西瓜书代码实战

🍉本教程为周志华老师的《机器学习》（简称“西瓜书”）配套代码实战教程，GitHub链接：https://github.com/datawhalechina/machine-learning-toy-code

【作者简介】

牧小熊，Datawhale成员&武汉分部负责人，建模算法工程师

知乎主页：https://www.zhihu.com/people/muxiaoxiong

🎃【相关推荐】

西瓜书配套公式推导教程——南瓜书：https://github.com/datawhalechina/pumpkin-book

本教程只完成了大纲部分，一些细节未完善，如在阅读过程中发现问题可与我联系，微信：muxiaoxiong666

第1章绪论

Sklearn（Scikit-Learn）是一个基于Python语言的强大的机器学习工具库，它建立在NumPy、SciPy、Pandas和Matplotlib等科学计算库之上，提供了一整套机器学习算法和数据预处理功能。

Sklearn 因其易用性、统一而优雅的API设计、丰富的算法支持以及活跃的社区，成为许多机器学习开发者和研究人员的首选工具。具体如下：

1. **主要功能**：Sklearn包含了六大任务模块：分类、回归、聚类、降维、模型选择和预处理。这些模块涵盖了机器学习实验的主要步骤，从数据预处理到模型的选择和评估。
2. **算法支持**：Sklearn提供了丰富的机器学习算法，包括线性模型（如线性回归）、决策树、支持向量机（SVM）、随机森林、K-means聚类和主成分分析（PCA）等。这些算法在监督学习（分类和回归）、无监督学习（聚类和降维）中都有广泛应用。
3. **特点优势**：Sklearn的设计强调一致性、可检验、标准类、可组合和默认值等原则。这意味着不同功能的API有统一的调用方式，便于用户快速上手并灵活应用。
4. **数据处理**：在数据预处理方面，Sklearn具备强大的功能，包括数据清洗、标准化、特征编码、特征提取等。这使得从原始数据到可用于模型训练的格式转换变得简单高效。

5. **模型评估**：为了帮助用户评估和优化模型性能，Sklearn提供了多种工具，包括交叉验证、网格搜索以及各种性能指标（如准确率、召回率和F1分数）。
6. **社区更新**：Sklearn拥有活跃的社区支持，并且持续更新。最新的版本要求使用Python 3.6或更高版本，保持与当前技术环境的兼容性。

sklearn安装

```
pip install -U scikit-learn

conda install scikit-learn
```

关联链接：

https://scikit-learn.org/stable/index.html

scikit-learn

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

https://www.scikitlearn.com.cn/

sklearn

scikit-learn 是基于 Python 语言的机器学习工具：简单高效的数据挖掘和数据分析工具；可供大家在各种环境中重复使用；建立在 NumPy ，SciPy 和 matplotlib 上；开源，可商业使用 - BSD许可证。

第2章模型评估与选择

2.2评估方法

2.2.1留出法

> “留出法”直接将数据集D划分为两个互斥的集合，其中一个集合作为训练集，另一个作为测试集

train_test_split方法能够将数据集按照用户的需要指定划分为训练集和测试集

train_data	待划分的样本特征集合
X_train	划分出的训练数据集数据
X_test	划分出的测试数据集数据
y_train	划分出的训练数据集的标签
y_test	划分出的测试数据集的标签

test_size	若在0~1之间，为测试集样本数目与原始样本数目之比；若为整数，则是测试集样本的数目
random_state	随机数种子，不同的随机数种子划分的结果不同
stratify	stratify是为了保持split前类的分布，例如训练集和测试集数量的比例是 A：B= 4：1，等同于split前的比例（80：20）。通常在这种类分布不平衡的情况下会用到stratify。

假设我们现在有数据集D，将数据集按照0.8:0.2的比例划分为数据集与测试集

```
from sklearn.datasets import load_iris

# 加载数据集
iris = load_iris()
X = iris.data
y = iris.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

2.2.2 交叉验证法

> 要在sklearn中实现交叉验证，可以使用**KFold函数**，将数据划分为K个大小互斥的子集
>
> 每次用K-1个字集的并集做训练集，余下的子集做测试集

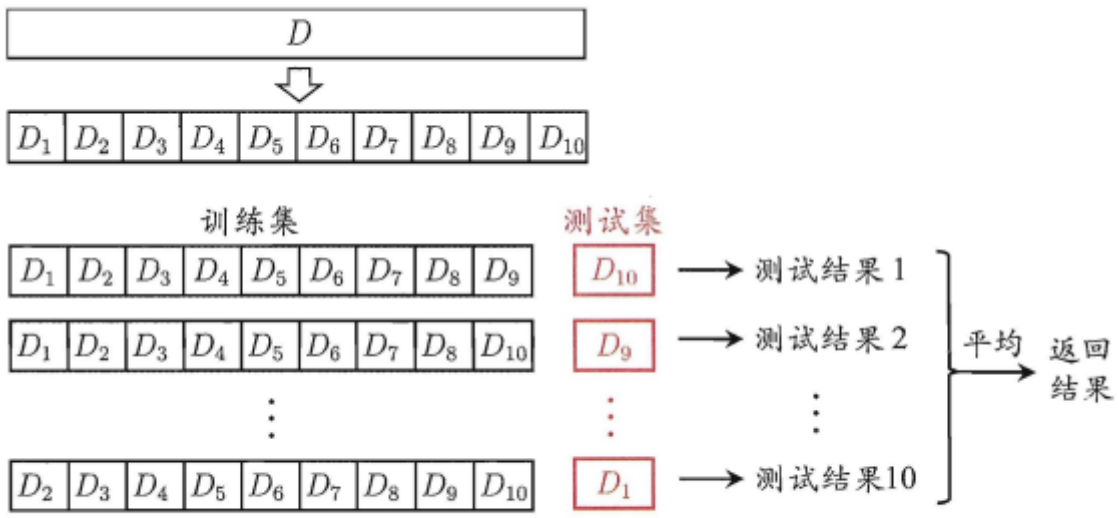


图 2.2　10 折交叉验证示意图

```python
from sklearn.model_selection import KFold
from sklearn.datasets import load_iris

# 加载数据集
iris = load_iris()
X, y = iris.data, iris.target

# 定义模型
# 假设现在有一个模型model
model

# 定义K折交叉验证
kf = KFold(n_splits=5)
result_list=[]
# 进行K折交叉验证
for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # 训练模型
    model.fit(X_train, y_train)

    # 预测测试集
    y_pred = model.predict(X_test)

    result_list.append(y_pred)

```

2.2.3自助法

> 每次随机从D中挑选一个样本，将其拷贝放入D'，重复执行m次后就得到了m个样本的数据集

```python
import numpy as np
from sklearn.utils import resample

#假设有一个数据集X和对应的标签y
X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])
y = np.array([0, 1, 0, 1, 0])

#使用自助法进行抽样
bootstrap_samples = []
for _ in range(10):  # 生成10个自助样本
    X_resampled, y_resampled = resample(X, y)
    bootstrap_samples.append((X_resampled, y_resampled))

```

```
#也可以使用参数n_samples 代替for循环
X_resampled, y_resampled = resample(X, y,n_samples=10, random_state=0)
#bootstrap_samples现在包含了10个自助样本，每个样本都是原始数据集的一个随机子集
```

2.3性能度量

均方误差

回归任务最常用的性能度量是“均方误差”(mean squared error)

$$E(f;D)=\frac{1}{m}\sum_{i=1}^{m}(f(\boldsymbol{x}_i)-y_i)^2 \quad . \tag{2.2}$$

```
#使用sklearn包实现
from sklearn.metrics import mean_squared_error

y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]

mse = mean_squared_error(y_true, y_pred)
print("MSE:", mse)
```

错误率

错误率定义为

$$E(f;D)=\frac{1}{m}\sum_{i=1}^{m}\mathbb{I}(f(\boldsymbol{x}_i)\neq y_i) \quad . \tag{2.4}$$

```
def error_rate(y_pred, y_true):
    """
    计算错误率的函数。

    参数:
    predicted (list): 预测值列表
    actual (list): 实际值列表

    返回:
    float: 错误率
    """
    total = len(y_pred)
```

```
    errors = sum(1 for p, a in zip(y_pred, y_true) if p != a)
    error_rate = errors / total
    return error_rate

# 示例
y_true = [1, 0, 1, 1, 0]
y_pred = [1, 1, 0, 1, 0]
error_rate = error_rate(y_pred , y_true )
print("错误率: ", error_rate)

#错误率可以用1-精度来获得
```

精度

精度则定义为

$$\begin{aligned} \mathrm{acc}(f;D) &= \frac{1}{m}\sum_{i=1}^{m}\mathbb{I}\left(f\left(\boldsymbol{x}_i\right)=y_i\right) \\ &= 1-E(f;D)\ . \end{aligned} \tag{2.5}$$

