模式识别实验报告

|  |  |
| --- | --- |
| 专业： | 人工智能 |
| 学号： | 58122231 |
| 年级： | 大二 |
| 姓名： | 陆文韬 |

签名：

时间：

**实验一 数据降维与分类实验**

1. **问题描述**

**概述**

利用两种降维技术对葡萄酒数据进行降维，并对降维前后的数据进行分类。

**数据说明**

所给数据集包含两个.csv 文件，分别为红葡萄酒数据（包括 1599 个样本）和白葡萄酒数据（包含 4898 个样本）, 每个样本含有 11 个特征（文件的前 11 列）: 固定酸度、挥发酸度、柠檬酸、残糖、氯化物、游离二氧化硫、总二氧化硫、密度、pH 值、硫酸盐、酒精和专家对此葡萄酒的打分（文件的最后一列）。

**实验内容**

一、对红白葡萄酒数据分别进行 PCA 和 LDA 降维处理，在降维后的数据集上完成基于logistic 回归分类器的训练和测试。要求比较应用降维技术前后，分类器准确率的变化（对于降维后的数据，可以尝试利用可视化方法展示结果）。

二、基于 MindSpore 平台实现算法，对相同的数据集进行训练，并与不使用 MindSpore 平台实现的算法对比结果（包括但不限于准确率、算法迭代收敛次数等指标），并分析结果中出现差异的可能原因，给出使用 MindSpore 的心得和建议。

三、（加分项）使用 MindSpore、sklearn 等平台提供的相似任务数据集（例如，其他的分类任务数据集）测试自己独立实现的算法，并与 MindSpore、sklearn 等平台上的官方实现算法进行对比，进一步分析差异及其成因。

1. **实现步骤与流程**

**第一部分**

**一、数据预处理**

因为原始的数据集分为两个csv文件，分别为winequality-white.csv和winequality-red.csv，但是在数据处理的时候，我们需要将两个数据集合并为一个进行PCA或者LDA处理，并进行后续的二分类任务，因此我们将两个数据集合并为一个，并增加一列label，值为0表示当前样本的类别属于白酒，值为1表示当前样本的类别属于红酒。

（1）移除异常值

移除异常值（outliers）是数据预处理中的一项重要任务，它对提高模型的准确性和稳定性具有显著影响。异常值可能是由于测量错误、数据输入错误或非典型的样本变异产生的，这些值可能会扭曲统计分析的结果，影响模型的训练，从而导致模型表现不佳。

移除异常值具有以下这些重要的意义和价值：

**改善模型性能：**异常值可能会导致模型学习到错误或偏差大的关系，去除这些值有助于减少噪声，使模型更好地学习数据中的真实模式。

**增强数据分析的准确性：**异常值可以影响数据的基本统计描述，如均值、标准差等，移除它们可以得到更准确的数据统计特征。

**避免过拟合：**特别是在小数据集中，异常值可能导致模型过度适应这些少数点，影响其泛化能力。

在本次实验中，我使用的是四分位数法（Interquartile Range, IQR），这是一种常用的统计方法来识别和移除异常值。具体步骤如下：

**1. 计算四分位数：**对于数据框中的每一列（假设最后一列为标签，不进行处理），计算第一四分位数（Q1，即25%分位数）和第三四分位数（Q3，即75%分位数）。

**2. 计算IQR：**IQR是第三四分位数和第一四分位数的差（`IQR = Q3 - Q1`）。IQR描述了数据的中间50%的扩散程度。

**3. 定义异常值范围：**根据IQR计算异常值的范围。通常异常值被定义为低于 `Q1 - 1.5 \* IQR` 或高于 `Q3 + 1.5 \* IQR` 的值。这里的1.5是一个常用的系数，但可以根据具体情况调整以识别更极端或更温和的异常值。

**4. 过滤异常值：**使用逻辑过滤条件，保留落在上述范围内的数据点，移除落在外面的数据点。

**5. 重置索引：**由于移除了一些行，因此需要重置DataFrame的索引以保持索引的连续性。

通过这种方式，程序能有效地从各个特征列中移除异常值，使得数据集更加干净，为后续的数据分析和模型训练创造更好的条件。

（2）标准化特征

标准化特征（Feature Standardization）是数据预处理中常见的一步，其目的是将数据转换为具有零均值和单位方差的形式。这一处理方式有助于统一不同量级或单位的特征的比例，使得模型训练过程更为平稳和高效。

