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.ARDRegression	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作一个几乎人人熟悉的方程：

$$\hat{y}_i = w_0 + w_1 x_{i1} + w_2 x_{i2} + \ldots + w_n x_{in}$$

w被统称为模型的参数，其中w_0被称为截距(intercept)，w_1~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}~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作：

$$\hat{\boldsymbol{y}} = w_0 + w_1 \boldsymbol{x_1} + w_2 \boldsymbol{x_2} + \ldots + w_n \boldsymbol{x_n}$$

其中$\boldsymbol{y}$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$\boldsymbol{w}$可以被看做是一个结构为(n+1,1)的列矩阵，$\boldsymbol{X}$是一个结构为(m,n+1)的特征矩阵，则有：

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \\ \ldots \\ \hat{y}_m \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & x_{13} & \ldots & x_{1n} \\ 1 & x_{21} & x_{22} & x_{23} & \ldots & x_{2n} \\ 1 & x_{31} & x_{32} & x_{33} & \ldots & x_{3n} \\ & & \ldots & & & \\ 1 & x_{m1} & x_{m2} & x_{m3} & \ldots & x_{mn} \end{bmatrix} * \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \ldots \\ w_n \end{bmatrix}$$

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w}$$

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$\boldsymbol{X}$和标签值$\boldsymbol{y}$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$\boldsymbol{w}$。**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：

$$\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{m}(y_i - \boldsymbol{X_i w})^2$$

其中y_i是样本i对应的真实标签，$\hat{y}_i$，也就是$\boldsymbol{X_i w}$是样本i在一组参数$\boldsymbol{w}$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：

$$\min_{w} {||\boldsymbol{y} - \boldsymbol{X w}||_2}^2$$

其中右下角的2表示向量$\boldsymbol{y} - \boldsymbol{X w}$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 Sqaured 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$\boldsymbol{w}$，**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$\boldsymbol{w}$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，感兴趣可以走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$\boldsymbol{w}$求导：

$$\begin{aligned}\frac{\partial RSS}{\partial \boldsymbol{w}} &= \frac{\partial {||\boldsymbol{y} - \boldsymbol{X w}||_2}^2}{\partial \boldsymbol{w}} \\ &= \frac{\partial (\boldsymbol{y} - \boldsymbol{X w})^T (\boldsymbol{y} - \boldsymbol{X w})}{\partial \boldsymbol{w}}\end{aligned}$$

$$\because (A - B)^T = A^T - B^T \text{并且} (AB)^T = B^T * A^T$$

$$\begin{aligned}\therefore &= \frac{\partial (\boldsymbol{y}^T - \boldsymbol{w^T X^T})(\boldsymbol{y} - \boldsymbol{X w})}{\partial \boldsymbol{w}} \\ &= \frac{\partial (\boldsymbol{y}^T \boldsymbol{y} - \boldsymbol{w^T X^T y} - \boldsymbol{y}^T \boldsymbol{X w} + \boldsymbol{w^T X^T X w})}{\partial \boldsymbol{w}}\end{aligned}$$

$$\because \text{矩阵求导中，} a\text{为常数，有如下规则：}$$

$$\frac{\partial a}{\partial A} = 0, \quad \frac{\partial A^T B^T C}{\partial A} = B^T C, \quad \frac{\partial C^T B A}{\partial A} = B^T C, \quad \frac{\partial A^T B A}{\partial A} = (B + B^T)A$$

$$
\begin{aligned}
&= 0 - \boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y} - \boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y} + 2\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\boldsymbol{w} \\
&= \boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\boldsymbol{w} - \boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}
\end{aligned}
$$

我们让求导后的一阶导数为0：

$$
\begin{aligned}
\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\boldsymbol{w} - \boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y} &= 0 \\
\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\boldsymbol{w} &= \boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}
\end{aligned}
$$

左乘一个$\left(\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\right)^{-1}$则有：

$$
\boldsymbol{w} = \left(\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}
$$

到了这里，我们希望能够将$\boldsymbol{w}$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$\boldsymbol{w}$的最优解了。这个功能非常容易实现，只需要我们左乘$\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}$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$\boldsymbol{w}$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$\boldsymbol{X}\boldsymbol{w}$，也就能够计算出我们的预测值$\hat{\boldsymbol{y}}$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$\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}$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y必须经由正态化处理（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
fit_intercept	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
normalize	布尔值，可不填，默认为False 当fit_intercept设置为False时，将忽略此参数。如果为True，则特征矩阵X在进入回归之前将会被减去均值（中心化）并除以L2范式（缩放）。如果你希望进行标准化，请在fit数据之前使用preprocessing模块中的标准化专用类StandardScaler
copy_X	布尔值，可不填，默认为True 如果为真，将在X.copy()上进行操作，否则的话原本的特征矩阵X可能被线性回归影响并覆盖
n_jobs	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joblib.parallel_backend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。

