姓　名	王凯
学　号	201912190060

成　绩	
评卷人	

中 南 财 经 政 法 大 学

研 究 生 课 程 考 试 试 卷

（《机器学习》课程论文）

论文题目 机器学习期末作业

课程名称 机器学习

完成时间 2020 年 4 月 25 日

专业年级 大数据商务统计专硕 2019 级

注：研究生必须在规定期限内完成课程考试论文，并用 A4 页面打印，加此封面装订成册后，送交评审教师。教师应及时评定成绩，并至迟在下学期开学后两周内将此课程论文及成绩报告单一并交本单位研究生秘书存档。（涉及外单位的，由研究生秘书转交学生所在单位研究生秘书存档）

目　　录

一、 回归分析

（一） 多元线性回归

1. 理论介绍

在统计学中，线性回归是利用称为线性回归方程的最小二乘函数对一个或多个自变量和因变量之间关系进行建模的一种回归分析。这种函数是一个或多个称为回归系数的模型参数的线性组合。只有一个自变量的情况称为简单回归，大于一个自变量情况叫做多元回归。

给定由 d 个属性描述的示例 $X=(x_1;x_2;\cdots;x_d)$，其中x_i是无在第 i 个属性上的取值，线性模型试图学得一个通过属性的线性组合来进行预测的函数，即

$$f(X)=w_1x_1+w_2x_2+\cdots+w_dx_d+b$$

一般用向量形式写成

$$f(X)=w^Tx+b$$

其中 $W=(w_1;w_2;\cdots;w_d)$。在 W 和 b 学得之后，模型就得以确定。

给定数据集$D=\{(X_1,y_1),(X_2,y_2),\cdots,(X_m,y_m)\}$, 其中$X_i=(x_{i1};x_{i2};\cdots;x_{id})$, $y_i\in R$。线性回归试图学得一个线性模型以尽可能准确地预测实值输出标记。

多元线性回归模型试图学得：

$$f(X_i)=W^TX_i+b, \text{使得} f(X_i)\cong y_i$$

这称为“多元线性回归”。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的波士顿房价数据集，该数据集有 13 个特征变量，1 个连续型目标变量，希望通过建立多元线性回归模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 使用 python 自带的波士顿房价数据集
from sklearn.datasets import load_boston
X,y=load_boston(return_X_y=True)
print("样本的个数 = {},      特征的个
数 = {}".format(X.shape[0],X.shape[1]))
print("Boston 房价数据集描述:",load_boston().DESCR)
# 采用留出法划分数据集
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0
.25,random_state=123)
# 将数据进行标准化
from sklearn.preprocessing import StandardScaler
scaler=StandardScaler().fit(X_train)
X_train=scaler.transform(X_train)
X_test=scaler.transform(X_test)
# 导入多元线性回归函数
```

```
from sklearn.linear_model import LinearRegression
model_lr=LinearRegression()  # 模型实例化
# 模型训练及预测
model_lr=model_lr.fit(X_train,y_train)
y_pred=model_lr.predict(X_test)
# 模型评价：采用 MSE 和 R^2 来进行评价
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
MSE=mean_squared_error(y_pred,y_test)
r2=r2_score(y_pred,y_test)
print("多元线性回归模型拟合的均方误差为:{}".format(round(MSE,3)))
print("多元线性回归可以解释原变量%.3f%%的信息"% (r2 *100))
```

（3） 程序结果：

```
# 导入多元线性回归函数
from sklearn.linear_model import LinearRegression
model_lr=LinearRegression()    # 模型实例化
# 模型训练及预测
model_lr=model_lr.fit(X_train,y_train)
y_pred=model_lr.predict(X_test)
# 模型评价：采用MSE和R^2来进行评价
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
MSE=mean_squared_error(y_pred,y_test)
r2=r2_score(y_pred,y_test)
print("多元线性回归模型拟合的均方误差为:{}".format(round(MSE,3)))
print("多元线性回归可以解释原变量%.3f%%的信息"% (r2 *100))
```

```
多元线性回归模型拟合的均方误差为:24.769
多元线性回归可以解释原变量60.536%的信息
```

（二） 多项式回归

1. 理论介绍

回归分析的目标是根据自变量（或自变量向量）x 的值来模拟因变量 y 的期望值。在简单的线性回归中，使用模型

$$y = \beta_0 + \beta_1 x + \varepsilon$$

其中 ε 是未观察到的随机误差，其以标量 x 为条件，均值为零。在该模型中，对于 x 值的每个单位增加，y 的条件期望增加β_1个单位。

在许多情况下，这种线性关系可能不成立。在某些情况下，响应变量和预测变量的真实关系可能是非线性的。为了体现响应变量和预测变量之间的非线性关系，将线性模型推广最自然的方法是将标准线性模型

$$y_i = \beta_0 + \beta_1 X_i + \varepsilon_i$$

换成一个多项式函数

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \beta_3 x_i^3 + \cdots + \beta_d x_i^d + \varepsilon_i$$

其中ε_i是误差项。这种方法称为多项式回归，对于阶数比较大的 d，多项式回归将呈现明显的非线性曲线。注意到多项式函数本质上可以视为预测变量$x_i, x_i^2, x_i^3, \cdots, x_i^d$的标准线性模型，因此用最小二乘回归的方法就能求解得到各项系数。对多项式阶数 d 的选择不宜过大，一般不大于 3 或者 4，这是因为 d 越大，多项式曲线就会越光滑甚至会在 X 变量定义域的边界处呈现异样的形状。

优点：相较于线性回归模型，能够拟合非线性可分的数据，总体上更加灵活，可以拟合一些相当复杂的关系。

缺点：需要完全控制要素变量进行建模，需要仔细的设计，考量一些数据的先验知识才能选择最佳的指数，如果指数选择不当，容易造成过拟合问题。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的波士顿房价数据集，该数据集有 13 个特征变量，1 个连续型目标变量，希望通过建立多项式回归模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 数据集仍为 Boston 房价数据集，且数据集划分和标准化处理均同上
# 导入线性回归函数和将特征生成多项式的函数，并将其结合为 pipeline
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
poly=PolynomialFeatures(degree=3)  # 将 degree 设为 3 次多项式
model_poly=make_pipeline(PolynomialFeatures(degree=2),LinearRe
gression())
# 模型训练及预测
model_poly=model_poly.fit(X_train,y_train)
y_pred=model_poly.predict(X_test)
# 模型评价：采用 MSE 和 R^2 来进行评价
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
MSE=mean_squared_error(y_pred,y_test)
r2=r2_score(y_pred,y_test)
print("多项式回归模型拟合的均方误差为:{}".format(round(MSE,3)))
print("多项式回归可以解释原变量%.3f%%的信息"% (r2 *100))
```

（3） 程序结果：

```
#  导入线性回归函数和将特征生成多项式的函数，并将其结合为pipeline
from  sklearn.linear_model  import  LinearRegression
from  sklearn.preprocessing  import  PolynomialFeatures
from  sklearn.pipeline  import  make_pipeline
poly=PolynomialFeatures(degree=3)    #  将degree设为3,3次多项式
model_poly=make_pipeline(PolynomialFeatures(degree=2),LinearRegression())
#  模型训练及预测
model_poly=model_poly.fit(X_train,y_train)
y_pred=model_poly.predict(X_test)
#  模型评价：采用MSE和R^2来进行评价
from  sklearn.metrics  import  mean_squared_error
from  sklearn.metrics  import  r2_score
MSE=mean_squared_error(y_pred,y_test)
r2=r2_score(y_pred,y_test)
print("多项式回归模型拟合的均方误差为:{}".format(round(MSE,3)))
print("多项式回归可以解释原变量%.3f%%的信息"%  (r2  *100))
```

```
多项式回归模型拟合的均方误差为:13.787
多项式回归可以解释原变量81.615%的信息
```

二、 判别分析

（一） Logistic 回归

1. 理论介绍

在使用线性模型进行学习时，我们处理的是回归任务，但是当目标变量变成分类变量时，我们应该如何处理。考虑二分类任务，其输出标记为 y∈{0,1}，而线性回归模型产生的预测值$z = \mathrm{W^T X_i} + \mathrm{b}$是实值，于是，我们需将实值 z 转换为 0-1 值。这时我们通过引入对数几率回归函数，

$$y = \frac{1}{1 + e^{-z}}$$

对数几率函数是一种“Sigmoid 函数”，它可以将 z 值转化为一个接近 0 或 1 的 y 值，并且其输出值在 z=0 附近变化很陡。将对数几率回归函数作为$g^-(\cdot)$，得到：

$$y = \frac{1}{1 + e^{-\mathrm{W^T X+b}}}$$

通过进一步变形可以得到：

$$ln\frac{y}{1-y} = \mathrm{W^T X + b}$$

若将 y 视为样本 X 作为正例的可能性，则 1-y 是其反例可能性，两者的比值$\frac{y}{1-y}$称为“几率”，反映了 X 作为正例的相对可能性。而这个式子实际上是在用线性回归模型的预测结果去逼近真实标记的对数几率，因此，我们将其称为逻辑回归。

优点：① 它是直接对分类可能性进行建模，无需事先假设数据分布，这样就避免了假设分布不准确所带来的问题；② 它不是仅预测出“类别”，而是可得到近似概率预测，这对许多需利用概率辅助决策的任务很有用；③ 对率函数是任意阶可导的凸函数，有很好的

数学性质，现有的许多数值优化算法都可直接用于求解最优解。

缺点：① Sigmoid 函数的导数值域在 0-1/4 之间，在进行求解梯度时会出现梯度保障或梯度消失的情况；② Sigmoid 函数会使得参数 W 值过大，所以需要在目标函数中加入正则项。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过建立 Logistic 回归模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 使用 python 自带的乳腺癌数据集（二分类）
from sklearn.datasets import load_breast_cancer
X,y=load_breast_cancer(return_X_y=True)
print("样本的个数 = {}, 特征的个
数 = {}".format(X.shape[0],X.shape[1]))
# 采用留出法划分数据集
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0
.25,random_state=123)
# 将数据进行标准化
from sklearn.preprocessing import StandardScaler
scaler=StandardScaler().fit(X_train)
X_train=scaler.transform(X_train)
X_test=scaler.transform(X_test)
# 导入逻辑回归函数
from sklearn.linear_model import LogisticRegression
model_logistic=LogisticRegression()
# 模型训练及预测
model_logistic=model_logistic.fit(X_train,y_train)
y_pred=model_logistic.predict(X_test)
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("逻辑回归模型拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
#  导入逻辑回归函数
from  sklearn.linear_model  import  LogisticRegression
model_logistic=LogisticRegression()
#  模型训练及预测
model_logistic=model_logistic.fit(X_train,y_train)
y_pred=model_logistic.predict(X_test)
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred,y_test))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("逻辑回归模型拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           0       0.98      1.00      0.99        53
           1       1.00      0.99      0.99        90

    accuracy                           0.99       143
   macro avg       0.99      0.99      0.99       143
weighted avg       0.99      0.99      0.99       143

逻辑回归模型拟合的准确率为:99.301%
```

（二） Probit 回归

1. 理论介绍

Probit 回归与 Logistic 回归相同，也是用于分类任务，同样是将线性回归模型得到的实值转化到 0-1 之间，同样是通过在线性回归模型套上分布函数来实现。

如果分布函数 F(·)使用的是标准的正态分布函数φ(·)，即

$$p(X_i) = \text{Prob}(Y_i = 1|X_i) = \varphi(X_i^T\beta) = \int_{-\infty}^{X_i^T\beta} \frac{1}{\sqrt{2\pi}} exp(-\frac{z^2}{2})dz$$

其中$\varphi(X_i^T\beta)$是正态分布的分布函数，取值范围是[0,1]。这时的概率模型我们就称为 Probit 模型。如果分布函数 F(·)使用的是 Logistic 模型，那么概率模型就称为 Logit 模型。

Logistic 回归与 Probit 回归的对比：

联系：① 参数的估计方法均使用最大似然估计；② 两个模型的估计系数均没有直接的经济意义，他们的边际影响不能像线性回归模型一样直接等于系数，这两种方法一般是计算其平均边际效应。

区别：① $Y_i = X_i^T\beta + \varepsilon_i$中，对$\varepsilon_i$的分布设定不同。Logit 模型中，$\varepsilon_i$服从标准 Logistic 分布，probit 模型中，ε_i服从标准的正态分布；② 两个模型估算的边际效应的差别主要体现在对尾部数据的解释上

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的

目标变量，希望通过建立 Probit 回归模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 使用 python 自带的乳腺癌数据集（二分类）
from sklearn.datasets import load_breast_cancer
X,y=load_breast_cancer(return_X_y=True)
print("样本的个数 = {}, 特征的个
数 = {}".format(X.shape[0],X.shape[1]))
# 采用留一法划分数据集
from sklearn.model_selection import LeaveOneOut
loo=LeaveOneOut()
Accuracy=[]
# loo.split():这里拆分的是 index
for train_index,test_index in loo.split(X):
    X_train,X_test=X[train_index],X[test_index]
    y_train,y_test=y[train_index],y[test_index]
    # 将数据进行标准化
    from sklearn.preprocessing import StandardScaler
    scaler=StandardScaler().fit(X_train)
    X_train=scaler.transform(X_train)
    X_test=scaler.transform(X_test)
    # 导入 Probit 回归函数
    from statsmodels.discrete.discrete_model import Probit
    import statsmodels.api as sm
    model = Probit(y_train, X_train)
    probit_model = model.fit()
    y_pred=probit_model.predict(X_test)
    y_pred=(y_pred>=0.5)*1
    # 模型评价
    from sklearn.metrics import accuracy_score
    Acc=accuracy_score(y_pred,y_test)
    Accuracy.append(Acc)
print("Accuracy 的个数为：{}".format(np.array(Accuracy).shape))
print("使用留一法划分数据集的 Probit 模型拟合的平均准确率
为:%.3f%%"% (np.mean(Accuracy) *100))
```

（3） 程序结果：

```python
#  使用python自带的乳腺癌数据集（二分类）
from  sklearn.datasets  import  load_breast_cancer
X,y=load_breast_cancer(return_X_y=True)
print("样本的个数  =  {},  特征的个数  =  {}".format(X.shape[0],X.shape[1]))
#  采用留一法划分数据集
from  sklearn.model_selection  import  LeaveOneOut
loo=LeaveOneOut()
Accuracy=[]
#  loo.split():这里拆分的是index
for  train_index,test_index  in  loo.split(X):
        X_train,X_test=X[train_index],X[test_index]
        y_train,y_test=y[train_index],y[test_index]
        #  将数据进行标准化
        from  sklearn.preprocessing  import  StandardScaler
        scaler=StandardScaler().fit(X_train)
        X_train=scaler.transform(X_train)
        X_test=scaler.transform(X_test)
        #  导入Probit回归函数
        from  statsmodels.discrete.discrete_model  import  Probit
        import  statsmodels.api  as  sm
        model  =  Probit(y_train,  X_train)
        probit_model  =  model.fit()
        y_pred=probit_model.predict(X_test)
        y_pred=(y_pred>=0.5)*1
        #  模型评价
        from  sklearn.metrics  import  accuracy_score
        Acc=accuracy_score(y_pred,y_test)
        Accuracy.append(Acc)
print("Accuracy的个数为：{}".format(np.array(Accuracy).shape))
print("使用留一法划分数据集的Probit模型拟合的平均准确率为:%.3f%%"%  (np.mean(Accuracy)  *100))
```

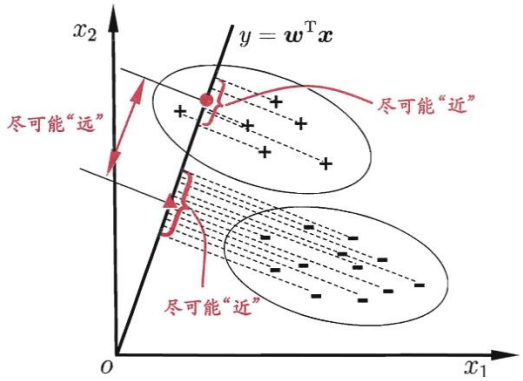
```
Accuracy的个数为：(569,)
使用留一法划分数据集的Probit模型拟合的平均准确率为:96.309%
```

（三） 线性判别分析

1. 理论介绍

线性判别分析（LDA）是一种经典的线性学习方法，在二分类问题上因为最早由[Fisher，1936]提出，亦称“Fisher 判别分析”。

LDA 的思想非常朴素：给定训练样例集，设法将样例投影到一条直线上，使得同类样例的投影点尽可能接近、异类样例的投影点尽可能远离；在对新样本进行分类时，将其投影到同样的这条直线上，再根据投影点的位置来确定新样本的类别。下图展示了 LDA 工作的二维示意图。



给定数据集 D={(x_i, y_i)}，$y_i \in \{0, 1\}$，令x_i、μ_i、$\sum_i$分别表示第 i∈ {0，1}类示例的集合、均值向量、协方差矩阵。若将数据投影到直线ω上，则两类样本的中心在直线上的投影分别为$\omega^T\mu_0$和$\omega^T\mu_1$；若将所有样本点都投影到直线上，则两类样本的协方差分别为$\omega^T\sum_0\omega$和$\omega^T\sum_1\omega$.由于直线是一维空间，因此$\omega^T\mu_0$、$\omega^T\mu_1$、$\omega^T\sum_0\omega$和$\omega^T\sum_1\omega$均为实数。

欲使同类样例的投影点尽可能接近，可以让同类样例投影点的协方差尽可能小，即$\omega^T\sum_0\omega + \omega^T\sum_1\omega$尽可能小；而欲使异类样例的投影点尽可能远离，可以让类中心之间的距离尽可能大，即$\|\omega^T\mu_0 - \omega^T\mu_1\|_2^2$尽可能大。同时考虑二者，则可得到欲最大化的目标

$$J = \frac{\|\omega^T\mu_0 - \omega^T\mu_1\|_2^2}{\omega^T\sum_0\omega + \omega^T\sum_1\omega}$$
$$= \frac{\omega^T(\mu_0 - \mu_1)(\mu_0 - \mu_1)^T\omega}{\omega^T(\sum_0 + \sum_1)\omega}$$

定义“类内散度矩阵”

$$S_w = \sum_0 + \sum_1 = \sum_{x\in x_0}(x - \mu_0)(x - \mu_0)^T + \sum_{x\in x_1}(x - \mu_1)(x - \mu_1)^T$$

以及“类间散度矩阵”

$$S_b = (\mu_0 - \mu_1)(\mu_0 - \mu_1)^T$$

则上式可重写为

$$J = \frac{\omega^T S_b\omega}{\omega^T S_w\omega}$$

这就是 LDA 欲最大化的目标，即S_b与S_w的“广义瑞利商” 。

优点：① 可以直接求得基于广义特征值问题的解析解，从而避免了在一般非线性算法中，如多层感知器，构建中所常遇到的局部最小问题无需对模式的输出类别进行人为的编码，从而使 LDA 对不平衡模式类的处理表现出尤其明显的优势；② 与神经网络方法相比，LDA 不需要调整参数，因而也不存在学习参数和优化权重以及神经元激活函数的选择等问题；对模式的归一化或随机化不敏感，而这在基于梯度下降的各种算法中则显得比较突出；③ 在某些实际情形中，LDA 具有与基于结构风险最小化原理的支持向量机（SVM）相当的甚至更优的推广性能，但其计算效率则远优于 SVM。正则判别分析法（CDA）寻找最优区分类别的坐标轴（k-1 个正则坐标，k 为类别的数量）。 这些线性函数是不相关的，实际上，它们通过 n 维数据定义了一个最优化的 k-1 个空间，能够最优的区分 k 个类（通过其在空间的投影）。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过建立线性判别分析模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入乳腺癌数据集，数据划分及标准化处理与
logistics 回归处理相同
# 导入线性判别函数：LDA
```

```
from sklearn.discriminant_analysis import LinearDiscriminantAn
alysis
model_lda=LinearDiscriminantAnalysis()
# 模型训练及预测
model_lda=model_lda.fit(X_train,y_train)
y_pred=model_lda.predict(X_test)
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("线性判别模型拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
# 导入线性判别函数：LDA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
model_lda=LinearDiscriminantAnalysis()
# 模型训练及预测
model_lda=model_lda.fit(X_train,y_train)
y_pred=model_lda.predict(X_test)
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("线性判别模型拟合的准确率为:%.3f%%"% (Acc *100))
```

```
              precision    recall  f1-score   support

           0       0.94      1.00      0.97        51
           1       1.00      0.97      0.98        92

    accuracy                           0.98       143
   macro avg       0.97      0.98      0.98       143
weighted avg       0.98      0.98      0.98       143

线性判别模型拟合的准确率为:97.902%
```

（四） 二次判别分析

1. 理论介绍

在进行分类任务时，可以使用贝叶斯定理进行分类，假设观测分成 K 类，K≥2，也就是说，定性的响应变量 Y 可以取 K 个不同的无序值。设π_k为一个随机选择的观测来自第 k 类的先验概率，即给定观测属于响应变量 Y 的第 k 类的概率。设$f_k(X) = Pr(X = x|Y = k)$表示第 k 类观测的 X 的密度函数，如果第 k 类的观测在 X≈x 附近有最大可能性，那么$f_k(x)$的值很大，反之$f_k(x)$的值则很小。贝叶斯定理可以表述为

$$\Pr(Y=k|X=x)=\frac{\pi_k f_k(x)}{\sum_{i=1}^{k}\pi_i f_i(x)}$$

上式是一种用π_k，$f_k(x)$来直接计算$\Pr(Y=k|X=x)$的方法。通常π_k的估计是容易求得的，取一些变量 Y 的随机样本，分别计算属于第 k 类的样本占总样本的比例。对于$f_k(x)$的估计要更复杂些，除非假设他们的密度函数形式简单。称$\Pr(Y=k|X=x)$为$X=x$的观测属于第 k 类的后验概率，即给定观测的预测变量值时，观测属于第 k 类的概率。

线性判别分析是假设每一类观测服从一个多元高斯分布，也就是说$f_k(x)$服从高斯正太分布，其中协方差矩阵对所有的k类都是相同的。而二次判别分析(QDA)提供了另一种方法。与线性判别分析一样，二次判别分析也是假设每一类观测都服从一个高斯分布，把参数估计带入到贝叶斯定理进行预测。然而与线性判别分析不同的是，二次判别分析假设每一类观测都有自己的协方差矩阵，也就是说它假设来自第 k 类的观测形如 X~N(μ_k，$\sum_k$)，$\sum_k$是第 k 类的协方差矩阵。在这种假设下贝叶斯分类器把观测$X=x$分入

$$\delta_k(x)=-\frac{1}{2}(x-\mu_k)^T\sum_k^{-1}(x-\mu_k)+log\pi_k$$

最大的一组。所以二次判别分析涉及把$\sum_k$，μ_k，π_k的估计带入上式中，然后将观测$X=x$分入使上式最大的一类里。由于上式是关于 x 的二次函数，所以取名为二次判别分析。

线性判别分析与二次判别分析的比较：

线性判别分析假设每一类观测都服从一个多元高斯分布，所有类的协方差矩阵相同，而二次判别分析假设每一类都有自己的协方差矩阵。LDA 没有 QDA 平滑，因此拥有更低方差，但是如果每一类的方差实际差距较大，LDA 将产生较大的偏差。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过建立二次判别分析模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
#  从 sklearn 的 datasets 数据库中导入乳腺癌数据集，数据划分及标准化处理与
logistics 回归处理相同
# 导入二次判别函数：QDA
from sklearn.discriminant_analysis import QuadraticDiscriminan
tAnalysis
model_qda=QuadraticDiscriminantAnalysis()
# 模型训练及预测
model_qda=model_qda.fit(X_train,y_train)
y_pred=model_qda.predict(X_test)
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
```

```
print("二次判别模型拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
#  导入二次判别函数：QDA
from  sklearn.discriminant_analysis  import  QuadraticDiscriminantAnalysis
model_qda=QuadraticDiscriminantAnalysis()
#  模型训练及预测
model_qda=model_qda.fit(X_train,y_train)
y_pred=model_qda.predict(X_test)
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred,y_test))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("二次判别模型拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           0       0.98      1.00      0.99        53
           1       1.00      0.99      0.99        90

    accuracy                           0.99       143
   macro avg       0.99      0.99      0.99       143
weighted avg       0.99      0.99      0.99       143

二次判别模型拟合的准确率为:99.301%
```

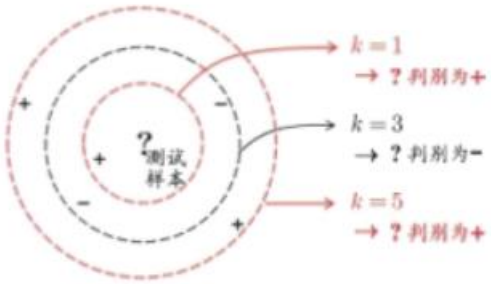
（五） K 近邻法

1. 理论介绍

K 近邻（KNN）学习是一种常用的监督学习方法，其工作机制非常简单：给定测试样本，基于某种距离度量找出训练集中与其最靠近的 K 个训练样本，然后基于这 K 个“邻居”的信息来进行预测。通常，在分类任务中可以使用“投票法”，即选择这 K 个样本中出现最多的类别标记作为预测结果；在回归任务中可使用“平均法”，即将这 K 个样本的实值输出标记的平均值作为预测结果；还可基于距离远近进行加权平均或加权投票，距离越近的样本权重越大。

同时，K 近邻与其他的机器学习算法不同，它似乎没有显式的训练过程。事实上，它是“懒惰学习”的著名代表，此类学习技术在训练阶段仅仅是把样本保存起来，训练时间开销为零，待收到测试样本后再进行处理；相应的，那些在训练阶段就对样本进行学习处理的方法，称为“急切学习”。

下图给出了 K 近邻分类器的一个示意图。显然，超参数 K 对 K 近邻算法的影响很大，当 K 取不同值时，分类结果会有显著不同。另一方面，若采用不同的距离计算公式，则找出的“近邻”可能有显著差别，从而也会导致分类结果有显著不同。



