Task 1、2

推导闭式解、梯度下降解法（多维输入情况）

• 一维中的推导

我们以**一元线性回归**为例，目标是找到最优的 `w, b`，让下面的函数最好地拟合数据：

$$y = wx + b$$

我们有一组训练数据(xi,yi)，目标是最小化误差（损失函数）：

$$L(w,b) = \frac{1}{n}\sum_{i=1}^{n}(wx_i + b - y_i)^2$$

- 闭式解（Closed-form Solution）

核心思想：使用矩阵求导，直接解出使损失最小的 `w` 和 `b`，不需要迭代。

我们构建设计矩阵 X 和标签向量 Y：

$$令\ X = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \cdots & \cdots \\ x_n & 1 \end{bmatrix}$$

$$令\ \theta = \begin{bmatrix} w \\ b \end{bmatrix}$$

目标变成求解：

$$\theta = (X^T X)^{-1} X^T Y$$

这就是**最小二乘法的闭式解。**

- 梯度下降（Gradient Descent）

核心思想：从随机初始值出发，逐步“沿着误差下降最快的方向”来更新参数 `w, b`，直到收敛。

损失函数：

$$L = \frac{1}{n}\sum_{i=1}^{n}(wx_i + b - y_i)^2$$

求梯度：

$$\frac{\partial L}{\partial w} = \frac{2}{n}\sum x_i(wx_i + b - y_i)$$
$$\frac{\partial L}{\partial b} = \frac{2}{n}\sum (wx_i + b - y_i)$$

每一步更新：

$$w := w - \eta \cdot \frac{\partial L}{\partial w}$$
$$b := b - \eta \cdot \frac{\partial L}{\partial b}$$

其中 η 是学习率（learning rate）。

• 多维度下的推导

这里我们使用**均方误差 (Mean Squared Error, MSE)**作为损失函数来衡量模型预测的好坏，损失函数 J(w,b) 定义为：

$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2 = \frac{1}{2m}\sum_{i=1}^{m}(w^T x^{(i)} + b - y^{(i)})^2$$

这里的 1/2 是为了方便后续求导计算。我们的目标就是找到使 J(w,b) 最小的 w* 和 b*

- 闭式解 (Normal Equation)

为了简化计算，我们将偏置项 b 和权重向量 w 合并成一个新的权重向量：

- 新的权重向量 $\hat{w} = \begin{pmatrix} w \\ b \end{pmatrix} \in \mathbb{R}^{d+1}$
- 新的特征向量 $\hat{x} = \begin{pmatrix} x \\ 1 \end{pmatrix} \in \mathbb{R}^{d+1}$

这样，我们的线性模型就可以被重写为：

$$\hat{y} = \hat{w}^T \hat{x}$$

现在，我们将整个数据集表示为矩阵形式：

设计矩阵 $\mathbf{X}_b \in \mathbb{R}^{m\times(d+1)}$，每一行代表一个样本（并加上了值为1的一列）：

$$\mathbf{X}_b = \begin{pmatrix} (x^{(1)})^T & 1 \\ (x^{(2)})^T & 1 \\ \vdots & \vdots \\ (x^{(m)})^T & 1 \end{pmatrix}$$

真实标签向量 $y \in \mathbb{R}^m$：

$$y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix}$$

所有样本的预测值可以表示为：

$$\mathbf{X}_b\hat{w}$$

因此，损失函数可以写成：

$$J(\hat{w}) = \frac{1}{2m}(\mathbf{X}_b\hat{w} - y)^T(\mathbf{X}_b\hat{w} - y)$$

为了找到使损失函数最小的 w^，我们需要对 w^ 求梯度，并令其为零。

$$\nabla_{\hat{w}} J(\hat{w}) = \nabla_{\hat{w}} \left(\frac{1}{2m}(\mathbf{X}_b\hat{w} - y)^T(\mathbf{X}_b\hat{w} - y) \right) = 0$$

展开后求导

$$\frac{1}{m}(\mathbf{X}_b^T\mathbf{X}_b\hat{w} - \mathbf{X}_b^T y) = 0$$

$$\mathbf{X}_b^T\mathbf{X}_b\hat{w} = \mathbf{X}_b^T y$$

最后得到闭式解：

$$\hat{w}^* = (\mathbf{X}_b^T\mathbf{X}_b)^{-1}\mathbf{X}_b^T y$$

其中，w^* 的前 d 个元素是 w*，最后一个元素是 b*

梯度下降法 (Gradient Descent)

首先需要计算损失函数对 w 和 b 的偏导数。

对 w_j 求偏导：

$$\frac{\partial J}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}(w^T x^{(i)} + b - y^{(i)}) \cdot \frac{\partial}{\partial w_j}(w^T x^{(i)} + b)$$

$$\frac{\partial J}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)}) \cdot x_j^{(i)}$$

其中 $x_j^{(i)}$ 是第 i 个样本的第 j 个特征。

对 b 求偏导：

$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(w^T x^{(i)} + b - y^{(i)}) \cdot \frac{\partial}{\partial b}(w^T x^{(i)} + b)$$

$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})$$

在梯度下降的每一步中，我们都按照以下规则更新参数 w 和 b。设 α 为学习率（一个控制更新步长的超参数）。

更新 wj：

$$w_j := w_j - \alpha\frac{\partial J}{\partial w_j}$$

$$w_j := w_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})x_j^{(i)}$$

更新 b：

$$b := b - \alpha\frac{\partial J}{\partial b}$$

$$b := b - \alpha\frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})$$

算法流程：

1. 随机初始化 w 和 b。
2. 重复以下步骤直到收敛（例如，损失函数变化很小或达到最大迭代次数）：
   a. 计算当前 w 和 b 下的梯度 $\frac{\partial J}{\partial w_j}$ 和 $\frac{\partial J}{\partial b}$。
   b. **同时**更新所有参数 w_j (for $j = 1, ..., d$) 和 b。
3. 最终得到的 w 和 b 就是我们要求的近似最优解 w^* 和 b^*。

理解张量与 PyTorch 基础张量操作

什么是张量

张量（Tensor）是**任意维度的数组**，是 PyTorch 中最基本的数据结构，相当于：

- **标量**（0维张量）→ `torch.tensor(3.0)`
- **向量**（1维张量）→ `torch.tensor([1.0, 2.0])`
- **矩阵**（2维张量）→ `torch.tensor([[1.0, 2.0], [3.0, 4.0]])`
- **更高维张量**：图像是 3D（宽 × 高 × 通道），视频是 4D

PyTorch 中所有计算都基于张量，比如矩阵乘法、求导、线性回归等。

PyTorch 张量的基础操作

```
import torch

# 创建张量
a = torch.tensor([1.0, 2.0, 3.0])   # 一维张量（向量）
b = torch.tensor([[1, 2], [3, 4]])  # 二维张量（矩阵）

# 张量形状
print(b.shape)  # 输出 torch.Size([2, 2])

# 张量加法
c = a + torch.tensor([4.0, 5.0, 6.0])

# 张量乘法（点乘 vs 矩阵乘法）
dot = torch.dot(a, torch.tensor([1.0, 1.0, 1.0]))     # 点乘
mat = torch.matmul(b, torch.tensor([[1], [1]]))       # 矩阵乘法

# 改变形状
x = torch.tensor([1, 2, 3, 4])
x = x.view(2, 2)   # 改成 2x2 矩阵

# 自动求导
w = torch.tensor([1.0], requires_grad=True)
y = w * 2
y.backward()
print(w.grad)  # 输出 tensor([2.])
```

```
D:\Documents\WeChat Files\wxid_xwmilvwf4p2f22\FileStorage\File\2025-07\经典机器学习课程及lab2222\经典机器学习课程及lab>p
ython test.py
torch.Size([2, 2])
tensor([2.])
```