```
#使用sklearn 计算精度
from sklearn.metrics import accuracy_score

# 假设y_true是实际标签，y_pred是预测标签
y_true = [1, 0, 1, 1, 0]
y_pred = [1, 1, 0, 1, 0]

# 计算准确率
accuracy = accuracy_score(y_true, y_pred)
print("精度: ", accuracy)


```

查准率

$$precision = \frac{TP}{TP + FP}$$

```
##########
from sklearn.metrics import precision_score

```

```
# 假设y_true是真实的标签列表，y_pred是预测的标签列表
y_true = [1, 0, 1, 1, 0, 1]
y_pred = [1, 1, 0, 1, 0, 1]

# 计算精确率
precision = precision_score(y_true, y_pred)

print("精确率: ", precision)

```

查全率

$$Recall = \frac{TP}{TP + FN}$$

```
####################
from sklearn.metrics import recall_score

# 假设y_true是真实的标签列表，y_pred是预测的标签列表
y_true = [1, 0, 1, 1, 0, 1]
y_pred = [1, 1, 0, 1, 0, 1]

# 计算召回率
recall = recall_score(y_true, y_pred)

print("召回率: ", recall)

```

F1值

$$F1 = \frac{2 \times P \times R}{P + R} = \frac{2 \times TP}{样例总数 + TP - TN}, \tag{2.10}$$

```
#使用sklearn 计算F1值
from sklearn.metrics import f1_score
y_true = [1, 0, 1, 1, 0]
y_pred = [1, 1, 0, 1, 0]
f1 = f1_score(y_true, y_pred)
print("F1值: ", f1)
```

AUC值

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^+ 个正例和 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

$$\ell_{rank} = \frac{1}{m^+ m^-} \sum_{\boldsymbol{x}^+ \in D^+} \sum_{\boldsymbol{x}^- \in D^-} \left(\mathbb{I}\left(f(\boldsymbol{x}^+) < f(\boldsymbol{x}^-)\right) + \frac{1}{2}\mathbb{I}\left(f(\boldsymbol{x}^+) = f(\boldsymbol{x}^-)\right) \right), \tag{2.21}$$

即考虑每一对正、反例, 若正例的预测值小于反例, 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 曲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

$$\mathrm{AUC} = 1 - \ell_{rank}. \tag{2.22}$$

```python
# 假设y_true是真实标签，y_pred是预测概率
from sklearn.metrics import roc_auc_score
y_true = [1, 0, 1, 1, 0]
y_pred = [0.9, 0.8, 0.7, 0.6, 0.5]

# 计算AUC值
auc = roc_auc_score(y_true, y_pred)
print("AUC值为：", auc)

```

2.4比较检验

2.4.1假设检验

假设检验中的“假设”是对学习器泛化错误率分布的某种判断或猜想, 例如“$\epsilon = \epsilon_0$”. 现实任务中我们并不知道学习器的泛化错误率, 只能获知其测试错误率 $\hat{\epsilon}$. 泛化错误率与测试错误率未必相同, 但直观上, 二者接近的可能性应比较大, 相差很远的可能性比较小. 因此, 可根据测试错误率估推出泛化错误率的分布.

$$\tau_t = \frac{\sqrt{k}(\mu - \epsilon_0)}{\sigma}$$

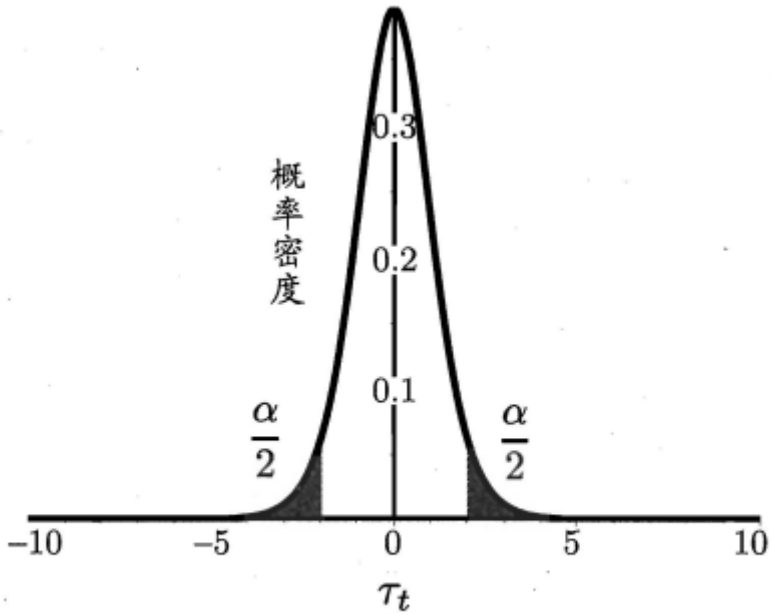


图 2.7 t 分布示意图$(k = 10)$

```python
import numpy as np
from scipy import stats

def t_test(k_fold_errors, a):
    """
    使用T检验推断模型的泛化错误率是否为K样本错误率的平均值。

    参数:
    k_fold_errors (list): K折交叉验证得到的K个预测的错误率列表。
    a (float): 假设模型预测的错误泛化率。

    返回:
    tuple: T统计量和p值。
    """
    # 计算K样本错误率的平均值
    mean_k_fold_error = np.mean(k_fold_errors)

    # 计算标准误差 因为se的平方是方差 所以我们计算标准差
    se = np.std(k_fold_errors, ddof=1) / np.sqrt(len(k_fold_errors))

    # 计算T统计量
    t_statistic = (mean_k_fold_error-a) / se

    # 计算自由度
    df = len(k_fold_errors) - 1

    # 计算p值 也就是累计分布函数
    p_value = 2 * (1 - stats.t.cdf(np.abs(t_statistic), df))

```

```
    return t_statistic, p_value

# 示例
k_fold_errors = [0.1, 0.15, 0.12, 0.13, 0.14]  # K折交叉验证得到的K个预测的错误率列表
a = 0.16  # 假设模型在的泛化错误率为0.16

t_statistic, p_value = t_test(k_fold_errors, a)
print("T统计量: ", t_statistic)
print("p值: ", p_value)

#也可以用scipy库中的单样本T检验
from scipy import stats
t_statistic, p_value = stats.ttest_1samp(k_fold_errors, a)
print("T统计量: ", t_statistic)
print("P值: ", p_value)
```

2.4.2交叉验证T检验

对两个学习器 A 和 B，若我们使用 k 折交叉验证法得到的测试错误率分别为 $\epsilon_1^A, \epsilon_2^A, \ldots, \epsilon_k^A$ 和 $\epsilon_1^B, \epsilon_2^B, \ldots, \epsilon_k^B$，其中 ϵ_i^A 和 ϵ_i^B 是在相同的第 i 折训练/测试集上得到的结果，则可用 k 折交叉验证“成对 t 检验”(paired t-tests)来进行比较检验. 这里的基本思想是若两个学习器的性能相同，则它们使用相同的训练/测试集得到的测试错误率应相同，即 $\epsilon_i^A = \epsilon_i^B$.

```
import numpy as np
from scipy import stats

def t_test(model_a_errors, model_b_errors):
    """
    使用T检验比较两个模型的预测错误率是否有显著差异。

    参数:
    model_a_errors (list): 模型A的K折交叉验证预测错误率列表
    model_b_errors (list): 模型B的K折交叉验证预测错误率列表

    返回:
    t_statistic (float): T统计量
    p_value (float): p值
    """
    # 计算均值和标准差
    mean_a = np.mean(model_a_errors)
    mean_b = np.mean(model_b_errors)
```

```
    std_a = np.std(model_a_errors, ddof=1)
    std_b = np.std(model_b_errors, ddof=1)
    n_a = len(model_a_errors)
    n_b = len(model_b_errors)

    # 计算T统计量
    t_statistic = (mean_a - mean_b) / np.sqrt((std_a**2 / n_a) + (std_b**2 / n_b))

    # 计算自由度
    df = n_a + n_b - 2

    # 计算p值
    p_value = 2 * (1 - stats.t.cdf(np.abs(t_statistic), df))

    return t_statistic, p_value

# 示例数据
model_a_errors = [0.1, 0.15, 0.12, 0.14, 0.16]
model_b_errors = [0.13, 0.17, 0.14, 0.15, 0.18]

# 进行T检验
t_statistic, p_value = t_test(model_a_errors, model_b_errors)
print("T统计量：", t_statistic)
print("p值：", p_value)

#也可以用scipy库中的双样本T检验
from scipy.stats import ttest_ind
t_statistic, p_value = ttest_ind(model_a_errors, model_b_errors)

print("T统计量：", t_statistic)
print("P值：", p_value)
```

2.4.3McNemar检验（配对卡方检验）

表 2.4　两学习器分类差别列联表

算法 B	算法 A	
	正确	错误
正确	e_{00}	e_{01}
错误	e_{10}	e_{11}

```python
import numpy as np
from scipy.stats import chi2

import numpy as np
from scipy.stats import chi2_contingency

def mcnemar_test(y_true, y_pred1, y_pred2):
    """
    使用McNemar检验评估两个模型的预测性能是否有差异。