优点：① KNN 算法是最简单有效的分类算法，简单且容易实现；② 对于类内间距小，类间间距大的数据集分类效果好，而且对于边界不规则的数据效果好于线性分类器。

缺点：① KNN 对于随机分布的数据集分类效果较差， 对于样本不均衡的数据效果不好，需要进行改进。改进的方法是对 k 个近邻数据赋予权重，比如距离测试样本越近，权重越大；② KNN 很耗时，一般用于样本量较小的数据集，不适用于大数据集。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过建立 K 近邻模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入乳腺癌数据集，数据划分及标准化处理与 logistics 回归处理相同
# 导入 K 近邻函数：KNN
from sklearn.neighbors import KNeighborsClassifier
model_knn=KNeighborsClassifier(n_neighbors=5)
# 模型训练及预测
model_knn=model_knn.fit(X_train,y_train)
y_pred=model_knn.predict(X_test)
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("K 近邻模型拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
#  导入K近邻函数：KNN
from  sklearn.neighbors  import  KNeighborsClassifier
model_knn=KNeighborsClassifier(n_neighbors=5)
#  模型训练及预测
model_knn=model_knn.fit(X_train,y_train)
y_pred=model_knn.predict(X_test)
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred,y_test))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("K近邻模型拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           0       0.96      0.98      0.97        53
           1       0.99      0.98      0.98        90

    accuracy                           0.98       143
   macro avg       0.98      0.98      0.98       143
weighted avg       0.98      0.98      0.98       143

K近邻模型拟合的准确率为:97.902%
```

（六）朴素贝叶斯

1. 理论介绍

贝叶斯决策论是概率框架下实施决策的基本方法。对分类任务来说，在所有概率都已知的理想情形下，贝叶斯决策论考虑如何基于这些概率和误判损失来选择最优的类别标记。下面我们以多分类任务为例来解释其基本原理。

假设有 N 种可能的类别标记，即$y = \{c1, c2, \ldots, cN\}$，$\lambda_{ij}$是将一个事实标记为$c_j$的样本误分类为$c_i$所产生的损失。基于后验概率$P(c_i|x)$可获得将样本 x 分类为$c_i$所产生的期望损失，即在样本 x 上的“条件风险”。

$$P(c_i|x) = \sum_{j=1}^{N} \lambda_{ij} P(c_j|x)$$

我们的任务是寻找一个判定准则 h：$X \to y$以最小化风险

$$R(h) = E_x[R(h(x)|x)]$$

显然，对每个样本 x，若 h 能最小化条件风险$R(c|x)$，则总体风险$R(h)$也将被最小化。这就产生了贝叶斯判断准则：为最小化总体风险，只需在每个样本上选择那个能使条件风险$R(c|x)$最小的类别标记，即：

$$h^*(x) = \underset{c \in y}{\operatorname{argmin}} R(c|x)$$

此时，h^*称为贝叶斯最优分类器，与之对应的总体风险$R(h^*)$称为贝叶斯风险。$1 - R(h^*)$

反映了分类器所能达到的最好性能，即通过机器学习所能产生的模型精度的理论上限。

理论上，概率模型分类器使一个条件概率模型：$P(c|x_1,x_2,\dots,x_n)$，独立的类别 C 有若干类别，条件依赖于若干特征变量$x_1,x_2,\dots,x_n$。但问题在于如果特征数量 n 较大或者每个特征能取大量值时，基于概率模型列出概率表变得不现实。所以我们修改这个模型使之变得可行。贝叶斯定理有以下式子：

$$P(c|x_1,x_2,\dots x_n)=\frac{P(c)*P(x_1,x_2,\dots,x_n|c)}{P(x_1,x_2,\dots,x_n)}$$

在实际中，我们只关心分式中的分子部分，因为分母不依赖于 c 而且特征x_i的值是给定的，于是分母可以认为是一个常数。这样分母就等价于联合分布模型。

现在，“朴素”的条件独立假设开始发挥作用，假设每个特征x_i对于其他特征x_j，$i\neq j$是条件独立的。这就意味着$P(x_i|c,x_j)=P(x_i|c)$

对于$i\neq j$，所以联合分布模型可以表达为

$$P(c|x_1,x_2,\dots x_n)\propto P(c,x_1,x_2,\dots x_n)\propto P(c)P(x_1|c)P(x_2|c)\cdots P(x_n|c)\propto P(c)\prod_{i=1}^{n}P(x_i|c)$$

这样我们就 i 导出了独立分布特征模型，也就是朴素贝叶斯概率模型。朴素贝叶斯分类器包括了这种模型和相应的决策规则。一个普通的规则就是选出最有可能的那个：最大后验概率决策准则。相应的分类器便是如下定义的 classify 公式：

$$classify(x_1,\dots,x_n)=argmaxP(C=c)\prod_{i=1}^{n}P(X=x_i|C=c)$$

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过建立高斯朴素贝叶斯模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入乳腺癌数据集，数据划分及标准化处理与 logistics 回归处理相同
# 导入高斯朴素贝叶斯函数
from sklearn.naive_bayes import GaussianNB
model_bayes=GaussianNB()
# 模型训练及预测
model_bayes=model_bayes.fit(X_train,y_train)
y_pred=model_bayes.predict(X_test)
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("高斯朴素贝叶斯模型拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
#  导入高斯朴素贝叶斯函数
from  sklearn.naive_bayes  import  GaussianNB
model_bayes=GaussianNB()
#  模型训练及预测
model_bayes=model_bayes.fit(X_train,y_train)
y_pred=model_bayes.predict(X_test)
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred,y_test))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("高斯朴素贝叶斯模型拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           0       0.91      1.00      0.95        49
           1       1.00      0.95      0.97        94

    accuracy                           0.97       143
   macro avg       0.95      0.97      0.96       143
weighted avg       0.97      0.97      0.97       143

高斯朴素贝叶斯模型拟合的准确率为:96.503%
```

（七） 各模型比较

Logistic 回归和线性判别分析是紧密相连的。如果考虑二分类问题，p=1 只有一个预测变量的情况，设$p_1(x)$ *和* $p_2(x) = 1 - p_1(x)$分别为预测属于类 1 和类 2 的概率。在线性判别框架中，可以得到：

$$\log(\frac{p_1(x)}{1-p_1(x)}) = \log(\frac{p_1(x)}{p_2(x)}) = c_0 + c_1 x$$

其中c_0，c_1为μ_1，μ_2和σ^2的函数。而 Logistic 回归模型为

$$\log\left(\frac{p_1}{1-p_1}\right) = \beta_0 + \beta_1 x$$

由此可以看出，Logistic 回归和线性判别都产生一个线性决策边界。两种方法唯一的不同是β_0，β_1是由极大似然估计出来的，而c_0，c_1是通过估计的正态分布均值和方差计算出来的。

由于 Logistic 回归和线性判别分析只是在拟合过程中有一些差异，所以两者得到的结果应该是接近的。事实上，这一情况经常发生，但并非必然。线性判别分析假设观测服从每一类协方差矩阵都相同的高斯分布，所以当这个假设近似成立时，线性判别分析应该比 Logistic 回归能提供更好的结果。相反，如果高斯分布的假设不满足，Logistic 回归比线性判别效果更好。

而 KNN 与 Logistic 回归以及线性判别分析的原理完全不同。对于观测 X=x 进行预测，

首先确定 K 个距离 x 最近的训练观测，那么 X 分入这些观测中大多数所属的类别中。因此 KNN 是一个彻底的非参数方法：对决策边界的形状并没有做出任何假设。因此，当决策边界高度非线性时，用该方法会优于线性判别分析和 Logistic 回归以及。而另一方面，KNN 并没有给出哪些预测变量是重要的，所以也就不存在系数估计表。

最后，二次判别分析是非参数 KNN 方法和线性判别分析、Logistic 回归方法之间的一个折中的办法。因为二次判别分析得到一个二次的决策边界，所以它比线性方法的应用范围更广。虽然不如 KNN 光滑度高，但是二次判别分析在固定训练数据量的问题上一般比 KNN 有更好的效果，原因是二次判别分析对决策边界的形状做了一些假设。

三、 数据集划分及模型评价

（一） 留一法

1. 理论介绍

在训练机器学习模型之前，我们都会将数据集进行划分，其中训练集用来训练模型，测试集用来评估模型。假设数据集 D 中包含 m 个样本，如果将数据集划分为 K=m 个，每次用 m-1 个样本训练模型，1 个样本测试模型，这种划分数据集的方法就叫做留一法。

优点：① 留一法不受随机样本划分方式的影响；② 留一法使用的训练集比初始数据集相比只少了一个样本，这就使得在绝大多数情况下，留一法中被实际评估的模型与期望评估的用 D 训练出的模型很相似，因此，留一法的评估结果往往被认为比较准确。

缺点：① 在数据集比较大时，训练 m 个模型的计算开销可能是难以忍受的（例如数据集包含 100 万个样本，则需要训练 100 万个模型），而这还是在未考虑算法调参的情况下；② 留一法的估计结果也未必永远比其他评估方法准确。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过留一法来进行数据集的划分，并建立高斯朴素贝叶斯模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 使用 python 自带的乳腺癌数据集（二分类）
from sklearn.datasets import load_breast_cancer
X,y=load_breast_cancer(return_X_y=True)
print("样本的个数 = {}, 特征的个
数 = {}".format(X.shape[0],X.shape[1]))
# 采用留一法划分数据集
from sklearn.model_selection import LeaveOneOut
loo=LeaveOneOut()
Accuracy=[]
# loo.split():这里拆分的是 index
for train_index,test_index in loo.split(X):
```

```
        X_train,X_test=X[train_index],X[test_index]
        y_train,y_test=y[train_index],y[test_index]
        model_bayes=model_bayes.fit(X_train,y_train)
        y_pred=model_bayes.predict(X_test)
        # 模型评价
        from sklearn.metrics import accuracy_score
        Acc=accuracy_score(y_pred,y_test)
        Accuracy.append(Acc)
    print("Accuracy 的个数为：{}".format(np.array(Accuracy).shape))
    print("使用留一法划分数据集的高斯朴素贝叶斯模型拟合的平均准确率
    为:%.3f%%"% (np.mean(Accuracy) *100))
```

（3） 程序结果：

```
#  使用python自带的乳腺癌数据集（二分类）
from  sklearn.datasets  import  load_breast_cancer
X,y=load_breast_cancer(return_X_y=True)
print("样本的个数  =  {},  特征的个数  =  {}".format(X.shape[0],X.shape[1]))
#  采用留一法划分数据集
from  sklearn.model_selection  import  LeaveOneOut
loo=LeaveOneOut()
Accuracy=[]
#  loo.split():这里拆分的是index
for  train_index,test_index  in  loo.split(X):
        X_train,X_test=X[train_index],X[test_index]
        y_train,y_test=y[train_index],y[test_index]
        model_bayes=model_bayes.fit(X_train,y_train)
        y_pred=model_bayes.predict(X_test)
        #  模型评价
        from  sklearn.metrics  import  accuracy_score
        Acc=accuracy_score(y_pred,y_test)
        Accuracy.append(Acc)
print("Accuracy的个数为：{}".format(np.array(Accuracy).shape))
print("使用留一法划分数据集的高斯朴素贝叶斯模型拟合的平均准确率为:%.3f%%"%
        (np.mean(Accuracy)  *100))
```

```
样本的个数 = 569, 特征的个数 = 30
Accuracy的个数为：(569,)
使用留一法划分数据集的高斯朴素贝叶斯模型拟合的平均准确率为:93.849%
```

（二）K 折交叉验证

1. 理论介绍

“交叉验证法”先将数据集 D 划分为 k 个大小相似的互斥子集，即$D = D_1 \cup D_2 \cup \ldots \cup D_k$，$D_i \cap D_j = \emptyset \ (i \neq j)$。每个子集$D_i$都尽可能保持数据分布的一致性，即从 D 中通过分层采样得到。然后，每次用 k-1 个子集的并集作为训练集，余下的子集作为测试集；这样就可以获得 k 组训练/测试集，从而可以进行 k 次训练和测试，最终返回的是 k 个测试结果的均值。显然，交叉验证法评估结果的稳定性和保真性在很大程度上取决于 k 的取值，为了强调

这一点，通常把交叉验证法称为"k 折交叉验证"，k 通常取 10，因此又称为 10 折交叉验证。

优点：可以从有限的数据中尽可能挖掘多的信息，从各种角度去学习我们现有的有限的数据，避免出现局部的极值。在这个过程中无论是训练样本还是测试样本都得到了尽可能多的学习。

缺点：当数据集比较大时，训练模型的开销较大。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过 K 折交叉验证法来进行数据集的划分，然后建立高斯朴素贝叶斯模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 使用 python 自带的乳腺癌数据集（二分类）
from sklearn.datasets import load_breast_cancer
X,y=load_breast_cancer(return_X_y=True)
print("样本的个数 = {}, 特征的个
数 = {}".format(X.shape[0],X.shape[1]))
# 采用交叉验证法划分数据集
from sklearn.model_selection import cross_val_score
# 进行 10 折交叉验证训练及评价
scores=cross_val_score(model_bayes,X,y,cv=10)
Acc=np.mean(scores)
print("Accuracy 的个数为：{}".format(np.array(scores).shape))
print("使用十折交叉验证法划分数据集的高斯朴素贝叶斯模型拟合的平均准确率
为:%.3f%%"% (np.mean(Acc) *100))
```

（3） 程序结果：

```
# 使用python自带的乳腺癌数据集（二分类）
from sklearn.datasets import load_breast_cancer
X,y=load_breast_cancer(return_X_y=True)
print("样本的个数 = {}, 特征的个数 = {}".format(X.shape[0],X.shape[1]))
# 采用交叉验证法划分数据集
from sklearn.model_selection import cross_val_score
# 进行10折交叉验证训练及评价
scores=cross_val_score(model_bayes,X,y,cv=10)
Acc=np.mean(scores)
print("Accuracy的个数为：{}".format(np.array(scores).shape))
print("使用十折交叉验证法划分数据集的高斯朴素贝叶斯模型拟合的平均准确率为:%.3f%%"%
      (np.mean(Acc) *100))
```

```
样本的个数 = 569, 特征的个数 = 30
Accuracy的个数为：(10,)
使用十折交叉验证法划分数据集的高斯朴素贝叶斯模型拟合的平均准确率为:93.678%
```

（三） 自助法

1. 理论介绍

我们希望评估的是用 D 训练出的模型，但在留出法和交叉验证法中，由于保留了一部分样本用于测试，因此实际评估的模型所使用的训练集比 D 小，这必然会引入一些因训练样本规模不同而导致的估计偏差。留一法受训练样本规模变化的影响较小，但计算复杂度又太高了。

“自助法”是一个比较好的解决方案，它直接以自助采样法为基础。给定包含 m 个样本的数据集 D，我们对它进行采样产生数据集 D′：每次随机从 D 中挑选一个样本，将其拷贝到 D′，再将样本放回原始数据集 D，使其在下次采样时仍有可能被采到，这个过程执行 m 次后，我们就得到了包含 m 个样本的数据集 D′，这就是自助采样的结果。显然，D 中有一部分样本会在 D′中多次出现，而另一部分样本不出现。可以做一个简单的估计，样本在 m 次采样中始终不被采到的概率是$(1-\frac{1}{m})^m$，取极限得到：

$$\lim_{n\to\infty}(1-\frac{1}{m})^m \to \frac{1}{e} \approx 0.368$$

通过自助采样，初始数据集 D 中约有 36.8%的样本未出现在采样集 D′里。于是我们可将 D′用作训练集，D\D′用作测试集。这样，实际评估的模型与期望评估的模型都是使用 m 个样本，而我们仍有数据总量约 1/3 的没在训练集出现过的样本用于测试。

优点：① 自助法在数据集较小，难以划分训练/测试集时很有用；② 自助法能从初始数据集中产生多个不同的训练集，这对集成学习等方法有很大的好处。

缺点：自助法产生的数据集改变了初始数据集的分布，这会引入估计偏差。因此，在初始数据量足够时，留出法和交叉验证法更常用一些。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过自助法来进行数据集的划分，然后建立高斯朴素贝叶斯模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 使用 python 自带的乳腺癌数据集（二分类）
from sklearn.datasets import load_breast_cancer
X,y=load_breast_cancer(return_X_y=True)
print("样本的个数 = {}, 特征的个
数 = {}".format(X.shape[0],X.shape[1]))
# 采用自助法划分数据集(可重复抽样)
X_train=pd.DataFrame(X).sample(frac=1.0,replace=True)
X_test=pd.DataFrame(X).loc[pd.DataFrame(X).index.difference(X_
train.index)].copy()
y_train=y[X_train.index]
y_test=y[X_test.index]
# 模型训练及预测
model_bayes=model_bayes.fit(np.array(X_train),y_train)
```

```
y_pred=model_bayes.predict(np.array(X_test))
# 模型评价
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("使用自助法划分数据集的高斯朴素贝叶斯模型拟合的平均准确率
为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
# 使用python自带的乳腺癌数据集（二分类）
from sklearn.datasets import load_breast_cancer
X,y=load_breast_cancer(return_X_y=True)
print("样本的个数 = {}, 特征的个数 = {}".format(X.shape[0],X.shape[1]))
# 采用自助法划分数据集(可重复抽样)
X_train=pd.DataFrame(X).sample(frac=1.0,replace=True)
X_test=pd.DataFrame(X).loc[pd.DataFrame(X).index.difference(X_train.index)].copy()
y_train=y[X_train.index]
y_test=y[X_test.index]
# 模型训练及预测
model_bayes=model_bayes.fit(np.array(X_train),y_train)
y_pred=model_bayes.predict(np.array(X_test))
# 模型评价
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("使用自助法划分数据集的高斯朴素贝叶斯模型拟合的平均准确率为:%.3f%%"% (Acc *100))
```

```
样本的个数 = 569, 特征的个数 = 30
使用自助法划分数据集的高斯朴素贝叶斯模型拟合的平均准确率为:93.897%
```

（四） ROC 与 AUC

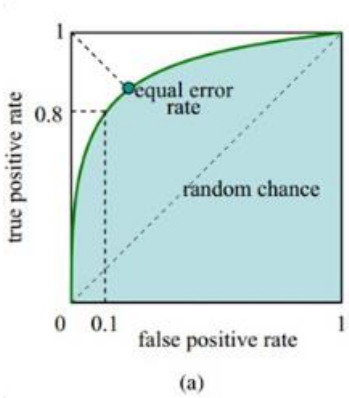
1. 理论介绍

ROC 全称时“受试者工作特征”曲线，它源于“二战”中用于敌机检测的雷达信号分析技术，二十世纪六七十年代开始被用于一些心理学、医学检测应用中，此后被引入机器学习领域。与 P-R 曲线相似，我们根据学习器的预测结果对样例进行排序，按此顺序逐个把样本作为正例进行预测，每次计算出两个重要量的值，分别以它们为横、纵坐标作图，就得到了"ROC 曲线”。与 P-R 曲线使用查准率、查全率为纵、横轴不同，ROC 曲线的纵轴是“真正例率”(True Positive Rate,简称 TPR),横轴是“假正例率” （False Positive Rate，简称 FPR).

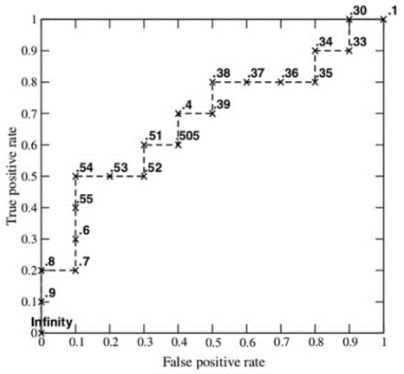
两者分别定义为

$$\mathrm{TPR}=\frac{TP}{TP+FN} \quad , \quad \mathrm{FPR}=\frac{FP}{TN+FP}$$

显示 ROC 曲线的图称为“ROC 图”。下图给出了一个示例图，显然，对角线对应于“随机猜测”模型，而点（0，1）则对应于将所有正例排在所有反例之前的“理想模型”。



(a)



现实任务中通常是利用有限个试样例来绘制 ROC 图，此时仅能获得有限个（真正例率，假正例率）坐标对，无法产生光滑 ROC 曲线，只能绘制出如上图右所示的近似 ROC 曲线。

进行学习器的比较时，与 P-R 图相似，若一个学习器的 ROC 曲线被另一个学习器的曲线完全“包住”，则可断言后者的性能优于前者；若两个学习器的 ROC 线发生交叉，则难以一般性地断言两者孰优孰劣，此时如果一定要进行比较，则较为合理的判据是比较 ROC 曲线下的面积，即 AUC。

从定义可知，AUC 可通过对 ROC 曲线下各部分的面积求和而得．假定 ROC 曲线是由坐标为$\{(x_1,y_1),(x_2,y_2),\ldots,(x_m,y_m)\}$的点按序连接而形成 (x1=0,xm=1)，则 AUC 可估算为

$$\mathrm{AUC}=\frac{1}{2}\sum_{i=1}^{m-1}(x_{i+1}-x_i)\cdot(y_i+y_{i+1})$$

形式化地看，AUC 考虑的是样本预测的排序质量，因此它与排序误差有紧密联系。给定 m+ 个正例 m−个反例，令 D+和 D−分别表示正、反例集合，则排序“损失”（loss）定义为

$$\mathrm{l_{rank}}=\frac{1}{m^+m^-}\sum_{x^+\in D^+}\sum_{x^-\in D^-}\left(I\big(f(\boldsymbol{x}^+)<f(\boldsymbol{x}^-)\big)+\frac{1}{2}I\big(f(\boldsymbol{x}^+)=f(\boldsymbol{x}^-)\big)\right)$$

即考虑每一对正、反例，若正例的预值小于反例，则记一个“罚分”；若相等，则记 0.5 个“罚分”。容易看出，ℓ_{rank}对应的是 ROC 曲线之上的面积：若一个正例在 ROC 线上对应标记点的坐标为 (x,y)，则 x 恰是排序在其之前的反例所占的比例，即假正例率。因此有

$$\mathrm{AUC}=1-\ell_{\mathrm{rank}}$$

2. 案例实现

（1） 数据来源及解释

本案例通过 Python 产生了一个二分类数据集，该数据集有 20 征变量，1 个二分类的目标变量，希望通过 ROC 和 AUC 曲线来评价 Logistics 回归、Probit 回归、LDA、QDA、KNN、Gauss Navie Bayes 模型在该数据集上的表现效果。

（2） 代码实现：

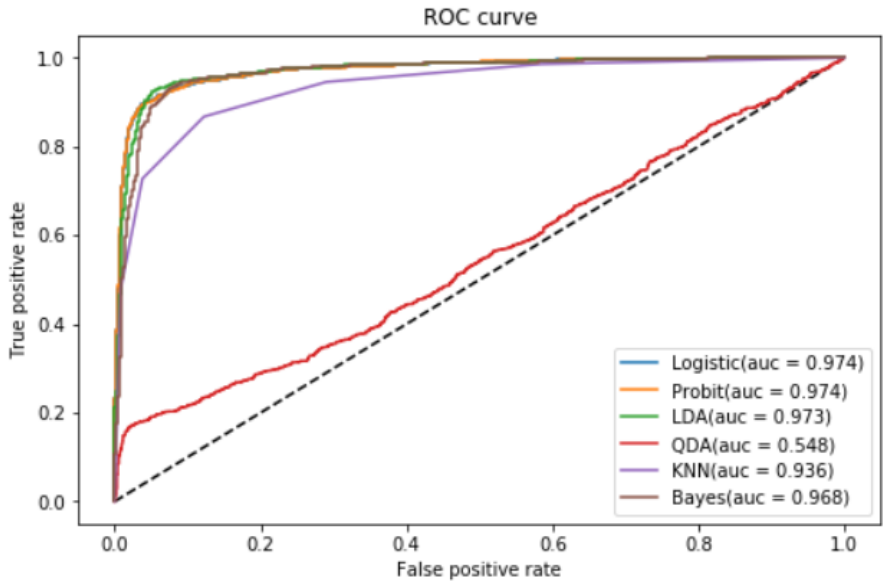
```
# 产生一个二分类数据集，n_samples=10000
import numpy as np
```

```
np.random.seed(123)
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=10000)
print(X.shape)
X_train, X_test, y_train, y_test = train_test_split(X, y, test
_size=0.25)
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
# logistic 回归
model_logistic=model_logistic.fit(X_train,y_train)
y_pred_logistic=model_logistic.predict_proba(X_test)[:,1]
fpr_logistic, tpr_logistic,_= roc_curve(y_test, y_pred_logisti
c, pos_label=1)
area_logistic=auc(fpr_logistic, tpr_logistic)
# probit 回归
model = Probit(y_train, X_train)
model_probit = model.fit()
y_pred_probit=model_probit.predict(X_test)
fpr_probit, tpr_probit,_= roc_curve(y_test, y_pred_probit, pos
_label=1)
area_probit=auc(fpr_probit, tpr_probit)
# LDA
model_lda=model_lda.fit(X_train,y_train)
y_pred_lda=model_lda.predict_proba(X_test)[:,1]
fpr_lda, tpr_lda,_= roc_curve(y_test, y_pred_lda, pos_label=1)
area_lda=auc(fpr_lda, tpr_lda)
# QDA
model_qda=model_qda.fit(X_train,y_train)
y_pred_qda=model_qda.predict_proba(X_test)[:,1]
fpr_qda, tpr_qda,_= roc_curve(y_test, y_pred_qda, pos_label=1)
area_qda=auc(fpr_qda, tpr_qda)
# KNN
model_knn=model_knn.fit(X_train,y_train)
y_pred_knn=model_knn.predict_proba(X_test)[:,1]
fpr_knn, tpr_knn,_= roc_curve(y_test, y_pred_knn, pos_label=1)
area_knn=auc(fpr_knn, tpr_knn)
# Gauss Navie Bayes
model_bayes=model_bayes.fit(X_train,y_train)
y_pred_bayes=model_bayes.predict_proba(X_test)[:,1]
fpr_bayes, tpr_bayes,_= roc_curve(y_test, y_pred_bayes, pos_la
bel=1)
area_bayes=auc(fpr_bayes, tpr_bayes)
# 画图
fig=plt.figure(figsize=(8,5))
```