标准化特征具有以下这些重要的意义和价值：

**改进算法收敛速度：**在使用梯度下降等优化算法时，特征标准化可以帮助算法更快地收敛。

消除量纲影响：不同特征可能有不同的量纲和量级，标准化后，这些特征对模型的影响被统一，可以公平地评价和比较它们的重要性。

**提高模型性能：**对于很多机器学习算法（如支持向量机、逻辑回归等），预处理后的数据能够提高模型的性能。

**满足算法要求：**一些算法（如K-近邻、PCA）对数据的尺度非常敏感，未经标准化处理的数据可能导致模型效果不佳。

本次实验中使用pandas实现了对DataFrame中特征的标准化。具体步骤如下：

**1. 提取特征：**选取DataFrame中除最后一列（通常为标签列）之外的所有列作为特征。

**2. 计算均值和标准差：**对每个特征计算均值和标准差。这两个统计量将用于标准化过程。

**3. 标准化特征：**对特征进行标准化处理。这一步骤确保了每个特征的均值变为0，标准差变为1。

**4. 更新DataFrame：**将标准化后的特征值赋回原来的DataFrame中对应的位置。

**5. 返回处理后的DataFrame：**函数返回更新后的DataFrame，现在所有特征都已标准化。

通过这种方式，函数能够确保数据集中的特征都具有相同的尺度，有助于后续的数据分析和机器学习模型的训练过程。

**二、共线性检测**

共线性检测是统计学和数据科学中的一项重要的任务，它涉及检查输入变量之间的线性关系，当多个变量之间存在高度线性关系时，在进行数据预测的过程中，我们就难以获知因变量的变化究竟是因为哪个自变量的变化造成的，使得模型更加难以去训练，同时也会对模型的准确性和稳定性产生负面的影响。