    参数:
    y_pred1: 模型A预测结果
    y_pred2: 模型B预测结果
    y_true: 真实的预测标签

    返回:
    chi2: 统计量
    p_value: McNemar检验的p值
    """
    # Create a contingency table
    contingency_table = np.zeros((2, 2))
    for i in range(len(y_true)):
        if y_pred1[i] == y_true[i] :
            if y_pred2[i] == y_true[i]:
                contingency_table[0, 0] += 1
            else:
                contingency_table[0, 1] += 1
        else:
            if y_pred2[i] == y_true[i]:
                contingency_table[1, 0] += 1
            else:
                contingency_table[1, 1] += 1

    # 打印列联表
    print(contingency_table)
    chi2, p_value, _, _ = chi2_contingency(contingency_table)
    return chi2, p_value

# Example usage:
y_true  = [1, 0, 1, 0, 0, 0, 1, 1, 1, 0]
y_pred1 = [1, 0, 1, 0, 0, 0, 1, 0, 1, 1]
y_pred2 = [1, 1, 1, 0, 0, 0, 1, 1, 1, 1]

chi2,p_value = mcnemar_test(y_true, y_pred1, y_pred2)
print("统计量: ", chi2_statistic)
print("p值: ", p_value)

```

2.4.4 Friedman检验与Nemenyi后续检验（弗里德曼检验）

> Nemenyi后续检验在实际的应用中不常用，感兴趣的同学可以自行实现

```python
from scipy.stats import friedmanchisquare

# 假设模型A、B、C在相同数据集上的性能评分
scores_A = [1, 2, 3, 4, 5]
scores_B = [2, 3, 4, 5, 6]
scores_C = [4, 5, 6, 7, 8]

# 执行Friedman检验
statistic, p_value = friedmanchisquare(scores_A, scores_B, scores_C)

print(f"Friedman检验统计量: {statistic}, p值: {p_value}")
```

2.5偏差与方差

> 本节的思想主要是需要理解 机器学习中 偏差、误差、方差的概念，代码实际只提供了实现方法，如需要深入学习请访问参考链接
>
> 关联链接：https://www.zhihu.com/question/27068705/answer/3076463397

$$bias^2(\boldsymbol{x}) = \left(\bar{f}(\boldsymbol{x}) - y\right)^2 . \tag{2.40}$$

```python
import math
#期望输出与真实标签的差别
y_true = [1, 0, 1, 1, 0]
y_pred = [0.9, 0.8, 0.7, 0.6, 0.5]

bias_2=sum([(x - y) ** 2 for x, y in zip(y_true , y_pred )])
bias=math.sqrt(bias_2)
```

$$var(\boldsymbol{x}) = \mathbb{E}_D\left[\left(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x})\right)^2\right] , \tag{2.38}$$

```python
import numpy as np
y_true = [1, 0, 1, 1, 0]
y_pred = [0.9, 0.8, 0.7, 0.6, 0.5]
```

```
4
5   # 计算方差
6   mean_true = np.mean(y_true )
7   squared_errors = sum((y_pred - mean_true) ** 2)
8   var= np.mean(squared_errors)
9   print("方差: ", var)
```

第3章线性模型

本节主要实现线性模型中的线性回归与逻辑回归

3.2线性回归

线性回归是比较简单的机器学习算法，通常作为机器学习入门第一课。

线性回归的一般形式为

$$f(\boldsymbol{x}) = w_1 x_1 + w_2 x_2 + \ldots + w_d x_d + b \,, \tag{3.1}$$

如何确定 w 和 b 呢? 显然, 关键在于如何衡量 $f(x)$ 与 y 之间的差别. 2.3 节介绍过, 均方误差 (2.2)是回归任务中最常用的性能度量, 因此我们可试图让均方误差最小化, 即

$$\begin{aligned}(w^*, b^*) &= \arg\min_{(w,b)} \sum_{i=1}^{m} \left(f\left(x_i\right) - y_i\right)^2 \\ &= \arg\min_{(w,b)} \sum_{i=1}^{m} (y_i - w x_i - b)^2 \,. \end{aligned} \tag{3.4}$$

通常情况下系数w和b 无法之间求解，我们往往会用梯度下降法进行解决，具体而言，就是给w 和 b 一个初始值，计算均方误差的梯度，从而继续更新参数，于是对上面的3.4的公式求偏导可以得到

$$\frac{\partial loss}{\partial w} = \sum_{i=1}^{m} 2\left(w x_i + b - y_i\right) x_i$$

$$\frac{\partial loss}{\partial b} = \sum_{i=1}^{m} 2\left(w x_i + b - y_i\right)$$

构造数据集

```
1   import numpy as np
2
3   #生成一个样本量为100 特征量为10的数据集
4   sample=100
```

```
feature=10
X=np.random.rand(sample,10)
#构造目标值 y=3+2*x1+5*x2-3*x3
y=3+2*X[:,0]+5*X[:,1]-3*X[:,2]+np.random.randn(sample) # 增加一点噪音
#np.random.randn 生成满足正态分布的随机数
```

构建模型

```
class LR():
    def __init__(self):
        self.w=None
        self.b=None

    def get_loss(self,y,y_pre):
        """
        定义损失函数f(x)--> 均方误差
        y: 真实值
        y_pre: 预测值
        """
        loss=np.mean((y-y_pre)**2)
        return loss

    def fit(self,x,y,learning_rate=0.01,n_iterations=500):
        """
        x 训练数据
        y 目标值
        learning_rate 学习率
        n_iterations 迭代次数
        """
        sample,feature=x.shape

        if self.w==None:
            #初始化数据
            self.w=np.random.randn(feature)
            self.b=0

        #开始训练
        for i in range(n_iterations):
            y_pre=np.dot(x,self.w)+self.b
            #计算预测值与真实值之差
            # !!注意这里是预测值减真实值
            diff=y_pre-y
            #计算损失函数的梯度
            dw=2/sample*np.dot(x.T,diff)
            db=2/sample*np.sum(diff,axis=0)
```

```
38
39              #更新参数
40              self.w=self.w-learning_rate*dw
41              self.b=self.b-learning_rate*db
42
43              #计算loss
44              loss=self.get_loss(y,y_pre)
45              print("epoch:{}  loss:{}".format(i,loss))
46
47      def predict(self,x):
48          y_pre=np.dot(x,self.w)+self.b
49          return y_pre
```

预测结果

```
1   #生成一个样本量为100 特征量为10的数据集
2   sample=1
3   feature=10
4   test_x=np.random.rand(sample,10)
5   #构造目标值 y=3+2*x1+5*x2-3*x3
6   test_y=3+2*test_x[:,0]+5*test_x[:,1]-3*test_x[:,2]+np.random.randn(sample) #
    增加一点噪音
7   #np.random.randn 生成满足正态分布的随机数
8
9   model=LR()
10  model.fit(X,y)
11  y_pre=model.predict(test_x)
12
```

使用sklearn包实现

```
1   #构造数据集
2   import numpy as np
3
4   #生成一个样本量为100 特征量为10的数据集
5   sample=100
6   feature=10
7   X=np.random.rand(sample,feature)
8   #构造目标值 y=3+2*x1+5*x2-3*x3
9   y=3+2*X[:,0]+5*X[:,1]-3*X[:,2]+np.random.randn(sample) # 增加一点噪音
10  #np.random.randn 生成满足正态分布的随机数
11
12  from sklearn.linear_model import LinearRegression # 线性回归模型
```

```
from sklearn.metrics import mean_squared_error # 评价指标

model=LinearRegression()
model.fit(X,y)
y_pre=model.predict(test_x)

#显示权重系数
weights = model.coef_
intercept = model.intercept_

print("权重：", weights)
print("偏移：", intercept)
```

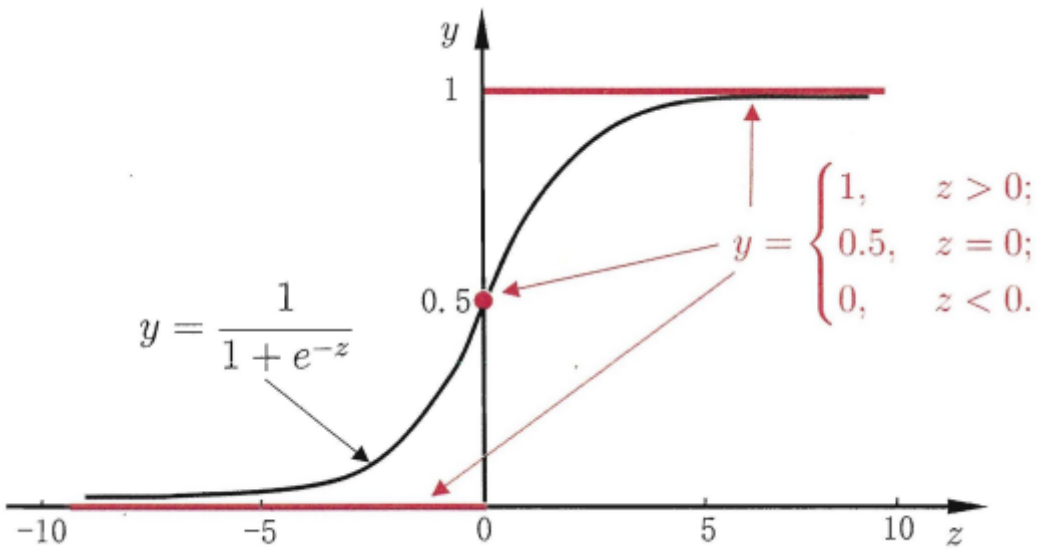
参数讲解

fit() 函数中带的参数

- fit_intercept: 是否计算截距。默认为 True，表示计算截距。
- normalize: 是否在回归前对数据进行归一化。默认为 False。
- copy_X: 是否复制X。默认为 True。
- n_jobs: 用于计算的作业数。默认为 None，表示使用1个作业。如果设置为 -1，则使用所有CPU。

3.3对数几率回归

Sigmoid 函数



实现sigmod函数