用pytorch实现2个解法的计算过程（单维输入情况）

• 完成2处TODO注释的代码，并将代码和运行结果记录在实验报告中

Task 1这里使用线性回归的闭式解，通过统计学公式一次性求出最优的斜率 w 和截距 b，使得拟合直线 y=wx+by 尽可能贴近样本点。它先计算输入 x 和输出 y 的均值，然后用协方差除以方差得到 w，再通过

$$b = \bar{y} - w\bar{x}$$

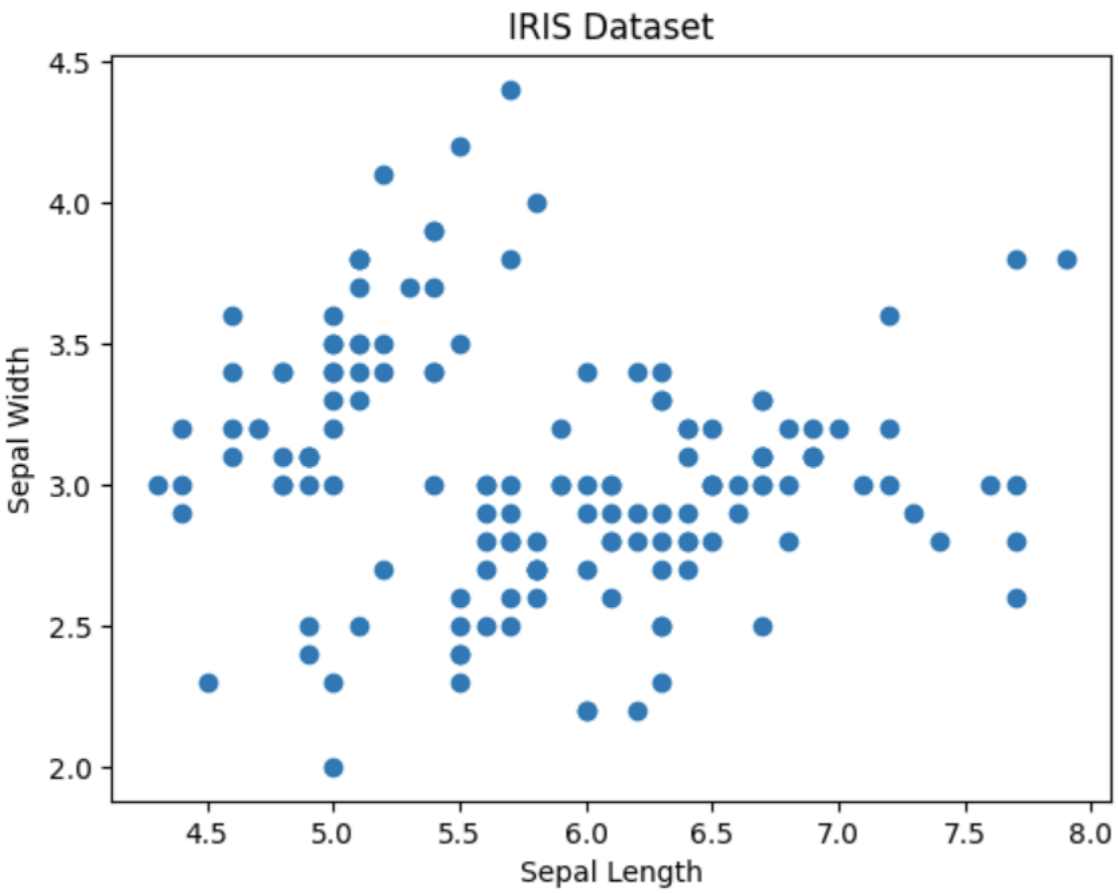
求得截距。

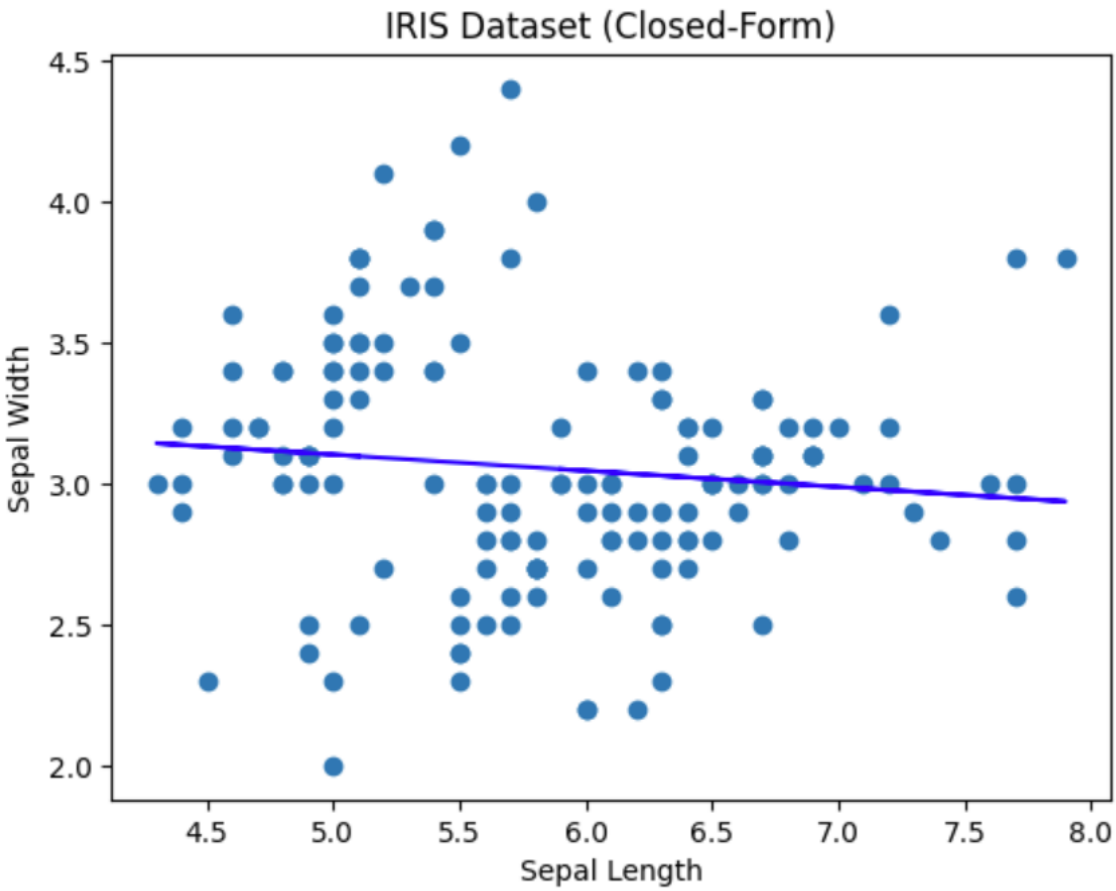
```
# Task 1: 闭式解 y = w * x + b
def lr_cf(X, Y):
    x = torch.tensor(X)
    y = torch.tensor(Y)
    n = len(x)

    # 使用正规方程解线性回归: w = cov(x, y) / var(x), b = ȳ - w * x̄
    x_mean = torch.mean(x)
    y_mean = torch.mean(y)
    w = torch.sum((x - x_mean) * (y - y_mean)) / torch.sum((x - x_mean) ** 2)
    b = y_mean - w * x_mean
    return w.item(), b.item()
```

