &\approx f(x_i) * \epsilon \end{aligned}$$

非常幸运的是，在我们后验概率的计算过程中，我们可以将常量 c 抵消掉，然后我们就可以利用 $f(x_i)$ 的某种变化来估计我们的 $P(x_i|Y)$ 了。现在，我们就将求解连续型变量下某个点取值的概率问题，转化成了求解一个函数 $f(x)$ 在点 x_i 上的取值的问题。那接下来只要找到我们的 $f(x)$ ，我们就可以求解出不同的条件概率了。

在现实中，我们往往假设我们的 $f(x)$ 是满足某种统计学中的分布的，最常见的就是高斯分布（正太分布，像我们购买汉堡的例子），常用的还有伯努利分布，多项式分布。这些分布对应着不同的贝叶斯算法，其实他们的本质都是相同的，只不过他们计算之中的 $f(x)$ 不同。每个 $f(x)$ 都对应着一系列需要我们去估计的参数，因此在贝叶斯中，我们的fit过程其实是在估计对应分布的参数，predict过程是在该参数下的分布中去进行概率预测。

1.3 sklearn中的朴素贝叶斯

Sklearn基于这些分布以及这些分布上的概率估计的改进，为我们提供了四个朴素贝叶斯的分类器。

类	含义
naive_bayes.BernoulliNB	伯努利分布下的朴素贝叶斯
naive_bayes.GaussianNB	高斯分布下的朴素贝叶斯
naive_bayes.MultinomialNB	多项式分布下的朴素贝叶斯
naive_bayes.ComplementNB	补集朴素贝叶斯
linear_model.BayesianRidge	贝叶斯岭回归，在参数估计过程中使用贝叶斯回归技术来包括正则化参数

虽然朴素贝叶斯使用了过于简化的假设，这个分类器在许多实际情况中都运行良好，著名的是文档分类和垃圾邮件过滤。而且由于贝叶斯是从概率角度进行估计，它所需要的样本量比较少，极端情况下甚至我们可以使用1%的数据作为训练集，依然可以得到很好的拟合效果。当然，如果样本量少于特征数目，贝叶斯的效果就会被削弱。

与SVM和随机森林相比，朴素贝叶斯运行速度更快，因为求解 $P(X_i|Y)$ 本质是在每个特征上单独对概率进行计算，然后再求乘积，所以每个特征上的计算可以是独立并且并行的，因此贝叶斯的计算速度比较快。不过相对的，贝叶斯的运行效果不是那么好，所以贝叶斯的接口调用的predict_proba其实也不是总指向真正的分类结果，这一点需要注意。

2 不同分布下的贝叶斯

2.1 高斯朴素贝叶斯 GaussianNB

2.1.1 认识高斯朴素贝叶斯

```
class sklearn.naive_bayes.GaussianNB(priors=None, var_smoothing=1e-09)
```

高斯朴素贝叶斯，通过假设 $P(x_i|Y)$ 是服从高斯分布（也就是正态分布），来估计每个特征下每个类别上的条件概率。对于每个特征下的取值，高斯朴素贝叶斯有如下公式：

$$\begin{aligned}
 P(x_i|Y) &= f(x_i; \mu_y, \sigma_y) * \epsilon \\
 &= \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)
 \end{aligned}$$

对于任意一个Y的取值，贝叶斯都以求解最大化的 $P(x_i|Y)$ 为目标，这样我们才能够比较在不同标签下我们的样本究竟更靠近哪一个取值。以最大化 $P(x_i|Y)$ 为目标，高斯朴素贝叶斯会为我们求解公式中的参数 σ_y 和 μ_y 。求解出参数后，带入一个 x_i 的值，就能够得到一个 $P(x_i|Y)$ 的概率取值。

这个类包含两个参数：

参数	含义
prior	可输入任何类数组结构，形状为 (n_classes,) 表示类的先验概率。如果指定，则不根据数据调整先验，如果不指定，则自行根据数据计算先验概率 $P(Y)$ 。
var_smoothing	浮点数，可不填（默认值= 1e-9） 在估计方差时，为了追求估计的稳定性，将所有特征的方差中最大的方差以某个比例添加到估计的方差中。这个比例，由var_smoothing参数控制。

但在实例化的时候，我们不需要对高斯朴素贝叶斯类输入任何的参数，调用的接口也全部sklearn中比较标准的一些搭配，可以说是一个非常轻量级的类，操作非常容易。但过于简单也意味着贝叶斯没有太多的参数可以调整，因此贝叶斯算法的成长空间并不是太大，如果贝叶斯算法的效果不是太理想，我们一般都会考虑换模型。

无论如何，先来进行一次预测试试看吧：

1. 展示我所使用的设备以及各个库的版本

在这里我们来使用watermask这个便利的模块来帮助我们，这是一个能够帮助我们一行代码查看设备和库的版本的模块。如果没有watermask的你可能需要在cmd中运行pip来安装。也可以直接使用魔法命令%%cmd作为一个cell的开头来帮助我们在jupyter lab中安装你的watermark。

```

%%cmd
pip install watermark

#在这里必须分开cell，魔法命令必须是一个cell的第一部分内容
#注意load_ext这个命令只能够执行一次，再执行就会报错，要求用reload命令
%load_ext watermark

%watermark -a "TsaiTsai" -d -v -m -p numpy,pandas,matplotlib,scipy,sklearn

```

2. 导入需要的库和数据

```

import numpy as np
import matplotlib.pyplot as plt
from sklearn.naive_bayes import GaussianNB
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
X, y = digits.data, digits.target

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

```

3. 建模，探索建模结果

```

gnb = GaussianNB().fit(Xtrain,Ytrain)

#查看分数
acc_score = gnb.score(Xtest,Ytest)

acc_score

#查看预测结果
Y_pred = gnb.predict(Xtest)

#查看预测的概率结果
prob = gnb.predict_proba(Xtest)

prob.shape

prob.shape #每一列对应一个标签下的概率

prob[1,:].sum() #每一行的和都是一

prob.sum(axis=1)

```

4. 使用混淆矩阵来查看贝叶斯的分类结果

```

from sklearn.metrics import confusion_matrix as CM
CM(Ytest,Y_pred)

#注意，ROC曲线是不能用于多分类的。多分类状况下最佳的模型评估指标是混淆矩阵和整体的准确度

```

2.1.2 探索贝叶斯：高斯朴素贝叶斯擅长的数据集

那高斯普斯贝叶斯擅长什么样的数据集呢？我们还是使用常用的三种数据分布：月亮型，环形数据以及二分型数据。注意这段代码曾经在决策树中详细讲解过，在SVM中也有非常类似的代码，核心就是构建分类器然后画决策边界，只不过更换了需要验证的模型而已，因此在这里就不对这段代码是如何实现的进行赘述了。需要了解的小伙伴可以去查看第一章决策树完整版：决策树在合成数据集上的表现，或者查看SVM第一期完整版中，SVC在不同数据集上的表现。

```

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.naive_bayes import GaussianNB

h = .02

names = ["Multinomial", "Gaussian", "Bernoulli", "Complement"]

classifiers = [MultinomialNB(), GaussianNB(), BernoulliNB(), ComplementNB()]

X, y = make_classification(n_features=2, n_redundant=0, n_informative=2,
                           random_state=1, n_clusters_per_class=1)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [make_moons(noise=0.3, random_state=0),
            make_circles(noise=0.2, factor=0.5, random_state=1),
            linearly_separable
            ]

figure = plt.figure(figsize=(6, 9))
i = 1

for ds_index, ds in enumerate(datasets):
    X, y = ds
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.4,
    random_state=42)
    x1_min, x1_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    x2_min, x2_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    array1, array2 = np.meshgrid(np.arange(x1_min, x1_max, 0.2),
                                np.arange(x2_min, x2_max, 0.2))
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])
    ax = plt.subplot(len(datasets), 2, i)
    if ds_index == 0:
        ax.set_title("Input data")
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train,
               cmap=cm_bright, edgecolors='k')
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test,
               cmap=cm_bright, alpha=0.6, edgecolors='k')
    ax.set_xlim(array1.min(), array1.max())
    ax.set_ylim(array2.min(), array2.max())
    ax.set_xticks(())
    ax.set_yticks(())
    i += 1
    ax = plt.subplot(len(datasets), 2, i)

```

```

clf = GaussianNB().fit(X_train, y_train)
score = clf.score(X_test, y_test)

Z = clf.predict_proba(np.c_[array1.ravel(), array2.ravel()])[:, 1]
Z = Z.reshape(array1.shape)
ax.contourf(array1, array2, Z, cmap=cm, alpha=.8)

ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
           edgecolors='k')
ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
           edgecolors='k', alpha=0.6)

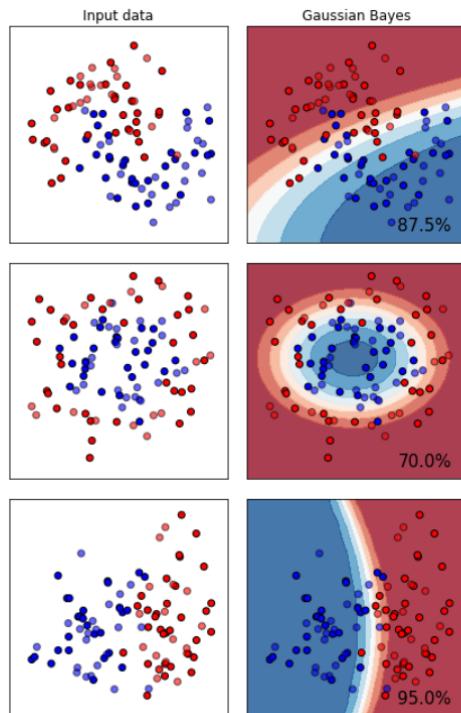
ax.set_xlim(array1.min(), array1.max())
ax.set_ylim(array2.min(), array2.max())
ax.set_xticks(())
ax.set_yticks(())

if ds_index == 0:
    ax.set_title("Gaussian Bayes")

ax.text(array1.max() - .3, array2.min() + .3, ('{:.1f}%'.format(score*100)),
        size=15, horizontalalignment='right')
i += 1

plt.tight_layout()
plt.show()

```



从图上来看，高斯贝叶斯属于比较特殊的一类分类器，其分类效果在二分数据和月亮型数据上表现优秀，但是环形数据不太擅长。我们之前学过的模型中，许多线性模型比如逻辑回归，线性SVM等等，在线性数据集上会绘制直线决策边界，因此难以对月亮型和环形数据进行区分，但高斯朴素贝叶斯的决策边界是曲线，可以是环形也可以是弧线，所以尽管贝叶斯本身更加擅长线性可分的二分数据，但朴素贝叶斯在环形数据和月亮型数据上也可以有远胜过其他线性模型的表现。

2.1.3 探索贝叶斯：高斯朴素贝叶斯的拟合效果与运算速度

我们已经了解高斯朴素贝叶斯属于分类效果不算顶尖的模型，但我们依然好奇，这个算法在拟合的时候还有哪些特性呢？比如说我们了解，决策树是天生过拟合的模型，而支持向量机是不调参数的情况下就非常接近极限的模型。我们希望通过绘制高斯朴素贝叶斯的学习曲线与分类树，随机森林和支持向量机的学习曲线的对比，来探索高斯朴素贝叶斯算法在拟合上的性质。过去绘制学习曲线都是以算法类的某个参数的取值为横坐标，今天我们来使用sklearn中自带的绘制学习曲线的类learning_curve，在这个类中执行交叉验证并从中获得不同样本量下的训练和测试的准确度。

1. 首先导入需要的模块和库

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.tree import DecisionTreeClassifier as DTC
from sklearn.linear_model import LogisticRegression as LR
from sklearn.datasets import load_digits
from sklearn.model_selection import learning_curve
from sklearn.model_selection import ShuffleSplit
from time import time
import datetime
```

2. 定义绘制学习曲线的函数

```
def plot_learning_curve(estimator, title, X, y,
                       ax, #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
                      ):
    train_sizes, train_scores, test_scores = learning_curve(estimator, X, y
                                                            , cv=cv, n_jobs=n_jobs)
    ax.set_title(title)
    if ylim is not None:
        ax.set_ylim(*ylim)
    ax.set_xlabel("Training examples")
    ax.set_ylabel("Score")
    ax.grid() #显示网格作为背景，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
```

3. 导入数据，定义循环

```

digits = load_digits()
X, y = digits.data, digits.target

X.shape

X #是一个稀疏矩阵

title = ["Naive Bayes", "DecisionTree", "SVM, RBF kernel", "RandomForest", "Logistic"]
model = [GaussianNB(), DTC(), SVC(gamma=0.001),
         ,RFC(n_estimators=50), LR(C=.1,solver="lbgf")]
cv = ShuffleSplit(n_splits=50, test_size=0.2, random_state=0)

```

4. 进入循环，绘制学习曲线