共线性检测的意义：

**1.改善模型的稳定性和准确性：**

当模型中存在高度共线的特征时，它们会相互影响，导致模型估计参数的不稳定性和不准确性。参数估计可能会非常敏感，即使是微小的数据变动也可能导致大的变化。

模型的预测性能可能因为不必要的复杂性而下降，尤其是在有限的数据集中。

**2. 提高参数的解释性：**

在没有共线性的情况下，模型的每个参数都可以较清晰地解释其对预测变量的影响。共线性存在时，由于变量之间的高度相关性，很难解释各个参数的具体作用。

**3. 避免过度拟合：**

共线性可能导致模型过度适应训练数据中的特定噪声和错误，从而降低了模型对新数据的泛化能力。

**4.提高计算效率：**

通过识别并处理共线性，可以减少模型中的变量数量，从而减少计算量，提高模型的运行效率。

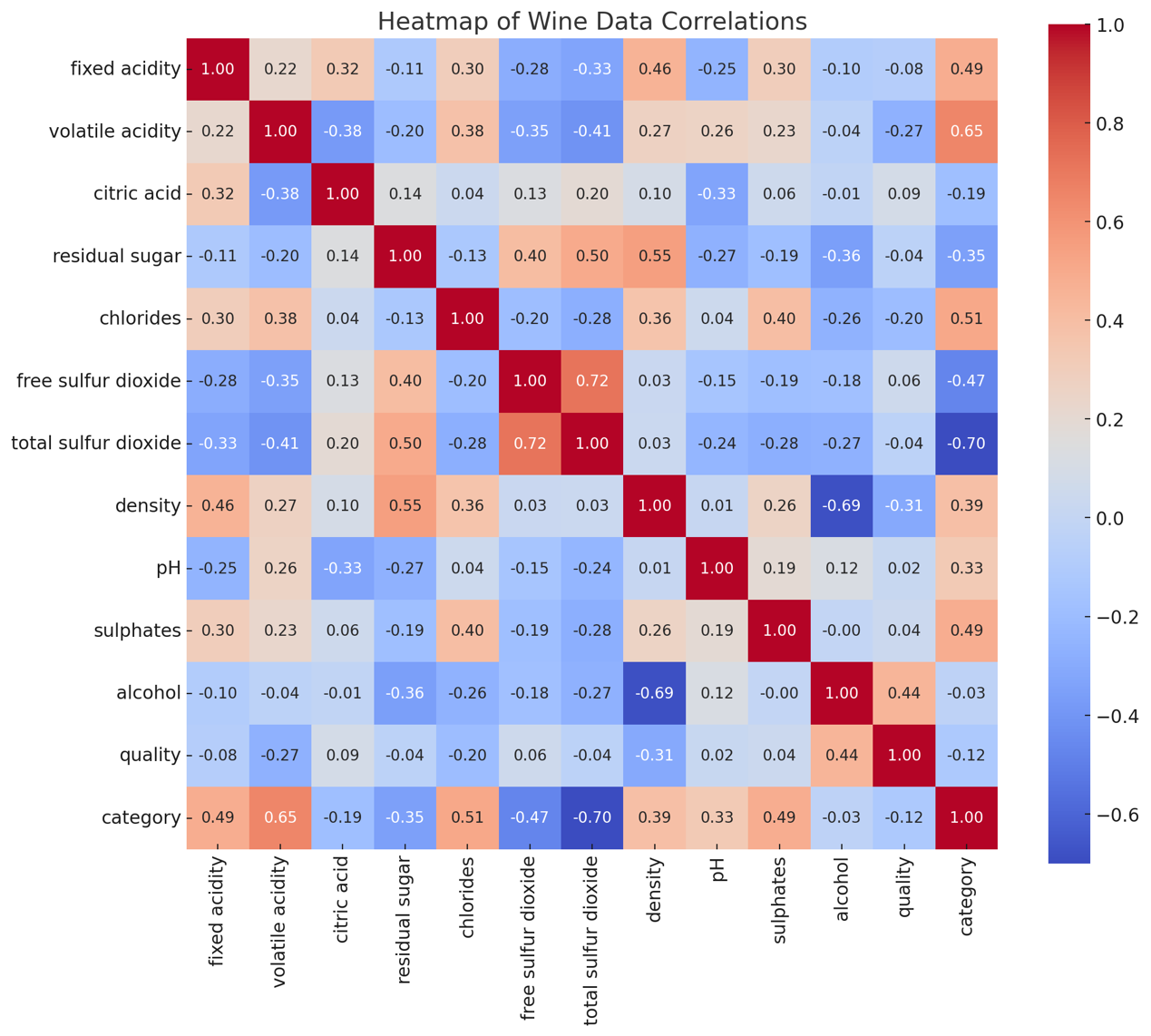
共线性检测的常用方法：

**方差膨胀因子（VIF）：**这是一个量化因子，用来衡量预测变量与其他预测变量线性相关程度的指标。VIF值大于10通常被认为指示有严重的共线性。

**相关系数矩阵：**通过查看预测变量之间的相关系数，可以直观地发现哪些变量之间存在高度相关。

**特征值检查：**在执行主成分分析（PCA）时，特征值接近零可能表明存在共线性。

共线性检测和处理是构建有效和稳健的预测模型的关键步骤。通过识别并解决这些问题，可以确保模型的性能和解释性都达到最优。



在这张葡萄酒数据集的相关系数的热力图当中，列出了葡萄酒的所有的特征，通过相关系数的分析我们可以直观的发现，在葡萄酒的不同特征之间，存在一些线性关系，通过对这些具有高线性关系的特征进行PCA处理，我们可以显著提升模型预测的性能，同时让训练更加迅速和高效。

**三、PCA处理**

PCA（主成分分析）是一种常用的数据降维技术，通过保留数据中的主要特征来简化数据，同时尽量减少信息的丢失。它广泛应用于数据可视化、噪音滤除、特征提取和数据压缩等领域。

PCA 处理的一般步骤：

**1. 标准化数据：**

数据标准化是 PCA 的第一步，目的是确保每个特征的均值为 0，标准差为 1。这一步是必要的，因为它确保了所有特征在分析中具有相同的重要性。

**2. 计算协方差矩阵：**

协方差矩阵描述了数据中各个变量之间的协方差，即它们如何一起变化。如果两个变量增减趋势一致，它们的协方差就是正的；如果一个变量增加而另一个减少，协方差就是负的。

**3. 计算协方差矩阵的特征值和特征向量：**

这一步是 PCA 的核心，特征值和特征向量决定了数据中的主成分（PCs）。特征向量是方向，表示在多维空间中数据最大方差的方向；而特征值则表示该方向的大小，即在那个方向上的方差。