```
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr_logistic,tpr_logistic,label="Logistic(auc = %0.3f
)" % area_logistic)
plt.plot(fpr_probit,tpr_probit,label="Probit(auc = %0.3f)" % a
rea_probit)
plt.plot(fpr_lda,tpr_lda,label="LDA(auc = %0.3f)" % area_lda)
plt.plot(fpr_qda,tpr_qda,label="QDA(auc = %0.3f)" % area_qda)
plt.plot(fpr_knn,tpr_knn,label="KNN(auc = %0.3f)" % area_knn)
plt.plot(fpr_bayes,tpr_bayes,label="Bayes(auc = %0.3f)" % area
_bayes)
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve')
plt.legend(loc='best')
plt.show()
```

（3） 程序结果：

```
fig=plt.figure(figsize=(8,5))
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr_logistic, tpr_logistic, label="Logistic(auc = %0.3f)" % area_logistic)
plt.plot(fpr_probit, tpr_probit, label="Probit(auc = %0.3f)" % area_probit)
plt.plot(fpr_lda, tpr_lda, label="LDA(auc = %0.3f)" % area_lda)
plt.plot(fpr_qda, tpr_qda, label="QDA(auc = %0.3f)" % area_qda)
plt.plot(fpr_knn, tpr_knn, label="KNN(auc = %0.3f)" % area_knn)
plt.plot(fpr_bayes, tpr_bayes, label="Bayes(auc = %0.3f)" % area_bayes)
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve')
plt.legend(loc='best')
plt.show()
```



四、 特征选择及降维

（一） 子集选择

1. 理论介绍

最优子集选择，即对 p 个预测变量的所有可能组合分别使用最小二乘回归进行拟合。对含有一个预测变量的模型，拟合 p 个模型；对含有两个预测变量的模型，拟合$\binom{p}{2} = p(p-1)/2$个模型，依次类推。最后在所有可能模型中选取一个最优模型。

在(2^p)个模型中进行最优模型选择的过程通常可以分解为两个阶段，具体过程见算法：

(1) 记不含预测变量的零模型为M_0，只用于估计个观测的样本均值。

(2) 对于 k=1，2，…，p：

a) 拟合$\binom{p}{k}$个包含 k 个预测变量的模型；

b) 在$\binom{p}{k}$个模型中选择 RSS 最小或R^2最大的作为最优模型，记为M_k

(3) 根据交叉验证预测误差、$C_p(AIC)$、BIC 或者调整R^2从$M_0, M_1, \cdots, M_p$个模型中选择出一个最优模型。

在算法中，步骤 2 先在不同的子集规模下，进行模型选择（基于训练样本集），将从2^p个模型中选择一个模型的问题转化为从 p+1 个备选模型中选择一个模型的问题。显然，在变量数目给定的情况下，这些备选模型的残差平方和最小、R^2最大。

接下来，从这 p+1 个模型中选择出一个最优模型。随着纳入模型的特征数目的增加，这 p+1 个模型的 RSS 单调下降，R^2单调增加。如果只使用这些统计量进行模型选择，最终选出的模型将包含所有变量。低 RSS 及高R^2表明模型的训练误差低，但是我们的目标是选择一个测试误差低的模型，所以我们在步骤 3 应使用交叉验证预测误差、$C_p(AIC)$、BIC 或者调整R^2从$M_0, M_1, \cdots, M_p$个模型中选择出一个最优模型。

优点：简单直观

缺点： ① 计算效率不高，随着变量数 p 的增加，可选模型的数量也在迅速增加；② 只对最小二乘线性回归模型有效。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的波士顿房价数据集，该数据集有 13 个特征变量，1 个连续型目标变量，希望通过最优子集选择的方法，来寻找得到该数据集的最优子集。。

（2） 代码实现：

```
import numpy as np
from scipy import linalg
from scipy.stats import norm
import random

def turnbits_rec(p):
    if (p == 1):
```

```
        return np.array([[True, False], [True, True]])
    else:
        tmp1 = np.c_[turnbits_rec(p - 1),
                     np.array([False] * (2**(p - 1))).reshape(
(2**(p - 1), 1))]
        tmp2 = np.c_[turnbits_rec(p - 1),
                     np.array([True] * (2**(p - 1))).reshape((
2**(p - 1), 1))]
        return np.r_[tmp1, tmp2]

def mse(xtx_t, xty_t, beta):
    return (np.sum(np.dot(xtx_t, beta) * beta) - 2 * np.sum(xt
y_t * beta))

def solve_sym(xtx, xty):
    L = linalg.cholesky(xtx)
    return linalg.lapack.dpotrs(L, xty)[0]

# 自定义最优子集函数
class BestSubsetReg(object):
    def __init__(self, x, y, inter=True, isCp=True, isAIC=True
, isCV=True):
        self.__n, self.__p = x.shape
        if inter:
            self.__x = np.c_[np.ones((self.__n, 1)), x]
            self.__ind_var = turnbits_rec(self.__p)
        else:
            self.__x = x
            self.__ind_var = turnbits_rec(self.__p)[:, 1:]
        self.__y = y
        self.__xTx = np.dot(self.__x.T, self.__x)
        self.__xTy = np.dot(self.__x.T, self.__y)
        self.__b = [
            solve_sym(self.__xTx[ind][:, ind], self.__xTy[ind]
)
            for ind in self.__ind_var
        ]
        self.__isCp = isCp
        self.__isAIC = isAIC
        self.__isCV = isCV

    def __Cp_AIC(self):
        rss = np.dot(self.__y, self.__y) - [
            np.sum(np.dot(self.__xTx[ind][:, ind], b_) * b_)
```

```
            for ind, b_ in zip(self.__ind_var, self.__b)
        ]
        d = np.sum(self.__ind_var, axis=1)
        if self.__isCp:
            self.Cp = rss + 2 * d * rss[-
1] / (self.__n - self.__p - 1)
        if self.__isAIC:
            self.AIC = self.__n * np.log(rss) + 2 * d

    def __cvreg(self):
        K = 10
        indexs = np.array_split(np.random.permutation(np.arang
e(0, self.__n)), K)

        def cvk(ind, index):
            txx = self.__xTx[ind][:, ind] - np.dot(
                (self.__x[index][:, ind]).T, self.__x[index][:
, ind])
            txy = self.__xTy[ind] - np.dot(
                (self.__x[index][:, ind]).T, self.__y[index])
            tcoe = solve_sym(txx, txy)
            return np.sum(
                (self.__y[index] - np.dot(self.__x[index][:, i
nd], tcoe))**2)

        self.cverr = np.sum(np.array([[cvk(ind, index) for ind
ex in indexs]
                                      for ind in self.__ind_va
r]),
                            axis=1) / self.__n

    def output(self, isPrint=True):
        """
        If inter=True, first item is intercept, Otherwise it i
s X1.
        If print=False, save results only and do not print.
        """
        if self.__isCp | self.__isAIC:
            self.__Cp_AIC()
            if self.__isCp:
                min_id = np.argmin(self.Cp)
                self.Cp = [self.__ind_var[min_id][0:], self.__
b[min_id]]
                if isPrint:
```

```
                    print("Cp: \nVariable: ", self.Cp[0])
                    print("Coefficient: ", self.Cp[1])
            if self.__isAIC:
                min_id = np.argmin(self.AIC)
                self.AIC = [self.__ind_var[min_id][0:], self._
_b[min_id]]
                if isPrint:
                    print("AIC: \nVariable: ", self.AIC[0])
                    print("Coefficient: ", self.AIC[1])
        if self.__isCV:
            self.__cvreg()
            min_id = np.argmin(self.cverr)
            self.cverr = [self.__ind_var[min_id][0:], self.__b
[min_id]]
            if isPrint:
                print("Cross Validation: \nVariable:
", self.cverr[0])
                print("Coefficient: ", self.cverr[1])
# 使用 python 自带的波士顿房价数据集
from sklearn.datasets import load_boston
X,y=load_boston(return_X_y=True)
print("样本的个数 = {},特征的个
数 = {}".format(X.shape[0],X.shape[1]))
# 将数据进行标准化
from sklearn.preprocessing import StandardScaler
scaler=StandardScaler().fit(X)
X=scaler.transform(X)
# 导入最优子集回归模型并进行训练
model_subset = BestSubsetReg(x=X, y=y)
# 输出子集选择结果
model_subset.output()
model_subset.Cp
```

（3） 程序结果：

```
#  使用python自带的波士顿房价数据集
from  sklearn.datasets  import  load_boston
X,y=load_boston(return_X_y=True)
print("样本的个数  =  {},特征的个数  =  {}".format(X.shape[0],X.shape[1]))
#  将数据进行标准化
from  sklearn.preprocessing  import  StandardScaler
scaler=StandardScaler().fit(X)
X=scaler.transform(X)
#  导入最优子集回归模型并进行训练
model_subset  =  BestSubsetReg(x=X,  y=y)
#  输出子集选择结果
model_subset.output()
model_subset.Cp
```

```
样本的个数 = 506,特征的个数 = 13
Cp:
Variable:  [ True  True  True False  True  True  True False  True  True  True  True
  True  True]
Coefficient:  [22.53280632 -0.93160036  1.06815915  0.68985505 -2.01150261  2.66841378
 -3.14011016  2.60618941 -1.98306322 -2.04714825  0.84736785 -3.72789722]
AIC:
Variable:  [ True  True  True False  True  True  True False  True  True  True  True
  True  True]
Coefficient:  [22.53280632 -0.93160036  1.06815915  0.68985505 -2.01150261  2.66841378
 -3.14011016  2.60618941 -1.98306322 -2.04714825  0.84736785 -3.72789722]
Cross Validation:
Variable:  [ True  True  True False  True  True  True False  True  True  True  True
  True  True]
Coefficient:  [22.53280632 -0.93160036  1.06815915  0.68985505 -2.01150261  2.66841378
 -3.14011016  2.60618941 -1.98306322 -2.04714825  0.84736785 -3.72789722]
[array([ True,  True,  True, False,  True,  True,  True, False,  True,
         True,  True,  True,  True,  True]),
 array([22.53280632, -0.93160036,  1.06815915,  0.68985505, -2.01150261,
         2.66841378, -3.14011016,  2.60618941, -1.98306322, -2.04714825,
         0.84736785, -3.72789722])]
```

（二） 岭回归

1. 理论介绍

在进行最小二乘回归估计时，常常会出现过拟合的情况，为了避免过拟合，我们引入一个正则项$\lambda 2$，故岭回归的损失函数变成下式：

$$\sum_{i=1}^{n}(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij})^2+\lambda\sum_{j=1}^{p}\beta_j^2=RSS+\lambda\sum_{j=1}^{p}\beta_j^2$$

其中$\lambda>0$是一个调节参数，将单独确定。上式是两个不同标准的权衡。与最小二乘相同，岭回归通过最小化 RSS 寻求能较好地拟合数据的估计量。此外，另一项$\lambda\sum_{j=1}^{p}\beta_j^2$，称为压缩惩罚，当$\beta_1,\cdots,\beta_p$接近零时比较小，因此具有将$\beta_j$估计值往零的方向进行压缩的作用。调节参数$\lambda$的作用是控制这两项对回归系数估计的相对影响程度。当$\lambda=0$时，惩罚项不产生作用，岭回归与最小二乘估计结果相同。随着$\lambda\to\infty$，压缩惩罚项的影响力增加，岭回归系数估计值越来越接近零。与最小二乘得到唯一的估计结果不同，岭回归得到的系数估计结果$\hat{\beta}^R$随λ的变化而变化。选择合适的λ值十分重要。

在上式中对β_1，…，β_p施加了压缩惩罚，但未对常数项β_0进行惩罚。我们要缩减与响应变量存在关联关系的预测变量的系数但是并不缩减截距项，因为截距项用于测量当$x_{i1} = x_{i2} = \cdots = x_{ip} = 0$时响应变量的均值。

岭回归与最小二乘回归的比较：① 最小二乘回归的系数估计尺度是不变的，但是岭回归系数估计值可能发生显著变化。所以在使用岭回归前，需要对数据进行标准化；② 岭回归可以提升最小二乘回归的效果，因为岭回归的优势在于它综合权衡了误差与方差。随着 λ 的增加，岭回归拟合结果的光滑度降低，虽然方差降低，但是偏差在增加。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的波士顿房价数据集，该数据集有 13 个特征变量，1 个连续型目标变量，希望通过建立 Ridge 回归来拟合特征变量与目标变量之间的关系。

（2） 代码实现：

```
# 使用 python 自带的波士顿房价数据集
from sklearn.datasets import load_boston
X,y=load_boston(return_X_y=True)
print("样本的个数 = {},特征的个
数 = {}".format(X.shape[0],X.shape[1]))
# 采用留出法划分数据集
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0
.25,random_state=123)
# 将数据进行标准化
from sklearn.preprocessing import StandardScaler
scaler=StandardScaler().fit(X_train)
X_train=scaler.transform(X_train)
X_test=scaler.transform(X_test)
# 导入岭回归函数
from sklearn.linear_model import Ridge
model_ridge=Ridge(alpha=0.5)  # 模型实例化
# 模型训练及预测
model_ridge=model_ridge.fit(X_train,y_train)
y_pred=model_ridge.predict(X_test)
# 模型评价：采用 MSE 和 R^2 来进行评价
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
MSE=mean_squared_error(y_pred,y_test)
r2=r2_score(y_pred,y_test)
print("岭回归模型拟合的均方误差为:{}".format(round(MSE,3)))
print("岭回归可以解释原变量%.3f%%的信息"% (r2 *100))
# 查看各变量的系数
print("各变量的系数为:\n",model_ridge.coef_)
```

（3） 程序结果：

```
#  导入岭回归函数
from  sklearn.linear_model  import  Ridge
model_ridge=Ridge(alpha=0.5)    #  模型实例化
#  模型训练及预测
model_ridge=model_ridge.fit(X_train,y_train)
y_pred=model_ridge.predict(X_test)
#  模型评价：采用MSE和R^2来进行评价
from  sklearn.metrics  import  mean_squared_error
from  sklearn.metrics  import  r2_score
MSE=mean_squared_error(y_pred,y_test)
r2=r2_score(y_pred,y_test)
print("岭回归模型拟合的均方误差为:{}".format(round(MSE,3)))
print("岭回归可以解释原变量%.3f%%的信息"%  (r2  *100))
#  查看各变量的系数
print("各变量的系数为:\n",model_ridge.coef_)
```

```
岭回归模型拟合的均方误差为:24.77
岭回归可以解释原变量60.464%的信息
各变量的系数为:
 [-0.87661291  0.94946098  0.38801763  0.2961452  -1.7725836   3.10616001
 -0.01613174 -2.88171978  2.3796612  -2.05736629 -2.0040423   0.60486402
 -3.92557963]
```

（三） Lasso 回归

1. 理论介绍

岭回归产生的模型包含全部的 p 个变量，增加λ的值会减小系数绝对值，但是依然无法剔除任何变量。而 Lasso 可以用于克服岭回归的缺点，Lasso 的系数$\hat{\beta}_\lambda^L$通过求解下式的最小值得到：

$$\sum_{i=1}^{n}(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij})^2+\lambda\sum_{j=1}^{p}|\beta_j|=RSS+\lambda\sum_{j=1}^{p}|\beta_j|$$

可以看出 Lasso 与岭回归具有非常相似的拟合公式。唯一的区别是岭回归公式中的β_j^2项在 Lasso 中被替换为$|\beta_j|$。用统计的说法，Lasso 采用l_1惩罚项，而不采用l_2惩罚项。

与岭回归相同，Lasso 也将系数估计值往 0 的方向进行缩减。然而，当调节参数λ足够大时，l_1惩罚项具有将其中某些系数的估计值强制设定为 0 的作用。因此，与最优子集选择方法相似，Lasso 也完成了变量选择。故而 Lasso 建立的模型与岭回归建立的模型相比更易于解释。所以说 Lasso 得到了稀疏模型——只包含所有变量的一个子集的模型。同样，选择一个合适的λ值对 Lasso 也十分重要。

Lasso 回归与岭回归对比：① Lasso 回归较岭回归有较大的优势，因为它可以得到只包含部分变量的简单易解释模型。一般情况下，当一小部分预测变量是真实有效的而其他预测变量系数非常小或者等于零时，Lasso 要更为出色；当响应变量是很多预测变量的函数并且这些变量系数大致相等时，岭回归较为出色。但是对于一个真实的数据集，与响应变量有关

的变量个数无法事先知道。因此可以使用像交叉验证这样的方法，来决定哪个方法更适合特定数据集。② 岭回归与 Lasso 回归表现出两种不同的系数压缩方式。岭回归中每个最小二乘系数以相同比例压缩。相比而言，Lasso 回归中每个最小二乘系数以 λ/2 为阈值压缩至零，即那些绝对值小于 λ/2 的系数被完全压缩至零。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的波士顿房价数据集，该数据集有 13 个特征变量，1 个连续型目标变量，希望通过建立 Lasso 回归来拟合特征变量与目标变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入波士顿房价数据集，数据划分及数据标准化与岭回归相同
# 导入 Lasso 回归函数
from sklearn.linear_model import Lasso
model_lasso=Lasso(alpha=0.1)  # 模型实例化
# 模型训练及预测
model_lasso=model_lasso.fit(X_train,y_train)
y_pred=model_lasso.predict(X_test)
# 模型评价：采用 MSE 和 R^2 来进行评价
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
MSE=mean_squared_error(y_pred,y_test)
r2=r2_score(y_pred,y_test)
print("Lasso 回归模型拟合的均方误差为:{}".format(round(MSE,3)))
print("Lasso 回归可以解释原变量%.3f%%的信息"% (r2 *100))
# 查看各变量的系数
print("各变量的系数为:\n",model_lasso.coef_)
```

（3） 程序结果：

```
#  导入Lasso回归函数
from  sklearn.linear_model  import  Lasso
model_lasso=Lasso(alpha=0.1)     #  模型实例化
#  模型训练及预测
model_lasso=model_lasso.fit(X_train,y_train)
y_pred=model_lasso.predict(X_test)
#  模型评价：采用MSE和R^2来进行评价
from  sklearn.metrics  import  mean_squared_error
from  sklearn.metrics  import  r2_score
MSE=mean_squared_error(y_pred,y_test)
r2=r2_score(y_pred,y_test)
print("Lasso回归模型拟合的均方误差为:{}".format(round(MSE,3)))
print("Lasso回归可以解释原变量%.3f%%的信息"%  (r2  *100))
#  查看各变量的系数
print("各变量的系数为:\n",model_lasso.coef_)
```

```
Lasso回归模型拟合的均方误差为:26.114
Lasso回归可以解释原变量56.578%的信息
各变量的系数为:
 [-0.59635319  0.61150794 -0.          0.26879248 -1.2058338   3.22791697
 -0.         -2.25261646  0.95094981 -0.80891399 -1.83453419  0.47667922
 -3.93031518]
```

（四） 弹性网

1. 理论介绍

岭回归和 Lasso 回归是经常使用的学习方法，但是在实际应用中，这两个方法都会受到一定程度的限制。当参数个数比样本个数还要多的时候，l_1约束的最小二乘学习法的非零参数个数最大为 n。那么此时就会产生一定的局限性,解决上述问题的方案，就是使用弹性网进行解决，其数学表达式为：

$$\hat{\beta} = arg\min_{\beta}\{\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij})^2 + (1-\lambda)\sum_{j=1}^{p}|\beta_j| + \lambda\sum_{j=1}^{p}\beta_j^2\}$$

其中λ是满足$0 \le \lambda \le 1$的标量。当$\lambda = 0$时，弹性网就变为 Lasso；当$\lambda = 1$时，弹性网就变为岭回归；当$0 \le \lambda \le 1$时，约束$(1-\lambda)\sum_{j=1}^{p}|\beta_j| + \lambda\sum_{j=1}^{p}\beta_j^2 \le s$在参数坐标轴上保持尖形。因此，弹性网回归具有 Lasso 回归和岭回归的优点，既能达到变量选择的目的，又具有很好的群组效应，可以很好的处理多重共线性。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的波士顿房价数据集，该数据集有 13 个特征变量，1 个连续型目标变量，希望通过建立弹性网模型来拟合特征变量与目标变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入波士顿房价数据集，数据划分及数据标准化与岭回归相同
```

```
# 导入弹性网函数
from sklearn.linear_model import ElasticNet
model_elastic=ElasticNet()  # 模型实例化
# 模型训练及预测
model_elastic=model_elastic.fit(X_train,y_train)
y_pred=model_elastic.predict(X_test)
# 模型评价：采用 MSE 和 R^2 来进行评价
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
MSE=mean_squared_error(y_pred,y_test)
r2=r2_score(y_pred,y_test)
print("Elastic_Net 模型拟合的均方误差为:{}".format(round(MSE,3)))
print("Elastic_Net 可以解释原变量%.3f%%的信息"% (r2 *100))
# 查看各变量的系数
print("各变量的系数为:\n",model_elastic.coef_)
```

（3） 程序结果：

```
#  导入弹性网函数
from  sklearn.linear_model  import  ElasticNet
model_elastic=ElasticNet()    #  模型实例化
#  模型训练及预测
model_elastic=model_elastic.fit(X_train,y_train)
y_pred=model_elastic.predict(X_test)
#  模型评价：采用MSE和R^2来进行评价
from  sklearn.metrics  import  mean_squared_error
from  sklearn.metrics  import  r2_score
MSE=mean_squared_error(y_pred,y_test)
r2=r2_score(y_pred,y_test)
print("Elastic_Net模型拟合的均方误差为:{}".format(round(MSE,3)))
print("Elastic_Net可以解释原变量%.3f%%的信息"%  (r2  *100))
#  查看各变量的系数
print("各变量的系数为:\n",model_elastic.coef_)
```

```
Elastic_Net模型拟合的均方误差为:31.899
Elastic_Net可以解释原变量3.574%的信息
各变量的系数为:
 [-0.36754177  0.04144601 -0.26162353  0.09944001 -0.19657701  2.57774603
 -0.          -0.          -0.          -0.39236921 -1.25194015  0.31277891
 -2.3950601 ]
```

（五） 自适应 Lasso

1. 理论介绍

Adaptive Lasso 是 Oracle 方法，Oracle 方法在最初提出来得时候是用来判断一个模型方法的好坏。Lasso 在某些条件下不满足变量选择的相合性，Lasso 不算一个 Oracle 方法。而 Adaptive Lasso 当权重被合适的选择时，具有 Oracle 性质。

任选一个 v>0,定义权重向量$\hat{w}=\frac{1}{|\hat{\beta}|^{v^*}}$，自适应 Lasso 系数$\beta^{*(n)}$如下给出：

$$\hat{\beta}^{*(n)} = \arg\min_{\beta}(y - \sum_{j=1}^{p} x_j\beta_j)^2 + \lambda_n \sum_{j=1}^{p} \widehat{w}_j|\beta_j|$$

这里的自适应指的是权重本身是 data-dependent 的。当我们选择合适的λ_n时，自适应 Lasso 就会有 Oracle 性质。

假设$\frac{\lambda_n}{\sqrt{n}} \to 0$，$\lambda_n n^{\frac{v-1}{2}} \to \infty$，自适应 Lasso 的系数估计值满足下列性质：

（1）变量选择的相合性

（2）渐进正态性

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的波士顿房价数据集，该数据集有 13 个特征变量，1 个连续型目标变量，希望通过建立自适应 Lasso 模型来拟合特征变量与目标变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入波士顿房价数据集，数据划分及数据标准化与岭回归相同
# 自定义 Adaptive Lasso 模型
def Adaptive_LASSO(X_train,y_train,max_iterations = 1000,lasso_iterations = 10, alpha = 0.1, tol = 0.001, max_error_up = 5, title = ''):

    from sklearn.linear_model import Lasso
    # set checks
    higher  = float('inf')
    lower   = 0

    # set lists
    coefficients_list = []
    iterations_list   = []

    # set variables
    X_train  = X_train
    y_train  = y_train

    # set constants
    alpha    = alpha
    tol      = tol
    max_iter = max_iterations
    n_lasso_iterations = lasso_iterations

    g = lambda w: np.sqrt(np.abs(w))
```

```
    gprime = lambda w: 1. / (2. * np.sqrt(np.abs(w)) + np.finf
o(float).eps)

    n_samples, n_features = X_train.shape
    p_obj = lambda w: 1. / (2 * n_samples) * np.sum((y_train -
 np.dot(X_train, w)) ** 2) \
                      + alpha * np.sum(g(w))

    weights = np.ones(n_features)

    X_w = X_train / weights[np.newaxis, :]
    X_w  = np.nan_to_num(X_w)
    X_w  = np.round(X_w,decimals = 3)

    y_train    = np.nan_to_num(y_train)

    adaptive_lasso = Lasso(alpha=alpha, fit_intercept=False)

    adaptive_lasso.fit(X_w, y_train)

    for k in range(n_lasso_iterations):
        X_w = X_train / weights[np.newaxis, :]
        adaptive_lasso = Lasso(alpha=alpha, fit_intercept=Fals
e)
        adaptive_lasso.fit(X_w, y_train)
        coef_ = adaptive_lasso.coef_ / weights
        weights = gprime(coef_)

        print ('Iteration #',k+1,':   ',p_obj(coef_))  # shoul
d go down

        iterations_list.append(k)
        coefficients_list.append(p_obj(coef_))

    print (np.mean((adaptive_lasso.coef_ != 0.0) == (coef_ !=
0.0)))

    coef = pd.Series(adaptive_lasso.coef_, index = X_train.col
umns)
    print('==============================================')
    print("Adaptive LASSO picked " + str(sum(coef != 0)) + " v
ariables and eliminated the other " +  str(sum(coef == 0)) + "
 variables.")
    print('==============================================')
```