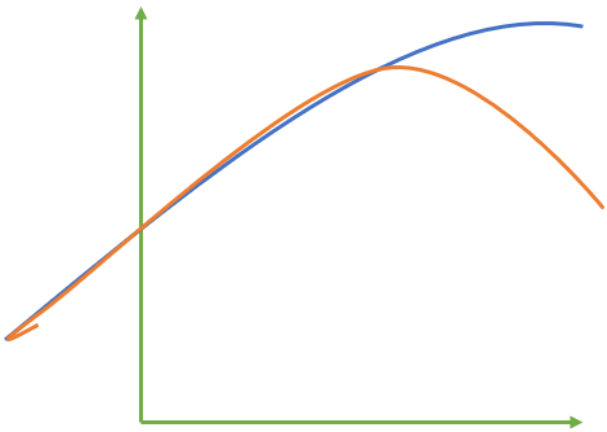
除了MSE，我们还有与MSE类似的MAE（Mean absolute error，绝对均值误差）：

$$MAE = \frac{1}{m}\sum_{i=0}^{m-1}|y_i - \hat{y}_i|$$

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足够的。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$\boldsymbol{y}$是我们的真实标签，$\hat{\boldsymbol{y}}$是我们的预测结果，$\bar{\boldsymbol{y}}$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R2的计算公式，R2明显和分类模型的指标中的accuracy或者precision不一样，R2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：

$$R^2 = 1 - \frac{\sum_{i=0}^{m}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{m}(y_i - \bar{y})^2} = 1 - \frac{RSS}{\sum_{i=0}^{m}(y_i - \bar{y})^2}$$

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：

$$TSS = \sum_{i=0}^{m}(y_i - \bar{y})^2$$

$$ESS = \sum_{i=0}^{m}(\hat{y}_i - \bar{y})^2$$

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，颠覆认知的时刻到来了——**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：

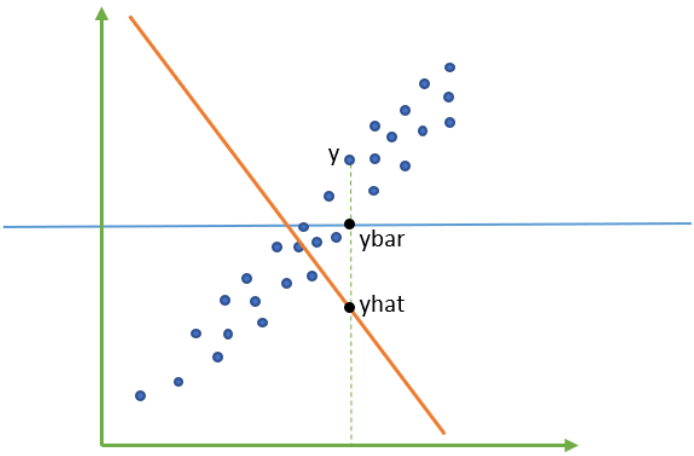
$$
1 = \frac{RSS}{TSS} + \frac{ESS}{TSS} + \frac{2\sum_{i=0}^{m}(y_i - \hat{y}_i)(\hat{y}_i - \bar{y})}{TSS}
$$

$$
1 - \frac{RSS}{TSS} = \frac{ESS + 2\sum_{i=0}^{m}(y_i - \hat{y}_i)(\hat{y}_i - \bar{y})}{TSS}
$$

$$
R^2 = \frac{ESS + 2\sum_{i=0}^{m}(y_i - \hat{y}_i)(\hat{y}_i - \bar{y})}{TSS}
$$

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：

$$\boldsymbol{X^T X w - X^T y = 0}$$

$$\boldsymbol{X^T X w = X^T y}$$

左乘一个$\boldsymbol{(X^T X)^{-1}}$则有：

$$\boldsymbol{w = (X^T X)^{-1} X^T y}$$

在最后一步中我们需要左乘$\boldsymbol{X^T X}$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：

$$\boldsymbol{A^{-1}} = \frac{1}{|\boldsymbol{A}|}\boldsymbol{A^*}$$

分字上$\boldsymbol{A^*}$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$|\boldsymbol{A}|$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$|\boldsymbol{X^T X}|$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$\boldsymbol{X}$结构为(m,n)，则$\boldsymbol{X^T X}$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。

$$\boldsymbol{X^T X} = (n, m) * (m, n) = (n, n)$$

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：

$$矩阵A = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

$$矩阵A的行列式 = \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} = |A|$$

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$\boldsymbol{A}$
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$detA$

