

机器学习实验报告

摘要

本实验旨在实现、比较、探索经典的机器学习算法，涵盖了线性回归、KNN、逻辑回归、决策树、SVM、贝叶斯分类、聚类、神经网络和集成学习等多个机器学习算法的实现和应用。实验使用不同的数据集和方法来探索这些算法在分类、回归、聚类问题上的性能。通过实验，我深入了解了各种算法的原理和应用，并进行了性能比较。

在实验中，首先介绍每个算法的原理和基本概念，然后详细讨论了它们的实现方法和应用场景，尝试自己编写代码实现，并与调用 sklearn 包的结果进行比较。通过这些实验，我对不同算法在不同任务上的性能有了全面的了解，这些实验为进一步学习和研究应用提供了基础和参考，对于选择合适的算法和优化模型性能具有重要的指导意义。

关键词：机器学习 算法 分类 回归 聚类

Experiment Report of Machine Learning

ABSTRACT

This experiment aims to implement, compare, and explore classic machine learning algorithms, covering multiple machine learning algorithms such as linear regression, KNN, logistic regression, decision tree, SVM, Bayesian classification, clustering, neural network, and ensemble learning. Experiments use different data sets and methods to explore the performance of these algorithms on classification, regression, and clustering problems. Through experiments, I gained an in-depth understanding of the principles and applications of various algorithms and conducted performance comparisons.

In the experiment, I first introduced the principles and basic concepts of each algorithm, then discussed their implementation methods and application scenarios in detail, and tried to write the code implementation myself to compare with the results of calling the sklearn package. Through these experiments, I have a comprehensive understanding of the performance of different algorithms on different tasks. These experiments provide a basis and reference for further learning and research applications, and have important guiding significance for selecting appropriate algorithms and optimizing model performance..

Key words: Machine learning Algorithm Classification Regression Clustering

目录

第 1 章	线性回归	2
1.1	基本概念	2
1.2	梯度下降算法求解多元线性回归	2
1.3	蒙特卡洛模拟方法计算 p-value	6
1.4	adam 算法及 sklearn 库调用比较	9
第 2 章	KNN	13
2.1	基本概念	13
2.2	实现 KNN 算法	13
2.3	KNN 算法的应用: iris	14
2.4	调用 sklearn 库分类	17
第 3 章	逻辑回归	19
3.1	基本概念	19
3.2	梯度下降求解逻辑回归	20
3.3	逻辑回归的应用: iris 数据集	21
第 4 章	决策树	26
4.1	基本概念	26
4.2	决策树的应用: iris	26
4.3	参数探究	28
4.4	完成非线性分类	35
第 5 章	SVM	37
5.1	基本概念	37
5.2	SVM 的应用: iris	39
5.3	SVM 的应用: LR-testSet2.txt	39
第 6 章	贝叶斯分类	43
6.1	基本概念	43
6.2	贝叶斯的应用: iris	44
第 7 章	聚类	47
7.1	K-means	47
7.2	DBscan	51
7.3	DBSCAN 与 K-means 的比较	59

第 8 章 神经网络	62
8.1 基本概念	62
8.2 编写反向传播算法	64
8.3 分类任务及参数探究	67
8.4 求解手写数字识别问题	72
第 9 章 集成学习	75
9.1 基本概念	75
9.2 Bagging 与随机森林：求解数字识别问题	75
9.3 信用卡申请识别	78
结束语	110
参考文献	111

第1章 线性回归

1.1 基本概念

线性回归是回归问题中的一种，线性回归假设目标值与特征之间线性相关，即满足一个多元一次方程。给定数据集 $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_m, y_m)\}$ ，其中 $\mathbf{x}_i = (x_{i1}; x_{i2}; \dots; x_{id})$, $y_i \in \mathbb{R}$ 。“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，即

$$f(\mathbf{x}_i) = \mathbf{w}^T \mathbf{x}_i + b, \text{ 使得 } f(\mathbf{x}_i) \simeq y_i,$$

这称为多元线性回归。

如何确定 \mathbf{w} 和 b 呢？显然关键在于如何衡量 $f(\mathbf{x}_i)$ 和 y_i 之间的差别。为便于讨论，把 \mathbf{w} 和 b 吸收入向量形式 $\hat{\mathbf{w}} = (\mathbf{w}; b)$ ，再把标记也写成向量形式 $\mathbf{y} = (y_1; y_2; \dots; y_m)$ ，则有

$$\hat{\mathbf{w}}^* = \arg \min_{\hat{\mathbf{w}}} (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^T (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}).$$

令 $E_{\hat{\mathbf{w}}} = (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^T (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})$ ，对 $\hat{\mathbf{w}}$ 求导得到

$$\frac{\partial E_{\hat{\mathbf{w}}}}{\partial \hat{\mathbf{w}}} = 2\mathbf{X}^T (\mathbf{X}\hat{\mathbf{w}} - \mathbf{y}).$$

对此，可以令其为零求得最优解的闭式解，也可以使用梯度下降算法求解。

1.2 梯度下降算法求解多元线性回归

梯度下降算法需要在这里引入一个概念：学习率，即移动的步长 η 。其步骤为：

Step 1：随机选取一个 \mathbf{w}^0 ；

Step 2：计算微分，也就是当前的斜率，根据斜率来判定移动的方向（梯度负方向）；

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \vdots \\ \frac{\partial L}{\partial w_n} \end{bmatrix}$$

- ◆ 大于 0 向右移动（增加 w ）
- ◆ 小于 0 向左移动（减少 w ）

Step3：根据学习率移动；

重复 Step2 和 Step3，直到找到最低点。

编写梯度下降的函数如下：

```

1. def gradient_descent(x, y, learning_rate=0.01, num_iterations=10000):
2.     n_samples, n_features = x.shape
3.     theta = np.zeros(n_features+1) # 初始化参数向量（包括截距项）
4.     x = np.column_stack((np.ones(n_samples), x)) # 添加偏置列
5.     loss_record = [] # 记录损失函数值的变化
6.     gradient_norm_record = [] # 记录梯度的模的变化
7.
8.     for i in range(num_iterations):
9.         y_fit = np.dot(x, theta) # 计算拟合值
10.        gradient = np.dot(x.T, (pd.DataFrame(y_fit)-y))/n_samples # 计算梯度
11.        theta = theta-learning_rate*gradient # 更新参数
12.        loss = np.mean((y_fit-y)**2) # 计算 loss function 值
13.        loss_record.append(loss) # 记录 loss function 值
14.        gradient_norm_record.append(np.linalg.norm(gradient)) # 计算并记录梯度模
15.        theta = theta[:, 0]
16.
17.    y_fit = pd.DataFrame(y_fit[:, 0]).values
18.
19.    R_square = (np.sum(np.square(y-np.mean(y)))-np.sum(np.square(y-y_fit)))/np.sum(np.square(y-np.mean(y)))
20.    F = (np.sum(np.square(y-np.mean(y)))-np.sum(np.square(y-y_fit)))/np.sum(np.square(y-y_fit))*(n_samples-n_features-1)/(n_features)
21.    return theta, R_square, F, loss_record, gradient_norm_record

```

开始时，初始化参数向量，其中需要注意的是初始化的参数向量应等于特征数+1（截距项），且为 X 添加一列全为 1 的偏置列。同时创建列表以记录损失函数值的变化及梯度的模的变化，便于之后作图。在循环中，根据梯度下降的步骤计算出梯度，再沿着梯度负方向更新，在每一步中记录损失函数值及梯度的模。一定的迭代次数后，计算拟合值 \hat{y} ，可决系数 R^2 以及 F ，便于之后 p-value 的计算。

同时编写绘制损失函数值及梯度的模随迭代次数变化的折线图函数如下：

```

1. def draw(num_iterations, loss_record, gradient_norm_record):
2.     plt.figure(figsize=(10, 4))
3.     plt.subplot(1, 2, 1)
4.     plt.plot(range(num_iterations), loss_record)
5.     plt.xlabel('迭代次数')
6.     plt.ylabel('损失函数值')
7.     plt.title('损失函数变化')
8.
9.     plt.subplot(1, 2, 2)
10.    plt.plot(range(num_iterations), gradient_norm_record)
11.    plt.xlabel('迭代次数')

```

```
12.     plt.ylabel('梯度模')
13.     plt.title('梯度模变化')
14.
15.     plt.tight_layout()
16.     plt.show()
```

1.2.1 应用梯度下降线性回归

使用的数据集来源：“Data for Admission in the University” on Kaggle

① 数据探索

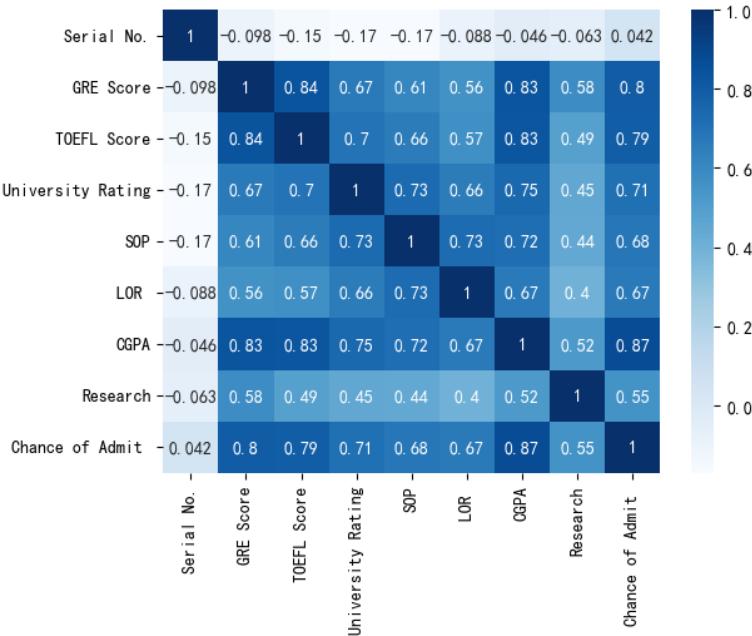
读取数据集后使用 *data.info()* 函数：

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 9 columns):
 #   Column           Non-Null Count  Dtype  
---  --  
 0   Serial No.       400 non-null    int64  
 1   GRE Score        400 non-null    int64  
 2   TOEFL Score      400 non-null    int64  
 3   University Rating 400 non-null    int64  
 4   SOP              400 non-null    float64 
 5   LOR              400 non-null    float64 
 6   CGPA             400 non-null    float64 
 7   Research          400 non-null    int64  
 8   Chance of Admit  400 non-null    float64 
dtypes: float64(4), int64(5)
memory usage: 28.3 KB
```

可以看出，数据集中均无缺失值，故无需进行缺失值处理。各变量的含义如下：

- Serial No.: 序号
- GRE Score: GRE 分数
- TOFEL Score: 托福分数
- University Rating: 大学评级
- SOP: Statement of Purpose Strength
- LOR: Letter of Recommendation Strength
- CGPA: Undergraduate GPA
- Research: Research Experience
- Chance of Admit: 录取几率

调用 *sns.heatmap()* 函数绘制相关度热图：



可以发现, Chance of Admit 与其他特征均有较强相关性, 因此可使用多元线性回归。

② 使用多元线性回归

在进行多元线性回归前, 需要对数据集进行处理:

```

1. # 划分 x 与 y
2. x = data.iloc[:, :-1].values
3. y = data.iloc[:, -1].values
4. y = y.reshape(-1, 1)
5.
6. # 划分训练集与测试集
7. from sklearn.model_selection import train_test_split
8. x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25, random_state=0)
9.
10. # 标准化
11. from sklearn.preprocessing import StandardScaler
12. sc = StandardScaler()
13. x_train = sc.fit_transform(x_train)
14. x_test = sc.transform(x_test)

```

处理完成后, 便可调用之前编写的线性回归函数和折线图绘制函数进行多元线性回归并可视化损失函数、梯度的变化过程。调用的模板如下:

```

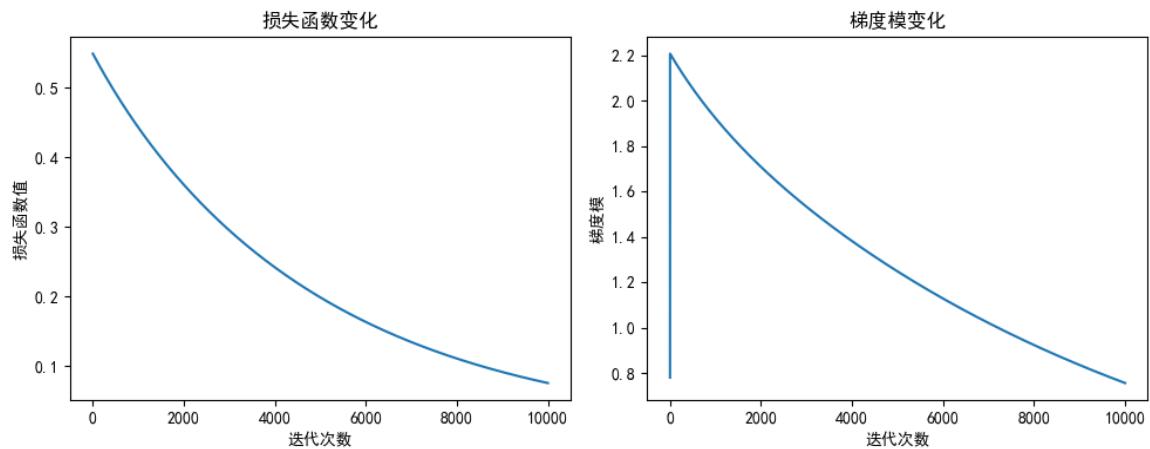
1. theta, R_square, F, lr, gr = gradient_descent(x_train, y_train, 0.01, 10000)
2. draw(10000, lr, gr)

```

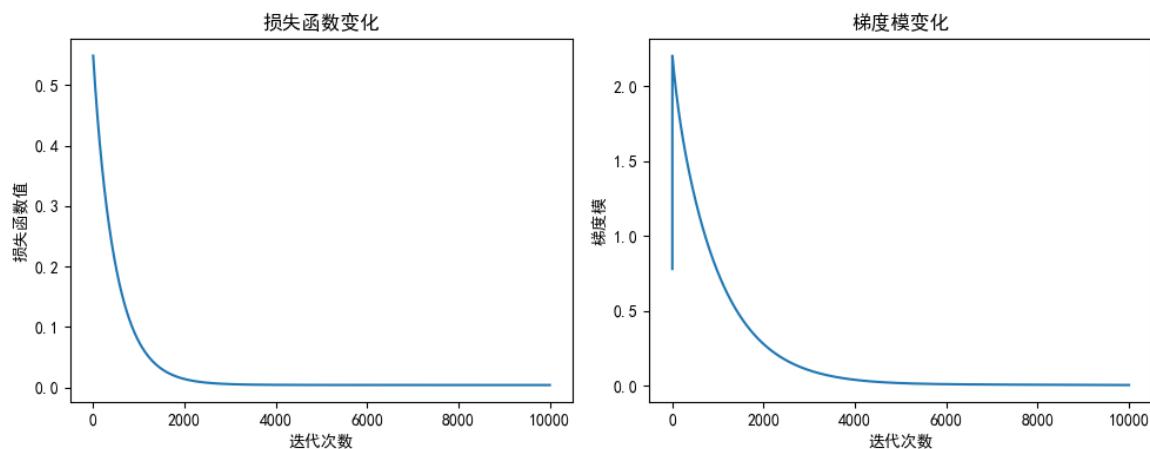
lr, gr 分别用来接收线性回归函数输出的损失函数列表、梯度的模列表。损失函数调用时, 可以更改学习率及迭代次数, 以比较不同参数对回归结果的影响。

固定迭代次数为 10000，探究不同学习率对于算法收敛情况的影响：

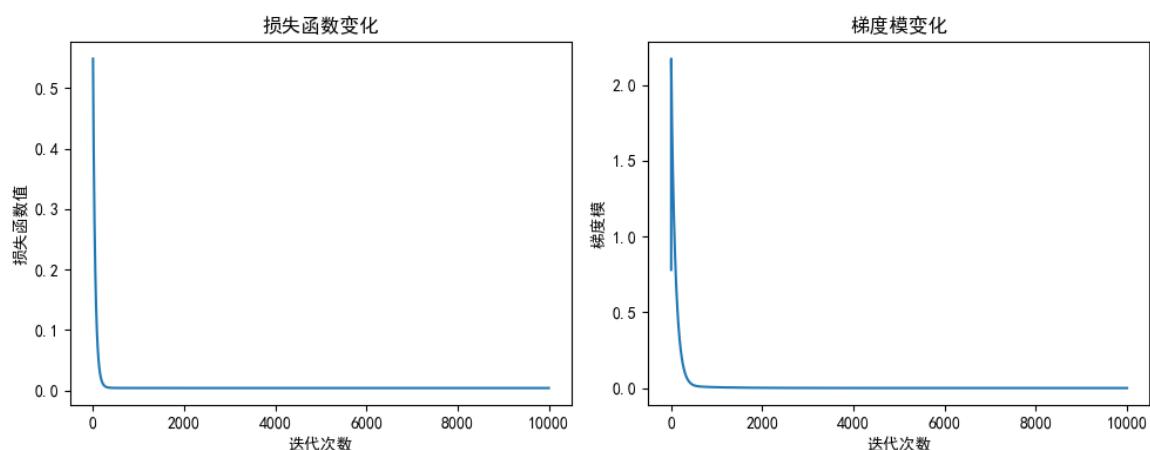
学习率 $\eta=0.0001$



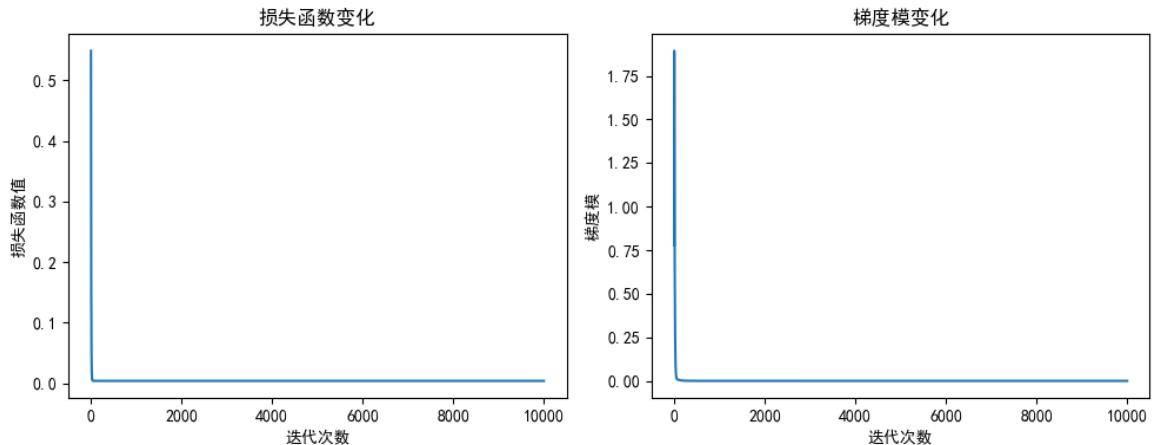
学习率 $\eta=0.001$



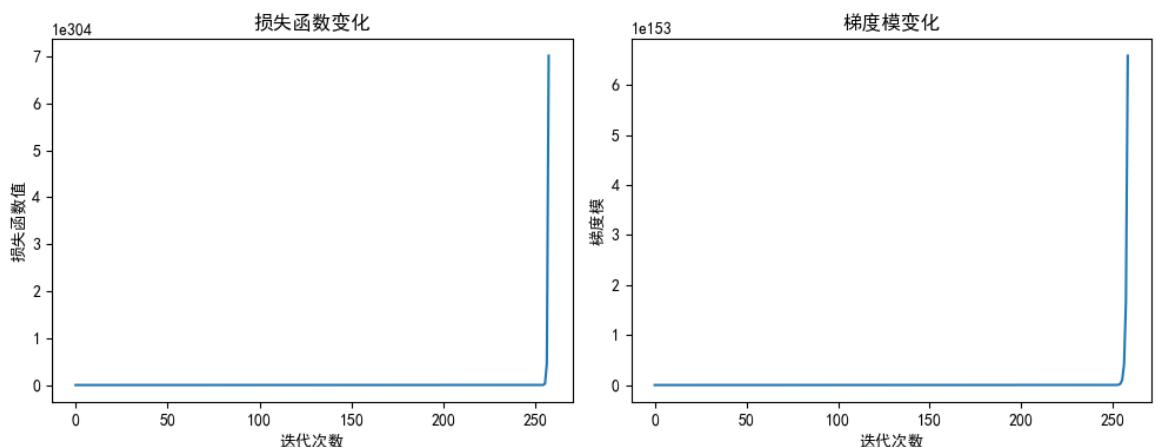
学习率 $\eta=0.01$



学习率 $\eta=0.1$



学习率 $\eta=1$



根据上面不同学习率多元线性回归的结果，可以发现当学习率小时，收敛速度慢；学习率大时，收敛速度快。但是，注意到 $\eta=0.0001$ （学习率过小）时，在迭代次数为 10000 次时，尚未达到收敛，即需要更多轮次数的迭代才能收敛。而 $\eta=1$ （学习率过大）时，会出现梯度爆炸的现象导致不能收敛。

从中可以得出结论，在进行学习率选择时需合理，既不能太大，也不能太小。过小的学习率会使收敛速度过慢，需要更多的迭代次数才能使其收敛。而过大的学习率则可能导致梯度爆炸，继而无法收敛。

1.3 蒙特卡洛模拟方法计算 p-value

要想计算 p-value，首先得清楚 R^2 以及 F 的概念。

可决系数 R^2 是回归平方和占总平方和的比例，反映了回归模型对因变量变异的解释能力。该统计量表示自变量共同解释因变量方差的百分比，其大小衡量了预测值对真

值的拟合好坏程度。其计算公式为

$$R^2 = \frac{SS(\text{mean}) - SS(\text{fit})}{SS(\text{mean})} = \frac{TSS - RSS}{TSS},$$

通常情况下， R^2 越大，回归模型越符合观测结果。

F 统计量应用于模型的总体显著性检验，就是检验全部解释变量对被解释变量的共同影响是否显著。即检验方程 $y_t = b_0 + b_1 x_{1t} + b_2 x_{2t} + \dots + b_k x_{kt} + u_t$ 中参数是否整体显著不为 0。按照假设检验的原理与程序，提出原假设与备择假设为

$$\begin{aligned} H_0: \quad & b_1 = b_2 = \dots = b_k = 0, \\ H_1: \quad & b_j (j=1, 2, \dots, k) \text{ 不全为 } 0. \end{aligned}$$

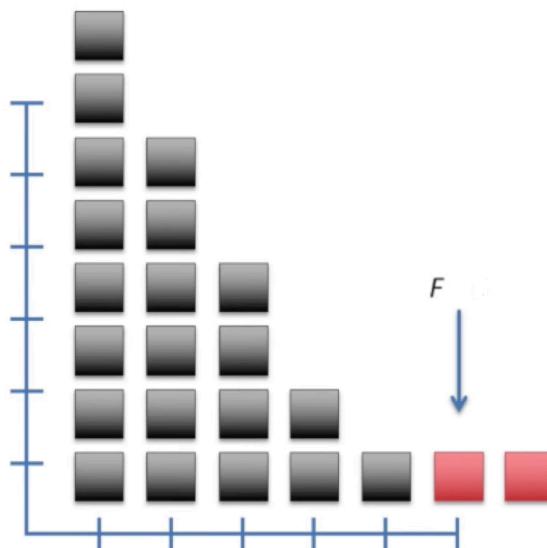
考虑自由度进行方差分析，有如下方差分析表：

变异来源	平方和	自由度
源于回归	ESS	k
源于残差	RSS	$n-k-1$
总变差	TSS	$n-1$

根据数理统计学中关于 F 统计量的定义可知，在 H_0 成立的条件下，

$$F = \frac{ESS/k}{RSS/(n-k-1)}.$$

对于任意一组参数，我们都可以计算出一个 F 值。将这些 F 值同我们回归得出的 F 值进行比较，便可求出 p 值。 p 值可通俗理解为现在 \mathbf{X} 与 \mathbf{y} 的关系是碰运气碰到的概率，故 p 值越小越好。



1.3.1 计算 p 值函数的编写

将计算 p 值的代码编写为一个函数以方便调用。

```

1. def calculate_p_value(x, y, actual_F, n_simulations=1000):
2.     n_samples, n_features = x.shape
3.     simulated_slopes = []
4.
5.     for i in range(n_simulations):
6.         random_X = np.random.rand(n_samples, n_features)
7.         random_y = pd.DataFrame(np.random.rand(n_samples)).values
8.
9.         # 拟合
10.        theta, R_sqare, F, loss_record, gradient_norm_record = gradient_descent(ra
ndom_X, random_y, 0.0001, 1000)
11.        simulated_slopes.append(F)
12.
13.
14.    p_value = np.mean(np.array(simulated_slopes) >= actual_F)
15.    return p_value

```

此函数的核心思想是随机生成 \mathbf{X} 与 \mathbf{y} ，然后调用之前编写的梯度下降线性回归函数计算出 F 值。经过 n 轮模拟后，将本次回归得到的 F 值与这些模拟得到的 F 比较，计算有多少模拟得到的 F 值大于本次回归得到的 F 值，比值 $\frac{\text{count}(F_{\text{simulate}} > F_{\text{actual}})}{\text{rounds}}$ 即为 p 值。

1.3.2 在应用中计算 p 值

```

1. theta,R_square, F, lr, gr = gradient_descent(x_train, y_train, 0.01, 10000)
2. p = calculate_p_value(x_train, y_train, F, 10)
3. print("R_square=",R_square)
4. print("F=",F)
5. print("p_value=",p)

```

在计算 p 值前，首先需进行线性回归，计算出 F 值，再将其作为参数传入计算 p 值的函数中。在本训练集上，上述代码的输出为：

```
R_square= 0.8238709738862324
F= 195.12507367296868
p_value= 0.0
```

调用 `statsmodels.api` 进行 p 值的计算

```

1. import statsmodels.api as sm
2. sm_model = sm.OLS(y_train, np.column_stack((np.ones(x_train.shape[0]), x_train)))
3. results = sm_model.fit()
4. summary = results.summary()
5. print(summary)

```

其输出为：

OLS Regression Results						
Dep. Variable:		y	R-squared:	0.824		
Model:		OLS	Adj. R-squared:	0.820		
Method:		Least Squares	F-statistic:	195.1		
Date:		Sat, 23 Dec 2023	Prob (F-statistic):	3.84e-106		
Time:		20:20:19	Log-Likelihood:	412.74		
No. Observations:		300	AIC:	-809.5		
Df Residuals:		292	BIC:	-779.8		
Df Model:		7				
Covariance Type:		nonrobust				
	coef	std err	t	P> t	[0.025	0.975]
const	0.7262	0.004	203.004	0.000	0.719	0.733
x1	0.0227	0.008	2.859	0.005	0.007	0.038
x2	0.0143	0.008	1.905	0.058	-0.000	0.029
x3	0.0056	0.006	0.871	0.385	-0.007	0.018
x4	-0.0060	0.006	-0.939	0.348	-0.019	0.007
x5	0.0194	0.006	3.369	0.001	0.008	0.031
x6	0.0796	0.009	9.078	0.000	0.062	0.097
x7	0.0106	0.004	2.399	0.017	0.002	0.019
Omnibus:	52.322	Durbin-Watson:			2.084	
Prob(Omnibus):	0.000	Jarque-Bera (JB):			96.806	
Skew:	-0.941	Prob(JB):			9.53e-22	
Kurtosis:	5.049	Cond. No.			6.38	
Notes:						
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.						

比较 R^2 、F、p 值，与自己编写的代码计算结果相同。p=0，说明现在这种情况是碰运气碰到的概率接近为 0，模型整体显著。

1.4 adam 算法及 sklearn 库调用比较

adam 算法是一种基于“momentum”思想的随机梯度下降优化方法，通过迭代更新之前每次计算梯度的 moment，并计算滑动平均值，后用来更新当前的参数。这种思想结合了 momentum 算法的处理稀疏型数据，又结合了 RMSProp 算法可以处理非稳态的数据，取得了非常好的测试性能。

1.4.1 adam 算法编写

```

1. def Adam(x, y, learning_rate=0.01, num_iterations=10000, momentum=0.9, decay=0.9):
2.     n_samples, n_features = x.shape
3.     theta = np.zeros(n_features+1) # 初始化参数向量（包括截距项）
4.     x = np.column_stack((np.ones(n_samples), x))
5.     loss_record = [] # 记录损失函数值的变化
6.     gradient_norm_record = [] # 记录梯度的模的变化
7.     velocity = np.zeros_like(theta)
8.     rmsprop_cache = np.zeros_like(theta)
9.
10.    for i in range(num_iterations):
11.        y_fit = np.dot(x, theta) # 计算拟合值
12.        gradient = np.dot(x.T, (pd.DataFrame(y_fit)-y))/n_samples # 计算梯度
13.
14.        # 使用动量
15.        velocity = momentum * velocity + learning_rate * gradient
16.
17.
18.        # 使用 RMSProp 更新参数
19.        rmsprop_cache = decay * rmsprop_cache + (1 - decay) * gradient**2
20.        theta = theta - learning_rate / np.sqrt(rmsprop_cache) * velocity
21.
22.        loss = np.mean((y_fit-y)**2) # 计算 loss function 值
23.        loss_record.append(loss) # 记录 loss function 值
24.        gradient_norm_record.append(np.linalg.norm(gradient)) # 计算并记录梯度模
25.        theta = theta[:,0]
26.    print(theta)
27.
28.    y_pre = pd.DataFrame(np.dot(x, theta))
29.    F = (np.sum((y-np.mean(y))**2)-np.sum((y-y_pre)**2))/(np.sum((y-y_pre)**2)*
30.                                              (n_samples-n_features)/(n_features-1))
31.    return theta, F, loss_record, gradient_norm_record

```

开始时，与之前梯度下降求解线性回归函数 `gradient_descent()`一样，初始化参数向量，参数向量长度应等于特征数+1（截距项），且为 X 添加一列全为 1 的偏置列。同时创建列表以记录损失函数值的变化及梯度的模的变化，便于之后作图。与之前函数不同的是，`adam` 还需要初始化一个用于存储动量的向量及一个用于存储 RMSProp 值的向量，以便之后的参数更新。

在循环过程中，首先拟合并计算梯度，再综合使用 `momentum` 和 `RMSProp` 进行参数更新，最后计算并记录 loss function 的值及梯度的模。迭代完成后，可以计算拟合的 y 值，并计算 F 统计量。

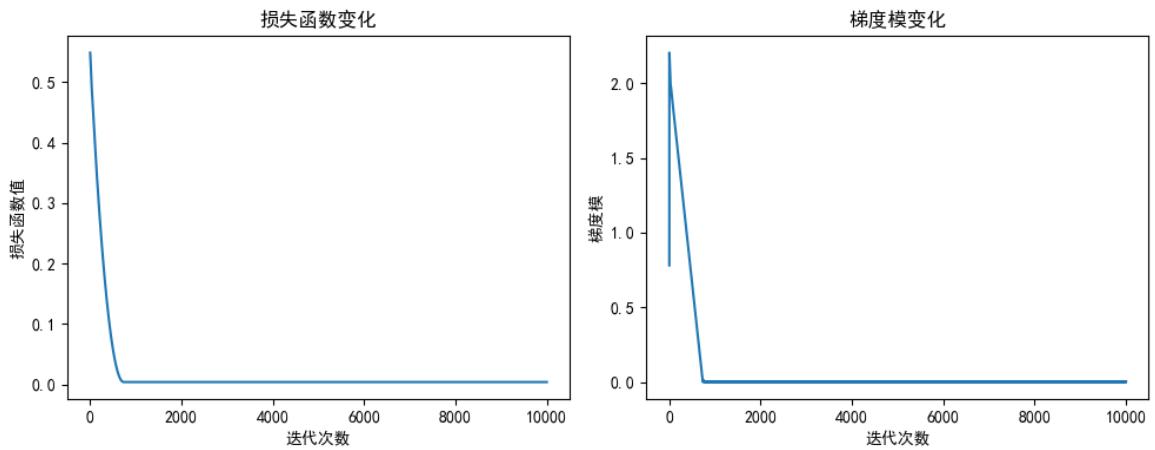
1.4.2 adam 算法的应用及比较

设置迭代次数为 10000 次，学习率为 0.01，momentum 参数 $\lambda = 0.9$ ，RMSProp 参数 $\alpha = 0.9$ ，调用编写的 adam 算法函数对之前的数据集进行拟合并作图：

```
1. theta, F, lr, gr = Adam(x_train, y_train, 0.01, 10000, 0.9, 0.9)
2. draw(10000, lr, gr)
```

其输出为：

```
[ 0.72621375  0.02261761  0.01426891  0.00558776 -0.00606063
 0.01933097  0.07952987  0.01058682]
```



可以发现相较于之前的 `grediant_descent()`，对于相同的参数（迭代次数为 10000 次，学习率为 0.01），adam 算法收敛的速度较慢。其原因可能是 momentum 参数 $\lambda = 0.9$ 和 RMSProp 参数 $\alpha = 0.9$ 导致对之前方向的考虑过大，对当前的变化不太敏感。

接下来调用 sklearn 包进行线性回归：

```
1. from sklearn.linear_model import LinearRegression
2.
3. model = LinearRegression()
4. model.fit(x_train, y_train)
```

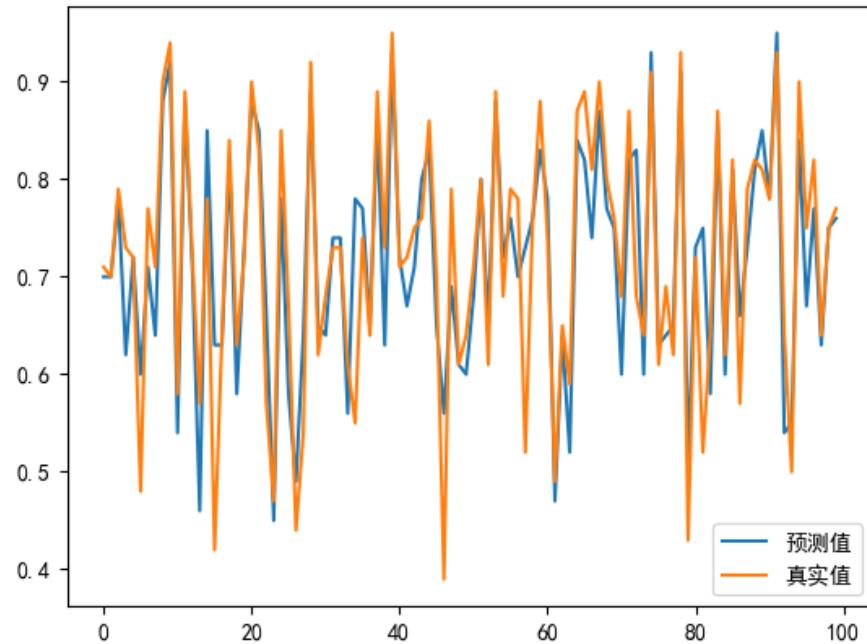
调用 `model.coef_` 以及 `model.intercept_` 获取参数和截距项，输出结果分别为

```
array([[ 0.02266006,  0.01431135,  0.0056302 , -0.00601818,
 0.01937341,  0.07957232,  0.01062926]])
array([0.72623333])
```

与 adam 输出结果进行比较，两者相差不大，大致相同。

最后，观察在测试集上的拟合程度，绘制折线图

```
1. y_pred = np.round(model.predict(x_test),2)
2. x = range(len(y_pred))
3. plt.figure()
4. plt.plot(x, y_pred, label='预测值')
5. plt.plot(x, y_test, label='真实值')
6. plt.legend()
7. plt.show()
```



可以看出，预测值与真实值拟合度较高，线性回归模型具有良好的预测效果。

第 2 章 KNN

2.1 基本概念

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。

Algorithm k-Nearest Neighbor(KNN)

Input:

$(x, y) \in D$ ：训练样本

(x', y') ：测试样本

K：考虑的近邻个数