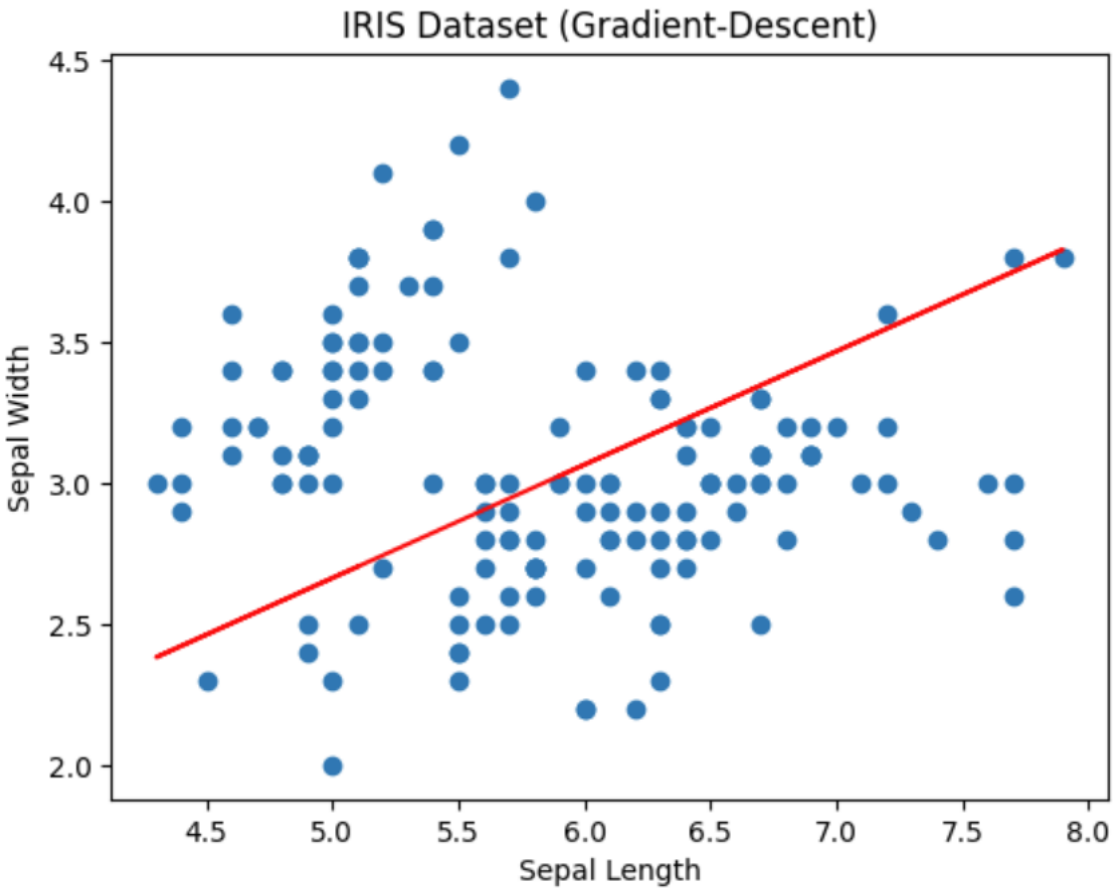
Task 2这里我们通过不断迭代调整模型参数 w 和bb，最小化预测值与真实值之间的均方误差（MSE）损失。在每一轮中，模型根据当前 w,b 计算预测值，计算损失后使用反向传播求出梯度，然后按梯度方向更新参数。如此迭代多次后，w 和 b 会逐渐逼近最优解，从而使直线尽量贴近数据点。

```
# Task 2: 梯度下降
def lr_gradient_descent(X, Y):
    x = torch.tensor(X)
    y = torch.tensor(Y)

    w = torch.tensor(0.0, requires_grad=True)
    b = torch.tensor(0.0, requires_grad=True)

    lr = 0.01  # 学习率
    epochs = 500

    for epoch in range(epochs):
        y_pred = w * x + b
        loss = torch.mean((y_pred - y) ** 2)  # MSE

        # 反向传播
        loss.backward()

        # 手动更新参数
        with torch.no_grad():
```

```
            w -= lr * w.grad
            b -= lr * b.grad
            w.grad.zero_()
            b.grad.zero_()

    return w.item(), b.item()
```




IRIS Dataset (Closed-Form)
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0

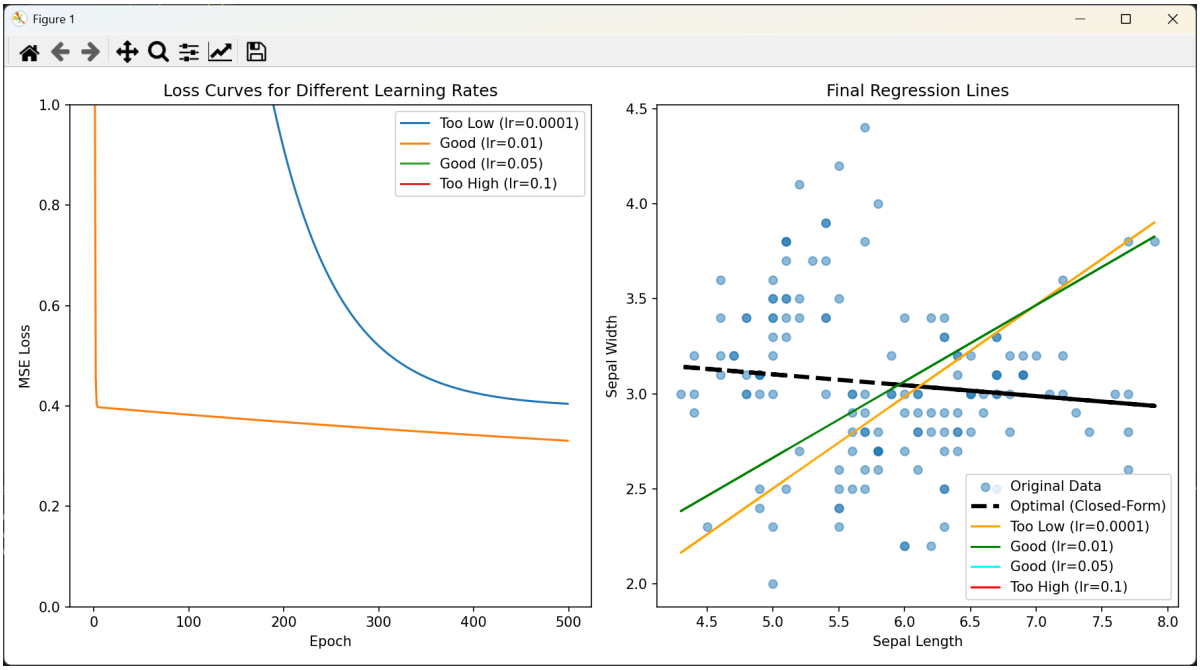

IRIS Dataset (Gradient-Descent)
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0