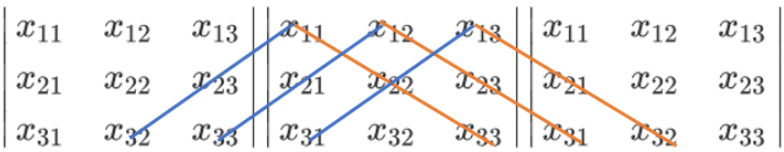
任何矩阵都可以有行列式。以一个3*3的行列式为例，我们来看看行列式是如何计算的：

$$|A| = \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix}$$

$$= x_{11}x_{22}x_{33} + x_{12}x_{23}x_{31} + x_{13}x_{21}x_{32} - x_{11}x_{23}x_{32} - x_{12}x_{21}x_{33} - x_{13}x_{22}x_{31}$$

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：

$$\begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix}$$



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：

$$|A| = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{vmatrix}$$

$$= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31}$$

$$= a_{11}a_{22}a_{33} + a_{12}a_{23} * 0 + a_{13} * 0 * 0 - a_{11}a_{23} * 0 - a_{12} * 0 * a_{33} - a_{13}a_{22} * 0$$

$$= a_{11}a_{22}a_{33}$$

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：

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 4 \end{bmatrix}, B = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 4.002 \end{bmatrix}, C = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 17 \end{bmatrix}$$

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：

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 4 \end{bmatrix}, B = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 4.002 \end{bmatrix}, C = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 17 \end{bmatrix} - 第一行 * 2$$

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 0.002 \end{bmatrix}, C = \begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 13 \end{bmatrix}$$

继续进行变换：

$$A=\begin{bmatrix}1&1&2\\5&3&11\\0&0&0\end{bmatrix},\ B=\begin{bmatrix}1&1&2\\5&3&11\\0&0&0.002\end{bmatrix},\ C=\begin{bmatrix}1&1&2\\5&3&11\\0&0&13\end{bmatrix}\circlearrowleft -\ 第一行*5$$

$$A=\begin{bmatrix}1&1&2\\0&-2&1\\0&0&0\end{bmatrix},\ B=\begin{bmatrix}1&1&2\\0&-2&1\\0&0&0.002\end{bmatrix},\ C=\begin{bmatrix}1&1&2\\0&-2&1\\0&0&13\end{bmatrix}$$

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$\boldsymbol{X^TX}$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。**

$$(\boldsymbol{X^TX})^{-1}=\frac{1}{|\boldsymbol{X^TX}|}(\boldsymbol{X^TX})^*\rightarrow\frac{1}{0}(\boldsymbol{X^TX})^*\rightarrow 除零错误$$

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：

$$(\boldsymbol{X^TX})^{-1}=\frac{1}{|\boldsymbol{X^TX}|}(\boldsymbol{X^TX})^*\rightarrow\frac{1}{非常接近0的数}(\boldsymbol{X^TX})^*\rightarrow\infty$$

$$\boldsymbol{w}=(\boldsymbol{X^TX})^{-1}\boldsymbol{X^Ty}\rightarrow\infty$$

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。

$$(\boldsymbol{X^TX})^{-1}=\frac{1}{|\boldsymbol{X^TX}|}(\boldsymbol{X^TX})^*\rightarrow\frac{1}{不是非常接近于0的常数}(\boldsymbol{X^TX})^*\rightarrow 逆矩阵的大小正常$$

$$\boldsymbol{w}=(\boldsymbol{X^TX})^{-1}\boldsymbol{X^Ty}\rightarrow 拟合出适合的w$$

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。

$$w = (X^T X)^{-1} X^T y$$

逆必须存在，则行列式不能为0 → 行列式不能为0，则要求矩阵必须满秩 → 矩阵必须满秩，则特征之间不能存在多重共线性

- **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，我们都有可能会无视掉它。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：

$$\min_{w} \ {||\boldsymbol{Xw}-\boldsymbol{y}||_2}^2+\alpha{||\boldsymbol{w}||_2}^2$$

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：

$$\begin{aligned}\frac{\partial(RSS+\alpha{||\boldsymbol{w}||_2}^2)}{\partial \boldsymbol{w}} &= \frac{\partial\left({||\boldsymbol{y}-\boldsymbol{Xw}||_2}^2+\alpha{||\boldsymbol{w}||_2}^2\right)}{\partial \boldsymbol{w}} \\ &= \frac{\partial(\boldsymbol{y}-\boldsymbol{Xw})^T(\boldsymbol{y}-\boldsymbol{Xw})}{\partial \boldsymbol{w}}+\frac{\partial\alpha{||\boldsymbol{w}||_2}^2}{\partial \boldsymbol{w}}\end{aligned}$$

前半部分我们推导过，后半部分对w求导非常简单：

$$= 0-2\boldsymbol{X^T y}+2\boldsymbol{X^T Xw}+2\alpha\boldsymbol{w}$$

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：

$$\begin{aligned}&= (\boldsymbol{X^T X}+\alpha\boldsymbol{I})\boldsymbol{w}-\boldsymbol{X^T y} \\ (\boldsymbol{X^T X}+\alpha\boldsymbol{I})\boldsymbol{w} &= \boldsymbol{X^T y}\end{aligned}$$