```

1  for 每个测试样本  $z = (x', y')$  do
2      计算  $z$  和每个训练样本  $(x, y) \in D$  的距离
3      选择离  $z$  最近的 K 个训练样本的集合
4       $y' = \arg \max \sum_{(x, y) \subseteq D_z}^K I(v = y_i)$  (根据 K 个近邻样本确定测试样本类别)
5  end

```

2.2 实现 KNN 算法

KNN 算法简单易理解。由于本次实验的任务是对 iris 数据进行分类，距离度量采用欧氏距离，最后投票决定测试样本类别。根据上述伪代码，将 KNN 算法编写成一个类：

```

1.  class KNN:
2.      def __init__(self, k=3):
3.          self.k = k
4.
5.      def fit(self, X_train, y_train):

```

```
6.         self.X_train = X_train
7.         self.y_train = y_train
8.
9.     def predict(self, X_test):
10.        predictions = [self._predict(x) for x in X_test]
11.        return np.array(predictions)
12.
13.    def _predict(self, x):
14.        # 计算距离
15.        distances = [np.linalg.norm(x - x_train) for x_train in self.X_train]
16.
17.        # 获取距离最近的 k 个训练样本
18.        k_neighbors_indices = np.argsort(distances)[:self.k]
19.
20.        # 获取 k 个最近邻居的标签
21.        k_neighbor_labels = [self.y_train[i] for i in k_neighbors_indices]
22.
23.        # 投票决定
24.        most_common = np.bincount(k_neighbor_labels).argmax()
25.        return most_common
```

其中`__init__`用于实例化 KNN 模型，默认的 K 值为 3，也可自己设置其他 K 值，一般为奇数，以方便“投票”。`fit`函数的功能及其简单，仅仅只是记录训练集（KNN 算法没有训练过程，被称为“懒惰学习”）。而在`predict`函数中，会对每个测试集的样本调用`_predict`函数，即进行距离计算、获取 K 个近邻、根据近邻标签划分类别的步骤。

2.3 KNN 算法的应用：iris

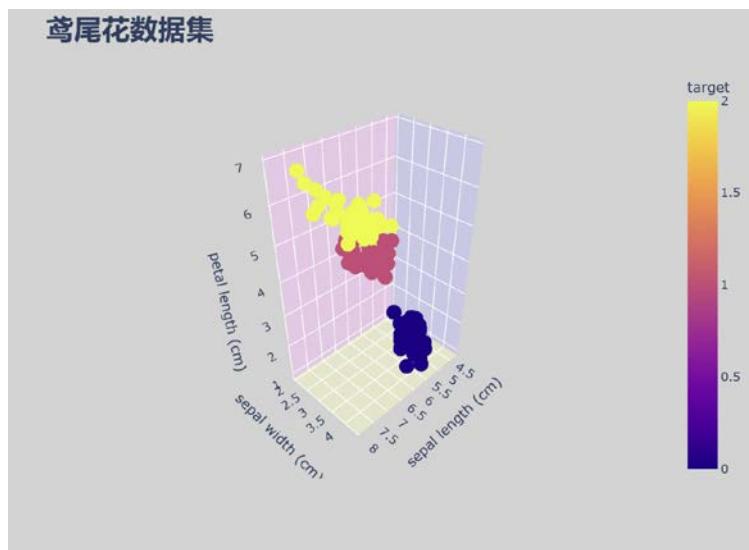
首先使用 sklearn 加载 iris 数据集，并将数据集划分为训练集和测试集。

```
1. from sklearn.datasets import load_iris
2. from sklearn.model_selection import train_test_split
3. from sklearn.metrics import accuracy_score
4.
5. iris = load_iris()
6. X = iris.data
7. y = iris.target
8.
9. X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

接下来对数据集进行初步的探索。数据集中有四个特征，分别为 sepal length (cm)、sepal width (cm)、petal length (cm)、petal width (cm)。每个样本的类别（target）用 0、1、2 进行区分。选取其中的三个特征绘制 3D 图：

```
1. import plotly.express as px
```

```
2. import pandas as pd
3.
4. df_iris=pd.DataFrame(iris.data, columns=iris.feature_names)#将 data 加入到数据框中
5. df_iris['target']=iris.target#将 target 加入到数据框中
6. fig = px.scatter_3d(df_iris, x='sepal length (cm)', y='sepal width (cm)', z='petal length (cm)', color='target')
7. fig.update_layout(title='<b>鸢尾花数据集</b>',
8.                     titlefont=dict({'size':28, 'family': 'Courier New'}),
9.                     template='plotly',
10.                    paper_bgcolor='lightgray',
11.                    width=750, height=550,
12.                    )
13. fig.update_layout(scene = dict(
14.                         xaxis = dict(
15.                             backgroundcolor="rgb(200, 200, 230)",
16.                             gridcolor="white",
17.                             showbackground=True,
18.                             zerolinecolor="black",),
19.                         yaxis = dict(
20.                             backgroundcolor="rgb(230, 200,230)",
21.                             gridcolor="white",
22.                             showbackground=True,
23.                             zerolinecolor="black"),
24.                         zaxis = dict(
25.                             backgroundcolor="rgb(230, 230,200)",
26.                             gridcolor="white",
27.                             showbackground=True,
28.                             zerolinecolor="black"),
29.                         ),
30.                     )
31. fig.show()
```



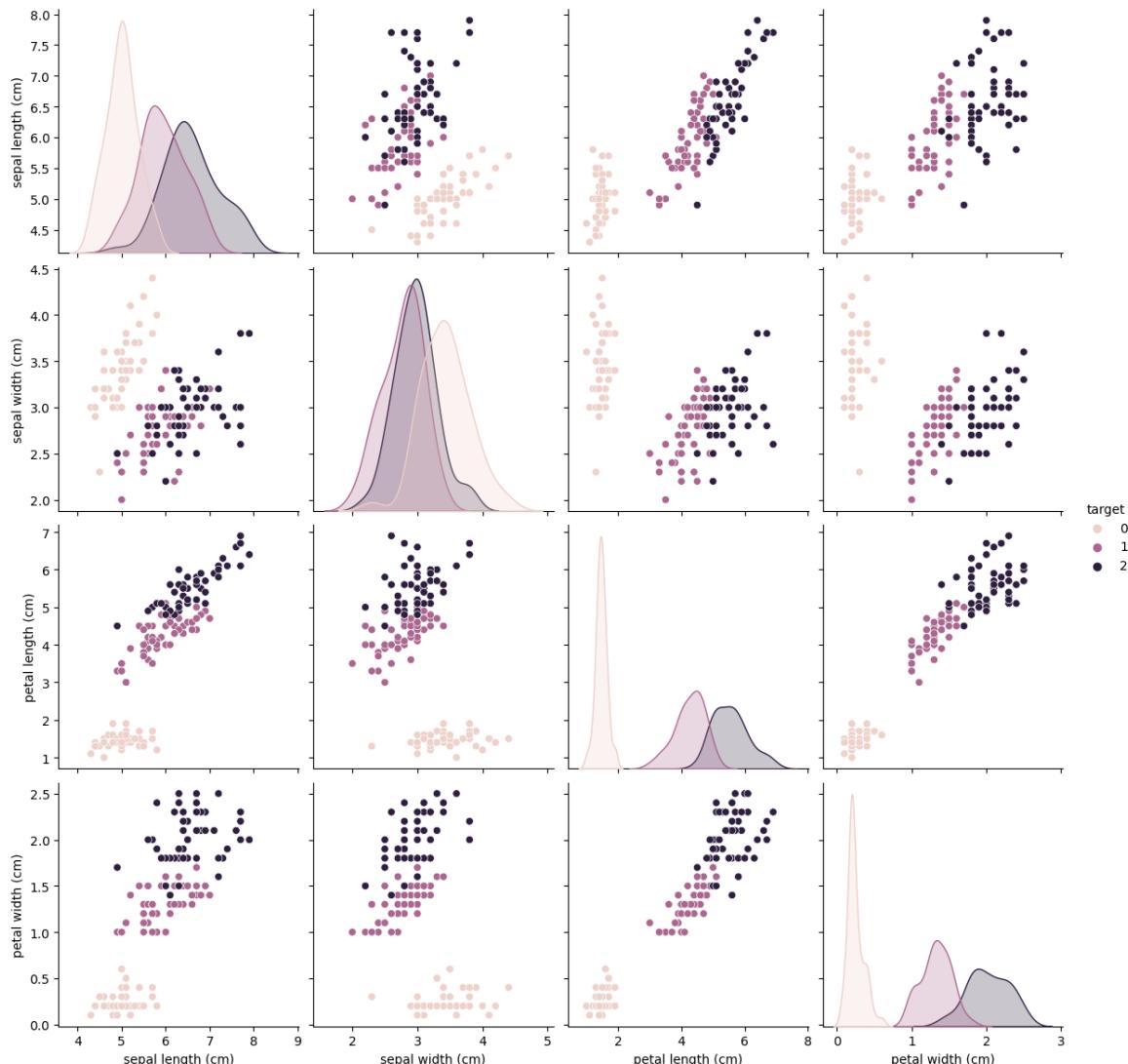
由 3D 图可以看出，3 种类别的鸢尾花具有很好的区分度。由于有四个特征，调用 seaborn 库绘制特征两两之间的相关性图：

```

1. import seaborn as sns
2. import matplotlib.pyplot as plt
3.
4. sns.pairplot(df_iris, hue="target", height=3)

```

结果如下：



下面调用自己编写的 KNN 算法对 iris 数据进行分类：

```

1. knn = KNN(k=3)
2. knn.fit(X_train, y_train)
3.
4. y_pred = knn.predict(X_test)
5.
6. accuracy = accuracy_score(y_test, y_pred)
7. print("Accuracy:", accuracy)

```

输出结果为

Accuracy: 1.0

在测试集上的分类准确率达到 100% !

接下来, 改变 K 值, 观察比较 K 值对分类结果的影响:

K=5 时, 输出结果为

Accuracy: 1.0

K=7 时, 输出结果为

Accuracy: 0.966666

K=11 时, 输出结果为

Accuracy: 1.0

K=81 时, 输出结果为

Accuracy: 0.633333

K=120 时, 输出结果为

Accuracy: 0.3

选择较小的 K 值, 就相当于用较小的领域中的训练实例进行预测, “学习”近似误差会减小, 只有与输入实例较近或相似的训练实例才会对预测结果起作用, 与此同时带来的问题是“学习”的估计误差会增大, 换句话说, K 值的减小就意味着整体模型变得复杂, 容易发生过拟合。在本例中由于三种类别分得较开, 故影响不大;

选择较大的 K 值, 就相当于用较大领域中的训练实例进行预测, 其优点是可以减少学习的估计误差, 但缺点是学习的近似误差会增大。这时候, 与输入实例较远(不相似的)训练实例也会对预测器作用, 使预测发生错误, 且 K 值的增大就意味着整体的模型变得简单;

K=N (N 为训练样本个数, 本例中为 120), 则完全不可取, 因为此时无论输入实例是什么, 都只是简单的预测它属于在训练实例中最多的类, 模型过于简单, 忽略了训练实例中大量有用信息。

2.4 调用 sklearn 库分类

```
1. from sklearn.neighbors import KNeighborsClassifier  
2.  
3. knn_classifier = KNeighborsClassifier(n_neighbors=3)  
4. knn_classifier.fit(X_train, y_train)
```

```
5.  
6. # 预测测试集  
7. y_pred_sk = knn_classifier.predict(X_test)  
8.  
9. # 计算准确度  
10. accuracy = accuracy_score(y_test, y_pred_sk)  
11. print("Accuracy:", accuracy)
```

输出结果为

Accuracy: 1.0

调用 sklearn 中 KNN 算法的分类结果与自己编写的 KNN 算法分类结果完全相同！

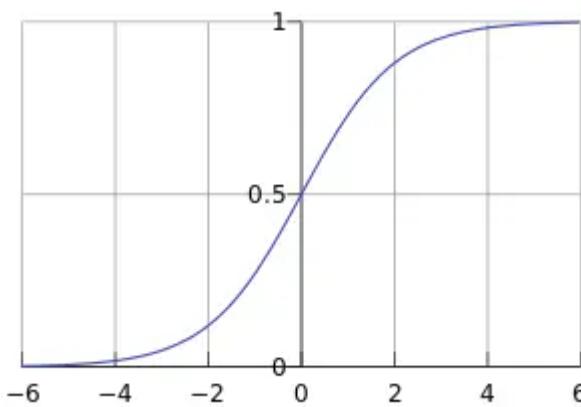
第3章 逻辑回归

3.1 基本概念

逻辑回归，即 Logistic Regression，名字的由来是因为算法流程中使用到了一个关键的 Logistic 函数，该函数是一个比较简单的单调递增函数，表达式为

$$h_{\theta}(x) = \frac{1}{1 + e^{\theta^T x}},$$

该函数也称作 sigmoid 函数，逻辑回归用 sigmoid 函数来计算样本对应的后验概率。



从上图可以看到 sigmoid 函数是一个 s 形的曲线，它的取值在 $[0, 1]$ 之间，在远离 0 的地方函数的值会很快接近 0 或者 1，这个特性对于解决二分类问题十分重要。

对于一个二分类问题：

$$\text{正类: } P(y=1|x; \theta) = h_{\theta}(x),$$

$$\text{负类: } P(y=0|x; \theta) = 1 - h_{\theta}(x),$$

则对所有样本可表示为 $P(y|x; \theta) = (h_{\theta}(x))^y (1 - h_{\theta}(x))^{1-y}$ 。随后采用极大似然估计，对数似然函数为

$$\begin{aligned} \log(M(\theta)) &= \sum_{i=1}^m [y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))] \\ \Rightarrow L(\theta) &= -\log(M(\theta)) = -\sum_{i=1}^m [y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))]. \end{aligned}$$

之后对 L 求导，便可得梯度：

$$\frac{dL}{d\theta} = \frac{dL}{dh} \frac{dh}{d\theta} = \sum_{i=1}^m x^{(i)} (h_{\theta}(x^{(i)}) - y^{(i)}).$$

3.2 梯度下降求解逻辑回归

根据上述对梯度的推导，我们可以使用梯度下降算法求解逻辑回归问题。与梯度下降求解线性回归类似，将求解逻辑回归问题的代码封装成类，以便于使用：

```
1. class LogisticRegression:
2.     def __init__(self, learning_rate=0.01, num_iterations=10000):
3.         self.learning_rate = learning_rate
4.         self.num_iterations = num_iterations
5.         self.weights = None
6.         self.bias = None
7.         self.loss_record = []
8.         self.gradient_record = []
9.         self.accuracy_record = []
10.
11.    def sigmoid(self, z):
12.        return 1 / (1 + np.exp(-z))
13.
14.    def fit(self, X, y):
15.        m, n = X.shape
16.        self.weights = np.zeros(n)
17.        self.bias = 0
18.
19.        for iteration in range(self.num_iterations):
20.            # 计算预测值
21.            z = np.dot(X, self.weights) + self.bias
22.            predictions = self.sigmoid(z)
23.
24.            # 计算损失函数
25.            cost = -1/m * np.sum(y * np.log(predictions) + (1 - y) * np.log(1 - predictions))
26.            self.loss_record.append(cost)
27.
28.            # 计算梯度
29.            dw = 1/m * np.dot(X.T, (predictions - y))
30.
31.            db = 1/m * np.sum((predictions - y))
32.            self.gradient_record.append(np.linalg.norm(dw))
33.
34.            # 计算准确率
35.            accuracy = self.compute_accuracy(predictions.round(), y)
36.            self.accuracy_record.append(accuracy)
37.
38.            # 更新参数
39.            self.weights -= (self.learning_rate * dw)
40.            self.bias -= self.learning_rate * db
```

```
41.  
42.         return self.loss_record, self.gradient_record  
43.  
44.     def compute_accuracy(self, predictions, y):  
45.         accuracy = np.mean(predictions == y)  
46.         return accuracy  
47.  
48.     def predict(self, X):  
49.         z = np.dot(X, self.weights) + self.bias  
50.         predictions = self.sigmoid(z)  
51.         return (predictions > 0.5).astype(int)  
52.  
53.     def plot_accuracy(self):  
54.         plt.plot(range(1, self.num_iterations + 1), self.accuracy_record, marker='o'  
55.                  , color='green')  
56.         plt.title('Accuracy over iterations')  
57.         plt.xlabel('Iterations')  
58.         plt.ylabel('Accuracy')  
59.         plt.show()  
60.     def get_theta(self):  
61.         return self.weights, self.bias
```

构造函数`__init__`用于实例化一个逻辑回归分类器，同时设置学习率、迭代次数，创建用于记录损失函数值、梯度模、正确率的列表，便于之后作图。`sigmoid()`函数依据定义编写。

`fit()`函数用于训练，内部的循环是梯度下降的迭代过程。梯度下降的整体逻辑与之前线性回归部分的梯度下降算法类似，只是损失函数及梯度的计算方式不同。在每一步迭代时，计算准确率并记录，便于之后准确率的作图。

`predict()`函数用于预测。其方法为将梯度下降后的参数与样本值点乘，再传入`sigmoid`函数，即 $\frac{1}{1+e^{-w^T x+b}}$ 。最后将`sigmoid`函数输出的预测值中大于 0.5 的样本判断为正类（1）。

`get_theta()`函数用于返回梯度下降完成后的最优参数值。`compute_accuracy()`及`plot_accuracy()`函数则用于准确率的计算和准确率折线图的绘制。

3.3 逻辑回归的应用：iris 数据集

通过上一章对 iris 数据的探索，可知每条鸢尾花数据包含四个特征，分别是：sepal length (cm)、sepal width (cm)、petal length (cm)、petal width (cm)。总共有三个鸢尾花类别，在 target 列分别于 0、1、2 表示。而逻辑回归是一个二分类算法，即只能处理二分

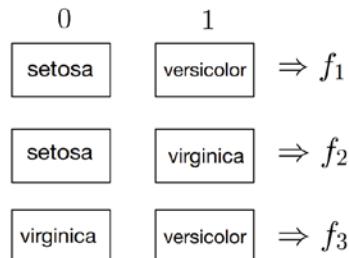
类问题，故需要思考怎么利用二分类处理多分类问题。

在这里，可以使用拆解法，将一个多分类任务拆为若干个二分类任务求解。有三种拆解方法：

$$\begin{cases} \text{一对一} & \text{OvO} \\ \text{一对多} & \text{OvR} \\ \text{多对多} & \text{MvM} \end{cases}$$

由于本次任务是个三分类问题，考虑选择使用 OvO 的方法，训练三个分类器，此方法具备训练时间短的优点。

基本思路是：针对三个类别 setosa、versicolor、virginica，每次训练的分类器将其中一个作为正类，另一个作为负类。对于新的样本，用训练完成的三个分类器进行预测，后采用“投票”的方法确定其类别。



① 分类器 1 (setosa vs. versicolor)

首先划分训练集和测试集，并找出 target 为 0 和 1 的样本：

```

1. X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=66)
2.
3. indices_01 = (y_train == 0) | (y_train == 1) # 找出类别为 0 和 1 的索引
4. indices_01_t = (y_test == 0) | (y_test == 1)
5.
6. X_train_1 = X_train[indices_01]
7. X_test_1 = X_test[indices_01_t]
8.
9. y_train_1 = y_train[indices_01]
10. y_test_1 = y_test[indices_01_t]

```

找出样本后，尝试使用自己编写的逻辑回归算法进行训练和预测

```

1. model_1 = LogisticRegression()
2. ls_1, gr_1 = model_1.fit(X_train_1, y_train_1)
3.
4. print(model_1.predict(X_test_1))
5. print(y_test_1)

```

其输出为

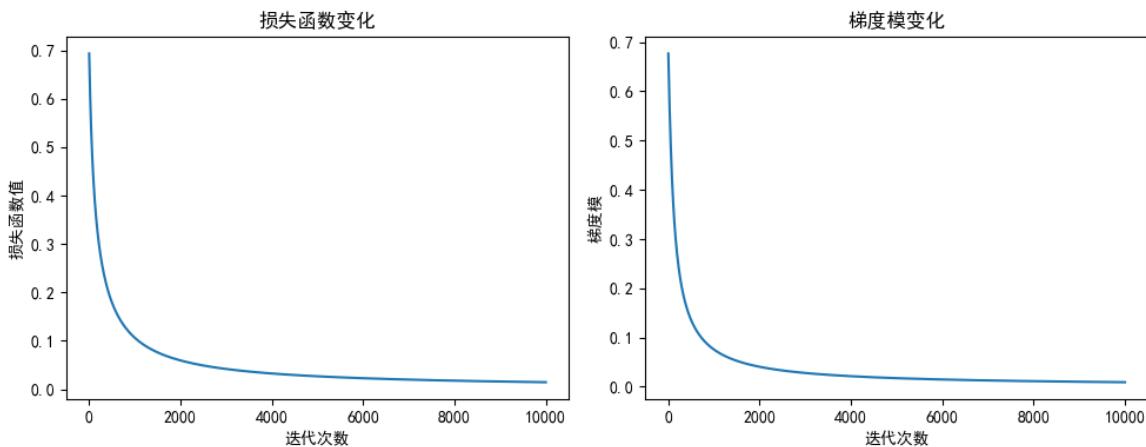
```
[1 1 1 0 1 1 0 0 0 0 0 1 1 0 1 1 1 0 0]  
[1 1 1 0 1 1 0 0 0 0 0 1 1 0 1 1 1 0 0]
```

可以发现在测试集上的分类结果完全正确，模型效果良好。

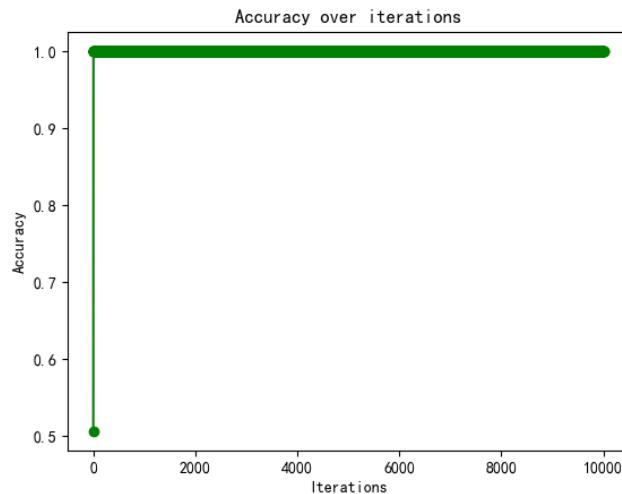
之后绘制损失函数、梯度模随迭代次数变化的折线图

```
1. draw(10000, ls, gr)
```

折线图如下



```
1. model_1.plot_accuracy() # 绘制准确率曲线
```



准确率曲线反映出模型在训练集上的表现良好，经过少量次数的迭代便能在训练集上的准确率达到 100%。

接下来调用 sklearn 库中的逻辑回归算法：

```
1. from sklearn.linear_model import LogisticRegression  
2. model_s1 = LogisticRegression()  
3. model_s1.fit(X_train_1, y_train_1)  
4.  
5. print("预测结果: ", model_1.predict(X_test_1))
```

输出结果为

```
预测结果: [1 1 1 0 1 1 0 0 0 0 0 1 1 0 1 1 1 0 0]
```

与自己编写的逻辑回归算法在测试集上的预测效果、实际的测试集类别完全相同。

② 分类器 2 (setosa vs. virginica)

与分类器 1 的处理方法类似，首先找出 target 为 0 和 2 的样本，将标签转为 0 和 1：

```
1. indices_02 = (y_train == 0) | (y_train == 2) # 找出类别为 0 和 2 的索引
2. indices_02_t = (y_test == 0) | (y_test == 2)
3.
4. X_train_2 = X_train[indices_02]
5. X_test_2 = X_test[indices_02_t]
6.
7. y_train_2 = y_train[indices_02]
8. y_test_2 = y_test[indices_02_t]
9.
10. y_train_2[y_train_2 == 2] = 1
11. y_test_2[y_test_2 == 2] = 1
```

找出样本后，使用自己编写的逻辑回归算法进行训练和预测

```
1. # 实例化和训练逻辑回归模型
2. model_2 = LogisticRegression()
3. model_2.fit(X_train_2, y_train_2)
4.
5. print(model_2.predict(X_test_2))
6. print(y_test_2)
```

其输出为

```
[0 0 0 0 1 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1]
[0 0 0 0 1 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1]
```

可以发现在测试集上的分类结果完全正确，模型效果良好。

③ 分类器 3 (versicolor vs. virginica)

与分类器 1、分类器 2 同理，故详细代码不再在报告中展示。其在测试集上进行预测并与真实类别进行比较的输出结果为

```
[1 1 1 1 1 0 0 0 0 0 1 1 0 0 1 1 0 1 0 0 0]
[1 1 1 1 1 0 0 0 0 0 1 1 0 0 1 1 0 1 0 0 0]
```

可以发现在测试集上的分类结果完全正确，模型效果良好。

在获得了上述三个分类器后，便可实现对鸢尾花数据集的多（三）分类。首先分别用三个分类器对完整的测试集进行预测，预测结果均为 0/1。

```

1. p1 = model_1.predict(X_test)
2. p2 = model_2.predict(X_test)
3. p3 = model_3.predict(X_test)
4.
5. print(p1)

```

以分类器 1 为例，查看其是否对完整的测试集都进行了预测。其预测结果为

```
[1 1 1 0 1 1 0 0 0 1 1 0 1 1 0 1 1 1 0 1 1 1 1 0 0 1 1]
```

在获得三个分类器的分类结果后，最后根据“投票”的规则确定最终类别：

```

1. pre = np.zeros(X_test.shape[0])
2.
3. for i in range(X_test.shape[0]):
4.     sp_dict = {'0': 0, '1': 0, '2': 0}
5.     if p1[i] == 0:
6.         sp_dict['0'] += 1
7.     else:
8.         sp_dict['1'] += 1
9.     if p2[i] == 0:
10.        sp_dict['0'] += 1
11.    else:
12.        sp_dict['2'] += 1
13.    if p3[i] == 0:
14.        sp_dict['2'] += 1
15.    else:
16.        sp_dict['1'] += 1
17. max_key = max(sp_dict, key=sp_dict.get)
18. pre[i] = (max_key)

```

pre 列表用于储存最终的分类结果。对于每个测试集中的样本，sp_dict 字典用于记录投票结果，若在第一个分类器中 label 为 0，则给 0(setosa)投一票(+1)，否则给 1(versicolor)投一票(+1)；若在第二个分类器中 label 为 0，则给 0(setosa)投一票(+1)，否则给 2(virginica)投一票(+1)；若在第三个分类器中 label 为 0，则给 2(virginica)投一票(+1)，否则给 1(versicolor)投一票(+1)。最后根据 sp_dict 字典中的投票结果确定类别。预测结果为：

```
array([1., 1., 1., 0., 1., 0., 0., 0., 0., 2., 2., 2., 0., 2., 2., 0.,
       1., 1., 2., 2., 0., 1., 2., 1., 2., 0., 0., 2., 2.])
```

与实际类别完全相同，预测完全正确！

```
array([1, 1, 1, 0, 1, 1, 0, 0, 0, 2, 2, 2, 0, 2, 2, 0, 1, 1, 2, 2, 0,
       1, 1, 2, 1, 2, 0, 0, 2, 2])
```

第4章 决策树

4.1 基本概念

决策树(decision tree)是一类常见的机器学习方法。一般的，一棵决策树包含一个根结点、若干个内部结点和若干个叶结点；叶结点对应于决策结果，其他每个结点则对应于一个属性测试；每个结点包含的样本集合根据属性测试的结果被划分到子节点中；根结点包含样本全集。从根结点到每个叶结点的路径对应了一个判定测试序列。其基本流程遵循简单而直观的“分而治之”策略：

Algorithm decision tree

Input: 训练集 $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$; 属性集 $A = \{a_1, a_2, \dots, a_d\}$.

```

1 生成结点 node;
2 if D 中样本全属于同一类别 C then
3   将 node 标记为 C 类叶结点; return
4 end if
5 if A = ∅ OR D 中样本在 A 上取值 then
6   将 node 标记为叶结点, 其类别标记为 D 中样本数最多的类; return
7 end if
8 从 A 中选择最优划分属性  $a_*$ ;
9 for  $a_*$  的每一个值  $a_*^v$  do
10   为 node 生成一个分支; 令  $D_v$  表示 D 中在  $a_*$  上取值为  $a_*^v$  的样本子集;
11   if  $D_v$  为空 then
12     将分支结点标记为叶结点, 其类别标记为 D 中样本最多的类; return
13   else
14     以 TreeGenerate( $D_v, A \setminus \{a_*\}$ ) 为分支结点
15   end if
16 end for

```

4.2 决策树的应用: iris

sklearn 库中已经有现成的决策树算法，在加载 iris 数据集，划分训练集与测试集后，调用来对 iris 数据集进行训练

```

1. # 加载 Iris 数据集
2. iris = load_iris()
3. X = iris.data
4. y = iris.target
5.
6. # 数据集划分
7. X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
8.
9. # 初始化决策树分类器
10. clf = DecisionTreeClassifier()
11.
12. # 在训练集上训练模型
13. clf.fit(X_train, y_train)

```

训练完成后，使用 *graphviz* 库来可视化生成的决策树，可视化结果如下



可以看到生成的决策树是一个二叉树。查询相关资料得知，*DecisionTreeClassifier()* 决策树是 CART 决策树，其用基尼(gini)系数作为划分标准。在测试集上进行测试，观察准确率：

```
1. # 在测试集上进行预测
2. y_pred = clf.predict(X_test)
3.
4. # 计算准确度
5. accuracy = accuracy_score(y_test, y_pred)
6. print(f'Accuracy: {accuracy}')
```

输出的准确率为

```
Accuracy: 1.0
```

模型在测试集上的预测完全正确！

4.3 参数探究

首先获取模型的参数

```
1. # 输出决策树的一些参数
2. print(f'Decision Tree Parameters: {clf.get_params()}')
```

```
Decision Tree Parameters: {'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini', 'max_depth': None, 'max_features': None, 'max_leaf_nodes': None, 'min_impurity_decrease': 0.0, 'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'random_state': None, 'splitter': 'best'}
```

查阅相关资料，各参数的含义如下：

- **ccp_alpha:** 定义了用于剪枝的代价复杂性参数的阈值
- **class_weight:** 为不同的类别赋予不同的权重
- **criterion:** 表示划分数据集的准则。可以选择 "gini" (基尼系数) 或 "entropy" (信息增益)
- **max_depth:** 树的最大深度。用于防止树在训练过程中变得过拟合
- **max_features:** 寻找最佳分裂时考虑的特征数量。可以是整数 (考虑的特征数量) 或浮点数 (考虑的特征比例)
- **max_leaf_nodes:** 指定最大叶子节点数目，用于防止过拟合
- **min_impurity_decrease:** 在进行节点分裂时，只有当节点的分裂导致不纯度减少值大于等于指定的阈值时，才进行分裂
- **min_samples_leaf:** 叶子节点必须具有的最小样本数
- **min_samples_split:** 规定了一个节点在分裂之前必须包含的最小样本数
- **min_weight_fraction_leaf:** 叶子节点上的最小样本权重和

接下来使用 iris 数据集，探究决策树模型不同参数对模型的影响。这里定义一个实例化决策树分类器的函数，便于之后参数的调整与复用。

```
1. # 定义决策树分类器
2. def train_decision_tree(max_depth=None, max_leaf_nodes=None, max_features=None, min_
   samples_split=2, min_samples_leaf=1):
3.     clf = DecisionTreeClassifier(max_depth=max_depth, max_leaf_nodes=max_leaf_nodes,
   max_features=max_features, min_samples_split=min_samples_split, min_samples_
   leaf=min_samples_leaf)
4.     clf.fit(X_train, y_train)
5.     return clf
```

① max_depth（最大深度）

获取之前训练好的树模型的深度：

```
1. # 获取训练好的树的深度
2. tree_depth = clf.tree_.max_depth
3. tree_depth
```

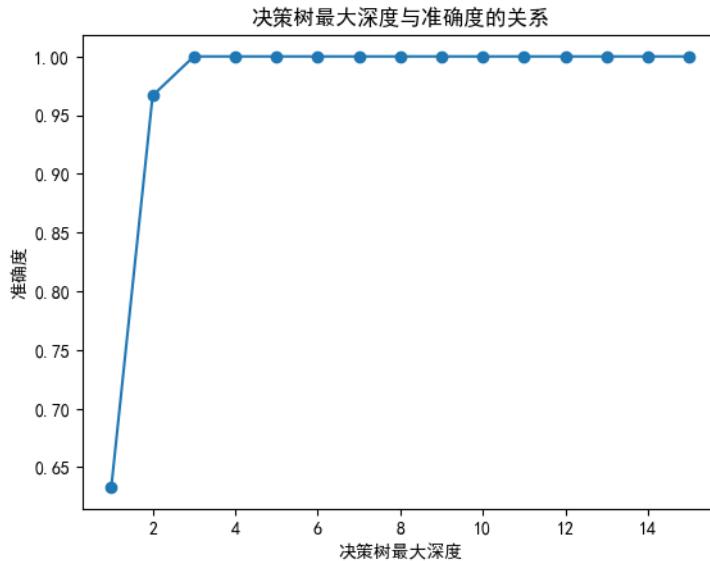
输出为 6，即树模型的深度为 6。

现在尝试改变模型的 max_depth 参数值，观察在测试集上准确率的变化：

```
1. depth_range = range(1, 16)
2. accuracy_scores = []
3.
4. for depth in depth_range:
5.     clf = train_decision_tree(max_depth=depth)
6.     y_pred = clf.predict(X_test)
7.     accuracy = accuracy_score(y_test, y_pred)
8.     accuracy_scores.append(accuracy)
9.
10. # 绘制深度与准确度的关系图
11. plt.plot(depth_range, accuracy_scores, marker='o')
12. plt.title('决策树最大深度与准确度的关系')
13. plt.xlabel('决策树最大深度')
14. plt.ylabel('准确度')
15. plt.show()
```

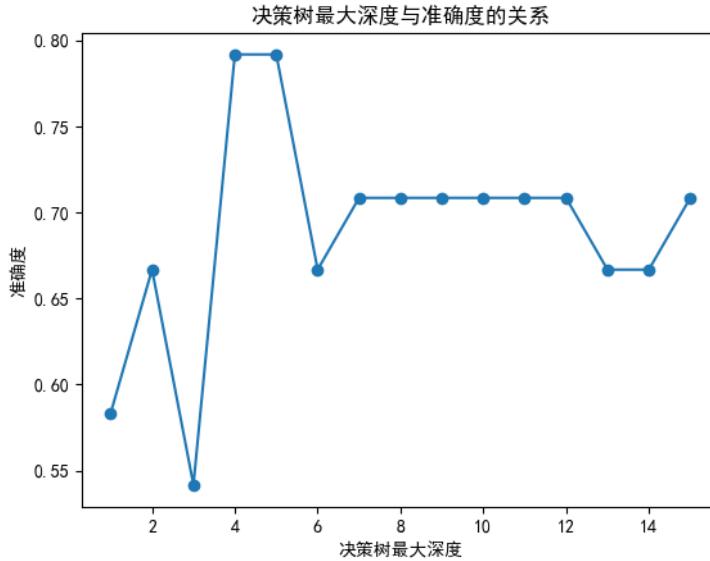
创建一个列表存储需要传入的 max_depth 参数，accuracy_scores 列表用于储存准确度。在循环中，每次传入一个不同的 max_depth 参数值构建决策树模型，训练完成后测试其在测试集上的准确率，并记录。循环完成后绘制决策树最大深度与准确度的关系的折线图。

输出如下：



从中发现，随着决策树最大深度的增加，其准确率上升，其原因是当决策树的深度不足时，模型欠拟合。但根据已有知识，随着深度的增加，模型将出现过拟合现象，反映到测试集则会出现准确率下降的现象，而图中并没有体现。可能是因为 iris 数据集的三种类别易于区分，故再增加最大深度对模型并不产生影响，实际的模型并没有达到这个深度。