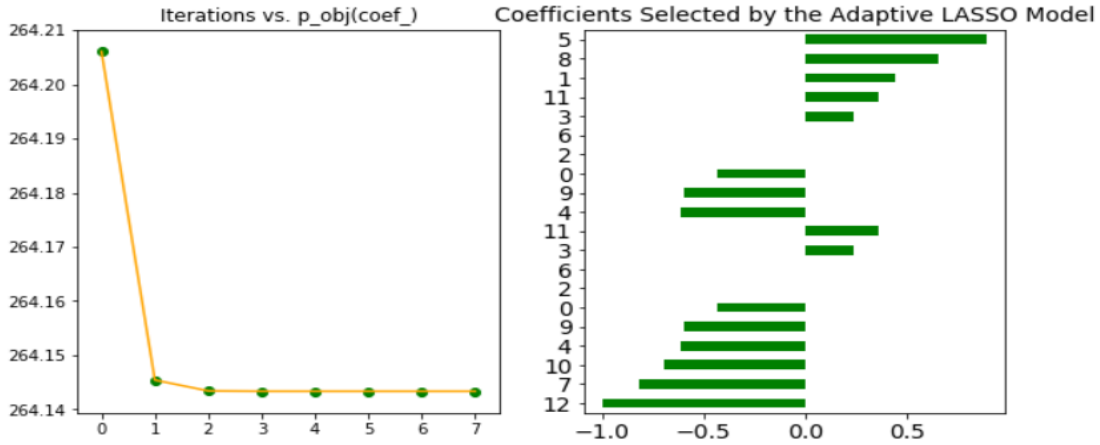
```
        plt.rcParams["figure.figsize"] = (10,5)
        plt.figure()
        # subplot of the predicted vs. actual
        plt.subplot(1,2,1)
        plt.plot(iterations_list,coefficients_list,color = 'orange
')
        plt.scatter(iterations_list,coefficients_list,color = 'gre
en')
        plt.title('Iterations vs. p_obj(coef_)')

        # plot of the coefficients'
        plt.subplot(1,2,2)
        imp_coef = pd.concat([coef.sort_values().head(10),coef.sor
t_values().tail(10)])
        imp_coef.plot(kind = "barh", color = 'green',  fontsize=14
)
        plt.title("Coefficients Selected by the Adaptive LASSO Mod
el", fontsize = 14)
        plt.show()
        return adaptive_lasso
    # 模型训练
    X_train=pd.DataFrame(X_train)
    model_lasso=Adaptive_LASSO(X_train,y_train,lasso_iterations=8)
```

（3） 程序结果：

```
X_train=pd.DataFrame(X_train)
model_lasso=Adaptive_LASSO(X_train,y_train,lasso_iterations=8)
```

```
Iteration # 1 :    264.20608057659007
Iteration # 2 :    264.1454183301053
Iteration # 3 :    264.1434144508424
Iteration # 4 :    264.14335384316723
Iteration # 5 :    264.1433519584512
Iteration # 6 :    264.1433519061564
Iteration # 7 :    264.14335190824914
Iteration # 8 :    264.14335190925414
1.0
================================================================================
Adaptive LASSO picked 11 variables and eliminated the other 2 variables.
================================================================================
```



（六） SCAN

1. 理论介绍

Fan and Li（2001）从惩罚函数的角度出发，认为一个好的惩罚函数应使得模型中的解具有以下三个理论性质：

（1） 无偏性：当位置参数真值较大时，估计值应该几乎无偏

（2） 稀疏性：有某个阈值准则自动将较小的估计系数降至 0，以降低模型复杂度

（3） 连续性：为了避免模型在预测时的不稳定性，估计值应该是最小二乘估计值的某种连续函数

Fan and Li 证明，SCAD 回归在一定条件下满足神谕性，即 SCAD 估计的表现与真实模型已知时（假定已知真实模型中参数为零的系数）的表现一样好，也就是说，SCAD 估计能准确地进行变量选择。

一般来说，一个加罚的最小二乘估计的一般形式为：

$$\frac{1}{2}(y-x\beta)^2+\lambda\sum_{j=1}^{p}p_j(|\beta_j|)$$

(x 是正交矩阵)可简化为：$\frac{1}{2}(z-\beta)^2+p_\lambda(|\beta|)$,其中$z=x^Ty$，对β求解一阶导：

$$\text{sign}(\beta)\{|\beta|+p'_\lambda(|\beta|)\}-z$$

分析可知：

（1）$p'_\lambda(|\beta|)=0$对于大的|β|成立

（2）满足稀疏性的条件是$|\beta|+p'_\lambda(|\beta|)$的最小值是正数

（3）连续性条件是$|\beta|+p'_\lambda(|\beta|)$的最小值在 0 点达到

Fan and Li 提出了满足以上三个性质的 SCAD 惩罚函数，一般来说，通常的 SCAD 惩罚函数指的是这一函数的导数：

$$p'_\lambda=\lambda\left[I(\beta\le\lambda)+\frac{(\alpha\lambda-\beta)_+}{(\alpha-1)\lambda}I(\beta>\lambda)\right]，\ \alpha>0，\ \beta>0$$

SCAD 的估计为：

$$\hat{\theta}=\begin{cases}sgn(z)(|z|-\lambda)_+ & when|z|\le 2\lambda\\ \{(a-1)z-sgn(z)a\lambda\}/(a-2) & when2\lambda<|z|\le a\lambda\\ z & when|z|>a\lambda\end{cases}$$

Fan 和 Li 证明，SCAD 估计在一定条件下满足神谕性，也即 SCAD 估计能准确地进行变量选择。

（七） 主成分分析

1. 理论介绍

主成分分析是一种降维技术，它可以从多个变量中得到低维变量。它利用正交变换来对一系列可能相关的变量的观测值进行线性变换，从而投影为一些列不相关变量的值，这些不相关变量称为主成分。具体地，主成分可以看作一个线性方程，其包含一系列线性系数来指

示投影方向。

基本思想：① 将坐标轴中心移到数据的中心，然后旋转坐标轴，使得数据在 C1 轴上的方差最大，即全部 n 个数据个体在该方向上的投影最为分散。意味着更多的信息被保留下来。C1 成为第一主成分。② C2 第二主成分：找一个 C2，使得 C2 与 C1 的协方差（相关系数）为 0，以免与 C1 信息重叠，并且使数据在该方向的方差尽量最大。③ 以此类推，找到第三主成分，第四主成分……第 p 个主成分。p 个随机变量可以有 p 个主成分。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过主成分分析实现特征的降维。

（2） 代码实现：

```
# 使用 python 自带的乳腺癌数据集（二分类）
from sklearn.datasets import load_breast_cancer
X,y=load_breast_cancer(return_X_y=True)
print("样本的个数 = {},      变量的个
数 = {}".format(X.shape[0],X.shape[1]))
# 将特征数据进行标准化
from sklearn.preprocessing import StandardScaler
scaler=StandardScaler().fit(X)
X=scaler.transform(X)
# 导入 PCA 函数
from sklearn.decomposition import PCA
# 进行主成分降维，并比较在不同主成分个数的情况下的方差累计贡献率
for i in range(1,11):
    pca=PCA(n_components=i)
    pca.fit(X)
    variance_ratio=pca.explained_variance_ratio_
    print(i,"个主成分可以解释原变量%.3f%%的信息
"% (np.sum(variance_ratio) *100))
```

（3） 程序结果：

```
#  使用python自带的乳腺癌数据集（二分类）
from  sklearn.datasets  import  load_breast_cancer
X,y=load_breast_cancer(return_X_y=True)
print("样本的个数  =  {},          变量的个数  =  {}".format(X.shape[0],X.shape[1]))
#  将特征数据进行标准化
from  sklearn.preprocessing  import  StandardScaler
scaler=StandardScaler().fit(X)
X=scaler.transform(X)
#  导入PCA函数
from  sklearn.decomposition  import  PCA
#  进行主成分降维，并比较在不同主成分个数的情况下的方差累计贡献率
for  i  in  range(1,11):
    pca=PCA(n_components=i)
    pca.fit(X)
    variance_ratio=pca.explained_variance_ratio_
    print(i,"个主成分可以解释原变量%.3f%%的信息"%  (np.sum(variance_ratio)  *100))
```

```
样本的个数 = 569,      变量的个数 = 30
1 个主成分可以解释原变量44.272%的信息
2 个主成分可以解释原变量63.243%的信息
3 个主成分可以解释原变量72.636%的信息
4 个主成分可以解释原变量79.239%的信息
5 个主成分可以解释原变量84.734%的信息
6 个主成分可以解释原变量88.759%的信息
7 个主成分可以解释原变量91.010%的信息
8 个主成分可以解释原变量92.598%的信息
9 个主成分可以解释原变量93.988%的信息
10 个主成分可以解释原变量95.157%的信息
```

五、 决策树

（一） ID3 决策树

1. 理论介绍

ID3 决策树学习算法是以信息增益准则来选择划分属性的。要了解信息增益的原理，我们首先需要对信息熵进行定义。信息熵是度量样本纯度最常用的一种指标。假定当前样本集合 D 中第 k 类样本所占的比例为$p_k(k = 1,2,\cdots,|y|)$，则 D 的信息熵定义为

$$\mathrm{Ent(D)} = -\sum_{k=1}^{|y|} p_k log_2 p_k$$

Ent(D)的值越小，则 D 的纯度越高。

假定离散属性 a 有 V 个可能的取值$\{a^1,a^2,\cdots,a^v\}$，若使用 a 来对样本集 D 进行划分，则会产生 V 个分支结点，其中第 v 个分支结点包含了 D 中所有在属性 a 上取值为a^v的样本，记为D^v。我们可以根据上式计算出D^v的信息熵，再考虑到不同的分支结点所包含的样本数不同，给分支结点赋予权重$|D^v|/|D|$，即样本数越多的分支结点的影响越大，于是可计算出用属性 a 对样本集 D 进行划分所获得的“信息增益”

$$\mathrm{Gain(D,a)} = \mathrm{Ent(D)} - \sum_{v=1}^{V} \frac{|D^v|}{|D|}\mathrm{Ent}(D^v)$$

一般而言，信息增益越大，意味着使用属性 a 来进行划分所获得的“纯度提升”越

大。因此，我们可以用信息增益来进行决策树的划分属性选择，著名的 ID3 决策树学习算法就是以信息增益为准则来选择划分属性的。

优点：① 很容易获得相应的决策规则；② 理论清晰，方法简单，学习能力较强；③ 构建速度比较快

缺点：① 只能处理分类属性，连续属性需转化为分类属性处理；② 对训练样本的质量依赖性很强，训练样本的质量主要是指是否存在噪声和是否存在足够的样本；③ 生成的决策树是一棵多叉树，分支的数量取决于分裂属性有多少个不同的取值，这并不利于分裂属性取值数目较多的情况；④ 此算法中并不包括树的修剪，这样模型受噪声数据和统计波动的影响比较大；⑤ 在不重建整棵树的条件下，不能方便的对决策树做更改。

2. 案例实现

（1） 数据来源及解释

本案例选取了 R 自带的 iris 数据集，该数据集有 4 个特征变量，1 个三分类的目标变量，希望通过建立 ID3 模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 通过 rpy2 函数调用 R 程序
from rpy2.robjects import r
from rpy2.robjects.packages import importr
import numpy as np
# 通过 python 程序调用 R 自带的 iris 数据集
data_sample="""
#使用 R 自带的 iris 数据集
data=iris
head(iris)
# 通过留出法划分数据集
set.seed(123)
train_index=sample(nrow(data),0.7*nrow(data))
train_data=data[train_index,]
test_data=data[-train_index,]
"""
# 通过调用 rpy2 包，用 python 执行 R 程序
r(data_sample)
# 将 y_true 数据格式转化为 np.array
y_true=np.array(r("test_data$Species"))
# 导入 ID3 决策树需要的函数包
importr('rpart')
importr('rpart.plot')
ID3_tree="""
ID3_tree=rpart(train_data$Species~.,data=train_data,method="cl
ass",
                parms=list(split="information"),minsplit=0)
```

```
rpart.plot(ID3_tree,branch=1,type=2, fallen.leaves=T,cex=0.8,
sub="ID3—未剪枝")
pred_ID3_tree=predict(ID3_tree,test_data,type='class')
"""
# 通过 python 执行 R 中的 ID3 决策树程序
r(ID3_tree)
# 导出 ID3 决策树（未剪枝）的预测值，并转换为 np.array 格式
y_pred_ID3=np.array(r("pred_ID3_tree"))
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred_ID3,y_true))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred_ID3,y_true)
print("ID3 决策树（未剪枝）拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

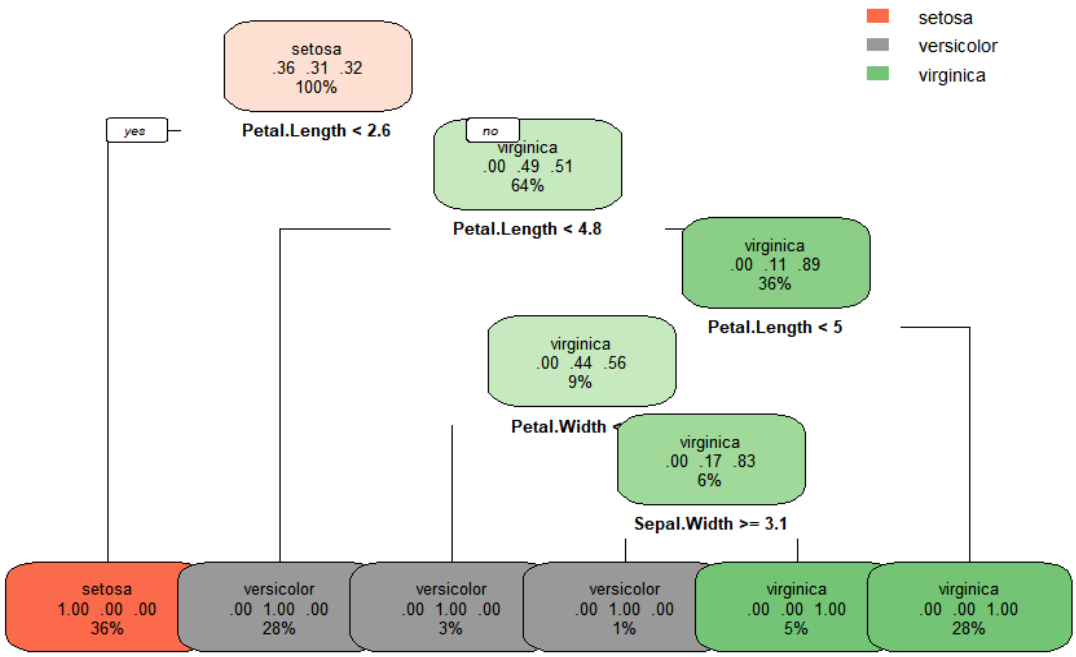
```
#  导入ID3决策树需要的函数包
importr('rpart')
importr('rpart.plot')
ID3_tree="""
ID3_tree=rpart(train_data$Species~.,data=train_data,method="class",
                    parms=list(split="information"),minsplit=0)
rpart.plot(ID3_tree,branch=1,type=2,  fallen.leaves=T,cex=0.8,  sub="ID3—未剪枝")
pred_ID3_tree=predict(ID3_tree,test_data,type='class')
"""
#  通过python执行R中的ID3决策树程序
r(ID3_tree)
#  导出ID3决策树（未剪枝）的预测值，并转换为np.array格式
y_pred_ID3=np.array(r("pred_ID3_tree"))
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred_ID3,y_true))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred_ID3,y_true)
print("ID3决策树（未剪枝）拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        14
           2       0.89      1.00      0.94        16
           3       1.00      0.87      0.93        15

    accuracy                           0.96        45
   macro avg       0.96      0.96      0.96        45
weighted avg       0.96      0.96      0.96        45

ID3决策树（未剪枝）拟合的准确率为:95.556%
```



ID3—未剪枝

（二） C4.5 决策树

1. 理论介绍

由于信息增益准则对可取数目较多的属性有所偏好，为减少这种偏好可能带来的不利影响，C4.5 决策树算法不直接使用信息增益，而是使用“增益率”来选择最优划分属性，增益率定义为：

$$\text{Gain\_ratio(D, a)} = \frac{Gain(D,a)}{IV(a)}$$

其中

$$\text{IV(a)} = -\sum_{v=1}^{V} \frac{|D^v|}{|D|} log_2 \frac{|D^v|}{|D|}$$

称为属性 a 的“固有值”。属性 a 的可能取值数目越多（即 V 越大），则IV(a)的值通常会越大。需注意的是，增益率准则对可取值数目较少的属性有所偏好，因此，C4.5 算法并不是直接选择增益率最大的候选划分属性，而是使用一个启发式：先从划分属性中找出信息增益高于平均水平的属性，再从中选择增益率最高的。

优点：① 速度快：计算量相对较小，且容易转化成分类规则。只要沿着树根向下一直走到叶，沿途的分裂条件就能够唯一确定一条分类的路径；② 准确率高：挖掘出的分类规则准确性高，便于理解，决策树可以清晰的显示哪些字段比较重要；③ 非参数学习，不需要设置参数。

缺点：① 为处理大数据集或连续量的种种改进算法（离散化、取样）不仅增加了分类算法的额外开销，而且降低了分类的准确性，对连续性的字段比较难预测，当类别太多时，

错误可能就会增加的比较快，对有时间顺序的数据，需要很多预处理的工作。

2. 案例实现

（1） 数据来源及解释

本案例选取了 R 自带的 iris 数据集，该数据集有 4 个特征变量，1 个三分类的目标变量，希望通过建立 C4.5 模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 仍旧使用 iris 数据集，数据集的划分及预处理均与 ID3 相同
# 导入 C4.5 决策树需要的函数包
importr('RWeka')
importr('partykit')
C4_5_tree="""
 # 将每个叶节点最小样本数设为 20
C4_5_tree=J48(factor(train_data$Species)~.,data=train_data,con
trol=Weka_control(M=2))
plot(C4_5_tree,main="C4.5—未剪枝")
pred_C4_5_tree=predict(C4_5_tree,test_data,type='class')
"""
# 通过 python 执行 R 中的 C4.5 决策树程序
r(C4_5_tree)
# 导出 C4.5 决策树（未剪枝）的预测值，并转换为 np.array 格式
y_pred_C4_5=np.array(r("pred_C4_5_tree"))
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred_C4_5,y_true))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred_C4_5,y_true)
print("C4.5 决策树（未剪枝）拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

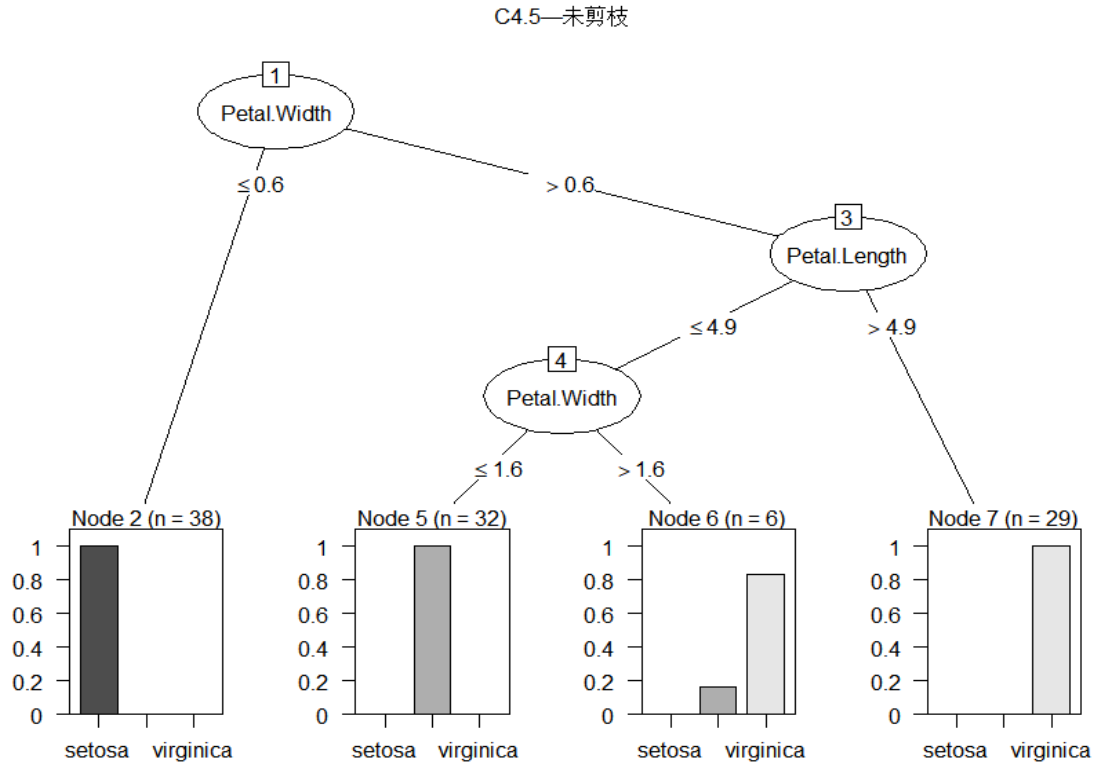
```
#  导入C4.5决策树需要的函数包
importr('RWeka')
importr('partykit')
C4_5_tree="""
  #  将每个叶节点最小样本数设为20
C4_5_tree=J48(factor(train_data$Species)~.,data=train_data,control=Weka_control(M=2))
plot(C4_5_tree,main="C4.5—未剪枝")
pred_C4_5_tree=predict(C4_5_tree,test_data,type='class')
"""
#  通过python执行R中的C4.5决策树程序
r(C4_5_tree)
#  导出C4.5决策树（未剪枝）的预测值，并转换为np.array格式
y_pred_C4_5=np.array(r("pred_C4_5_tree"))
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred_C4_5,y_true))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred_C4_5,y_true)
print("C4.5决策树（未剪枝）拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        14
           2       0.94      1.00      0.97        17
           3       1.00      0.93      0.96        14

    accuracy                           0.98        45
   macro avg       0.98      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45
```

C4.5决策树（未剪枝）拟合的准确率为:97.778%



（三） CART 决策树

1. 理论介绍

CART 决策树使用“基尼指数”来选择划分属性。数据集 D 的纯度可用基尼值来度量：

$$\text{Gini(D)} = \sum_{k=1}^{|y|}\sum_{k'\neq k} p_k p_{k'} = 1 - \sum_{k=1}^{|y|} p_k^2$$

直观来说，Gini(D)反映了从数据集 D 中随机抽取两个样本，其类别标记不一致的概率。因此，Gini(D)越小，则数据集 D 的纯度越高。其中属性 a 的基尼指数定义为：

$$\text{Gini\_index(D, a)} = \sum_{v=1}^{V} \frac{|D^v|}{|D|} Gini(D^v)$$

于是，我们在候选属性集合 A 中，选择那个使得划分后基尼指数最小的属性作为最优划分属性，即$a_* = arg\min_{a\in A} Gini_\text{index(D, a)}$。

2. 案例实现

（1） 数据来源及解释

本案例选取了 R 自带的 iris 数据集，该数据集有 4 个特征变量，1 个三分类的目标变量，希望通过建立 CART 模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 仍旧使用 iris 数据集，数据集的划分及预处理均与 ID3 相同
# 导入 CART 决策树需要的函数包
importr('rpart')
importr('rpart.plot')
CART_tree="""
CART_tree=rpart(train_data$Species~.,data=train_data,method="class",
                parms=list(split="gini"),minsplit=0)
rpart.plot(CART_tree,branch=1,type=2, fallen.leaves=T,cex=0.8,
 sub="CART—未剪枝")
pred_CART_tree=predict(CART_tree,test_data,type='class')
"""
# 通过 python 执行 R 中的 CART 决策树程序
r(CART_tree)
# 导出 CART 决策树（未剪枝）的预测值，并转换为 np.array 格式
y_pred_CART=np.array(r("pred_CART_tree"))
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred_CART,y_true))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred_CART,y_true)
print("CART 决策树（未剪枝）拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

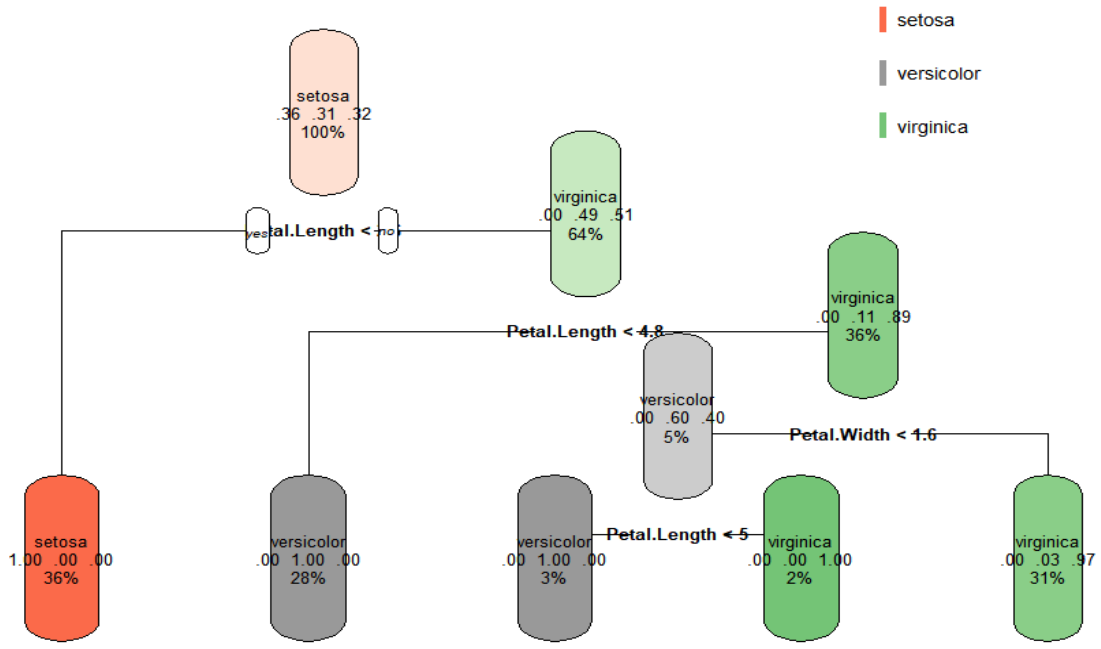
```
#  导入CART决策树需要的函数包
importr('rpart')
importr('rpart.plot')
CART_tree="""
CART_tree=rpart(train_data$Species~.,data=train_data,method="class",
                parms=list(split="gini"),minsplit=0)
rpart.plot(CART_tree,branch=1,type=2,  fallen.leaves=T,cex=0.8,  sub="CART—未剪枝")
pred_CART_tree=predict(CART_tree,test_data,type='class')
"""
#  通过python执行R中的CART决策树程序
r(CART_tree)
#  导出CART决策树（未剪枝）的预测值，并转换为np.array格式
y_pred_CART=np.array(r("pred_CART_tree"))
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred_CART,y_true))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred_CART,y_true)
print("CART决策树（未剪枝）拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        14
           2       0.89      1.00      0.94        16
           3       1.00      0.87      0.93        15

    accuracy                           0.96        45
   macro avg       0.96      0.96      0.96        45
weighted avg       0.96      0.96      0.96        45
```