- **Task #2：尝试不同的学习率，并将实验结果记录在实验报告中**

```
import pandas as pd
import matplotlib.pyplot as plt
import torch


# 加载数据
def load_data():
    names = ["sepal-length", "sepal-width", "petal-length", "petal-width",
"class"]
    # Ensure the CSV file is available or provide the correct path
    try:
        dataset = pd.read_csv("iris.data.csv", names=names)
    except FileNotFoundError:
        print(
            "Error: 'iris.data.csv' not found. Please download it or place it in
the correct directory."
        )
        return None, None
    X = dataset["sepal-length"].to_numpy()
    Y = dataset["sepal-width"].to_numpy()
    return X.tolist(), Y.tolist()


# 闭式解（作为基准）
def lr_cf(X, Y):
    x = torch.tensor(X, dtype=torch.float32)
    y = torch.tensor(Y, dtype=torch.float32)
    x_mean = torch.mean(x)
    y_mean = torch.mean(y)
    w = torch.sum((x - x_mean) * (y - y_mean)) / torch.sum((x - x_mean) ** 2)
    b = y_mean - w * x_mean
    return w.item(), b.item()


# 梯度下降（修改后）
def lr_gradient_descent(X, Y, lr, epochs=500):
    x = torch.tensor(X, dtype=torch.float32)
    y = torch.tensor(Y, dtype=torch.float32)

    w = torch.tensor(0.0, requires_grad=True)
    b = torch.tensor(0.0, requires_grad=True)

    losses = []

    for epoch in range(epochs):
        y_pred = w * x + b
        loss = torch.mean((y_pred - y) ** 2)
        losses.append(loss.item())
```

```
        loss.backward()

        with torch.no_grad():
            w -= lr * w.grad
            b -= lr * b.grad
            w.grad.zero_()
            b.grad.zero_()

    return w.item(), b.item(), losses


# --- 主程序 ---
X, Y = load_data()
if X is None:
    exit()

# 计算闭式解作为最优基准
w_cf, b_cf = lr_cf(X, Y)

# 设定不同的学习率进行实验
learning_rates = {
    "Too Low (lr=0.0001)": 0.0001,
    "Good (lr=0.01)": 0.01,
    "Good (lr=0.05)": 0.05,
    "Too High (lr=0.1)": 0.1,
}

results = {}
plt.figure(figsize=(12, 6))

# --- 1. 绘制损失函数下降曲线 ---
plt.subplot(1, 2, 1)
for name, lr in learning_rates.items():
    w, b, losses = lr_gradient_descent(X, Y, lr)
    results[name] = {"w": w, "b": b}
    plt.plot(losses, label=name)

plt.title("Loss Curves for Different Learning Rates")
plt.xlabel("Epoch")
plt.ylabel("MSE Loss")
plt.legend()
plt.ylim(0, 1)  # 限制y轴范围以便观察

# --- 2. 绘制最终拟合的回归线 ---
plt.subplot(1, 2, 2)
plt.scatter(X, Y, label="Original Data", alpha=0.5)
# 绘制最优的闭式解
y_line_cf = w_cf * torch.tensor(X) + b_cf
plt.plot(
    X,
    y_line_cf,
    color="black",
```

```
    linestyle="--",
    linewidth=3,
    label="Optimal (Closed-Form)",
)

colors = ["orange", "green", "cyan", "red"]
for i, (name, res) in enumerate(results.items()):
    y_line = res["w"] * torch.tensor(X) + res["b"]
    plt.plot(X, y_line, color=colors[i], label=name)

plt.title("Final Regression Lines")
plt.xlabel("Sepal Length")
plt.ylabel("Sepal Width")
plt.legend()
plt.tight_layout()
plt.show()
```



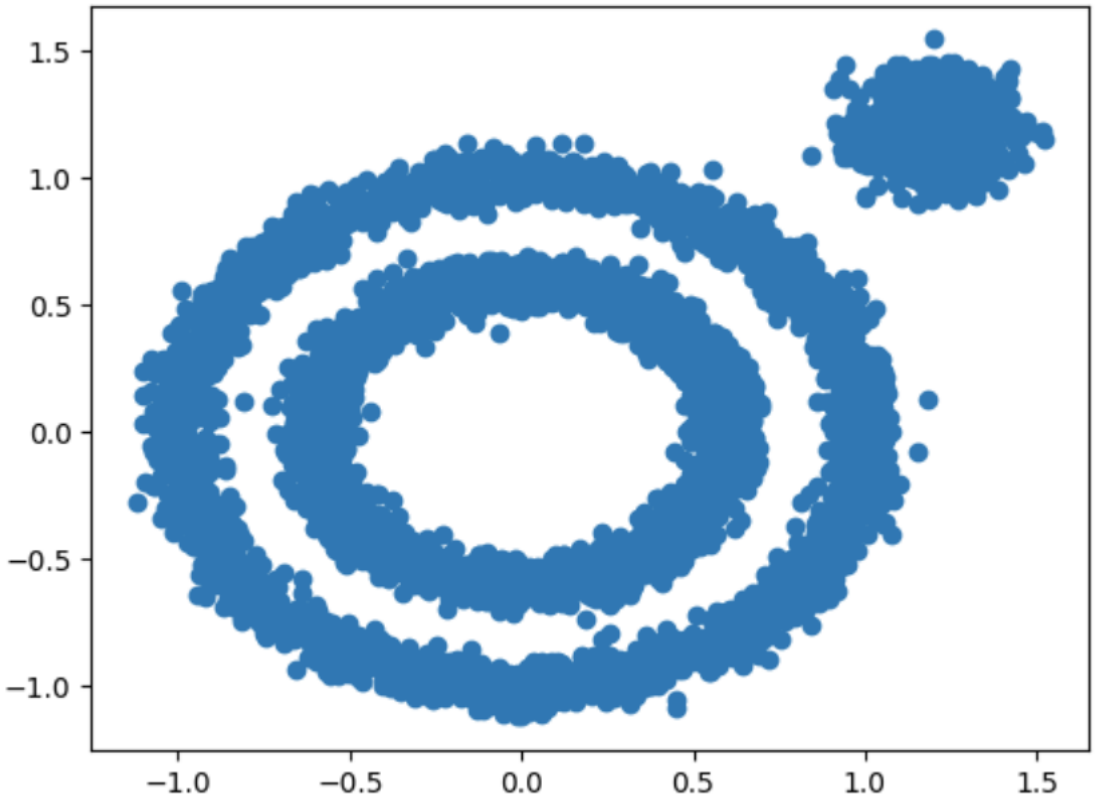
学习率	损失曲线行为	最终拟合效果	结论
0.0001	下降极其缓慢，远未收敛	差，与最优解差距大	**学习过慢**
0.01	平稳、快速下降并收敛	优，与最优解重合	**良好**
0.05	非常快速地下降并收敛	优，与最优解重合	**高效/更优**
0.1	剧烈震荡并数值发散	无效	**学习过快/发散**