**4. 选择主成分：**

选择顶部的几个主成分。这些通常是对应于最大特征值的特征向量，因为它们解释了大部分数据中的方差。

**5. 构造投影矩阵：**

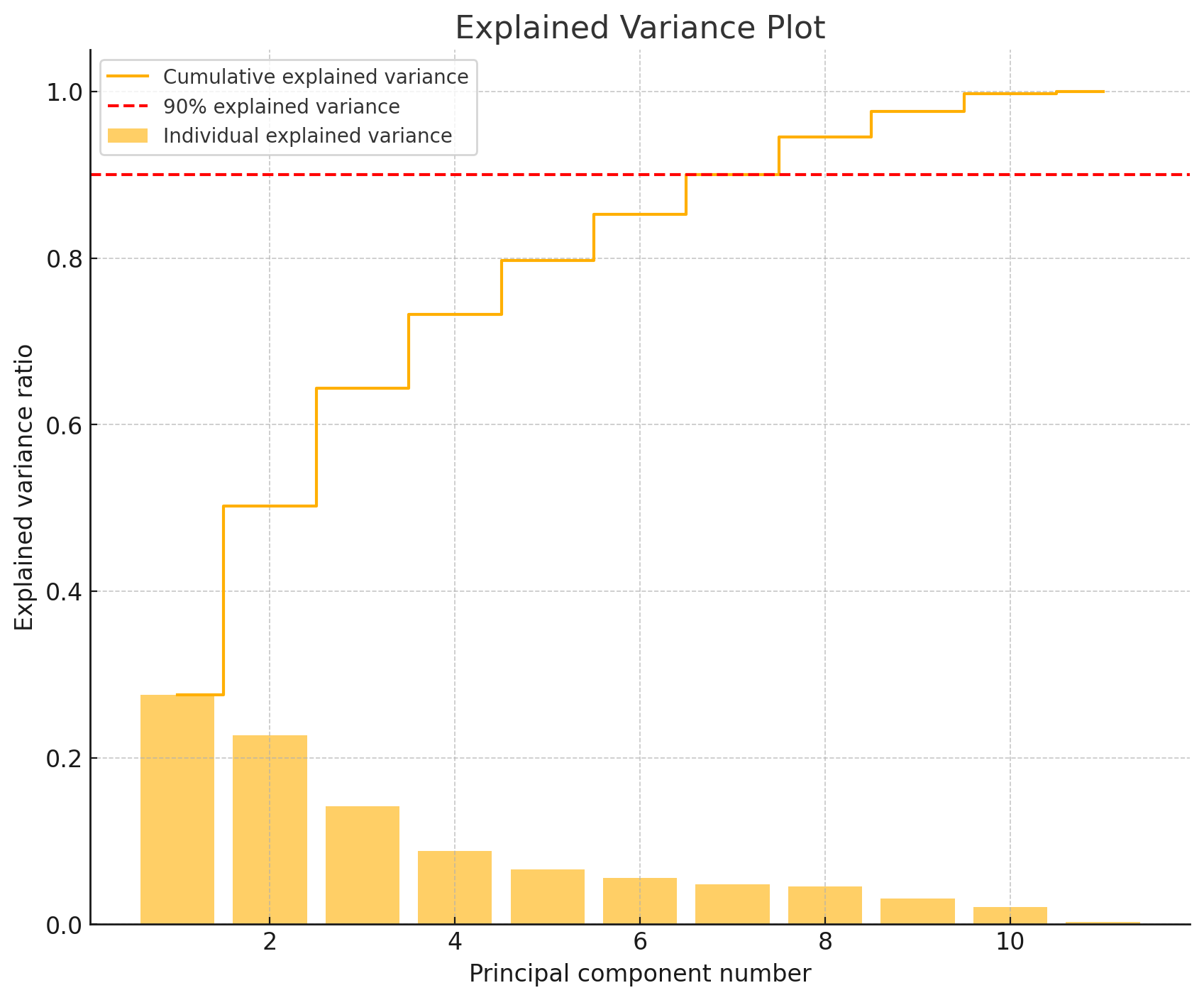
从所选的主成分构造投影矩阵。这个矩阵是一个方向性工具，用于将原始数据转换（投影）到新的特征子空间中。

**6. 转换数据到新的子空间：**

使用投影矩阵将原始数据转换到较低维度的新子空间。这一步通常涉及到数据矩阵和投影矩阵的矩阵乘法，结果是一个新的数据集，其维度少于原始数据集，但仍保留了最重要的统计属性。

通过这些步骤，PCA 使得可以在损失较小的情况下简化数据，非常适用于处理高维数据集。

在下图中，我们绘制了随着保留主成分数量的增加，每个主成分的解释方差以及累计解释方差的变化图，我们设定90%为解释方差的阈值，在图中我们可以看到，对当前数据集，保留7个主成分时，以及达到了90%的阈值，因此，我们可以选择n\_componnets为7进行模型的训练。