CART决策树（未剪枝）拟合的准确率为:95.556%



CART—未剪枝

（四） C5.0 决策树

1. 理论介绍

C5.0 算法是 Quinlan 在 C4.5 算法的基础上提出的商用改进版本，目前是对含有大量数据的数据集进行分析。C5.0 算法对于结点的选择主要是采用 Boosting 方式提高模型准确率，又被称为 Boosting Tree，在软件上的计算速度比较快，占用的内存资源较少。

C4.5 决策树与 C5.0 决策树的比较：① C5.0 决策树是主要应用于大数据集上的分类算法，主要在执行效率和内存使用方面进行了改进；② C5.0 决策树在面对数据遗漏和输入字段很多的问题时非常稳健；③ C5.0 提供强大技术以提高分类的精度；④ C5.0 比一些其它类型的模型易于理解，模型推出的规则有非常直观的解释。

2. 案例实现

（1） 数据来源及解释

本案例选取了 R 自带的 iris 数据集，该数据集有 4 个特征变量，1 个三分类的目标变量，希望通过建立 C5.0 模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

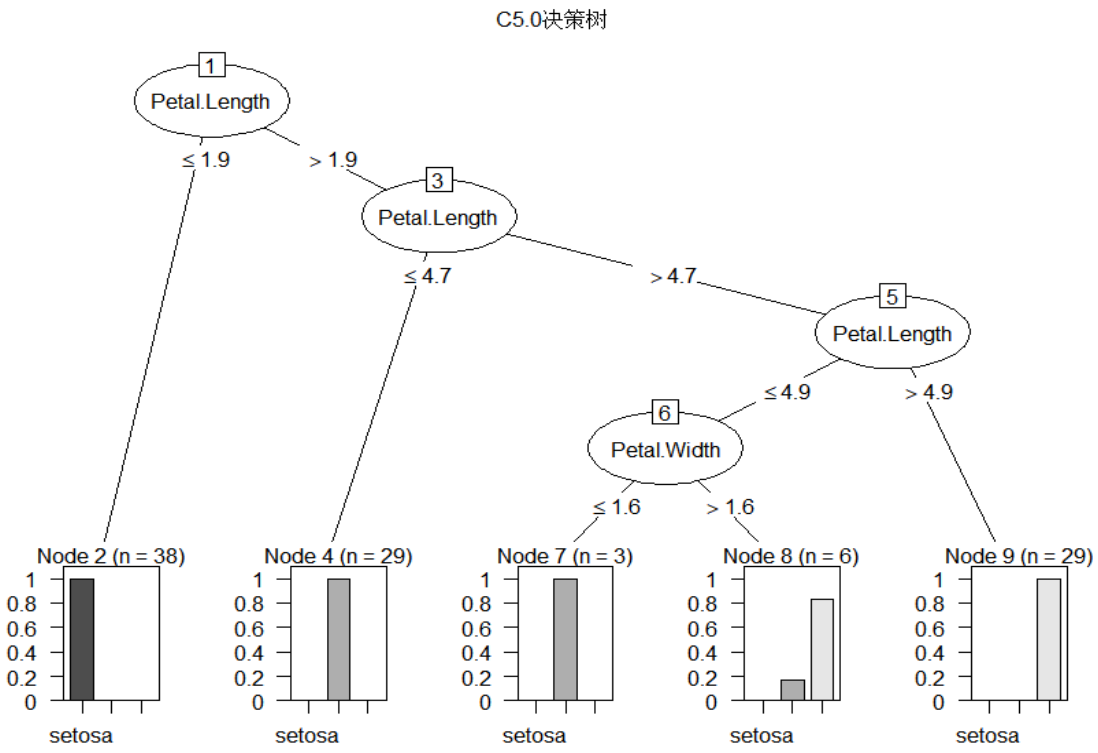
```
# 仍旧使用 iris 数据集，数据集的划分及预处理均与 ID3 相同
# 导入 C5.0 决策树需要的函数包
importr('C50')
C50_tree="""
C50_tree=C5.0(x=train_data[,-5],y=factor(train_data$Species))
plot(C50_tree,main="C5.0 决策树")
pred_C50_tree=predict(C50_tree,test_data,type='class')
"""
# 通过 python 执行 R 中的 C5.0 决策树程序
r(C50_tree)
# 导出 C5.0 决策树的预测值，并转换为 np.array 格式
y_pred_C50=np.array(r("pred_C50_tree"))
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred_C50,y_true))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred_C50,y_true)
print("C5.0 决策树拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
#  导入C5.0决策树需要的函数包
importr('C50')
C50_tree="""
C50_tree=C5.0(x=train_data[,-5],y=factor(train_data$Species))
plot(C50_tree,main="C5.0决策树")
pred_C50_tree=predict(C50_tree,test_data,type='class')
"""
#  通过python执行R中的C5.0决策树程序
r(C50_tree)
#  导出C5.0决策树的预测值，并转换为np.array格式
y_pred_C50=np.array(r("pred_C50_tree"))
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred_C50,y_true))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred_C50,y_true)
print("C5.0决策树拟合的准确率为:%.3f%%"%  (Acc  *100))
```

	precision	recall	f1-score	support
1	1.00	1.00	1.00	14
2	0.94	1.00	0.97	17
3	1.00	0.93	0.96	14
accuracy			0.98	45
macro avg	0.98	0.98	0.98	45
weighted avg	0.98	0.98	0.98	45

C5.0决策树拟合的准确率为:97.778%



（五） 预剪枝

1. 理论介绍

预剪枝是指在决策树生成过程中，对每个结点在划分前先进行估计，若当前结点的划分不能带来决策树泛化性能提升，则停止划分并将当前结点标记为叶结点；首先分别计算划分前（即直接将该结点作为叶结点）及划分后的验证集精度，判断是否需要划分，若划分后能提高验证集精度，则划分，对划分后的属性，执行同样判断，否则不划分。

优点：① 可以降低过拟合的风险；② 显著减少训练时间和测试时间开销。

缺点：① 过早的预剪枝可能会造成欠拟合的风险，有些分支的当前划分虽然不能提升泛化性能，但在其基础上进行的后续划分却有可能导致性能显著提高；② 预剪枝基于“贪心”本质禁止这些分支展开，带来了欠拟合的风险。

2. 案例实现

（1） 数据来源及解释

本案例选取了 R 自带的 iris 数据集，该数据集有 4 个特征变量，1 个三分类的目标变量，希望通过对事先建立的 ID3、C4.5 和 CART 决策树模型进行预剪枝，来判定预剪枝操作对模型拟合的影响。

（2） 代码实现：

```
# 仍旧使用 iris 数据集，数据集的划分及预处理均与 ID3 相同
# 导入 ID3 决策树需要的函数包
importr('rpart')
importr('rpart.plot')
ID3_tree_pre="""
 # 将每个叶节点最小样本数设为 20
ID3_tree_pre=rpart(train_data$Species~.,data=train_data,method
="class",
                   parms=list(split="information"),minsplit=20)
rpart.plot(ID3_tree_pre,branch=1,type=2, fallen.leaves=T,cex=0
.8, sub="ID3—预剪枝")
pred_ID3_tree_pre=predict(ID3_tree_pre,test_data,type='class')
"""
# 通过 python 执行 R 中的 ID3 决策树程序
r(ID3_tree_pre)
# 导出 ID3 决策树（预剪枝）的预测值，并转换为 np.array 格式
y_pred_ID3_pre=np.array(r("pred_ID3_tree_pre"))
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred_ID3_pre,y_true))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred_ID3_pre,y_true)
print("ID3 决策树（预剪枝）拟合的准确率为:%.3f%%"% (Acc *100))
```

```
# 导入 C4.5 决策树需要的函数包
importr('RWeka')
importr('partykit')
C4_5_tree_pre="""
#预剪枝
C4_5_tree_pre=J48(factor(train_data$Species)~.,data=train_data
,
                  control=Weka_control(U=T,M=5))
plot(C4_5_tree_pre,main="C4.5—预剪枝")
pred_C4_5_tree_pre=predict(C4_5_tree_pre,test_data,type='class
')
"""
# 通过 python 执行 R 中的 C4.5 决策树程序
r(C4_5_tree_pre)
# 导出 C4.5 决策树（预剪枝）的预测值，并转换为 np.array 格式
y_pred_C4_5_pre=np.array(r("pred_C4_5_tree_pre"))
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred_C4_5_pre,y_true))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred_C4_5_pre,y_true)
print("C4.5 决策树（预剪枝）拟合的准确率为:%.3f%%"% (Acc *100))

# 导入 CART 决策树需要的函数包
importr('rpart')
importr('rpart.plot')
CART_tree_pre="""
#将每个叶节点最小样本数设为 20
CART_tree_pre=rpart(train_data$Species~.,data=train_data,metho
d="class",
                parms=list(split="gini"),minsplit=20)
rpart.plot(CART_tree_pre,branch=1,type=2, fallen.leaves=T,cex=
0.8, sub="CART—预剪枝")
pred_CART_tree_pre=predict(CART_tree_pre,test_data,type='class
')
"""
# 通过 python 执行 R 中的 CART 决策树程序
r(CART_tree_pre)
# 导出 CART 决策树（预剪枝）的预测值，并转换为 np.array 格式
y_pred_CART_pre=np.array(r("pred_CART_tree_pre"))
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred_CART_pre,y_true))
from sklearn.metrics import accuracy_score
```

```
Acc=accuracy_score(y_pred_CART_pre,y_true)
print("CART 决策树（预剪枝）拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

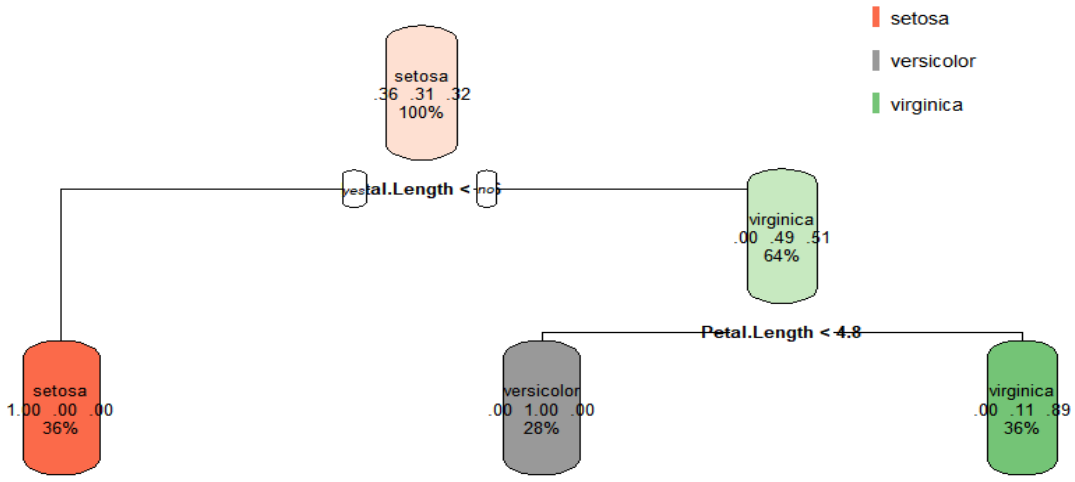
```
#  导入ID3决策树需要的函数包
importr('rpart')
importr('rpart.plot')
ID3_tree_pre="""
  #  将每个叶节点最小样本数设为20
ID3_tree_pre=rpart(train_data$Species~.,data=train_data,method="class",
                   parms=list(split="information"),minsplit=20)
rpart.plot(ID3_tree_pre,branch=1,type=2,  fallen.leaves=T,cex=0.8,  sub="ID3—预剪枝")
pred_ID3_tree_pre=predict(ID3_tree_pre,test_data,type='class')
"""
#  通过python执行R中的ID3决策树程序
r(ID3_tree_pre)
#  导出ID3决策树（预剪枝）的预测值，并转换为np.array格式
y_pred_ID3_pre=np.array(r("pred_ID3_tree_pre"))
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred_ID3_pre,y_true))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred_ID3_pre,y_true)
print("ID3决策树（预剪枝）拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        14
           2       1.00      0.95      0.97        19
           3       0.92      1.00      0.96        12

    accuracy                           0.98        45
   macro avg       0.97      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45

ID3决策树（预剪枝）拟合的准确率为:97.778%
```



ID3—预剪枝

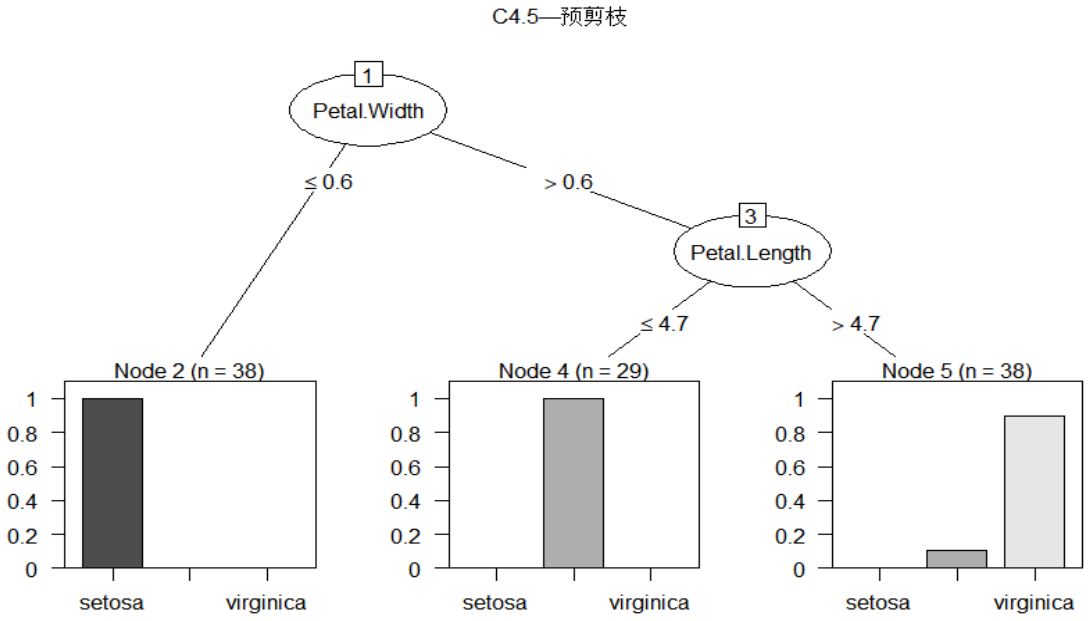
```
#  导入C4.5决策树需要的函数包
importr('RWeka')
importr('partykit')
C4_5_tree_pre="""
#预剪枝
C4_5_tree_pre=J48(factor(train_data$Species)~.,data=train_data,
                                    control=Weka_control(U=T,M=5))
plot(C4_5_tree_pre,main="C4.5—预剪枝")
pred_C4_5_tree_pre=predict(C4_5_tree_pre,test_data,type='class')
"""
#  通过python执行R中的C4.5决策树程序
r(C4_5_tree_pre)
#  导出C4.5决策树（预剪枝）的预测值，并转换为np.array格式
y_pred_C4_5_pre=np.array(r("pred_C4_5_tree_pre"))
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred_C4_5_pre,y_true))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred_C4_5_pre,y_true)
print("C4.5决策树（预剪枝）拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        14
           2       0.89      0.94      0.91        17
           3       0.92      0.86      0.89        14

    accuracy                           0.93        45
   macro avg       0.94      0.93      0.93        45
weighted avg       0.93      0.93      0.93        45
```

C4.5决策树（预剪枝）拟合的准确率为:93.333%



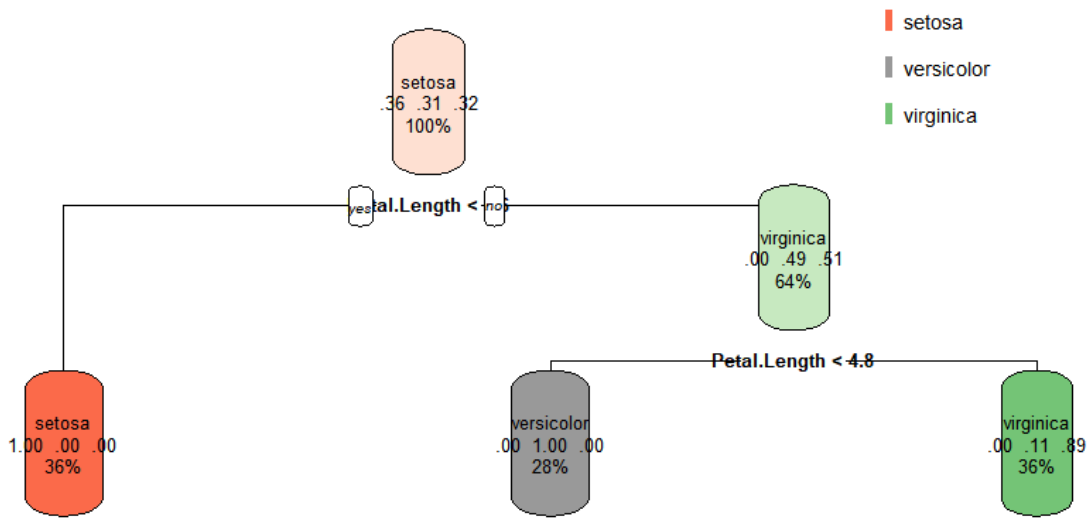
```
#  导入CART决策树需要的函数包
importr('rpart')
importr('rpart.plot')
CART_tree_pre="""
#将每个叶节点最小样本数设为20
CART_tree_pre=rpart(train_data$Species~.,data=train_data,method="class",
                    parms=list(split="gini"),minsplit=20)
rpart.plot(CART_tree_pre,branch=1,type=2,  fallen.leaves=T,cex=0.8,  sub="CART—预剪枝")
pred_CART_tree_pre=predict(CART_tree_pre,test_data,type='class')
"""
#  通过python执行R中的CART决策树程序
r(CART_tree_pre)
#  导出CART决策树（预剪枝）的预测值，并转换为np.array格式
y_pred_CART_pre=np.array(r("pred_CART_tree_pre"))
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred_CART_pre,y_true))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred_CART_pre,y_true)
print("CART决策树（预剪枝）拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        14
           2       1.00      0.95      0.97        19
           3       0.92      1.00      0.96        12

    accuracy                           0.98        45
   macro avg       0.97      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45
```

CART决策树（预剪枝）拟合的准确率为:97.778%



CART—预剪枝

（六）后剪枝

1. 理论介绍

后剪枝与预剪枝的原理相似，但是后剪枝是先从训练集生成一颗完整的决策树，然后从最底部的结点开始判断是否将其剪除。若当前结点的划分不能带来决策树泛化性能提升，则停止划分并将当前结点标记为叶结点。

优点：后剪枝比预剪枝保留了更多的分支，欠拟合风险小，泛化性能往往优于预剪枝决策树。

缺点：训练时间开销大，后剪枝过程是在生成完全决策树之后进行的，需要自底向上对所有非叶节点进行考察。

2. 案例实现

（1） 数据来源及解释

本案例选取了 R 自带的 iris 数据集，该数据集有 4 个特征变量，1 个三分类的目标变量，希望通过对事先建立的 ID3 和 CART 决策树模型进行后剪枝，来判定预剪枝操作对模型拟合的影响。

（2） 代码实现：

```
# 仍旧使用 iris 数据集，数据集的划分及预处理均与 ID3 相同
# 导入 ID3 决策树需要的函数包
importr('rpart')
importr('rpart.plot')
ID3_tree_after="""
#后剪枝，将 CP 值设为 0.1
ID3_tree_after<-prune(ID3_tree,cp=0.1)
rpart.plot(ID3_tree_after,branch=1,type=2, fallen.leaves=T,cex
=0.8, sub="ID3—后剪枝")
pred_ID3_tree_after=predict(ID3_tree_after,test_data,type='cla
ss')
"""
# 通过 python 执行 R 中的 ID3 决策树程序
r(ID3_tree_after)
# 导出 ID3 决策树（后剪枝）的预测值，并转换为 np.array 格式
y_pred_ID3_after=np.array(r("pred_ID3_tree_after"))
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred_ID3_after,y_true))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred_ID3_after,y_true)
print("ID3 决策树（后剪枝）拟合的准确率为:%.3f%%"% (Acc *100))

# 导入 CART 决策树需要的函数包
importr('rpart')
```

```
importr('rpart.plot')
CART_tree_after="""
#后剪枝，将 CP 值设为 0.1
CART_tree_after<-prune(CART_tree,cp=0.1)
rpart.plot(CART_tree_after,branch=1,type=2, fallen.leaves=T,ce
x=0.8, sub="CART—后剪枝")
pred_CART_tree_after=predict(CART_tree_after,test_data,type='c
lass')
"""
# 通过 python 执行 R 中的 CART 决策树程序
r(CART_tree_after)
# 导出 CART 决策树（后剪枝）的预测值，并转换为 np.array 格式
y_pred_CART_after=np.array(r("pred_CART_tree_after"))
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred_CART_after,y_true))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred_CART_after,y_true)
print("CART 决策树（后剪枝）拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

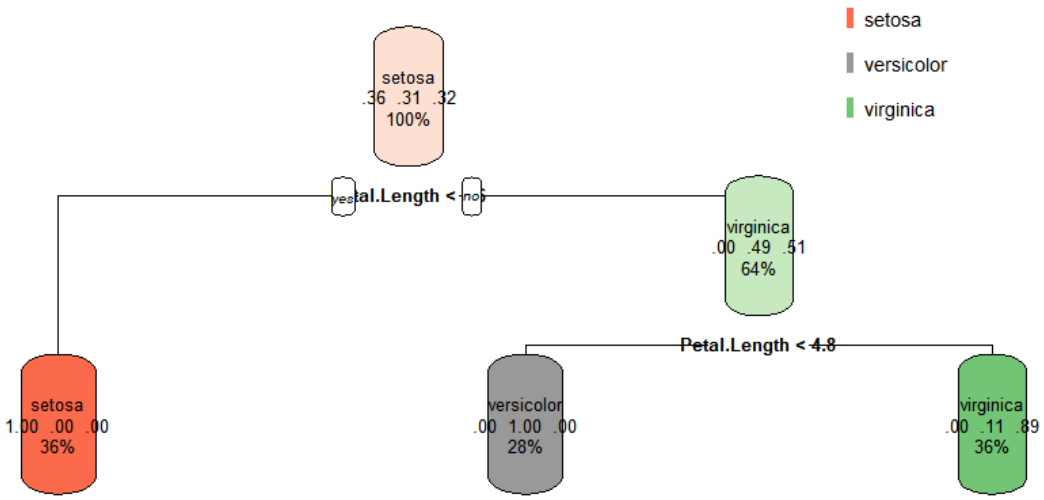
```
# 导入ID3决策树需要的函数包
importr('rpart')
importr('rpart.plot')
ID3_tree_after="""
#后剪枝，将CP值设为0.1
ID3_tree_after<-prune(ID3_tree,cp=0.1)
rpart.plot(ID3_tree_after,branch=1,type=2,  fallen.leaves=T,cex=0.8,  sub="ID3—后剪枝")
pred_ID3_tree_after=predict(ID3_tree_after,test_data,type='class')
"""
# 通过python执行R中的ID3决策树程序
r(ID3_tree_after)
# 导出ID3决策树（后剪枝）的预测值，并转换为np.array格式
y_pred_ID3_after=np.array(r("pred_ID3_tree_after"))
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred_ID3_after,y_true))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred_ID3_after,y_true)
print("ID3决策树（后剪枝）拟合的准确率为:%.3f%%"% (Acc *100))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        14
           2       1.00      0.95      0.97        19
           3       0.92      1.00      0.96        12

    accuracy                           0.98        45
   macro avg       0.97      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45

ID3决策树（后剪枝）拟合的准确率为:97.778%
```