更换数据集(LR-testSet2.txt)，再次进行实验：

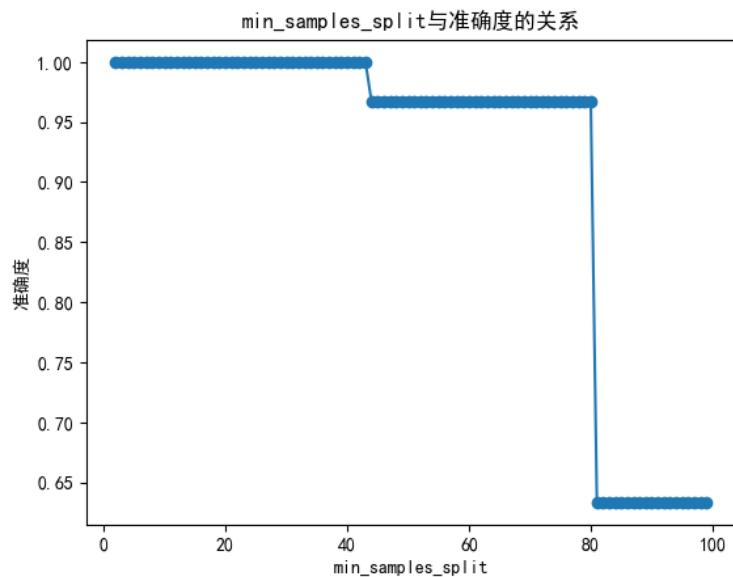


这次的实验结果良好，即深度小时，模型欠拟合；深度大时，模型过拟合。

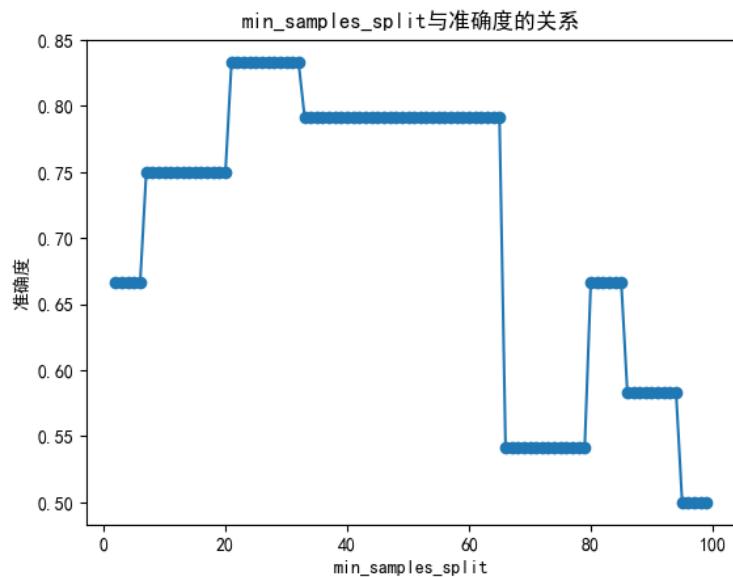
类似地，对 `min_samples_split`（一个节点在分裂之前必须包含的最小样本数）、`min_samples_leaf`（叶子节点必须具有的最小样本数）、`max_features`（寻找最佳分裂时考虑的特征数量）、`max_leaf_nodes`（最大叶子节点数目）四个参数的影响展开探究，在 iris 数据集和 LR-testSet2.txt 上进行训练、测试，并绘制折线图。

② min_samples_split (一个节点在分裂之前必须包含的最小样本数)

在 iris 数据集:



在 LR-testSet2.txt 数据集:

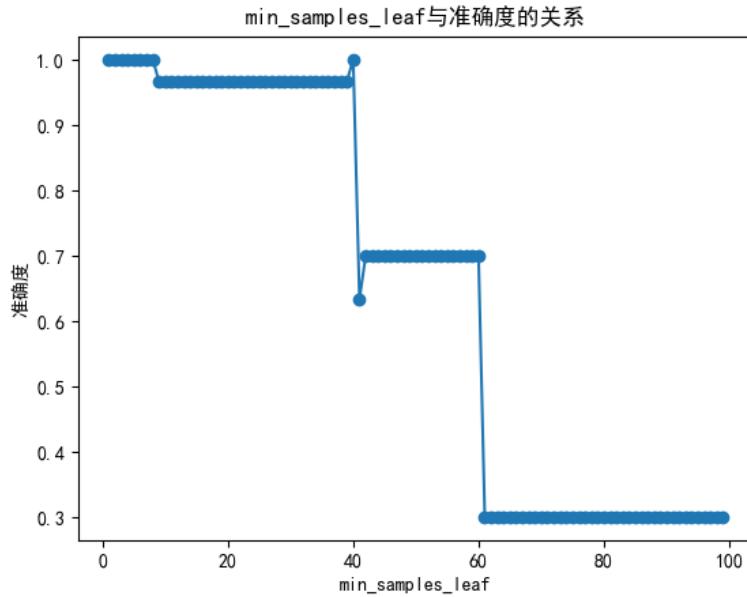


min_samples_split 限定一个结点必须要包含至少 min_samples_split 个训练样本，这个结点才运行被分支，否则分支就不会发生。从准确度折线图可以看出这个参数的数量设置得太小会引起过拟合，设置得太大则会阻止模型学习数据，从而引起欠拟合。

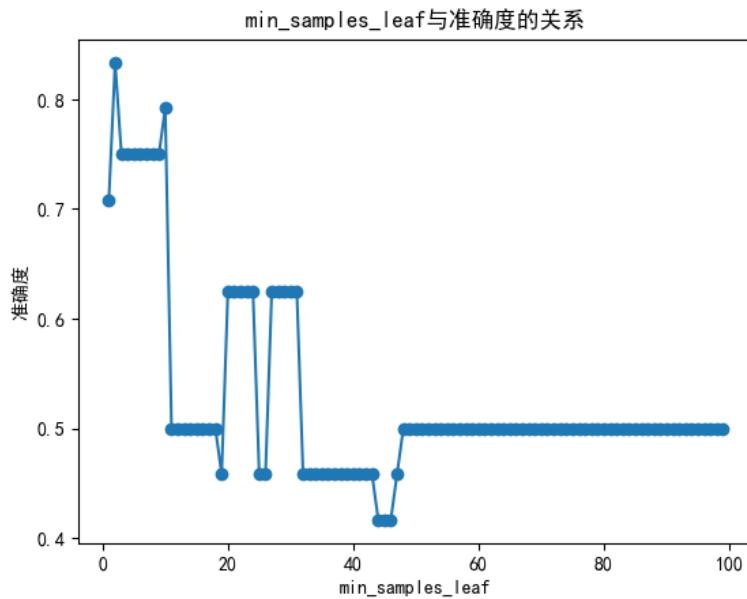
在 iris 数据集上，当 min_samples_split 设置得小时，模型在测试集上的准确率仍达到 1。原因可能与之前的 max_depth 参数类似，与 iris 数据集的特性有关。即 iris 数据集的三种类别易于区分，决策树非常擅于处理这类数据，较少模型的 min_samples_split 参数并未对模型产生实质性的影响，min_samples_split 只是限定了“最小”的结点分支条件。

③ min_samples_leaf (叶子节点必须具有的最小样本数)

在 iris 数据集:



在 LR-testSet2.txt 数据集:



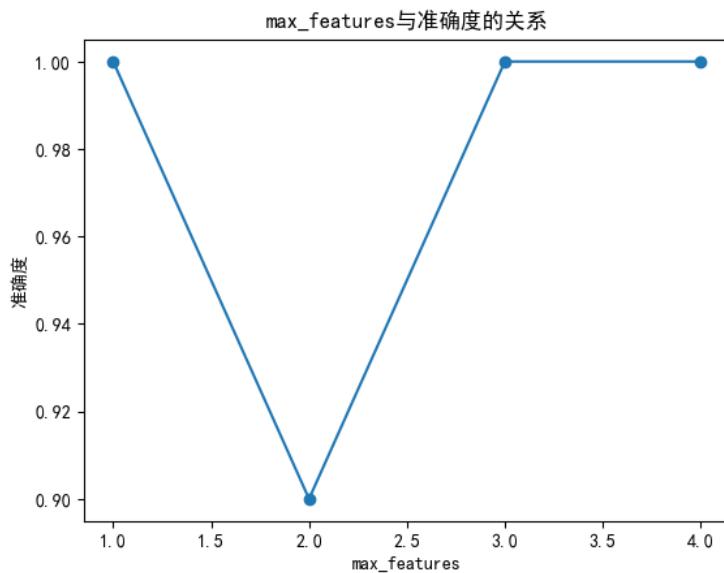
当 min_samples_leaf 的值较小时，决策树会生成更深的树，每个叶子节点上的样本数量可能会很少甚至只有一个。这样的模型会更加灵活，能够更好地拟合训练数据，但也容易过拟合，导致在新数据上的性能下降。

当 min_samples_leaf 的值较大时，决策树会生成较浅的树，叶子节点上的样本数量会增加。这样的模型更加简化，能够更好地泛化到新数据，减少过拟合的风险。然而，较大的 min_samples_leaf 值也可能导致模型欠拟合，无法充分利用训练数据的细节。

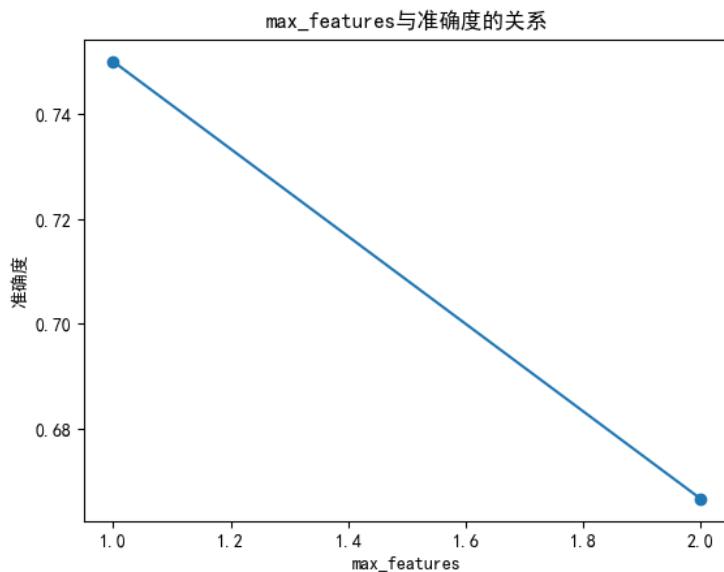
通过调整 min_samples_leaf 的值，可以在模型的偏差和方差之间进行权衡。

④ max_features (寻找最佳分裂时考虑的特征数量)

在 iris 数据集:



在 LR-testSet2.txt 数据集:



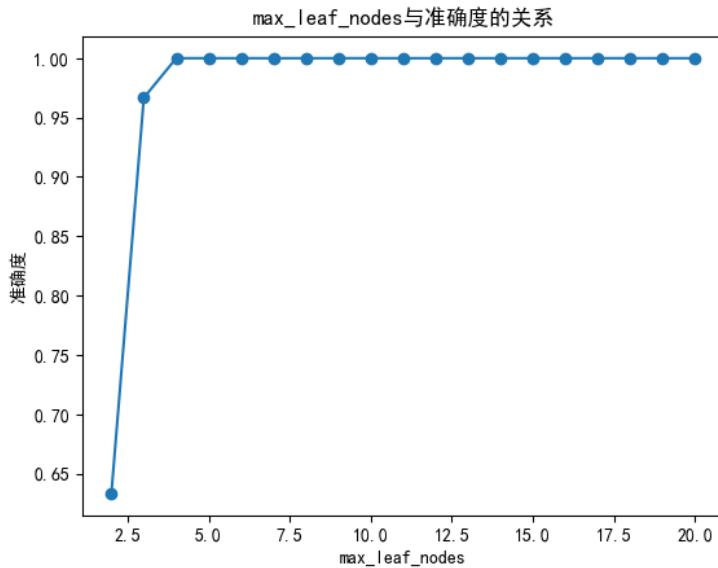
一般来说，当 max_features 的值较小时，决策树会在每个节点上只考虑少量的特征进行拆分。这样的模型会更加简化，减少了特征空间的搜索范围，有助于降低模型的复杂性和计算成本。然而，较小的 max_features 值也可能导致模型无法捕捉到数据中的重要特征，从而降低模型的表达能力。

当 max_features 的值较大时，决策树会在每个节点上考虑更多的特征进行拆分。这样的模型具有更高的灵活性，能够更好地拟合训练数据，但也容易过拟合。

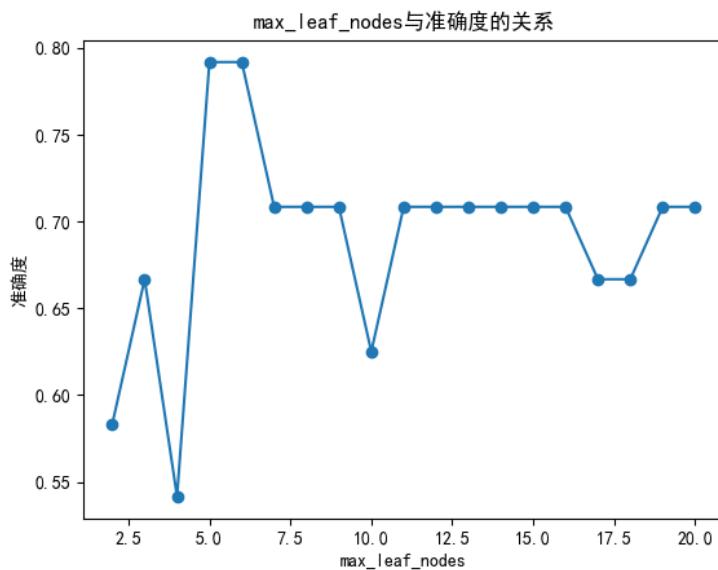
在 iris 数据集上，分别设置 max_features 为 1、2、3、4，在设置为 3 时准确率为 0.9，其余情况下均为 1。

⑤ max_leaf_nodes (最大叶子节点数目)

在 iris 数据集:



在 LR-testSet2.txt 数据集:



当 `max_leaf_nodes` 的值较小时, 决策树会生成较浅的树, 叶子节点的数量会减少。这样的模型更加简化, 具有较低的复杂性, 有助于减少过拟合的风险。然而, 较小的 `max_leaf_nodes` 值也可能导致模型欠拟合, 无法充分利用训练数据的细节。

当 `max_leaf_nodes` 的值较大时, 决策树会生成更深的树, 叶子节点的数量会增加。这样的模型具有更高的灵活性, 能够更好地拟合训练数据, 但也容易过拟合, 特别是当训练数据较少时。

LR-testSet2.txt 数据集体现了上述规律。而在 iris 数据集上的现象可能与之前类似, 即继续增大 `max_leaf_nodes` 并不会对模型产生实质性的改变, 只是规定了“最大”。

4.4 完成非线性分类

在之前的 LR-testSet2.txt 数据集上，默认模型参数
输出结果为：

```
Accuracy: 0.708333333333334
```

接下来调整参数，令最大深度的参数为 4，重新训练并在测试集上测试

```
1. classifier_1 = DecisionTreeClassifier(max_depth=4)
2. classifier_1.fit(X_train, y_train)
3.
4. y_pred = classifier_1.predict(X_test)
5.
6. # 计算准确度
7. accuracy = accuracy_score(y_test, y_pred)
8. print(f'Accuracy: {accuracy}')
```

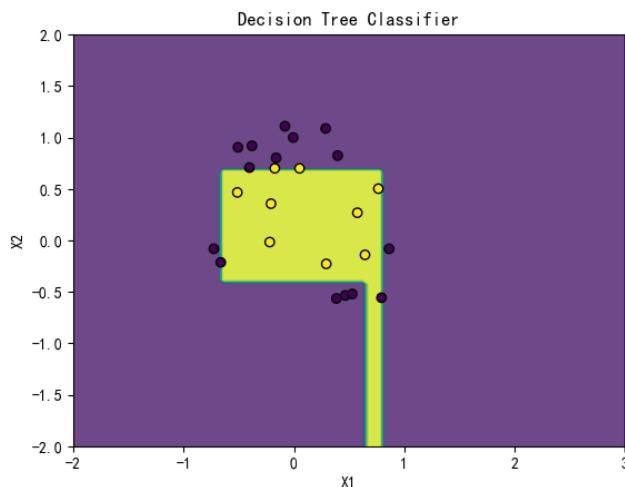
输出结果为：

```
Accuracy: 0.7916666666666666
```

模型在测试集上准确率提高，之前的模型过拟合。

下面尝试可视化决策边界

```
1. xx, yy = np.meshgrid(np.linspace(-2, 3, 100),
2.                      np.linspace(-2, 2, 100))
3. Z = classifier_1.predict(np.c_[xx.ravel(), yy.ravel()])
4. Z = Z.reshape(xx.shape)
5.
6. plt.contourf(xx, yy, Z, alpha=0.8)
7. plt.scatter(X_test[:, 0], X_test[:, 1], c=y_pred, edgecolors='k')
8. plt.xlabel('X1')
9. plt.ylabel('X2')
10. plt.title('Decision Tree Classifier')
11. plt.show()
```



尝试生成一组数据，并用决策树分类：

```
1. x1, y1 = datasets.make_circles(n_samples=2000, factor=0.5, noise=0.05, random_state=1017)
2. x2, y2 = datasets.make_blobs(n_samples=1000, centers=[[1.2,1.2]], cluster_std=[[.1]], random_state=1017)
3. y2 = np.full_like(y2, 2)
4. # 将向量纵向拼接
5. x_all = np.vstack((x1, x2))
6. y_all = np.concatenate((y1, y2))
```

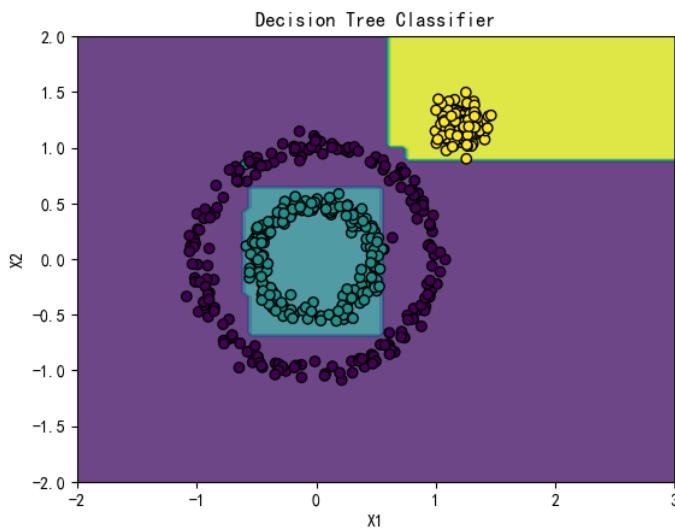
数据集划分后用决策树模型训练并在测试集上预测

```
1. # 数据集划分
2. X_train, X_test, y_train, y_test = train_test_split(x_all, y_all, test_size=0.2, random_state=1017)
3.
4. cclf = DecisionTreeClassifier()
5. cclf.fit(X_train, y_train)
6.
7. y_pred = cclf.predict(X_test)
8.
9. # 计算准确度
10. accuracy = accuracy_score(y_test, y_pred)
11. print(f'Accuracy: {accuracy}')
```

输出为：

```
Accuracy: 0.9966666666666667
```

可视化决策边界为

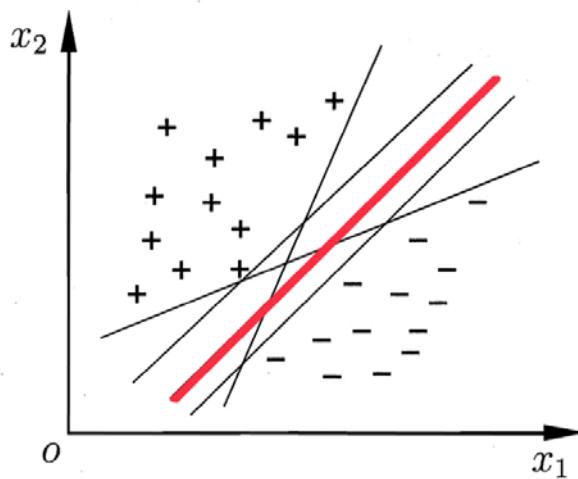


可以发现，决策树对数据的非线性分类本质上是通过线性分类组合实现的，通过线性决策边界的组合实现对数据的分类。

第5章 SVM

5.1 基本概念

给定训练样本集 $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_m, y_m)\}$, $y_i \in \{-1, +1\}$, 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。能将训练样本分开的超平面可能有很多, 直观上看, 应该去找位于两类训练样本“正中间”的划分超平面, 因为该划分超平面对训练样本局部扰动的“容忍”性最好。换言之, 这个划分超平面所产生的分类结果是最鲁棒的。



在样本空间中, 划分超平面可通过如下线性方程来描述:

$$\mathbf{w}^T \mathbf{x} + b = 0,$$

其中 $\mathbf{w} = (w_1; w_2; \dots; w_d)$ 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。假设超平面 (\mathbf{w}, b) 能将训练样本正确分类, 即对于 $(\mathbf{x}_i, y_i) \in D$, 若 $y_i = +1$, 则有 $\mathbf{w}^T \mathbf{x}_i + b \geq 0$; 若 $y_i = -1$, 则有 $\mathbf{w}^T \mathbf{x}_i + b \leq 0$ 。距离超平面最近的几个样本使 $\mathbf{w}^T \mathbf{x}_i + b = 0$ 成立, 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

$$\gamma = \frac{2}{\|\mathbf{w}\|},$$

它被称为“间隔”(margin)。

欲找到具有“最大间隔”的划分超平面, 也就是要找到能满足约束的参数 \mathbf{w} 和 b ,

使得 γ 最大，即

$$\begin{aligned} \max_{\mathbf{w}, b} \quad & \frac{2}{\|\mathbf{w}\|} \\ \text{s.t.} \quad & y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1, \quad i = 1, 2, \dots, m. \end{aligned}$$

显然，为了最大化间隔，仅需最大化 $\|\mathbf{w}\|^{-1}$ ，这等价于最小化 $\|\mathbf{w}\|^2$ ，故可表述为

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & \frac{1}{2} \|\mathbf{w}\|^2 \\ \text{s.t.} \quad & y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1, \quad i = 1, 2, \dots, m. \end{aligned}$$

这就是支持向量机的基本型。该问题本身是一个凸二次规划问题，可通过已有的优化计算包求解，也可以应用拉格朗日乘子法构造拉格朗日对偶函数，再通过求解其对偶问题得到原始问题的最优解。

然而在现实任务中，原始样本空间也许并不存在一个能正确划分两类样本的超平面。对这样的问题，可将样本从原始空间映射到更高维的特征空间，使得样本在这个特征空间内线性可分。

令 $\phi(\mathbf{x})$ 表示将 \mathbf{x} 映射后的特征向量，于是，在特征空间中划分超平面所对应的模型可表示为

$$f(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + b.$$

类似地，可将问题表述为

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & \frac{1}{2} \|\mathbf{w}\|^2 \\ \text{s.t.} \quad & y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b) \geq 1, \quad i = 1, 2, \dots, m. \end{aligned}$$

在对偶问题的求解过程中，由于特征空间维数可能很高，甚至可能是无穷维，因此直接计算 $\phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$ 通常是困难的。为此，使用“核函数”，使得

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j).$$

常用核函数：

名称	表达式
线性核	$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$
多项式核	$\kappa(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T \mathbf{x}_j)^d$
高斯核	$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\ \mathbf{x}_i - \mathbf{x}_j\ ^2}{2\sigma^2}\right)$
Sigmoid 核	$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\beta \mathbf{x}_i^T \mathbf{x}_j + \theta)$

5.2 SVM 的应用: iris

加载 iris 数据集并划分训练集、测试集后，使用 *StandardScaler* 进行特征缩放：

```
1. # 特征缩放, 使用 StandardScaler
2. scaler = StandardScaler()
3. X_train_scaled = scaler.fit_transform(X_train)
4. X_test_scaled = scaler.transform(X_test)
```

特征缩放后，初始化 SVM 分类器，在训练集上训练模型，并在测试集上测试，计算准确度

```
1. svm_classifier = SVC(kernel='linear', C=1.0)
2. svm_classifier.fit(X_train_scaled, y_train)
3.
4. y_pred = svm_classifier.predict(X_test_scaled)
5.
6. # 计算准确度
7. accuracy = accuracy_score(y_test, y_pred)
8. print(f'使用线性核函数 Accuracy: {accuracy * 100:.2f}%')
```

输出为：

```
使用线性核函数 Accuracy: 96.67%
```

使用线性核函数，SVM 具有良好的分类能力，在测试集上的准确率高达 96.67%。这说明在原始的特征空间（4 维），基本完全线性可分。

下面尝试使用其他核函数进行训练，准确度分别为：

◆ 高斯核（径向基核、RBF 核）

```
使用径向基核函数 Accuracy: 100.00%
```

◆ 多项式核

```
使用多项式核函数 Accuracy: 96.67%
```

◆ Sigmoid 核

```
使用 sigmoid 核函数 Accuracy: 90.00%
```

在这些核函数中，高斯核函数在测试集上的测试效果最好，达到了 100% 的分类准确率！由于 iris 数据集中有 4 个特征（4 维），故进行可视化较为困难。

5.3 SVM 的应用：LR-testSet2.txt

与之前类似，读取数据并划分数据集后，分别使用线性核、多项式核、高斯核、Sigmoid 核，观察比较。

◆ 线性核

使用线性核函数 Accuracy: 37.50%

◆ 高斯核（径向基核、RBF 核）

使用径向基核函数 Accuracy: 83.33%

◆ 多项式核

使用多项式核函数 Accuracy: 70.83%

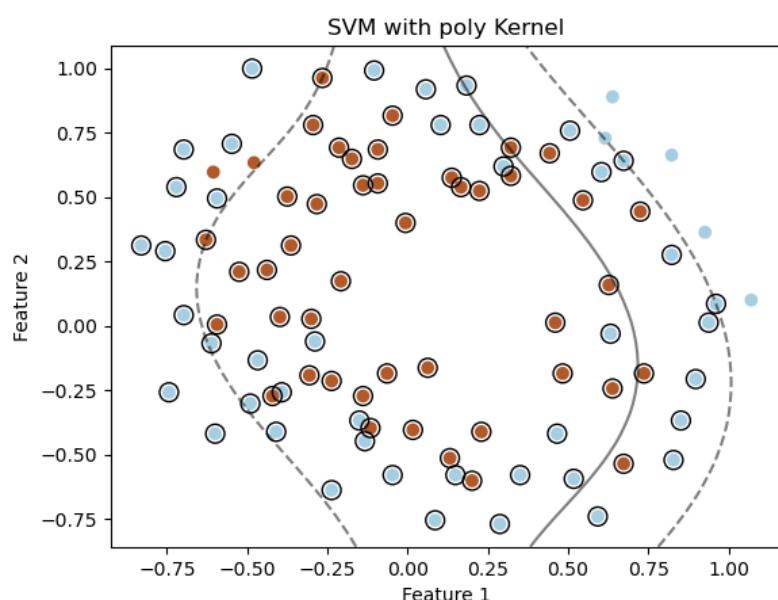
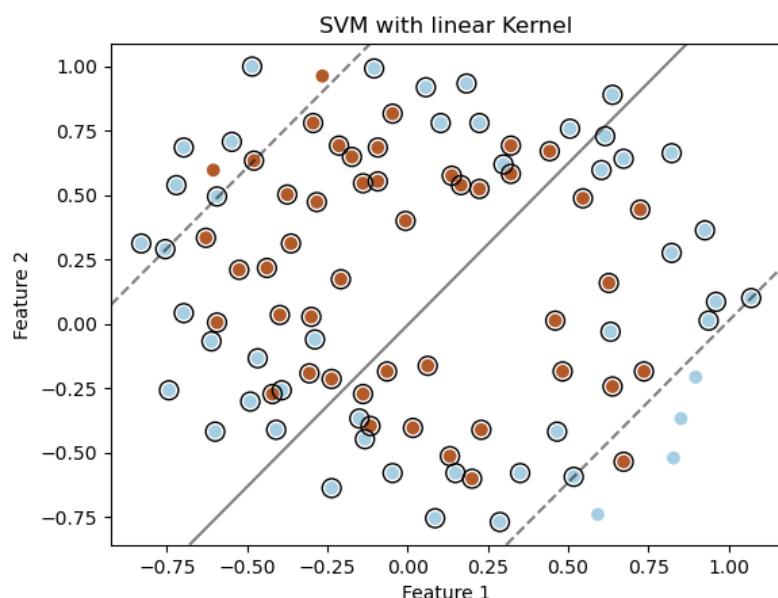
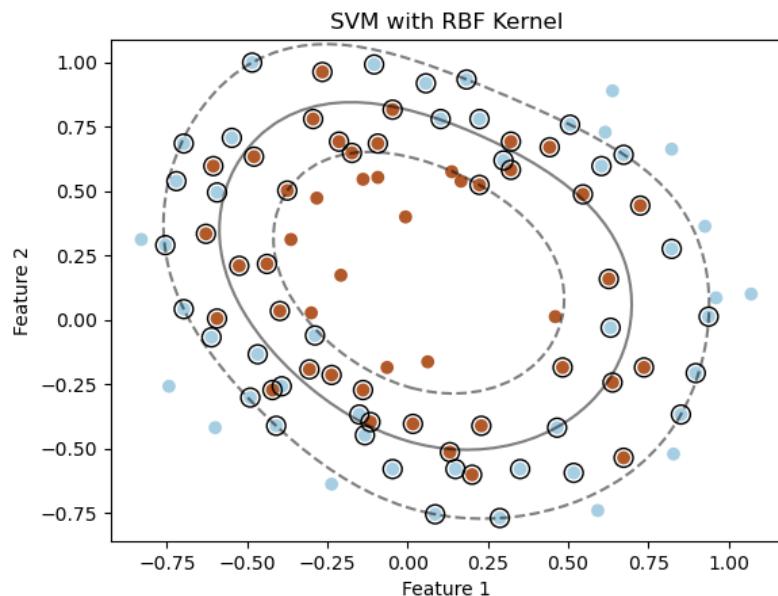
◆ Sigmoid 核

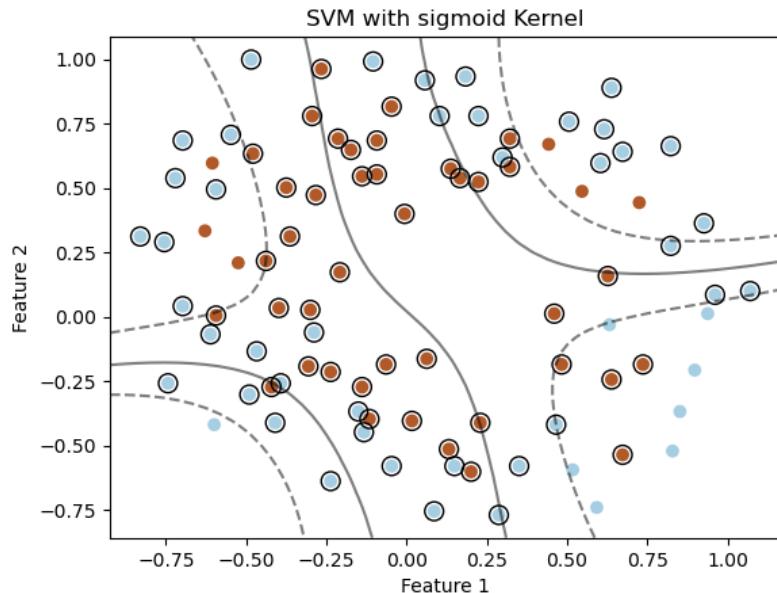
使用 sigmoid 核函数 Accuracy: 37.50%

在这些核函数中，径向基核函数在测试集上的表现最好，下面进行可视化。以高斯核函数为例，可视化的代码框架基本如下：

```
1. plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.Paired)
2.
3. # 绘制支持向量
4. plt.scatter(svm_classifier_rbf.support_vectors_[:, 0], svm_classifier_rbf.support_ve
   ctors_[:, 1], s=100, facecolors='none', edgecolors='k')
5.
6. # 绘制决策边界
7. ax = plt.gca()
8. xlim = ax.get_xlim()
9. ylim = ax.get_ylim()
10.
11. # 创建网格来评估模型
12. xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 50), np.linspace(ylim[0], ylim[1]
   , 50))
13. Z = svm_classifier_rbf.decision_function(np.c_[xx.ravel(), yy.ravel()])
14. Z = Z.reshape(xx.shape)
15.
16. # 绘制等高线图
17. plt.contour(xx, yy, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-',
   '-'])
18.
19. plt.title('SVM with RBF Kernel')
20. plt.xlabel('Feature 1')
21. plt.ylabel('Feature 2')
22. plt.show()
```

对于不同的核函数，绘制决策边界等高线图，突出显示支持向量，以便于进一步的分析讨论。





从图中也可以看出，高斯核较适合于此数据集的分类。

查阅相关资料，对不同核函数的适用情况总结如下：

核函数	适用情况
线性核函数	<ul style="list-style-type: none"> 1. 适用于特征空间中线性可分的数据集 2. 当数据集的特征在原始空间中可以通过一个超平面完全分开时，线性核函数效果较好
多项式核函数	<ul style="list-style-type: none"> 1. 适用于数据集中存在多项式关系的情况 2. 多项式核函数的一个参数是多项式的度数，可以通过调整这个参数来控制多项式的复杂度
高斯核函数	<ul style="list-style-type: none"> 1. 适用于大多数情况，尤其是在数据集中存在复杂的非线性关系时 2. 高斯核函数的一个参数是带宽（bandwidth），带宽越小，影响范围越小，决策边界更加复杂
Sigmoid 核函数	<ul style="list-style-type: none"> 1. 适用于某些特殊情况，但在实践中往往不常使用 2. 这种核函数在一些特定的问题中可能表现良好，但在大多数情况下，高斯核函数或多项式核函数更常用

第6章 贝叶斯分类

6.1 基本概念

假设有 N 种可能的类别标记，即 $\mathcal{Y}=\{c_1, c_2, \dots, c_N\}$ ， λ_{ij} 是将一个真实标记为 c_j 的样本误分类为 c_i 所产生的损失。基于后验概率 $P(c_i|\mathbf{x})$ 可获得将样本 \mathbf{x} 分类为 c_i 所产生的期望损失，即在样本 \mathbf{x} 上的“条件风险”

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

我们的任务是寻找一个判定准则 $h: \mathcal{X} \mapsto \mathcal{Y}$ 以最小化总体风险

$$R(h) = \mathbb{E}_{\mathbf{x}}[R(h(\mathbf{x})|\mathbf{x})].$$

显然，对每个样本 \mathbf{x} ，若 h 能最小化条件风险 $R(h(\mathbf{x})|\mathbf{x})$ ，则总体风险 $R(h)$ 也将被最小化。这就产生了贝叶斯判定准则：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 $R(c|\mathbf{x})$ 最小的类别标记，即

$$h^*(\mathbf{x}) = \arg \min_{c \in \mathcal{Y}} R(c|\mathbf{x}),$$

此时， h^* 称为贝叶斯最优分类器。

具体来说，若目标是最小化分类错误率，则误判损失 λ_{ij} 可写为

$$\lambda_{ij} = \begin{cases} 0, & \text{if } i = j; \\ 1, & \text{otherwise,} \end{cases}$$