```
import numpy as np

```

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# 示例
x = np.array([5])
print(sigmoid(x))

```

构造数据集

```
import numpy as np
#构造数据集
#生成一个样本量为100 特征量为10的数据集
sample=100
feature=10
X=np.random.rand(sample,10)
#构造目标值 y=3+2*x1+5*x2-3*x3
y=3+2*X[:,0]+5*X[:,1]-3*X[:,2]+np.random.randn(sample) # 增加一点噪音
# 我们需要构建一个分类的问题
mean_=np.mean(y)
# 一半的数据低于平均值 一半的数据高于平均值
y=np.where(y>=mean_,1,0)
#样本就变成了分类事实，数据是否比平均值大
"""
array([1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1,
       1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
       0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1,
       1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0,
       1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1])
"""

#构造数据集

```

使用sklearn包实现

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import numpy as np
#构造数据集
#生成一个样本量为100 特征量为10的数据集
sample=100
```

```
feature=10
X=np.random.rand(sample,10)
#构造目标值 y=3+2*x1+5*x2-3*x3
y=3+2*X[:,0]+5*X[:,1]-3*X[:,2]+np.random.randn(sample) # 增加一点噪音
# 我们需要构建一个分类的问题
mean_=np.mean(y)
# 一半的数据低于平均值 一半的数据高于平均值
y=np.where(y>=mean_,1,0)
#样本就变成了分类事实，数据是否比平均值大

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 创建逻辑回归模型
log_reg = LogisticRegression()

# 训练模型
log_reg.fit(X_train, y_train)

# 预测
predictions = log_reg.predict(X_test)

# 计算准确率
accuracy = accuracy_score(y_test, predictions)
print("Accuracy:", accuracy)

```

参数讲解

fit() 函数中带的参数

1. **基本参数**

   penalty：该参数用于指定正则化项的类型，可选值为“l1”、“l2”和“elasticnet”，默认为“l2”。L1正则化倾向于生成稀疏权重矩阵，有助于特征选择；L2正则化则倾向于避免模型权重过大，从而防止过拟合。

   C：此参数是正则化强度的倒数，即惩罚系数λ的倒数。较小的C值表示正则化强度更大，会使模型更简单，提高泛化能力。

   fit_intercept：该布尔参数决定是否在模型中计算截距项，即是否需要偏置，默认为True。

2. **求解器相关参数**

   solver：该参数用于选择优化算法，不同的算法适用于不同的情况。例如，“liblinear”适合小数据集或L1正则化；“lbfgs”、“newton-cg”和“sag”适合大数据集且仅支持L2正则化；“saga”则既适用于L1也适用于L2正则化。

max_iter：这是最大迭代次数，用于指定优化算法的收敛阈值，即达到多少次迭代后停止训练。

3. **容忍度及随机状态**

   tol：该参数用于设置求解器的容忍度，即多小的变化会被认为是收敛，不再继续迭代。

   random_state：该参数用于设置随机种子，确保每次运行的结果可复现。

4. **多元分类策略**

   multi_class：当处理多分类问题时，该参数决定了采用何种策略，"ovr"（one-vs-rest）或者"multinomial"（many-vs-many），前者在每个二元分类问题上独立训练一个分类器，后者则同时考虑所有类别。

5. **对偶及权重参数**

   dual：该参数只在使用“liblinear”求解器并选择L2正则化时有用，当样本数大于特征数时建议设置为False。

   class_weight：用于设定各类别的权重，可以是字典形式，也可以是“balanced”让类库自动计算权重，特别适用于处理类别不平衡的数据。

6. **其他参数：**

   verbose：对于liblinear和lbfgs求解器，将verbose设置为任何正数以表示详细程度。用于开启/关闭迭代中间输出的日志。

   n_jobs：如果multi_class = ‘ovr’ ，则在对类进行并行化时使用的CPU数量。 无论是否指定’ multi_class’ ，当 `solver` 设置为’ liblinear’ 时，都会忽略此参数。 如果给定值-1，则使用所有CPU。

3.4线性判别分析

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# 加载数据集
iris = load_iris()
X = iris.data
y = iris.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 创建LDA模型
lda = LinearDiscriminantAnalysis()
```

```
16
17  # 训练模型
18  lda.fit(X_train, y_train)
19
20  # 预测测试集
21  y_pred = lda.predict(X_test)
22
23  # 计算准确率
24  accuracy = accuracy_score(y_test, y_pred)
25  print("LDA分类器的准确率：", accuracy)
26
```

3.5多分类学习

```
1   from sklearn.datasets import load_iris
2   from sklearn.model_selection import train_test_split
3   from sklearn.linear_model import LogisticRegression
4   from sklearn.metrics import accuracy_score
5
6   # 加载鸢尾花数据集
7   iris = load_iris()
8   X = iris.data
9   y = iris.target
10
11  # 将数据集划分为训练集和测试集
12  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
13
14  # 创建逻辑回归模型
15  logreg = LogisticRegression(multi_class='multinomial', solver='lbfgs',
    max_iter=1000)
16
17  # 训练模型
18  logreg.fit(X_train, y_train)
19
20  # 预测测试集
21  y_pred = logreg.predict(X_test)
22
23  # 计算准确率
24  accuracy = accuracy_score(y_test, y_pred)
25  print("Accuracy: {:.2f}".format(accuracy))
26
```

3.6类别不平衡问题

```
make_classification(
                    n_samples,
                    n_features,
                    n_informative,
                    n_redundant,
                    n_repeated,
                    n_classes,
                    n_clusters_per_class,
                    weights,
                    flip_y,
                    class_sep,
                    hypercube,
                    shift,
                    scale,
                    shuffle,
                    random_state
)
```

1. n_samples: 一个整数表示将要生成的数据总量。默认值为100。
2. n_features: 一个整数，表示特征的数量。在make_classification中默认值为20，在make_regression中默认值为100。
3. n_informative: 一个整数，表示特征中比较重要的特征的数量（即可以提供更多信息量的特征数量）。在make_classification中默认值为2，在make_regression中默认值为10。
4. n_redundant: 一个整数，表示特征中冗余特征的数量（即不能提供更多信息量的特征数量）。在make_classification中默认值为2。
5. n_repeated: 一个整数，表示特征中重复特征的数量。可以模拟实际问题中因为数据提取不好造成的数据重复问题。在make_classification中默认值为0。
6. n_classes: 一个整数，表示分类问题中的目标类型的数量。在make_classification中默认值为2。
7. n_clusters_per_class：一个整数，表示分类问题中每类拥有的数据簇的数量。在make_classification中默认值为2。
8. weights: 浮点数的列表，表示每一类数据占总数据的比重。注意当列表中所有浮点数的值之和大于1时，可能会产生意想不到的结果。在make_classification中默认值为None。
9. flip_y：一个浮点数，表示噪音值。这个数越大就会使分类更困难。在make_classification中默认值为0.01。

10. class_sep：一个浮点数，表示类与类之间的间距。这个值越大就会使分类更容易。在make_classification中默认值为0.01。
11. hypercube: 一个布尔值，当为真时，表示数据簇是从超立方体（想象一下问题空间，二维问题就是正方形，三维就是立方体，更高维就是超立方体了）的顶点开始产生的。否则就表示数据簇是从随机的多平面体的顶点上生成的。说得更直白一些的话，当这个值为真时，生成的数据会更均匀一些。在make_classification中默认值为True。
12. shift：一个浮点数或一个长度为n_features的浮点数组或者None。表示将特征值通过某个值进行平移，不然生成的特征值就分布在0点的周围了。在make_classification中默认值为0.0。
13. scale：一个浮点数或一个长度为n_features的浮点数组或者None。表示将特征值与某个值相乘后的结果赋值给这个特征值，注意是先发生shift再scale，学过线性代数的同学肯定一下子就可以发现这就是对特征值做一个一维线性变换。在make_classification中默认值为1.0。
14. shuffle: 一个布尔值，表示是否要打乱生成的数据。在make_regression中默认为True。
15. random_state：None或者一个整数，当输入为一个整数时，表示这次生成数据过程的随机因子。换句话说，如果两次生成数据时，如果random_state是同一个整数，且其他参数都相同，则生成的数据是一样的。

```
from sklearn.datasets import make_classification

X, y = make_classification(n_samples=100, n_features=20, n_informative=2,
n_redundant=10, n_classes=3, random_state=42,n_clusters_per_class=1)
print("特征数据: \n", X)
print("标签数据: \n", y)
```