ID3—后剪枝

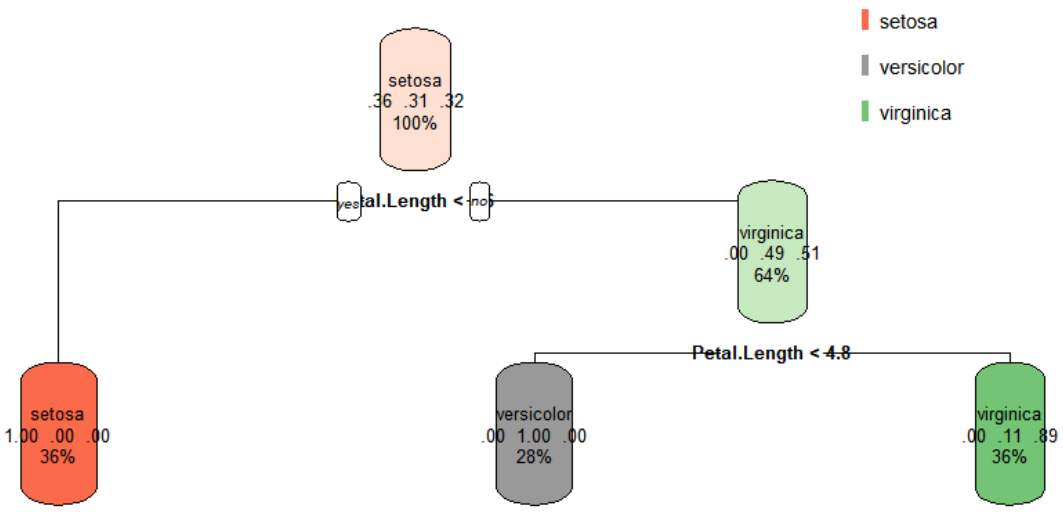
```
#  导入CART决策树需要的函数包
importr('rpart')
importr('rpart.plot')
CART_tree_after="""
#后剪枝，将CP值设为0.1
CART_tree_after<-prune(CART_tree,cp=0.1)
rpart.plot(CART_tree_after,branch=1,type=2,  fallen.leaves=T,cex=0.8,  sub="CART—后剪枝")
pred_CART_tree_after=predict(CART_tree_after,test_data,type='class')
"""
#  通过python执行R中的CART决策树程序
r(CART_tree_after)
#  导出CART决策树（后剪枝）的预测值，并转换为np.array格式
y_pred_CART_after=np.array(r("pred_CART_tree_after"))
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred_CART_after,y_true))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred_CART_after,y_true)
print("CART决策树（后剪枝）拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           1       1.00      1.00      1.00        14
           2       1.00      0.95      0.97        19
           3       0.92      1.00      0.96        12

    accuracy                           0.98        45
   macro avg       0.97      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45

CART决策树（后剪枝）拟合的准确率为:97.778%
```



CART—后剪枝

（七） 各模型比较

从效果上来看，C5.0 和 CART 决策树的效果最优，ID3 效果最差；从对数据处理来看，C5.0 算法可以应对大数据，其他三个算法消耗的时间较多，不适合对大数据进行建模。

ID3 算法有几个缺点：① 对于具有很多值的属性它是非常敏感的，例如，如果我们数据集中的某个属性值对不同的样本基本上是不相同的，甚至更极端点，对于每个样本都是唯一的，如果我们用这个属性来划分数据集，它会得到很大的信息增益，但是，这样的结果并不是我们想要的；② ID3 算法不能处理具有连续值的属性；③ ID3 算法不能处理属性具有缺失值的样本；④ 会生成很深的树，容易产生过拟合现象。

对此，我们引入 C4.5 算法，C4.5 通过信息增益率来度量，可以避免多值属性的敏感，而且可以处理连续型变量。

C5.0 是一个商业软件，对于公众是不可得到的。它是在 C4.5 算法做了一些改进，C5.0 主要增加了对 Boosting 的支持，它同时也用更少地内存。它与 C4.5 算法相比，它构建了更小地规则集，因此它更加准确。

CART 与 C4.5 算法是非常相似的，但是 CART 支持预测连续的值（回归）。CART 构建二叉树，而 C4.5 则不一定。

CART 用训练集和交叉验证集不断地评估决策树的性能来修剪决策树，从而使训练误差和测试误差达到一个很好地平衡点

六、 支持向量机

（一） 线性支持向量机

1. 理论介绍

在机器学习中，支持向量机（常简称为 SVM，又名支持向量网络）是在分类与回归分析中分析数据的监督式学习模型与相关的学习算法。给定一组训练实例，每个训练实例被标记

为属于两个类别中的一个或另一个，SVM 训练算法创建一个将新的实例分配给两个类别之一的模型，使其成为非概率二元线性分类器。SVM 模型是将实例表示为空间中的点，这样映射就使得单独类别的实例被尽可能宽的明显的间隔分开。然后，将新的实例映射到同一空间，并基于它们落在间隔的哪一侧来预测所属类别。

给定训练样本集 D=$\{(x_1,y_1),(x_2,y_2),\cdots,(x_m,y_m)\}, y_i \in \{-1,+1\}$，分类学习最基本的想法就是基于训练集 D 在样本空间中找到一个划分超平面，将不同类别的样本分开。但能将训练样本分开的划分超平面可能有很多，我们应该去找到哪一个呢？

直观上看，应该去找到位于两类训练样本“正中间”的划分超平面。在样本空间中，划分超平面可通过如下线性方程来描述：

$$w^T x + b = 0 \qquad ①$$

其中$\mathrm{w} = (w_1, w_2, \cdots, w_d)$为法向量，决定了超平面的方向，b 为位移项，决定了超平面与原点之间的距离。划分超平面可被法向量 w 和位移 b 确定，下面我们将其记为(w,b)。样本空间中任意点 x 到超平面(w,b)的距离可写为

$$\gamma = \frac{|w^T x + b|}{||w||} \qquad ②$$

假设超平面(w,b)能将训练样本正确分类，即对于$(x_i,y_i) \in \mathrm{D}$，若$y_i = +1$，则有$w^T x_i + b > 0$；若$y_i = -1$，则有$w^T x_i + b < 0$。令

$$\begin{cases} w^T x_i + b \geq 1, & y_i = +1 \\ w^T x_i + b \leq 1, & y_i = -1 \end{cases} \qquad ③$$

距离超平面最近的几个训练样本点使上式的等号成立，他们被称为“支持向量”，两个异类支持向量到超平面的距离之和为

$$\gamma = \frac{2}{||w||} \qquad ④$$

上式被称为间隔。

预找到具有“最大间隔”的划分超平面，也就是要找到能满足式③中约束的参数 w 和 b，使得γ最大，即

$$\max_{w,\ b} \frac{2}{||w||} \qquad ⑤$$

$$\text{s.t } y_i(w^T x_i + b) \geq 1, \quad i = 1,2,\cdots,m$$

显然，为了最大化间隔，仅需最大化$\frac{1}{||w||}$，这等价于最小化$||w||^2$。于是，上式可以重写为

$$\min_{w,b} \frac{1}{2}||w||^2 \qquad ⑥$$

$$\text{s.t } y_i(w^T x_i + b) \geq 1, \quad i = 1,2,\cdots,m$$

这就是支持向量机的基本型。

对上式使用拉格朗日乘子法可以得到其“对偶问题”。具体来说，对式⑥的每条约束添

加拉格朗日乘子$\alpha_i \geq 0$，则该问题的拉格朗日函数可以写为

$$\mathrm{L}(\mathrm{w},\mathrm{b},\alpha) = \frac{1}{2}||w||^2 + \sum_{i=1}^{m} \alpha_i(1 - y_i(w^T x_i + b)) \qquad ⑦$$

其中$\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_m)$。令$\mathrm{L}(\mathrm{w},\mathrm{b},\alpha)$对 w 和 b 的偏导为零可得

$$\mathrm{w} = \sum_{i=1}^{m} \alpha_i y_i x_i \qquad ⑧$$

$$0 = \sum_{i=1}^{m} \alpha_i y_i \qquad ⑨$$

将式⑧代入⑦，即可将$\mathrm{L}(\mathrm{w},\mathrm{b},\alpha)$中的 w 和 b 消去，再考虑式⑨的约束，就得到式⑤的对偶问题

$$\max_{\alpha} \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j x_i^T x_j \qquad ⑩$$

$$\text{s.t. } \sum_{i=1}^{m} \alpha_i y_i = 0，\ \alpha_i \geq 0，\ \mathrm{i} = 1,2,\cdots,\mathrm{m}$$

解出α后，求出 w 与 b 即可得到模型

$$\mathrm{f}(\mathrm{x}) = w^T x + b = \sum_{i=1}^{m} \alpha_i y_i x_i^T x + b \qquad （11）$$

从对偶问题⑩解出的α_i是拉格朗日函数中的拉格朗日乘子，它恰对应着训练样本(x_i, y_j)。注意到式⑤中有不等式约束，因此上述过程满足 KKT 条件，即要求

$$\begin{cases} \alpha_i \geq 0 \\ y_i f(x_i) - 1 \geq 0 \\ \alpha_i(y_i f(x_i) - 1) = 0 \end{cases} \qquad (12)$$

于是，对任意训练样本(x_i, y_j)，总有$\alpha_i = 0$或$y_i f(x_i) = 1$。若$\alpha_i = 0$，则该样本将不会在式（11）的求和中出现，也就不会对 f(x)有任何影响；若$\alpha_i > 0$，则必有$y_i f(x_i) = 1$，所对应的样本点位于最大间隔边界上，是一个支持向量。这显示出支持向量机的一个重要性质：训练完成后，大部分的训练样本都不需保留，最终模型仅与支持向量有关。

那么，如何求解式⑩呢？不难发现，这是一个二次规划问题，可使用通用的二次规划算法来求解；然而，该问题的规模正比于训练样本数，人们通过利用问题本身的特性，提出了很多高效算法，SMO 是其中一个著名的代表。

优点：① 应用范围广，有良好的数学解释能力；② 对于线性可分的数据而言，分类效果较好。

缺点：① 对于线性不可分的数据而言，线性支撑向量机将失去作用。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过建立线性支持向量机模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入乳腺癌数据集，数据划分及标准化处理与
logistics 回归处理相同
# 导入支持向量机函数：核函数为线性核
from sklearn.svm import SVC
model_svc=SVC(kernel='linear')
```

```
# 模型训练及预测
model_svc=model_svc.fit(X_train,y_train)
y_pred=model_svc.predict(X_test)
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("线性支持向量机模型拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
# 导入支持向量机函数：核函数为线性核
from sklearn.svm import SVC
model_svc=SVC(kernel='linear')
# 模型训练及预测
model_svc=model_svc.fit(X_train,y_train)
y_pred=model_svc.predict(X_test)
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("线性支持向量机模型拟合的准确率为:%.3f%%"% (Acc *100))
```

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        52
           1       1.00      0.98      0.99        91

    accuracy                           0.99       143
   macro avg       0.98      0.99      0.99       143
weighted avg       0.99      0.99      0.99       143

线性支持向量机模型拟合的准确率为:98.601%
```

（二） 核支持向量机

1. 理论介绍

当遇到线性不可分的情况时，线性支撑向量机将失去用处，这时我们可以引入核函数，将线性不可分的数据映射到高维特征空间中，使其变成线性可分。即如果不存在一个能正确划分两类样本的超平面时，我们可以将样本从原始空间映射到一个更高维的特征空间，使得样本在这个特征空间内线性可分。但是由于映射到高维后的计算成本将大幅增加，所以利用核函数来进行简化。当不知道应该选择什么样的核函数时，通常采用线性核或高斯核，但是如果核函数选择不好，很可能导致支持向量机的性能不佳。

优点：① 应用范围广，可以处理线性不可分的任务；② 选择不同的核函数可以处理不同类型的数据。

缺点：① 需要选取一个合适的核函数，需要反复调整参数选择一个最优的模型；② 计算效率低，运算量高。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过建立核支持向量机模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入乳腺癌数据集，数据划分及标准化处理与 logistics 回归处理相同
# 导入支持向量机函数：核函数为 RBF 核
from sklearn.svm import SVC
model_svc_rbf=SVC(kernel='rbf')
# 模型训练及预测
model_svc_rbf=model_svc_rbf.fit(X_train,y_train)
y_pred=model_svc_rbf.predict(X_test)
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("RBF 核支持向量机模型拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
# 导入支持向量机函数：核函数为RBF核
from sklearn.svm import SVC
model_svc_rbf=SVC(kernel='rbf')
# 模型训练及预测
model_svc_rbf=model_svc_rbf.fit(X_train,y_train)
y_pred=model_svc_rbf.predict(X_test)
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("RBF核支持向量机模型拟合的准确率为:%.3f%%"% (Acc *100))
```

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98        52
           1       1.00      0.98      0.99        91

    accuracy                           0.99       143
   macro avg       0.98      0.99      0.99       143
weighted avg       0.99      0.99      0.99       143

RBF核支持向量机模型拟合的准确率为:98.601%
```

七、 神经网络

（一） BP 神经网络

1. 理论介绍

人工神经网络无需事先确定输入输出之间映射关系的数学方程，仅通过自身的训练，学习某种规则，在给定输入值时得到最接近期望输出值的结果。作为一种智能信息处理系统，人工神经网络实现其功能的核心是算法。BP 神经网络是一种按误差反向传播（简称误差反传）训练的多层前馈网络，其算法称为 BP 算法，它的基本思想是梯度下降法，利用梯度搜索技术，以期使网络的实际输出值和期望输出值的误差均方差为最小。

基本 BP 算法包括信号的前向传播和误差的反向传播两个过程。即计算误差输出时按从输入到输出的方向进行，而调整权值和阈值则从输出到输入的方向进行。正向传播时，输入信号通过隐含层作用于输出节点，经过非线性变换，产生输出信号，若实际输出与期望输出不相符，则转入误差的反向传播过程。误差反传是将输出误差通过隐含层向输入层逐层反传，并将误差分摊给各层所有单元，以从各层获得的误差信号作为调整各单元权值的依据。通过调整输入节点与隐层节点的联接强度和隐层节点与输出节点的联接强度以及阈值，使误差沿梯度方向下降，经过反复学习训练，确定与最小误差相对应的网络参数(权值和阈值)，训练即告停止。此时经过训练的神经网络即能对类似样本的输入信息，自行处理输出误差最小的经过非线形转换的信息。

BP 网络是在输入层与输出层之间增加若干层(一层或多层)神经元，这些神经元称为隐单元，它们与外界没有直接的联系，但其状态的改变，则能影响输入与输出之间的关系，每一层可以有若干个节点。

BP 神经网络的计算过程由正向计算过程和反向计算过程组成。正向传播过程，输入模式从输入层经隐单元层逐层处理，并转向输出层，每～层神经元的状态只影响下一层神经元的状态。如果在输出层不能得到期望的输出，则转入反向传播，将误差信号沿原来的连接通路返回，通过修改各神经元的权值，使得误差信号最小。

对每个训练样例，BP 算法执行以下操作：先将输入示例提供给输入神经元，然后逐层将信号前传，直到产生输出层的结果；然后计算输出层的误差，再将误差逆向传播至隐层神经元，最后根据隐层神经元的误差来对连接权和阈值进行调整。该迭代过程循环进行，知道达到某些停止条件为止，例如训练误差达到一个很小的值。

需注意的是，BP 算法的目标是要最小化训练集 D 上的累计误差

$$\mathrm{E} = \frac{1}{m}\sum_{k=1}^{m} E_k$$

但我们上面介绍的“标准 BP 算法”每次仅针对一个训练样例更新连接权和阈值，也就是说，该算法的更新规则是基于单个的E_k推到而得。如果类似地推导出基于累积误差最小化的更新规则，就得到了累积误差逆传播算法。累积 BP 算法标准 BP 算法都很常用。一

般来说，标准 BP 算法每次更新只针对单个样例，参数更新的非常频繁，而且对不同样例进行更新的效果可能出现“抵消”现象。因此，为了达到同样的累积误差极小点，标准 BP 算法往往需进行更多次数的迭代。累积 BP 算法直接针对累积误差最小化，它在读取整个训练集 D 一遍后才对参数进行更新，其参数的更新的频率低很多。但是在很多任务中，累计误差下降到一定程度后，进一步下降会非常缓慢，这时标准 bp 往往会更快获得较好的解，尤其是在训练集 D 非常大时更明显。

优点：① 有很强的非线性映射能力和柔性的网络结构；② 网络的中间层数、各层的神经元个数可根据具体情况任意设定，并且结构多变，表达能力强。

缺点：① 学习速度慢，即使是一个简单的问题，一般也需要几百次甚至上千次的学习才能收敛；② 容易陷入局部最小值；③ 网络层数、神经元个数的选择没有理论指导；④ 网络推广能力有限。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过建立 BP 神经网络模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入乳腺癌数据集，数据划分及标准化处理与 logistics 回归处理相同
# 导入定义神经网络所需的函数
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import SGD
# 自定义三层神经网络模型
model = Sequential([
    Dense(64, input_dim=30,activation="relu"),
    Dense(32,activation="relu"),
    Dense(1,activation="sigmoid")
])
print(model.summary())
# 模型编译
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy',optimizer=sgd,
              metrics=['accuracy'])
# 模型训练及评价
model.fit(X_train, y_train,epochs=500,batch_size=32,verbose=0)
loss,Acc= model.evaluate(X_test, y_test, batch_size=32)
print("BP 神经网络模型拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

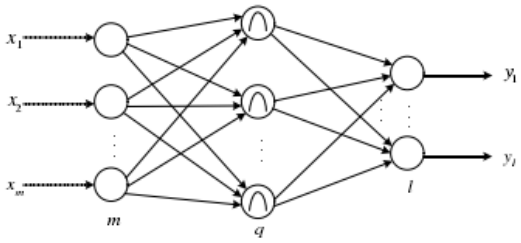
```
#  导入定义神经网络所需的函数
from  keras.models  import  Sequential
from  keras.layers  import  Dense,  Dropout
from  keras.optimizers  import  SGD
#  自定义三层神经网络模型
model  =  Sequential([
        Dense(64,  input_dim=30,activation="relu"),
        Dense(32,activation="relu"),
        Dense(1,activation="sigmoid")
])
print(model.summary())
#  模型编译
sgd  =  SGD(lr=0.01,  decay=1e-6,  momentum=0.9,  nesterov=True)
model.compile(loss='binary_crossentropy',optimizer=sgd,
                    metrics=['accuracy'])
#  模型训练及评价
model.fit(X_train,  y_train,epochs=500,batch_size=32,verbose=0)
loss,Acc=  model.evaluate(X_test,  y_test,  batch_size=32)
print("BP神经网络模型拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
Using TensorFlow backend.
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                1984
_________________________________________________________________
dense_2 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 33
=================================================================
Total params: 4,097
Trainable params: 4,097
Non-trainable params: 0
_________________________________________________________________
None
143/143 [==============================] - 0s 161us/step
BP神经网络模型拟合的准确率为:97.902%
```

（二） RBF 神经网络

1. 理论介绍

RBF 神经网络其结构如下图所示。第一层即输入层，它的主要作用是把输入变量传送到由径向基函数组成的隐含层，从而在隐含层中实现对输入变量的非线性变换，该网络的输出层也成为线性层，它的主要作用是对隐含层单元的输出进行线性变换从而得到整体的输出。



假设 RBF 神经网络从输入到输出三层的节点数分别为 m、q、l。RBF 神经网络的隐含层中常用的作用函数由以下这些方式组成，

$$u_i(x) = exp[-(x^T x/\delta_i^2)]$$

$$u_i(x) = 1/(\delta_i^2 + x^T x)^{\alpha}, \alpha > 0$$

$$u_i(x) = (\delta_i^2 + x^T x)^{\beta}, 0 < \beta < 1$$

其中高斯函数是一种应用最广泛的径向基函数，此时径向基函数神经网络隐含层的输出如下所示

$$u_i(x) = R(||x - c_i||) = \exp\left[-\frac{(x - c_i)^T (x - c_i)}{2\sigma_i^2}\right]$$

式中，$i = 1,2,\cdots,q$；u_i是第 i 个隐节点的输出；σ_i表示基函数的方差；$x = [x_1, x_2, \cdots x_m]^T$为输入变量；$c_i = [c_{i1}, c_{i2}, \cdots, c_{im}]^T$为高斯函数的中心向量；$||x - c_i||$为欧式范数。

RBF 神经网络输出层的主要作用是对输进输出层的信号进行线性变换，从而可以把隐含层的输出通过对线性映射作为整体的输出，如下式所示

$$y_k = \sum_{i=1}^{q} w_{ki}\, u_i - \theta_k$$

式中，k 表示节点数，y_k表示网络输出层的输出；w_{ki}为网络的加权系数；θ_k表示输出层的阈值。

RBF 神经网络和 BP 神经网络的区别：① 中间层神经元的区别：RBF 神经网络的神经元是一个以径向基函数为作用函数的神经元，而 BP 神经网络往往使用 Sigmoid 函数作为作用函数；② 中间层数的区别：RBF 神经网络只有一层隐层，而 BP 神经网络可能有多层的隐层；③ 运行速度的区别：RBF 神经网络由于层数较少，参数也比 BP 神经网络的少，所以其运算速度快很多。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过建立 RBF 神经网络模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入乳腺癌数据集，数据划分及标准化处理与
logistics 回归处理相同
```

```
from keras.layers import Layer
from keras import backend as K
# 自定义 RBF 激活函数
class RBFLayer(Layer):
    def __init__(self, units, gamma, **kwargs):
        super(RBFLayer, self).__init__(**kwargs)
        self.units = units
        self.gamma = K.cast_to_floatx(gamma)

    def build(self, input_shape):
        self.mu = self.add_weight(name='mu',
                  shape=(int(input_shape[1]), self.units),
                  initializer='uniform',
                  trainable=True)
        super(RBFLayer, self).build(input_shape)

    def call(self, inputs):
        diff = K.expand_dims(inputs) - self.mu
        l2 = K.sum(K.pow(diff,2), axis=1)
        res = K.exp(-1 * self.gamma * l2)
        return res

    def compute_output_shape(self, input_shape):
        return (input_shape[0], self.units)
# 自定义二层 RBF 神经网络
model = Sequential()
model.add(Dense(16, input_shape=(30,)))
model.add(RBFLayer(16,0.5))
model.add(Dense(1,activation="sigmoid"))
print(model.summary())
# 模型编译
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])
# 模型训练及评价
model.fit(X_train, y_train,epochs=500,batch_size=32,verbose=0)
loss,Acc= model.evaluate(X_test, y_test, batch_size=32)
print("RBF 神经网络模型拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
#  自定义二层RBF神经网络
model  =  Sequential()
model.add(Dense(16,  input_shape=(30,)))
model.add(RBFLayer(16,0.5))
model.add(Dense(1,activation="sigmoid"))
print(model.summary())
#  模型编译
sgd  =  SGD(lr=0.01,  decay=1e-6,  momentum=0.9,  nesterov=True)
model.compile(loss='binary_crossentropy',
                    optimizer=sgd,
                    metrics=['accuracy'])
#  模型训练及评价
model.fit(X_train,  y_train,epochs=500,batch_size=32,verbose=0)
loss,Acc=  model.evaluate(X_test,  y_test,  batch_size=32)
print("RBF神经网络模型拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 16)                496
_________________________________________________________________
rbf_layer_1 (RBFLayer)       (None, 16)                256
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 17
=================================================================
Total params: 769
Trainable params: 769
Non-trainable params: 0
_________________________________________________________________
None
143/143 [==============================] - 0s 164us/step
RBF神经网络模型拟合的准确率为:97.902%
```

（三） GRNN

1. 理论介绍

GRNN 是径向基网络的另外一种变形形式。GRNN 建立在非参数回归的基础上，以样本数据为后验条件，执行 Parzen 非参数估计，依据最大概率原则计算网络输出。广义回归网路以径向基网络为基础，因此具有良好的非线性逼近性能，与径向基网络相比，训练更为方便。

假设 x、y 是两个随机变量，其联合概率密度为 f(x,y)，若已知 x 的观测值为x_0，y 相对 x 的回归为

$$E(y|x_0)=\frac{\int_{-\infty}^{0} yf(x_0,y)dy}{\int_{-\infty}^{0} f(x_0,y)dy}$$

$y(x_0)$即为在输入为x_0的条件下，y 的预测输出。应用 Parzen 非参数估计，可由样本数据集$\{x_i,y_i\}_{i=1}^{n}$按下式估算密度函数$f(x_0,y)$:

$$\mathrm{f}(x_0,y)=\frac{1}{n(2\pi)^{(p+1)/2}\sigma^{p+1}}\sum_{i=1}^{n}e^{-d(x_0,x_i)}\,e^{-d(y_0,y_i)}$$

$$d(x_0,x_i)=\sum_{j=1}^{p}[(x_{0j},x_{ij})/\sigma]^2\,,d(y,y_i)=[y-y_i]^2$$

式中 n 为样本容量，p 为随机变量 x 的维数。σ称光滑因子，实际上就是高斯函数的标准差。将上上式代入，并交换积分与求和的顺序，有：

$$\mathrm{y}(x_0)=\frac{\sum_{i=1}^{n}(e^{-d(x_0,x_i)}\int_{-\infty}^{+\infty}y\,e^{-d(y_0,y_i)}dy)}{\sum_{i=1}^{n}(e^{-d(x_0,x_i)}\int_{-\infty}^{+\infty}e^{-d(y_0,y_i)}\,dy)}$$

由于$\int_{-\infty}^{+\infty}xe^{-x^2}dx=0$，化简上式，可得

$$\mathrm{y}(x_0)=\frac{\sum_{i=1}^{n}ye^{-d(y_0,y_i)}}{\sum_{i=1}^{n}e^{-d(y_0,y_i)}}$$

显然，在上式中，分子为所有训练样本算得的y_i值的加权和，权值为$e^{-d(y_0,y_i)}$。这里需要注意的是光滑因子σ的取值，广义回归神经网络不需要训练，但光滑因子的值对网络性能影响很大，需要优化取值。Specht 提出的 GRNN 对所有隐含层神经元的基函数采用相同的光滑因子，因子网络的训练过程只需完成对σ的一维寻优即可。若光滑因子取值非常大，$d(x_0,x_i)$趋近于零，$\mathrm{y}(x_0)$近似于所有样本因变量的平均值。若光滑因子趋近于零，则$\mathrm{y}(x_0)$与训练样本的值非常接近，但一旦给定新的输入，预测效果就会急剧变差，使网络失去推广能力，这种现象称为过学习。确定一个适中的光滑因子值时，所有的训练样本的因变量都被考虑了进去，但又考虑了不同训练样本点与测试输入样本的距离，离测试样本近的训练样本会被赋予更大的权值。

GRNN 在结构上与 RBF 网络较为相似。它由四层构成，分别为输入层、模式层、求和层和输出层。对应网络输入$\mathrm{X}=[x_1,x_2,\cdots,x_n]^T$，其输出为$\mathrm{Y}=[y_1,y_2,\cdots,y_k]^T$。

（a） 输入层

输入层神经元的数目等于学习样本中输入向量的维数，各神经元是简单的分布单元，直接将输入变量传递给模式层

（b） 模式层

模式层神经元数目等于学习样本的数目 n，各神经元对应不同的样本，模式层神经元传递函数为

$$p_i=exp\left[-\frac{(X-X_i)^T(X-X_i)}{2\sigma^2}\right]\quad i=1,2,\cdots,n$$

神经元 i 的输出为输入变量与其对应的样本 X 之间 Euclid 距离平方的指数平方$D_i^2=(X-X_i)^T(X-X_i)$的指数形式。式中，X 为网络输入变量；X_i为第 i 个神经元对应的学习样本。

（c） 求和层

求和层中使用两种类型神经元进行求和。

一类的计算公式为$\sum_{i=1}^{n} exp\left[-\frac{(X-X_i)^T(X-X_i)}{2\sigma^2}\right]$，他对所有的模式层神经元的输出进行算术求和，其模式层与各神经元的连接权值为 1，传递函数为

$$S_D = \sum_{i=1}^{n} P_i$$

令一类计算公式为$\sum_{i=1}^{n} Y_i exp\left[-\frac{(X-X_i)^T(X-X_i)}{2\sigma^2}\right]$，它对所有模式层的神经元进行加权求和，模式层中第 i 个神经元与求和层中第 j 个分子求和神经元之间的连接权值为第 i 个输出样本Y_i中的第 j 个元素，传递函数为

$$S_{Nj} = \sum_{i=1}^{n} y_{ij} P_i \quad j = 1,2,\cdots,k$$

（d） 输出层

输出层中的神经元数目等于学习样本中输出向量的维数 k，各神经元将求和层的输出相除，神经元 j 的输出对应估计结果$\hat{Y}(X)$的第 j 个元素，即

$$y_j = \frac{S_{Nj}}{S_D} \quad j = 1,2,\cdots,k$$

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的波士顿房价数据集，该数据集有 13 个特征变量，1 个连续型的目标变量，希望通过建立 GRNN 神经网络模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入波士顿房价数据集，数据划分及标准化处理与岭回归处理相同
import numpy as np
import pandas as pd

# 自定义广义回归神经网络模型（GRNN）
class GRNN(object):
    def __init__(self,X_train,y_train,X_test,y_test):
        self.x_train= X_train
        self.y_train= y_train
        self.x_test= X_test
        self.y_test= y_test
        #np.random.rand(1,self.train_y.size) #Standard deviations(std) are sometimes called RBF widths.
        self.std = np.ones((1,self.y_train.size))
```

```
    def activation_func(self,distances): # gaussian kernel

        return np.exp(- (distances**2) / 2*(self.std**2) )

    def output(self,i):#sometimes called weight
        distances=np.sqrt(np.sum((self.x_test[i]-
self.x_train)**2,axis=1)) # euclidean distance
        return self.activation_func(distances)

    def denominator(self,i):
        return np.sum(self.output(i))

    def numerator(self,i):
        return  np.sum(self.output(i) * self.y_train)

    def predict(self):
        predict_array = np.array([])
        for i in range(self.y_test.size):
            predict=np.array([self.numerator(i)/self.denominat
or(i)])
            predict_array=np.append(predict_array,predict)

        return predict_array

    def squared_error(self):
        return (self.predict()-self.y_test)**2

    def root_squared_error(self):
        return np.sqrt(self.squared_error())
# 导入自定义的 GRNN，并进行训练和预测
model_grnn=GRNN(X_train,y_train,X_test,y_test)
y_pred=model_grnn.predict()
# 模型评价
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
MSE=mean_squared_error(y_pred,y_test)
r2=r2_score(y_pred,y_test)
print("GRNN 模型拟合的均方误差为:{}".format(round(MSE,3)))
print("GRNN 模型自定义的 MSE
为:%.3f"%(np.mean(model_grnn.squared_error())))
print("GRNN 可以解释原变量%.3f%%的信息"% (r2 *100))
```