此时条件风险

$$R(c|\mathbf{x}) = 1 - P(c|\mathbf{x}),$$

于是，最小化分类错误率的贝叶斯最优分类器为

$$h^*(\mathbf{x}) = \arg \max_{c \in \mathcal{Y}} P(c|\mathbf{x}),$$

即对每个样本 \mathbf{x} ，选择能使后验概率 $P(c|\mathbf{x})$ 最大的类别标记。

基于贝叶斯定理， $P(c|\mathbf{x})$ 可写为

$$P(c|\mathbf{x}) = \frac{P(c)P(\mathbf{x}|c)}{P(\mathbf{x})},$$

$P(c)$ 是类“先验概率”， $P(\mathbf{x}|c)$ 是条件概率，或称为“似然”。

6.2 贝叶斯的应用: iris

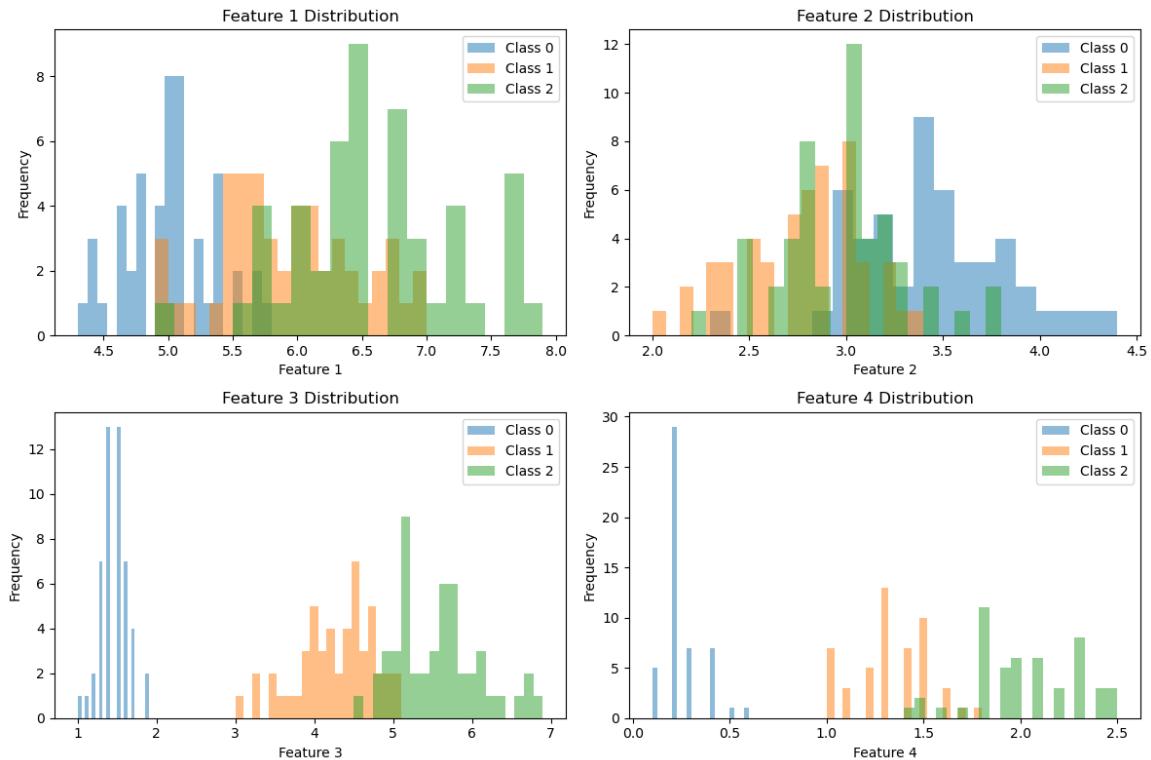
加载 iris 数据集后，可视化每个类别在每个特征上的分布

```

1. plt.figure(figsize=(12, 8))
2.
3. for i in range(X.shape[1]):
4.     plt.subplot(2, 2, i + 1)
5.     for j in range(len(np.unique(y))):
6.         plt.hist(X[y == j, i], bins=20, label=f'Class {j}', alpha=0.5)
7.     plt.title(f'Feature {i + 1} Distribution')
8.     plt.xlabel(f'Feature {i + 1}')
9.     plt.ylabel('Frequency')
10.    plt.legend()
11.
12.    plt.tight_layout()
13. plt.show()

```

可视化图形如下：



基于贝叶斯公式来估计后验概率 $P(c|\mathbf{x})$ 的主要困难在于：类条件概率 $P(\mathbf{x}|c)$ 是所有属性上的联合概率，难以从有限的训练样本直接估计而得。为避开这个障碍，朴素贝叶斯分类器采用了“属性条件独立性假设”：对已知类别，假设所有属性相互独立

$$P(c|\mathbf{x}) = \frac{P(c)P(\mathbf{x}|c)}{P(\mathbf{x})} = \frac{P(c)}{P(\mathbf{x})} \prod_{i=1}^d P(x_i|c).$$

sklearn 库中已经有现成的朴素贝叶斯分类算法，调用实现对 iris 数据集的分类。

```
1. # 划分数据集为训练集和测试集
2. X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1017)
3.
4. model = GaussianNB()
5. model.fit(X_train, y_train)
6.
7. # 预测测试集
8. y_pred = model.predict(X_test)
9.
10. # 计算准确度
11. accuracy = accuracy_score(y_test, y_pred)
12. print(f'Accuracy: {accuracy * 100:.2f}%')
```

输出为：

```
Accuracy: 90.00%
```

可以看到贝叶斯分类器的分类效果较好。由于有四个维度，故难以对分类结果进行可视化。

接下来考虑将特征使用主成分分析降成两个维度，再进行分类

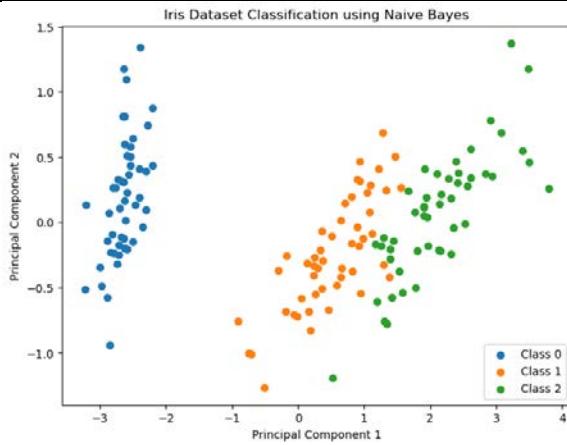
```
1. # 特征降维到二维
2. pca = PCA(n_components=2)
3. X_reduced = pca.fit_transform(X)
```

降维后使用贝叶斯模型在训练集上进行训练，并在测试集上测试，计算准确率的输出为：

```
Accuracy: 86.67%
```

可以发现，主成分分析降维虽然最大限度保留了特征的原始信息，可还是会造成员耗，从而使得贝叶斯分类的准确率下降。可视化分类结果：

```
1. plt.figure(figsize=(8, 6))
2.
3. for i in range(3):
4.     plt.scatter(X_reduced[y == i, 0], X_reduced[y == i, 1], label=f'Class {i}')
5.
6. plt.title('Iris Dataset Classification using Naive Bayes')
7. plt.xlabel('Principal Component 1')
8. plt.ylabel('Principal Component 2')
9. plt.legend()
10. plt.show()
```



类似地，将特征使用主成分分析降成三个维度，再进行分类。贝叶斯分类器在测试集上的准确率为：

Accuracy: 86.67%

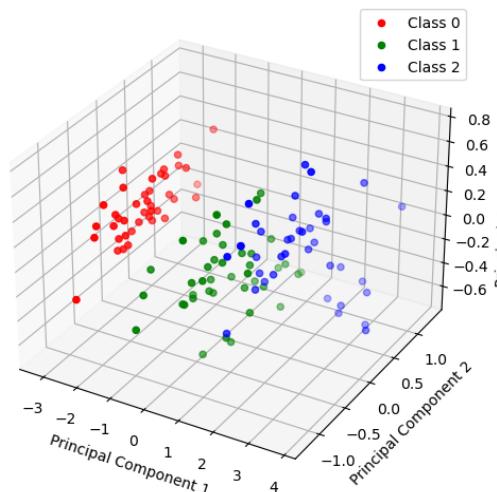
在测试集上的准确率与降至两维的情形相同。进行可视化：

```

1. fig = plt.figure(1, figsize=(8, 6))
2. ax = fig.add_subplot(111, projection='3d')
3.
4. colors = ['red', 'green', 'blue']
5. for i in range(3):
6.     indices = y_train == i
7.     ax.scatter(X_train[indices, 0], X_train[indices, 1], X_train[indices, 2], c=colors[i], label=f'Class {i}')
8.
9. ax.set_xlabel('Principal Component 1')
10. ax.set_ylabel('Principal Component 2')
11. ax.set_zlabel('Principal Component 3')
12. ax.set_title('Training Set - Bayesian Classification of Iris Dataset (3D)')
13. ax.legend()
14. plt.show()

```

Training Set - Bayesian Classification of Iris Dataset (3D)



第 7 章 聚类

7.1 K-means

K-means 算法属于一种原型聚类算法。原型聚类亦称“基于原型的聚类”，此类算法假设聚类结构能通过一组原型刻画，在现实聚类任务中极为常用。通常情形下，算法先对原型进行初始化，然后对原型进行迭代更新求解。

给定样本集 $D = \{x_1, x_2, \dots, x_m\}$, K 均值算法针对聚类所得簇划分 $\mathcal{C} = \{C_1, C_2, \dots, C_k\}$ 最小化平方误差

$$E = \sum_{i=1}^k \sum_{x \in C_i} \|x - \mu_i\|_2^2,$$

其中 $\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$ 是簇 C_i 的均值向量。最小化平方误差不容易，找到它的最优解需

考察样本集 D 所有可能的簇划分，这是一个 NP 难问题。因此，K 均值算法采用了贪心策略，通过迭代优化来近似求解式。

输入: 样本集 $D = \{x_1, x_2, \dots, x_m\}$;
聚类簇数 k .

过程:

```

1: 从  $D$  中随机选择  $k$  个样本作为初始均值向量  $\{\mu_1, \mu_2, \dots, \mu_k\}$ 
2: repeat
3:   令  $C_i = \emptyset$  ( $1 \leq i \leq k$ )
4:   for  $j = 1, 2, \dots, m$  do
5:     计算样本  $x_j$  与各均值向量  $\mu_i$  ( $1 \leq i \leq k$ ) 的距离:  $d_{ji} = \|x_j - \mu_i\|_2$ ;
6:     根据距离最近的均值向量确定  $x_j$  的簇标记:  $\lambda_j = \arg \min_{i \in \{1, 2, \dots, k\}} d_{ji}$ ;
7:     将样本  $x_j$  划入相应的簇:  $C_{\lambda_j} = C_{\lambda_j} \cup \{x_j\}$ ;
8:   end for
9:   for  $i = 1, 2, \dots, k$  do
10:    计算新均值向量:  $\mu'_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$ ;
11:    if  $\mu'_i \neq \mu_i$  then
12:      将当前均值向量  $\mu_i$  更新为  $\mu'_i$ 
13:    else
14:      保持当前均值向量不变
15:    end if
16:  end for
17: until 当前均值向量均未更新

```

输出: 簇划分 $\mathcal{C} = \{C_1, C_2, \dots, C_k\}$

根据上述的 K-means 算法逻辑，编写代码如下：

```

1. def k_means(data, k, max_iterations=100):
2.     # 初始化聚类中心
3.     centroids = data[np.random.choice(range(len(data)), k, replace=False)]
4.
5.     for _ in range(max_iterations):
6.         # 计算每个样本到各个聚类中心的距离
7.         distances = np.linalg.norm(data - centroids[:, np.newaxis], axis=2)
8.
9.         # 将每个样本分配到距离最近的聚类中心
10.        labels = np.argmin(distances, axis=0)
11.
12.        # 更新聚类中心
13.        new_centroids = np.array([data[labels == i].mean(axis=0) for i in range(k)])
14.
15.        # 计算损失函数值
16.        loss = np.sum(np.square(np.linalg.norm(data - new_centroids[labels])))
17.
18.        # 如果聚类中心不再发生变化，停止迭代
19.        if np.all(centroids == new_centroids):
20.            break
21.
22.        centroids = new_centroids
23.
24.    return labels, centroids, loss

```

在代码中，首先初始化聚类中心，随机选取 k 个样本作为均值向量 $\{\mu_1, \mu_2, \dots, \mu_k\}$ 。之后进入迭代过程，计算每个样本到聚类中心的距离后进行分配，分配后更新聚类中心并计算损失函数值，直到聚类中心不再发生变化停止迭代。

接下来使用编写的 k-means 算法对 kmeans.txt 数据进行聚类。由于 kmeans.txt 数据中的两个特征大小相似，故此处无须做数据归一化或数据标准化处理。在聚类前，需确定超参数 k 的值。 k 值的选取对 K-means 影响很大，这也是 K-means 最大的缺点，常见的方法有：手肘法、Gap statistic 方法。这里选择使用手肘法，尝试使用不同的 k 值进行聚类，计算平方误差，并绘制“碎石图”：

```

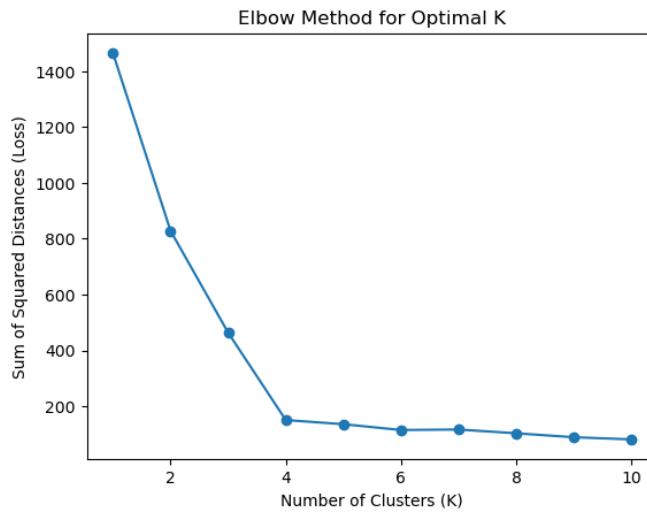
1. # 尝试不同的聚类数并计算损失函数值
2. losses = []
3. k_values = range(1, 11) # 尝试聚类数从 1 到 10
4. for k in k_values:
5.     label, center, loss_value = k_means(data_matrix, k)
6.     losses.append(loss_value)
7.

```

```

8. # 绘制碎石图
9. plt.plot(k_values, losses, marker='o')
10. plt.title('Elbow Method for Optimal K')
11. plt.xlabel('Number of Clusters (K)')
12. plt.ylabel('Sum of Squared Distances (Loss)')
13. plt.show()

```



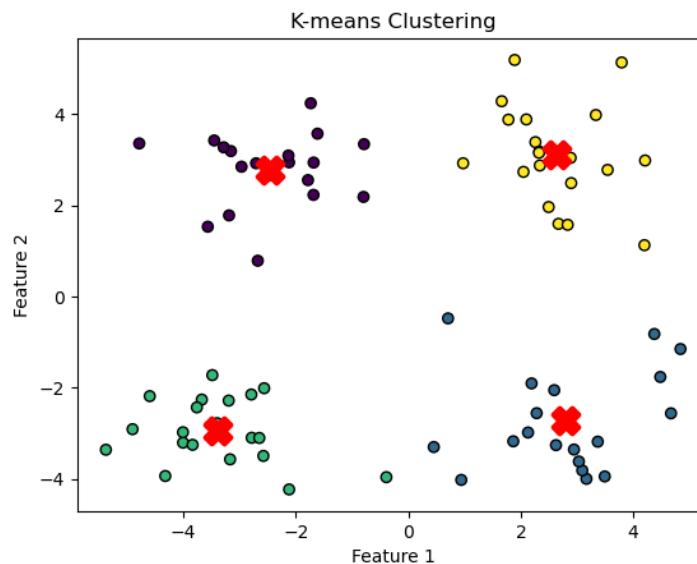
在碎石图上，选择误差平方和突然变小时对应的 k 值，即 k=4。

调用之前编写的 kmeans 算法函数进行聚类，并可视化聚类结果与最终的聚类中心：

```

1. label, center, loss_value = k_means(data_matrix, 4)
2.
3. plt.scatter(data_matrix[:, 0], data_matrix[:, 1], c=label, cmap='viridis', edgecolor
   s='k')
4. plt.scatter(center[:, 0], center[:, 1], marker='X', s=200, linewidths=3, color='red'
   )
5. plt.title('K-means Clustering')
6. plt.xlabel('Feature 1')
7. plt.ylabel('Feature 2')
8. plt.show()

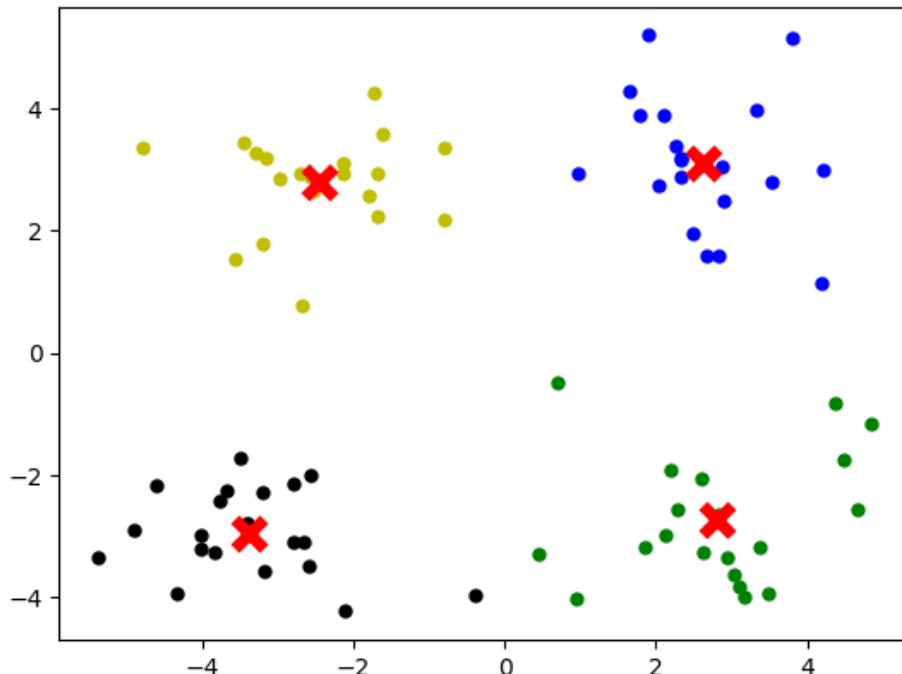
```



再调用 sklearn 库中的 K-means 算法，观察 k=4 时的聚类结果：

```
1. from sklearn.cluster import KMeans
2.
3. kmeans = KMeans(n_clusters=4, n_init='auto')
4.
5. # 进行聚类
6. kmeans.fit(data_matrix)
7.
8. # 获取每个数据点的标签
9. labels = kmeans.labels_
10.
11. # 获取每个簇的中心点
12. centers = kmeans.cluster_centers_
13.
14. # 可视化聚类结果
15. colors = ["k.", "b.", "g.", "y."]
16.
17. for i in range(len(data_matrix)):
18.     plt.plot(data_matrix[i][0], data_matrix[i][1], colors[labels[i]], markersize=10)
19.
20. plt.scatter(centers[:, 0], centers[:, 1], marker="x", s=150, linewidths=5, zorder=10
21.             , color='red')
22. plt.show()
```

可视化结果聚类结果和簇中心如下



与自己编写的 K-means 算法聚类结果比较，两者完全相同!!!

7.2 DBscan

DBscan 是一种著名的密度聚类算法，它基于一组“邻域”参数($\epsilon, MinPts$)来刻画样本分布的紧密程度。给定数据集 $D = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ ，定义下面这几个概念：

- ◆ ϵ -邻域：对 $\mathbf{x}_j \in D$ ，其 ϵ -邻域包含样本 D 中与 \mathbf{x}_j 的距离不大于 ϵ 的样本，即

$$N_\epsilon(\mathbf{x}_j) = \{\mathbf{x}_i \in D \mid \text{dist}(\mathbf{x}_i, \mathbf{x}_j) \leq \epsilon\};$$
- ◆ 核心对象：若 \mathbf{x}_j 的 ϵ -邻域至少包含 $MinPts$ 个样本，即 $|N_\epsilon(\mathbf{x}_j)| \geq MinPts$ ，则 \mathbf{x}_j 是一个核心对象；
- ◆ 密度直达：若 \mathbf{x}_j 位于 \mathbf{x}_i 的 ϵ -邻域中，且 \mathbf{x}_i 是核心对象，则称 \mathbf{x}_j 由 \mathbf{x}_i 密度直达；
- ◆ 密度可达：对 \mathbf{x}_i 与 \mathbf{x}_j ，若存在样本序列 $\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n$ ，其中 $\mathbf{p}_1 = \mathbf{x}_i$, $\mathbf{p}_n = \mathbf{x}_j$ 且 \mathbf{p}_{i+1} 由 \mathbf{p}_i 密度直达，则称 \mathbf{x}_j 由 \mathbf{x}_i 密度可达；
- ◆ 密度相连：对 \mathbf{x}_i 与 \mathbf{x}_j ，若存在 \mathbf{x}_k 使得 \mathbf{x}_i 与 \mathbf{x}_j 均由 \mathbf{x}_k 密度可达，则称 \mathbf{x}_i 与 \mathbf{x}_j 密度相连。

基于这些概念，DBSCAN 将“簇”定义为：由密度可达关系导出的最大的密度相连样本集合。算法描述如下：

输入：样本集 $D = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m\}$;
 邻域参数 $(\epsilon, MinPts)$.

过程：

- 1: 初始化核心对象集合: $\Omega = \emptyset$
- 2: **for** $j = 1, 2, \dots, m$ **do**
- 3: 确定样本 \mathbf{x}_j 的 ϵ -邻域 $N_\epsilon(\mathbf{x}_j)$;
- 4: **if** $|N_\epsilon(\mathbf{x}_j)| \geq MinPts$ **then**
- 5: 将样本 \mathbf{x}_j 加入核心对象集合: $\Omega = \Omega \cup \{\mathbf{x}_j\}$
- 6: **end if**
- 7: **end for**
- 8: 初始化聚类簇数: $k = 0$
- 9: 初始化未访问样本集合: $\Gamma = D$
- 10: **while** $\Omega \neq \emptyset$ **do**
- 11: 记录当前未访问样本集合: $\Gamma_{\text{old}} = \Gamma$;
- 12: 随机选取一个核心对象 $\mathbf{o} \in \Omega$ ，初始化队列 $Q = <\mathbf{o}>$;
- 13: $\Gamma = \Gamma \setminus \{\mathbf{o}\}$;
- 14: **while** $Q \neq \emptyset$ **do**
- 15: 取出队列 Q 中的首个样本 \mathbf{q} ;
- 16: **if** $|N_\epsilon(\mathbf{q})| \geq MinPts$ **then**
- 17: 令 $\Delta = N_\epsilon(\mathbf{q}) \cap \Gamma$;
- 18: 将 Δ 中的样本加入队列 Q ;
- 19: $\Gamma = \Gamma \setminus \Delta$;
- 20: **end if**
- 21: **end while**
- 22: $k = k + 1$, 生成聚类簇 $C_k = \Gamma_{\text{old}} \setminus \Gamma$;
- 23: $\Omega = \Omega \setminus C_k$
- 24: **end while**

输出：簇划分 $\mathcal{C} = \{C_1, C_2, \dots, C_k\}$

根据上述 DBSCAN 算法逻辑，将其封装成类：

```
1. class DBscan:
2.     def __init__(self, min_samples=10, eps=0.15):
3.         self.min_samples = min_samples
4.         self.eps = eps
5.         self.X = None
6.         self.label = None
7.         self.n_class = 0
8.
9.     def fit(self, X):
10.        self.X = X
11.        self.label = np.zeros(X.shape[0])
12.        for i in range(len(self.X)):
13.            if self.label[i] != 0:
14.                continue
15.
16.            neighbors = self.region_query(i)
17.            if len(neighbors) < self.min_samples:
18.                self.label[i] = 0
19.            else:
20.                self.n_class += 1
21.                self.expand_cluster(i, neighbors)
22.
23.    def region_query(self, i):
24.        return np.where(np.sqrt(np.sum((self.X - self.X[i])**2, axis=1)) <= self.eps)[0]
25.
26.    def expand_cluster(self, i, neighbors):
27.        self.label[i] = self.n_class
28.        j = 0
29.        while j < len(neighbors):
30.            neighbor_idx = neighbors[j]
31.            if self.label[neighbor_idx] == -1:
32.                self.label[neighbor_idx] = self.n_class
33.            elif self.label[neighbor_idx] == 0:
34.                self.label[neighbor_idx] = self.n_class
35.                new_neighbors = self.region_query(neighbor_idx)
36.                if len(new_neighbors) >= self.min_samples:
37.                    neighbors = np.concatenate((neighbors, new_neighbors))
38.                j += 1
39.
40.    def plot_dbscan(self):
41.        plt.rcParams['font.sans-serif'] = ["SimHei"]
42.        plt.rcParams['axes.unicode_minus']=False
43.        for i in range(self.n_class+1):
```

```

44.         if i == 0:
45.             label = 'Anomaly data'
46.         else:
47.             label = 'Class'+str(i)
48.         plt.scatter(self.X[self.label==i,0], self.X[self.label==i,1],label=label)
49.     plt.legend(loc='upper left', fontsize='small', bbox_to_anchor=(1.0, 1.0))
50. plt.show()

```

其中，`__init__`用于初始化，可以设置 ϵ 和 $MinPts$ 。`fit()`用于执行聚类过程，实现DBSCAN 算法的核心逻辑，对每个数据点进行遍历，如果该点还没有被标记，则进行区域查询(*region_query*)以找到邻域内的点。如果邻域内的点数量小于 `min_samples`，则将该点标记为离群点；否则，标记为新的簇，并扩展该簇(*expand_cluster*)。`region_query`实现 DBSCAN 中的邻域查询，找到距离当前点距离小于等于 ϵ 的点的索引。`expand_cluster`方法用于扩展簇，将当前点标记为簇的一部分，并递归地将邻域内的点加入簇中。而 `plot_dbscan` 方法：用于可视化 DBSCAN 的结果：遍历每个簇，将其点按簇标签不同进行散点图展示，同时将离群点单独展示。

下面使用 kmeans.txt 数据探究 ϵ 和 $MinPts$ 对聚类结果的影响：

① ϵ 对聚类结果的影响

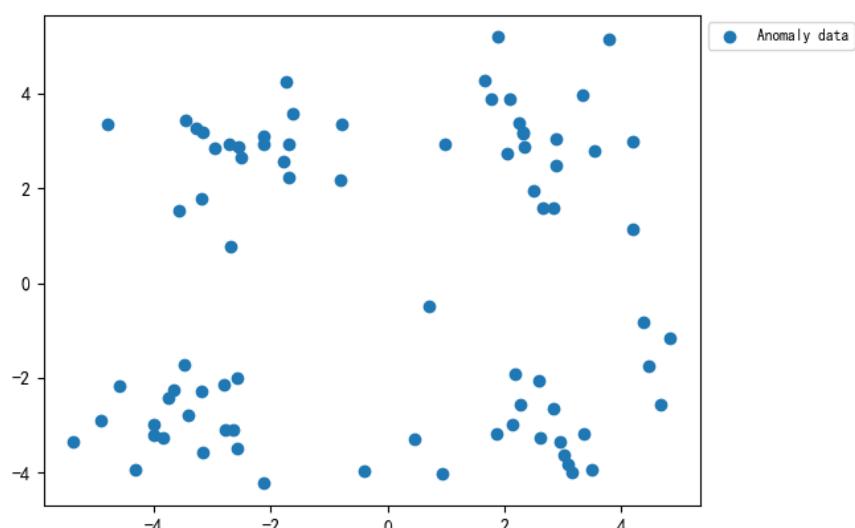
固定 $MinPts=10$ ，改变 ϵ 分别为 0.5, 1.0, 1.5, 2.0, 2.5, 5.0。

以 $MinPts=10$ ， $\epsilon=0.5$ 时为例，使用方法如下：

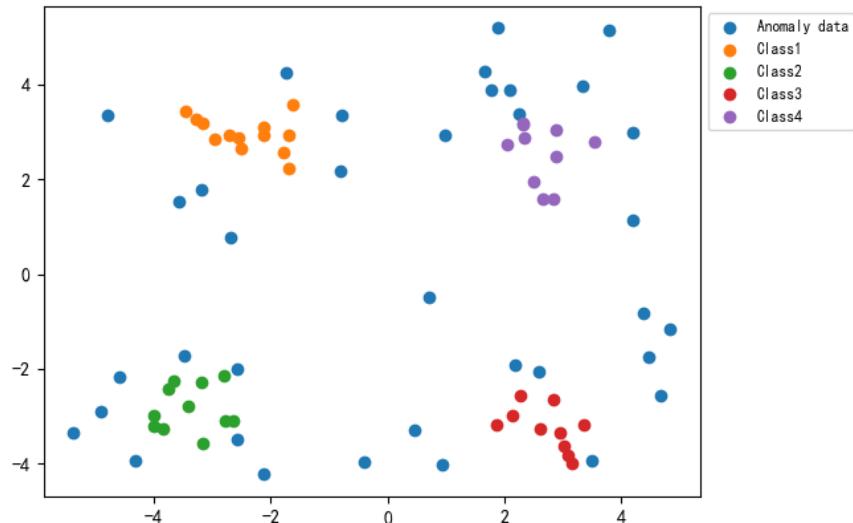
```

1. db_0 = DBscan(10, 0.5)
2. db_0.fit(data_matrix)
3. db_0.plot_dbscan()

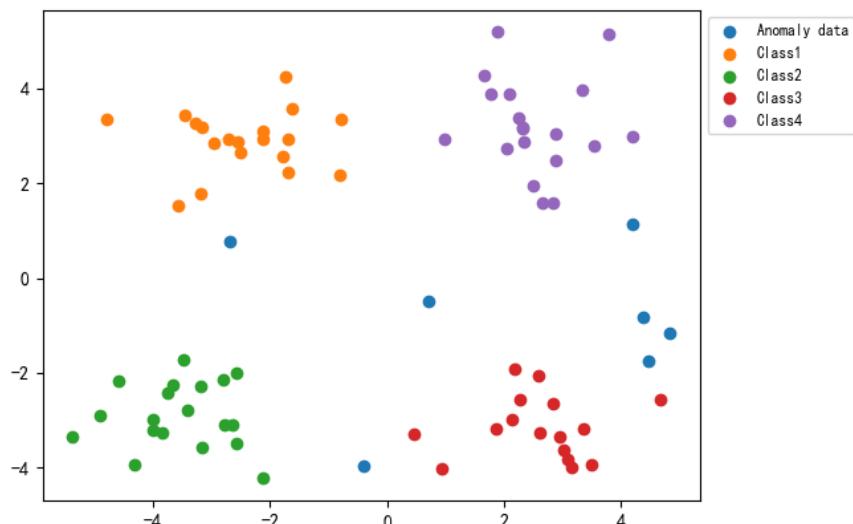
```



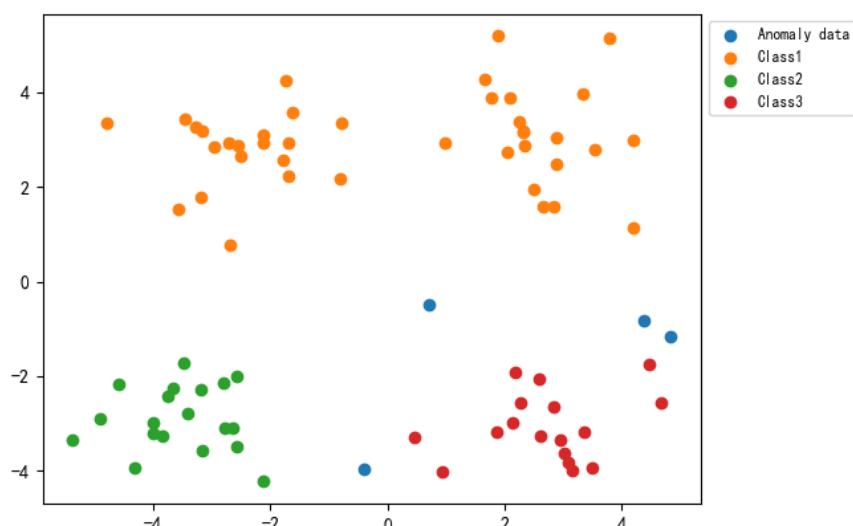
$MinPts = 10, \epsilon = 1.0$ 时



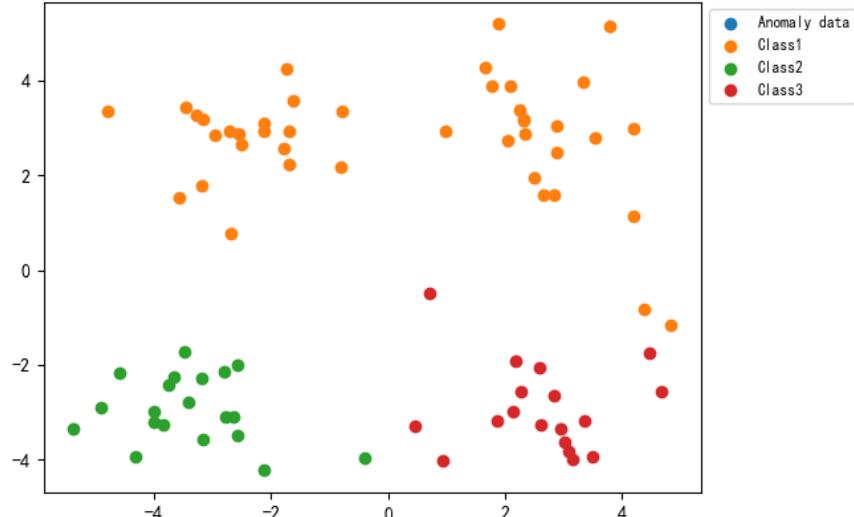
$MinPts = 10, \epsilon = 1.5$ 时



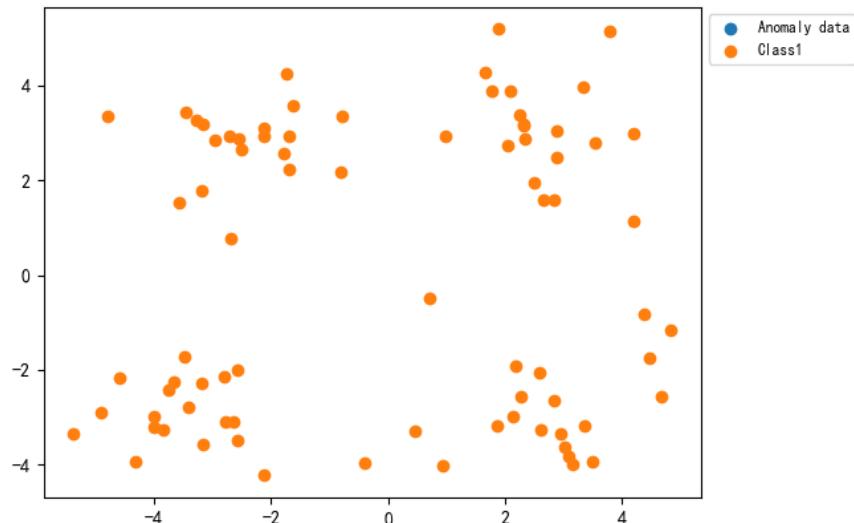
$MinPts = 10, \epsilon = 2.0$ 时



$MinPts = 10, \epsilon = 2.5$ 时



$MinPts = 10, \epsilon = 5$ 时



可以发现，随着 ϵ 值的增大，噪点减少，更多的点被聚类。

ϵ 过大，则更多的点会落在核心对象的 ϵ -邻域，此时类别数可能会减少，本来不应该是一类的样本也会被划为一类；反之 ϵ 过小则类别数可能会增多，本来是一类的样本却被划分开。