现在，只要$(\boldsymbol{X^T X}+\alpha\boldsymbol{I})$存在逆矩阵，我们就可以求解出$\boldsymbol{w}$。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$\boldsymbol{X^T X}$就会不满秩（存在全为零的行）：

$$\boldsymbol{X^TX} = \begin{vmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{3n} \\ & \dots & & & \\ 0 & 0 & 0 & \dots & 0 \end{vmatrix}$$

此时方阵$\boldsymbol{X^TX}$就是没有逆的，最小二乘法就无法使用。然而，加上了$\alpha\boldsymbol{I}$之后，我们的矩阵就大不一样了：

$$\alpha\boldsymbol{I} = \alpha * \begin{vmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ & \dots & & & \\ 0 & 0 & 0 & \dots & 1 \end{vmatrix} = \begin{vmatrix} \alpha & 0 & 0 & \dots & 0 \\ 0 & \alpha & 0 & \dots & 0 \\ 0 & 0 & \alpha & \dots & 0 \\ & \dots & & & \\ 0 & 0 & 0 & \dots & \alpha \end{vmatrix}$$

$$\boldsymbol{X^TX} + \alpha\boldsymbol{I} = \begin{vmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{3n} \\ & \dots & & & \\ 0 & 0 & 0 & \dots & 0 \end{vmatrix} + \begin{vmatrix} \alpha & 0 & 0 & \dots & 0 \\ 0 & \alpha & 0 & \dots & 0 \\ 0 & 0 & \alpha & \dots & 0 \\ & \dots & & & \\ 0 & 0 & 0 & \dots & \alpha \end{vmatrix}$$

$$= \begin{vmatrix} a_{11}+\alpha & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22}+\alpha & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33}+\alpha & \dots & a_{3n} \\ & & \dots & & \\ 0 & 0 & 0 & \dots & \alpha \end{vmatrix}$$

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$\boldsymbol{X^TX}$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$\boldsymbol{X^TX} + \alpha\boldsymbol{I}$永远都是满秩。在sklearn中，$\alpha$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$\boldsymbol{w}$就可以写作：

$$(\boldsymbol{X^TX} + \alpha\boldsymbol{I})\boldsymbol{w} - \boldsymbol{X^Ty} = 0$$

$$(\boldsymbol{X^TX} + \alpha\boldsymbol{I})\boldsymbol{w} = \boldsymbol{X^Ty}$$

左乘一个$(\boldsymbol{X^TX} + \alpha\boldsymbol{I})^{-1}$则有：

$$\boldsymbol{w} = (\boldsymbol{X^TX} + \alpha\boldsymbol{I})^{-1}\boldsymbol{X^Ty}$$

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$\boldsymbol{X^TX} + \alpha\boldsymbol{I}$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。

$$(\boldsymbol{X}^T\boldsymbol{X}+\alpha\boldsymbol{I})^{-1}=\frac{1}{|\boldsymbol{X}^T\boldsymbol{X}+\alpha\boldsymbol{I}|}(\boldsymbol{X}^T\boldsymbol{X}+\alpha\boldsymbol{I})^*$$