```

fig, axes = plt.subplots(1,5, figsize=(30,6))
for ind, title_, estimator in zip(range(len(title)), title, model):
    times = time()
    plot_learning_curve(estimator, title_, X, y,
                         ax=axes[ind], ylim=[0.7, 1.05], n_jobs=4, cv=cv)
    print("{}:{}{}".format(title_, datetime.datetime.fromtimestamp(time() - times).strftime("%M:%S:%F")))
plt.show()

```

三个模型表现出的状态非常有意思。

我们首先返回的结果是**各个算法的运行时间**。可以看到，决策树和贝叶斯不相伯仲（如果你没有发现这个结果，那么可以多运行几次，你会发现贝叶斯和决策树的运行时间逐渐变得差不多）。决策树之所以能够运行非常快速是因为sklearn中的分类树在选择特征时有所“偷懒”，没有计算全部特征的信息熵而是随机选择了一部分特征来进行计算，因此速度快可以理解，但我们也知道决策树的运算效率随着样本量逐渐增大会越来越慢，但朴素贝叶斯却可以在很少的样本上获得不错的结果，因此我们可以预料，随着样本量的逐渐增大贝叶斯会逐渐变得比决策树更快。朴素贝叶斯计算速度远远胜过SVM，随机森林这样复杂的模型，逻辑回归的运行受到最大迭代次数的强烈影响和输入数据的影响（逻辑回归一般在线性数据上运行都比较快，但在这里应该是受到了稀疏矩阵的影响）。因此在运算时间上，朴素贝叶斯还是十分有优势的。

紧接着，我们来看一下每个算法在**训练集上的拟合**。手写数字数据集是一个较为简单的数据集，决策树，森林，SVC和逻辑回归都成功拟合了100%的准确率，但贝叶斯的最高训练准确率都没有超过95%，这也应证了我们最开始说的，朴素贝叶斯的分类效果其实不如其他分类器，贝叶斯天生学习能力比较弱。并且我们注意到，随着训练样本量的逐渐增大，其他模型的训练拟合都保持在100%的水平，但贝叶斯的训练准确率却逐渐下降，这证明样本量越大，贝叶斯需要学习的东西越多，对训练集的拟合程度也越来越差。反而比较少量的样本可以让贝叶斯有较高的训练准确率。

再来看看**过拟合问题**。首先一眼看到，所有模型在样本量很少的时候都是出于过拟合状态的（训练集上表现好，测试集上表现糟糕），但随着样本的逐渐增多，过拟合问题都逐渐消失了，不过每个模型的处理手段不同。比较强大的分类器们，比如SVM，随机森林和逻辑回归，是依靠快速升高模型在测试集上的表现来减轻过拟合问题。相对的，决策树虽然也是通过提高模型在测试集上的表现来减轻过拟合，但随着训练样本的增加，模型在测试集上的表现善生却非常缓慢。朴素贝叶斯独树一帜，是依赖训练集上的准确率下降，测试集上的准确率上升来逐渐解决过拟合问题。

接下来，看看每个算法在**测试集上的拟合结果，即泛化误差的大小**。随着训练样本数量的上升，所有模型的测试表现都上升了，但贝叶斯和决策树在测试集上的表现远远不如SVM，随机森林和逻辑回归。SVM在训练数据量增大到1500个样本左右的时候，测试集上的表现已经非常接近100%，而随机森林和逻辑回归的表现也在95%以上，而决策树和朴素贝叶斯还徘徊在85%左右。但这两个模型所面临的情况十分不同：决策树虽然测试结果不高，但是却依然具

有潜力，因为它的过拟合现象非常严重，我们可以通过减枝来让决策树的测试结果逼近训练结果。然而贝叶斯的过拟合现象在训练样本达到1500左右的时候已经几乎不存在了，训练集上的分数和测试集上的分数非常接近，只有在非常少的时候测试集上的分数才能够比训练集上的结果更高，所以我们基本可以判断，85%左右就是贝叶斯在这个数据集上的极限了。可以预测到，如果我们进行调参，那决策树最后应该可以达到90%左右的预测准确率，但贝叶斯却几乎没有潜力了。

在这个对比之下，我们可以看出：贝叶斯是速度很快，但分类效果一般，并且初次训练之后的结果就很接近算法极限的算法，几乎没有调参的余地。也就是说，如果我们追求对概率的预测，并且希望越准确越好，那我们应该先选择逻辑回归。如果数据十分复杂，或者是稀疏矩阵，那我们坚定地使用贝叶斯。如果我们分类的目标不是要追求对概率的预测，那我们完全可以先试试看高斯朴素贝叶斯的效果（反正它运算很快速，还不需要太多的样本），如果效果很不错，我们就很幸运地得到了一个表现优秀又快速的模型。如果我们没有得到比较好的结果，那我们完全可以选择再更换成更加复杂的模型。

2.2 概率类模型的评估指标

混淆矩阵和精确性可以帮助我们了解贝叶斯的分类结果。然而，我们选择贝叶斯进行分类，大多数时候都不是为了单纯追求效果，而是希望看到预测的相关概率。这种概率给出预测的可信度，所以对于概率类模型，我们希望能够由其他的模型评估指标来帮助我们判断，模型在“概率预测”这项工作上，完成得如何。接下来，我们就来看看概率模型独有的评估指标。

2.2.1 布里尔分数Brier Score

概率预测的准确程度被称为“校准程度”，是衡量算法预测出的概率和真实结果的差异的一种方式。一种比较常用的指标叫做布里尔分数，它被计算为是概率预测相对于测试样本的均方误差，表示为：

$$\text{Brier Score} = \frac{1}{N} \sum_{i=1}^n (p_i - o_i)^2$$

其中N是样本数量， p_i 为朴素贝叶斯预测出的概率， o_i 是样本所对应的真实结果，只能取到0或者1，如果事件发生则为1，如果不发生则为0。这个指标衡量了我们的概率距离真实标签结果的差异，其实看起来非常像是均方误差。**布里尔分数的范围是从0到1，分数越高则预测结果越差劲，校准程度越差，因此布里尔分数越接近0越好。**由于它的本质也是在衡量一种损失，所以在sklearn当中，布里尔得分被命名为brier_score_loss。我们可以从模块metrics中导入这个分数来衡量我们的模型评估结果：

```
from sklearn.metrics import brier_score_loss

#注意，第一个参数是真实标签，第二个参数是预测出的概率值
#在二分类情况下，接口predict_proba会返回两列，但SVC的接口decision_function却只会返回一列
#要随时注意，使用了怎样的概率分类器，以辨别查找置信度的接口，以及这些接口的结构
brier_score_loss(ytest, prob[:,1], pos_label=1)
#我们的pos_label与prob中的索引一致，就可以查看这个类别下的布里尔分数是多少
```

布里尔分数可以用于任何可以使用predict_proba接口调用概率的模型，我们来探索一下在我们的手写数字数据集上，逻辑回归，SVC和我们的高斯朴素贝叶斯的效果如何：

```
from sklearn.metrics import brier_score_loss
brier_score_loss(ytest, prob[:,8], pos_label=8)
```

```

from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression as LR

logi = LR(C=1., solver='lbfgs', max_iter=3000, multi_class="auto").fit(Xtrain, Ytrain)
svc = SVC(kernel = "linear", gamma=1).fit(Xtrain, Ytrain)

brier_score_loss(Ytest, logi.predict_proba(Xtest)[:,1], pos_label=1)

#由于svc的置信度并不是概率，为了可比性，我们需要将svc的置信度“距离”归一化，压缩到[0,1]之间
svc_prob = (svc.decision_function(Xtest) -
            svc.decision_function(Xtest).min())/(svc.decision_function(Xtest).max() -
            svc.decision_function(Xtest).min())

brier_score_loss(Ytest, svc_prob[:,1], pos_label=1)

```

如果将每个分类器每个标签类别下的布里尔分数可视化：

```

import pandas as pd
name = ["Bayes", "Logistic", "SVC"]
color = ["red", "black", "orange"]

df = pd.DataFrame(index=range(10), columns=name)
for i in range(10):
    df.loc[i, name[0]] = brier_score_loss(Ytest, prob[:,i], pos_label=i)
    df.loc[i, name[1]] = brier_score_loss(Ytest, logi.predict_proba(Xtest)[:,i], pos_label=i)
    df.loc[i, name[2]] = brier_score_loss(Ytest, svc_prob[:,i], pos_label=i)
for i in range(df.shape[1]):
    plt.plot(range(10), df.iloc[:,i], c=color[i])
plt.legend()
plt.show()

df

```

可以观察到，逻辑回归的布里尔分数有着压倒性优势，SVC的效果明显弱于贝叶斯和逻辑回归（如同我们之前在SVC的讲解中说明过的一样，SVC是强行利用sigmoid函数来压缩概率，因此SVC产出的概率结果并不那么可靠）。贝叶斯位于逻辑回归和SVC之间，效果也不错，但比起逻辑回归，还是不够精确和稳定。

2.2.2 对数似然函数Log Loss

另一种常用的概率损失衡量是对数损失 (log_loss)，又叫做对数似然，逻辑损失或者交叉熵损失，它是多元逻辑回归以及一些拓展算法，比如神经网络中使用的损失函数。它被定义为，对于一个给定的概率分类器，在预测概率为条件的情况下，真实概率发生的可能性的负对数（如何得到这个损失函数的证明过程和推导过程在逻辑回归的章节中有完整得呈现）。**由于是损失，因此对数似然函数的取值越小，则证明概率估计越准确，模型越理想。**值得注意得是，对数损失只能用于评估分类型模型。

对于一个样本，如果样本的真实标签 y_{true} 在{0,1}中取值，并且这个样本在类别1下的概率估计为 y_{pred} ，则这个样本所对应的对数损失是：

$$-\log P(y_{true}|y_{pred}) = -(y_{true} * \log(y_{pred}) + (1 - y_{true}) * \log(1 - y_{pred}))$$

和我们逻辑回归的损失函数一模一样：

$$J(\theta) = - \sum_{i=1}^m (y_i * \log(y_\theta(x_i)) + (1 - y_i) * \log(1 - y_\theta(x_i)))$$

只不过在逻辑回归的损失函数中，我们的真实标签是由 y_i 表示，预测值（概率估计）是由 $y_\theta(x_i)$ 来表示，仅仅是表示方式的不同。注意，这里的 \log 表示以 e 为底的自然对数。

在sklearn中，我们可以从metrics模块中导入我们的对数似然函数：

```
from sklearn.metrics import log_loss

log_loss(Ytest, prob)
log_loss(Ytest, logi.predict_proba(Xtest))
log_loss(Ytest, svc_prob)
```

第一个参数是真实标签，第二个参数是我们预测的概率。如果我们使用shift tab来查看`log_loss`的参数，会发现第二个参数写着`y_pred`，这会让人误解为这是我们的预测标签。由于`log_loss`是专门用于产出概率的算法的，因此它假设我们预测出的`y`就是以概率形式呈现，但在sklearn当中，我们的`y_pred`往往是已经根据概率归类后的类别{0, 1, 2}，真正的概率必须要以接口`predict_proba`来调用，千万避免混淆。

注意到，**我们用`log_loss`得出的结论和我们使用布里尔分数得出的结论不一致**：当使用布里尔分数作为评判标准的时候，SVC的估计效果是最差的，逻辑回归和贝叶斯的结果相接近。而使用对数似然的时候，虽然依然是逻辑回归最强，但贝叶斯却没有SVC的效果好。为什么会有这样的不同呢？

因为逻辑回归和SVC都是以最优化为目的来求解模型，然后进行分类的算法。而朴素贝叶斯中，却没有最优化的过程。对数似然函数直接指向模型最优化的方向，甚至就是逻辑回归的损失函数本身，因此在逻辑回归和SVC上表现得更好。

那什么时候使用对数似然，什么时候使用布里尔分数？

在现实应用中，对数似然函数是概率类模型评估的黄金指标，往往是我们评估概率类模型的优先选择。但是它也有一些缺点，首先它没有界，不像布里尔分数有上限，可以作为模型效果的参考。其次，它的解释性不如布里尔分数，很难与非技术人员去交流对数似然存在的可靠性和必要性。第三，它在以最优化为目标的模型上明显表现更好。而且，它还有一些数学上的问题，比如不能接受为0或1的概率，否则的话对数似然就会取到极限值（考虑以 e 为底的自然对数在取到0或1的时候的情况）。所以因此通常来说，我们有以下使用规则：

需求	优先使用对数似然	优先使用布里尔分数
衡量模型	要对比多个模型，或者衡量模型的不同变化	衡量单一模型的表现
可解释性	机器学习和深度学习之间的行家交流，学术论文	商业报告，老板开会，业务模型的衡量
最优化指向	逻辑回归，SVC	朴素贝叶斯
数学问题	概率只能无限接近于0或1，无法取到0或1	概率可以取到0或1，比如树，随机森林

回到我们的贝叶斯来看，如果贝叶斯的模型效果不如其他模型，而我们又不想更换模型，那怎么办呢？如果以精确度为指标来调整参数，贝叶斯估计是无法拯救了——不同于SVC和逻辑回归，贝叶斯的原理简单，根本没有什么可用的参数。但是产出概率的算法有自己的调节方式，就是**调节概率的校准程度**。校准程度越高，模型对概率的预测越准确，算法在做判断时就越有自信，模型就会更稳定。如果我们追求模型在概率预测上必须尽量贴近真实概率，那我们就可以使用可靠性曲线来调节概率的校准程度。

2.2.3 可靠性曲线Reliability Curve

可靠性曲线 (reliability curve) , 又叫做概率校准曲线 (probability calibration curve) , 可靠性图 (reliability diagrams) , 这是一条以预测概率为横坐标, 真实标签为纵坐标的曲线。我们希望预测概率和真实值越接近越好, 最好两者相等, 因此一个模型/算法的概率校准曲线越靠近对角线越好。校准曲线因此也是我们的模型评估指标之一。和布里尔分数相似, 概率校准曲线是对于标签的某一类来说的, 因此一类标签就会有一条曲线, 或者我们可以使用一个多类标签下的平均来表示整个模型的概率校准曲线。但通常来说, 曲线用于二分类的情况最多, 大家如果感兴趣可以自行探索多分类的情况。

根据这个思路, 我们来绘制一条曲线试试看。

1. 导入需要的库和模块

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification as mc
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression as LR
from sklearn.metrics import brier_score_loss
from sklearn.model_selection import train_test_split
```

2. 创建数据集

```
X, y = mc(n_samples=100000, n_features=20 #总共20个特征
           ,n_classes=2 #标签为2分类
           ,n_informative=2 #其中两个代表较多信息
           ,n_redundant=10 #10个都是冗余特征
           ,random_state=42)

#样本量足够大, 因此使用1%的样本作为训练集
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, y
                                                ,test_size=0.99
                                                ,random_state=42)
Xtrain

np.unique(Ytrain)
```

3. 建立模型, 绘制图像

```
gnb = GaussianNB()
gnb.fit(Xtrain,Ytrain)
y_pred = gnb.predict(Xtest)
prob_pos = gnb.predict_proba(Xtest)[:,1] #我们的预测概率 - 横坐标
#Ytest - 我们的真实标签 - 纵坐标

#在我们的横纵表坐标上, 概率是由顺序的 (由小到大), 为了让图形规整一些, 我们要先对预测概率和真实标签按照预测概率进行一个排序, 这一点我们通过DataFrame来实现

df = pd.DataFrame({"ytrue":Ytest[:500], "probability":prob_pos[:500]})
```

```

df

df = df.sort_values(by="probability")
df.index = range(df.shape[0])

df

#紧接着我们就可以画图了
fig = plt.figure()
ax1 = plt.subplot()
ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated") #得做一条对角线来对比呀
ax1.plot(df["probability"],df["ytrue"], "s-", label="%s (%1.3f)" % ("Bayes", clf_score))
ax1.set_ylabel("True label")
ax1.set_xlabel("predcited probability")
ax1.set_ylim([-0.05, 1.05])
ax1.legend()
plt.show()

```

这个图像看起来非常可怕，完全不之所云！为什么存在这么多上下穿梭的直线？反应快的小伙伴可能很快就发现了，我们是按照预测概率的顺序进行排序的，而预测概率从0开始到1的过程中，真实取值不断在0和1之间变化，而我们是绘制折线图，因此无数个纵坐标分布在0和1的被链接起来了，所以看起来如此混乱。