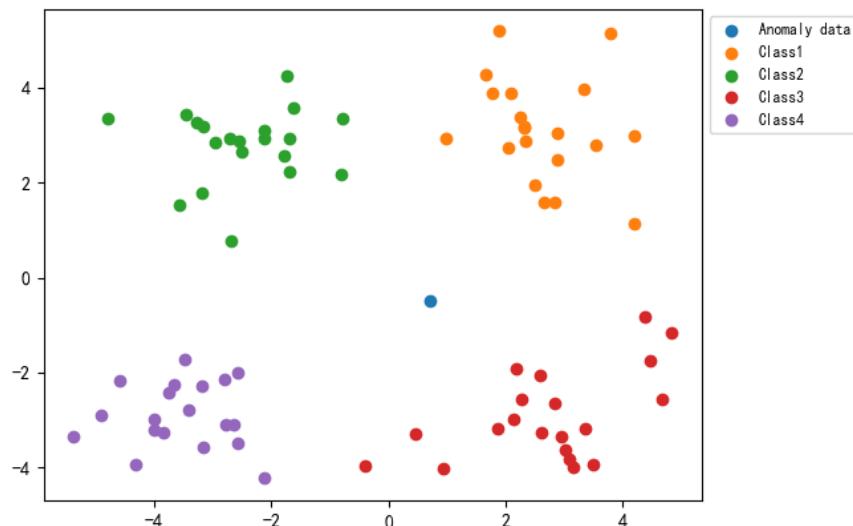
② $MinPts$ 对聚类结果的影响

固定 $\epsilon = 1.5$ ，改变 $MinPts$ 分别为 2, 5, 7, 10, 21。

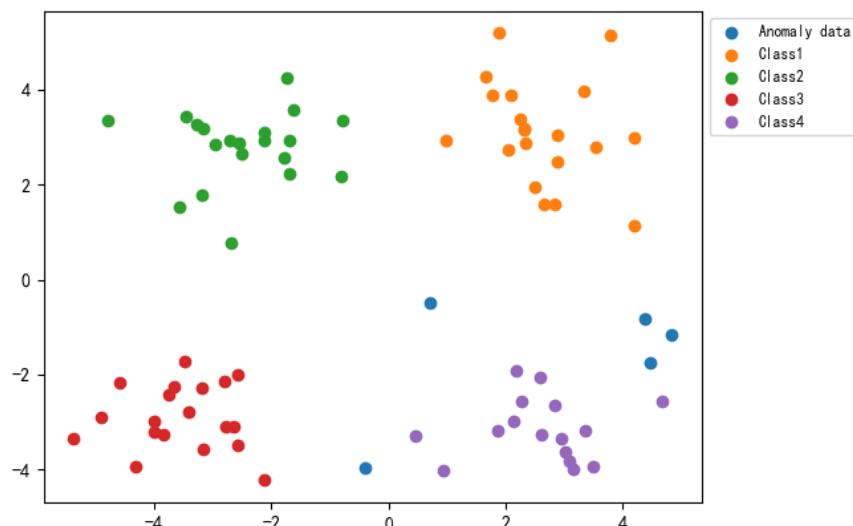
以 $MinPts = 2, \epsilon = 1.5$ 时为例，使用方法如下：

1. db_5 = DBscan(2, 1.5)
2. db_5.fit(data_matrix)
3. db_5.plot_dbSCAN()

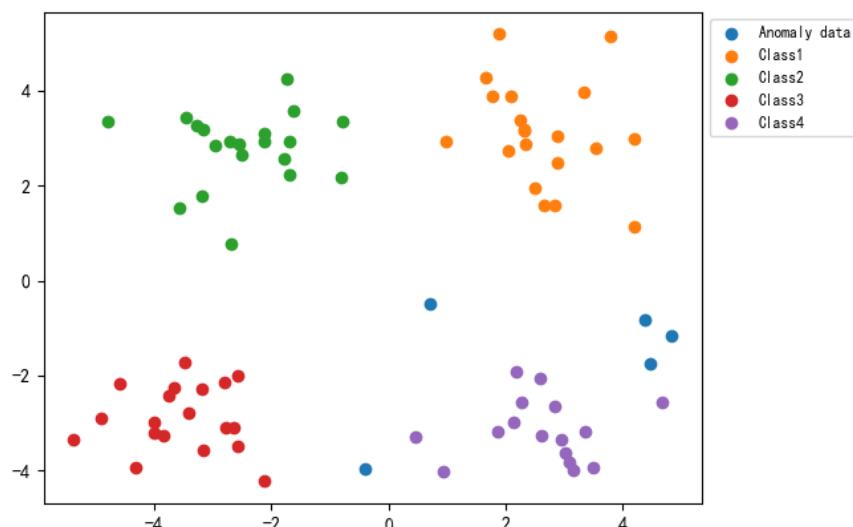
$MinPts = 2, \epsilon = 1.5$ 时



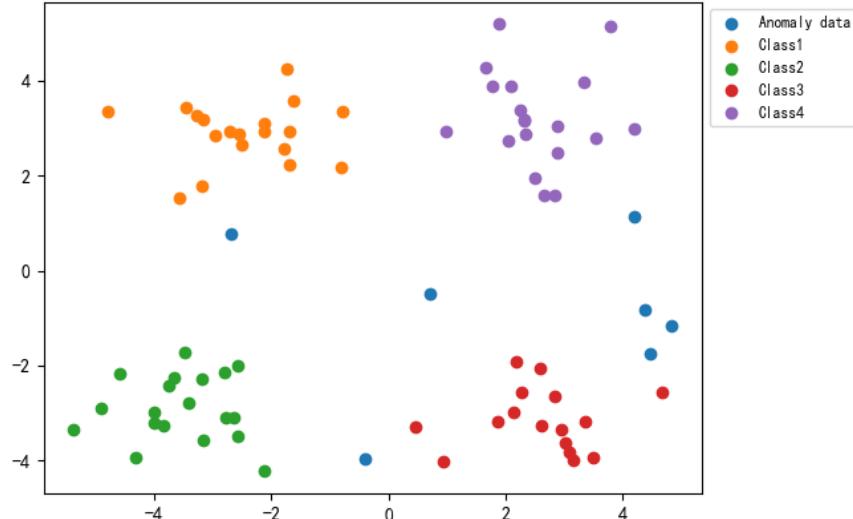
$MinPts = 5, \epsilon = 1.5$ 时



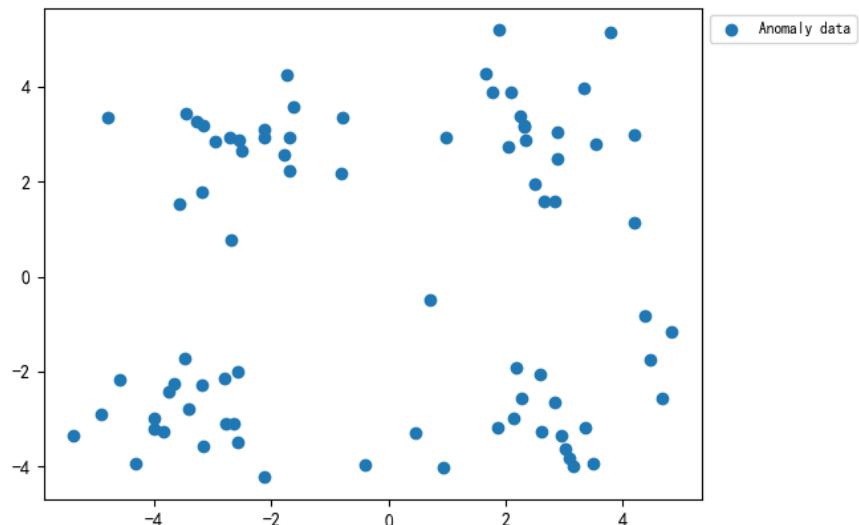
$MinPts = 7, \epsilon = 1.5$ 时



$MinPts = 10, \epsilon = 1.5$ 时



$MinPts = 21, \epsilon = 1.5$ 时



观察、比较发现，在 ϵ 一定的情况下， $MinPts$ 过大，则核心对象会过少，此时簇内部分本来是一类的样本可能会被标为噪音点；反之 $MinPts$ 过小的话，则会产生大量的核心对象，可能导致类别数变少。

探究完成后，尝试选择最优参数，对 kmeans.txt 数据进行密度聚类：

$MinPts$ 的选择：查阅资料， $MinPts$ 的值通常取数据的维度数+1，本数据集中数据维度为 2，因此取 $MinPts = 2 + 1 = 3$ 。

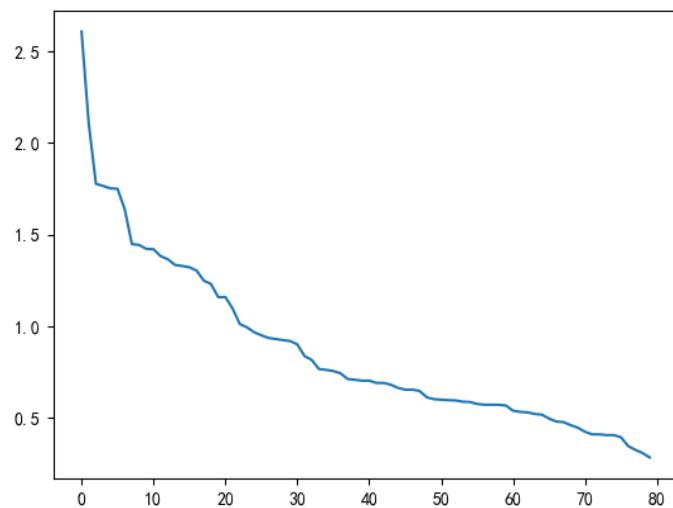
ϵ 的选择：查阅资料，选择 ϵ 的方法与 k-means 的“手肘法”有些类似，绘制 K-距离图（K-Distance Plot）可以帮助找到合适的 ϵ 。K-距离图显示每个数据点到其第 k 个最近邻居的距离，通常在距离图发生显著变化的点处，就是合适的 ϵ 值。

```

1. def select_MinPts(data,k):
2.     k_dist = []
3.     for i in range(data.shape[0]):
4.         dist = (((data[i] - data)**2).sum(axis=1)**0.5)
5.         dist.sort()
6.         k_dist.append(dist[k])
7.     return np.array(k_dist)
8.
9. k = 3
10. k_dist = select_MinPts(data_matrix,k)
11. k_dist.sort()
12. plt.plot(np.arange(k_dist.shape[0]),k_dist[::-1])

```

输出结果为

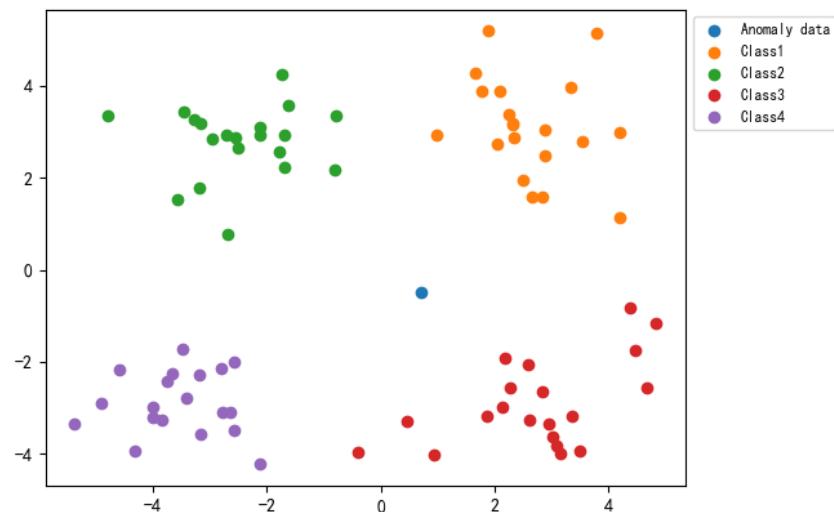


观察 K-距离图，选择 ϵ 为 1.5。参数选取完成后，调用自己编写的 DBSCAN 类进行聚类

```

1. db_b = DBscan(3, 1.5)
2. db_b.fit(data_matrix)
3. db_b.plot_dbSCAN()

```



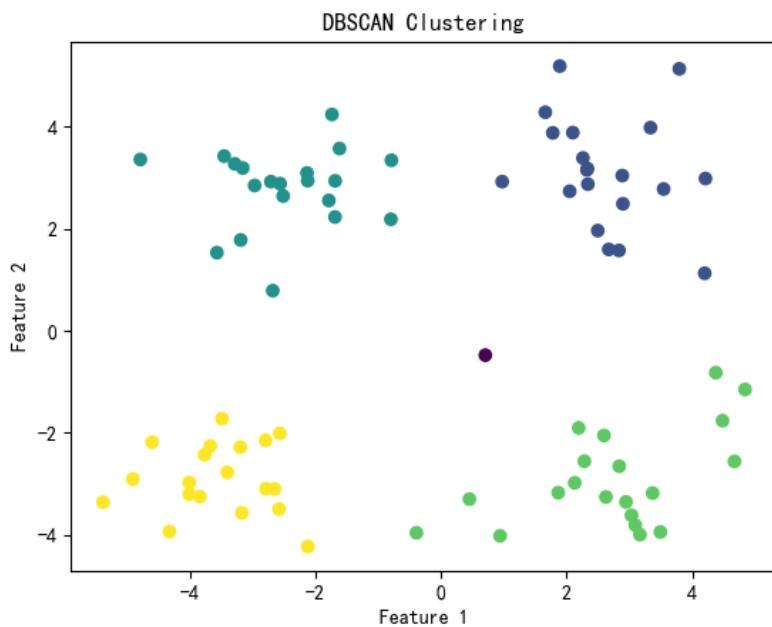
再调用 sklearn 库中的 DBSCAN 算法进行聚类，比较

```

1. # 使用 DBSCAN 进行聚类
2. dbSCAN = DBSCAN(eps=1.5, min_samples=4)
3. labels = dbSCAN.fit_predict(data_matrix)
4.
5. # 绘制聚类结果
6. plt.scatter(data_matrix[:, 0], data_matrix[:, 1], c=labels, cmap='viridis')
7. plt.title('DBSCAN Clustering')
8. plt.xlabel('Feature 1')
9. plt.ylabel('Feature 2')
10. plt.show()

```

聚类结果可视化为：



对比发现，自己编写的 DBSCAN 算法与调用 sklearn 库的聚类结果完全相同!!!

7.3 DBSCAN 与 K-means 的比较

为比较 DBSCAN 与 K-means 两种聚类算法，对同一数据集分别使用两种聚类算法，观察比较聚类结果。使用的数据和之前决策树章节使用的数据一样：

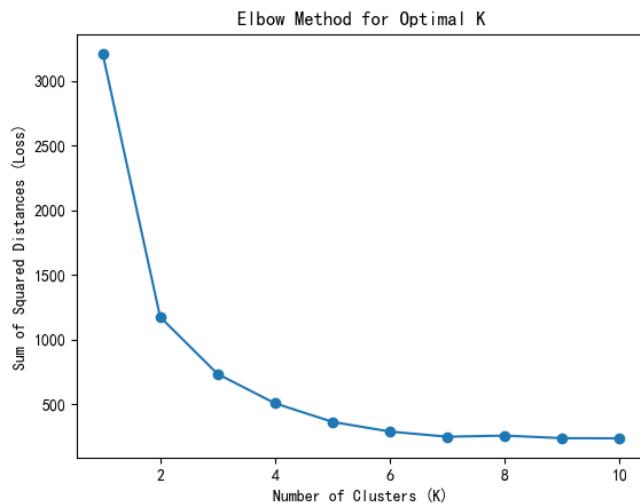
```

1. x1, y1 = datasets.make_circles(n_samples=2000, factor=0.5, noise=0.05, random_state=
   1017)
2.
3. x2, y2 = datasets.make_blobs(n_samples=1000, centers=[[1.2,1.2]], cluster_std=[[.1]]
   , random_state=1017)
4.
5. # 将向量纵向拼接
6. x_all = np.vstack((x1, x2))
7. y_all = np.concatenate((y1, y2))

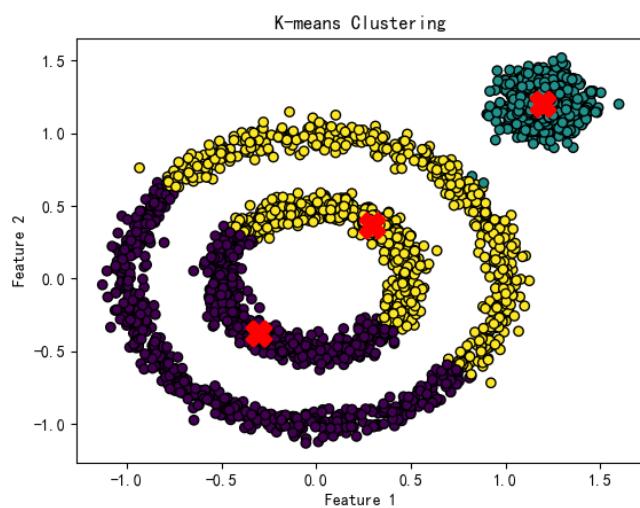
```

① 首先使用 K-means 聚类

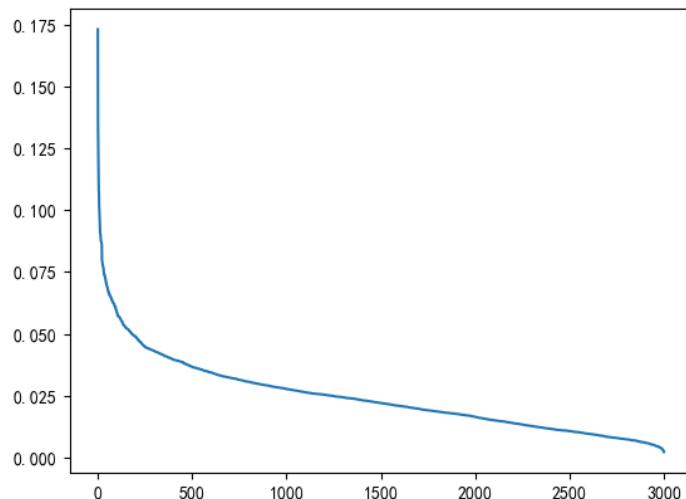
选取超参数 K:



观察碎石图，选定 K=3，进行聚类，代码与之前类似，故在此省略。可视化聚类结果为：

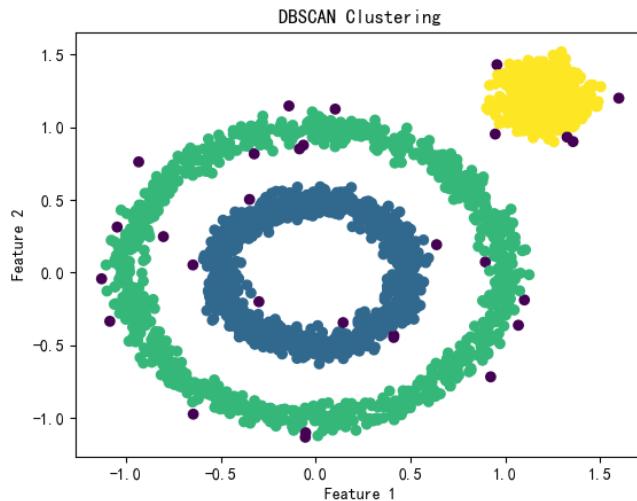


② 再使用 DBSCAN 聚类

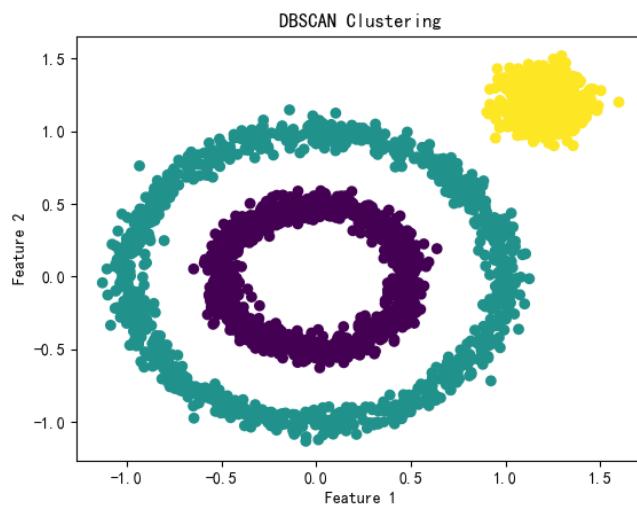


观察 K-距离图, 选择 ϵ 为 0.06。进行聚类, 代码与之前类似, 故在此省略。可视化

聚类结果为:



发现有一些离群点未被聚类, 进一步调整 $MinPts = 50$, $\epsilon=0.2$ 进行聚类:



此时聚类的效果很好, 这也从侧面说明参数选择是一项复杂的任务, 需要根据具体的数据和任务来进行调整。

比较 DBSCAN 与 K-means 如下:

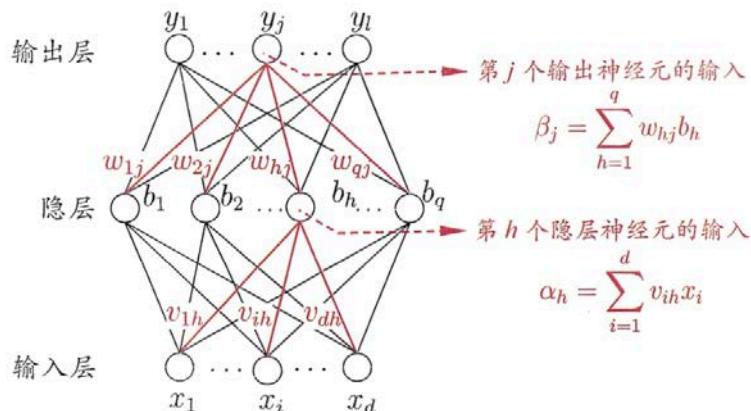
	DBSCAN	K-means
簇的形状	能够识别任意形状的簇, 不受簇的形状限制	对簇的形状有假设, 每个簇被假设为一个凸形
簇的数量	不需要预先指定簇的数量, 可以自动发现数据中的簇	需要预先指定簇的数量 K
处理噪声	对噪声具有鲁棒性, 噪声数据点不会被分到任何簇	对噪声敏感, 异常值可能会影响簇的形成

第8章 神经网络

8.1 基本概念

神经网络是一种运算模型，由大量的节点（或称“神经元”）和之间相互的联接构成。每个节点代表一种特定的输出函数，称为激励函数、激活函数（activation function）。每两个节点间的联接都代表一个对于通过该连接信号的加权值，称之为权重，这相当于人工神经网络的记忆。网络的输出则依网络的连接方式，权重值和激励函数的不同而不同。而网络自身通常都是对自然界某种算法或者函数的逼近，也可能是对一种逻辑策略的表达

BP（反向传播）算法是迄今最成功的神经网络学习算法。给定训练集 $D = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \dots, (\mathbf{x}_m, \mathbf{y}_m)\}$, $\mathbf{x}_i \in \mathbb{R}^d$, $\mathbf{y}_i \in \mathbb{R}^l$, 即输入示例由 d 个属性描述，输出 l 维实值向量。为便于讨论，下图给出了一个多层前馈网络结构：



其中输出层第 j 个神经元的阈值用 θ_j 表示，隐层第 h 个神经元的阈值用 γ_h 表示。输入层第 i 个神经元与隐层第 h 个神经元之间的连接权为 v_{ih} ，隐层第 h 个神经元与输出层第 j 个神经元之间的连接权为 w_{ij} 。记隐层第 h 个神经元接收到的输入为 $\alpha_h = \sum_{i=1}^d v_{ih} x_i$ ，输出层第 j 个神经元接收到的输入为 $\beta_j = \sum_{h=1}^q w_{hj} b_h$ ，其中 b_h 为隐层第 h 个神经元的输出。假设隐层和输出层神经元都使用 Sigmoid 函数。

对训练例 $(\mathbf{x}_k, \mathbf{y}_k)$ ，假定神经网络的输出为 $\hat{\mathbf{y}}_k = (\hat{y}_1^k, \hat{y}_2^k, \dots, \hat{y}_l^k)$ ，即

$$\hat{y}_j^k = f(\beta_j - \theta_j) ,$$

则网络在 $(\mathbf{x}_k, \mathbf{y}_k)$ 上的均方误差为

$$E_k = \frac{1}{2} \sum_{j=1}^l (\hat{y}_j^k - y_j^k)^2.$$

下面以隐层到输出层的连接权 w_{hj} 为例进行推导。

BP 算法基于梯度下降策略，以目标的负梯度方向对参数进行调整。对误差 E_k ，给定学习率 η ，有

$$\Delta w_{hj} = -\eta \frac{\partial E_k}{\partial w_{hj}}.$$

注意到 w_{hj} 先影响到第 j 个输出层神经元的输入值 β_j ，再影响到其输出值 \hat{y}_j^k 。然后影响到 E_k ，有

$$\frac{\partial E_k}{\partial w_{hj}} = \frac{\partial E_k}{\partial \hat{y}_j^k} \cdot \frac{\partial \hat{y}_j^k}{\partial \beta_j} \cdot \frac{\partial \beta_j}{\partial w_{hj}}.$$

根据 β_j 的定义，显然有

$$\frac{\partial \beta_j}{\partial w_{hj}} = b_h.$$

Sigmoid 函数有一个很好的性质：

$$f'(x) = f(x)(1-f(x)),$$

于是有

$$\begin{aligned} g_j &= -\frac{\partial E_k}{\partial \hat{y}_j^k} \cdot \frac{\partial \hat{y}_j^k}{\partial \beta_j} \\ &= -(\hat{y}_j^k - y_j^k) f'(\beta_j - \theta_j) \\ &= \hat{y}_j^k (1 - \hat{y}_j^k) (y_j^k - \hat{y}_j^k). \end{aligned}$$

故可得 BP 算法中关于 w_{hj} 的更新公式

$$\Delta w_{hj} = \eta g_j b_h.$$

类似可得

$$\Delta \theta_j = -\eta g_j,$$

$$\Delta v_{ih} = \eta e_h x_i,$$

$$\Delta \gamma_h = -\eta e_h.$$

8.2 编写反向传播算法

依据上述反向传播的逻辑，编写代码。为便于使用，将功能封装成类。若是分类任务，则最后的损失函数为交叉熵，对应的类为

```
1. class NeuralNetwork_clf:
2.     def __init__(self, layer_sizes, learning_rate=0.01):
3.         self.layer_sizes = layer_sizes
4.         self.learning_rate = learning_rate
5.         self.num_layers = len(layer_sizes)
6.
7.         # 初始化权重和偏置
8.         self.weights = [np.random.rand(layer_sizes[i], layer_sizes[i+1]) for i in range(self.num_layers - 1)]
9.         self.biases = [np.zeros((1, size)) for size in layer_sizes[1:]]
10.
11.    def sigmoid(self, x):
12.        return 1 / (1 + np.exp(-x))
13.
14.    def sigmoid_derivative(self, x):
15.        return x * (1 - x)
16.
17.    def cross_entropy_loss(self, y_true, y_pred):
18.        epsilon = 1e-15 # 避免对数中出现无穷大值
19.        y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
20.        return -np.sum(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
21.
22.    def softmax(self, x):
23.        exp_values = np.exp(x - np.max(x, axis=1, keepdims=True))
24.        return exp_values / np.sum(exp_values, axis=1, keepdims=True)
25.
26.    def forward(self, X):
27.        self.layer_outputs = [X]
28.        for i in range(self.num_layers - 1):
29.            layer_input = np.dot(self.layer_outputs[-1], self.weights[i]) + self.biases[i]
30.            if i == self.num_layers - 2:
31.                layer_output = self.softmax(layer_input)
32.            else:
33.                layer_output = self.sigmoid(layer_input)
34.            self.layer_outputs.append(layer_output)
35.        return self.layer_outputs[-1]
36.
37.    def backward(self, X, y_true):
38.        # 反向传播
```

```
39.         output_error = self.layer_outputs[-1] - y_true
40.
41.         self.weight_gradients = []
42.         self.bias_gradients = []
43.
44.         for i in range(self.num_layers - 2, -1, -1):
45.             # 计算梯度
46.             weight_gradient = np.dot(self.layer_outputs[i].T, output_error)
47.             bias_gradient = np.sum(output_error, axis=0, keepdims=True)
48.
49.             # 保存梯度
50.             self.weight_gradients.insert(0, weight_gradient)
51.             self.bias_gradients.insert(0, bias_gradient)
52.
53.             # 更新 output_error 用于下一层
54.             output_error = output_error.dot(self.weights[i].T) * self.sigmoid_derivative(self.layer_outputs[i])
55.
56.     def update_weights(self):
57.         # 更新权重和偏置
58.         for i in range(self.num_layers - 1):
59.             self.weights[i] -= self.weight_gradients[i] * self.learning_rate
60.             self.biases[i] -= np.sum(self.bias_gradients[i], axis=0, keepdims=True)
61.                 * self.learning_rate
61.
62.     def train(self, X, y_true, epochs=1000):
63.         losses = []
64.
65.         for epoch in range(epochs):
66.             # 前向传播
67.             self.forward(X)
68.
69.             # 计算损失
70.             loss = self.cross_entropy_loss(y_true, self.layer_outputs[-1])
71.
72.             # 反向传播
73.             self.backward(X, y_true)
74.
75.             # 更新权重和偏置
76.             self.update_weights()
77.
78.             # 记录损失
79.             losses.append(loss)
80.
81.             # 可视化损失
82.             plt.plot(range(epochs), losses)
```

```
83.     plt.xlabel('训练次数')
84.     plt.ylabel('损失函数值')
85.     plt.title('训练过程中损失函数变化')
86.     plt.show()
```

当创建一个 NeuralNetwork_clf 实例时，可以通过传递层大小（layer_sizes）和可选的学习率（learning_rate）来初始化神经网络。神经网络使用 sigmoid 激活函数来处理隐藏层，使用 softmax 激活函数来处理输出层，最后输出类别概率分布。

在初始化时，权重矩阵和偏置向量被随机初始化。然后，包含了一系列方法来执行前向传播、反向传播和权重更新。

➤ 前向传播 (forward):

接收输入 X，通过每一层的权重和偏置进行加权求和，并通过激活函数（sigmoid 或 softmax）得到每一层的输出。这些输出存储在 self.layer_outputs 列表中，最终的输出是网络的预测。

➤ 损失函数 (cross_entropy_loss):

采用交叉熵损失函数来计算预测输出与实际标签之间的误差。

➤ 反向传播 (backward):

通过计算输出误差和梯度，从输出层开始反向传播梯度，同时更新权重和偏置。这个过程使用了链式法则，从输出层向输入层逐层传递误差和梯度。

➤ 权重更新 (update_weights):

使用梯度下降法更新权重和偏置，减去学习率乘以对应的梯度。

➤ 训练函数 (train):

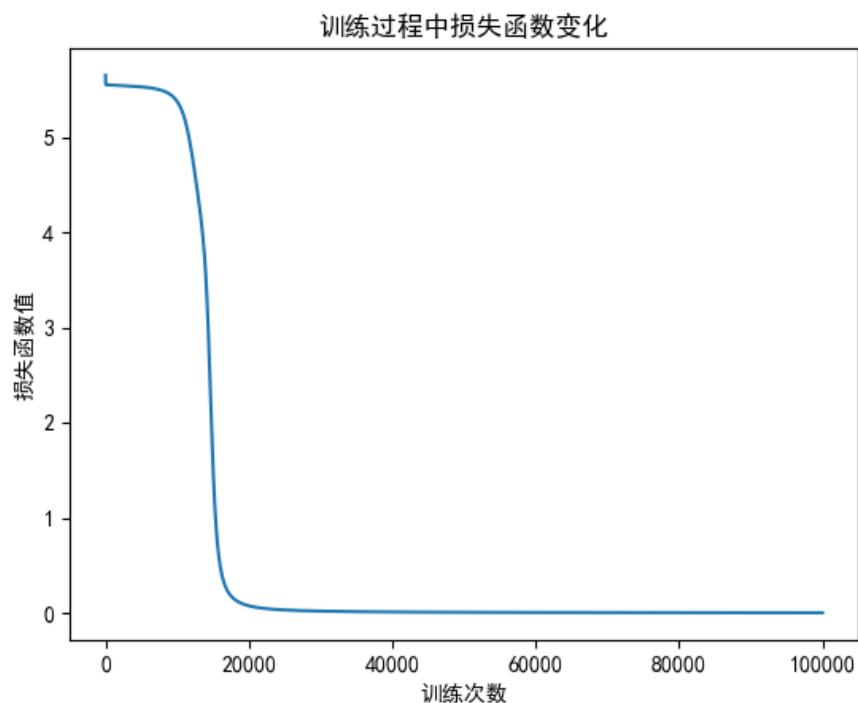
通过指定的迭代次数（epochs）来执行训练。每个迭代包括前向传播、计算损失、反向传播和权重更新。损失值被记录，最后通过 matplotlib 绘制损失函数随训练次数变化的图表。

若是一个回归任务，则最后的损失函数为均方差，对应的类不在本实验报告中展示，但大致的逻辑与处理分类任务一致。

接下来定义一个简单的二分类任务，测试上述代码是否能跑通

```
1. layer_sizes = [2, 4, 3, 2] # 输入层 2 个节点，隐藏层分别有 4 个和 3 个节点，输出层 2 个节点
2. learning_rate = 0.01
3.
4. nn_clf = NeuralNetwork_clf(layer_sizes, learning_rate)
```

```
5.  
6. X_train = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  
7.  
8. # One-hot 编码的标签  
9. y_train_one_hot = np.array([[1, 0], [0, 1], [0, 1], [1, 0]])  
10.  
11. # 训练模型  
12. nn_clf.train(X_train, y_train_one_hot, epochs=100000)  
13.  
14. # 测试模型  
15. test_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  
16. predictions = nn_clf.forward(test_data)  
17. print("\n 预测值:\n", predictions)
```



输出为：

```
预测值：  
[[ 9.99653872e-01  3.46128481e-04]  
[ 2.98668659e-04  9.99701331e-01]  
[ 2.98503990e-04  9.99701496e-01]  
[ 9.99681332e-01  3.18668428e-04]]
```

代码成功跑通!!!