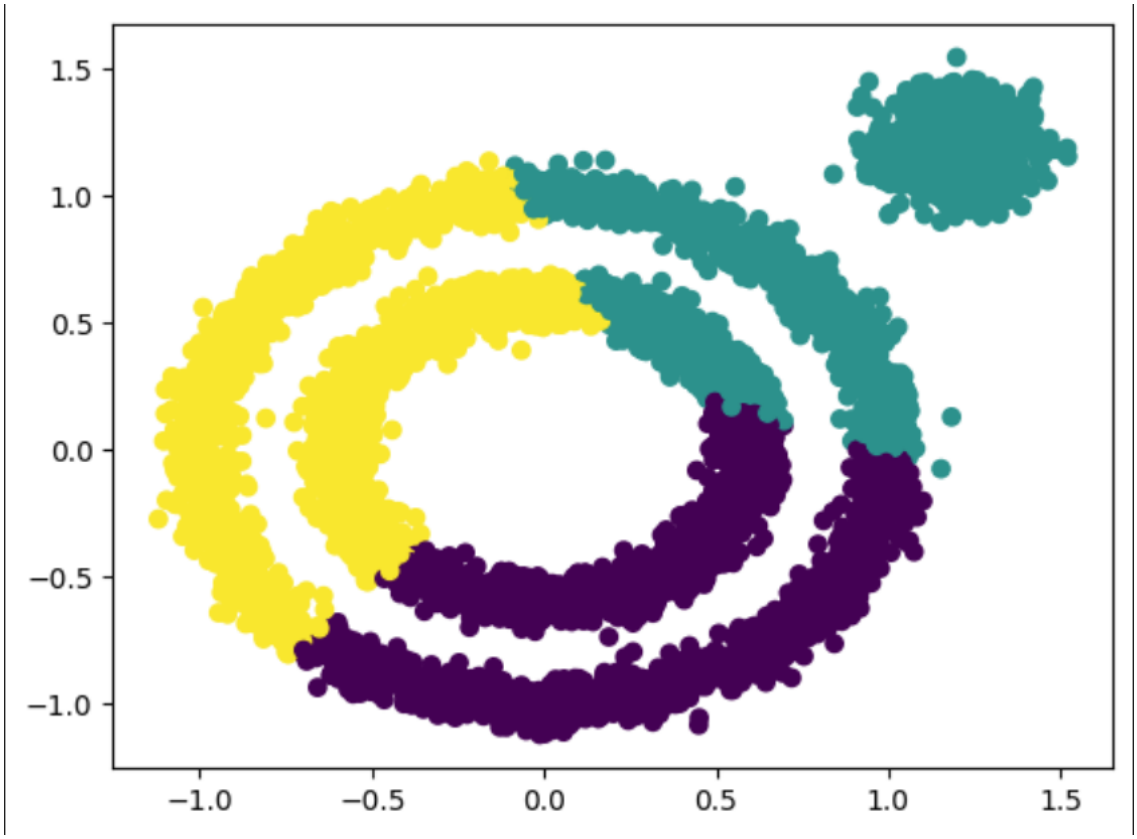
Task 3

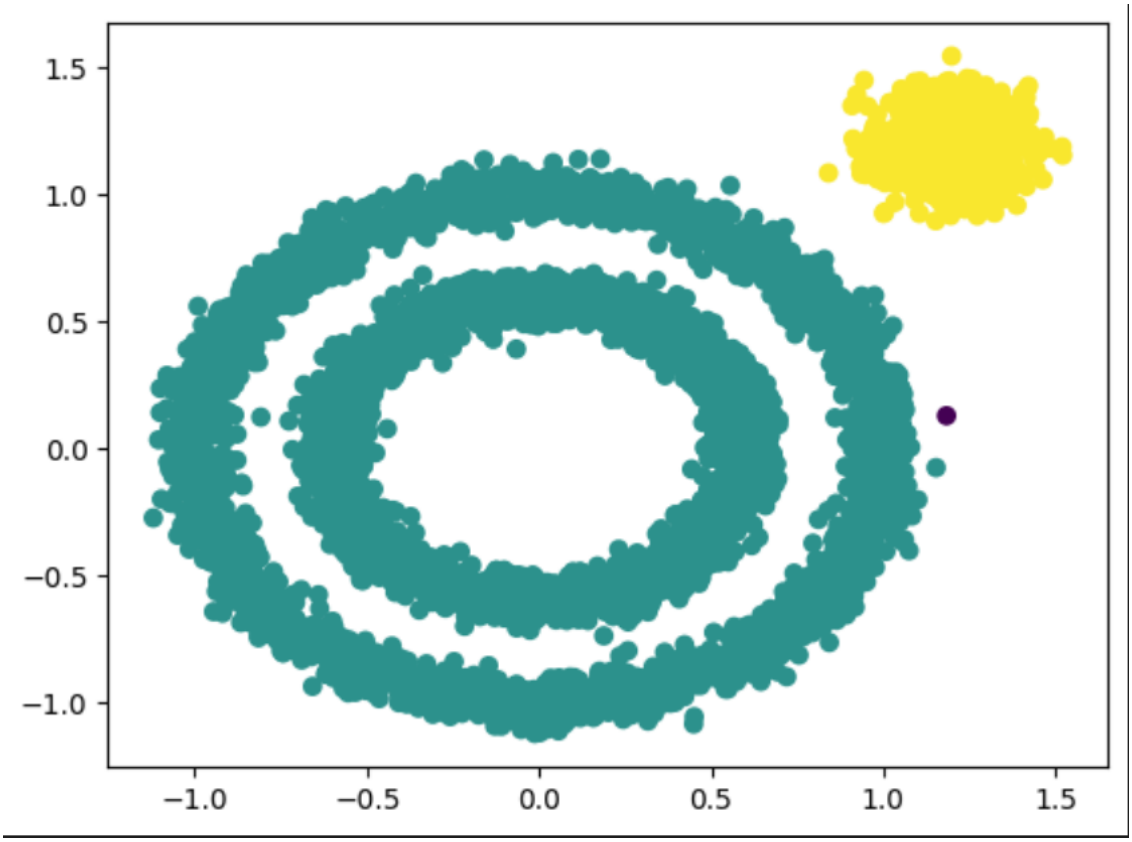
调api

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
# 导入所需的聚类算法类
from sklearn.cluster import KMeans, DBSCAN

X1, y1=datasets.make_circles(n_samples=5000, factor=.6,
                                      noise=.05)
X2, y2 = datasets.make_blobs(n_samples=1000, n_features=2, centers=[[1.2,1.2]],
cluster_std=[[.1]],
                         random_state=9)

X = np.concatenate((X1, X2))
plt.scatter(X[:, 0], X[:, 1], marker='o')
plt.show()

"""
任务1：调用K-Means
"""
def apply_kmeans(X):
    ### TODO：查阅文档，调用 ###
    # 实例化KMeans，设置为3个簇，并进行拟合预测
    kmeans = KMeans(n_clusters=3, random_state=9, n_init='auto')
    Y = kmeans.fit_predict(X)
    return Y


y_pred = apply_kmeans(X)

### 可视化聚类结果
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
plt.show()


"""
任务2：调用DBScan
"""
def apply_dbscan(X):
    ### TODO：查阅文档，调用 ###
    # 实例化DBSCAN并进行拟合预测，eps=0.1
    dbscan = DBSCAN(eps=0.1)
    Y = dbscan.fit_predict(X)
    return Y
```

```
y_pred = apply_dbscan(X)
### 可视化聚类结果
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
plt.show()
```







整理超参作用

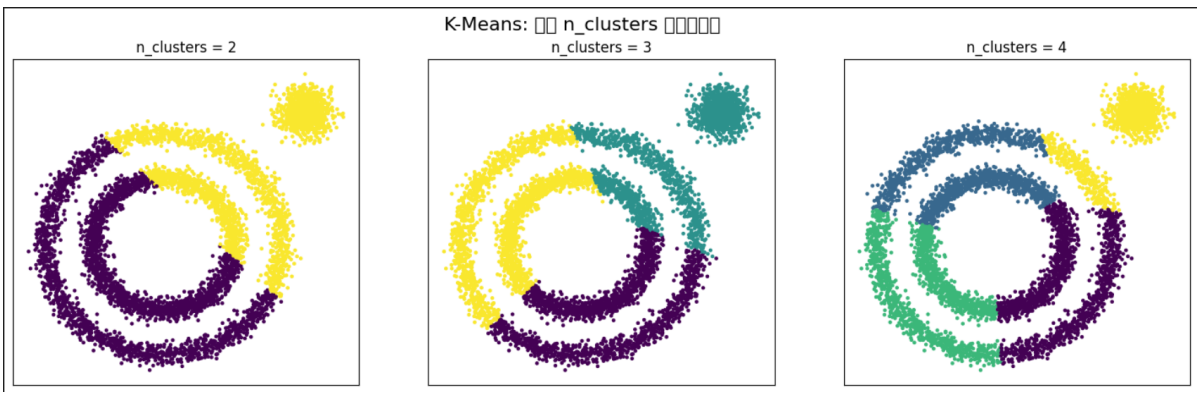
算法	超参数	作用	选择建议
K-Means	n_clusters	（必须）指定要形成的簇的数量。	最重要。使用“手肘法”或根据业务知识确定。
	init	初始化簇中心的方法。	保持默认的 'k-means++'。
	n_init	使用不同初始值运行算法的次数。	保持默认的 'auto'。
	random_state	保证结果可复现。	实验和分享时务必设置。
DBSCAN	eps	定义邻域的半径大小，决定密度尺度。	最关键。使用“k-距离图”寻找拐点。
	min_samples	定义成为核心点的最小邻居数，决定密度阈值。	根据数据维度（如 2*D）和噪声水平设定。

对不同的超参取值进行实验

K-Means

对于K-Means，核心实验是确定最佳的簇数量 `k` ，我们直接运行 `n_clusters` 分别为2、3、4时的K-Means，看看哪一个结果在视觉上最符合数据的直观结构

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans

# --- 生成数据 ---
X1, y1 = datasets.make_circles(n_samples=5000, factor=.6, noise=.05,
random_state=1)
X2, y2 = datasets.make_blobs(n_samples=1000, n_features=2, centers=[[1.2, 1.2]],
cluster_std=[.1], random_state=9)
X = np.concatenate((X1, X2))

# --- K-Means视觉对比实验 ---
k_values = [2, 3, 4]
fig, axes = plt.subplots(1, 3, figsize=(18, 5))
fig.suptitle('K-Means: 不同 n_clusters 的视觉对比', fontsize=16)

for i, k in enumerate(k_values):
    kmeans = KMeans(n_clusters=k, random_state=9, n_init='auto')
    y_pred = kmeans.fit_predict(X)

    ax = axes[i]
    ax.scatter(X[:, 0], X[:, 1], c=y_pred, s=5)
    ax.set_title(f'n_clusters = {k}')
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
```