（3） 程序结果：

```
#  导入自定义的GRNN，并进行训练和预测
model_grnn=GRNN(X_train, y_train, X_test, y_test)
y_pred=model_grnn.predict()
#  模型评价
from  sklearn.metrics  import  mean_squared_error
from  sklearn.metrics  import  r2_score
MSE=mean_squared_error(y_pred, y_test)
r2=r2_score(y_pred, y_test)
print("GRNN模型拟合的均方误差为:{}".format(round(MSE, 3)))
print("GRNN模型自定义的MSE为:%.3f"%(np.mean(model_grnn.squared_error())))
print("GRNN可以解释原变量%.3f%%的信息"%  (r2  *100))
```

```
GRNN模型拟合的均方误差为:23.402
GRNN模型自定义的MSE为:23.402
GRNN可以解释原变量53.297%的信息
```

八、 聚类分析

（一） K-Means 聚类

1. 理论介绍

K-Means 聚类算法是一种迭代求解的聚类分析算法，其步骤是，预将数据分为 K 组，则随机选取 K 个对象作为初始的聚类中心，然后计算每个对象与各个种子聚类中心之间的距离，把每个对象分配给距离它最近的聚类中心。聚类中心以及分配给它们的对象就代表一个聚类。每分配一个样本，聚类的聚类中心会根据聚类中现有的对象被重新计算。这个过程将不断重复直到满足某个终止条件。终止条件可以是没有（或最小数目）对象被重新分配给不同的聚类，没有（或最小数目）聚类中心再发生变换，误差平方和局部最小。

建模流程：先随机选取 K 个对象作为初始的聚类中心，然后计算每个对象与各个种子聚类中心之间的距离，把每个对象分配给距离它最近的聚类中心。聚类中心以及分配给它们的对象就代表一个聚类。一旦全部对象都被分配了，每个聚类的聚类中心会根据聚类中现有的对象被重新计算。这个过程将不断重复直到满足某个终止条件。终止条件可以是以下任何一个：①没有（或最小数目）对象被重新分配给不同的聚类；②没有（或最小数目）聚类中心再发生变换；③误差平方和局部最小。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的 iris 数据集，该数据集有 4 个特征变量，1 个三分类的目标变量，希望通过建立 K-Means 模型来对特征变量进行聚类分析。

（2） 代码实现：

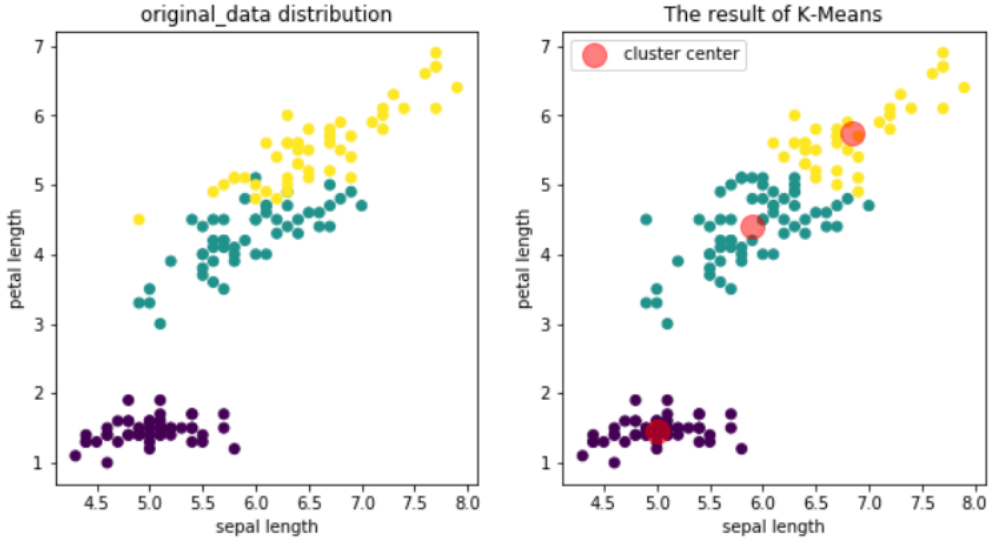
```
# 使用 python 自带的鸢尾花数据集（三分类）
from sklearn.datasets import load_iris
X,y=load_iris(return_X_y=True)
print("样本的个数 = {}, 特征的个
数 = {}".format(X.shape[0],X.shape[1]))
```

```
print("特征描述：{}".format(load_iris().feature_names))
# 画出 sepal、petal length 特征的分布图
plt.subplot(1,2,1)
plt.scatter(X[:,0],X[:,2],c=y)
plt.xlabel('sepal length')
plt.ylabel('petal length')
plt.title('original_data distribution')
# 进行 K-Means 聚类分析
from sklearn.cluster import KMeans
model_kmeans=KMeans(n_clusters=3)
model_kmeans=model_kmeans.fit(X)
y_kmeans=model_kmeans.labels_
# 聚类结果评价：同质性得分
from sklearn.metrics.cluster import homogeneity_score
score=homogeneity_score(y,y_kmeans)
print("K-Means 同质性得分为:{}".format(round(score,3)))
# 画出聚类后的结果
plt.subplot(1,2,2)
plt.scatter(X[:,0],X[:,2],c=y_kmeans)
centers=model_kmeans.cluster_centers_
plt.scatter(centers[:,0],centers[:,2],c="red",s=200,alpha=0.5,
label="cluster center")
plt.xlabel('sepal length')
plt.ylabel('petal length')
plt.title('The result of K-Means')
plt.legend(loc='best')
plt.show()
```

（3） 程序结果：

```
#  聚类结果评价：同质性得分
from  sklearn.metrics.cluster  import  homogeneity_score
score=homogeneity_score(y,y_kmeans)
print("K-Means同质性得分为:{}".format(round(score,3)))
#  画出聚类后的结果
plt.subplot(1,2,2)
plt.scatter(X[:,0],X[:,2],c=y_kmeans)
centers=model_kmeans.cluster_centers_
plt.scatter(centers[:,0],centers[:,2],c="red",s=200,alpha=0.5,label="cluster  center")
plt.xlabel('sepal  length')
plt.ylabel('petal  length')
plt.title('The  result  of  K-Means')
plt.legend(loc='best')
plt.show()
```

```
样本的个数 = 150, 特征的个数 = 4
特征描述：['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
K-Means同质性得分为:0.751
```



（二） 系统聚类

1. 理论介绍

与 K-Means 聚类不同的是，系统聚类不需要预先设定聚类个数 K，系统聚类的做法是开始时把买个样品作为一类，然后把最靠近的样品（即距离最小的样本）首先聚为小类，再将已聚合的小类按其类间距离再合并，不断继续下去，最后把所有子类都聚合到一个大类。

常用的系统聚类法是以距离为相似统计量，确定新类与其他各类之间距离的方法，如最短距离法、最长距离法、中间距离法、重心法、群平均法、离差平方和法、欧式距离等。

确定了距离和相似系数后就要进行分类。分类有许多种方法，最常用的一种方法是在样品距离的基础上定义类与类之间的距离。首先将 n 个样品分成 n 类，每个样品自成一类，然后每次将具有最小距离的两类合并，合并后重新计算类与类之间的距离，这个过程一直持续到所有的样品归为一类为止，并把这个过程画成一张聚类图，参照聚类图可方便地进行分类。因为聚类图很像一张系统图，所以这种方法就叫系统聚类法。系统聚类法是在实际中使用最多的一种方法，从上面的分析可以看出，虽然我们已给出了计算样品之间距离的方法，但在实际计算过程中还要定义类与类之间的距离。定义类与类之间的距离也有许多方法，不同的

方法就产生了不同的系统聚类方法，常用的有如下六种：

① 最短距离法：类与类之间的距离等于两类最近样品之间的距离；

② 最长距离法：类与类之间的距离等于两类最远样品之间的距离；

③ 类平均法：类与类之间的距离等于各类元素两两之间的平方距离的平均；

④ 重心法：类与类之间的距离定义为对应这两类重心之间的距离对样品分类来说，每一类的类中心就是该类样品的均值；

⑤ 中间距离法：最长距离法夸大了类间距离，最短距离法低估了类间距离介于两者间的距离法即为中间距离法，类与类之间的距离既不采用两类之间最近距离，也不采用最远距离，而是采用介于最远和最近之间的距离；

⑥ 离差平方和法：基于方差分析的思想，如果分类正确，同类样品之间的离差平方和应当较小，类与类之间的离差平方和应当较大。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的 iris 数据集，该数据集有 4 个特征变量，1 个三分类的目标变量，希望通过建立系统聚类模型来对特征变量进行聚类分析。

（2） 代码实现：

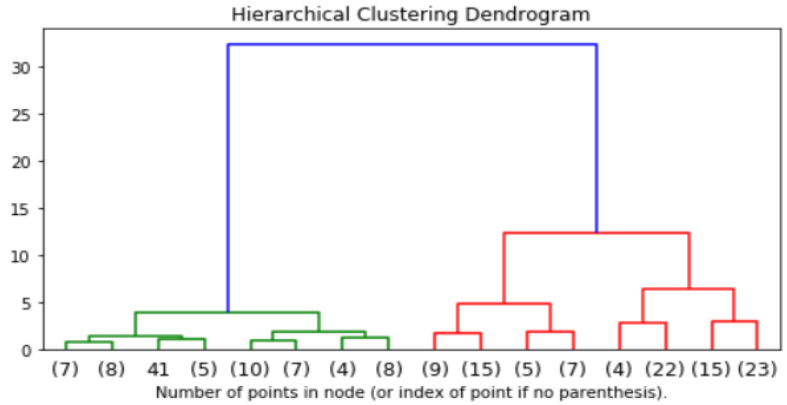
```
from scipy.cluster.hierarchy import dendrogram
# 定义谱系图
def plot_dendrogram(model, **kwargs):
    # Create linkage matrix and then plot the dendrogram
    # create the counts of samples under each node
    counts = np.zeros(model.children_.shape[0])
    n_samples = len(model.labels_)
    for i, merge in enumerate(model.children_):
        current_count = 0
        for child_idx in merge:
            if child_idx < n_samples:
                current_count += 1  # leaf node
            else:
                current_count += counts[child_idx - n_samples]
        counts[i] = current_count
    linkage_matrix = np.column_stack([model.children_, model.d
istances_,
                                      counts]).astype(float)
    # Plot the corresponding dendrogram
    dendrogram(linkage_matrix, **kwargs)
# 使用 python 自带的鸢尾花数据集（三分类）
from sklearn.datasets import load_iris
X,y=load_iris(return_X_y=True)
```

```
print("样本的个数 = {}, 特征的个
数 = {}".format(X.shape[0],X.shape[1]))
print("特征描述：{}".format(load_iris().feature_names))
# 导入系统聚类模型并训练
from sklearn.cluster import AgglomerativeClustering
model_Aggcluster = AgglomerativeClustering(distance_threshold=
0, n_clusters=None)
model_Aggcluster= model_Aggcluster.fit(X)
y_aggcluster=model_Aggcluster.labels_
# 聚类结果评价：同质性得分
from sklearn.metrics.cluster import homogeneity_score
score=homogeneity_score(y,y_aggcluster)
print("系统聚类同质性得分为:{}".format(round(score,3)))
# 根据聚类结果画出谱系图
plt.figure(figsize=(8,4))
plt.title('Hierarchical Clustering Dendrogram')
plot_dendrogram(model_Aggcluster, truncate_mode='level', p=3)
plt.xlabel("Number of points in node (or index of point if no
parenthesis).")
plt.show()
```

（3） 程序结果：

```
# 导入系统聚类模型并训练
from sklearn.cluster import AgglomerativeClustering
model_Aggcluster = AgglomerativeClustering(distance_threshold=0, n_clusters=None)
model_Aggcluster= model_Aggcluster.fit(X)
y_aggcluster=model_Aggcluster.labels_
# 聚类结果评价：同质性得分
from sklearn.metrics.cluster import homogeneity_score
score=homogeneity_score(y,y_aggcluster)
print("系统聚类同质性得分为:{}".format(round(score,3)))
# 根据聚类结果画出谱系图
plt.figure(figsize=(8,4))
plt.title('Hierarchical Clustering Dendrogram')
plot_dendrogram(model_Aggcluster, truncate_mode='level', p=3)
plt.xlabel("Number of points in node (or index of point if no parenthesis).")
plt.show()
```

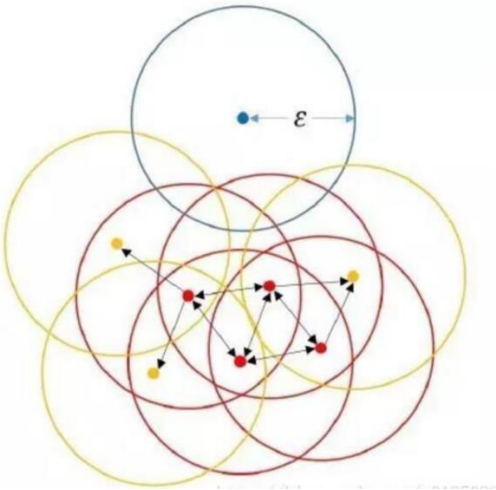
```
样本的个数 = 150, 特征的个数 = 4
特征描述：['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
系统聚类同质性得分为:1.0
```



（三） 密度聚类

1. 理论介绍

相比其他的聚类方法，基于密度的聚类方法可以在有噪音的数据中发现各种形状和各种大小的簇。DBSCAN 是该类方法中最典型的代表算法之一，其核心思想就是先发现密度较高的点，然后把相近的高密度点逐步都连成一片，进而生成各种簇。算法实现上，对每个数据点为圆心，以 eps 为半径画个圈（称为领域 eps-neigbourhood），然后数有多少个点在这个圈内，这个数就是该点密度值。然后我们可以选取一个密度阈值 MinPts，如圈内点数小于 MinPts 的圆心点为低密度的点，而大于或等于 MinPts 的圆心点为高密度点（称为核心点）。如果有一个高密度的点在另一个高密度的点的圈内，我们就把这两个点连接起来，这样我们可以把好多点不断地串联出来。之后，如果有低密度的点也在高密度的点的圈内，把它也连到最近的高密度点上，称之为边界点。这样所有能连到一起的点就成了一个簇，而不在任何高密度点的圈内的低密度点就是异常点，下图展示了 DBSCAN 的工作原理。



优点：① 可以对任意形状的稠密数据集进行聚类，相对的，K-Means 之类的聚类蒜贩一般只适用于凸数据集；② 可以在聚类的同时发现异常点，对数据集中的异常点不敏感；③ 聚类结果没有偏倚，相对的，K-Means 之类的聚类算法初始值对聚类结果有很大的影响。

缺点：① 如果样本集的密度不均匀、聚类间距差相差很大时，聚类质量较差，这时用 DBSCAN 聚类一般不适合；② 如果样本集较大时，聚类收敛时间较长，此时可以对搜索最近邻时建立的 KD 树或者球树进行规模限制来改进；③ 调参相对于传统的 K-Means 之类的聚类算法稍复杂，主要需要对距离阈值，领域样本数阈值 MinPts 联合调参，不同的参数组合对最后的聚类效果有较大影响。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的 iris 数据集，该数据集有 4 个特征变量，1 个三分类的目标变量，希望通过建立密度聚类模型来对特征变量进行聚类分析。

（2） 代码实现：

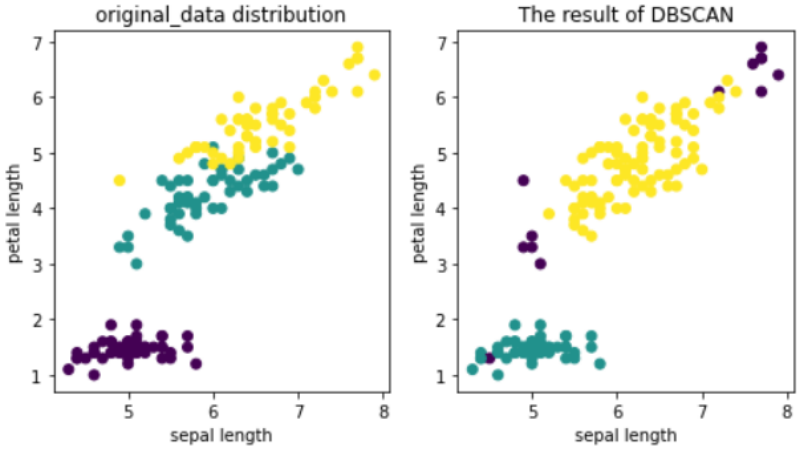
```
# 使用 python 自带的鸢尾花数据集（三分类）
```

```
from sklearn.datasets import load_iris
X,y=load_iris(return_X_y=True)
print("样本的个数 = {}, 特征的个
数 = {}".format(X.shape[0],X.shape[1]))
print("特征描述：{}".format(load_iris().feature_names))
# 画出 sepal、petal length 特征的分布图
plt.subplot(1,2,1)
plt.scatter(X[:,0],X[:,2],c=y)
plt.xlabel('sepal length')
plt.ylabel('petal length')
plt.title('original_data distribution')
# 导入密度聚类模型并训练
from sklearn.cluster import DBSCAN
model_dbscan=DBSCAN(eps=0.6, min_samples=10).fit(X)
y_dbscan=model_dbscan.labels_
core_samples_mask = np.zeros_like(y_dbscan, dtype=bool)
core_samples_mask[model_dbscan.core_sample_indices_] = True
n_clusters_ = len(set(y_dbscan)) - (1 if -
1 in y_dbscan else 0)
n_noise_ = list(y_dbscan).count(-1)
print('Estimated number of clusters: %d' % n_clusters_)
print('Estimated number of noise points: %d' % n_noise_)
# 聚类结果评价：同质性得分
from sklearn.metrics.cluster import homogeneity_score
score=homogeneity_score(y,y_dbscan)
print("K-Means 同质性得分为:{}".format(round(score,3)))
# 画出聚类后的结果
plt.subplot(1,2,2)
plt.scatter(X[:,0],X[:,2],c=y_dbscan)
plt.xlabel('sepal length')
plt.ylabel('petal length')
plt.title('The result of DBSCAN')
plt.show()
```

（3） 程序结果：

```
#  聚类结果评价：同质性得分
from  sklearn.metrics.cluster  import  homogeneity_score
score=homogeneity_score(y,y_dbscan)
print("K-Means同质性得分为:{}".format(round(score,3)))
#  画出聚类后的结果
plt.subplot(1,2,2)
plt.scatter(X[:,0],X[:,2],c=y_dbscan)
plt.xlabel('sepal  length')
plt.ylabel('petal  length')
plt.title('The  result  of  DBSCAN')
plt.show()
```

```
样本的个数 = 150, 特征的个数 = 4
特征描述: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Estimated number of clusters: 2
Estimated number of noise points: 13
K-Means同质性得分为:0.563
```



九、 集成学习

（一） Random-Forest

1. 理论介绍

随机森林（简称 RF）是 Bagging 的一个扩展变体。RF 在以决策树为基学习器构建 Bagging 集成的基础上，进一步在决策树的训练过程中引入了随机属性选择。具体来说，传统决策树在选择划分属性时是在当前结点的属性集合（假定有 d 个属性）中选择一个最优属性；而在 RF 中，对基决策树的每个结点，先从该结点的属性集合中随机选择一个包含 k 个属性的子集，然后再从这个子集中选择一个最优属性用于划分，这里的参数 k 控制了随机性的引入程度：若令 k=d，则基决策树的构建与传统决策树相同；若令 k=1，则是随机选择一个属性用于划分；一般情况下，推荐值$k = log_2 d$。

优点：① 随机森林简单、容易实现、计算开销小；② 性能强大，泛化性能高。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过建立随机森林模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入乳腺癌数据集，数据集的划分及标准化均与 logistic 回归相同
# 导入随机森林函数,使用二十棵决策树进行集成
from sklearn.ensemble import RandomForestClassifier
model_rf=RandomForestClassifier(n_estimators=20)
# 模型训练及预测
model_rf=model_rf.fit(X_train,y_train)
y_pred=model_rf.predict(X_test)
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("随机森林模型拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
#  导入随机森林函数,使用二十棵决策树进行集成
from  sklearn.ensemble  import  RandomForestClassifier
model_rf=RandomForestClassifier(n_estimators=20)
#  模型训练及预测
model_rf=model_rf.fit(X_train,y_train)
y_pred=model_rf.predict(X_test)
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred,y_test))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("随机森林模型拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           0       0.98      1.00      0.99        53
           1       1.00      0.99      0.99        90

    accuracy                           0.99       143
   macro avg       0.99      0.99      0.99       143
weighted avg       0.99      0.99      0.99       143