8.3 分类任务及参数探究

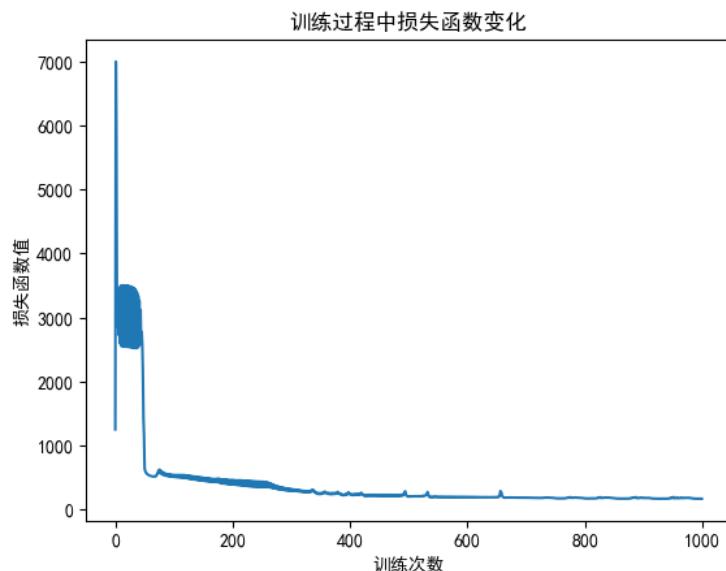
自己编写的反向传播算法成功跑通后，下面尝试进行一个分类任务，并探究参数对模型的影响。

生成一个二分类数据集

```
1. # 生成一个二分类数据集
2. X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, random_state=1017)
3.
4. y_onehot = np.zeros((y.shape[0], 2))
5. y_onehot[np.arange(y.shape[0]), y.flatten()] = 1
6.
7. X_train, X_test, y_train, y_test = train_test_split(X, y_onehot, test_size=0.2, random_state=1017)
```

后使用编写的反向传播神经网络训练并预测：

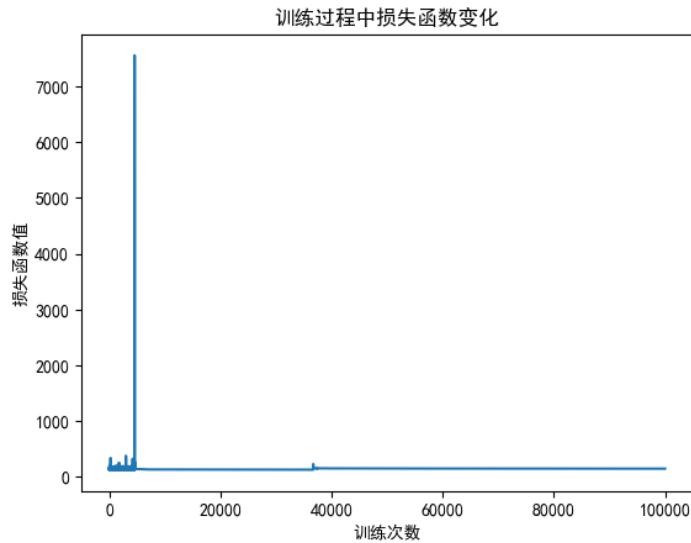
```
1. # 定义层维度
2. layer_dims = [X.shape[1], 5, 3, 2]
3.
4. model = NeuralNetwork_clf(layer_dims)
5.
6. model.train(X_train, y_train, epochs=1000)
7.
8. train_predictions = model.forward(X_train)
9. train_predictions = (train_predictions > 0.5).astype(int)
10.
11. test_predictions = model.forward(X_test)
12. test_predictions = (test_predictions > 0.5).astype(int)
13.
14. # 计算准确率
15. train_accuracy = accuracy_score(y_train.flatten(), train_predictions.flatten())
16. test_accuracy = accuracy_score(y_test.flatten(), test_predictions.flatten())
17.
18. print(f"训练集准确率:{train_accuracy*100}%")
19. print(f"测试集准确率:{test_accuracy*100}%")
```



输出为

训练集准确率: 94.875%
测试集准确率: 92.5%

模型表现良好。增加迭代次数到 100000，重新训练并预测，结果为

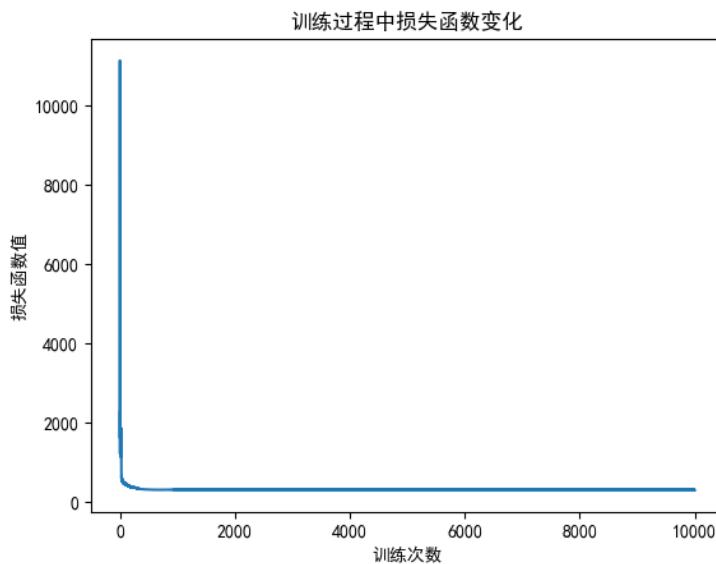


训练集准确率: 96.0%
测试集准确率: 91.0%

可以发现，次数为 1000 时，模型已经基本训练完成。过多的训练次数可能会使模型过拟合，即出现在训练集上表现变好，而在测试集上表现变差（泛化能力变弱）。

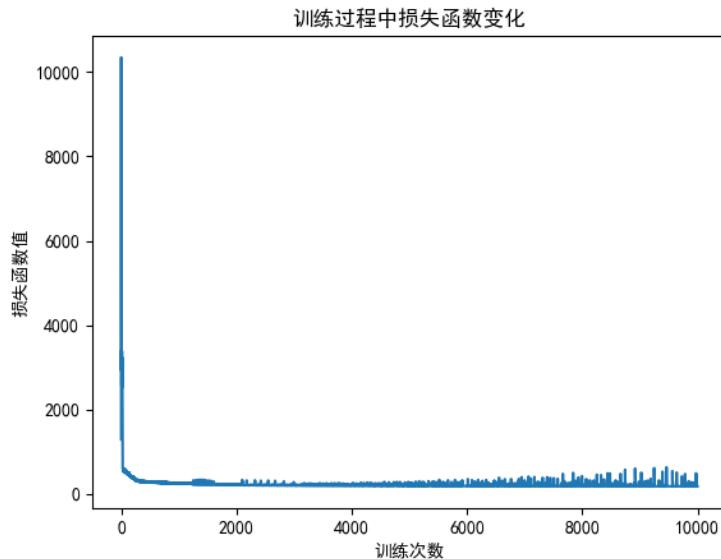
① 探究隐层层数对模型效果的影响

隐层层数=1，有 5 个神经元 (`layer_dims = [X.shape[1], 5, 2]`):



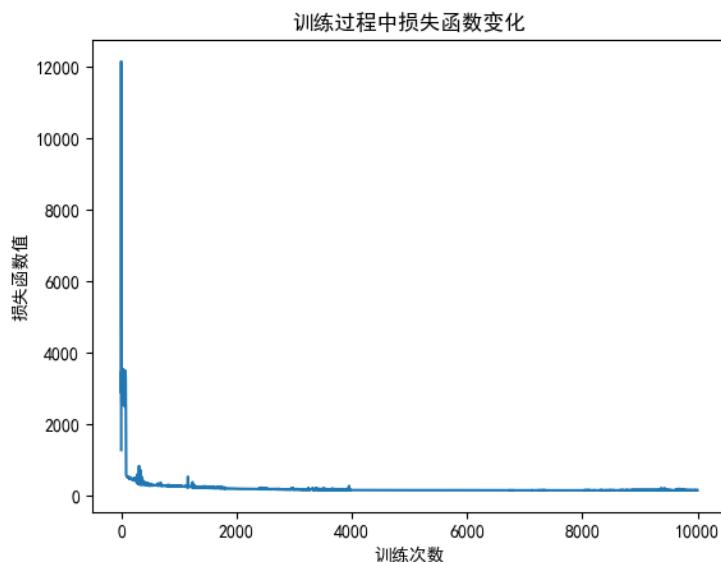
训练集准确率: 91.875%
测试集准确率: 92.0%

隐层层数=2，每层都有5个神经元（layer_dims = [X.shape[1], 5, 5, 2]）：



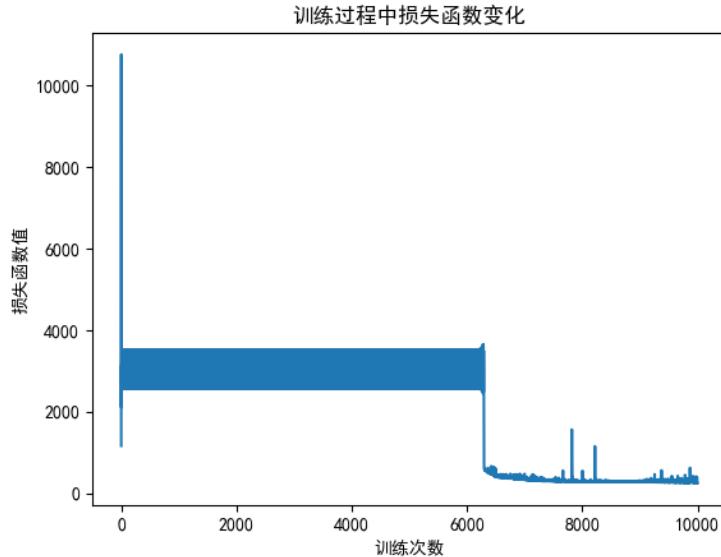
训练集准确率: 96.625%
测试集准确率: 93.0%

隐层层数=3，每层5个神经元（layer_dims = [X.shape[1], 5, 5, 5, 2]）：



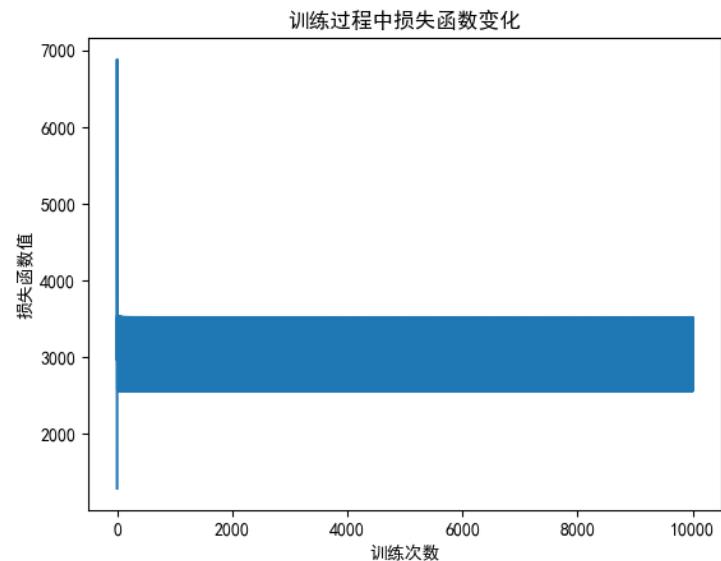
训练集准确率: 97.25%
测试集准确率: 95.5%

隐层层数=4，每层5个神经元（layer_dims = [X.shape[1], 5, 5, 5, 5, 2]）：



训练集准确率：95.0%
测试集准确率：94.5%

隐层层数=5，每层5个神经元（layer_dims = [X.shape[1], 5, 5, 5, 5, 5, 2]）：

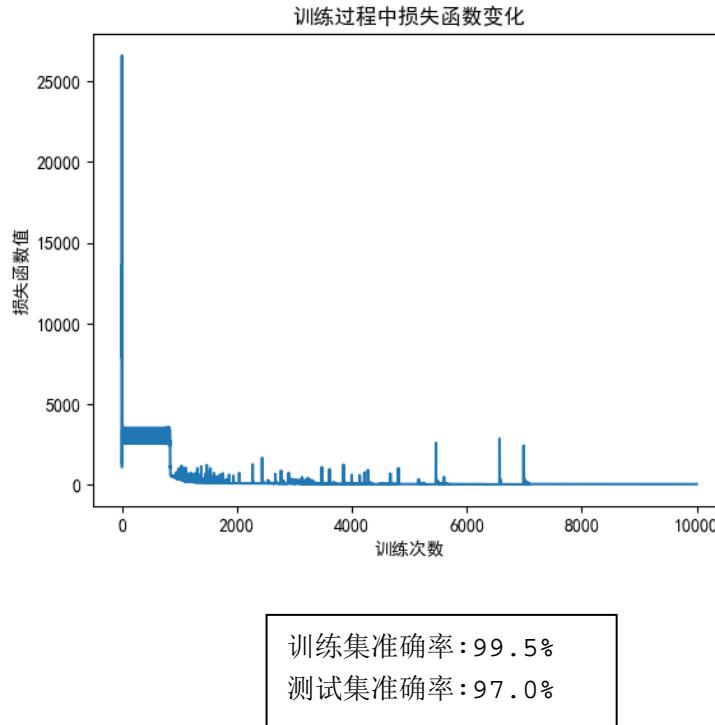


训练集准确率：48.125%
测试集准确率：56.000%

可以发现刚开始时，随着网络层数的增加，模型的效果得到了很好的提升；之后，可能是由于过多的层数增加训练难度，使得梯度消失，难以收敛，导致模型效果下降。

② 增加隐层中神经元个数

隐层层数=3，每层10个神经元（layer_dims = [X.shape[1], 10, 10, 10, 2]）：



与隐层层数=3，每层5个神经元的情形比较，模型效果得到了提高。

8.4 求解手写数字识别问题

Sklearn库中已封装好多层感知机算法，可以直接调用。

```

1. # 加载手写数字数据集
2. digits = datasets.load_digits()
3. X = digits.data
4. y = digits.target
5.
6. # 将数据集拆分为训练集和测试集
7. X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1017)
8.
9. # 创建多层感知机模型
10. mlp = MLPClassifier(hidden_layer_sizes=(100,), max_iter=500, activation='relu', solver='adam', random_state=1017)
11.
12. mlp.fit(X_train, y_train)
13.
14. y_pred = mlp.predict(X_test)
15.
16. # 计算准确度
17. accuracy = accuracy_score(y_test, y_pred)
18. print(f'准确度: {accuracy:.2f}')

```

输出为：

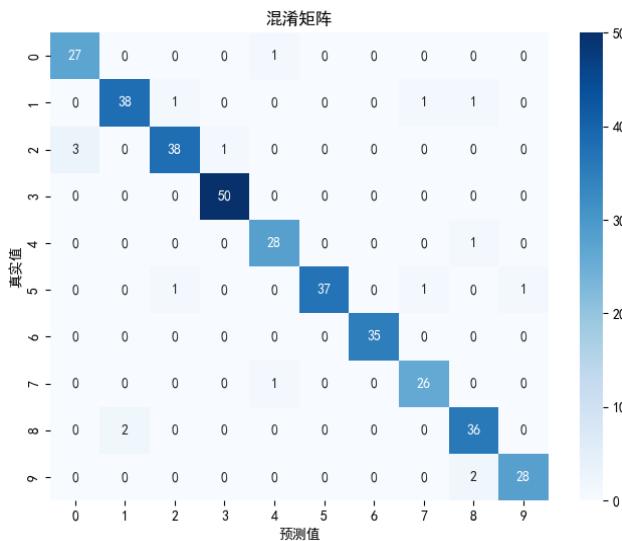
准确度： 0.95

模型效果很好。计算混淆矩阵并绘制热力图

```

1. # 计算混淆矩阵
2. cm = confusion_matrix(y_test, y_pred)
3.
4. # 使用 Seaborn 库绘制热力图
5. plt.figure(figsize=(8, 6))
6. sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=digits.target_names,
   yticklabels=digits.target_names)
7. plt.title('混淆矩阵')
8. plt.xlabel('预测值')
9. plt.ylabel('真实值')
10. plt.show()

```



尝试使用自己编写的神经网络算法求解手写数字识别问题：

```

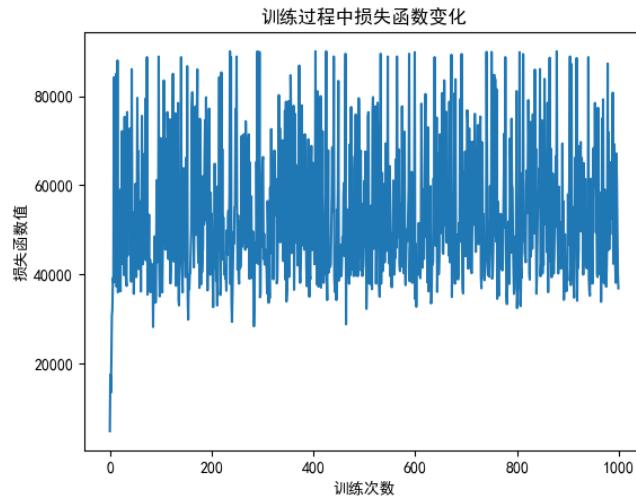
1. # 定义层维度
2. layer_dims = [X.shape[1], 3, 3, 10]
3.
4. model_shouxie = NeuralNetwork_clf(layer_dims, 0.1)
5.
6. model_shouxie.train(X_train, y_train, epochs=100000)
7.
8. # 在训练集上进行预测
9. train_predictions = model_shouxie.forward(X_train)
10. train_predictions = (train_predictions > 0.5).astype(int)
11.
12. # 在测试集上进行预测
13. test_predictions = model_shouxie.forward(X_test)
14. test_predictions = (test_predictions > 0.5).astype(int)

```

```

15.
16. # 计算准确率
17. train_accuracy = accuracy_score(y_train.flatten(), train_predictions.flatten())
18. test_accuracy = accuracy_score(y_test.flatten(), test_predictions.flatten())
19.
20. print(f"训练集准确率:{train_accuracy*100}%")
21. print(f"测试集准确率:{test_accuracy*100}%")

```



训练集准确率:82.08768267223383%
测试集准确率:81.555555555555556%

此处出现一个问题，从损失函数折线图可以看出，模型训练效果明显不好，为什么准确率还这么高？回看代码，发现上述计算准确率的方法只适用于二分类问题的准确率计算，更改计算方式

```

1. y_pred_classes = np.argmax(test_predictions, axis=1)
2. y_true_classes = np.argmax(y_test, axis=1)
3.
4. # 计算准确分类的样本数
5. correct_predictions = np.sum(y_pred_classes == y_true_classes)
6.
7. # 计算准确率
8. accuracy = correct_predictions / len(y_test)
9. print(f"测试集准确率:{accuracy*100}%")

```

新的准确率输出为：

测试集准确率:7.777777777777778%

新的准确率计算正常。

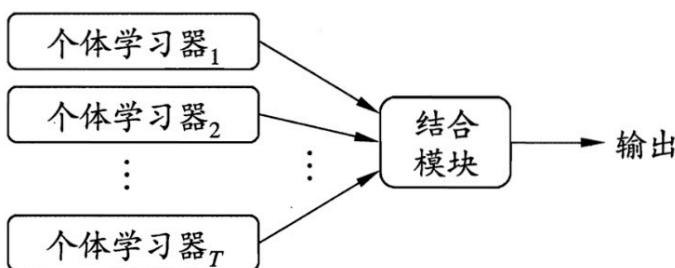
尝试更改参数（增加隐藏层层数、增加每层神经元个数……）重新训练，自己编写的神经网络算法计算速度太慢，故后续没有继续尝试。。。

第9章 集成学习

9.1 基本概念

集成学习通过构建并结合多个学习器来完成学习任务，有时也被称为多分类器系统、基于委员会的学习等。集成学习通过将多个学习器进行结合，常可获得比单一学习器显著优越的泛化性能。

如图显示出集成学习的一般结构：



先产生一组“个体学习器”，再用某种策略将它们结合起来。个体学习器通常由一个现有的学习算法从训练数据中产生，此时集成中只包含同种类型的个体学习器，这样的集成是同质的。同质集成中的个体学习器亦称“基学习器”，相应的学习算法称为“基学习算法”。集成也可包含不同类型的个体学习器，这样的集成是“异质”的。异质集成中的个体学习器由不同的学习算法生成，这是就不再有基学习算法；相应的，个体学习器一般不称为基学习器，常称为“组件学习器”或直接称为个体学习器。

9.2 Bagging 与随机森林：求解数字识别问题

Bagging 是并行式集成学习方法最著名的代表，它基于自助采样法。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，得到含 m 个样本的采样集，初始训练集中有的样本在采样集里多次出现，有的则未出现。

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 通常对分类任务进行简单投票法，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，则最简单的做法是随机选择一个，也可进一步考察学习器投票的置信度来确定最终胜者。Bagging 算法的描述为：

输入: 训练集 $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_m, y_m)\}$;
 基学习算法 \mathfrak{L} ;
 训练轮数 T .

过程:

- 1: **for** $t = 1, 2, \dots, T$ **do**
- 2: $h_t = \mathfrak{L}(D, \mathcal{D}_{bs})$
- 3: **end for**

输出: $H(\mathbf{x}) = \arg \max_{y \in \mathcal{Y}} \sum_{t=1}^T \mathbb{I}(h_t(\mathbf{x}) = y)$

随机森林(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。

Sklearn 库中已封装好随机森林算法，直接调用

```

1. digits = load_digits()
2. X, y = digits.data, digits.target
3.
4. # 数据集划分
5. X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1017)
6.
7. # 创建随机森林分类器
8. rf_classifier = RandomForestClassifier(n_estimators=100, random_state=1017)
9.
10. rf_classifier.fit(X_train, y_train)
11.
12. y_pred = rf_classifier.predict(X_test)
13.
14. # 计算准确率
15. accuracy = accuracy_score(y_test, y_pred)
16. print(f'Accuracy: {accuracy}')

```

输出为：

Accuracy: 0.9666666666667

随机森林模型效果优秀。

计算混淆矩阵并绘制热力图

```

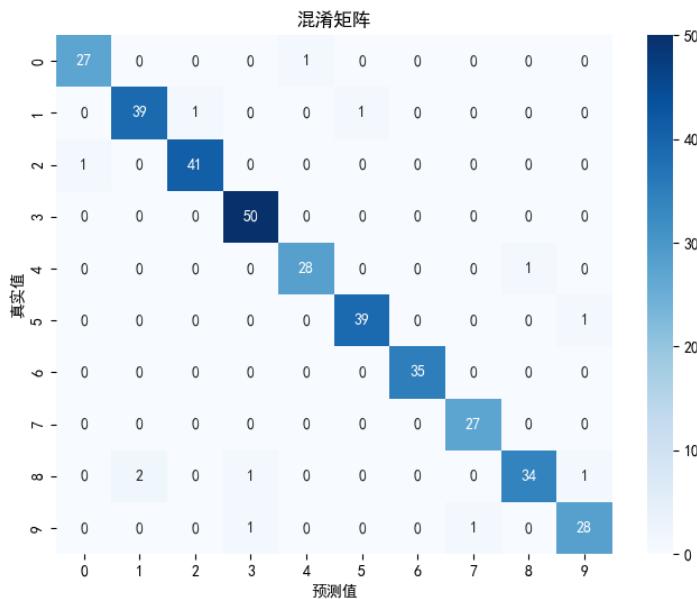
1. # 计算混淆矩阵
2. cm = confusion_matrix(y_test, y_pred)
3.
4. # 使用 Seaborn 库绘制热力图
5. plt.figure(figsize=(8, 6))
6. sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=digits.target_names,
   yticklabels=digits.target_names)

```

```

7. plt.title('混淆矩阵')
8. plt.xlabel('预测值')
9. plt.ylabel('真实值')
10. plt.show()

```

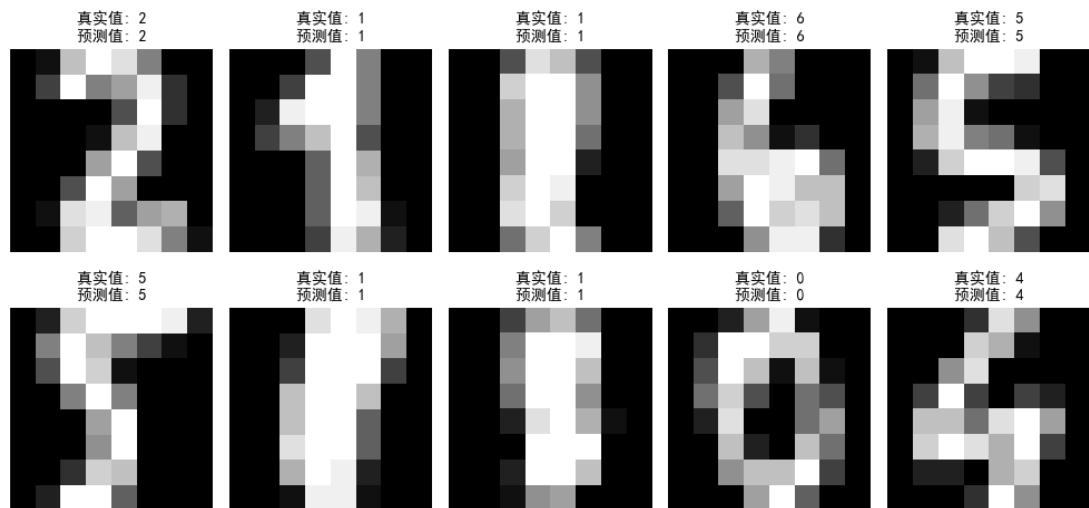


接着可视化一些样本的预测结果

```

1. n_samples = 10
2. indices = np.random.choice(len(X_test), n_samples, replace=False)
3.
4. plt.figure(figsize=(10, 5))
5. for i, index in enumerate(indices):
6.     plt.subplot(2, n_samples//2, i + 1)
7.     plt.imshow(X_test[index].reshape(8, 8), cmap='gray', interpolation='none')
8.     plt.title(f'真实值: {y_test[index]}\n预测值: {y_pred[index]}', fontsize=10)
9.     plt.axis('off')
10.
11. plt.tight_layout()
12. plt.show()

```



上述随机森林的基学习器数目为 100，调整基学习器数目分别为 10、50，重新训练并计算准确率。

① 基分类器数目=10

Accuracy: 0.9465

② 基分类器数目=50

Accuracy: 0.9660714285714286

比较发现，增加随机森林中的基学习器，可以

- 提高模型的准确性：随机森林中的每个基学习器都是独立训练的，增加基学习器的数目可以增加模型的表达能力，从而提高模型的准确性。
- 降低过拟合的风险：随机森林通过多个基学习器的集成来减少过拟合的风险。增加基学习器的数目可以增加模型的多样性，从而降低过拟合的风险。
- 提高模型的鲁棒性：随机森林中的每个基学习器都是基于不同的随机样本和随机特征进行训练的。增加基学习器的数目可以增加模型的多样性，提高模型对噪声和异常值的鲁棒性。

但同时，过大的基学习器数目会导致训练和预测时长的增加。

9.3 信用卡申请识别

数据来源于 kaggle 平台。原始数据分为两个文件，一个文件中存储特征数，另一个文件记录标签数据（0：通过，1：不通过）。

首先导入数据，对数据进行初步概览：

```
1. data = pd.read_csv("D:\BaiduSyncdisk\机器学习\shangji\data\Credit_card.csv") # 读取  
    特征数据  
2.  
3. data_label = pd.read_csv("D:\BaiduSyncdisk\机器学习  
    \shangji\data\Credit_card_label.csv") # 读取类别数据  
4.  
5. data.info()
```

输出如下：

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 1548 entries, 0 to 1547  
Data columns (total 18 columns):
```

#	Column	Non-Null Count	Dtype
0	Ind_ID	1548	non-null
1	GENDER	1541	non-null
2	Car_Owner	1548	non-null
3	Propert_Owner	1548	non-null
4	CHILDREN	1548	non-null
5	Annual_income	1525	non-null
6	Type_Income	1548	non-null
7	EDUCATION	1548	non-null
8	Marital_status	1548	non-null
9	Housing_type	1548	non-null
10	Birthday_count	1526	non-null
11	Employed_days	1548	non-null
12	Mobile_phone	1548	non-null
13	Work_Phone	1548	non-null
14	Phone	1548	non-null
15	EMAIL_ID	1548	non-null
16	Type_Occupation	1060	non-null
17	Family_Members	1548	non-null
dtypes:		float64(2), int64(8), object(8)	
memory usage:		217.8+	KB

可以看到 GENDER, Annual_income, Birthday_count, Type_Occupation 列存在缺失值。各字段意义如下：

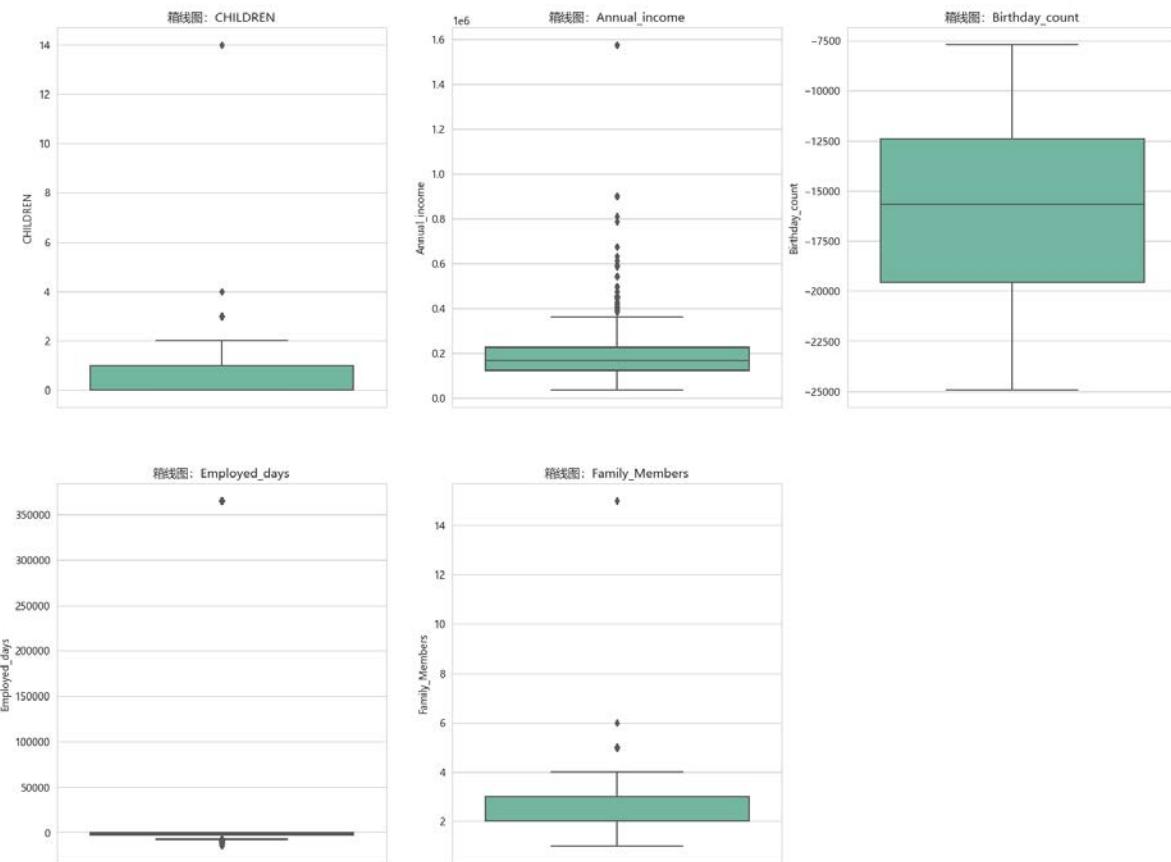
字段	说明
Ind_ID	客户ID
Gender	性别信息
Car_owner	是否有车
Propert_owner	是否有房产
Children	子女数量
Annual_income	年收入
Type_Income	收入类型
Education	教育程度
Marital_status	婚姻状况
Housing_type	居住方式
Birthday_count	以当前日期为0, 往前倒数天数
Employed_days	就业开始日期。当前日期为0, 往前倒数天数 (正值则意味着个人目前未就业)
Mobile_phone	手机号码
Work_phone	工作电话
Phone	电话号码
EMAIL_ID	电子邮箱
Type_Occupation	职业
Family_Members	家庭人数

将两个文件的数据按照 id 对应拼接

```
1. data = pd.merge(data, data_label, on="Ind_ID") # 将特征和类别按照对应的 ID 进行拼接
```

绘制箱线图：

```
1. columns = ['CHILDREN', 'Annual_income', 'Birthday_count', 'Employed_days', 'Family_M  
embers'] # 对数值型数据绘制箱线图
2. plt.figure(figsize=(20,15))
3. for i,col in enumerate(columns,1):
4.     plt.subplot(2,3,i)
5.     sns.boxplot(y=data[col],palette="Set2")
6.     plt.title(f'箱线图: {col}'')
```



可以看到，

CHILDREN 列有一个值为 14，可能是异常值；

Employed_days 列有个很大的正值，换算成年将近 1000 年，可能为异常值；

Family_Members 列有个极大值为 15，可能是异常值。

下面对部分数据进行处理

① 数据类型转换

```
1. data['Ind_ID'] = data['Ind_ID'].astype(str) #将 Ind_ID 转化为 object 格式
```

将 id 从 int 类型转换为 object 类型。

② 缺失值处理

之前已经发现 GENDER, Annual_income, Birthday_count, Type_Occupation 列存在缺失值。分别对它们的缺失值进行众数填充、中位数填充、文本填充等操作。

```

1. gender_mode = data['GENDER'].mode()[0]
2. data['GENDER'].fillna(gender_mode,inplace=True) # 对‘GENDER’的缺失值采用众数填充
3.
4. data['Type_Occupation'].fillna('unknown',inplace=True) # 对‘Type_Occupation’的缺失值
   填充“unknown”
5.
6. annual_income_median = data['Annual_income'].median()
7. data['Annual_income'].fillna(annual_income_median,inplace=True) # 对‘Annual_income’
   的缺失值采用中位数填充
8.
9. birthday_count_median = data['Birthday_count'].median()
10. data['Birthday_count'].fillna(birthday_count_median,inplace=True) # 对
    ‘Birthday_count’的缺失值采用中位数填充

```

填充后使用 `data.isna().sum()` 命令查看是否还存在缺失值

Ind_ID	0
GENDER	0
Car_Owner	0
Propert_Owner	0
CHILDREN	0
Annual_income	0
Type_Income	0
EDUCATION	0
Marital_status	0
Housing_type	0
Birthday_count	0
Employed_days	0
Mobile_phone	0
Work_Phone	0
Phone	0
EMAIL_ID	0
Type_Occupation	0
Family_Members	0
label	0

缺失值均已被填充！

③ 数据转换

Birthday_count 已天数为单位，且存在负号，将其转换为年份并去掉负号。

```

1. data['Age'] = (-data['Birthday_count'] / 365).astype(int) # 将‘Birthday_count’列转化
   为‘Age’列
2. data.drop('Birthday_count',axis=1, inplace=True)

```

④ 异常数据处理

对 CHILDREN 列和 Family_Members 列中的异常值，均采用中位数替换。

```

1. data.loc[data['CHILDREN'] == 14, 'CHILDREN'] = data['CHILDREN'].median() # 使用中位数替换异常值
2.
3. data.loc[data['Family_Members'] == 15, 'Family_Members'] = data['Family_Members'].median() # 使用中位数替换异常值

```

而对 Employed_days，按理来说应该全为非正数，需进一步讨论

```

1. #查看'Employed_days'正值的情况
2. positive_employed_days = data[data['Employed_days'] > 0]['Employed_days']
3. positive_employed_days.describe()

```

count	261.0
mean	365243.0
std	0.0
min	365243.0
25%	365243.0
50%	365243.0
75%	365243.0
max	365243.0
Name:	Employed_days, dtype: float64

可以看出，共有 261 行数据的“Employed_days”为正值，且值都相同，均为 365243。故 365243 可能代表特殊含义！挑选“Employed_days”为 365243 的数据，探究其规律：

```

1. positive_employed_data = data[data['Employed_days'] == 365243]
2. positive_employed_data.describe() # “Employed_days”为正值的数值型特征概览

```

	CHILDREN	Annual_Income	Employed_days	Mobile_phone	Work_Phone	Phone	EMAIL_ID	Family_Members	label	Age
count	261.000000	261.000000	261.0	261.0	261.0	261.000000	261.000000	261.000000	261.000000	261.000000
mean	0.034483	151277.396552	365243.0	1.0	0.0	0.306513	0.015326	1.659004	0.134100	59.199234
std	0.237694	81253.131354	0.0	0.0	0.0	0.461931	0.123081	0.563817	0.341414	5.457401
min	0.000000	33750.000000	365243.0	1.0	0.0	0.000000	0.000000	1.000000	0.000000	35.000000
25%	0.000000	99000.000000	365243.0	1.0	0.0	0.000000	0.000000	1.000000	0.000000	57.000000
50%	0.000000	135000.000000	365243.0	1.0	0.0	0.000000	0.000000	2.000000	0.000000	60.000000
75%	0.000000	180000.000000	365243.0	1.0	0.0	1.000000	0.000000	2.000000	0.000000	63.000000
max	3.000000	630000.000000	365243.0	1.0	0.0	1.000000	1.000000	5.000000	1.000000	68.000000

```

1. explore_data = data[data['Employed_days'] == 365243][['Type_Income', 'Type_Occupation']]
2. type_income_counts = explore_data['Type_Income'].value_counts()
3. type_occupation_counts = explore_data['Type_Occupation'].value_counts()
4. print(type_income_counts)
5. print("-----")
6. print(type_occupation_counts)