那我们换成散点图来试试看呢？

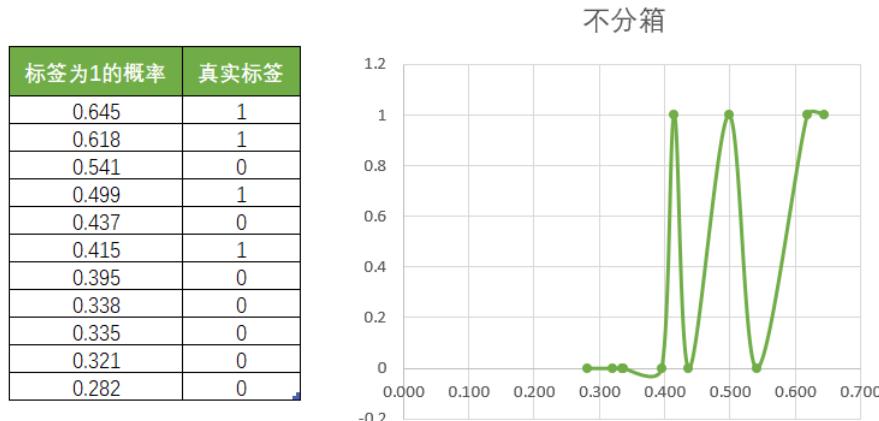
```

fig = plt.figure()
ax1 = plt.subplot()
ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")
ax1.scatter(df["probability"],df["ytrue"], s=10)
ax1.set_ylabel("True label")
ax1.set_xlabel("predcited probability")
ax1.set_ylim([-0.05, 1.05])
ax1.legend()
plt.show()

```

可以看到，由于真实标签是0和1，所以所有的点都在 $y=1$ 和 $y=0$ 这两条直线上分布，这完全不是我们希望看到的图像。回想一下我们的可靠性曲线的横纵坐标：横坐标是预测概率，而纵坐标是真实值，我们希望预测概率很靠近真实值，那我们的真实取值必然也需要是一个概率才可以，如果使用真实标签，那我们绘制出来的图像完全是没有意义的。但是，我们去哪里寻找真实值的概率呢？这是不可能找到的——如果我们能够找到真实的概率，那我们何必还用算法来估计概率呢，直接去获取真实的概率不就好了么？所以真实概率在现实中是不可获得的。但是，我们可以获得类概率的指标来帮助我们进行校准。一个简单的做法是，将数据进行分箱，然后规定每个箱子中真实的少数类所占的比例为这个箱上的真实概率trueproba，这个箱子中预测概率的均值为这个箱子的预测概率predproba，然后以trueproba为纵坐标，predproba为横坐标，来绘制我们的可靠性曲线。

举个例子，来看下面这张表，这是一组数据不分箱时表现出来的图像：



再来看看分箱之后的图像：



可见，分箱之后样本点的特征被聚合到了一起，曲线明显变得单调且平滑。这种分箱操作本质相当于是一种平滑，在sklearn中，这样的做法可以通过绘制可靠性曲线的类**calibration_curve**来实现。和ROC曲线类似，类**calibration_curve**可以帮助我们获取我们的横纵坐标，然后使用matplotlib来绘制图像。该类有如下参数：

参数	含义
y_true	真实标签
y_prob	预测返回的，正类别下的概率值或置信度
normalize	布尔值，默认False 是否将y_prob中输入的内容归一化到[0,1]之间，比如说，当y_prob并不是真正的概率的时候可以使用。如果这是为True，则会将y_prob中最小的值归一化为0，最大值归一化为1。
n_bins	整数值，表示分箱的个数。如果箱数很大，则需要更多的数据。
返回	含义
trueproba	可靠性曲线的纵坐标，结构为(n_bins,)，是每个箱子中少数类(Y=1)的占比
predproba	可靠性曲线的横坐标，结构为(n_bins,)，是每个箱子中概率的均值

4. 使用可靠性曲线的类在贝叶斯上绘制一条校准曲线

```
from sklearn.calibration import calibration_curve

#从类calibration_curve中获取横坐标和纵坐标
trueproba, predproba = calibration_curve(Ytest, prob_pos
                                            ,n_bins=10 #输入希望分箱的个数
                                            )

fig = plt.figure()
ax1 = plt.subplot()
ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")
ax1.plot(predproba, trueproba,"s-",label="%s (%1.3f)" % ("Bayes", clf_score))
ax1.set_ylabel("True probability for class 1")
ax1.set_xlabel("Mean predicted probability")
ax1.set_yticks([-0.05, 1.05])
ax1.legend()
plt.show()
```

5. 不同的n_bins取值下曲线如何改变?

```
fig, axes = plt.subplots(1,3,figsize=(18,4))
for ind,i in enumerate([3,10,100]):
    ax = axes[ind]
    ax.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")
    trueproba, predproba = calibration_curve(Ytest, prob_pos,n_bins=i)
    ax.plot(predproba, trueproba,"s-",label="n_bins = {}".format(i))
    ax.set_ylabel("True probability for class 1")
    ax.set_xlabel("Mean predicted probability")
    ax.set_yticks([-0.05, 1.05])
    ax.legend()
plt.show()
```

很明显可以看出, n_bins越大, 箱子越多, 概率校准曲线就越精确, 但是太过精确的曲线不够平滑, 无法和我们希望的完美概率密度曲线相比较。n_bins越小, 箱子越少, 概率校准曲线就越粗糙, 虽然靠近完美概率密度曲线, 但是无法真实地展现模型概率预测地结果。因此我们需要取一个既不是太大, 也不是太小的箱子个数, 让概率校准曲线既不是太精确, 也不是太粗糙, 而是一条相对平滑, 又可以反应出模型对概率预测的趋势的曲线。通常来说, 建议先试试看箱子数等于10的情况。箱子的数目越大, 所需要的样本量也越多, 否则曲线就会太过精确。

6. 建立更多模型

```
name = ["GaussianBayes", "Logistic", "svc"]

gnb = GaussianNB()
logi = LogisticRegression(C=1., solver='lbfgs', max_iter=3000, multi_class="auto")
svc = SVC(kernel = "linear", gamma=1)
```

7. 建立循环, 绘制多个模型的概率校准曲线

```
fig, ax1 = plt.subplots(figsize=(8,6))
ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")
```

```

for clf, name_ in zip([gnb, logi, svc], name):
    clf.fit(Xtrain, Ytrain)
    y_pred = clf.predict(Xtest)
    #hasattr(obj, name): 查看一个类obj中是否存在名字为name的接口, 存在则返回True
    if hasattr(clf, "predict_proba"):
        prob_pos = clf.predict_proba(Xtest)[:, 1]
    else: # use decision function
        prob_pos = clf.decision_function(Xtest)
        prob_pos = (prob_pos - prob_pos.min()) / (prob_pos.max() - prob_pos.min())
    #返回布里尔分数
    clf_score = brier_score_loss(Ytest, prob_pos, pos_label=y.max())
    trueproba, predproba = calibration_curve(Ytest, prob_pos, n_bins=10)
    ax1.plot(predproba, trueproba, "s-", label="%s (%1.3f)" % (name_, clf_score))

ax1.set_ylabel("True probability for class 1")
ax1.set_xlabel("Mean predicted probability")
ax1.set_ylim([-0.05, 1.05])
ax1.legend()
ax1.set_title('calibration plots (reliability curve)')
plt.show()

```

从图像的结果来看，我们可以明显看出，逻辑回归的概率估计是最接近完美的概率校准曲线，所以逻辑回归的效果最完美。相对的，高斯朴素贝叶斯和支持向量机分类器的结果都比较糟糕。支持向量机呈现类似于sigmoid函数的形状，而高斯朴素贝叶斯呈现和Sigmoid函数相反的形状。

对于贝叶斯，如果概率校准曲线呈现sigmoid函数的镜像的情况，则说明数据集中的特征不是相互条件独立的。贝叶斯原理中的“朴素”原则：特征相互条件独立原则被违反了（这其实是我们自己的设定，我们设定了10个冗余特征，这些特征就是噪音，他们之间不可能完全独立），因此贝叶斯的表现不够好。

而支持向量机的概率校准曲线效果其实是典型的置信度不足的分类器(under-confident classifier)的表现：**大量的样本点集中在决策边界的附近，因此许多样本点的置信度靠近0.5左右，即便决策边界能够将样本点判断正确，模型本身对这个结果也不是非常确信的。**相对的，离决策边界很远的点的置信度就会很高，因为它很大可能性上不会被判断错误。支持向量机在面对混合度较高的数据的时候，有着天生的置信度不足的缺点。

2.2.4 预测概率的直方图

我们可以通过绘制直方图来查看模型的预测概率的分布。直方图是以样本的预测概率分箱后的结果为横坐标，每个箱中的样本数量为纵坐标的一个图像。注意，这里的分箱和我们在可靠性曲线中的分箱不同，这里的分箱是将预测概率均匀分为一个个的区间，与之前可靠性曲线中为了平滑的分箱完全是两码事。我们来绘制一下我们的直方图：

```

fig, ax2 = plt.subplots(figsize=(8, 6))

for clf, name_ in zip([gnb, logi, svc], name):
    clf.fit(Xtrain, Ytrain)
    y_pred = clf.predict(Xtest)
    #hasattr(obj, name): 查看一个类obj中是否存在名字为name的接口, 存在则返回True
    if hasattr(clf, "predict_proba"):
        prob_pos = clf.predict_proba(Xtest)[:, 1]
    else: # use decision function
        prob_pos = clf.decision_function(Xtest)

```

```

prob_pos = (prob_pos - prob_pos.min()) / (prob_pos.max() - prob_pos.min())
ax2.hist(prob_pos
         ,bins=10
         ,label=name_
         ,histtype="step" #设置直方图为透明
         ,lw=2 #设置直方图每个柱子描边的粗细
         )

ax2.set_ylabel("Distribution of probability")
ax2.set_xlabel("Mean predicted probability")
ax2.set_xlim([-0.05, 1.05])
ax2.set_xticks([0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1])
ax2.legend(loc=9)
plt.show()

```

可以看到，高斯贝叶斯的概率分布是两边非常高，中间非常低，几乎90%以上的样本都在0和1的附近，可以说是置信度最高的算法，但是贝叶斯的布里尔分数却不如逻辑回归，这证明贝叶斯中在0和1附近的样本中有一部分是被分错的。支持向量贝叶斯完全相反，明显是中间高，两边低，类似于正态分布的状况，证明了我们刚才所说的，大部分样本都在决策边界附近，置信度都徘徊在0.5左右的情况。而逻辑回归位于高斯朴素贝叶斯和支持向量机的中间，即没有太多的样本过度靠近0和1，也没有形成像支持向量机那样的正态分布。一个比较健康的正样本的概率分布，就是逻辑回归的直方图显示出来的样子。

避免混淆：概率密度曲线和概率分布直方图

大家也许还记得我们说过，我们是假设样本的概率分布为高斯分布，然后使用高斯的方程来估计连续型变量的概率。怎么现在我们绘制出的概率分布结果中，高斯普斯贝叶斯的概率分布反而完全不是高斯分布了呢？

注意，千万不要把概率密度曲线和概率分布直方图混淆。

在称重汉堡的时候所绘制的曲线，是概率密度曲线，横坐标是样本的取值，纵坐标是落在这个样本取值区间中的样本个数，衡量的是每个X的取值区间之内有多少样本。服从高斯分布的是X的取值上的样本分布。

现在我们的概率分布直方图，横坐标是概率的取值[0,1]，纵坐标是落在这个概率取值范围中的样本的个数，衡量的是每个概率取值区间之内有多少样本。这个分布，是没有任何假设的。

我们已经得知了朴素贝叶斯和SVC预测概率的效果各方面都不如逻辑回归，那在这种情况下，我们如何来帮助模型或者算法，让他们对自己的预测更有信心，置信度更高呢？我们可以使用等近似回归来矫正概率算法。

2.2.5 校准可靠性曲线

等近似回归有两种回归可以使用，一种是基于Platt的Sigmoid模型的参数校准方法，一种是基于等渗回归 (isotonic calibration) 的非参数的校准方法。概率校准应该发生在测试集上，必须是模型未曾见过的数据。在数学上，使用这两种方式来对概率进行校准的原理十分复杂，而此过程我们在sklearn中无法进行干涉，大家不必过于去深究，如果希望深入研究利用回归校准概率的细节，可以查看sklearn中的如下案例。这是一个基于鸢尾花数据集的，三分类数据上的概率校准过程，大家如果感兴趣可以仔细研究：

https://scikit-learn.org/stable/auto_examples/calibration/plot_calibration_multiclass.html#sphx-glr-auto-examples-calibration-plot-calibration-multiclass-py

在这里，我主要来为大家展示如果使用sklearn中的概率校正类CalibratedClassifierCV来对二分类情况下的数据集进行概率校正。

```
class sklearn.calibration.CalibratedClassifierCV(base_estimator=None, method='sigmoid', cv='warn')
```

这是一个带交叉验证的概率校准类，它使用交叉验证生成器，对交叉验证中的每一份数据，它都在训练样本上进行模型参数估计，在测试样本上进行概率校准，然后为我们返回最佳的一组参数估计和校准结果。每一份数据的预测概率会被求解平均。**注意，类CalibratedClassifierCV没有接口decision_function，要查看这个类下校准过后的模型生成的概率，必须调用predict_proba接口。**

base_estimator

需要校准其输出决策功能的分类器，必须存在predict_proba或decision_function接口。如果参数cv = prefit，分类器必须已经拟合数据完毕。

cv

整数，确定交叉验证的策略。可能输入是：

- None，表示使用默认的3折交叉验证
- 任意整数，指定折数

对于输入整数和None的情况下来说，如果时二分类，则自动使用类sklearn.model_selection.StratifiedKFold进行折数分割。如果y是连续型变量，则使用sklearn.model_selection.KFold进行分割。

- 已经使用其他类建好的交叉验证模式或生成器cv
- 可迭代的，已经分割完毕的测试集和训练集索引数组
- 输入"prefit"，则假设已经在分类器上拟合完毕数据。在这种模式下，使用者必须手动确定用来拟合分类器的数据与即将倍校准的数据没有交集

在版本0.20中更改：在0.22版本中输入"None"，将由使用3折交叉验证改为5折交叉验证

method

进行概率校准的方法，可输入"sigmoid"或者"isotonic"