欠采样

```
from sklearn.datasets import make_classification
from imblearn.under_sampling import RandomUnderSampler
from collections import Counter

# 创建一个不平衡的数据集
X, y = make_classification(n_classes=2, class_sep=2, weights=[0.1, 0.9],
n_informative=3, n_redundant=1, flip_y=0, n_features=20,
n_clusters_per_class=1, n_samples=1000, random_state=10)

# 打印原始数据集的类别分布
print("原始数据集类别分布: ", Counter(y))

# 实例化RandomUnderSampler对象
rus = RandomUnderSampler(random_state=42)
```

```
13
14  # 对数据集进行欠采样
15  X_resampled, y_resampled = rus.fit_resample(X, y)
16
17  # 打印欠采样后的数据集类别分布
18  print("欠采样后数据集类别分布: ", Counter(y_resampled))
19
```

过采样

```
1   from sklearn.datasets import make_classification
2   from imblearn.over_sampling import RandomOverSampler
3   from collections import Counter
4
5   # 创建一个不平衡的数据集
6   X, y = make_classification(n_classes=2, class_sep=2, weights=[0.1, 0.9],
    n_informative=3, n_redundant=1, flip_y=0, n_features=20,
    n_clusters_per_class=1, n_samples=1000, random_state=10)
7
8   # 打印原始数据集的类别分布
9   print("原始数据集类别分布: ", Counter(y))
10
11  # 实例化RandomOverSampler对象
12  ros = RandomOverSampler(random_state=42)
13
14  # 对数据集进行过采样
15  X_resampled, y_resampled = ros.fit_resample(X, y)
16
17  # 打印过采样后的数据集类别分布
18  print("过采样后数据集类别分布: ", Counter(y_resampled))
19
```

阈值移动

可以通过设置 `class_weight` 参数来调整类别权重，以平衡不同类别的损失贡献

class_weight：用于设定各类别的权重，可以是字典形式，也可以是“balanced”让类库自动计算权重，特别适用于处理类别不平衡的数据。或者可以用字典的形式传参，例如class_weight = {0:1,1:3}

```
1   #分类的时候，当不同类别的样本量差异很大时，很容易影响分类结果，因此要么每个类别的数据量大
    致相同，要么就要进行校正。
2   #sklearn的做法可以是加权，加权就要涉及到class_weight和sample_weight
3   #当不设置class_weight参数时，默认值是所有类别的权值为1
```

```
4
5   #那么'balanced'的计算方法是什么呢?
6   import numpy as np
7
8   y = [0,0,0,0,0,0,0,0,1,1,1,1,1,1,2,2]  #标签值，一共16个样本
9
10  a = np.bincount(y)  # array([8, 6, 2], dtype=int64) 计算每个类别的样本数量
11  aa = 1/a  #倒数 array([0.125     , 0.16666667, 0.5       ])
12  print(aa)
13
14  from sklearn.utils.class_weight import compute_class_weight
15  y = [0,0,0,0,0,0,0,0,1,1,1,1,1,1,2,2]
16
17  # 计算类别权重
18  class_weights = compute_class_weight('balanced', classes=[0, 1,2], y=y)
19  print("类别权重: ", class_weights) # [0.66666667 0.88888889 2.66666667]
20
21
22  #weight_ = n_samples / (n_classes * np.bincount(y))
23
24  print(16/(3*8))  #输出 0.6666666666666666
25  print(16/(3*6))  #输出 0.8888888888888888
26  print(16/(3*2))  #输出 2.6666666666666665
```

```
1   from sklearn.datasets import make_classification
2   from sklearn.model_selection import train_test_split
3   from sklearn.linear_model import LogisticRegression
4   from sklearn.metrics import accuracy_score
5
6   # 创建一个不平衡的数据集
7   X, y = make_classification(n_classes=2, class_sep=2, weights=[0.1, 0.9],
    n_informative=3, n_redundant=1, flip_y=0, n_features=20,
    n_clusters_per_class=1, n_samples=1000, random_state=10)
8
9   # 划分训练集和测试集
10  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
11
12  # 创建逻辑回归模型，并设置class_weight为'balanced'
13  #class_weight 可以设置为 {0:99:, 1:1}
14  logreg = LogisticRegression(class_weight='balanced', solver='liblinear',
    max_iter=1000)
15
16  # 训练模型
17  logreg.fit(X_train, y_train)
```

```
18
19  # 预测测试集
20  y_pred = logreg.predict(X_test)
21
22  # 计算准确率
23  accuracy = accuracy_score(y_test, y_pred)
24  print("逻辑回归模型的准确率: ", accuracy)
25
```

第4章决策树

sklearn.datasets 中主要包含了一些公共数据集

- load_iris(): 加载鸢尾花数据集，这是一个常用的多类分类数据集。
- load_digits(): 加载手写数字数据集，每个实例都是一张8x8的数字图像及其对应的数字类别。
- load_boston(): 加载波士顿房价数据集，这是一个回归问题的数据集。
- load_breast_cancer(): 加载乳腺癌数据集，这是一个二分类问题的数据集。
- load_diabetes(): 加载糖尿病数据集，这个数据集可以用于回归分析。

sklearn中决策树的用法

- from sklearn.tree import DecisionTreeClassifier 分类树
- from sklearn.tree import DecisionTreeRegressor 回归树
- from sklearn.tree import plot_tree 画决策树

```
1   from sklearn.datasets import load_iris
2   from sklearn.model_selection import train_test_split
3   from sklearn.tree import DecisionTreeClassifier
4   from sklearn.metrics import accuracy_score
5   from sklearn.tree import plot_tree
6   from matplotlib import rcParams
7   # 加载数据集
8   iris = load_iris()
9   X = iris.data
10  y = iris.target
11
12  # 划分训练集和测试集
13  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
14
15  # 创建决策树模型
```

```
clf = DecisionTreeClassifier()

# 训练模型
clf.fit(X_train, y_train)

# 预测
predictions = clf.predict(X_test)

# 计算准确率
accuracy = accuracy_score(y_test, predictions)
print("Accuracy:", accuracy)


####画决策树曲线
rcParams['figure.figsize'] = 16, 10  # 设置为16宽，10高的图像大小
plot_tree(clf)
```

决策树模型的参数：

1. **基本参数**
   - **Criterion**：该参数用于指定在分裂节点时使用的标准。可选的值有"gini"和"entropy"，其中"gini"表示使用基尼不纯度，而"entropy"则表示使用信息增益。默认值为"gini"。
   - **Splitter**：该参数用于设置在每个节点上选择分裂的策略。可选的值有"best"和"random"，其中"best"表示从所有特征中选择最优的进行分裂，而"random"则表示随机选择一个特征子集进行分裂。默认值为"best"。
   - **Max depth**：该参数用于设置决策树的最大深度。如果设置为None，那么决策树会扩展到所有叶子都变得纯净或直到所有叶子节点包含的样本数小于min_samples_split为止。这可以防止模型过拟合。
   - **Min samples split**：该参数用于设置一个节点在分裂时所需的最小样本数量。如果这个值较大，可以减少模型的过拟合风险。
   - **Min samples leaf**：该参数用于设置一个叶子节点所需的最小样本数量。设置一个合适的值可以避免叶子节点过于稀疏，从而降低过拟合的风险。
   - **Min weight fraction leaf**：该参数用于设置叶节点中样本权重总和的最小比例。如果未提供sample_weight，则所有权重相等。
   - **Max features**：该参数用于设置寻找最佳分裂时要考虑的特征数量。可以是int、float或"auto"、"sqrt"、"log2"之一。如果是int，考虑max_features个特征；如果是float，考虑max_features 乘以特征总数；如果是"auto"，则考虑特征总数的平方根。
2. **高级参数**
   - **Random state**：该参数是随机种子，用于控制分裂特征的随机性。这可以确保在同一份数据上多次运行算法时能得到相同的结果。

- **Max leaf nodes**：该参数用于设置最大叶子节点的数量。通过限制叶子节点数量，可以防止决策树过度生长。
- **Min impurity decrease**：该参数用于设置如果分裂后杂质减少量大于该值，则进行分裂。这可以用来控制决策树的生长速度。
- **Class weight**：该参数用于设定各类别的权重，可以是字典形式{class_label: weight}，也可以是"balanced"，后者会根据样本数量和类别自动调整权重