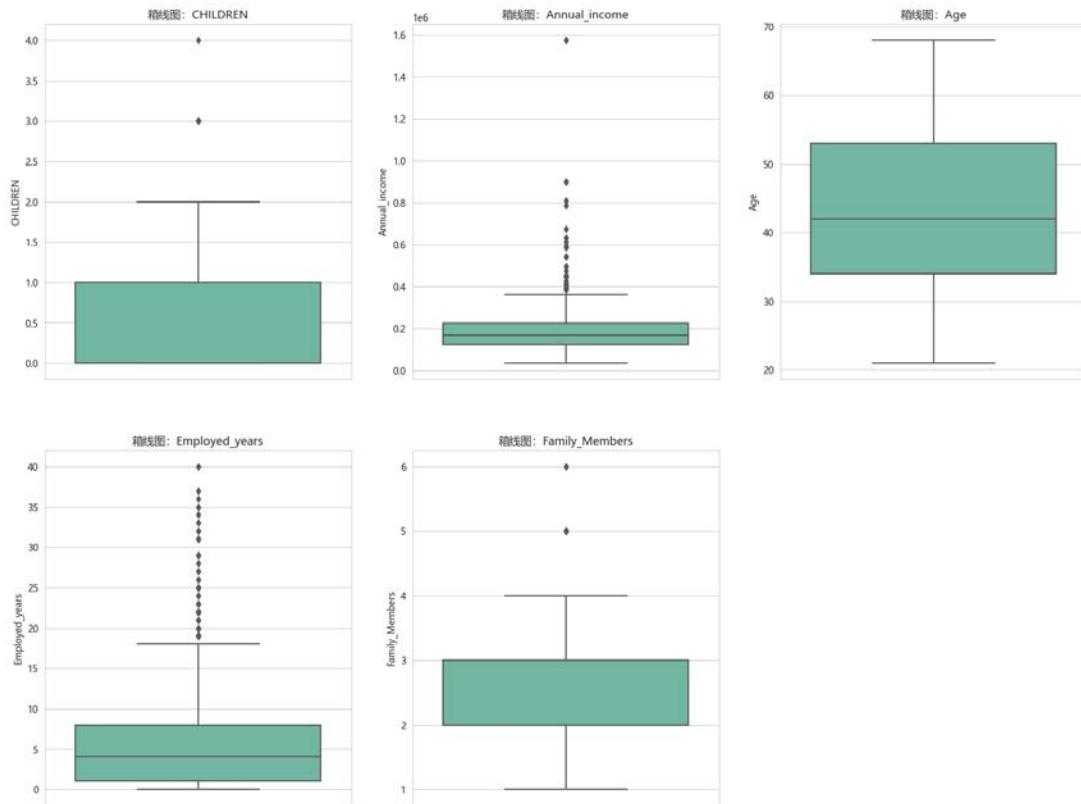
```

```
Type_Income
Pensioner    261
Name: count, dtype: int64
-----
Type_Occupation
unknown      261
Name: count, dtype: int64
```

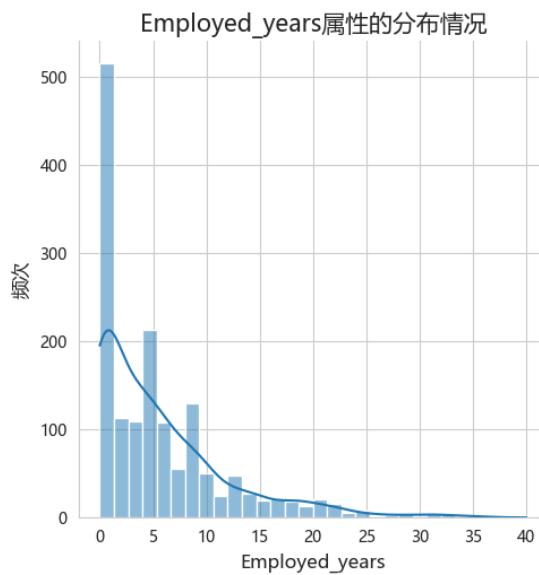
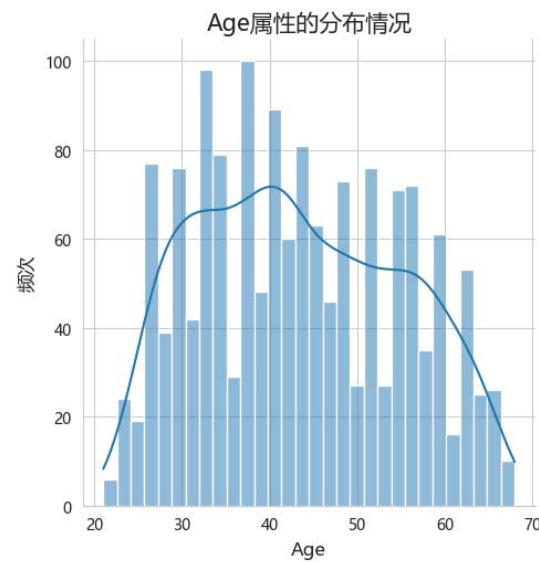
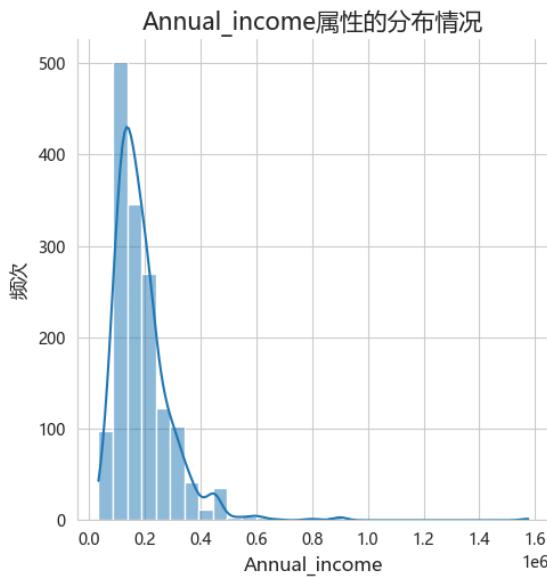
发现，所有“Employed_days”为正值的行，其‘Type_Income’和‘Type_Occupation’均相同，分别为 **Pensioner**（退休人士）和 **unknown**。故对这部分数据进一步处理，并将受聘从天数转为年数：

```
1. #将“Employed_days”为正值的行的‘Type_Occupation’列设置成‘Pensioner’
2. data.loc[data['Employed_days'] == 365243, 'Type_Occupation'] = 'Pensioner'
3.
4. data['Is_Retired'] = (data['Employed_days'] == 365243).astype(int) # 创建一个新的二
   元特征 “Is_Retired”，为退休人员设置为 1，其他人为 0
5. data.loc[data['Employed_days'] == 365243, 'Employed_days'] = 0 # 将 Employed_days 列
   中的正值转换为 0
6. #将 Employed_days 转化为 Employed_years
7. data['Employed_years'] = (-data['Employed_days'] / 365).astype(int)
8. #删除‘Employed_days’
9. data.drop('Employed_days', axis=1, inplace=True)
```

重新绘制箱线图



接着进行数据探索，绘制数值型变量、分类变量及申请结果的分布图，详细的代码在本报告中省略：

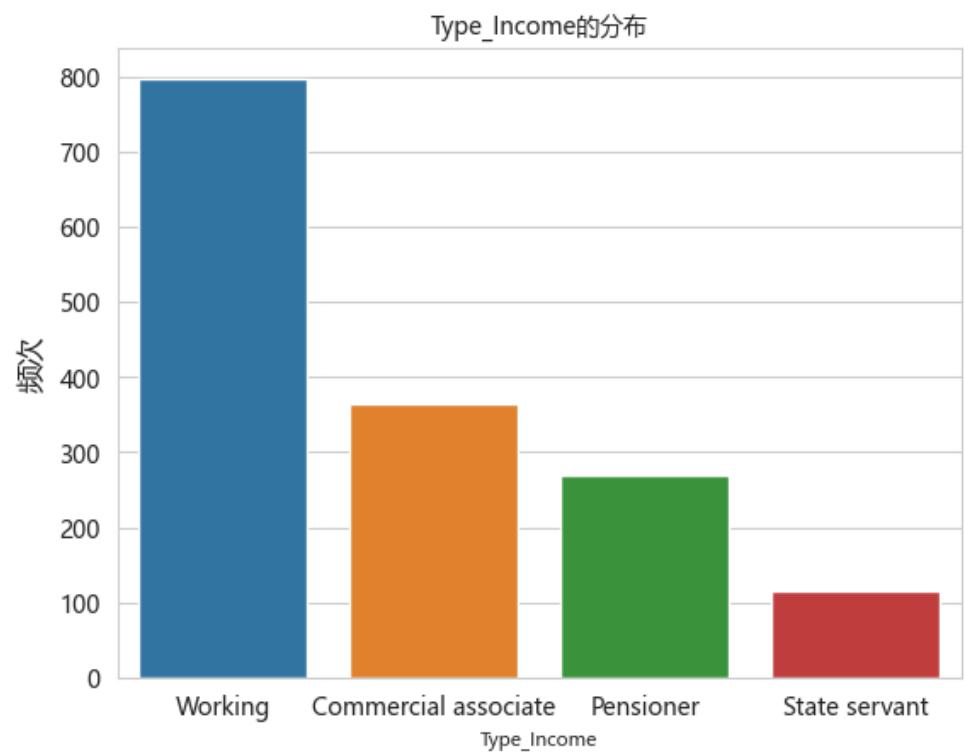
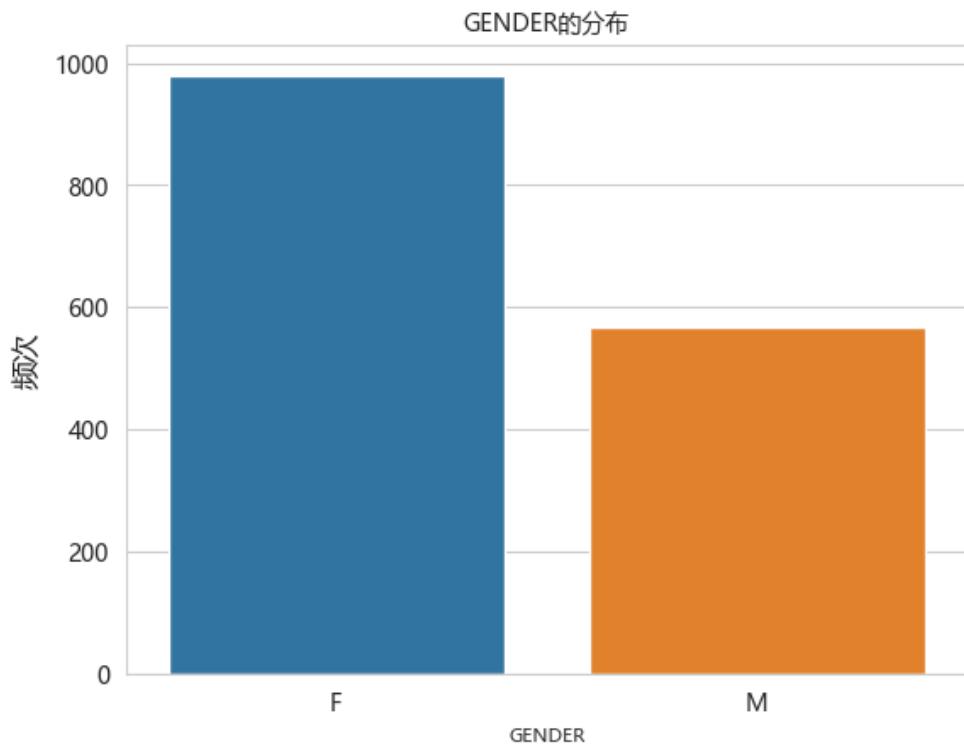


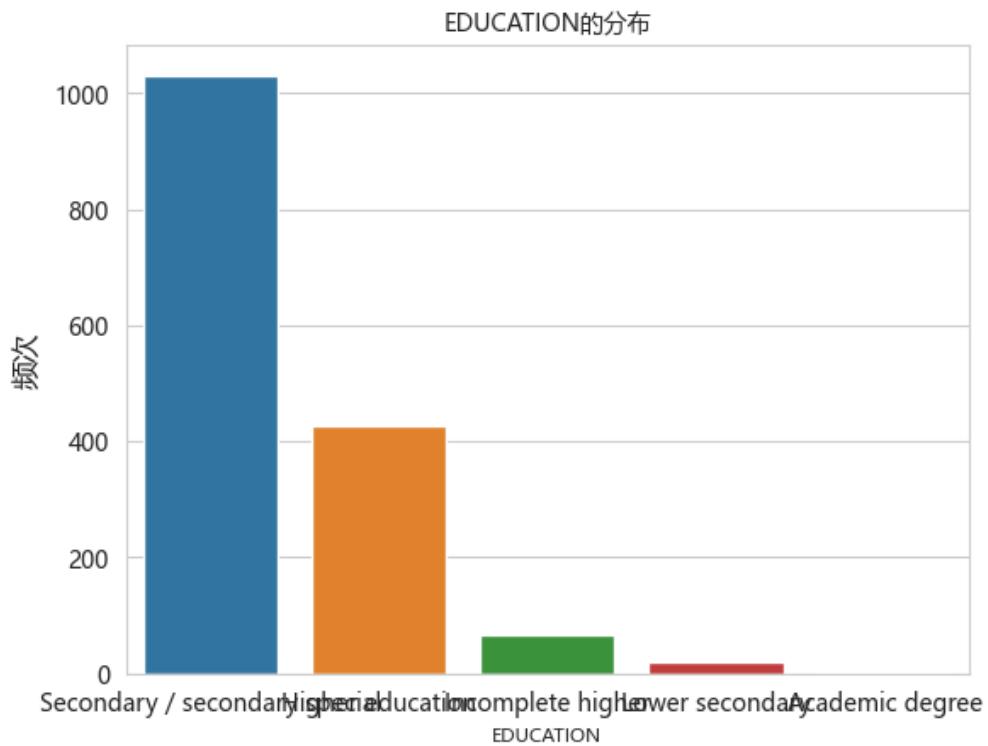
可以发现，数值型变量中：

大多数人收入较低，极少数人收入较高；

年龄分布在 20 岁—70 岁间，分布较为均匀；

大多数人工龄较低，其中为 0 的占多数，这部分可能是由于将退休人员的工龄也设置为 0。



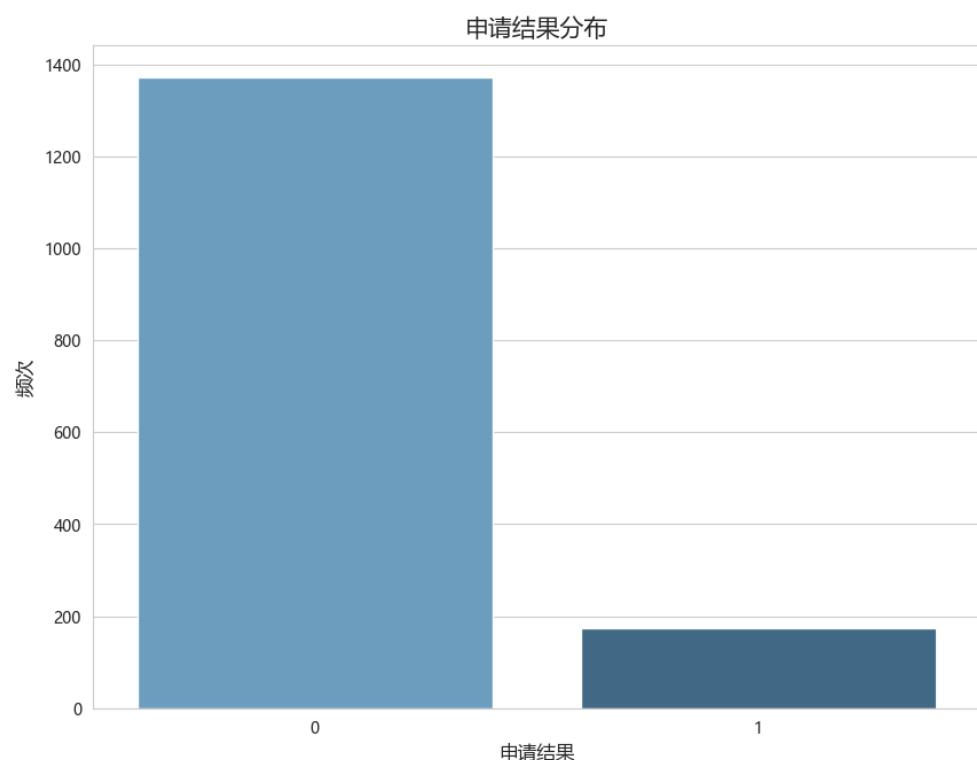


可以发现，分类变量中：

数据集中女性数量多于男性

收入类型中，Working 类型占大多数

学历背景中，大多数人属于中等教育，极少人拥有高学历或低学历



可以明显看出数据集中通过的样本量明显多于不通过的样本量！

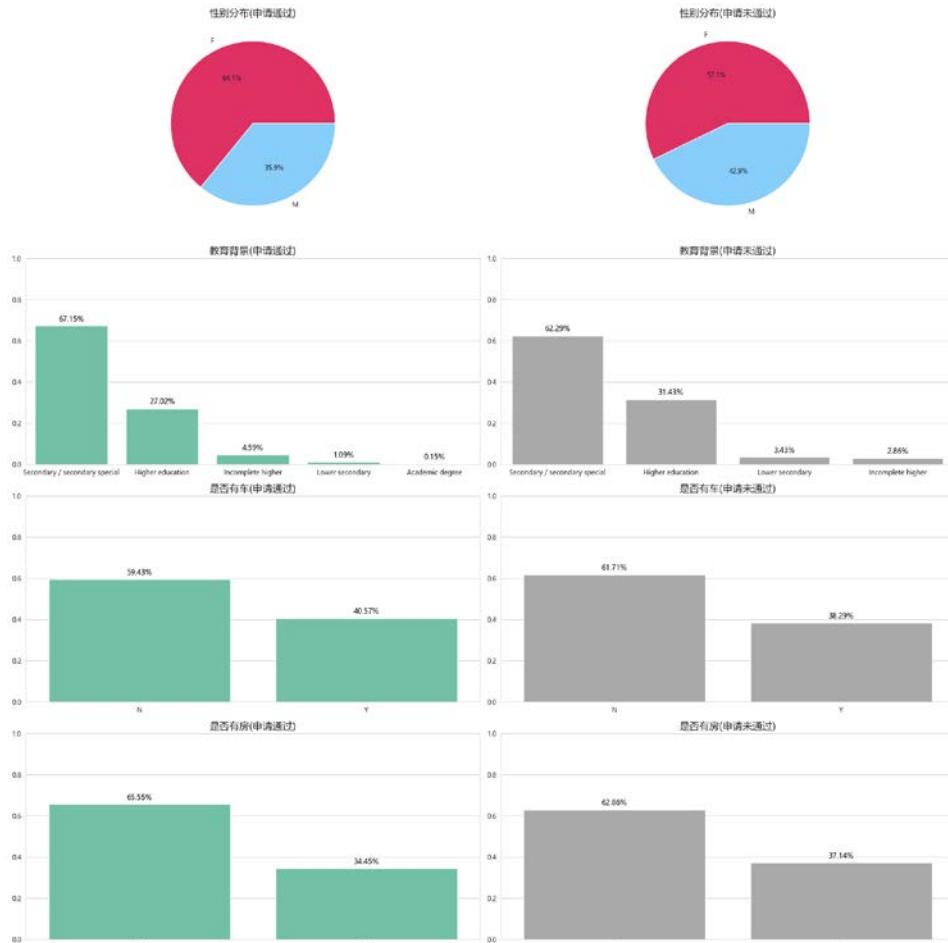
数据探索完成后，进行用户画像，根据 label 值分别筛选出通过的数据与不通过的数据，并进行对比分析。以通过申请的用户画像为例：

```

1. approved_data = data[data['label'] == 0] #通过申请的用户数据
2.
3. approved_profile = {
4.     'Gender Distribution': approved_data['GENDER'].value_counts(normalize=True),
5.     'Average Annual Income': approved_data['Annual_income'].mean(),
6.     'Education Distribution': approved_data['EDUCATION'].value_counts(normalize=True),
7.     'Car Ownership Distribution': approved_data['Car_Owner'].value_counts(normalize=True),
8.     'Property Ownership Distribution': approved_data['Propert_Owner'].value_counts(normalize=True),
9.     'Average Age': approved_data['Age'].mean(),
10.    'Occupation Distribution': approved_data['Type_Occupation'].value_counts(normalize=True),
11.    'Marital Status Distribution': approved_data['Marital_status'].value_counts(normalize=True)
12. }
13. print(approved_profile)

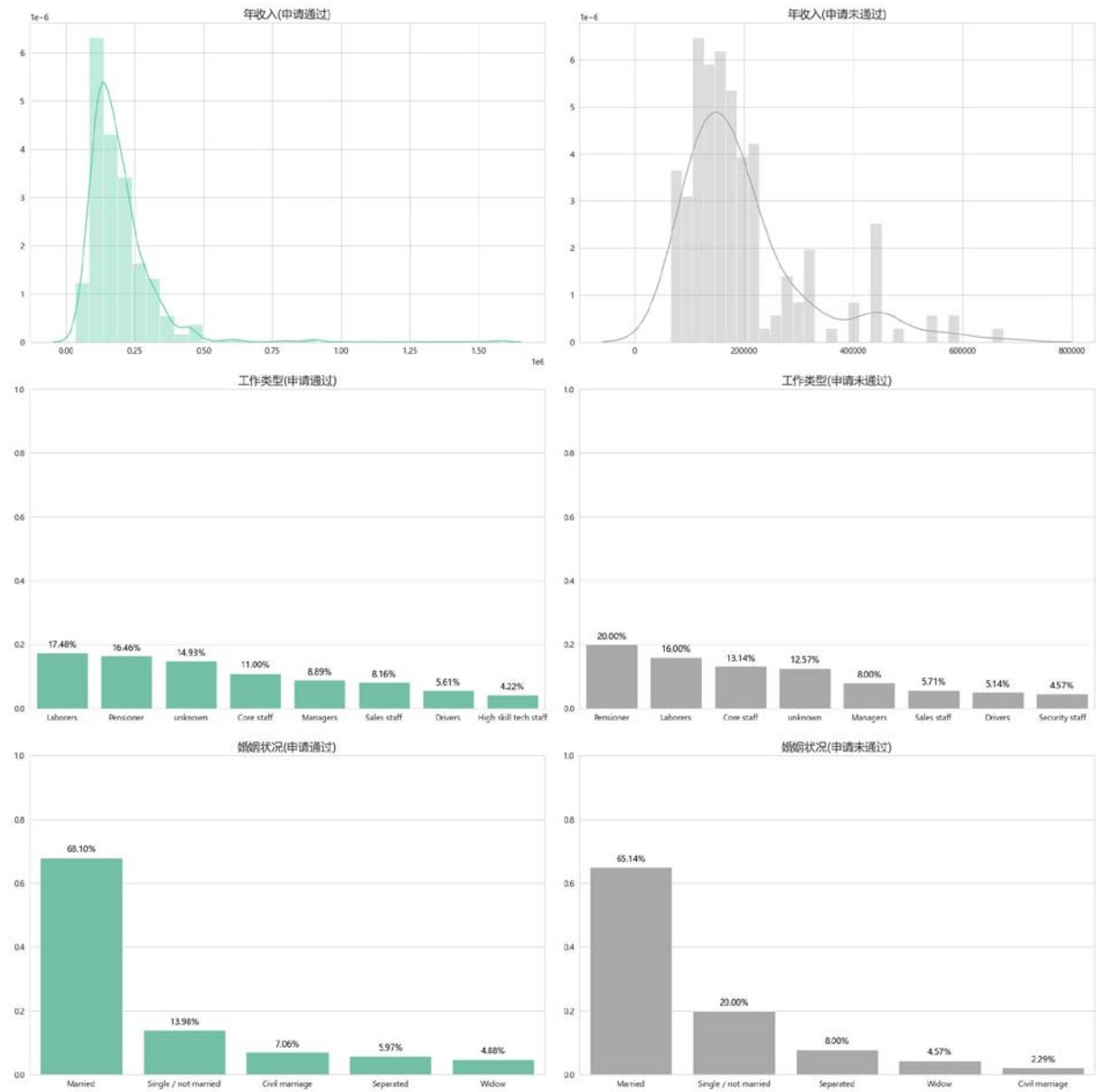
```

输出过长，在报告中不再展示，详情可见 jupyter notebook。提取数据后可视化对比



从上图可以看出：

- ◆ 女性申请者相较于男性申请者更加容易通过；
- ◆ 有房、有车的申请者相较于无房、无车的申请者更加容易通过，不过差距不大；
- ◆ 另外一个有意思的是，Academic degree 教育背景的申请者全部通过，这可能与样本资料的限制有关。



从比较中可以看出，

- ◆ 年收入高的人群，相反可能不会通过申请，猜测原因可能是高收入人群的风险也相对较高；
- ◆ 退休人士不通过的概率较高；
- ◆ Civil marriage(世俗婚姻)人群在申请通过样本中的占比相较于在申请未通过的样本中更高。

下面进行模型训练。在训练前，对部分特征进行处理，以方便后续模型的训练。具体有：

- 删除 Ind_ID;
- 将所有的 Y 替换成 1，所有的 N 替换成 0;
- 将 M(男)替换成 0，将 F(女)替换成 1;
- 将教育情况列改为有序编码;
- 对分类数据独热编码;
- 对连续变量进行标准化。

```
1. data.drop(columns=['Ind_ID'], axis=1, inplace=True)
2.
3. data.replace({'Y': 1, 'N': 0}, inplace=True)
4.
5. data.replace({'M': 0, 'F': 1}, inplace=True)
6.
7. education_mapping = {
8.     'Lower secondary': 0,
9.     'Secondary / secondary special': 1,
10.    'Incomplete higher': 2,
11.    'Higher education': 3,
12.    'Academic degree': 4
13. }
14. data['EDUCATION'] = data['EDUCATION'].map(education_mapping)
15.
16. categorical_cols = data.select_dtypes(include=['object']).columns
17. new_data = pd.get_dummies(data, columns=categorical_cols)
18.
19. numerical_cols = ['Annual_income', 'Age', 'Employed_years', 'Family_Members', 'CHILDREN', 'EDUCATION']
20. scaler = StandardScaler()
21. new_data[numerical_cols] = scaler.fit_transform(new_data[numerical_cols])
22.
23. x = new_data.drop('label', axis=1)
24. y = new_data['label']
25.
26. # 划分训练集与测试集
27. x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=1017, stratify=y)
```

由于之前已经发现，样本中类别不平衡，label=0 的样本远多于 label=1 的样本，又因为样本数量较少，因此考虑过采样方法使样本平衡。过采样是根据样本标签少的样本

的规律去生成更多该标签样本，这样使得数据趋向于平衡。

```

1. x_min= x_train[y_train == 1]
2. y_min = y_train[y_train == 1]
3. x_maj = x_train[y_train == 0]
4. y_maj = y_train[y_train == 0]
5.
6. x_min_resampled = resample(x_min, replace=True, n_samples=len(x_maj), random_state=1
                               017)
7. y_min_resampled = resample(y_min, replace=True, n_samples=len(y_maj), random_state=1
                               017)
8. x_train = pd.concat([x_maj, x_min_resampled])
9. y_train = pd.concat([y_maj, y_min_resampled])

```

欲使用集成学习方法，先尝试进行基础模型的训练，观察各模型效果。

9.3.1 逻辑回归

```

1. logreg = LogisticRegression(random_state=1017)
2. logreg.fit(x_train, y_train)
3.
4. y_pred = logreg.predict(x_test)
5. result_rep = classification_report(y_test, y_pred)
6. print(result_rep)

```

	precision	recall	f1-score	support
0	0.90	0.68	0.77	412
1	0.14	0.42	0.21	53
<hr/>				
accuracy			0.65	465
macro avg	0.52	0.55	0.49	465
weighted avg	0.81	0.65	0.71	465

逻辑回归模型评分如下：

1. 精确度：对于类别 0，精确度为 0.90，对于类别 1，精确度为 0.14
2. 召回率：对于类别 0，召回率为 0.68，对于类别 1，召回率为 0.42
3. F1 得分：对于类别 0，F1 得分为 0.77，对于类别 1，F1 得分为 0.21
4. 总体准确率为 0.65，模型效果一般

```

1. #绘制混淆矩阵
2. cm = confusion_matrix(y_test, y_pred)
3. plt.figure(figsize=(8, 6))
4. sns.heatmap(cm, annot=True, fmt='g', cmap='Blues',
5.               xticklabels=['Predicted 0', 'Predicted 1'],

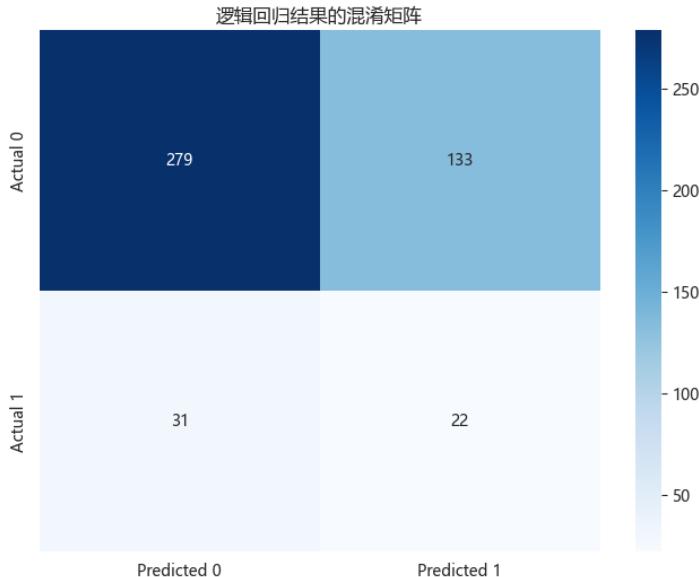
```

```

6.      yticklabels=['Actual 0', 'Actual 1'])
7.  plt.title('逻辑回归结果的混淆矩阵')
8.  plt.show()

```

绘制混淆矩阵如图：



9.3.2 支持向量机(SVM)

```

1.  svm_clf = SVC(kernel='rbf', probability=True, random_state=1017)
2.  svm_clf.fit(x_train, y_train)
3.
4.  y_pred_svm = svm_clf.predict(x_test)
5.  rep_svm = classification_report(y_test, y_pred_svm)
6.  print(rep_svm)

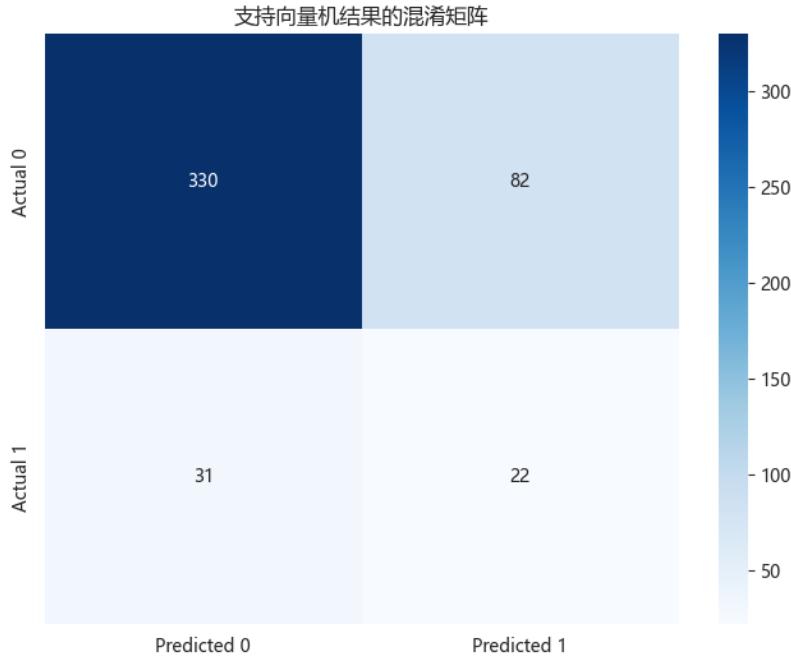
```

	precision	recall	f1-score	support
0	0.91	0.80	0.85	412
1	0.21	0.42	0.28	53
accuracy			0.76	465
macro avg	0.56	0.61	0.57	465
weighted avg	0.83	0.76	0.79	465

支持向量机模型评分如下：

1. 精确度：对于类别 0，精确度为 0.91，对于类别 1，精确度为 0.21
2. 召回率：对于类别 0，召回率为 0.80，对于类别 1，召回率为 0.42
3. F1 得分：对于类别 0，F1 得分为 0.85，对于类别 1，F1 得分为 0.28
4. 总体准确率为 0.76，模型效果一般，但优于逻辑回归模型

绘制混淆矩阵如图：



9.3.3 决策树

```

1. tree_clf = tree.DecisionTreeClassifier(random_state=1017)
2. tree_clf.fit(x_train,y_train)
3.
4. y_pred_tree = tree_clf.predict(x_test)
5. rep_tree = classification_report(y_test, y_pred_tree)
6. print(rep_tree)

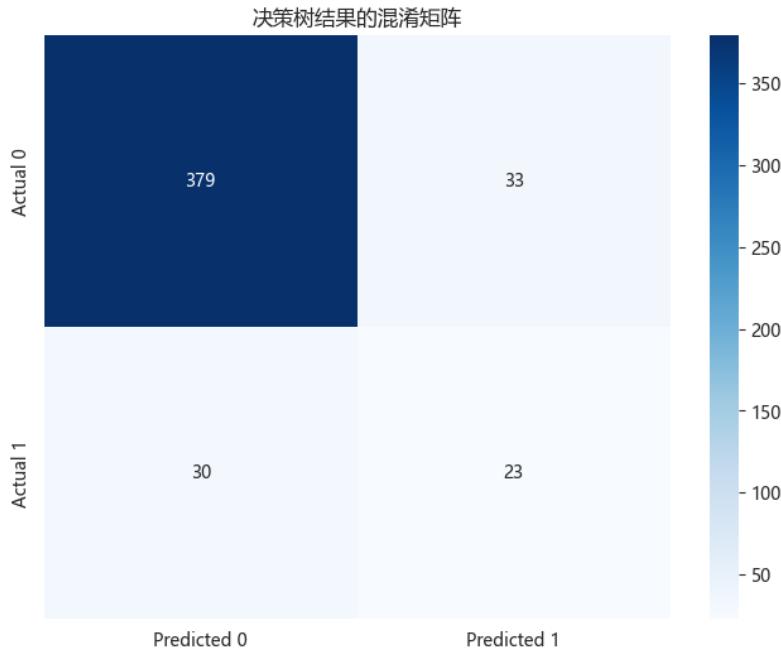
```

	precision	recall	f1-score	support
0	0.93	0.92	0.92	412
1	0.41	0.43	0.42	53
accuracy			0.86	465
macro avg	0.67	0.68	0.67	465
weighted avg	0.87	0.86	0.87	465

决策树模型评分如下：

- 精确度：对于类别 0，精确度为 0.93，对于类别 1，精确度为 0.41
- 召回率：对于类别 0，召回率为 0.92，对于类别 1，召回率为 0.43
- F1 得分：对于类别 0，F1 得分为 0.92，对于类别 1，F1 得分为 0.42
- 总体准确率为 0.86，模型效果良好，优于逻辑回归模型和支持向量机模型

绘制混淆矩阵如图：



9.3.4 贝叶斯

```

1. bys_clf = GaussianNB()
2. bys_clf.fit(x_train, y_train)
3.
4. y_pred_bys = bys_clf.predict(x_test)
5. rep_bys = classification_report(y_test, y_pred_bys)
6. print(rep_bys)

```

	precision	recall	f1-score	support
0	0.90	0.07	0.13	412
1	0.12	0.94	0.21	53
accuracy			0.17	465
macro avg	0.51	0.51	0.17	465
weighted avg	0.81	0.17	0.14	465

贝叶斯分类器评分如下：

1. 精确度：对于类别 0，精确度为 0.9，对于类别 1，精确度为 0.12
2. 召回率：对于类别 0，召回率为 0.07，对于类别 1，召回率为 0.94
3. F1 得分：对于类别 0，F1 得分为 0.13，对于类别 1，F1 得分为 0.21
4. 总体准确率为 0.17，模型效果极差

且从精确度（查准率）和召回率（查全率）可以看出，模型判断正类负类的阈值偏低，导致正类（1）的查准率小而查全率大。

9.3.5 神经网络

```

1. nw_clf = MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=200, random_state=1017
    )
2. nw_clf.fit(x_train, y_train)
3.
4. y_pred_nw = nw_clf.predict(x_test)
5. rep_nw = classification_report(y_test, y_pred_nw)
6. print(rep_nw)