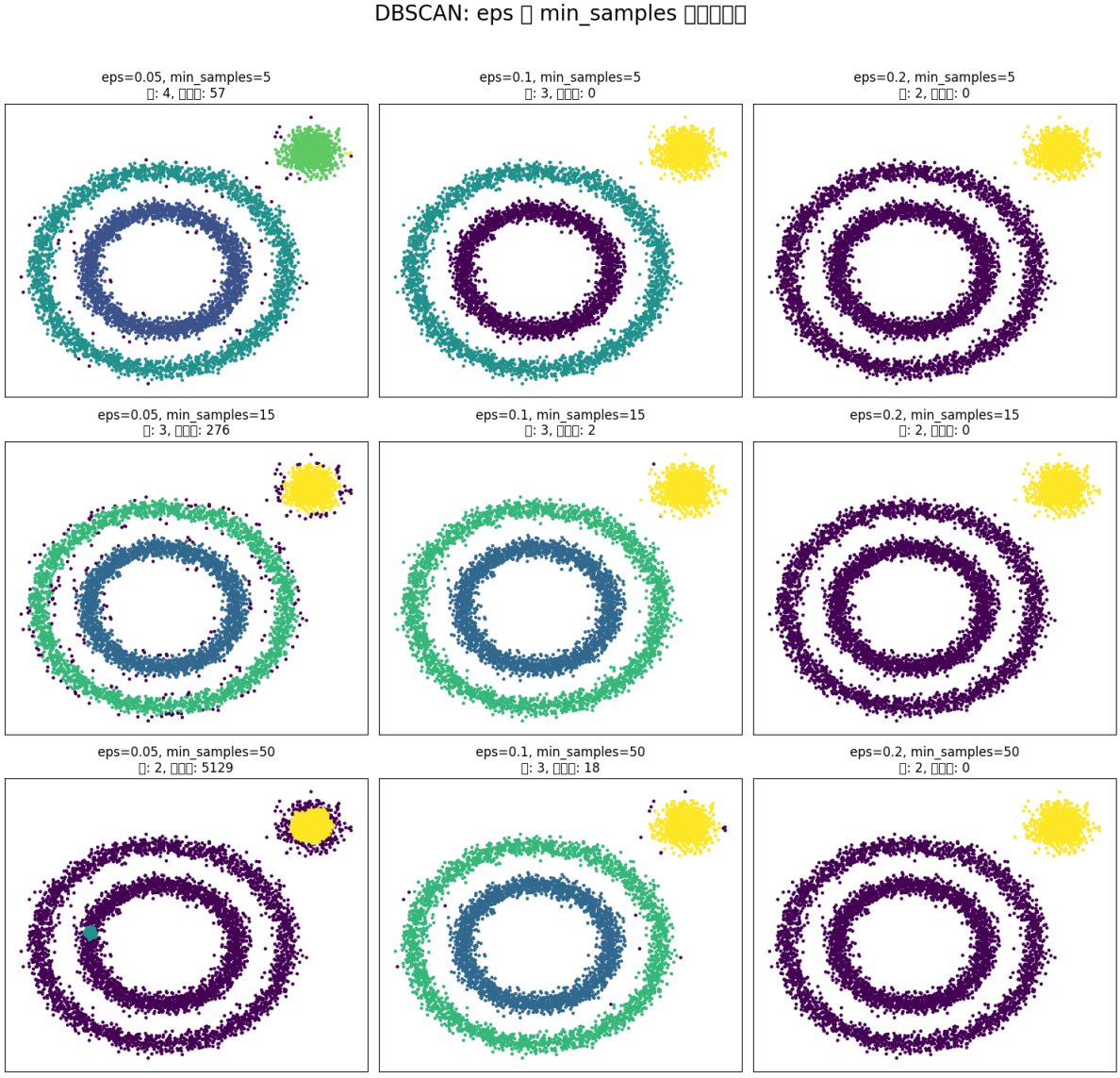


- `n_clusters = 2` : K-Means会粗暴地将数据切成两半。由于算法试图最小化方差，它会将同心圆的外环和内环的一部分合并，显然不符合数据的真实结构。

- `n_clusters = 3`：这个结果在视觉上最合理。K-Means大致将数据分为了三个部分：内环、外环和右下角的团块。
- `n_clusters = 4`：为了凑够4个簇，算法将某个自然的簇（如外环）再强行分割成两部分，导致了过分割（over-segmentation）

DBSCAN

对于DBSCAN，我们将进行网格搜索实验，观察 `eps` （邻域半径）和 `min_samples` （最小邻居数）这两个关键参数如何共同影响聚类结果



观察第一行（`min_samples=5`）：

- `eps=0.05` (左上): 半径太小，大部分点都被当成了噪声。
- `eps=0.1` (中上): **效果最佳**。半径大小适中，完美地识别出3个簇，且噪声很少。
- `eps=0.2` (右上): 半径过大，导致两个同心圆之间的“鸿沟”被跨越，它们被合并成了1个大簇。

观察第二列（`eps=0.1`）：

- `min_samples=5` (中上): 效果很好。

- `min_samples=15` (正中): 密度要求提高，但结果依然稳健，说明数据簇的密度足够。
- `min_samples=50` (中下): 密度要求变得非常苛刻。此时，同心圆的“线状”密度不足以满足要求，它们被完全视为噪声。