随机森林模型拟合的准确率为:99.301%
```

（二） Bagging

1. 理论介绍

Bagging 是并行式集成学习方法最著名的代表。从名字即可看出，它直接基于自助采样法。给定包含 m 个样本的数据集，我们先随机取出一个样本放入采样集中，再把该样本放回初始数据集，使得下次采样时该样本仍有可能被选中，这样，经过 m 次随机采样操作，我们得到含 m 个样本的采样集，初始训练集中有的样本再采样集里多次出现，有的则从未

出现。根据自助采样可知，初始训练集中约有 63.2%的样本出现在采样集中。

照这样，我们可采样出 T 个含 m 个训练样本的采样集，然后基于每个采样集训练出一个基学习器，再将这些基学习器进行结合，这就是 Bagging 的基本流程。在对预测输出进行结合时，Bagging 通常对分类任务使用简单投票法，对回归任务使用简单平均法。若分类预测时出现两个类收到同样票数的情形，则最简单的做法时随机选择一个，也可进一步考察学习器投票的置信度来确定最终获胜者。

事实上，包外样本还有许多其他用途。例如当基学习器是决策树时，可使用包外样本来辅助剪枝，或用于估计决策树中各结点的后验概率以辅助对岭训练样本结点的处理；当基学习器是神经网络时，可使用包外样本来辅助早期停止来减小过拟合风险。

从偏差-方差分解的角度看，Bagging 主要关注降低方差，因此它在不剪枝决策树、神经网络等易受样本扰动的学习器上效用更为明显。

优点：① 可用于多分类、回归任务；② 由于每个基学习器只使用了初始训练集约 63.2%的样本，剩下约 36.8%的样本可用作验证集来对泛化性能进行“包外估计”。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过建立 Bagging 模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入乳腺癌数据集，数据集的划分及标准化均与 logistic 回归相同
# 导入 BaggingClassifier 函数,使用 20 棵决策树作为基分类器
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
model_bag=BaggingClassifier(base_estimator=DecisionTreeClassifier(),n_estimators=20)
# 模型训练及预测
model_bag=model_bag.fit(X_train,y_train)
y_pred=model_bag.predict(X_test)
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("Bagging 模型拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
#  导入BaggingClassifier函数,使用20棵决策树作为基分类器
from  sklearn.tree  import  DecisionTreeClassifier
from  sklearn.ensemble  import  BaggingClassifier
model_bag=BaggingClassifier(base_estimator=DecisionTreeClassifier(),n_estimators=20)
#  模型训练及预测
model_bag=model_bag.fit(X_train,y_train)
y_pred=model_bag.predict(X_test)
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred,y_test))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("Bagging模型拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           0       0.98      1.00      0.99        53
           1       1.00      0.99      0.99        90

    accuracy                           0.99       143
   macro avg       0.99      0.99      0.99       143
weighted avg       0.99      0.99      0.99       143

Bagging模型拟合的准确率为:99.301%
```

（三） Adaboost

1. 理论介绍

AdaBoost 方法是一种迭代算法，在每一轮中加入一个新的弱分类器，直到达到某个预定的足够小的错误率。每一个训练样本都被赋予一个权重，表明它被某个分类器选入训练集的概率。如果某个样本点已经被准确地分类，那么在构造下一个训练集中，它被选中的概率就被降低；相反，如果某个样本点没有被准确地分类，那么它的权重就得到提高。通过这样的方式，AdaBoost 方法能“聚焦于”那些较难分（更富信息）的样本上。在具体实现上，最初令每个样本的权重都相等，对于第 K 次迭代操作，我们就根据这些权重来选取样本点，进而训练分类器C_k。然后就根据这个分类器，来提高被它分错的样本的权重，并降低被正确分类的样本权重。然后，权重更新过的样本集被用于训练下一个分类器C_k，整个训练过程如此迭代地进行下去。

AdaBoost 算法有多种推导方式，比较容易理解的是基于“加性模型”,即基学习器的线性组合

$$\mathrm{H(x)} = \sum_{t=1}^{T} \alpha_t \, h_t(x)$$

来最小化指数损失函数

$$l_{exp}(H|D) = E_{x\sim D}[e^{-f(x)\mathrm{H(x)}}]$$

若H(x)能令指数损失函数最小化，则考虑上式对H(x)的偏导

$$\frac{\partial l_{exp}(H|D)}{\partial \mathrm{H(x)}} = -e^{-\mathrm{H(x)}}P(f(x)=1|x) + e^{\mathrm{H(x)}}P(f(x)=-1|x)$$

令上式为零可解得

$$\mathrm{H(x)} = \frac{1}{2}ln\frac{P(f(x)=1|x)}{P(f(x)=-1|x)}$$

因此，有

$$\begin{aligned}\mathrm{sign(H(x))} &= \mathrm{sign}(\frac{1}{2}ln\frac{P(f(x)=1|x)}{P(f(x)=-1|x)}) \\ &= \begin{cases}1, & P(f(x)=1|x) > P(f(x)=-1|x) \\ -1, & P(f(x)=1|x) < P(f(x)=-1|x)\end{cases} \\ &= \arg\max_{y\in\{-1,+1\}} P(f(x)=y|x)\end{aligned}$$

这意味着sign(H(x))达到了贝叶斯最优错误率。换言之，若指数损失函数最小化，则分类错误率也将最小化；这说明指数损失函数是分类任务原本 $0/1$ 损失函数的一致的替代损失函数。由于这个替代函数有更好的数学性质，例如它是连续可微函数，因此我们用它替代 $0/1$ 损失函数作为优化目标.

在 AdaBoost 算法中，第一个基分类器h_1是通过直接将基学习算法用于初始数据分布而得;此后迭代地生成h_t和α_t，当基分类器h_t基于分布D_t产生后，该基分类器的权重α_t应使得$\alpha_t\mathrm{h}_t$最小化指数损失函数

$$\begin{aligned}l_{exp}(\alpha_t\mathrm{h}_t|D_t) &= E_{x\sim D}[e^{-f(x)\alpha_t\mathrm{h}_t(\mathrm{x})}] \\ &= E_{x\sim D}\left[e^{-\alpha_t}\amalg(f(x)=\mathrm{h}_t(\mathrm{x})) + e^{\alpha_t}\amalg(f(x)\neq\mathrm{h}_t(\mathrm{x}))\right] \\ &= e^{-\alpha_t}P_{x\sim D_t}(f(x)=\mathrm{h}_t(\mathrm{x})) + e^{\alpha_t}P_{x\sim D_t}(f(x)\neq\mathrm{h}_t(\mathrm{x})) \\ &= e^{-\alpha_t}(1-\epsilon_t) + e^{\alpha_t}\epsilon_t\end{aligned}$$

其中$\epsilon_t = P_{x\sim D_t}(f(x)\neq\mathrm{h}_t(\mathrm{x}))$。考虑指数损失函数的导数

$$\frac{\partial l_{exp}(\alpha_t\mathrm{h}_t|D_t)}{\partial\alpha_t} = -e^{-\alpha_t}(1-\epsilon_t) + e^{\alpha_t}\epsilon_t$$

令上式为零可解得

$$\alpha_t = \frac{1}{2}ln(\frac{1-\epsilon_t}{\epsilon_t})$$

AdaBoost 算法在获得H_{t-1}之后样本分布将进行调整，使下一轮的基学习器h_t能纠正H_{t-1}的一些错误。理想的h_t能纠正H_{t-1}的全部错误，即最小化

$$\begin{aligned}l_{exp}(\mathrm{H}_{t-1}+\mathrm{h}_t|D) &= E_{x\sim D}\left[e^{-f(x)(\mathrm{H}_{t-1}(x)+\mathrm{h}_t(x))}\right] \\ &= E_{x\sim D}\left[e^{-f(x)\mathrm{H}_{t-1}(x)}e^{-f(x)\mathrm{h}_t(x)}\right]\end{aligned}$$

注意到$f^2(x) = h_t^2(x) = 1$，上式可以用$e^{-f(x)\mathrm{h}_t(x)}$的泰勒展开式近似为

$$\begin{aligned}l_{exp}(\mathrm{H}_{t-1}+\mathrm{h}_t|D_t) &\cong E_{x\sim D}[e^{-f(x)\mathrm{H}_{t-1}(x)}(1-f(x)\mathrm{h}_t(x)+\frac{f^2(x)h_t^2(x)}{2})] \\ &= E_{x\sim D}[e^{-f(x)\mathrm{H}_{t-1}(x)}(1-f(x)\mathrm{h}_t(x)+\frac{1}{2})]\end{aligned}$$

于是，理想的基学习器

$$\mathrm{h}_t(x) = arg\min_{\mathrm{h}} l_{exp}(\mathrm{H}_{t-1} + \mathrm{h}|D)$$

$$= arg\min_{\mathrm{h}} E_{x\sim D}[e^{-f(x)\mathrm{H}_{t-1}(x)}(1 - f(x)\mathrm{h}(x) + \frac{1}{2})]$$

$$= arg\max_{h} E_{x\sim D}[e^{-f(x)\mathrm{H}_{t-1}(x)} f(x)\mathrm{h}(x)]$$

$$= arg\max_{h} E_{x\sim D}[\frac{e^{-f(x)\mathrm{H}_{t-1}(x)}}{E_{x\sim D}[e^{-f(x)\mathrm{H}_{t-1}(x)}]} f(x)\mathrm{h}(x)]$$

注意到$E_{x\sim D}[e^{-f(x)\mathrm{H}_{t-1}(x)}]$是一个常数。令$\mathrm{D}_t$表示一个分布

$$\mathrm{D}_t(\mathrm{x}) = \frac{D(x)e^{-f(x)\mathrm{H}_{t-1}(x)}}{E_{x\sim D}[e^{-f(x)\mathrm{H}_{t-1}(x)}]}$$

则根据数学期望的定义，这等价于令

$$\mathrm{h}_t(x) = arg\max_{\mathrm{h}} E_{x\sim D}[\frac{e^{-f(x)\mathrm{H}_{t-1}(x)}}{E_{x\sim D}[e^{-f(x)\mathrm{H}_{t-1}(x)}]} f(x)\mathrm{h}(x)]$$

$$= arg\max_{\mathrm{h}} E_{x\sim D}[f(x)\mathrm{h}(x)]$$

由$f(x)$，$\mathrm{h}(x) \in \{-1, +1\}$，有

$$f(x)\mathrm{h}(x) = 1 - 2\,\mathrm{II}\,(f(x) \neq \mathrm{h}_t(\mathrm{x}))$$

则理想的基学习器

$$\mathrm{h}_t(\mathrm{x}) = \arg\min_{h} E_{x\sim D}[\,\mathrm{II}\,(f(x) \neq \mathrm{h}_t(\mathrm{x}))]$$

由此可见，由此可见，理想的 ht 将在分布 A 下最小化分类误差.因此，弱分类器将基于分布来训练，且针对 A 的分类误差应小于 0.5.这在一定程度上类似“残差逼近”的思想.考虑到 A 和+1 的关系，有

$$\mathrm{D}_{t+1}(\mathrm{x}) = \frac{D(x)e^{-f(x)\mathrm{H}_t(x)}}{E_{x\sim D}[e^{-f(x)\mathrm{H}_t(x)}]}$$

$$= \frac{D(x)e^{-f(x)\mathrm{H}_{t-1}(x)}e^{-f(x)\alpha_t\mathrm{h}_t(x)}}{E_{x\sim D}[e^{-f(x)\mathrm{H}_t(x)}]}$$

$$= \mathrm{D}_t(\mathrm{x}) \cdot e^{-f(x)\alpha_t\mathrm{h}_t(x)} \frac{E_{x\sim D}[e^{-f(x)\mathrm{H}_{t-1}(x)}]}{E_{x\sim D}[e^{-f(x)\mathrm{H}_t(x)}]}$$

于是，我们从基于加性模型迭代式优化指数损失函数的角度推导出了 AdaBoost 算法。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过建立 AdaBoost 模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入乳腺癌数据集，数据集的划分及标准化均与 logistic 回归相同
# 导入 AdaBoost 函数,使用 20 棵决策树
from sklearn.ensemble import AdaBoostClassifier
model_ada=AdaBoostClassifier(n_estimators=20)
```

```
# 模型训练及预测
model_ada=model_ada.fit(X_train,y_train)
y_pred=model_ada.predict(X_test)
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("AdaBoost 模型拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
#  导入AdaBoost函数,使用20棵决策树
from  sklearn.ensemble  import  AdaBoostClassifier
model_ada=AdaBoostClassifier(n_estimators=20)
#  模型训练及预测
model_ada=model_ada.fit(X_train,y_train)
y_pred=model_ada.predict(X_test)
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred,y_test))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("AdaBoost模型拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           0       0.96      0.95      0.95        55
           1       0.97      0.98      0.97        88

    accuracy                           0.97       143
   macro avg       0.96      0.96      0.96       143
weighted avg       0.97      0.97      0.96       143

AdaBoost模型拟合的准确率为:96.503%
```

（四） GBDT

1. 理论介绍

梯度提升树是一种用于回归和分类问题的机器学习技术，其产生的预测模型是弱预测模型的集成，如采用典型的决策树作为弱预测模型，这时则为梯度提升树。像其他提升方法一样，它以分阶段的方式构建模型，但它通过允许对任意可微分损失函数进行优化作为对一般提升方法的推广。

梯度提升的思想源自 Leo Breiman 的一个观察：即可以将提升方法解释为针对适当成本函数的优化算法。它是决策树与 Boosting 方法相结合的应用。GBDT 每棵决策树训练的是前面决策树分类结果中的错误，这也是 Boosting 思想在 GBDT 中的体现。GBDT 的训练过程是线性的，无法像随机森林一样并行训练决策树。第一棵决策树T_1训练的结果与真实

值 T 的残差是第二棵树T_2训练优化的目标，而模型最终的结果是将每一颗决策树的结果进行加权和得到的，即：

$$T = T_1 + T_2 + \cdots T_n$$

对于迭代求优的损失函数的选择，有两种方式，一种方式是直接对残差进行优化，另一种是对梯度下降值进行优化。GBDT 与传统的 Boosting 不同，GBDT 每次迭代的优化目标，Boosting 每次迭代的是重新抽样的样本。

优点：① 可以降低过拟合的风险；② 显著减少训练时间和测试时间开销。

缺点：① 过早的预剪枝可能会造成欠拟合的风险，有些分支的当前划分虽然不能提升泛化性能，但在其基础上进行的后续划分却有可能导致性能显著提高；② 预剪枝基于“贪心”本质禁止这些分支展开，带来了欠拟合的风险。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过建立 GBDT 模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入乳腺癌数据集，数据集的划分及标准化均与 logistic 回归相同
# 导入 GBDT 函数：使用 20 棵决策树
from sklearn.ensemble import GradientBoostingClassifier
model_gbdt=GradientBoostingClassifier(n_estimators=20)
# 模型训练及预测
model_gbdt=model_gbdt.fit(X_train,y_train)
y_pred=model_gbdt.predict(X_test)
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("GBDT 模型拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
#  导入GBDT函数：使用20棵决策树
from  sklearn.ensemble  import  GradientBoostingClassifier
model_gbdt=GradientBoostingClassifier(n_estimators=20)
#  模型训练及预测
model_gbdt=model_gbdt.fit(X_train,y_train)
y_pred=model_gbdt.predict(X_test)
#  模型评价
from  sklearn.metrics  import  classification_report
print(classification_report(y_pred,y_test))
from  sklearn.metrics  import  accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("GBDT模型拟合的准确率为:%.3f%%"%  (Acc  *100))
```

```
              precision    recall  f1-score   support

           0       0.96      0.98      0.97        53
           1       0.99      0.98      0.98        90

    accuracy                           0.98       143
   macro avg       0.98      0.98      0.98       143
weighted avg       0.98      0.98      0.98       143

GBDT模型拟合的准确率为:97.902%
```

（五） XGBoost

1. 理论介绍

XGboost 算法思想就是不断地添加树，不断地进行特征分裂来生长一棵树，每次添加一个树，其实是学习一个新函数，去拟合上次预测的残差。当我们训练完成得到 k 棵树，我们要预测一个样本的分数，其实就是根据这个样本的特征，在每棵树中会落到对应的一个叶子节点，每个叶子节点就对应一个分数，最后只需要将每棵树对应的分数加起来就是该样本的预测值。

XGboost 考虑正则化项，目标函数定义如下：

$$L(\varphi) = \sum_i l(y_i, \hat{y}_i) + \sum_k \Omega(f_k)$$

其中$\Omega(f_k) = \gamma T + \frac{1}{2}\lambda||w||^2$，$\sum_i l(y_i, \hat{y}_i)$代表损失函数，而$\sum_k \Omega(f_k)$则代表正则化项。

$\hat{y}_i$为预测输出，y_i为 label 值，f_k为第 k 树模型，T 为树叶子节点树，w 为叶子权重值，γ为叶子树惩罚正则项，具有剪枝作用，λ为叶子权重惩罚正则项，防止过拟合。XGboost 也支持一阶正则化，容易优化叶子节点权重为 0，不过不常用。

$$\hat{y} = \varphi(x_i) = \sum_{k=1}^{K} f_k(x_i)$$

$$\text{where } F = \{f(x) = w_{q(x)}\}(q: R^m \to T, w \in R^T)$$

$w_{q(x)}$是叶子节点 q 的分数，q(x)是叶子节点的编号，f(x)是其中一颗回归树。也就是

说对于任意一个样本 x，其最后会落在树的某个叶子节点上，其值为$w_{q(x)}$。

新生成的树是要拟合上次预测的残差的，即当生成 t 棵树后，预测分数可以写成：

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

同时，可以将目标函数改写成（其中$f_t(x)$表示当前树上某个叶子节点上的值）：

$$L^{(t)} = \sum_{i=1} l(y_i, (\hat{y}_i^{(t-1)} + f_t(x_i))) + \Omega(f_t)$$

很明显，我们接下来就是要去找到一个f_t能够最小化目标函数。XGboost 的想法是利用其在$f_t = 0$处的泰勒二阶展开近似它。所以，目标函数近似为：

$$L^{(t)} \cong \sum_{i=1} [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

其中g_i为一阶导数，h_i为二阶导数

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

XGboost 使用牛顿法进行梯度的更新。其中g_i和h_i的含义是假设现有 t-1 棵树，这 t-1 棵树组成的模型对第 i 个样本有一个预测值$\hat{y}_i$。这个$\hat{y}_i$与第 i 个样本的真实标签y_i肯定有差距，这个差距可以用$l(y_i, \hat{y}_i)$这个损失函数来衡量。所以此处的g_i和h_i就是对于该损失函数的一阶导和二阶导。

可以注意到，对于每一个样本来说，都有与之对应的g_i和h_i，也就是说，在进行更新迭代的时候，所有的样本都可以并行计算，计算速度会大幅提升，而且 XGboost 可以支持自定义损失函数，只需满足二次可微即可。

由于前 t-1 棵树的预测分数与 y 的残差对目标函数优化不影响，可以直接去掉，所以有：

$$\hat{L}^{(t)} = \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

上式是将每个样本的损失函数值加起来，我们知道，每个样本都会最终落到一个叶子节点中，所以我们可以将所有同一个叶子节点的样本重组起来，过程如下：

$$\begin{aligned}
\hat{L}^{(t)} &= \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \\
&= \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \gamma T + \frac{1}{2}\lambda ||w||^2 \\
&= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \\
&= \frac{1}{2} \sum_{j=1}^{T} (H_j + \lambda)(w_j + \frac{G_j}{H_j + \lambda})^2 + \gamma T - \frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda}
\end{aligned}$$

其中G_j为落入叶子 i 中所有样本一阶梯度统计值总和，H_j为落入叶子 i 中所有样本二阶梯度统计值总和。

因为 XGboost 的基本框架是 boosting，也就是一棵树接着一棵树的形式，所以此处的$\hat{L}^{(t)}$中的 t 代表第 t 棵树，j 表示树的一个叶子节点，i 表示样本，根据树的判别条件，n 个样本点被分到了 T 个叶子节点中。

通过上式的改写，我们可以将目标函数改写成关于叶子节点数 w 的一个一元二次函数，求解最优的 w 和目标函数值就变得很简单。所以对于第 t 棵树而言，每个叶子节点最优的 w 为 $w_j^* = -\frac{G_j}{H_j+\lambda}$

当 w 取上式最优解时，之前$\hat{L}^{(t)}$公式中的第一项等于 0，此时$\hat{L}^*$剩下的项即为最终的目标函数公式：

$$\hat{L}^* = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j+\lambda} + \gamma T$$

在 XGboost 中，我们的分裂准则是直接与损失函数挂钩的准则，这个也是 XGboost 和 GBDT 不一样的地方，XGboost 选择的准则，是计算增益 Gain：

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

其实选择这个作为准则的原因很简单也很直观。由上面的$\hat{L}^*$式知道，对于一个结点，假设我们不分裂的话，那么对于左叶子节点和右叶子节点而言其包含样本点的一阶二阶导的和是合在一起的。那么该节点一阶导的和为$G_L + G_R$，二阶导的和为$H_L + H_R$，将其代入$\hat{L}^*$的式子中可以得到不分裂情况下的损失值

$$\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}$$

假设在这个节点分裂的话，分裂之后左右叶子结点的损失分别为

$$\frac{{G_L}^2}{H_L+H_R+\lambda} \text{和} \frac{{G_R}^2}{H_L+H_R+\lambda}$$

既然要分裂的时候，我们当然是选择分裂成左右子节点后，损失减少的最多的，也就是找到一种分裂有:

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

而γ的作用是控制树的复杂度，进一步来说，可以利用γ来作为阈值，大于这个值就分裂，如果没有就不分裂，它可以起到预剪枝的作用。

优点：① 精度更高：GBDT 只用到一阶泰勒展开，而 XGBoost 对损失函数进行了二阶泰勒展开。XGBoost 引入二阶导一方面是为了增加精度，另一方面也是为了能够自定义损失函数，二阶泰勒展开可以近似大量损失函数；② 灵活性更强：GBDT 以 CART 作为基分类器，XGBoost 不仅支持 CART 还支持线性分类器，使用线性分类器的 XGBoost 相当于带 和正则化项的逻辑斯蒂回归（分类问题）或者线性回归（回归问题）。此外，XGBoost 工具支

持自定义损失函数，只需函数支持一阶和二阶求导；③ 正则化：XGBoost 在目标函数中加入了正则项，用于控制模型的复杂度。正则项里包含了树的叶子节点个数、叶子节点权重的范式。正则项降低了模型的方差，使学习出来的模型更加简单，有助于防止过拟合，这也是 XGBoost 优于传统 GBDT 的一个特性；④ Shrinkage（缩减）：相当于学习速率。XGBoost 在进行完一次迭代后，会将叶子节点的权重乘上该系数，主要是为了削弱每棵树的影响，让后面有更大的学习空间。传统 GBDT 的实现也有学习速率；⑤ 列抽样：XGBoost 借鉴了随机森林的做法，支持列抽样，不仅能降低过拟合，还能减少计算。这也是 XGBoost 异于传统 GBDT 的一个特性；⑥ 缺失值处理：对于特征的值有缺失的样本，XGBoost 采用的稀疏感知算法可以自动学习出它的分裂方向；⑦ XGBoost 工具支持并行：boosting 不是一种串行的结构吗?怎么并行的？注意 XGBoost 的并行不是 tree 粒度的并行，XGBoost 也是一次迭代完才能进行下一次迭代的（第次迭代的代价函数里包含了前面次迭代的预测值）。XGBoost 的并行是在特征粒度上的。我们知道，决策树的学习最耗时的一个步骤就是对特征的值进行排序（因为要确定最佳分割点），XGBoost 在训练之前，预先对数据进行了排序，然后保存为 block 结构，后面的迭代中重复地使用这个结构，大大减小计算量。这个 block 结构也使得并行成为了可能，在进行节点的分裂时，需要计算每个特征的增益，最终选增益最大的那个特征去做分裂，那么各个特征的增益计算就可以开多线程进行；⑧ 可并行的近似算法：树节点在进行分裂时，我们需要计算每个特征的每个分割点对应的增益，即用贪心法枚举所有可能的分割点。当数据无法一次载入内存或者在分布式情况下，贪心算法效率就会变得很低，所以 XGBoost 还提出了一种可并行的近似算法，用于高效地生成候选的分割点。

缺点：① 虽然利用预排序和近似算法可以降低寻找最佳分裂点的计算量，但在节点分裂过程中仍需要遍历数据集；② 预排序过程的空间复杂度过高，不仅需要存储特征值，还需要存储特征对应样本的梯度统计值的索引，相当于消耗了两倍的内存。。

2. 案例实现

（1） 数据来源及解释

本案例选取了 Python 自带的乳腺癌数据集，该数据集有 56 个特征变量，1 个二分类的目标变量，希望通过建立 XGBoost 模型来拟合目标变量与特征变量之间的关系。

（2） 代码实现：

```
# 从 sklearn 的 datasets 数据库中导入乳腺癌数据集，数据集的划分及标准化均与 logistic 回归相同
# 导入 xgboost 模型，并设置参数
import xgboost as xgb
data_train=xgb.DMatrix(X_train,label=y_train)
data_test=xgb.DMatrix(X_test)
params={'booster':'gbtree','objective': 'binary:logistic',
        'gamma':0.15,'learning_rate' : 0.01}
watchlist=[(data_train,'train')]
model_xgb=xgb.train(params,data_train,num_boost_round=5,evals=
watchlist)
```

```
y_pred=model_xgb.predict(data_test)
y_pred=(y_pred>=0.5)*1
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("XGBoost 模型拟合的准确率为:%.3f%%"% (Acc *100))
```

（3） 程序结果：

```
# 导入xgboost模型，并设置参数
import xgboost as xgb
data_train=xgb.DMatrix(X_train,label=y_train)
data_test=xgb.DMatrix(X_test)
params={'booster':'gbtree','objective': 'binary:logistic',
        'gamma':0.15,'learning_rate' : 0.01}
watchlist=[(data_train,'train')]
model_xgb=xgb.train(params,data_train,num_boost_round=5,evals=watchlist)
y_pred=model_xgb.predict(data_test)
y_pred=(y_pred>=0.5)*1
# 模型评价
from sklearn.metrics import classification_report
print(classification_report(y_pred,y_test))
from sklearn.metrics import accuracy_score
Acc=accuracy_score(y_pred,y_test)
print("XGBoost模型拟合的准确率为:%.3f%%"% (Acc *100))
```

```
[0]     train-error:0.021127
[1]     train-error:0.021127
[2]     train-error:0.021127
[3]     train-error:0.021127
[4]     train-error:0.021127
              precision    recall  f1-score   support

           0       0.96      0.98      0.97        53
           1       0.99      0.98      0.98        90

    accuracy                           0.98       143
   macro avg       0.98      0.98      0.98       143
weighted avg       0.98      0.98      0.98       143

XGBoost模型拟合的准确率为:97.902%
```