**四、LDA处理**

LDA（线性判别分析）是一种统计技术，用于数据分类和降维。它主要用于监督学习场景，其中数据点已被标记为属于两个或多个类别。LDA 的目的是找到一个或多个线性组合的特征，这些特征可以最好地区分不同的类别。这些线性组合被称为“判别式”，它们可以用于数据分类和降维。

LDA 处理的一般步骤：

1. **准备数据：**

确保数据集包含已标记的类别。

分离特征和标签。

**2. 计算类内和类间的均值：**

对于每个类别，计算所有特征的均值。

计算所有数据的总体均值。

**3. 计算类内散度矩阵（Within-class scatter matrix）：**

这个矩阵衡量同一个类别内部的数据点相对于自己类别均值的分散程度。

**4. 计算类间散度矩阵（Between-class scatter matrix）：**

这个矩阵衡量不同类别的均值相对于总体均值的分散程度。

**5. 求解特征值和特征向量：**

构建优化问题以最大化类间散度与类内散度的比例。

通过求解特征值问题找到能最大化上述比例的线性组合。

**6. 选择最重要的线性判别式：**

根据特征值大小选择顶部的线性判别式（特征向量），特征值越大，表示该判别式在区分类别上越有效。

**7. 投影数据：**

使用选定的判别式将原始数据投影到新的空间，通常用于降维。

LDA 不仅帮助减少数据维度，还保持了类别之间最大的区分度，使得在新的低维空间中，同类数据点尽可能接近，不同类数据点尽可能分开。这使得 LDA 在模式识别和机器学习分类任务中特别有用。