参数优化

GridSearchCV

1. estimator 选择使用的分类器，也就是需要优化参数的模型
2. param_grid 需要最优化的参数的取值，值为字典或者列表
3. scoring=None
4. 模型评价标准，默认None,这时需要使用score函数；或者如scoring='roc_auc'，
5. n_jobs=1 n_jobs: 并行数，int：个数,-1：跟CPU核数一致, 1:默认值
6. cv 交叉验证参数，默认None，使用三折交叉验证。指定fold数量，默认为3，也可以是yield产生训练/测试数据的生成器。
7. verbose 日志冗长度，int：冗长度，0：不输出训练过程，1：偶尔输出，>1：对每个子模型都输出。

进行预测的常用方法和属性

grid.fit()：运行网格搜索

grid_scores_：给出不同参数情况下的评价结果

best_params_：描述了已取得最佳结果的参数的组合

best_score_：成员提供优化过程期间观察到的最好的评分

```
# 导入所需库和模块
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# 加载数据集
iris = load_iris()
X = iris.data
y = iris.target
```

```
11
12  # 划分训练集和测试集
13  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    random_state=42)
14
15  # 创建决策树模型
16  dt = DecisionTreeClassifier()
17
18  # 设置参数网格
19  param_grid = {
20      'criterion': ['gini', 'entropy'],
21      'max_depth': [None, 10, 20, 30],
22      'min_samples_split': [2, 5, 10],
23      'min_samples_leaf': [1, 2, 4]
24  }
25
26  # 使用网格搜索进行参数优化
27  grid_search = GridSearchCV(dt, param_grid, cv=5, scoring='accuracy')
28  grid_search.fit(X_train, y_train)
29
30  # 输出最佳参数
31  print("Best parameters found: ", grid_search.best_params_)
32
33  # 使用最佳参数训练模型
34  best_dt = grid_search.best_estimator_
35  best_dt.fit(X_train, y_train)
36
37  # 预测测试集并计算准确率
38  y_pred = best_dt.predict(X_test)
39  accuracy = accuracy_score(y_test, y_pred)
40  print("Accuracy on test set: {:.2f}".format(accuracy))
41
```

第5章神经网络

from sklearn.neural_network import MLPClassifier #分类问题

from sklearn.neural_network import MLPRegressor #回归问题

```
1  from sklearn.datasets import load_iris
2  from sklearn.model_selection import train_test_split
3  from sklearn.neural_network import MLPClassifier
4  from sklearn.metrics import accuracy_score
5
6  # 加载数据集
```

```
iris = load_iris()
X = iris.data
y = iris.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 创建多层感知机模型
mlp = MLPClassifier(hidden_layer_sizes=(100,), max_iter=300, random_state=42)

# 训练模型
mlp.fit(X_train, y_train)

# 预测
predictions = mlp.predict(X_test)

# 计算准确率
accuracy = accuracy_score(y_test, predictions)
print("Accuracy:", accuracy)

 (100)
 (100,100)
```

核心参数说明：

- **hidden_layer_sizes**：一个元组，表示隐藏层中神经元的数量。例如，`(100,)` 表示一个隐藏层，包含 100 个神经元
- **activation**：隐藏层的激活函数。选项有 `'identity'`、`'logistic'`、`'tanh'`、`'relu'`。
- **solver**：用于权重优化的优化器。`'lbfgs'` 是一种准牛顿法，`'sgd'` 是随机梯度下降，`'adam'` 是一种基于随机梯度的优化器。
- **alpha**：L2 惩罚（正则化项）参数。
- **batch_size**：随机优化器的小批量大小。
- **learning_rate**：权重更新的学习率调度。选项有 `'constant'`、`'invscaling'`、`'adaptive'`。
- **max_iter**：最大迭代次数。求解器迭代直到收敛或达到此迭代次数。
- **random_state**：随机数生成的种子。
- **shuffle** : 每次迭代, 是否洗牌, 可选, 缺省True,仅适用于sgd或adam

```python
import numpy as np
import pandas as pd
from sklearn.neural_network import MLPRegressor
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# 加载加州房价数据集
data = load_diabetes()
feature = data.data
target = data.target

xtrain, xtest, ytrain, ytest = train_test_split(feature, target, train_size=0.7, random_state=421)

nn = MLPRegressor(hidden_layer_sizes=(100,100), activation="identity", solver="lbfgs", alpha=0.01)
model = nn.fit(xtrain, ytrain)
pre = model.predict(xtest)


index = 0
for w in model.coefs_:
    index += 1
    print('第{}层网络层:'.format(index))
    print('权重矩阵:', w.shape)
    print('系数矩阵: ', w)

plt.plot(range(len(pre)), pre, color='red', label='Predicted')
plt.plot(range(len(ytest)), ytest, color='blue', label='Actual')

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

第6章支持向量机

> 支持向量机的优点有：
>
> - 在高维空间里也非常有效
> - 对于数据维度远高于数据样本量的情况也有效
> - 在决策函数中使用训练集的子集(也称为支持向量)，因此也是内存高效利用的。
> - 通用性：可以为决策函数指定不同的核函数。已经提供了通用核函数，但也可以指定自定义核函数。
>
> 支持向量机的缺点包括：

> - 如果特征数量远远大于样本数，则在选择核函数和正则化项时要避免过度拟合。
> - SVMs不直接提供概率估计， 这些计算使用昂贵的五倍交叉验证

from sklearn.svm import SVC 分类任务

from sklearn.svm import SVR 回归任务

当然还有其他的形式例如 NuSVC、LinearSVC类等

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_breast_cancer #乳腺癌

# 加载数据集
iris = load_iris()
X = iris.data
y = iris.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 创建支持向量机模型
svm = SVC(kernel='linear', C=1, random_state=42)

# 训练模型
svm.fit(X_train, y_train)

# 预测
predictions = svm.predict(X_test)

# 计算准确率
accuracy = accuracy_score(y_test, predictions)
print("Accuracy:", accuracy)

```

常见参数：

C （float参数 默认值为1.0）
表示错误项的惩罚系数C越大，即对分错样本的惩罚程度越大，因此在训练样本中准确率越高，但是泛化能力降低；相反，减小C的话，容许训练样本中有一些误分类错误样本，泛化能力强。对于训练样本带有噪声的情况，一般采用后者，把训练样本集中错误分类的样本作为噪声。

kernel （str参数 默认为‘rbf’）

该参数用于选择模型所使用的核函数，算法中常用的核函数有：
-- linear：线性核函数
-- poly：多项式核函数
--rbf：径像核函数/高斯核
--sigmod：sigmod核函数
--precomputed：核矩阵，该矩阵表示自己事先计算好的，输入后算法内部将使用你提供的矩阵进行计算

degree （int型参数 默认为3）
该参数只对'kernel=poly'(多项式核函数)有用，是指多项式核函数的阶数n，如果给的核函数参数是其他核函数，则会自动忽略该参数。

gamma （float参数 默认为auto）
该参数为核函数系数，只对‘rbf’，‘poly’，‘sigmod’有效。如果gamma设置为auto，代表其值为样本特征数的倒数，即1/n_features，也有其他值可设定。

coef0:（float参数 默认为0.0）
该参数表示核函数中的独立项，只有对‘poly’和‘sigmod’核函数有用，是指其中的参数c。

probability（ bool参数 默认为False）
该参数表示是否启用概率估计。 这必须在调用fit()之前启用，并且会使fit()方法速度变慢。

shrinkintol: float参数 默认为1e^-3g（bool参数 默认为True）
该参数表示是否选用启发式收缩方式。

tol（ float参数 默认为1e^-3）
svm停止训练的误差精度，也即阈值。

cache_size（float参数 默认为200）
该参数表示指定训练所需要的内存，以MB为单位，默认为200MB。

class_weight（字典类型或者‘balance’字符串。默认为None）
该参数表示给每个类别分别设置不同的惩罚参数C，如果没有给，则会给所有类别都给C=1，即前面参数指出的参数C。如果给定参数‘balance’，则使用y的值自动调整与输入数据中的类频率成反比的权重。

verbose （ bool参数 默认为False）
该参数表示是否启用详细输出。此设置利用libsvm中的每个进程运行时设置，如果启用，可能无法在多线程上下文中正常工作。一般情况都设为False，不用管它。

max_iter （int参数 默认为-1）
该参数表示最大迭代次数，如果设置为-1则表示不受限制。

random_state（int，RandomState instance ，None 默认为None）
该参数表示在混洗数据时所使用的伪随机数发生器的种子，如果选int，则为随机数生成器种子；如果选RandomState instance，则为随机数生成器；如果选None,则随机数生成器使用的是np.random。

模型训练结束后，可以使用下列参数：：

- support_：支持向量的下标。
- support_vectors_：支持向量。
- dual_coef_：支持向量的系数。
- coef_：线性核中的系数。
- intercept_：决策函数中的常数。
- classes_：分类标签集合

第7章贝叶斯方法

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

# 加载数据集
iris = load_iris()
X = iris.data
y = iris.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 创建高斯朴素贝叶斯模型
gnb = GaussianNB()