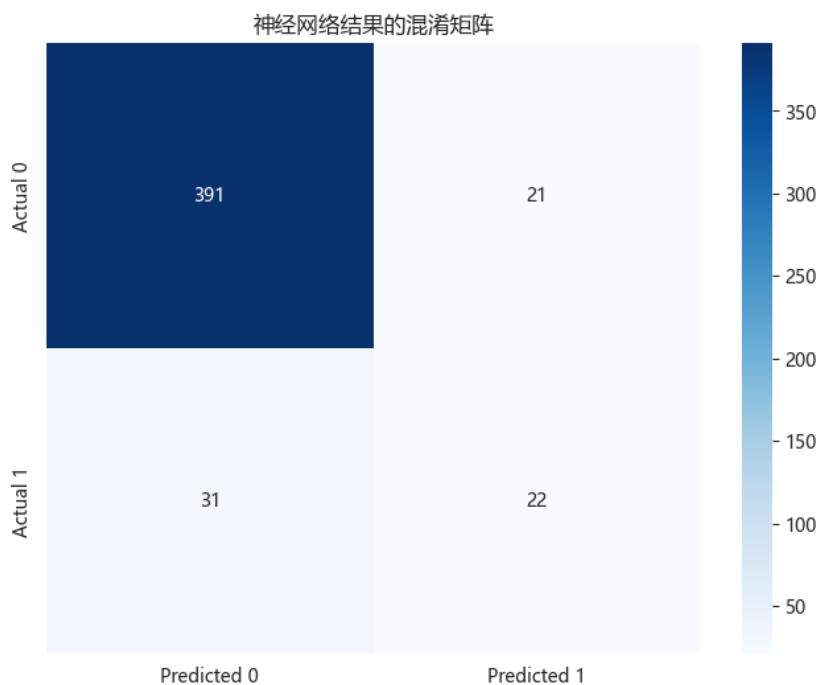
```

	precision	recall	f1-score	support
0	0.93	0.95	0.94	412
1	0.51	0.42	0.46	53
accuracy			0.89	465
macro avg	0.72	0.68	0.70	465
weighted avg	0.88	0.89	0.88	465

神经网络模型评分如下：

1. 精确度：对于类别 0，精确度为 0.93，对于类别 1，精确度为 0.51
2. 召回率：对于类别 0，召回率为 0.95，对于类别 1，召回率为 0.42
3. F1 得分：对于类别 0，F1 得分为 0.94，对于类别 1，F1 得分为 0.46
4. 总体准确率为 0.89，模型效果较好，目前为止最优

绘制混淆矩阵如图：



9.3.6 stacking

堆叠法 Stacking 是近年来模型融合领域最为热门的方法。作为强学习器的融合方法，Stacking 集模型效果好、可解释性强、适用复杂数据三大优点于一身，属于融合领域最为实用的先驱方法。

它的核心思想其实非常简单。首先，如下图所示，Stacking 结构中有两层算法串联，第一层叫做 level 0，第二层叫做 level 1，level 0 里面可能包含 1 个或多个强学习器，而 level 1 只能包含一个学习器。在训练中，数据会先被输入 level 0 进行训练，训练完毕后，level 0 中的每个算法会输出相应的预测结果。我们将这些预测结果拼凑成新特征矩阵，再输入 level 1 的算法进行训练。融合模型最终输出的预测结果就是 level 1 的学习器输出的结果。



根据 stacking 算法的思想，编写代码如下，将 stacking 算法编成一个类，便于使用。

```

1. class StackingClassifier(BaseEstimator, ClassifierMixin, TransformerMixin):
2.     def __init__(self, base_models, meta_model):
3.         self.base_models = base_models
4.         self.meta_model = meta_model
5.
6.     def fit(self, X, y):
7.         self.base_models_ = [clone(model) for model in self.base_models]
8.         self.meta_model_ = clone(self.meta_model)
9.
10.        # 训练基础模型
11.        for model in self.base_models_:
12.            model.fit(X, y)
13.
14.        # 生成基础模型的预测结果作为元特征
15.        meta_features = np.column_stack([
16.            model.predict(X) for model in self.base_models_
17.        ])

```

```
18.  
19.     # 训练元模型  
20.     self.meta_model_.fit(meta_features, y)  
21.  
22.     return self  
23.  
24. def predict(self, X):  
25.     meta_features = np.column_stack([  
26.         model.predict(X) for model in self.base_models_  
27.     ])  
28.     return self.meta_model_.predict(meta_features)
```

`__init__` 函数用于初始化，StackingClassifier 被初始化为两个参数：base_models 和 meta_model。base_models 是一个包含作为基础模型的机器学习模型的列表。meta_model 是一个单一的机器学习模型，将用作元模型。

`fit` 方法负责训练堆叠集成。它克隆基础模型和元模型，以避免修改原始实例。然后，在输入数据 X 和标签 y 上训练每个基础模型。基础模型在输入数据上的预测被水平堆叠以形成元特征，元模型在这些元特征和原始标签上进行训练。

`predict` 方法接受新的输入数据，它计算每个基础模型在输入数据上的预测，并将这些预测水平堆叠。然后，元模型基于这些堆叠的元特征进行最终预测。

① 基模型：逻辑回归+随机森林+SVM 元模型：逻辑回归

```
1. # 定义基础模型  
2. base_models = [  
3.     LogisticRegression(),  
4.     RandomForestClassifier(n_estimators=10, random_state=1017),  
5.     SVC()  
6. ]  
7.  
8. # 定义元模型  
9. meta_model = LogisticRegression()  
10.  
11. # 定义 Stacking 模型  
12. stacking = StackingClassifier(base_models, meta_model)  
13.  
14. # 拟合数据  
15. stacking.fit(x_train, y_train)  
16.  
17. # 预测数据  
18. y_pred = stacking.predict(x_test)  
19.  
20. rep_sta = classification_report(y_test, y_pred)  
21. print(rep_sta)
```

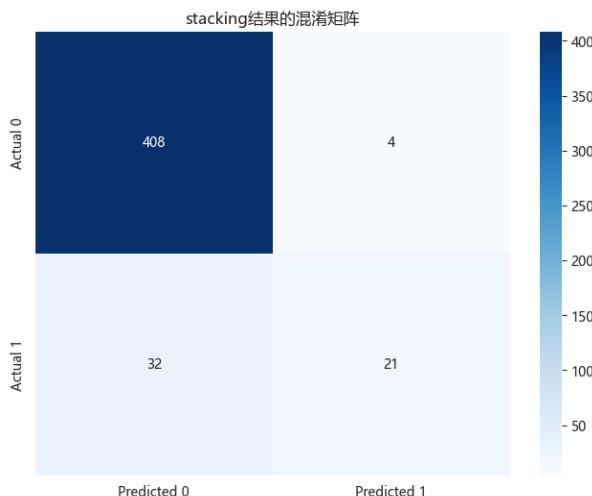
	precision	recall	f1-score	support
0	0.93	0.99	0.96	412
1	0.81	0.40	0.53	53
accuracy			0.92	465
macro avg	0.87	0.69	0.74	465
weighted avg	0.91	0.92	0.91	465

② 基模型：神经网络+随机森林+SVM 元模型：逻辑回归

使用方法与上面类似，输出为

	precision	recall	f1-score	support
0	0.93	0.99	0.96	412
1	0.84	0.40	0.54	53
accuracy			0.92	465
macro avg	0.88	0.69	0.75	465
weighted avg	0.92	0.92	0.91	465

绘制混淆矩阵：



可见，stacking 的结果较好，模型效果得到了提升。特别是在预测“不通过”类别上的准确率提升较大。基模型为神经网络+随机森林+SVM 和元模型为逻辑回归的组合较好，可能是因为 level 0 上的算法的职责是找出原始数据与标签的关系，即建立原始数据与标签之间的假设，故需要强大的学习能力。但 level 1 上的算法的职责是融合个体学习器做出的假设，并最终输出融合模型的结果，相当于在寻找“最佳融合规则”，而非直接建立原始数据与标签之间的假设。神经网络相较于逻辑回归具有更强的学习能力。

9.3.7 voting

对分类任务来说，学习器 h_i 将从类别标记集合 $\{c_1, c_2, \dots, c_N\}$ 中预测出一个标记，最常见的结合策略是使用投票(voting)。为便于讨论，我们将 h_i 在样本 \mathbf{x} 上的预测输出表示为一个 N 维向量 $(h_i^1(\mathbf{x}); h_i^2(\mathbf{x}), \dots, h_i^N(\mathbf{x}))$ ，其中 $h_i^j(\mathbf{x})$ 是 h_i 在类别 c_j 上的输出。以相对多数投票法为例，

$$H(\mathbf{x}) = c_{\arg \max_j} \sum_{i=1}^T h_i^j(\mathbf{x}).$$

即预测为得票最多的标记，若同时有多个标记获最高票，则从中随机选取一个。

在 Stacking 堆叠法中，我们用算法融合强学习器的结果。在投票法中，我们用投票方式融合强学习器的结果，这与 Stacking 的流程非常类似：

投票法Voting



① 三模型 voting: SVM+决策树+神经网络

```

1. three_predictions = np.array([y_pred_svm, y_pred_tree, y_pred_nw])
2. three_pred = np.apply_along_axis(lambda x: np.argmax(np.bincount(x)), axis=0, arr=three_predictions)
3.
4. rep_three = classification_report(y_test, three_pred)
5. print(rep_three)
    
```

	precision	recall	f1-score	support
0	0.93	0.94	0.93	412
1	0.48	0.43	0.46	53
accuracy			0.88	465
macro avg	0.70	0.69	0.69	465
weighted avg	0.88	0.88	0.88	465

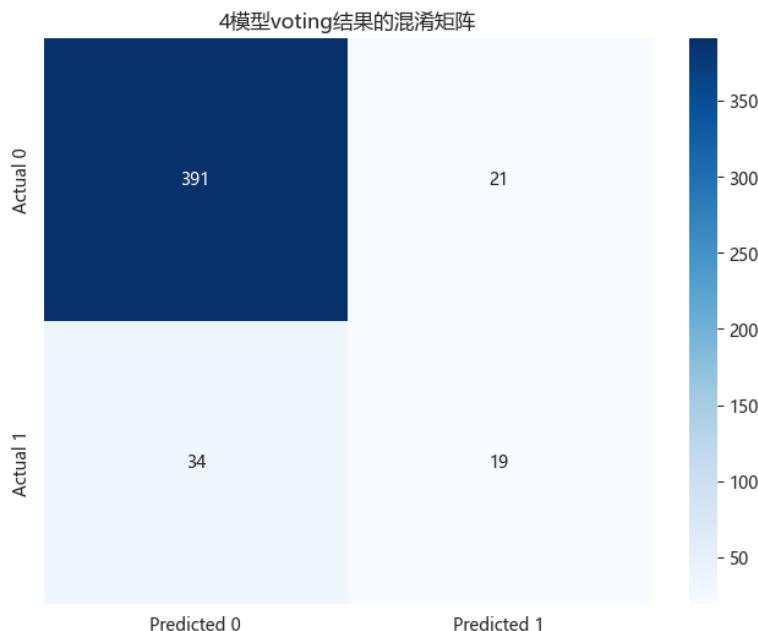
② 四模型 voting:

```

1. four_predictions = np.array([y_pred, y_pred_svm, y_pred_tree, y_pred_nw])
2. four_pred = np.apply_along_axis(lambda x: np.argmax(np.bincount(x)), axis=0, arr=four_predictions)
3.
4. rep_four = classification_report(y_test, four_pred)
5. print(rep_four)
    
```

	precision	recall	f1-score	support
0	0.92	0.95	0.93	412
1	0.47	0.36	0.41	53
accuracy			0.88	465
macro avg	0.70	0.65	0.67	465
weighted avg	0.87	0.88	0.87	465

绘制混淆矩阵图：



可以看到 voting 后模型的效果不升反降，特别是在不通过类型的识别上表现较差。模型总体效果不如 stacking。

9.3.8 Bagging

Bagging 集成算法的基本原理在之前求解手写数字识别问题时已经有过介绍，此处不再赘述。下面尝试自己编写 Bagging 集成算法：

```

1. class Bagging:
2.     def __init__(self, base_estimator, n_estimators):
3.         self.base_estimator = base_estimator
4.         self.n_estimators = n_estimators
5.         self.estimators = []
6.         self.selected_features = []
7.
8.     def fit(self, X, y):
9.         for _ in range(self.n_estimators):
10.             # 从原始训练集中有放回地抽样生成子训练集

```

```
11.         indices = np.random.choice(len(X), size=900, replace=True)
12.         X = np.array(X)
13.         y = np.array(y)
14.         X_subset = X[indices]
15.         y_subset = y[indices]
16.
17.         # 随机选择特征
18.         num_features = X.shape[1]
19.         selected_features = np.random.choice(num_features, size=30, replace=False)
20.         self.selected_features.append(selected_features)
21.         X_subset = X_subset[:, selected_features]
22.
23.         # 训练学习器并加入到集成模型中
24.         estimator = clone(self.base_estimator)
25.         estimator.fit(X_subset, y_subset)
26.         self.estimators.append(estimator)
27.
28.     def predict(self, X):
29.         predictions = np.zeros((len(X), self.n_estimators))
30.         for i, estimator in enumerate(self.estimators):
31.             X_subset = X.iloc[:, self.selected_features[i]]
32.             # 使用每个学习器进行预测
33.             predictions[:, i] = estimator.predict(X_subset)
34.
35.         # 使用投票策略集成预测结果
36.         ensemble_predictions = np.apply_along_axis(
37.             lambda x: np.argmax(np.bincount(x.astype(int))),
38.             axis=1,
39.             arr=predictions
40.         )
41.         return ensemble_predictions
```

`__init__`方法用于初始化。其中 `base_estimator` 用于初始化基学习器，即每个子模型所采用的学习算法。`n_estimators` 用于初始化集成学习器中包含的基学习器数量。`estimators` 用于初始化一个空列表，用于存储训练好的基学习器。`selected_features` 用于初始化一个空列表，用于存储每个基学习器所选择的特征集合。

`fit` 方法用于训练。通过循环 `n_estimators` 次，在原始训练集上进行有放回抽样，生成子训练集。针对每个子训练集，随机选择 `n`(此处 `n` 为 30)个特征进行训练，这些特征存储在 `selected_features` 中。值得注意的是，`n` 的设置是为了使用随机森林的方便。当 `n` 等于特征数量时，为经典的 bagging 方法；若小于特征数量，则可实现随机选取特征，当基学习器为决策树时即为随机森林算法。训练好的基学习器存储在 `estimators` 中。

predict 方法用于训练。初始化一个数组 `predictions`, 用于存储每个基学习器的预测结果。针对每个基学习器, 从输入数据中选择相应的特征, 进行预测, 结果存储在 `predictions` 中。最后, 使用投票策略, 对每个样本的多个基学习器的预测结果进行投票, 得到最终的集成预测结果。

① 随机森林算法: 决策树 bagging+随机特征选择 (30)

```

1. base_clf = tree.DecisionTreeClassifier()
2. bagging = Bagging(base_clf, n_estimators=100)
3.
4. bagging.fit(x_train, y_train)
5. y_pred_bag = bagging.predict(x_test)
6. rep_bag = classification_report(y_test, y_pred_bag)
7. print(rep_bag)

```

	precision	recall	f1-score	support
0	0.93	0.98	0.95	412
1	0.69	0.42	0.52	53
accuracy				
macro avg	0.81	0.70	0.73	465
weighted avg	0.90	0.91	0.90	465

随机森林的效果良好, 对申请不通过的区分能力有了不少的提升。

使用 sklearn 包中的随机森林:

	precision	recall	f1-score	support
0	0.93	0.99	0.96	412
1	0.84	0.40	0.54	53
accuracy				
macro avg	0.88	0.69	0.75	465
weighted avg	0.92	0.92	0.91	465

Sklearn 包中的随机森林模型的效果很好, 与自己编写的随机森林算法训练出的模型相比表现更优!

② 支持向量机 bagging (选择特征数=特征数 49)

设置基分类器为支持向量机模型, 基分类器个数设置为 10 个, 选取的特征数为 49, 即所有特征。

```

1. base_clf = SVC(kernel='rbf', probability=True)
2. bagging_svm = Bagging(base_clf, n_estimators=10)
3.
4. bagging_svm.fit(x_train, y_train)
5. y_pred_bag_svm = bagging_svm.predict(x_test)
6. rep_bag_svm = classification_report(y_test, y_pred_bag_svm)
7. print(rep_bag_svm)

```

	precision	recall	f1-score	support
0	0.91	0.80	0.85	412
1	0.20	0.38	0.26	53
accuracy				
macro avg	0.55	0.59	0.56	465
weighted avg	0.83	0.75	0.79	465

基于 SVM 的 bagging 效果并不好。

③ 神经网络 bagging

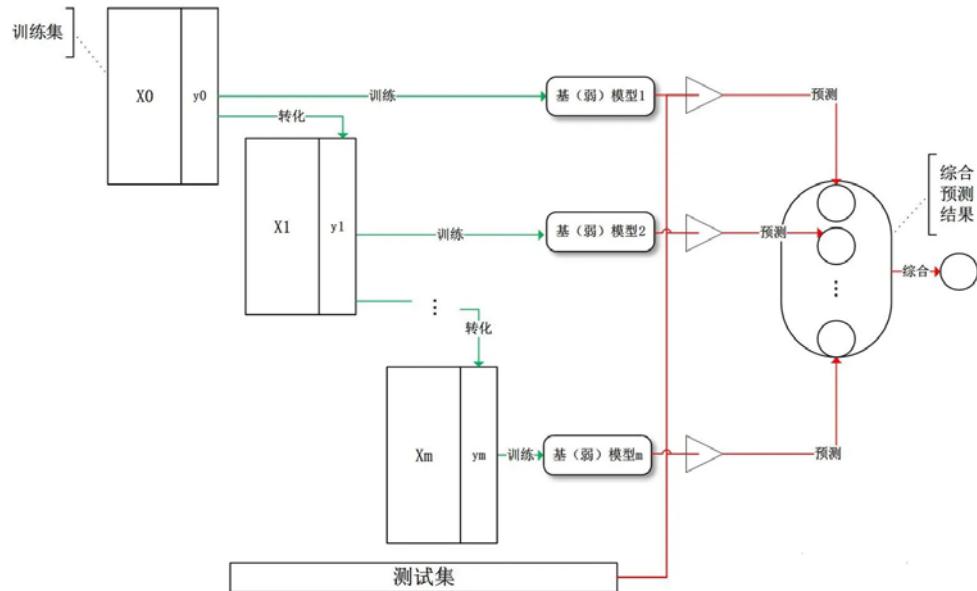
	precision	recall	f1-score	support
0	0.93	0.94	0.93	412
1	0.49	0.45	0.47	53
accuracy				
macro avg	0.71	0.70	0.70	465
weighted avg	0.88	0.88	0.88	465

与 9.3.5 单个神经网络模型对比，可以发现，神经网络做 bagging 的效果相较于单个神经网络模型并没有提升。

9.3.9 Boosting

Boosting 是一族可将弱学习器提升为强学习器的算法。这族算法的工作机制类似：先从初始训练集训练出一个基学习器，再根据基学习器的表现对训练样本进行调整，使得先前基学习器做错的训练样本在后续受到更多关注，然后基于调整后的样本分布来训练下一个基学习器；如此重复进行，直至基学习器数目达到事先指定的值 T ，最终将这 T 个学习器进行加权结合。

Boosting 的训练过程为阶梯状，基模型按次序一一进行训练，基模型的训练集按照某种策略每次都进行一定的转化，大致流程如图：



Boosting 族算法最著名的是 AdaBoost，其伪代码描述如下：

输入: 训练集 $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$;
 基学习算法 \mathcal{L} ;
 训练轮数 T .

过程:

- 1: $\mathcal{D}_1(\mathbf{x}) = 1/m$.
- 2: **for** $t = 1, 2, \dots, T$ **do**
- 3: $h_t = \mathcal{L}(D, \mathcal{D}_t)$;
- 4: $\epsilon_t = P_{\mathbf{x} \sim \mathcal{D}_t}(h_t(\mathbf{x}) \neq f(\mathbf{x}))$;
- 5: **if** $\epsilon_t > 0.5$ **then break**
- 6: $\alpha_t = \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$;
- 7: $\mathcal{D}_{t+1}(\mathbf{x}) = \frac{\mathcal{D}_t(\mathbf{x})}{Z_t} \times \begin{cases} \exp(-\alpha_t), & \text{if } h_t(\mathbf{x}) = f(\mathbf{x}) \\ \exp(\alpha_t), & \text{if } h_t(\mathbf{x}) \neq f(\mathbf{x}) \end{cases}$
 $= \frac{\mathcal{D}_t(\mathbf{x}) \exp(-\alpha_t f(\mathbf{x}) h_t(\mathbf{x}))}{Z_t}$
- 8: **end for**

输出: $H(\mathbf{x}) = \text{sign} \left(\sum_{t=1}^T \alpha_t h_t(\mathbf{x}) \right)$

自己编写 AdaBoost 算法：

```

1. class Boosting:
2.     def __init__(self, base_classifier, n_estimators=5):
3.         self.n_estimators = n_estimators
4.         self.models = []
5.         self.weights = []
6.         self.base_classifier = base_classifier
7.
8.     def fit(self, X, y):
9.         n_samples, n_features = X.shape
10.        # 初始化样本权重
11.        sample_weights = np.ones(n_samples) / n_samples
12.

```

```

13.     for i in range(self.n_estimators):
14.         model = clone(self.base_classifier)
15.         # 使用带权重的样本进行训练
16.         model.fit(X, y, sample_weight=sample_weights)
17.
18.         predictions = model.predict(X)
19.
20.         error = np.sum(sample_weights * (predictions != y)) / np.sum(sample_weights)
21.         model_weight = 0.5 * np.log((1 - error) / error)
22.
23.         # 更新样本权重
24.         sample_weights *= np.exp(-model_weight * y * predictions)
25.         sample_weights /= np.sum(sample_weights)
26.
27.         # 保存模型和权重
28.         self.models.append(model)
29.         self.weights.append(model_weight)
30.
31.     def predict(self, X):
32.         # 对每个模型的预测结果进行加权平均
33.         predictions = np.zeros(X.shape[0])
34.         for model, weight in zip(self.models, self.weights):
35.             predictions += weight * model.predict(X)
36.
37.         # 使用符号函数将结果映射到二分类标签
38.         return np.sign(predictions).astype(int)

```

`__init__(self, base_classifier, n_estimators=5)`方法是类的构造函数，用于初始化 Boosting 算法的实例。`base_classifier` 是基分类器，是提升算法的核心组成部分。`n_estimators` 是提升算法中弱学习器的迭代次数，默认为 5。`self.n_estimators`、`self.models` 和 `self.weights` 分别用于存储迭代次数、存储训练好的模型和其权重的列表。

`fit(self, X, y)`方法用于训练提升模型，即迭代地训练弱学习器，并更新样本权重。`X` 是输入特征，`y` 是对应的标签。在每次迭代中，基分类器被克隆，并使用带有当前样本权重的数据进行训练。然后计算基分类器的带权错误率，再计算模型权重，最后根据 AdaBoost 的权重更新公式更新样本权重。训练好的模型和其权重被保存在 `self.models` 和 `self.weights` 列表中。

`predict(self, X)`方法用于对输入数据进行预测。对每个训练好的模型，使用其权重对输入数据进行预测，并将预测结果进行加权平均。最终的预测结果通过将加权预测的总和应用符号函数（sign function），将结果映射到二进制标签，返回最终的预测结果。

① 决策树 boosting

```

1. base_clf = tree.DecisionTreeClassifier()
2. boost_tree = Boosting(base_clf, 10)
3. boost_tree.fit(x_train, y_train)
4.
5. y_pred_boost_tr = boost_tree.predict(x_test)
6. rep_boost_tr = classification_report(y_test, y_pred_boost_tr)
7. print(rep_boost_tr)

```

	precision	recall	f1-score	support
0	0.93	0.75	0.83	412
1	0.23	0.58	0.33	53
accuracy			0.73	465
macro avg	0.58	0.67	0.58	465
weighted avg	0.85	0.73	0.77	465

调用 sklearn 包中的 AdaBoost 算法

	precision	recall	f1-score	support
0	0.90	0.69	0.78	412
1	0.15	0.42	0.22	53
accuracy			0.66	465
macro avg	0.52	0.55	0.50	465
weighted avg	0.82	0.66	0.72	465

可以发现，无论是自己编写的决策树 boosting 集成算法还是调用 sklearn 包中的 AdaBoost，模型都未取得良好的效果。

② 支持向量机 Boosting

	precision	recall	f1-score	support
0	0.93	0.45	0.61	412
1	0.14	0.72	0.24	53
accuracy			0.48	465
macro avg	0.54	0.59	0.42	465
weighted avg	0.84	0.48	0.57	465

对支持向量机使用 Boosting 集成方法效果极差。

③ 调用 xgboost

```

1. xgb_clf = xgb.XGBClassifier(random_state=1017, use_label_encoder=False, eval_metric=
      'logloss')
2. xgb_clf.fit(x_train, y_train)
3.
4. y_pred_xgb = xgb_clf.predict(x_test)
5. class_report_xgb = classification_report(y_test, y_pred_xgb)
6. print(class_report_xgb)

```

	precision	recall	f1-score	support
0	0.93	0.96	0.95	412
1	0.60	0.45	0.52	53
accuracy			0.90	465
macro avg	0.77	0.71	0.73	465
weighted avg	0.89	0.90	0.90	465

xgboost 模型的效果较好。

9.3.10 参数调整

从以上模型的混淆矩阵可以看出，随机森林模型、xgboost 模型、stacking 模型具有良好的效果，故使用网格搜索进行调参，这里直接调用 sklearn 包中的算法。

① 随机森林

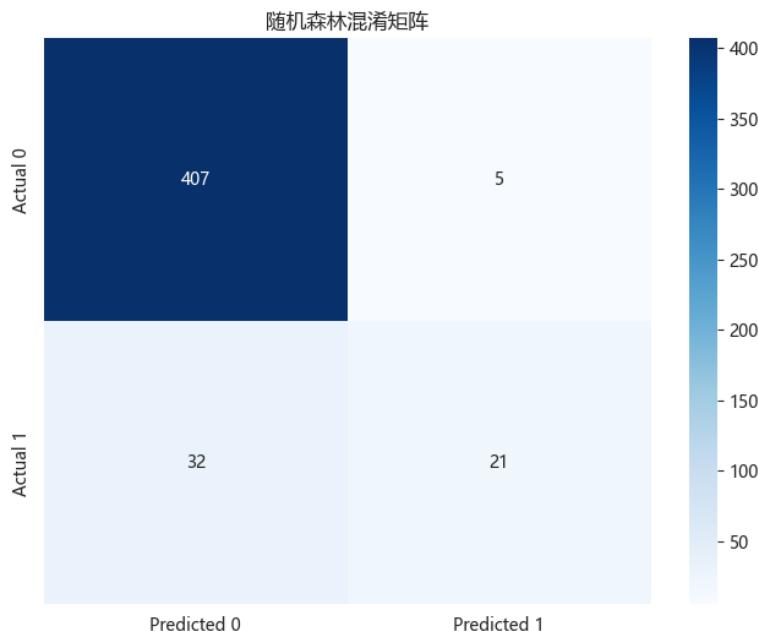
```

1. param_grid = {
2.     'n_estimators':[10,50,100], #决策树的数量
3.     'max_depth':[None,10,20,30],#决策树的最大深度
4.     'min_samples_split':[2,5,10],#决定节点分裂的最小样本数量
5.     'min_samples_leaf':[1,2,4],#决定叶节点的最小样本数量
6.     'max_features':['auto','sqrt','log2'] #考虑分裂时的特征数量
7. }
8.
9. rf = RandomForestClassifier()
10. grid_search = GridSearchCV(estimator=rf,param_grid=param_grid,cv=5,n_jobs=-1,verbose
     =1)
11. grid_search.fit(x_train,y_train)
12. print('Best parameters found:',grid_search.best_params_)
13.
14. best_rf = grid_search.best_estimator_
15. y_pred_orf = best_rf.predict(x_test)
16. class_report_orf = classification_report(y_test, y_pred_orf)
17. print(class_report_orf)

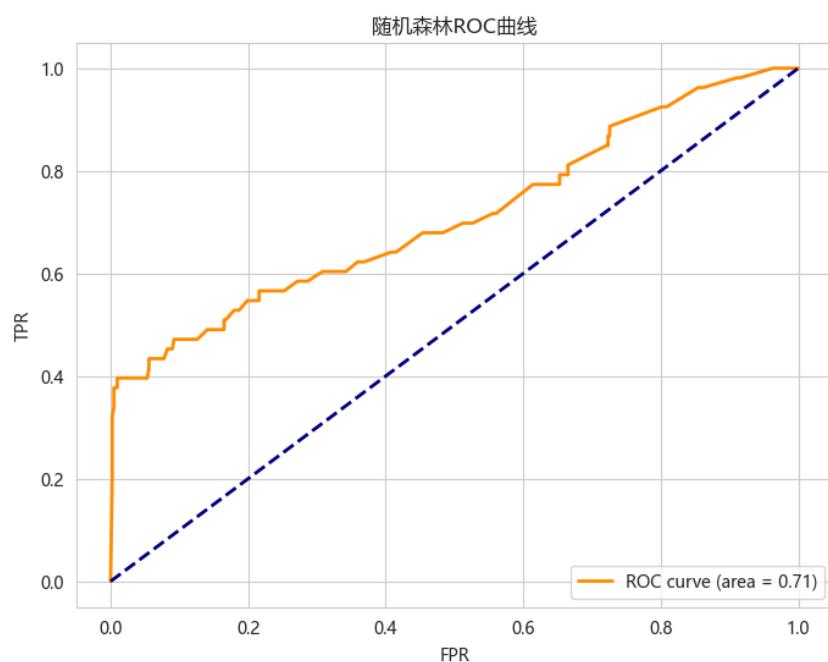
```

	precision	recall	f1-score	support
0	0.93	0.99	0.96	412
1	0.81	0.40	0.53	53
accuracy			0.92	465
macro avg	0.87	0.69	0.74	465
weighted avg	0.91	0.92	0.91	465

混淆矩阵图如下：



绘制 ROC 曲线



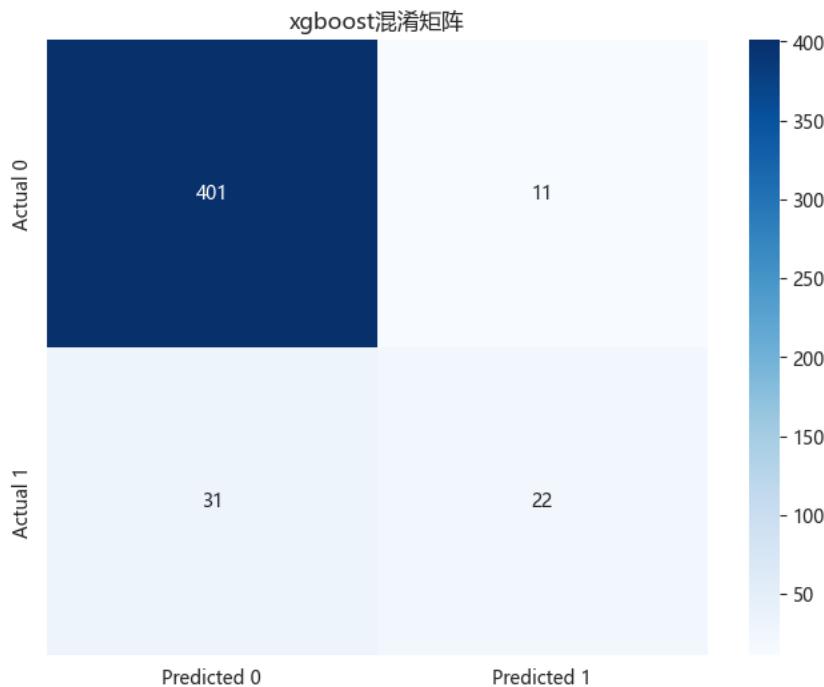
② xgboost

网格搜索的参数内容为

```
1. param_grid = {  
2.     'learning_rate':[0.01,0.05,0.1],  
3.     'n_estimators':[100,500,1000],  
4.     'max_depth':[3,5,10],  
5.     'min_child_weight':[1,3,5],  
6.     'gamma':[0,0.1,0.2],  
7.     'subsample':[0.8,1.0],  
8.     'colsample_bytree':[0.8,1.0],  
9.     'objective':['binary:logistic']  
10. }
```

混淆矩阵为

	precision	recall	f1-score	support
0	0.93	0.97	0.95	412
1	0.67	0.42	0.51	53
accuracy			0.91	465
macro avg	0.80	0.69	0.73	465
weighted avg	0.90	0.91	0.90	465



③ Stacking

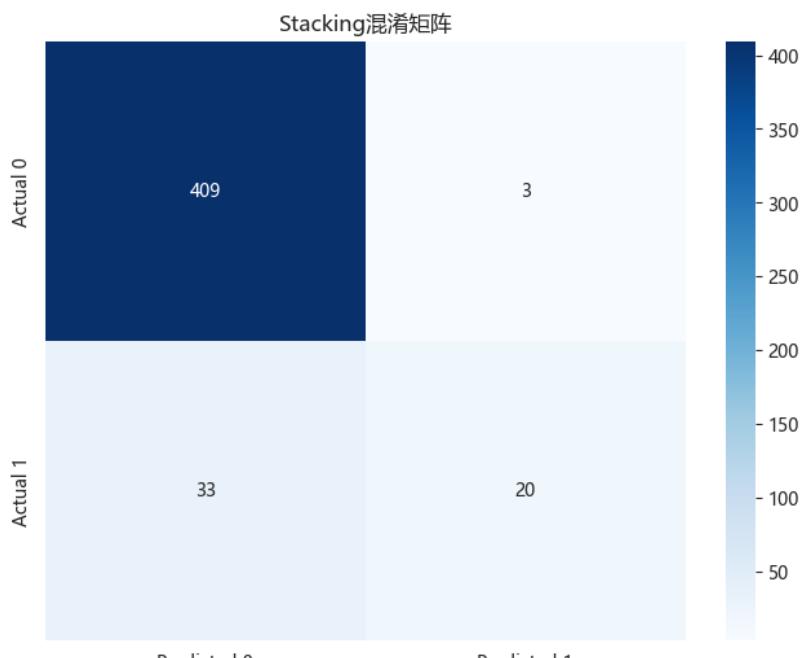
网格搜索内容为：

```

1. base_models = [
2.     ('mlp', MLPClassifier()),
3.     ('rf', RandomForestClassifier()),
4.     ('svm', SVC()),
5. ]
6. meta_model = LogisticRegression()
7.
8. stacking_model = StackingClassifier(estimators=base_models, final_estimator=meta_mod
e1)
9. # 定义参数网格
10. param_grid = {
11.     'rf__n_estimators': [10, 50],
12.     'svm__kernel': ['linear', 'rbf'],
13. }
14. grid_search = GridSearchCV(stacking_model, param_grid, cv=5, scoring='accuracy')
15. grid_search.fit(x_train, y_train)

```

	precision	recall	f1-score	support
0	0.93	0.99	0.96	412
1	0.87	0.38	0.53	53
accuracy			0.92	465
macro avg	0.90	0.69	0.74	465
weighted avg	0.92	0.92	0.91	465



综合来看，随机森林模型和 Stacking 模型整体表现较好。

```
1. rf_feature_importance = best_rf.feature_importances_
2. feature_names = x_train.columns
3. rf_feature_df = pd.DataFrame({
4.     'Feature': feature_names,
5.     'Importance': rf_feature_importance
6. })
7. sorted_rf_feature_df = rf_feature_df.sort_values(by='Importance', ascending=False).head(6) #筛选出重要特征
```

筛选出重要特征，结果如下：

	Feature	Importance
11	Age	0.157169
4	Annual_income	0.142510
13	Employed_years	0.128894
10	Family_Members	0.044000
5	EDUCATION	0.043111
3	CHILDREN	0.029716

在特征中，前 6 个重要特征为：Age>Annual_income>Employed_years>Family_Members>EDUCATION>CHILDREN，这对未来银行的信用识别具有一定的指导意义。

结束语

在此次机器学习实验的过程中，我深刻地体验到了这一领域的吸引力和挑战。通过尝试不同的算法并精心处理数据，我对模型的原理、训练和调优有了更为深刻的理解。我学到了如何处理真实世界的数据，以及如何根据问题的需求选择适当的模型。调整超参数、解决过拟合问题等一系列步骤使我更加熟悉机器学习的实际运用。

展望未来，我期待能够持续深入学习机器学习的各个方面，并将这些知识应用于更多的项目中。虽然实验已经结束，但学习之路仍在延伸，希望我能保持对新技术和方法的好奇心，不断探索、追求。

最后，感谢储老师一学期的辛勤教学，也感谢这次难得的实验机会，让我充分接触了机器学习。通过持续的实践和不断的尝试，我相信我将对这些算法和模型有更好的掌握，为未来的学术和职业生涯奠定坚实的基础。

参考文献

- [1] 周志华. 机器学习[M]. 北京:清华大学出版社,2016.
- [2] Peter Harrington. 机器学习实战[M]. 北京:人民邮电出版社,2013.
- [3] 谢文睿,秦州,贾彬彬. 机器学习公式详解 第2版[M]. 北京:人民邮电出版社,2023.