**第二部分**

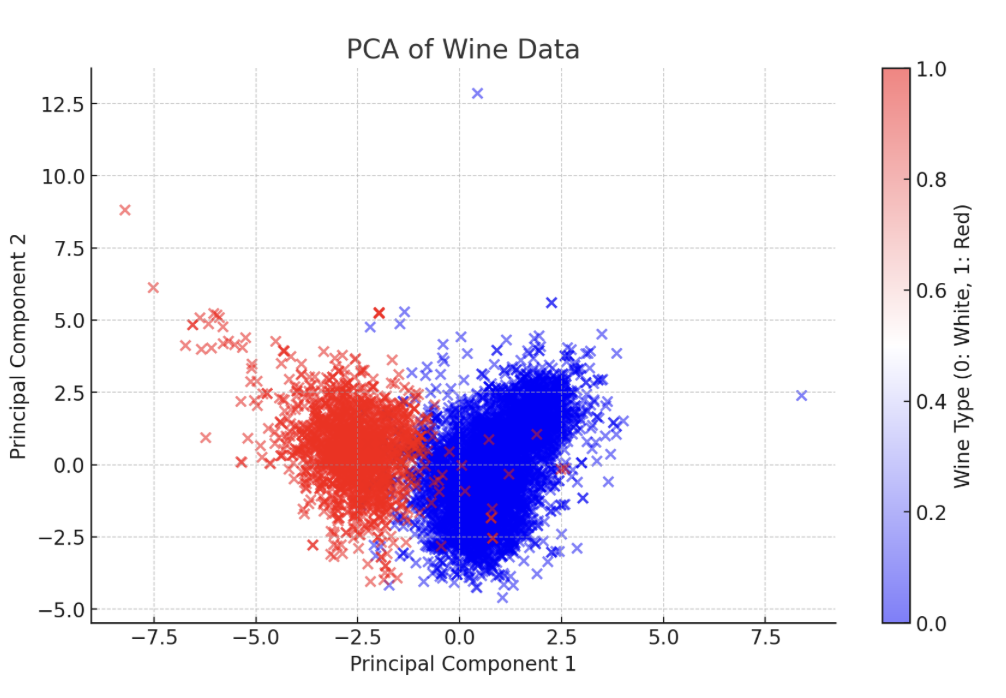
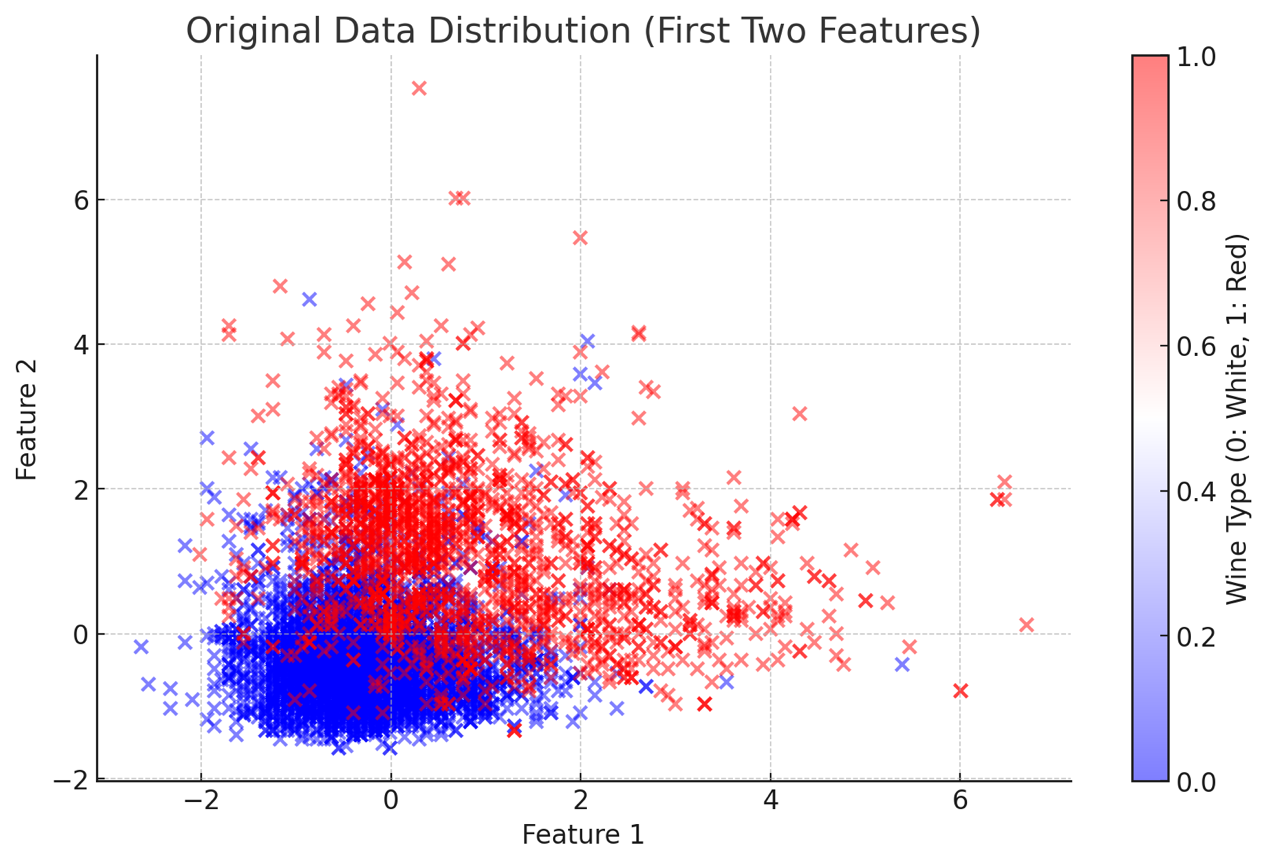
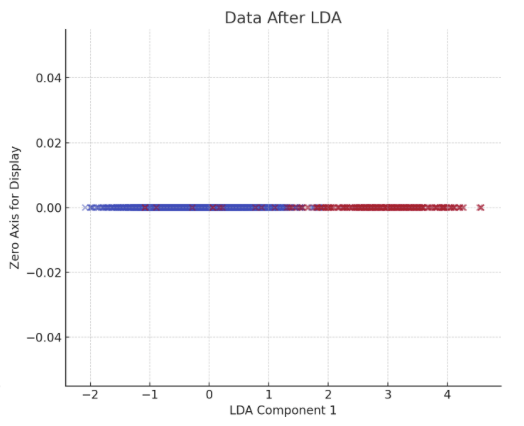
从sklearn.decomposition 中引用PCA 模块，从sklearn.discriminant\_analysis 中引用LinearDiscriminantAnalysis 模块，用sklearn的官方实现，跟自己实现的PCA和LDA算法进行比较。