- 输入'sigmoid'，使用基于Platt的Sigmoid模型来进行校准
- 输入'isotonic'，使用等渗回归来进行校准

当校准的样本量太少（比如，小于等于1000个测试样本）的时候，不建议使用等渗回归，因为它倾向于过拟合。样本量过少时请使用sigmoids，即Platt校准。

我们依然来使用之前建立的数据集。

1. 包装函数

首先，我们将之前绘制可靠性曲线和直方图的代码包装成函数。考虑函数的参数为：模型，模型的名字，数据，和需要分箱的个数。我们在这里将直方图和可靠性曲线打包在同一个函数中，让他们并排显示。

```
def plot_calib(models, name, Xtrain, Xtest, Ytrain, Ytest, n_bins=10):  
  
    import matplotlib.pyplot as plt  
    from sklearn.metrics import brier_score_loss  
    from sklearn.calibration import calibration_curve
```

```

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 6))
ax1.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")

for clf, name_ in zip(models, name):
    clf.fit(Xtrain, Ytrain)
    y_pred = clf.predict(Xtest)
    #hasattr(obj, name): 查看一个类obj中是否存在名字为name的接口，存在则返回True
    if hasattr(clf, "predict_proba"):
        prob_pos = clf.predict_proba(Xtest)[:, 1]
    else: # use decision function
        prob_pos = clf.decision_function(Xtest)
        prob_pos = (prob_pos - prob_pos.min()) / (prob_pos.max() - prob_pos.min())
    #返回布里尔分数
    clf_score = brier_score_loss(Ytest, prob_pos, pos_label=y.max())
    trueproba, predproba = calibration_curve(Ytest, prob_pos, n_bins=n_bins)
    ax1.plot(predproba, trueproba, "s-", label="%s (%1.3f)" % (name_, clf_score))
    ax2.hist(prob_pos, range=(0, 1), bins=n_bins, label=name_, histtype="step", lw=2)

ax2.set_ylabel("Distribution of probability")
ax2.set_xlabel("Mean predicted probability")
ax2.set_xlim([-0.05, 1.05])
ax2.legend(loc=9)
ax2.set_title("Distribution of probability")
ax1.set_ylabel("True probability for class 1")
ax1.set_xlabel("Mean predicted probability")
ax1.set_ylim([-0.05, 1.05])
ax1.legend()
ax1.set_title('Calibration plots(reliability curve)')
plt.show()

```

2. 设实例化模型，设定模型的名字

```

from sklearn.calibration import CalibratedClassifierCV

name = ["GaussianBayes", "Logistic", "Bayes+isotonic", "Bayes+sigmoid"]

gnb = GaussianNB()

models = [gnb
          , LR(C=1., solver='lbfgs', max_iter=3000, multi_class="auto")
          #定义两种校准方式
          , CalibratedClassifierCV(gnb, cv=2, method='isotonic')
          , CalibratedClassifierCV(gnb, cv=2, method='sigmoid')]

```

3. 基于函数进行绘图

```
plot_calib(models, name, Xtrain, Xtest, Ytrain, Ytest)
```

从校正朴素贝叶斯的结果来看，Isotonic等校正大大改善了曲线的形状，几乎让贝叶斯的效果与逻辑回归持平，并且布里尔分数也下降到了0.098，比逻辑回归还低一个点。Sigmoid校准的方式也对曲线进行了稍稍的改善，不过效果不明显。从直方图来看，Isotonic校正让高斯朴素贝叶斯的效果接近逻辑回归，而Sigmoid校正后的结果依然和原本的高斯朴素贝叶斯更相近。可见，当数据的特征之间不是相互条件独立的时候，使用Isotonic方式来校准概率曲线，可以得到不错的结果，让模型在预测上更加谦虚。

4. 基于校准结果查看精确性的变化

```
gnb = GaussianNB().fit(Xtrain, Ytrain)
gnb.score(Xtest, Ytest)

brier_score_loss(Ytest, gnb.predict_proba(Xtest)[:, 1], pos_label = 1)

gnbisotonic = CalibratedClassifierCV(gnb, cv=2, method='isotonic').fit(Xtrain, Ytrain)
gnbisotonic.score(Xtest, Ytest)

brier_score_loss(Ytest, gnbisotonic.predict_proba(Xtest)[:, 1], pos_label = 1)
```

可以看出，校准概率后，布里尔分数明显变小了，但整体的准确率却略有下降，这证明算法在校准之后，尽管对概率的预测更准确了，但模型的判断力略有降低。来思考一下：布里尔分数衡量模型概率预测的准确率，布里尔分数越低，代表模型的概率越接近真实概率，当进行概率校准后，本来标签是1的样本的概率应该会更接近1，而标签本来是0的样本应该会更接近0，没有理由布里尔分数提升了，模型的判断准确率居然下降了。但从我们的结果来看，模型的准确率和概率预测的正确性并不是完全一致的，为什么会这样呢？

对于不同的概率类模型，原因是不同的。对于SVC，决策树这样的模型来说，概率不是真正的概率，而更偏向于是一个“置信度”，这些模型也不是依赖于概率预测来进行分类（决策树依赖于树权而SVC依赖于决策边界），因此对于这些模型，可能存在着类别1下的概率为0.4但样本依然被分类为1的情况，这种情况代表着——模型很没有信心认为这个样本是1，但是还是坚持把这个样本的标签分类为1了。这种时候，概率校准可能会向着更加错误的方向调整（比如把概率为0.4的点调节得更接近0，导致模型最终判断错误），因此出现布里尔分数可能会显示和精确性相反的趋势。

而对于朴素贝叶斯这样的模型，却是另一种情况。注意在朴素贝叶斯中，我们有各种各样的假设，除了我们的“朴素”假设，还有我们对概率分布的假设（比如说高斯），这些假设使得我们的贝叶斯得出的概率估计其实是有偏估计，也就是说，这种概率估计其实不是那么准确和严肃。我们通过校准，让模型的预测概率更贴近于真实概率，本质是在统计学上让算法更加贴近我们对整体样本状况的估计，这样的一种校准在一组数据集上可能表现出让准确率上升，也可能表现出让准确率下降，这取决于我们的测试集有多贴近我们估计的真实样本的面貌。这一系列有偏估计使得我们在概率校准中可能出现布里尔分数和准确度的趋势相反的情况。

当然，可能还有更多更深层的原因，比如概率校准过程中的数学细节如何影响了我们的校准，类calibration_curve中是如何分箱，如何通过真实标签和预测值来生成校准曲线使用的横纵坐标的，这些过程中也可能有着让布里尔分数和准确率向两个方向移动的过程。

在现实中，**当两者相悖的时候，请务必以准确率为标准**。但是这不代表说布里尔分数和概率校准曲线就无效了。概率类模型几乎没有参数可以调整，除了换模型之外，鲜有更好的方式帮助我们提升模型的表现，概率校准是难得的可以帮助我们针对概率提升模型的方法。

5. 试试看对于SVC，哪种校准更有效呢？

```

name_svc = ["SVC", "Logistic", "SVC+isotonic", "SVC+sigmoid"]

svc = SVC(kernel = "linear", gamma=1)

models_svc = [svc
    ,LR(C=1., solver='lbfgs', max_iter=3000, multi_class="auto")
    #依然定义两种校准方式
    ,CalibratedClassifierCV(svc, cv=2, method='isotonic')
    ,CalibratedClassifierCV(svc, cv=2, method='sigmoid')]

plot_calib(models_svc, name_svc, Xtrain, Xtest, Ytrain, Ytest)

```

可以看出，对于SVC, sigmoid和isotonic的校准效果都非常不错，无论是从校准曲线来看还是从概率分布图来看，两种校准都让SVC的结果接近逻辑回归，其中sigmoid更加有效。来看看不同的SVC下的精确度结果（对于这一段代码，大家完全可以把它包括在原有的绘图函数中）：

```

name_svc = ["SVC", "SVC+isotonic", "SVC+sigmoid"]

svc = SVC(kernel = "linear", gamma=1)

models_svc = [svc
    ,CalibratedClassifierCV(svc, cv=2, method='isotonic')
    ,CalibratedClassifierCV(svc, cv=2, method='sigmoid')]

for clf, name in zip(models_svc, name_svc):
    clf.fit(Xtrain, Ytrain)
    y_pred = clf.predict(Xtest)
    if hasattr(clf, "predict_proba"):
        prob_pos = clf.predict_proba(Xtest)[:, 1]
    else:
        prob_pos = clf.decision_function(Xtest)
        prob_pos = (prob_pos - prob_pos.min()) / (prob_pos.max() - prob_pos.min())
    clf_score = brier_score_loss(Ytest, prob_pos, pos_label=y.max())
    score = clf.score(Xtest, Ytest)
    print("{}:{}".format(name))
    print("\tBrier:{}:.4f".format(clf_score))
    print("\tAccuracy:{}:.4f".format(score))

```

可以看到，对于SVC来说，两种校正都改善了准确率和布里尔分数。可见，概率校正对于SVC非常有效。这也说明，**概率校正对于原本的可靠性曲线是形容Sigmoid形状的曲线的算法比较有效。**

在现实中，我们可以选择调节模型的方向，我们不一定要追求最高的准确率或者追求概率拟合最好，我们可以根据自己的需求来调整模型。当然，对于概率类模型来说，由于可以调节的参数甚少，所以我们更倾向于追求概率拟合，并使用概率校准的方式来调节模型。如果你的确希望追求更高的准确率和Recall，可以考虑使用天生就非常准确的概率类模型逻辑回归，也可以考虑使用除了概率校准之外还有很多其他参数可调的支持向量机分类器。

2.3 多项式朴素贝叶斯以及其变化

2.3.1 多项式朴素贝叶斯MultinomialNB

多项式贝叶斯可能是除了高斯之外，最为人所知的贝叶斯算法了。它也是基于原始的贝叶斯理论，但假设概率分布是服从一个简单多项式分布。多项式分布来源于统计学中的多项式实验，这种实验可以具体解释为：实验包括n次重复试验，每项试验都有不同的可能结果。在任何给定的试验中，特定结果发生的概率是不变的。

举个例子，比如说一个特征矩阵 X 表示投掷硬币的结果，则得到正面的概率为 $P(X = \text{正面} | Y) = 0.5$ ，得到反面的概率为 $P(X = \text{反面} | Y) = 0.5$ ，只有这两种可能并且两种结果互不干涉，并且两个随机事件的概率加和为1，这就是一个二项分布。这种情况下，适合于多项式朴素贝叶斯的特征矩阵应该长这样：

测试编号	$X_1: \text{出现正面}$	$X_2: \text{出现反面}$
0	0	1
1	1	0
2	1	0
3	0	1

假设另一个特征 X' 表示投掷骰子的结果，则 i 就可以在[1,2,3,4,5,6]中取值，六种结果互不干涉，且只要样本量足够大，概率都为 $1/6$ ，这就是一个多项分布。多项分布的特征矩阵应该长这样：

测试编号	出现1	出现2	出现3	出现4	出现5	出现6
0	1	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	1	0	0	0
...						
m	0	0	0	0	0	1

可以看出：

1. **多项式分布擅长的是分类型变量**，在其原理假设中， $P(x_i | Y)$ 的概率是离散的，并且不同 x_i 下的 $P(x_i | Y)$ 相互独立，互不影响。虽然sklearn中的多项式分布也可以处理连续型变量，但现实中，如果我们真的想要处理连续型变量，我们应当使用高斯朴素贝叶斯。
2. 多项式实验中的实验结果都很具体，它所涉及的特征往往是次数，频率，计数，出现与否这样的概念，这些概念都是离散的正整数，因此**sklearn中的多项式朴素贝叶斯不接受负值的输入**。

由于这样的特性，多项式朴素贝叶斯的特征矩阵经常是稀疏矩阵（不一定总是稀疏矩阵），并且它经常被用于文本分类。我们可以使用著名的**TF-IDF向量技术**，也可以使用常见并且简单的**单词计数向量**手段与贝叶斯配合使用。这两种手段都属于常见的文本特征提取的方法，可以很简单地通过sklearn来实现，在案例中我们会来详细讲到。

从数学的角度来看，在一种标签类别 $Y = c$ 下，我们有一组分别对应特征的参数向量 $\theta_c = (\theta_{c1}, \theta_{c2}, \dots, \theta_{cn})$ ，其中n表示特征的总数。一个 θ_{ci} 表示这个标签类别下的第*i*个特征所对应的参数。这个参数被我们定义为：

$$\theta_{ci} = \frac{\text{特征 } X_i \text{ 在 } Y = c \text{ 这个分类下的所有样本的取值总和}}{\text{所有特征在 } Y = c \text{ 这个分类下的所有样本的取值总和}}$$

记作 $P(X_i|Y = c)$, 表示当 $Y=c$ 这个条件固定的时候, 一组样本在 X_i 这个特征上的取值被取到的概率。注意, 我们在高斯朴素贝叶斯中求解的概率 $P(x_i|Y)$ 是对于一个样本来说, 而我们现在求解的 $P(X_i|Y = c)$ 是对于一个特征 X_i 来说的概率。

对于一个在标签类别 $Y = c$ 下, 结构为 (m, n) 的特征矩阵来说, 我们有:

$$\mathbf{X}_y = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ x_{31} & x_{32} & x_{33} & \dots & x_{3n} \\ \vdots & & & & \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix}$$

其中每个 x_{ji} 都是特征 X_i 发生的次数。基于这些理解, 我们通过平滑后的最大似然估计来求解参数 θ_y :

$$\theta_{ci} = \frac{\sum_{y_j=c} x_{ji} + \alpha}{\sum_{i=1}^n \sum_{y_j=c} x_{ji} + \alpha n}$$

对于每个特征, $\sum_{y_j=c} x_{ji}$ 是特征 X_i 下所有标签为c的样本的特征取值之和, 其实就是特征矩阵中每一列的和。 $\sum_{i=1}^n \sum_{y_j=c} x_{ji}$ 是所有标签类别为c的样本上, 所有特征的取值之和, 其实就是特征矩阵 \mathbf{X}_y 中所有元素的和。 α 被称为平滑系数, 我们令 $\alpha > 0$ 来防止训练数据中出现过的一些词汇没有出现在测试集中导致的0概率, 以避免让参数 θ 为0的情况。如果我们将 α 设置为1, 则这个平滑叫做拉普拉斯平滑, 如果 α 小于1, 则我们把它叫做利德斯通平滑。两种平滑都属于自然语言处理中比较常用的用来平滑分类数据的统计手段。

在sklearn中, 用来执行多项式朴素贝叶斯的类MultinomialNB包含如下的参数和属性:

```
class sklearn.naive_bayes.MultinomialNB(alpha=1.0, fit_prior=True, class_prior=None)
```

参数

alpha: 浮点数, 可不填 (默认为1.0)

拉普拉斯或利德斯通平滑的参数 α , 如果设置为0则表示完全没有平滑选项。但是需要注意的是, 平滑相当于人为给概率加上一些噪音, 因此 α 设置得越大, 多项式朴素贝叶斯的精确性会越低 (虽然影响不是非常大), 布里尔分数也会逐渐升高。

fit_prior: 布尔值, 可不填 (默认为True)

是否学习先验概率 $P(Y = c)$ 。如果设置为false, 则不使用先验概率, 而使用统一先验概率 (uniform prior), 即认为每个标签类出现的概率是 $\frac{1}{n_classes}$ 。

class_prior: 形似数组的结构, 结构为 $(n_classes,)$, 可不填 (默认为None)