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，因此我们往往只能够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")
```

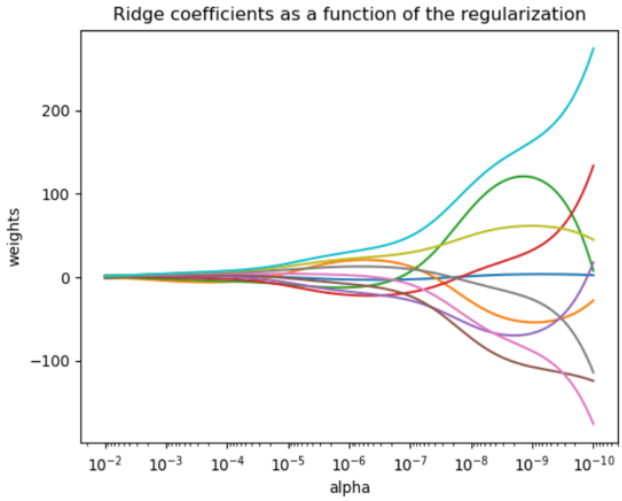
```
plt.legend()
plt.show()
```

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
```

```
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。岭回归最初始由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：

$$\min_{w} \ {||\boldsymbol{Xw}-\boldsymbol{y}||_2}^2+\alpha||\boldsymbol{w}||_1$$

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：

$$\begin{aligned}\frac{\partial(RSS+||\boldsymbol{w}||_1)}{\partial \boldsymbol{w}} &= \frac{\partial\left({||\boldsymbol{y}-\boldsymbol{Xw}||_2}^2+\alpha||\boldsymbol{w}||_1\right)}{\partial \boldsymbol{w}} \\ &= \frac{\partial(\boldsymbol{y}-\boldsymbol{Xw})^T(\boldsymbol{y}-\boldsymbol{Xw})}{\partial \boldsymbol{w}}+\frac{\partial\alpha||\boldsymbol{w}||_1}{\partial \boldsymbol{w}}\end{aligned}$$

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：

$$= 0-2\boldsymbol{X^T y}+2\boldsymbol{X^T Xw}+\alpha$$

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：

$$= \boldsymbol{X^T Xw}-\boldsymbol{X^T y}+\frac{\alpha \boldsymbol{I}}{2}$$

$$\boldsymbol{X^T Xw}=\boldsymbol{X^T y}-\frac{\alpha \boldsymbol{I}}{2}$$

大家可能已经注意到了，现在问题又回到了要求X^TX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X^TX加上一个单位矩阵，以此来防止方阵X^TX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X^TX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应该是"高度相关"，而如果我们假设方阵X^TX的逆是一定存在的，那我们可以有：

$$w = (X^TX)^{-1}(X^Ty - \frac{\alpha I}{2})$$

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：

$$\min_w \frac{1}{2n_{samples}}||Xw - y||_2^2 + \alpha||w||_1$$

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 converg