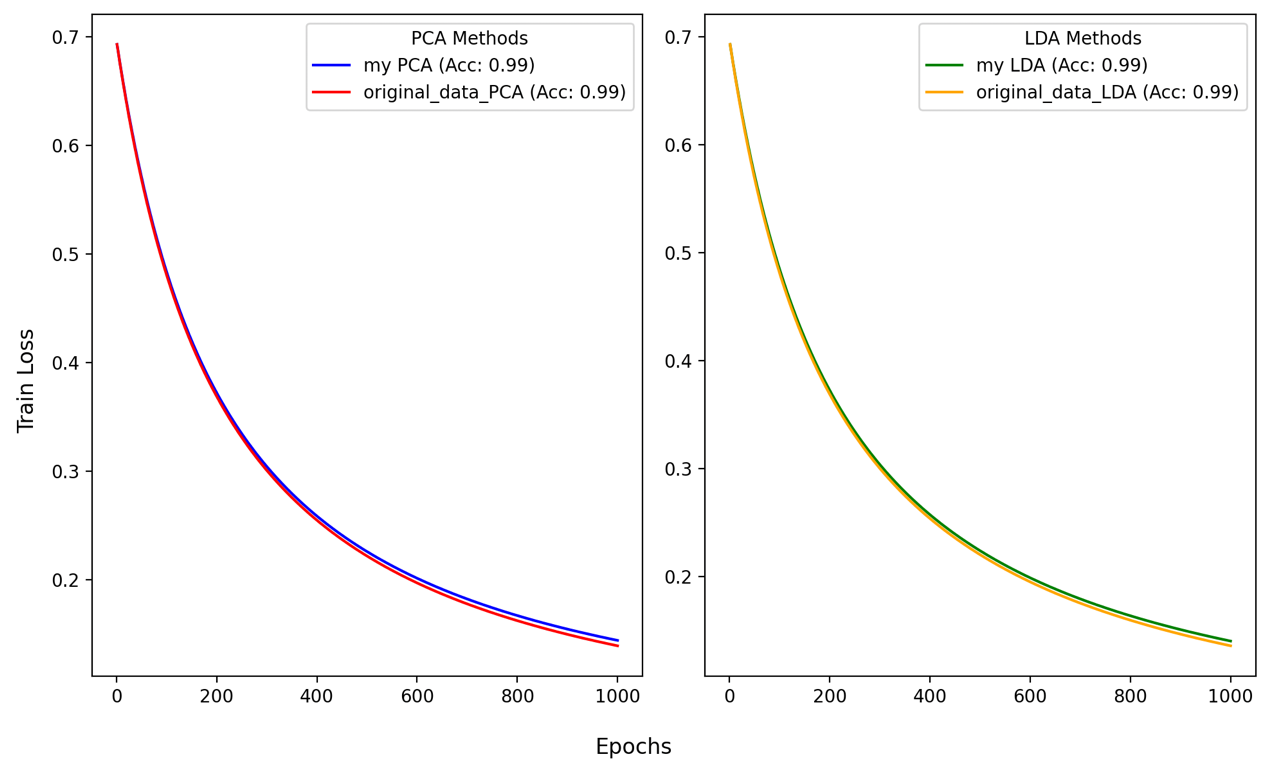
**第三部分**

从sklearn中加载MNIST数据集，因为MNIST数据集跟葡萄酒数据集相比，他不是一个简单的二分类数据集，因此之前的算法不能直接套用在MNIST数据集上，需要进行一系列的处理，比如对于标签需要进行独热编码，需要对输出的结果进行softmax，从而得出概率的结果等。

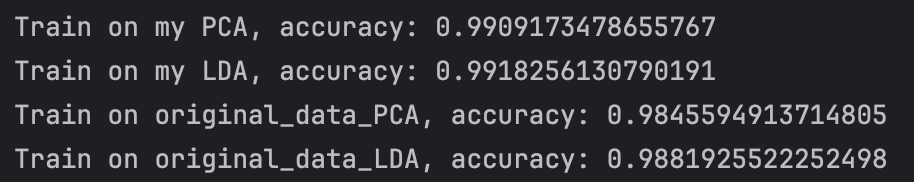
1. **实验结果与分析**

**第一部分**

我们对原始数据集和PCA处理之后的数据集和LDA处理之后的数据集进行了可视化，在三张图片中我们可以清晰的观察到，原数据集中红酒和白酒的数据点混杂在一起，不利于进行分类，而在PCA处理之后，我们可以发现，绝大部分的数据点都已经被明确区分了出来。LDA也成功将数据投影到一个方向上，使得不同的类别能够更好地区分出来。