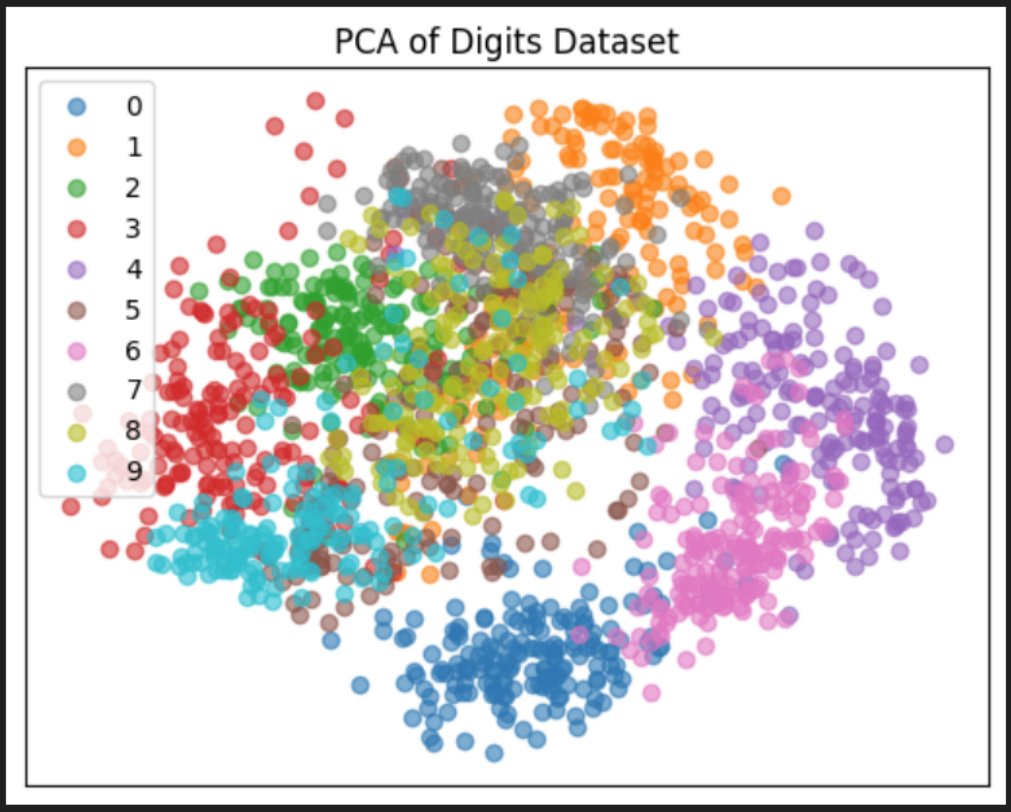
Task 4

调api

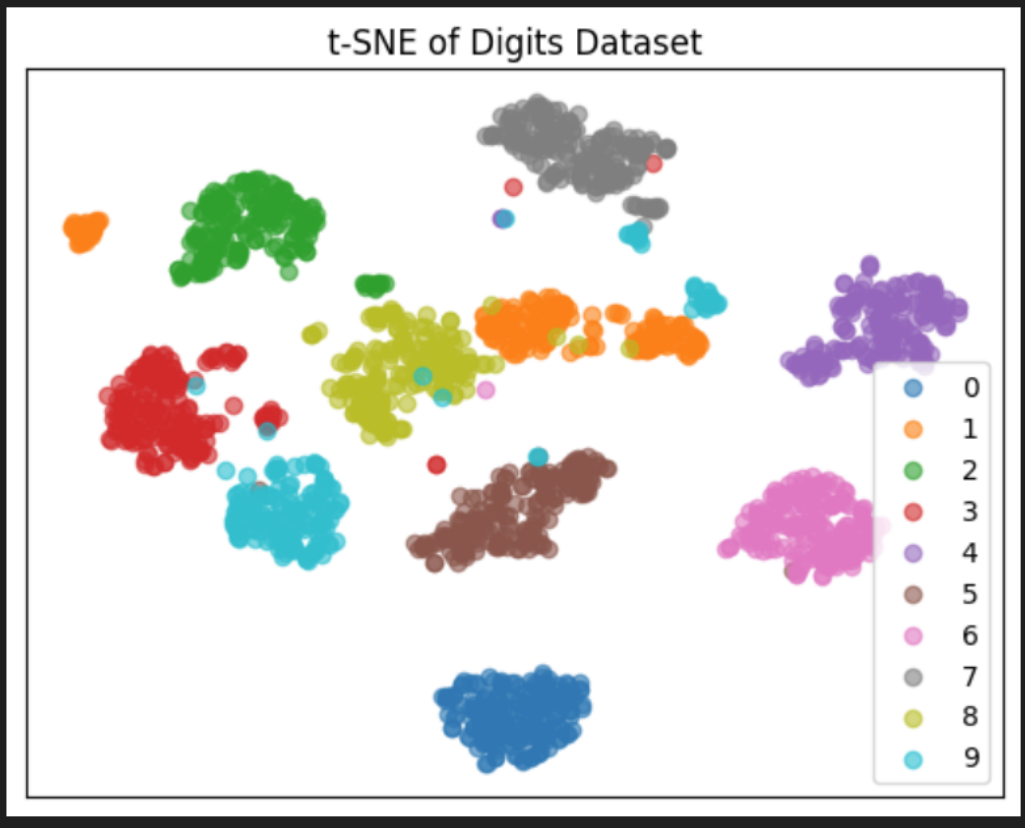
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

# 加载数据
data, labels = load_digits(return_X_y=True)
(n_samples, n_features), n_digits = data.shape, np.unique(labels).size

print(f"# digits: {n_digits}; # samples: {n_samples}; # features {n_features}")


"""
任务3：调用PCA
"""
def pca_2D_reduced(data):
    ### TODO：查阅文档，调用
    # 初始化一个PCA模型，设置目标维度为2
    pca = PCA(n_components=2)
    # 对数据进行拟合和降维
    reduced_data = pca.fit_transform(data)
    return reduced_data

print("\nRunning PCA...")
reduced_data_pca = pca_2D_reduced(data)

### 可视化降维结果
print("Plotting PCA result...")
# 为了更好的可视化，我们按数字类别给散点图上色
for i in range(n_digits):
    plt.scatter(reduced_data_pca[labels == i, 0], reduced_data_pca[labels == i,
1], alpha=0.6, label=str(i))
plt.title("PCA of Digits Dataset")
plt.legend()
plt.xticks(())
```

```
plt.yticks(())
plt.show()

"""
任务4：调用t-SNE
"""
def tsne_2D_reduced(data):
    ### TODO：查阅文档，调用
    # 初始化一个t-SNE模型，设置目标维度为2
    # init='pca' 和 random_state 可以让结果更稳定和可复现
    tsne = TSNE(n_components=2, init='pca', random_state=42, learning_rate='auto')
    # 对数据进行拟合和降维
    reduced_data = tsne.fit_transform(data)
    return reduced_data

print("\nRunning t-SNE... (This may take a moment)")
reduced_data_tsne = tsne_2D_reduced(data)

### 可视化降维结果
print("Plotting t-SNE result...")
# 为了更好的可视化，我们按数字类别给散点图上色
for i in range(n_digits):
    plt.scatter(reduced_data_tsne[labels == i, 0], reduced_data_tsne[labels == i,
1], alpha=0.6, label=str(i))
plt.title("t-SNE of Digits Dataset")
plt.legend()
plt.xticks(())
plt.yticks(())
plt.show()
```