类的先验概率 $P(Y = c)$ 。如果没有给出具体的先验概率则自动根据数据来进行计算。

通常我们在实例化多项式朴素贝叶斯的时候, 我们会让所有的参数保持默认。先来简单建一个多项式朴素贝叶斯的例子试试看:

1. 导入需要的模块和库

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_blobs
from sklearn.metrics import brier_score_loss
```

2. 建立数据集

```
class_1 = 500
class_2 = 500 #两个类别分别设定500个样本
centers = [[0.0, 0.0], [2.0, 2.0]] #设定两个类别的中心
clusters_std = [0.5, 0.5] #设定两个类别的方差
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 = train_test_split(X,y
                                                , test_size=0.3
                                                , random_state=420)
```

3. 归一化，确保输入的矩阵不带有负数

```
#先归一化，保证输入多项式朴素贝叶斯的特征矩阵中不带有负数
mms = MinMaxScaler().fit(Xtrain)
Xtrain_ = mms.transform(Xtrain)
Xtest_ = mms.transform(Xtest)
```

4. 建立一个多项式朴素贝叶斯分类器吧

```
mnb = MultinomialNB().fit(Xtrain_, Ytrain)

#重要属性：调用根据数据获取的，每个标签类的对数先验概率log(P(Y))
#由于概率永远是在[0,1]之间，因此对数先验概率返回的永远是负值
mnb.class_log_prior_

np.unique(Ytrain)

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

mnb.class_log_prior_.shape

#可以使用np.exp来查看真正的概率值
np.exp(mnb.class_log_prior_)

#重要属性：返回一个固定标签类别下的每个特征的对数概率log(P(xi|y))
mnb.feature_log_prob_

mnb.feature_log_prob_.shape
```

```
#重要属性：在fit时每个标签类别下包含的样本数。当fit接口中的sample_weight被设置时，该接口返回的值也会受到加权的影响
mnb.class_count_
mnb.class_count_.shape
```

5. 那分类器的效果如何呢？

```
#一些传统的接口
mnb.predict(xtest_)

mnb.predict_proba(xtest_)

mnb.score(xtest_, Ytest)

brier_score_loss(Ytest, mnb.predict_proba(xtest_)[:, 1], pos_label=1)
```

7. 效果不太理想，思考一下多项式贝叶斯的性质，我们能够做点什么呢？

```
#来试试看把xtrain转换成分类型数据吧
#注意我们的xtrain没有经过归一化，因为做哑变量之后自然所有的数据就不会有负数了
from sklearn.preprocessing import KBinsDiscretizer
kbs = KBinsDiscretizer(n_bins=10, encode='onehot').fit(xtrain)
xtrain_ = kbs.transform(xtrain)
xtest_ = kbs.transform(xtest)

mnb = MultinomialNB().fit(xtrain_, Ytrain)

mnb.score(xtest_, Ytest)

brier_score_loss(Ytest, mnb.predict_proba(xtest_)[:, 1], pos_label=1)
```

可以看出，多项式朴素贝叶斯的基本操作和代码都非常简单。同样的数据，如果采用哑变量方式的分箱处理，多项式贝叶斯的效果会突飞猛进。作为在文本分类中大放异彩的算法，我们将会在案例中来详细讲解多项式贝叶斯的使用，并为大家介绍文本分类的更多细节。

2.3.2 伯努利朴素贝叶斯BernoulliNB

多项式朴素贝叶斯可同时处理二项分布（抛硬币）和多项分布（掷骰子），其中二项分布又叫做伯努利分布，它是一种现实中常见，并且拥有很多优越数学性质的分布。因此，既然有着多项式朴素贝叶斯，我们自然也就又专门用来处理二项分布的朴素贝叶斯：伯努利朴素贝叶斯。

伯努利贝叶斯类BernoulliNB假设数据服从多元伯努利分布，并在此基础上应用朴素贝叶斯的训练和分类过程。多元伯努利分布简单来说，就是数据集中可以存在多个特征，但每个特征都是二分类的，可以以布尔变量表示，也可以表示为{0, 1}或者{-1, 1}等任意二分类组合。因此，这个类要求将样本转换为二分类特征向量，如果数据本身不是二分类的，那可以使用类中专门用来二值化的参数binarize来改变数据。

伯努利朴素贝叶斯与多项式朴素贝叶斯非常相似，都常用于处理文本分类数据。但由于伯努利朴素贝叶斯是处理二项分布，所以它更加在意的是“存在与否”，而不是“出现多少次”这样的次数或频率，这是伯努利贝叶斯与多项式贝叶斯的根本性不同。在文本分类的情况下，伯努利朴素贝叶斯可以使用单词出现向量（而不是单词计数向量）来训练分类器。文档较短的数据集上，伯努利朴素贝叶斯的效果会更好。如果时间允许，建议两种模型都试试看。

来看看伯努利朴素贝叶斯类的参数：

```
class sklearn.naive_bayes.BernoulliNB(alpha=1.0, binarize=0.0, fit_prior=True, class_prior=None)
```

伯努利朴素贝叶斯

alpha : 浮点数, 可不填 (默认为1.0)

拉普拉斯或利德斯通平滑的参数 α , 如果设置为0则表示完全没有平滑选项。但是需要注意的是，平滑相当于人为给概率加上一些噪音，因此 α 设置得越大，多项式朴素贝叶斯的精确性会越低（虽然影响不是非常大），布里尔分数也会逐渐升高。

binarize : 浮点数或None, 可不填, 默认为0

将特征二值化的阈值, 如果设定为None, 则会假定说特征已经被二值化完毕

fit_prior : 布尔值, 可不填 (默认为True)

是否学习先验概率 $P(Y = c)$ 。如果设置为false, 则不使用先验概率, 而使用统一先验概率 (uniform prior) , 即认为每个标签类出现的概率是 $\frac{1}{n_classes}$ 。

class_prior: 形似数组的结构, 结构为($n_classes$,), 可不填 (默认为None)

类的先验概率 $P(Y = c)$ 。如果没有给出具体的先验概率则自动根据数据来进行计算。

在sklearn中，伯努利朴素贝叶斯的实现也非常简单：

```
from sklearn.naive_bayes import BernoulliNB

#普通来说我们应该使用二值化的类sklearn.preprocessing.Binarizer来将特征一个个二值化
#然而这样效率过低, 因此我们选择归一化之后直接设置一个阈值

mms = MinMaxScaler().fit(xtrain)
xtrain_ = mms.transform(xtrain)
xtest_ = mms.transform(xtest)

#不设置二值化
bnl_ = BernoulliNB().fit(xtrain_, ytrain)
bnl_.score(xtest_, ytest)
brier_score_loss(ytest, bnl_.predict_proba(xtest_)[:, 1], pos_label=1)

#设置二值化阈值为0.5
bnl = BernoulliNB(binarize=0.5).fit(xtrain_, ytrain)
bnl.score(xtest_, ytest)
brier_score_loss(ytest, bnl.predict_proba(xtest_)[:, 1], pos_label=1)
```

和多项式贝叶斯一样，伯努利贝叶斯的结果也受到数据结构非常大的影响。因此，根据数据的模样选择贝叶斯，是贝叶斯模型选择中十分重要的一点。

2.3.3 探索贝叶斯：贝叶斯的样本不均衡问题

接下来，我们来探讨一个分类算法永远都逃不过的核心问题：样本不平衡。贝叶斯由于分类效力不算太好，因此对样本不平衡极为敏感，我们接下来就来看一看样本不平衡如何影响了贝叶斯。

1. 导入需要的模块，建立样本不平衡的数据集

```
from sklearn.naive_bayes import MultinomialNB, GaussianNB, BernoulliNB
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_blobs
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.metrics import brier_score_loss as BS, recall_score, roc_auc_score as AUC

class_1 = 50000 #多数类为50000个样本
class_2 = 500 #少数类为500个样本
centers = [[0.0, 0.0], [5.0, 5.0]] #设定两个类别的中心
clusters_std = [3, 1] #设定两个类别的方差
X, y = make_blobs(n_samples=[class_1, class_2],
                   centers=centers,
                   cluster_std=clusters_std,
                   random_state=0, shuffle=False)
X.shape

np.unique(y)
```

2. 查看所有贝叶斯在样本不平衡数据集上的表现

```
name = ["Multinomial", "Gaussian", "Bernoulli"]
models = [MultinomialNB(), GaussianNB(), BernoulliNB()]

for name,clf in zip(name,models):
    Xtrain, Xtest, Ytrain, Ytest = train_test_split(X,y
                                                    , test_size=0.3
                                                    , random_state=420)
    if name != "Gaussian":
        kbs = KBinsDiscretizer(n_bins=10, encode='onehot').fit(Xtrain)
        Xtrain = kbs.transform(Xtrain)
        Xtest = kbs.transform(Xtest)

    clf.fit(Xtrain,Ytrain)
    y_pred = clf.predict(Xtest)
    proba = clf.predict_proba(Xtest)[:,1]
    score = clf.score(Xtest,Ytest)
    print(name)
    print("\tBrier:{:.3f}".format(BS(Ytest,proba, pos_label=1)))
    print("\tAccuracy:{:.3f}".format(score))
    print("\tRecall:{:.3f}".format(recall_score(Ytest,y_pred)))
    print("\tAUC:{:.3f}".format(AUC(Ytest,proba)))
```

从结果上来看，多项式朴素贝叶斯判断出了所有的多数类样本，但放弃了全部的少数类样本，受到样本不均衡问题影响最严重。高斯比多项式在少数类的判断上更加成功一些，至少得到了43.8%的recall。伯努利贝叶斯虽然整体的准确度和布里尔分数不如多项式和高斯朴素贝叶斯和，但至少成功捕捉到了77.1%的少数类。可见，伯努利贝叶斯最能够忍受样本不均衡问题。

可是，伯努利贝叶斯只能用于处理二项分布数据，在现实中，强行将所有的数据都二值化不会永远得到好结果，在我们有多个特征的时候，我们更需要一个个去判断究竟二值化的阈值该取多少才能够让算法的效果优秀。这样做无疑是非常低效的。那如果我们的目标是捕捉少数类，我们应该怎么办呢？高斯朴素贝叶斯的效果虽然比多项式好，但是也没有好到可以用来帮助我们捕捉少数类的程度——43.8%，还不如抛硬币的结果。因此，孜孜不倦的统计学家们改进了朴素贝叶斯算法，修正了包括无法处理样本不平衡在内的传统朴素贝叶斯的众多缺点，得到了新兴贝叶斯算法：补集朴素贝叶斯。

2.3.4 改进多项式朴素贝叶斯：补集朴素贝叶斯ComplementNB

补集朴素贝叶斯 (complement naive Bayes, CNB) 算法是标准多项式朴素贝叶斯算法的改进。CNB的发明小组创造出CNB的初衷是为了解决贝叶斯中的“朴素”假设带来的各种问题，他们希望能够创造出数学方法以逃避朴素贝叶斯中的朴素假设，让算法能够不去关心所有特征之间是否是条件独立的。以此为基础，他们创造出了能够解决样本不平衡问题，并且能够一定程度上忽略朴素假设的补集朴素贝叶斯。在实验中，CNB的参数估计已经被证明比普通多项式朴素贝叶斯更稳定，并且它特别适合于样本不平衡的数据集。有时候，CNB在文本分类任务上的表现有时能够优于多项式朴素贝叶斯，因此现在补集朴素贝叶斯也开始逐渐流行。

关于补集朴素贝叶斯具体是如何逃避了我们的朴素假设，或者如何让我们的样本不均衡问题得到了改善，背后有深刻数学原理和复杂的数学证明过程，大家如果感兴趣可以参阅这篇论文：

- Rennie, J. D., Shih, L., Teevan, J., & Karger, D. R. (2003). [Tackling the poor assumptions of naive bayes text classifiers](#). In ICML (Vol. 3, pp. 616-623).

简单来说，CNB使用来自每个标签类别的补集的概率，并以此来计算每个特征的权重。

$$\hat{\theta}_{i,y \neq c} = \frac{\alpha_i + \sum_{y_j \neq c} x_{ij}}{\alpha_i n + \sum_{i,y \neq c} \sum_{i=1}^n x_{ij}}$$

其中 j 表示每个样本， x_{ij} 表示在样本 j 上对于特征 i 的下的取值，在文本分类中通常是计数的值或者是TF-IDF值。 α 是像标准多项式朴素贝叶斯中一样的平滑系数。可以看出，这个看似复杂的公式其实很简单， $\sum_{y_j \neq c} x_{ij}$ 其实指的就是，一个特征 i 下，所有标签类别不等于 c 值的样本的特征取值之和。而 $\sum_{i,y \neq c} \sum_{i=1}^n x_{ij}$ 其实指的就是，所有特征下，素有标签类别不等于 c 值得样本的特征取值之和。其实就是多项式分布的逆向思路。

$$w_{ci} = \log \hat{\theta}_{i,y \neq c}$$

或者我们可以选择：

$$w_{ci} = \frac{\log \hat{\theta}_{i,y \neq c}}{\sum_j |\log \hat{\theta}_{i,y \neq c}|}$$

对于这个概率，我们对它取对数后得到权重。我们还可以选择除以它的L2范式，以解决了在多项式分布中，特征取值比较多的样本（比如说比较长的文档）支配参数估计的情况。很多时候我们的特征矩阵是稀疏矩阵，但也不排除在有一些随机事件中，可以一次在两个特征中取值的情况。如果一个样本下的很多个随机事件可以同时发生，并且互不干涉，那么这个样本上可能有很多个特征下都有取值——文本分类中就有很多这样的情况。比如说我们可以有如下特征矩阵：

索引	X1	X2
0	1	1
1	0	1

这种状况下，索引为0的样本就会在参数估计中占更多的权重。

更甚，如果一个样本下的很多个随机事件同时发生，还在一次实验中发生了多次，那这个样本在参数估计中也会占有更大的权重。

索引	X1	X2
0	5	1
1	0	1

基于这个权重，补充朴素贝叶斯中一个样本的预测规则为：

$$p(Y \neq c | \mathbf{X}) = \arg \min_c \sum_i x_i w_{ci}$$

即我们求解出的最小补集概率所对应的标签就是样本的标签，因为 $Y \neq c$ 的概率越小，则意味着 $Y = c$ 的概率越大，所以样本属于标签类别c。

在sklearn中，补集朴素贝叶斯由类ComplementNB完成，它包含的参数和多项式贝叶斯也非常相似：