e well. You are advised to use the LinearRegression estimator

C:\Python\lib\site-packages\sklearn\linear_model\coordinate_descent.py:478: UserWarning: Coordinate descent wit
h 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 n
ot 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$\boldsymbol{x_1},\boldsymbol{x_2},...\boldsymbol{x_n}$。对于每一个$\alpha$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$\boldsymbol{w}$，其中包含了n+1个参数，分别是$w_0,w_1,w_2,...w_n$。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，用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，不过他们都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。

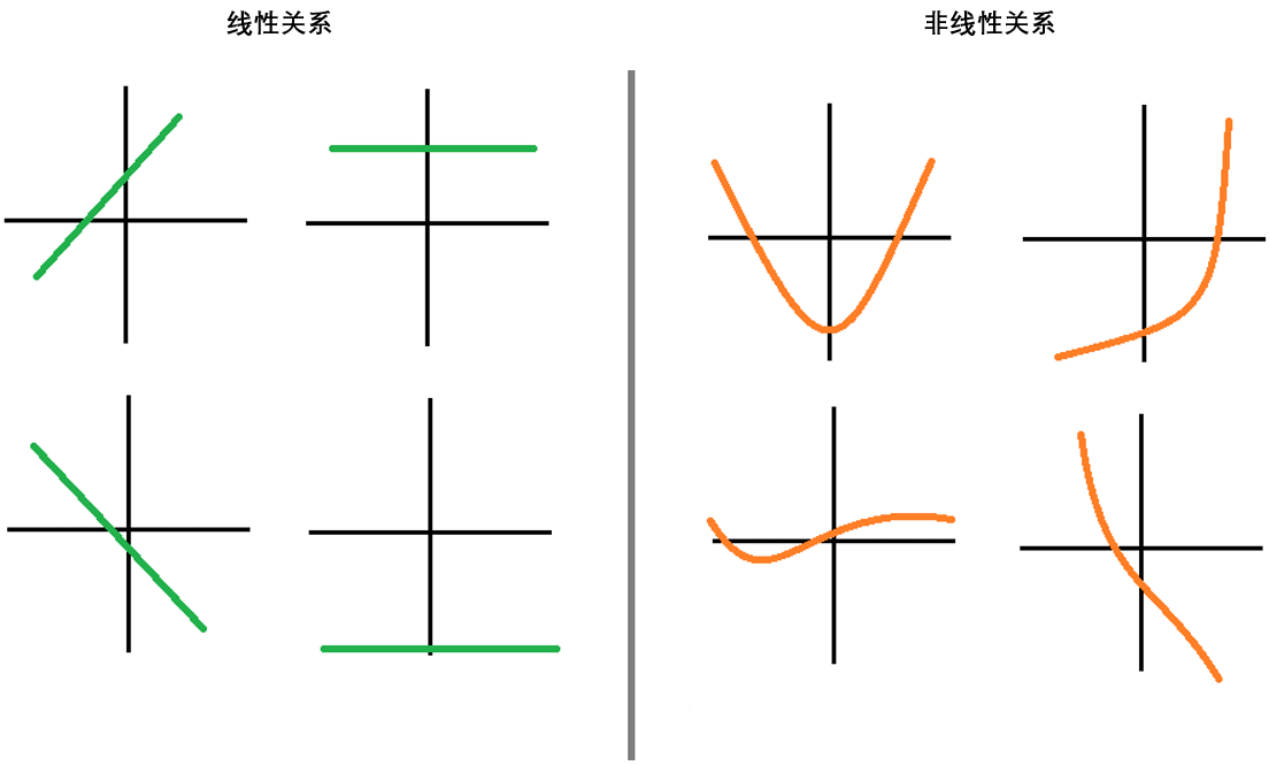
5 非线性问题：多项式回归

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。

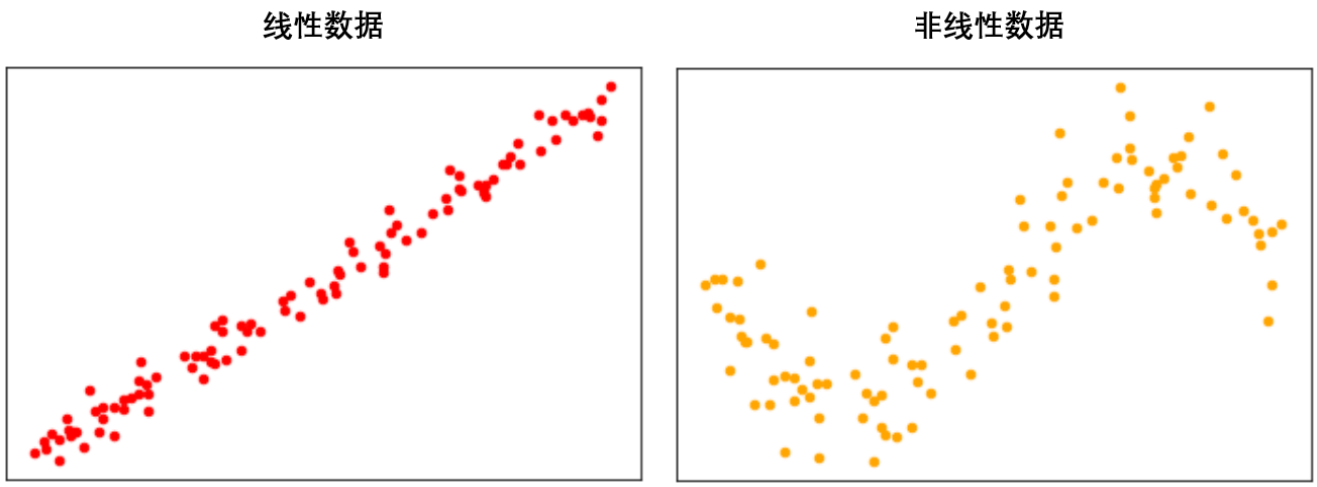
5.1.1 变量之间的线性关系

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）。

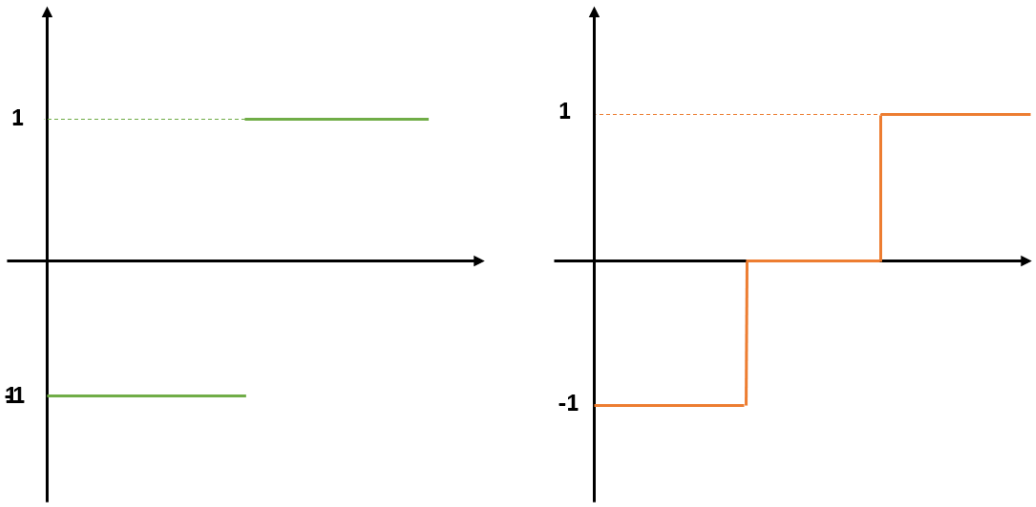


5.1.2 数据的线性与非线性

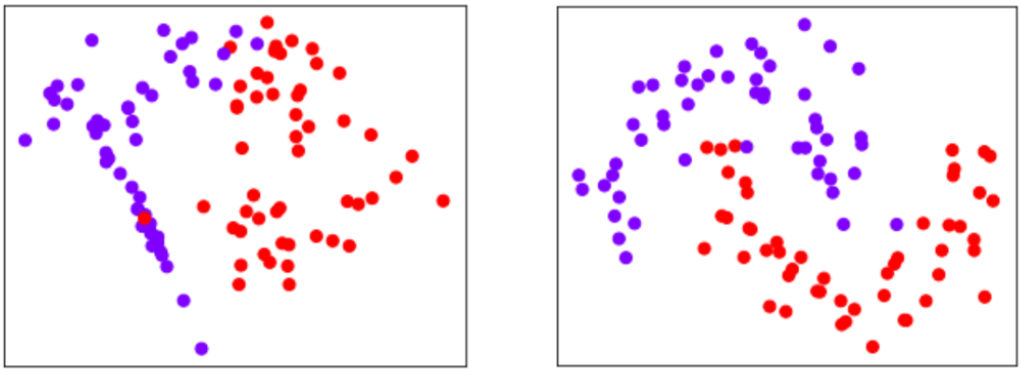
从线性关系这个概念出发，我们有了一种说法叫做“**线性数据**”。通常来说，一组数据由多个特征和标签组成。当这些特征分别与标签存在线性关系的时候，我们就说这一组数据是线性数据。当特征矩阵中任意一个特征与标签之间的关系需要使用三角函数，指数函数等函数来定义，则我们就说这种数据叫做“非线性数据”。对于线性和非线性数据，最简单的判别方法就是利用模型来帮助我们——如果是做分类则使用逻辑回归，如果做回归则使用线性回归，如果效果好那数据是线性的，效果不好则数据不是线性的。当然，也可以降维后进行绘图，绘制出的图像分布接近一条直线，则数据就是线性的。



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分别罢了。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使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，所以我们可以通过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：

$$y = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 \ldots . w_n x_n$$

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$\boldsymbol{w}$，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是**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?
```