# 训练模型
gnb.fit(X_train, y_train)

# 预测
predictions = gnb.predict(X_test)

# 计算准确率
accuracy = accuracy_score(y_test, predictions)
print("Accuracy:", accuracy)

```

贝叶斯参数优化

安装贝叶斯参数优化： pip install bayesian-optimization

> `bayesian-optimization` 是一个基于贝叶斯推理和高斯过程的约束全局优化包，它试图在尽可能少的迭代中找到未知函数的最值。贝叶斯最优化能够在不需要大量计算资源的情况下，有效探索参数空间。

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier
from bayes_opt import BayesianOptimization
import numpy as np

# 加载数据集
iris = load_iris()
X, y = iris.data, iris.target

# 定义优化目标函数
#先要定义一个目标函数。比如此时，函数输入为随机森林的所有参数，
#输出为模型交叉验证5次的AUC均值，作为我们的目标函数。
#因为bayes_opt库只支持最大值，所以最后的输出如果是越小越好，那么需要在前面加上负号，以转为最大值。
def decision_tree_cv(max_depth, min_samples_split):
    dt = DecisionTreeClassifier(max_depth=int(max_depth), min_samples_split=int(min_samples_split))
    scores = cross_val_score(dt, X, y, cv=5)
    return np.mean(scores)

# 定义Bayesian Optimization对象
optimizer = BayesianOptimization(
    f=decision_tree_cv,
    pbounds={"max_depth": (1, 10), "min_samples_split": (2, 20)},
    random_state=42,
)

# 执行优化过程
optimizer.maximize(init_points=5, n_iter=25)

# 输出最优参数
print("Best parameters found:")
print(optimizer.max["params"])

# 使用最优参数重新训练模型并评估性能
best_max_depth = int(optimizer.max["params"]["max_depth"])
best_min_samples_split = int(optimizer.max["params"]["min_samples_split"])
best_dt = DecisionTreeClassifier(max_depth=best_max_depth, min_samples_split=best_min_samples_split)
```

```
best_dt.fit(X, y)
accuracy = best_dt.score(X, y)
print("Test accuracy with best parameters:", accuracy)

```

第8章集成学习与实战

sklearn中集成学习方法主要包括Boosting、Bagging和Stacking等方法。以下是详细介绍：

1. **Boosting**：Boosting 是一种通过逐步训练弱学习器并加权组合来提升模型性能的方法。其代表算法包括 AdaBoost、Gradient Boosting 等。AdaBoost 通过调整样本权重来聚焦于之前被错误分类的样本，从而训练出多个弱学习器并进行加权组合。
2. **Bagging**：Bagging 通过自助采样法构建多个独立模型，然后综合这些模型的预测结果来提高整体性能。常见的 Bagging 方法包括随机森林。随机森林使用多个决策树对随机抽取的特征子集进行训练，并通过投票或平均来得到最终预测结果，从而降低过拟合风险并提高模型稳定性。
3. **Stacking**：Stacking 是一种多层集成方法，它通过将多个基模型的预测结果作为输入特征来训练一个高层模型，以综合基模型的预测。这种方法可以结合不同类型和性质的基模型，从而提高整体预测性能。

Boosting

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# 加载数据集
iris = load_iris()
X, y = iris.data, iris.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 创建AdaBoost分类器实例
ada_clf = AdaBoostClassifier(n_estimators=50, learning_rate=1.0, random_state=42)

# 训练模型
ada_clf.fit(X_train, y_train)

# 预测测试集
y_pred = ada_clf.predict(X_test)
```

```
21
22  # 计算准确率
23  accuracy = accuracy_score(y_test, y_pred)
24  print("Accuracy:", accuracy)
25
```

Bagging

```
1   from sklearn.ensemble import BaggingClassifier
2   from sklearn.datasets import load_iris
3   from sklearn.model_selection import train_test_split
4   from sklearn.metrics import accuracy_score
5
6   # 加载数据集
7   iris = load_iris()
8   X, y = iris.data, iris.target
9
10  # 划分训练集和测试集
11  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
12
13  # 创建Bagging分类器实例
14  bag_clf = BaggingClassifier(n_estimators=50, max_samples=0.8, bootstrap=True,
    random_state=42)
15
16  # 训练模型
17  bag_clf.fit(X_train, y_train)
18
19  # 预测测试集
20  y_pred = bag_clf.predict(X_test)
21
22  # 计算准确率
23  accuracy = accuracy_score(y_test, y_pred)
24  print("Accuracy:", accuracy)
25
```

随机森林

```
1   from sklearn.datasets import load_iris
2   from sklearn.model_selection import train_test_split
3   from sklearn.ensemble import RandomForestClassifier
4   from sklearn.metrics import accuracy_score
```

```
5
6   # 加载数据集
7   iris = load_iris()
8   X = iris.data
9   y = iris.target
10
11  # 划分训练集和测试集
12  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
13
14  # 创建随机森林模型
15  rf = RandomForestClassifier(n_estimators=100, random_state=42)
16
17  # 训练模型
18  rf.fit(X_train, y_train)
19
20  # 预测
21  predictions = rf.predict(X_test)
22
23  # 计算准确率
24  accuracy = accuracy_score(y_test, predictions)
25  print("Accuracy:", accuracy)
26
```

多模型融合

```
1   from sklearn.ensemble import VotingClassifier
2   from sklearn.datasets import load_iris
3   from sklearn.model_selection import train_test_split
4   from sklearn.metrics import accuracy_score
5   from sklearn.tree import DecisionTreeClassifier
6   from sklearn.linear_model import LogisticRegression
7   from sklearn.svm import SVC
8
9   # 加载数据集
10  iris = load_iris()
11  X, y = iris.data, iris.target
12
13  # 划分训练集和测试集
14  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
15
16  # 创建基学习器
17  clf1 = DecisionTreeClassifier(max_depth=4)
18  clf2 = LogisticRegression(solver='lbfgs', multi_class='multinomial')
```

```
clf3 = SVC(kernel='rbf', probability=True)

# 创建Voting分类器实例
voting_clf = VotingClassifier(estimators=[('dt', clf1), ('lr', clf2), ('svc', clf3)], voting='soft')

# 训练模型
voting_clf.fit(X_train, y_train)

# 预测测试集
y_pred = voting_clf.predict(X_test)

# 计算准确率
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

```

stacking策略

```
from sklearn.ensemble import StackingClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

# 加载数据集
iris = load_iris()
X, y = iris.data, iris.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 创建基学习器
clf1 = DecisionTreeClassifier(max_depth=4)
clf2 = LogisticRegression(solver='lbfgs', multi_class='multinomial')
clf3 = SVC(kernel='rbf', probability=True)

# 创建Stacking分类器实例
stacking_clf = StackingClassifier(estimators=[('dt', clf1), ('lr', clf2), ('svc', clf3)], final_estimator=LogisticRegression())

# 训练模型
```

```
stacking_clf.fit(X_train, y_train)

# 预测测试集
y_pred = stacking_clf.predict(X_test)

# 计算准确率
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

```

机器学习实战

贷款违约预测零基础入门金融风控-贷款违约预测_学习赛_赛题与数据_天池大赛

天池金融风控.ipynb	Add files via upload	5 minutes ago

第9章聚类

在sklearn中，提供了多种聚类算法供用户选择。以下是一些常用的聚类算法：

1. KMeans：基于距离的聚类算法，通过计算样本点到簇中心的距离来划分簇。
2. DBSCAN：基于密度的聚类算法，根据样本点的密度来确定簇的边界。
3. AgglomerativeClustering：层次聚类算法，通过逐步合并最近的簇来形成最终的簇结构。
4. MeanShift：基于核密度估计的聚类算法，通过寻找密度最大的区域来确定簇的中心。
5. SpectralClustering：基于图论的聚类算法，通过构建相似度矩阵并应用谱分解来划分簇。
6. AffinityPropagation：基于信息传递的聚类算法，通过迭代更新样本之间的亲和力来确定簇的结构。
7. OPTICS：基于密度和可达性的聚类算法，通过确定每个样本点的可达性和核心点来确定簇的结构。
8. BIRCH：平衡迭代规约和聚类的层次树状结构算法，适用于大规模数据集的聚类。
9. GaussianMixture：高斯混合模型聚类算法，通过拟合多个高斯分布来划分簇。
10. MiniBatchKMeans：基于小批量数据的KMeans聚类算法，适用于大规模数据集的聚类。

这些聚类算法各有特点，适用于不同的数据类型和问题场景。选择合适的聚类算法需要考虑数据的特点、规模以及聚类的目标等因素。

KMeans聚类进行实现

```
from sklearn.cluster import KMeans
```

```
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler

# 加载数据集
iris = load_iris()
X = iris.data

# 数据预处理
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 创建KMeans聚类实例
kmeans = KMeans(n_clusters=3, random_state=42)

# 训练模型
kmeans.fit(X_scaled)

# 获取聚类结果
labels = kmeans.labels_

print("Cluster labels:", labels)

```

第10章降维与度量

K近邻算法

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# 加载数据集
iris = load_iris()
X = iris.data
y = iris.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 创建KNN模型
knn = KNeighborsClassifier(n_neighbors=3)