```
class sklearn.naive_bayes.ComplementNB(alpha=1.0, fit_prior=True, class_prior=None, norm=False)
```

补集朴素贝叶斯

alpha : 浮点数, 可不填 (默认为1.0)

拉普拉斯或利德斯通平滑的参数 α ，如果设置为0则表示完全没有平滑选项。但是需要注意的是，平滑相当于人为给概率加上一些噪音，因此 α 设置得越大，多项式朴素贝叶斯的精确性会越低（虽然影响不是非常大），布里尔分数也会逐渐升高。

norm : 布尔值, 可不填, 默认False

在计算权重的时候是否适用L2范式来规范权重的大小。默认不进行规范，即不跟从补集朴素贝叶斯算法的全部内容，如果希望进行规范，请设置为True。

fit_prior : 布尔值, 可不填 (默认为True)

是否学习先验概率 $P(Y = c)$ 。如果设置为false，则不使用先验概率，而使用统一先验概率 (uniform prior)，即认为每个标签类出现的概率是 $\frac{1}{n_classes}$ 。

class_prior: 形似数组的结构, 结构为(n_classes,), 可不填 (默认为None)

类的先验概率 $P(Y = c)$ 。如果没有给出具体的先验概率则自动根据数据来进行计算。

那来看看，补集朴素贝叶斯在不平衡样本上的表现吧，同时我们来计算一下每种贝叶斯的计算速度：

```
from sklearn.naive_bayes import ComplementNB
from time import time
import datetime
```

```

name = ["Multinomial", "Gaussian", "Bernoulli", "Complement"]
models = [MultinomialNB(), GaussianNB(), BernoulliNB(), ComplementNB()]

for name,clf in zip(name,models):
    times = time()
    Xtrain, Xtest, Ytrain, Ytest = train_test_split(x,y
                                                    , test_size=0.3
                                                    , random_state=420)
    #预处理
    if name != "Gaussian":
        kbs = KBinsDiscretizer(n_bins=10, encode='onehot').fit(Xtrain)
        Xtrain = kbs.transform(Xtrain)
        Xtest = kbs.transform(Xtest)

    clf.fit(Xtrain,Ytrain)
    y_pred = clf.predict(Xtest)
    proba = clf.predict_proba(Xtest)[:,1]
    score = clf.score(Xtest,Ytest)
    print(name)
    print("\tBrier:{:.3f}".format(BS(Ytest,proba,pos_label=1)))
    print("\tAccuracy:{:.3f}".format(score))
    print("\tRecall:{:.3f}".format(recall_score(Ytest,y_pred)))
    print("\tAUC:{:.3f}".format(AUC(Ytest,proba)))
    print(datetime.datetime.fromtimestamp(time()-times).strftime("%M:%S:%f"))

```

可以发现，补集朴素贝叶斯牺牲了部分整体的精确度和布里尔指数，但是得到了十分高的召回率Recall，捕捉出了98.7%的少数类，并且在此基础上维持了和原本的多项式朴素贝叶斯一致的AUC分数。和其他的贝叶斯算法比起来，我们的补集朴素贝叶斯的运行速度也十分优秀。如果我们的目标是捕捉少数类，那我们毫无疑问会希望选择补集朴素贝叶斯作为我们的算法。

3 案例：贝叶斯分类器做文本分类

文本分类是现代机器学习应用中的一大模块，更是自然语言处理的基础之一。我们可以通过将文字数据处理成数字数据，然后使用贝叶斯来帮助我们判断一段话，或者一篇文章中的主题分类，感情倾向，甚至文章体裁。现在，绝大多数社交媒体数据的自动化采集，都是依靠首先将文本编码成数字，然后按分类结果采集需要的信息。虽然现在自然语言处理领域大部分由深度学习所控制，贝叶斯分类器依然是文本分类中的一颗明珠。现在，我们就来学习一下，贝叶斯分类器是怎样实现文本分类的。

3.1 文本编码技术简介

3.1.1 单词计数向量

在开始分类之前，我们必须先将文本编码成数字。一种常用的方法是单词计数向量。在这种技术中，一个样本可以包含一段话或一篇文章，这个样本中如果出现了10个单词，就会有10个特征($n=10$)，每个特征 X_i 代表一个单词，特征的取值 x_i 表示这个单词在这个样本中总共出现了几次，**是一个离散的，代表次数的，正整数**。

在sklearn当中，单词计数向量计数可以通过feature_extraction.text模块中的CountVectorizer类实现，来看一个简单的例子：

```

sample = ["Machine learning is fascinating, it is wonderful"
          , "Machine Learning is a sensational technology"
          , "Elsa is a popular character"]

from sklearn.feature_extraction.text import CountVectorizer

vec = CountVectorizer()

X = vec.fit_transform(sample)

# 使用接口get_feature_names()调用每个列的名称

import pandas as pd
# 注意稀疏矩阵是无法输入pandas的
cvresult = pd.DataFrame(X.toarray(), columns = vec.get_feature_names())

cvresult

```

从这个编码结果，我们可以发现两个问题。

首先，来回忆一下我们多项式朴素贝叶斯的计算公式：

$$\theta_{ci} = \frac{\sum_{y_j=c} x_{ji} + \alpha}{\sum_{i=1}^n \sum_{y_j=c} x_{ji} + \alpha n}$$

如果我们将每一列加和，除以整个特征矩阵的和，就是每一列对应的概率 θ_i 。由于是将 x_{ji} 进行加和，对于一个在很多个特征下都有值的样本来说，这个样本在对 θ_{ci} 的贡献就会比其他的样本更大。对于句子特别长的样本而言，这个样本对 θ_i 的影响是巨大的。因此补集朴素贝叶斯让每个特征的权重除以自己的L2范式，就是为了避免这种情况发生。

第二个问题，观察我们的矩阵，会发现"is"这个单词出现了四次，那经过计算，这个单词出现的概率就会最大，但其实它对我们的语义并没有什么影响（除非我们希望判断的是，文章描述的是过去的事件还是现在发生的事件）。可以遇见，如果使用单词计数向量，可能会导致一部分常用词（比如中文中的“的”）频繁出现在我们的矩阵中并且占有很高的权重，对分类来说，这明显是对算法的一种误导。为了解决这个问题，比起使用次数，我们使用单词在句子中所占的比例来编码我们的单词，这就是我们著名的TF-IDF方法。

3.1.2 TF-IDF

TF-IDF全称term frequency-inverse document frequency，词频逆文档频率，是通过单词在文档中出现的频率来衡量其权重，也就是说，IDF的大小与一个词的常见程度成反比，这个词越常见，编码后为它设置的权重会倾向于越小，以此来压制频繁出现的一些无意义的词。在sklearn当中，我们使用feature_extraction.text中类TfidfVectorizer来执行这种编码。

```

from sklearn.feature_extraction.text import TfidfVectorizer as TFIDF
vec = TFIDF()

X = vec.fit_transform(sample)

X

```

```
#同样使用接口get_feature_names()调用每个列的名称
TFIDFresult = pd.DataFrame(X.toarray(),columns=vec.get_feature_names())

TFIDFresult

#使用TF-IDF编码之后，出现得多的单词的权重被降低了么？

CVresult.sum(axis=0)/CVresult.sum(axis=0).sum()

TFIDFresult.sum(axis=0) / TFIDFresult.sum(axis=0).sum()
```

在之后的例子中，我们都会使用TF-IDF的编码方式。

3.2 探索文本数据

在现实中，文本数据的处理是十分耗时耗力的，尤其是不规则的长文本的处理方式，绝对不是一两句话能够说明白的，因此在这里我们将使用的数据集是sklearn中自带的文本数据集fetch_20newsgroup。这个数据集是20个网络新闻组的语料库，其中包含约2万篇新闻，全部以英文显示，如果大家希望使用中文则处理过程会更加困难，会需要自己加载中文的语料库。在这个例子中，主要目的是为大家展示贝叶斯的用法和效果，因此我们就使用英文的语料库。

```
from sklearn.datasets import fetch_20newsgroups

#初次使用这个数据集的时候，会在实例化的时候开始下载
data = fetch_20newsgroups()

#通常我们使用data来查看data里面到底包含了什么内容，但由于fetch_20newsgroups这个类加载出的数据巨大，数据结构中混杂很多文字，因此很难去看清

#不同类型的新闻
data.target_names

#其实fetch_20newsgroups也是一个类，既然是类，应该就有可以调用的参数
#面对简单数据集，我们往往在实例化的过程中什么都不写，但是现在data中数据量太多，不方便探索
#因此我们需要来看看我们的类fetch_20newsgroups都有什么样的参数可以帮助我们
```

```
sklearn.datasets.fetch_20newsgroups(data_home=None, subset='train', categories=None, shuffle=True,
random_state=42, remove=(), download_if_missing=True)
```

在这之中，我们来认识几个比较重要的参数：

fetch_20newsgroups 参数列表**subset** : 选择类中包含的数据子集

输入"train"表示选择训练集, "test"表示输入测试集, "all"表示加载所有的数据

categories : 可输入None或者数据所在的目录

选择一个子集下, 不同类型或不同内容的数据所在的目录。如果不输入默认None, 则会加载全部的目录。

download_if_missing: 可选, 默认是True

如果发现本地数据不全, 是否自动进行下载

shuffle : 布尔值, 可不填, 表示是否打乱样本顺序

对于假设样本之间互相独立并且服从相同分布的算法或模型(比如随机梯度下降)来说可能很重要。

现在我们就可以直接通过参数来提取我们希望得到的数据集了。

```

import numpy as np
import pandas as pd

categories = ["sci.space" #科学技术 - 太空
              , "rec.sport.hockey" #运动 - 曲棍球
              , "talk.politics.guns" #政治 - 枪支问题
              , "talk.politics.mideast"] #政治 - 中东问题

train = fetch_20newsgroups(subset="train", categories = categories)
test = fetch_20newsgroups(subset="test", categories = categories)

train
#可以观察到, 里面依然是类字典结构, 我们可以通过使用键的方式来提取内容

train.target_names

#查看总共有多少篇文章存在
len(train.data)

#随意提取一篇文章来看看
train.data[0]

#查看一下我们的标签
np.unique(train.target)

len(train.target)

#是否存在样本不平衡问题?
for i in [1,2,3]:
    print(i,(train.target == i).sum()/len(train.target))

```

3.3 使用TF-IDF将文本数据编码

```
from sklearn.feature_extraction.text import TfidfVectorizer as TFIDF
```

```

Xtrain = train.data
Xtest = test.data
Ytrain = train.target
Ytest = test.target

tfidf = Tfidf().fit(Xtrain)
Xtrain_ = tfidf.transform(Xtrain)
Xtest_ = tfidf.transform(Xtest)

Xtrain_

tosee = pd.DataFrame(Xtrain_.toarray(), columns=tfidf.get_feature_names())

tosee.head()

tosee.shape

```

3.4 在贝叶斯上分别建模，查看结果

```

from sklearn.naive_bayes import MultinomialNB, ComplementNB, BernoulliNB
from sklearn.metrics import brier_score_loss as BS

name = ["Multinomial", "Complement", "Bournulli"]
#注意高斯朴素贝叶斯不接受稀疏矩阵
models = [MultinomialNB(), ComplementNB(), BernoulliNB()]

for name,clf in zip(name,models):
    clf.fit(Xtrain_,Ytrain)
    y_pred = clf.predict(Xtest_)
    proba = clf.predict_proba(Xtest_)
    score = clf.score(Xtest_,Ytest)
    print(name)

    #4个不同的标签取值下的布里尔分数
    Bscore = []
    for i in range(len(np.unique(Ytrain))):
        bs = BS(Ytest,proba[:,i],pos_label=i)
        Bscore.append(bs)
        print("\tBrier under {}: {:.3f}".format(train.target_names[i],bs))

    print("\tAverage Brier:{:.3f}".format(np.mean(Bscore)))
    print("\tAccuracy:{:.3f}".format(score))
    print("\n")

```

从结果上来看，两种贝叶斯的效果都很不错。虽然补集贝叶斯的布里尔分数更高，但它的精确度更高。我们可以使用概率校准来试试看能否让模型进一步突破：

```
from sklearn.calibration import CalibratedClassifierCV
```

```

name = ["Multinomial"
        , "Multinomial + Isotonic"
        , "Multinomial + Sigmoid"
        , "Complement"
        , "Complement + Isotonic"
        , "Complement + Sigmoid"
        , "Bernoulli"
        , "Bernoulli + Isotonic"
        , "Bernoulli + Sigmoid"]

models = [MultinomialNB()
          , CalibratedClassifierCV(MultinomialNB(), cv=2, method='isotonic')
          , CalibratedClassifierCV(MultinomialNB(), cv=2, method='sigmoid')
          , ComplementNB()
          , CalibratedClassifierCV(ComplementNB(), cv=2, method='isotonic')
          , CalibratedClassifierCV(ComplementNB(), cv=2, method='sigmoid')
          , BernoulliNB()
          , CalibratedClassifierCV(BernoulliNB(), cv=2, method='isotonic')
          , CalibratedClassifierCV(BernoulliNB(), cv=2, method='sigmoid')
      ]

for name,clf in zip(name,models):
    clf.fit(Xtrain_,Ytrain)
    y_pred = clf.predict(xtest_)
    proba = clf.predict_proba(xtest_)
    score = clf.score(xtest_,Ytest)
    print(name)
    Bscore = []
    for i in range(len(np.unique(Ytrain))):
        bs = BS(Ytest,proba[:,i],pos_label=i)
        Bscore.append(bs)
        print("\tBrier under {}: {:.3f}".format(train.target_names[i],bs))
    print("\tAverage Brier:{:.3f}".format(np.mean(Bscore)))
    print("\tAccuracy:{:.3f}".format(score))
    print("\n")

```

可以观察到，多项式分布下无论如何调整，算法的效果都不如补集朴素贝叶斯来得好。因此我们在分类的时候，应该选择补集朴素贝叶斯。对于补集朴素贝叶斯来说，使用Sigmoid进行概率校准的模型综合最优秀：准确率最高，对数损失和布里尔分数都在0.1以下，可以说是非常理想的模型了。

对于机器学习而言，朴素贝叶斯也许不是最常用的分类算法，但作为概率预测算法中唯一一个真正依赖概率来进行计算，并且简单快捷的算法，朴素贝叶斯还是常常被人们提起。并且，朴素贝叶斯在文本分类上的效果的确非常优秀。由此可见，只要我们能够提供足够的数据，合理利用高维数据进行训练，朴素贝叶斯就可以为我们提供意想不到的效果。

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

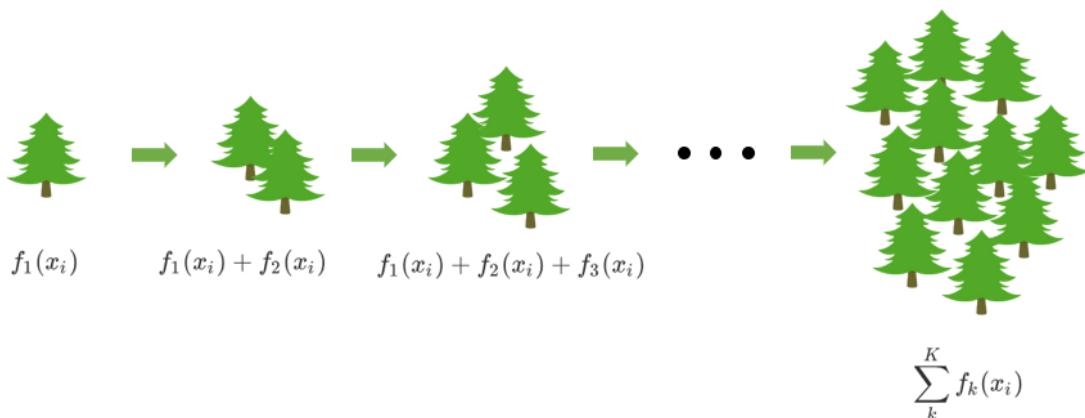
```
class xgboost.XGBRegressor(max_depth=3, learning_rate=0.1, n_estimators=100, silent=True,
objective='reg:linear', booster='gbtree', n_jobs=1, nthread=None, gamma=0, min_child_weight=1, max_delta_step=0,
subsample=1, colsample_bytree=1, colsample_bylevel=1, reg_alpha=0, reg_lambda=1, scale_pos_weight=1,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