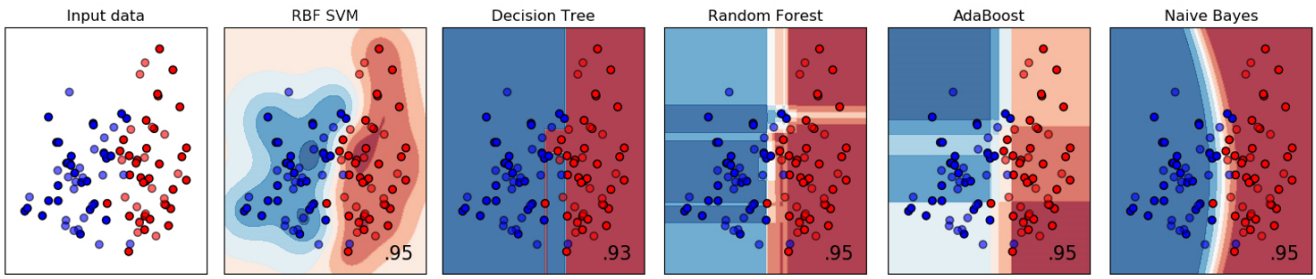
从图像上可以看出，线性回归无法拟合出这条带噪音的正弦曲线的真实面貌，只能够模拟出大概的趋势，而决策树却通过建立复杂的模型将几乎每个点都拟合出来了。可见，使用线性回归模型来拟合非线性数据的效果并不好，而决策树这样的模型却拟合得太细致，但是相比之下，还是决策树的拟合效果更好一些。