```
# digits: 10; # samples: 1797; # features 64

Running PCA...
Plotting PCA result...
```




```
Running t-SNE... (This may take a moment)
Plotting t-SNE result...
```



整理超参作用

算法	关键超参数	描述	调优策略
PCA	n_components	降维后的目标维度数或保留的方差比例。	最重要。使用累积解释方差图找“拐点”，或直接设为0.95等小数。
	whiten	是否对降维结果进行方差归一化。	若后续接分类/聚类模型，设为True可能效果更好。
t-SNE	perplexity	关注的近邻数量，平衡局部与全局结构。	最关键。通常在 5-50 间尝试，选择视觉效果最好的。
	learning_rate	优化的步长。	重要。设为 'auto' 是当前最佳实践。
	n_iter	优化迭代次数。	至少1000。若结果拥挤可尝试增加。
	init	初始布局方法。	设为 'pca' 能获得更稳定和更好的结果。

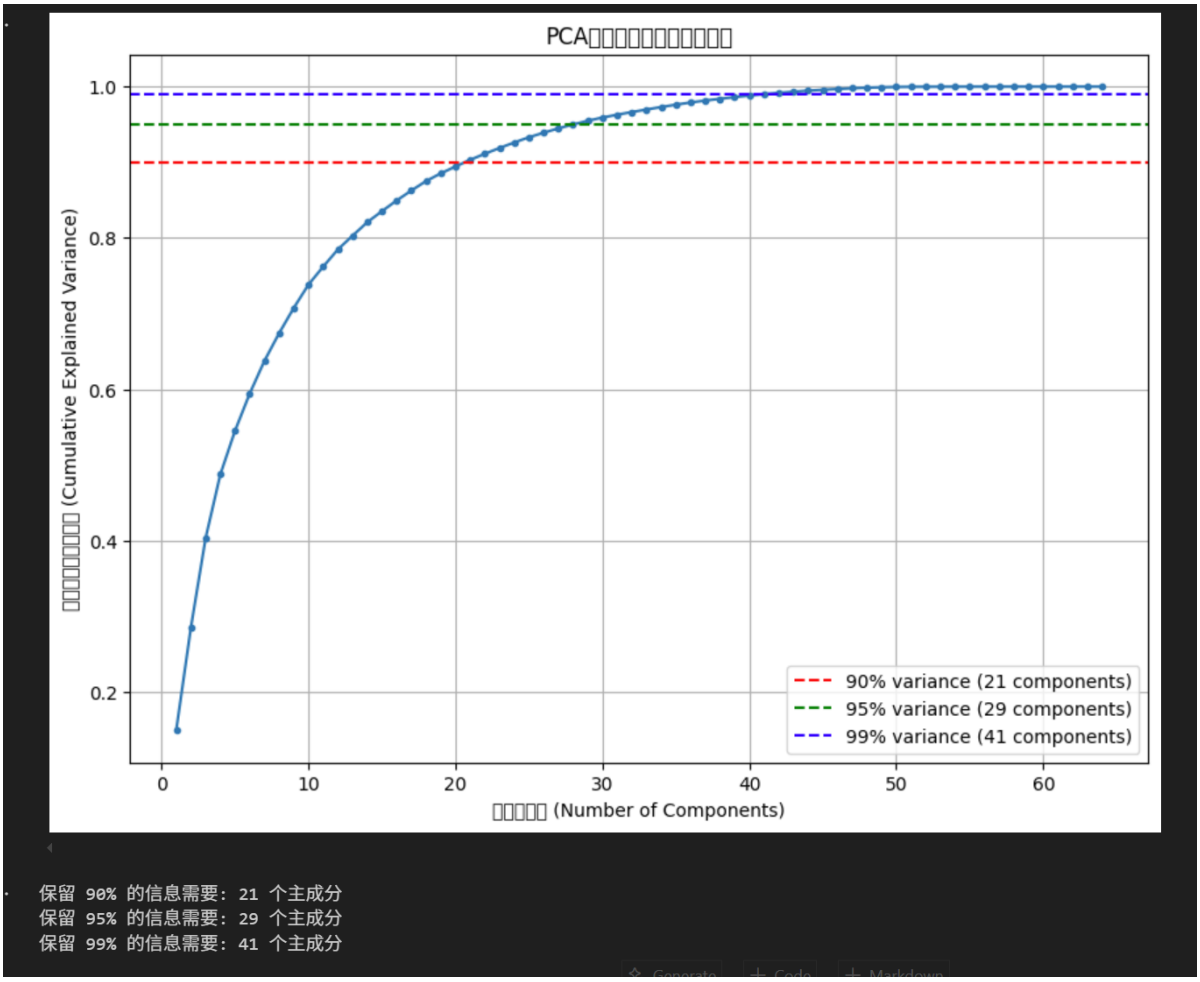
对不同的超参取值进行实验

PCA的超参数实验

对于PCA，最重要的“实验”是确定 `n_components` 的最佳取值

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

# 加载数据
data, labels = load_digits(return_X_y=True)

# 1. 初始化一个保留所有主成分的PCA模型
pca = PCA(n_components=None)
pca.fit(data)

# 2. 计算累积方差贡献率
# pca.explained_variance_ratio_ 包含了每个主成分解释的方差比例
cumulative_variance = np.cumsum(pca.explained_variance_ratio_)

# 3. 寻找达到特定方差阈值的维度数
# np.argmax(cumulative_variance >= 0.90) + 1  可以找到第一个超过阈值的索引
n_components_90 = np.argmax(cumulative_variance >= 0.90) + 1
n_components_95 = np.argmax(cumulative_variance >= 0.95) + 1
n_components_99 = np.argmax(cumulative_variance >= 0.99) + 1
```

```
# 4. 绘制曲线
plt.figure(figsize=(10, 7))
plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance, marker='.',
linestyle='-')
plt.xlabel("主成分数量 (Number of Components)")
plt.ylabel("累积解释方差贡献率 (Cumulative Explained Variance)")
plt.title("PCA累积解释方差贡献率曲线")
plt.grid(True)

# 添加参考线和注释
plt.axhline(y=0.90, color='r', linestyle='--', label=f'90% variance
({n_components_90} components)')
plt.axhline(y=0.95, color='g', linestyle='--', label=f'95% variance
({n_components_95} components)')
plt.axhline(y=0.99, color='b', linestyle='--', label=f'99% variance
({n_components_99} components)')
plt.legend(loc='best')
plt.show()

print(f"保留 90% 的信息需要: {n_components_90} 个主成分")
print(f"保留 95% 的信息需要: {n_components_95} 个主成分")
print(f"保留 99% 的信息需要: {n_components_99} 个主成分")
```



从图表中我们可以清晰地看到：

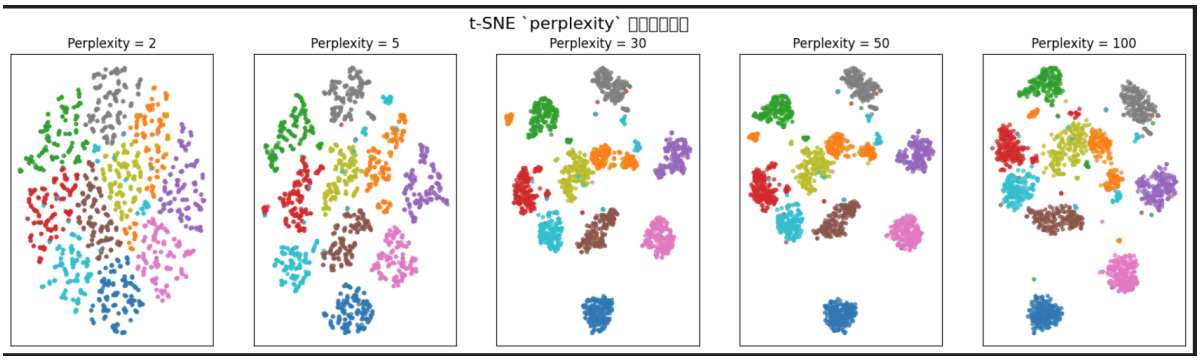
- **信息高度集中**：仅仅前10个主成分就已经解释了超过80%的数据方差。曲线在初期非常陡峭。

- **权衡取舍 (Trade-off)**：大约在10-20个主成分之后，曲线开始变得平缓（这就是“拐点”）。这意味着再增加主成分数量，能带来的信息增益越来越少。

t-SNE的超参数实验

对于t-SNE，我们固定其他参数，重点观察 `perplexity` **（困惑度）**这个最关键的超参数如何影响可视化结果

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.manifold import TSNE

# 加载数据
data, labels = load_digits(return_X_y=True)
n_digits = np.unique(labels).size

# 设定要实验的 perplexity 值
perplexity_values = [2, 5, 30, 50, 100]

# 创建一个 1x5 的子图网格
fig, axes = plt.subplots(1, len(perplexity_values), figsize=(20, 5))
fig.suptitle('t-SNE `perplexity` 参数实验对比', fontsize=16)

for i, perplexity in enumerate(perplexity_values):
    print(f"Running t-SNE with perplexity={perplexity}...")

    # 初始化t-SNE模型
    # 使用PCA初始化和固定随机状态来保证结果的稳定和可复现
    tsne = TSNE(n_components=2,
                perplexity=perplexity,
                init='pca',
                learning_rate='auto',
                random_state=42)

    reduced_data = tsne.fit_transform(data)

    # 在对应的子图中绘制结果
    ax = axes[i]
    for digit in range(n_digits):
        ax.scatter(reduced_data[labels == digit, 0],
                   reduced_data[labels == digit, 1],
                   alpha=0.7,
                   s=10) # 减小点的大小
    ax.set_title(f'Perplexity = {perplexity}')
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
```



- `Perplexity = 2`：困惑度极低，t-SNE过分关注每个点的最近邻。这导致数据被“撕裂”成许多孤立的小点或短线，完全无法体现出数字类别（0-9）的宏观聚类结构。
- `Perplexity = 5`：开始能看到一些聚类的趋势，但许多数字类别仍然是散乱的，或者被分成了多个子簇。
- `Perplexity = 30`：**效果理想**。我们可以清晰地看到10个分离良好的、紧凑的簇，每个簇对应一个数字。这表明 `perplexity=30` 在这个数据集上很好地平衡了局部细节和全局结构。
- `Perplexity = 50`：效果与30类似，依然很好。簇的边界变得更加平滑，各个簇之间也靠得更近一些。
- `Perplexity = 100`：困惑度过高。算法开始过多地考虑全局结构，导致一些原本分离的簇开始出现“融合”的趋势，簇的形状变得不那么自然。