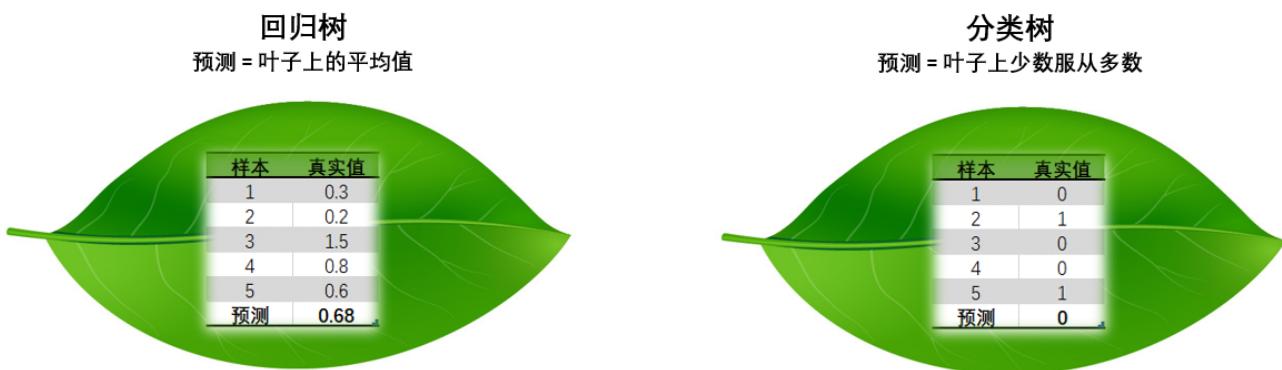
XGBoost的基础是梯度提升算法，因此我们必须先从了解梯度提升算法开始。梯度提升（Gradient boosting）是构建预测模型的最强大技术之一，它是集成算法中提升法（Boosting）的代表算法。**集成算法通过在数据上构建多个弱评估器，汇总所有弱评估器的建模结果，以获取比单个模型更好的回归或分类表现。**弱评估器被定义为是表现至少比随机猜测更好的模型，即预测准确率不低于50%的任意模型。

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

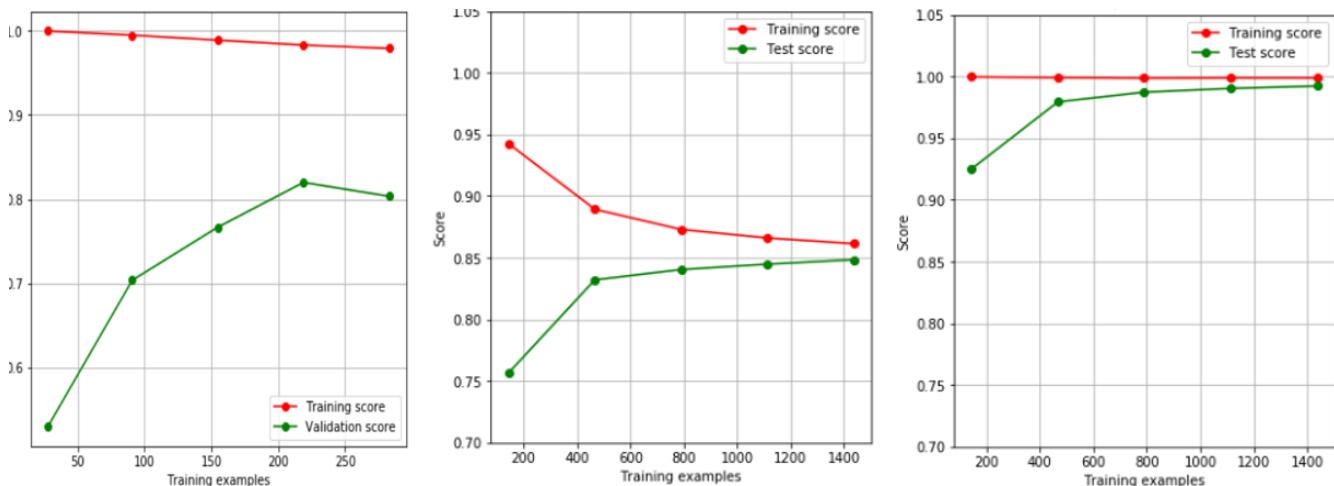
```

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。

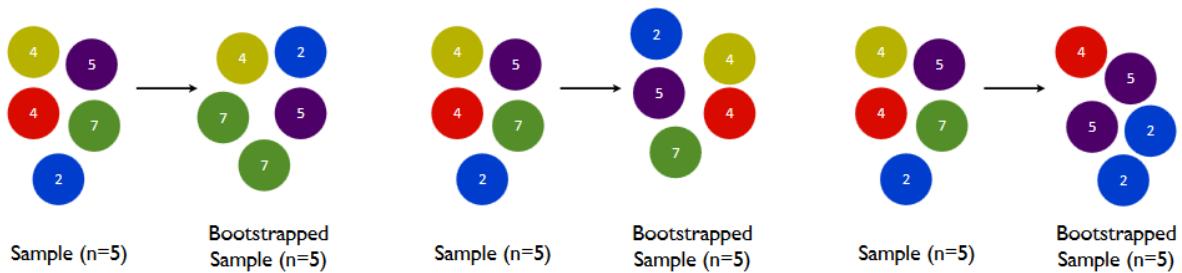
第三，树的数量提升对模型的影响有极限，最开始，模型的表现会随着XGB的树的数量一起提升，但到达某个点之后，树的数量越多，模型的效果会逐步下降，这也说明了暴力增加n_estimators不一定有效果。

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

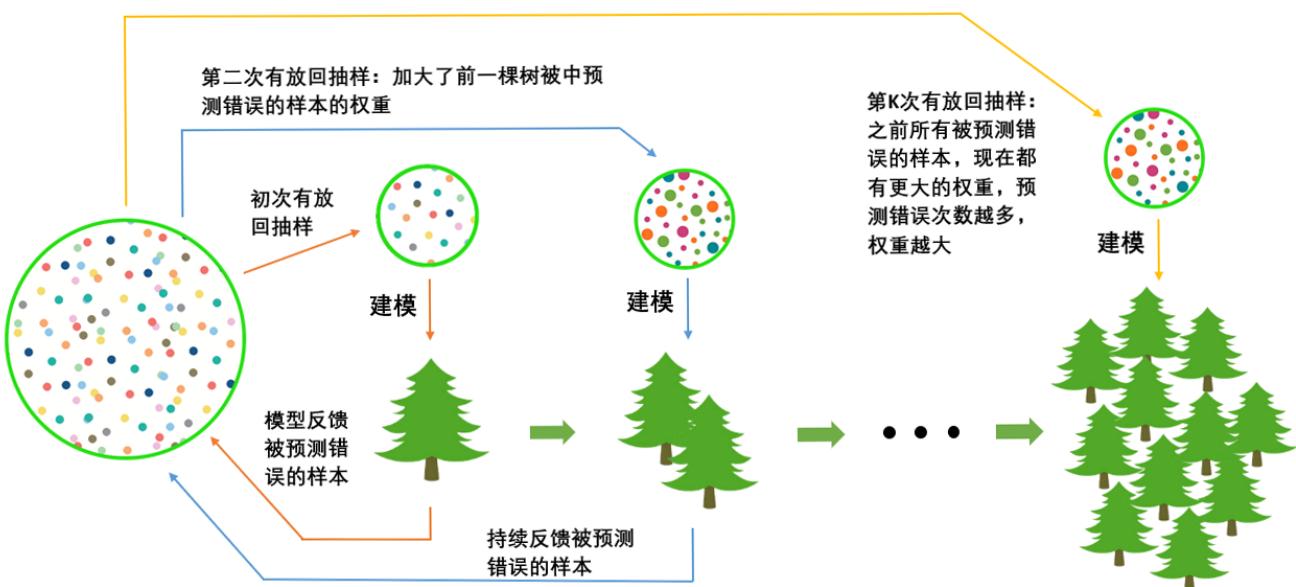
确认了有多少棵树之后，我们来思考一个问题：建立了众多的树，怎么就能够保证模型整体的效果变强呢？集成的目的是为了模型在样本上能表现出更好的效果，所以对于所有的提升集成算法，**每构建一个评估器，集成模型的效果都会比之前更好**。也就是随着迭代的进行，模型整体的效果必须要逐渐提升，最后要实现集成模型的效果最优。要实现这个目标，**我们可以首先从训练数据上着手**。

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？

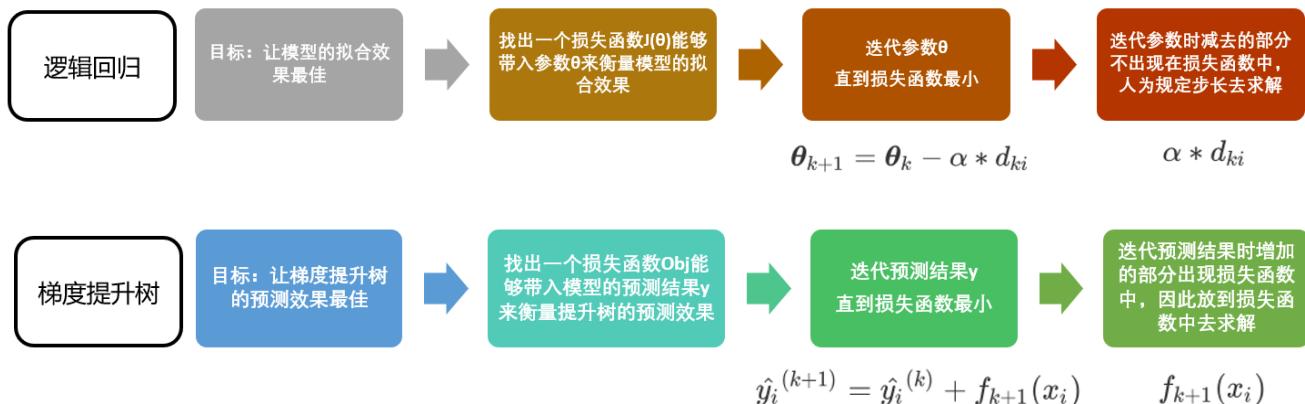
来回顾一下最优化问题的老朋友，我们的逻辑回归模型。在逻辑回归当中，我们有方程：

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

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：

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

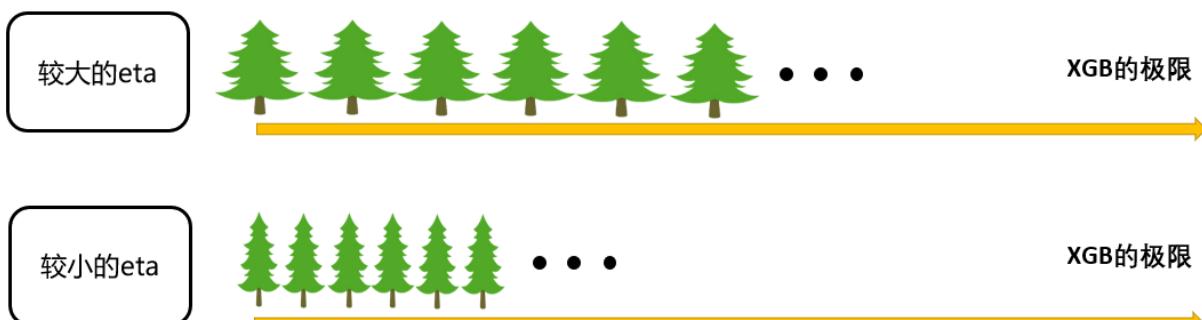
在 k 次迭代后，我们的集成算法中总共有 k 棵树，而我们前面讲明了， k 棵树的集成结果是前面所有树上的叶子权重的累加 $\sum_k^K f_k(x_i)$ 。所以我们让 k 棵树的集成结果 $\hat{y}_i^{(k)}$ 加上我们新建的树上的叶子权重 $f_{k+1}(x_i)$ ，就可以得到第 $k+1$ 次迭代后，总共 $k+1$ 棵树的预测结果 $\hat{y}_i^{(k+1)}$ 了。**我们让这个过程持续下去，直到找到能够让损失函数最小化的 \hat{y} ，这个 \hat{y} 就是我们模型的预测结果。**参数可以迭代，集成的树林也可以迭代，万事大吉！

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(kwagrs, 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有梯度提升树，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

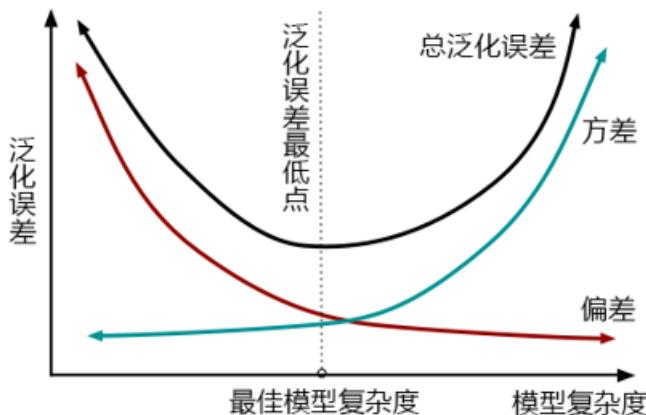
与其他算法一样，我们最小化目标函数以求模型效果最佳，并且我们可以通过限制*n_estimators*来限制我们的迭代次数，因此我们可以看到生成的每棵树所对应的目标函数取值。目标函数中的第二项看起来是与*K*棵树都相关，但我们的第一个式子看起来却只和样本量相关，仿佛只与当前迭代到的这棵树相关，这不是很奇怪么？

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。

<code>xgb.train()</code>	<code>xgb.XGBRegressor()</code>	<code>xgb.XGBClassifier()</code>
<code>obj</code> : 默认binary:logistic	<code>objective</code> : 默认reg:linear	<code>objective</code> : 默认binary:logistic

常用的选择有：

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

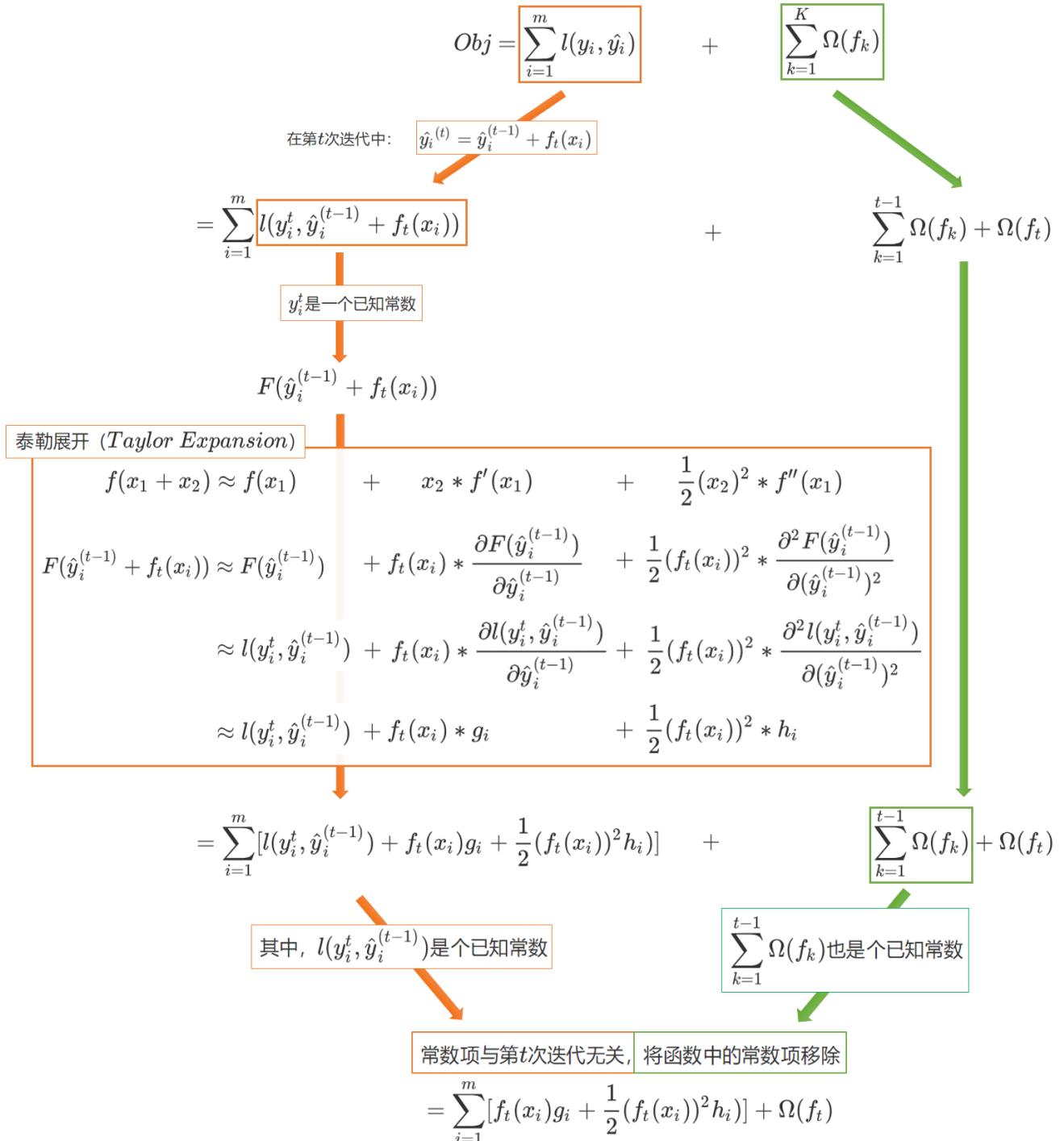
有了如下目标函数，我们来看看如何求解它。时刻记得我们求解目标函数的目的：为了求得在第 t 次迭代中最优的树 f_t 。在逻辑回归和支持向量机中，我们通常先将目标函数转化成一种容易求解的方式（比如对偶），然后使用梯度下降或者SMO之类的数学方法来执行我们的最优化过程。之前我们使用了逻辑回归的迭代过程来帮助大家理解在梯度提升树中树是如何迭代的，那我们是否可以使用逻辑回归的参数求解方式来求解XGB的目标函数呢？

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， $f_t(x_i)$ 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',
```