```

```
# 训练模型
knn.fit(X_train, y_train)

# 预测
predictions = knn.predict(X_test)

# 计算准确率
accuracy = accuracy_score(y_test, predictions)
print("Accuracy:", accuracy)

```

低维嵌入

```
from sklearn.manifold import TSNE
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

# 加载数据集
iris = load_iris()
X = iris.data
y = iris.target

# 创建t-SNE实例
tsne = TSNE(n_components=2, random_state=42)

# 进行低维嵌入
X_embedded = tsne.fit_transform(X)

# 可视化结果
plt.scatter(X_embedded[:, 0], X_embedded[:, 1], c=y)
plt.colorbar()
plt.show()

```

主成分分析

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

# 加载数据集
iris = load_iris()
```

```
X = iris.data
y = iris.target

# 创建PCA实例
pca = PCA(n_components=2)

# 进行主成分分析
X_pca = pca.fit_transform(X)

# 可视化结果
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y)
plt.colorbar()
plt.show()

```

核化线性降维

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

# 加载数据集
iris = load_iris()
X = iris.data
y = iris.target

# 创建KernelPCA实例
kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)

# 进行核化线性降维
X_kpca = kpca.fit_transform(X)

# 可视化结果
plt.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y)
plt.colorbar()
plt.show()

```

流形学习

在sklearn中，可以使用Isomap或t-SNE等算法来实现流形学习

```
from sklearn.manifold import Isomap
```

```
from sklearn.datasets import load_digits
import matplotlib.pyplot as plt

# 加载数据集
digits = load_digits()
X = digits.data
y = digits.target

# 创建Isomap实例
isomap = Isomap(n_components=2)

# 进行流形学习
X_isomap = isomap.fit_transform(X)

# 可视化结果
plt.scatter(X_isomap[:, 0], X_isomap[:, 1], c=y)
plt.colorbar()
plt.show()

```

度量学习

```
from sklearn.metrics import pairwise_kernels
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

# 加载数据集
iris = load_iris()
X = iris.data
y = iris.target

# 计算核矩阵
K = pairwise_kernels(X, metric='rbf', gamma=15)

# 可视化结果
plt.imshow(K, cmap='hot', interpolation='nearest')
plt.colorbar()
plt.show()

```

第11章特征选择与稀疏学习

子集搜索

在sklearn中，可以使用SelectKBest类来实现特征选择中的子集搜索

```
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectKBest, chi2
import numpy as np

# 加载数据集
iris = load_iris()
X = iris.data
y = iris.target

# 创建SelectKBest实例
selector = SelectKBest(chi2, k=2)

# 进行子集搜索
X_new = selector.fit_transform(X, y)

# 获取所选特征的索引
selected_features = selector.get_support(indices=True)
print("Selected features:", selected_features)

```

过滤式选择

在sklearn中，可以使用SelectKBest类来实现过滤式特征选择

```
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectKBest, chi2
import numpy as np

# 加载数据集
iris = load_iris()
X = iris.data
y = iris.target

# 创建SelectKBest实例
selector = SelectKBest(chi2, k=2)

# 进行过滤式特征选择
X_new = selector.fit_transform(X, y)

```

```
# 获取所选特征的索引
selected_features = selector.get_support(indices=True)
print("Selected features:", selected_features)

```

包裹式选择

```
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectFromModel
from sklearn.linear_model import LogisticRegression
import numpy as np

# 加载数据集
iris = load_iris()
X = iris.data
y = iris.target

# 创建Logistic回归模型实例
clf = LogisticRegression(solver='liblinear', penalty='l1', C=0.5)

# 创建SelectFromModel实例
selector = SelectFromModel(clf, threshold='mean')

# 进行包裹式特征选择
X_new = selector.fit_transform(X, y)

# 获取所选特征的索引
selected_features = selector.get_support(indices=True)
print("Selected features:", selected_features)

```

嵌入式选择

```
from sklearn.datasets import load_iris
from sklearn.linear_model import Lasso
import numpy as np

# 加载数据集
iris = load_iris()
X = iris.data
y = iris.target

```

```
# 创建Lasso实例
lasso = Lasso(alpha=0.1)

# 进行嵌入式特征选择
lasso.fit(X, y)

# 获取所选特征的系数
coef = lasso.coef_
print("Feature coefficients:", coef)

```

稀疏表示

```
from sklearn.datasets import load_digits
from sklearn.decomposition import SparsePCA
import matplotlib.pyplot as plt

# 加载数据集
digits = load_digits()
X = digits.data
y = digits.target

# 创建SparsePCA实例
spca = SparsePCA(n_components=2, alpha=1)

# 进行稀疏表示
X_spca = spca.fit_transform(X)

# 可视化结果
plt.scatter(X_spca[:, 0], X_spca[:, 1], c=y)
plt.colorbar()
plt.show()

```

压缩感知

```
from sklearn.datasets import load_digits
from sklearn.decomposition import SparsePCA
from sklearn.linear_model import Lasso
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.metrics import mean_squared_error
```

```
import numpy as np

# 加载数据集
digits = load_digits()
X = digits.data
y = digits.target

# 数据预处理
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 创建稀疏表示模型
spca = SparsePCA(n_components=2, alpha=1)
X_spca = spca.fit_transform(X_scaled)

# 创建稀疏编码器
lasso = Lasso(alpha=0.1)
lasso.fit(X_spca, y)
coder = lasso.coef_

# 重建数据
X_reconstructed = np.dot(X_spca, coder)

# 计算重构误差
mse = mean_squared_error(X_scaled, X_reconstructed)
print("Mean squared error:", mse)

```

第12章计算学习理论

暂无

第13章半监督学习

半监督SVM

```
from sklearn.datasets import load_digits
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.semi_supervised import LabelSpreading
import numpy as np

# 加载数据集
```

```
digits = load_digits()
X = digits.data
y = digits.target

# 数据预处理
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.2, random_state=42)

# 创建半监督学习模型
label_spread = LabelSpreading(kernel='knn', alpha=0.8)
label_spread.fit(X_train, y_train)

# 预测测试集
y_pred = label_spread.predict(X_test)

# 计算准确率
accuracy = np.mean(y_pred == y_test)
print("Accuracy:", accuracy)

```

图半监督学习

```
import numpy as np
from scipy.sparse import csr_matrix
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

class GraphConvolutionalNetwork:
    def __init__(self, n_features, n_classes):
        self.W = np.random.randn(n_features, n_classes)

    def forward(self, X, A):
        return np.dot(A, np.dot(X, self.W))

    def fit(self, X, y, A, epochs=100, learning_rate=0.01):
        for epoch in range(epochs):
            # Forward pass
            output = self.forward(X, A)
            # Backward pass (using gradient descent)
            error = output - y
```

```
            grad_W = np.dot(X.T, np.dot(A, error)) / X.shape[0]
            self.W -= learning_rate * grad_W

    def predict(self, X, A):
        return np.argmax(self.forward(X, A), axis=1)

# 生成模拟数据
n_samples = 1000
n_features = 20
n_classes = 3
X = np.random.randn(n_samples, n_features)
y = np.random.randint(0, n_classes, n_samples)
A = np.random.rand(n_samples, n_samples)
A = (A + A.T) / 2  # Make the adjacency matrix symmetric
A[np.diag_indices_from(A)] = 0  # Set diagonal elements to zero
A = csr_matrix(A)  # Convert to sparse matrix for efficiency

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 标准化特征
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# 训练模型
gcn = GraphConvolutionalNetwork(n_features, n_classes)
gcn.fit(X_train, y_train, A, epochs=100, learning_rate=0.01)

# 预测并评估性能
y_pred = gcn.predict(X_test, A)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

```

分歧方法监督学习

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.semi_supervised import LabelSpreading
from sklearn.metrics import accuracy_score

```

```
# 加载数据集
digits = load_digits()
X, y = digits.data, digits.target

# 数据预处理
scaler = StandardScaler()
X = scaler.fit_transform(X)

# 划分标记和未标记数据
n_labeled = 100
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
y_train[:n_labeled] = y_train[:n_labeled].astype(np.int64)
y_train[n_labeled:] = -1

# 训练模型
label_spread = LabelSpreading(kernel='knn', alpha=0.8)
label_spread.fit(X_train, y_train)

# 预测并评估性能
y_pred = label_spread.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

```

半监督聚类

```
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.semi_supervised import LabelPropagation
import numpy as np

# 创建一个示例数据集
data, labels = make_blobs(n_samples=200, centers=3, random_state=42)
labels[10:15] = -1  # 将部分标签设置为-1，表示未标记

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.2, random_state=42)

# 初始化LabelPropagation对象
label_prop_model = LabelPropagation()

# 对训练集进行拟合和预测
label_prop_model.fit(X_train, y_train)
```

```
y_pred = label_prop_model.predict(X_test)

print("预测结果: ", y_pred)

```