决策树无法写作一个方程（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，而有的模型可以既可以是线性，也可以是非线性模型！**接下来，我们就来一一讨论这些问题。

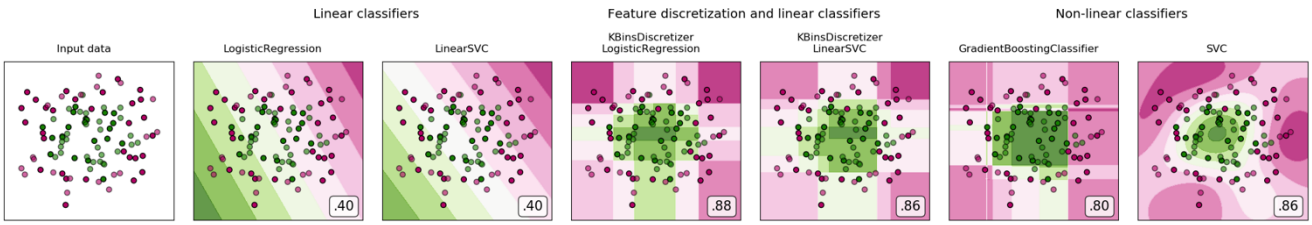
- **非线性模型拟合线性数据**

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。

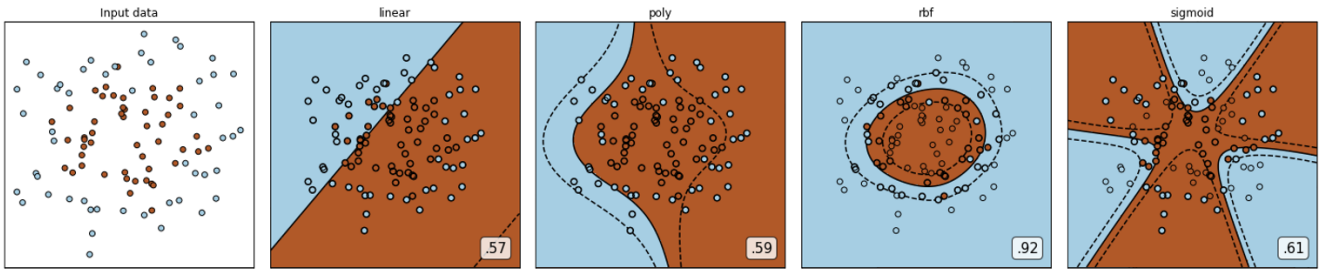


从上面的图中，我们可以观察出一个特性：线性模型们的决策边界都是一条条平行的直线，而非线性模型们的决策边界是交互的直线（格子），曲线，环形等等。对于分类模型来说，这是我们判断模型是线性还是非线性的重要评判因素：**线性模型的决策边界是平行的直线，非线性模型的决策边界是曲线或者交叉的直线。**之前我们提到，模型上如果自变量上的最高次方为1，则模型是线性的，但这种方式只适用于回归问题。分类模型中，我们很少讨论模型是否线性，因为我们很少使用线性模型来执行分类任务（逻辑回归是一个特例）。但从上面我们总结出的结果来看，我们可以认为**对分类问题而言，如果一个分类模型的决策边界上自变量的最高次方为1，则我们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够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"：使用做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?
```

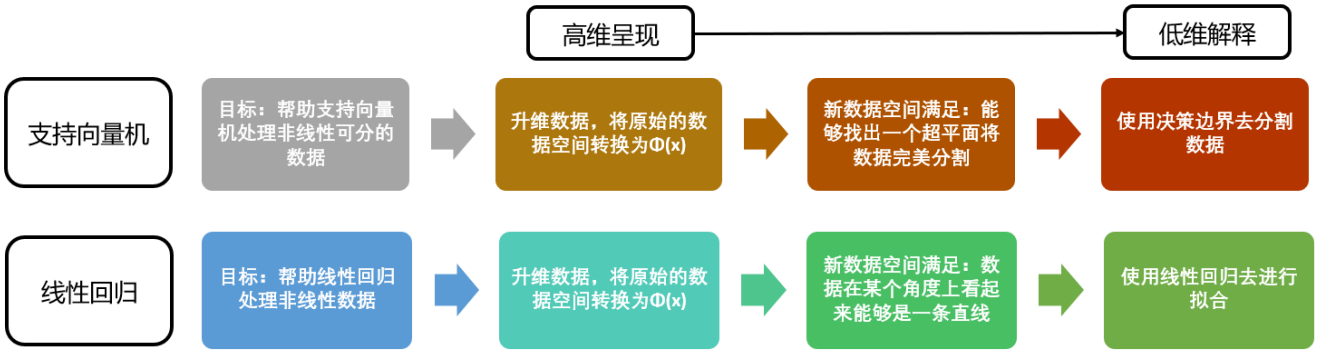
```
from sklearn.model_selection import cross_val_score as CVS
import numpy as np

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：

$$x_0 \quad x \quad x^2 \quad x^3$$

```
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：

$$y = w_0x_0 + w_1x + w_2x^2 + w_3x^3, \ (x_0 = 1)$$

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：

$$y = w_0x_0 + w_1x + w_2x^2 + w_3x^3 \dots w_nx^n, \ (x_0 = 1)$$

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的列作为截距乘数的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?
#对于多项式回归来说，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_intercpet=False).fit(xxx,y).coef_
```

```
#查看截距
LinearRegression(fit_intercpet=False).fit(xxx,y).intercept_
```

不过，这只是一维状况的表达，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：

$$x_0 \quad x_1 \quad x_2 \quad x_1^2 \quad x_1x_2 \quad x_2^2$$

```
array([[ 1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  2.,  3.,  4.,  6.,  9.],
       [ 1.,  4.,  5., 16., 20., 25.]])
```

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：

$$[x_1, x_2] \rightarrow [x_0, x_1, x_2, x_1^2, x_1x_2, x_2^2]$$

$$y = w_0x_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_1x_2 + w_5x_2^2$$

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：

$$x_0 \quad x_1 \quad x_2 \quad x_1^2 \quad x_1x_2 \quad x_2^2 \quad x_1^3 \quad x_1^2x_2 \quad x_1x_2^2 \quad x_2^3$$

```
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.]])
```

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是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：

$$[x_0, x_1, x_2, x_1^2, x_1x_2, x_2^2]$$

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：

$$y = w_0z_0 + w_1z_1 + w_2z_2 + w_3z_3 + w_4z_4 + w_5z_5 \tag{2}$$

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够与线性回归连用。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。