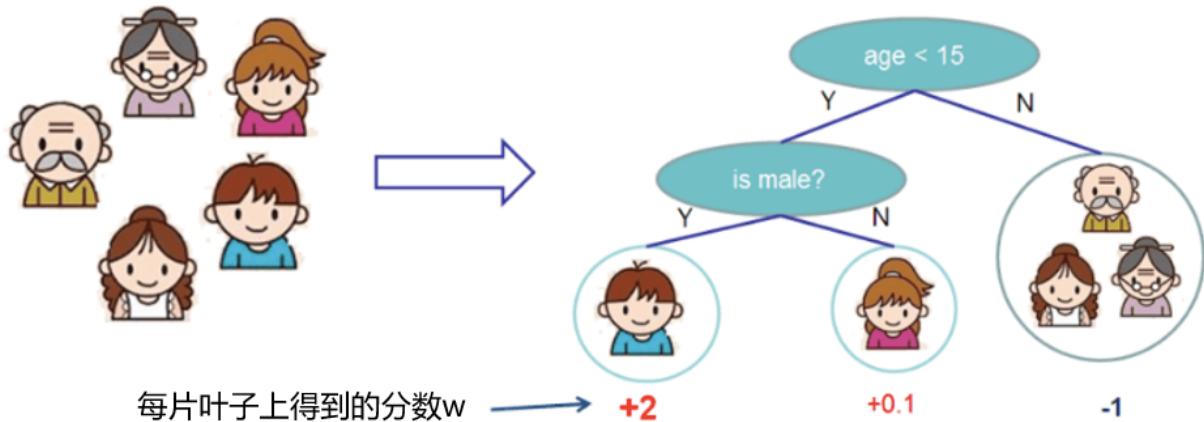
`**kwargs)*`

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

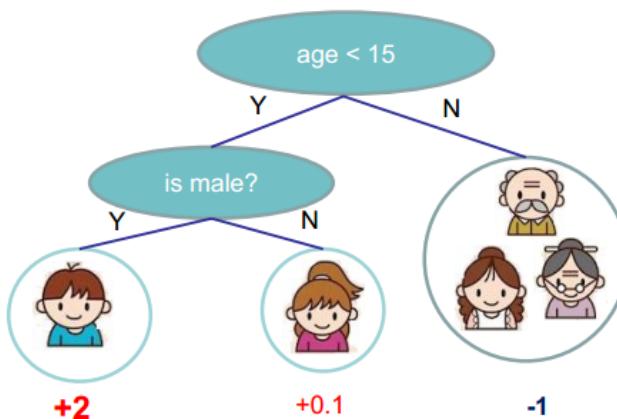
这个人有多喜欢电脑游戏呢？



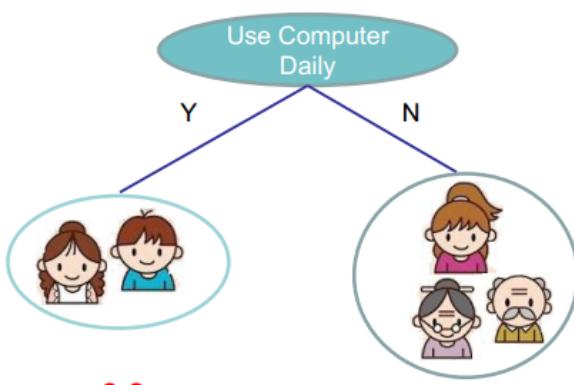
当有多棵树的时候，集成模型的回归结果就是所有树的预测分数之和，假设这个集成模型中总共有 K 棵决策树，则整个模型在这个样本*i*上给出的预测结果为：

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

第1棵树



第2棵树



$$f(\text{boy}) = 2 + 0.9 = 2.9 \quad f(\text{old man}) = -1 - 0.9 = -1.9$$

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。

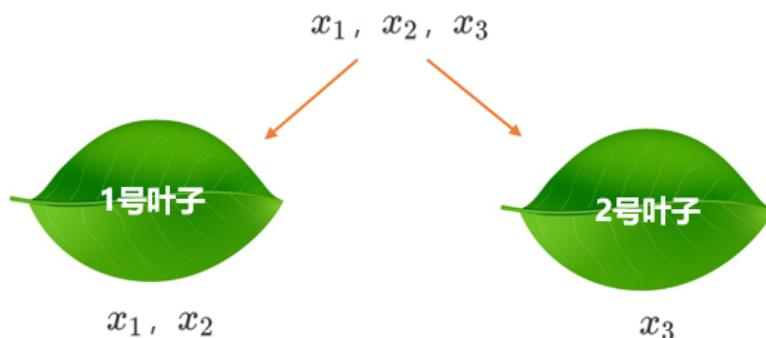
$$\begin{aligned} & \sum_{i=1}^m [f_t(x_i)g_i + \frac{1}{2}(f_t(x_i))^2 h_i] + \Omega(f_t) \\ &= \sum_{i=1}^m [w_{q(x_i)} g_i + \frac{1}{2} w_{q(x_i)}^2 h_i] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \\ &= \boxed{\sum_{i=1}^m w_{q(x_i)} g_i} + \boxed{\sum_{i=1}^m \frac{1}{2} w_{q(x_i)}^2 h_i}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

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：

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

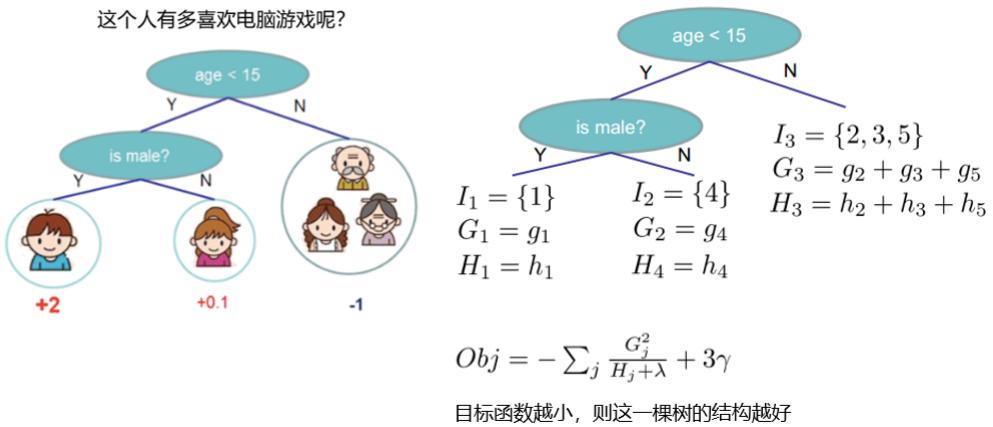
把这个公式带入目标函数，则有：

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

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

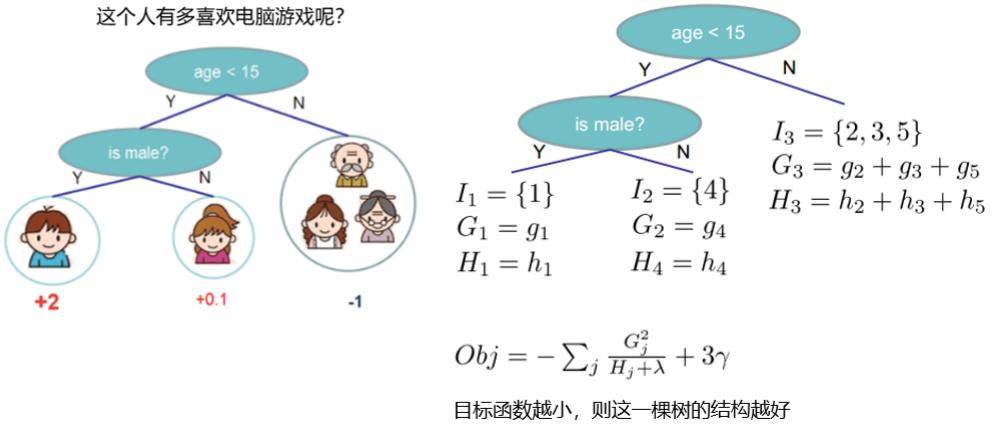
贪婪算法指的是控制局部最优来达到全局最优的算法，决策树算法本身就是一种使用贪婪算法的方法。XGB作为树的集成模型，自然也想到采用这样的方法来进行计算，所以我们认为，如果每片叶子都是最优，则整体生成的树结构就是最优，如此就可以避免去枚举所有可能的树结构。



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治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 设定越大，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, +∞]	gamma， 默认0， 取值范围[0, +∞]

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()

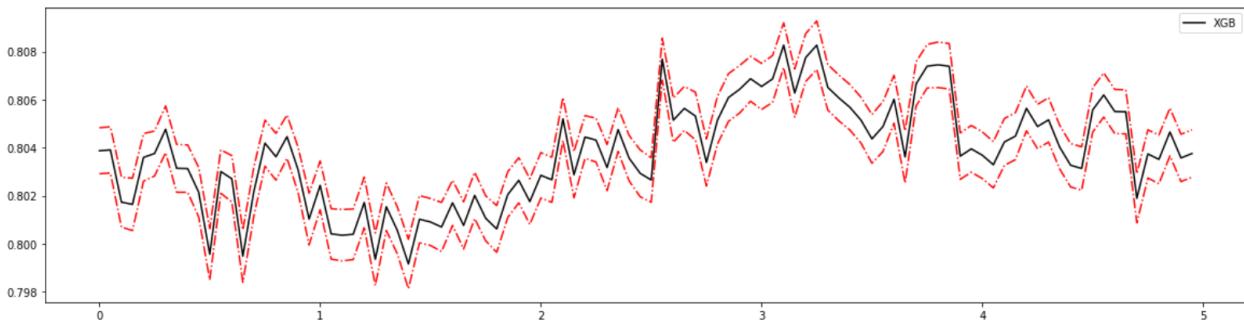
```

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 pred = clf.predict(Xtest)
clf.score(Xtest,Ytest)

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

recall(Ytest,y pred)

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

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

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

recall(Ytest,y 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。