图中展示了在原始数据集、PCA处理后的数据集和LDA处理后的数据集上进行训练时，模型随训练轮次的增加的收敛情况。



我们可以看到，在采取了PCA和LDA之后，模型在数据集上的预测准确率都获得了提升。

**第二部分**

在实验的第二部分，要求我们对自己实现的PCA和LDA算法和sklearn库中的PCA和LDA算法进行比较。

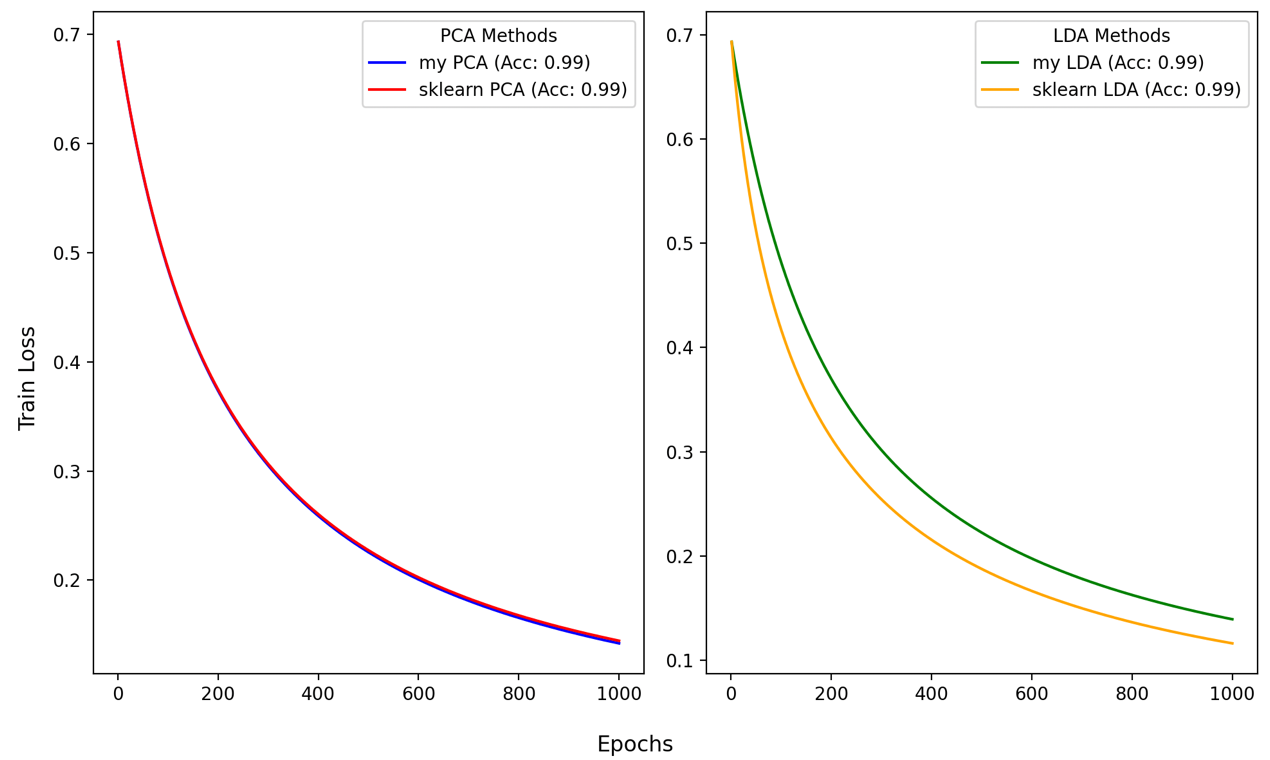
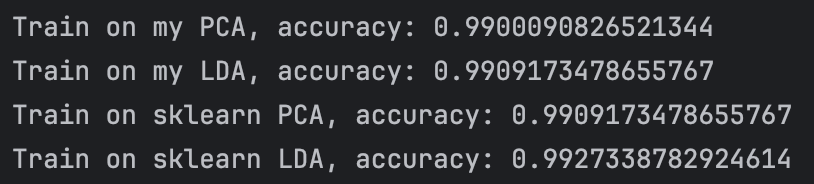
从图中我们可以看到，在模型预测准确度方面，我们自己实现的算法的准确率已经和sklearn库中的算法的准确率接近了，在收敛速度方面，两个PCA算法的收敛速度大致相近，LDA算法的收敛速度有一些差别，可能的原因是：

**1. 算法实现细节：**Scikit-learn的LDA实现进行了多种优化，以确保高效率和数值稳定性。例如，它可能包括更高效的矩阵运算优化、更精确的数值处理方法（例如更精确的逆矩阵计算或特征值分解），以及处理数据特征缩放和奇异值问题的技术。

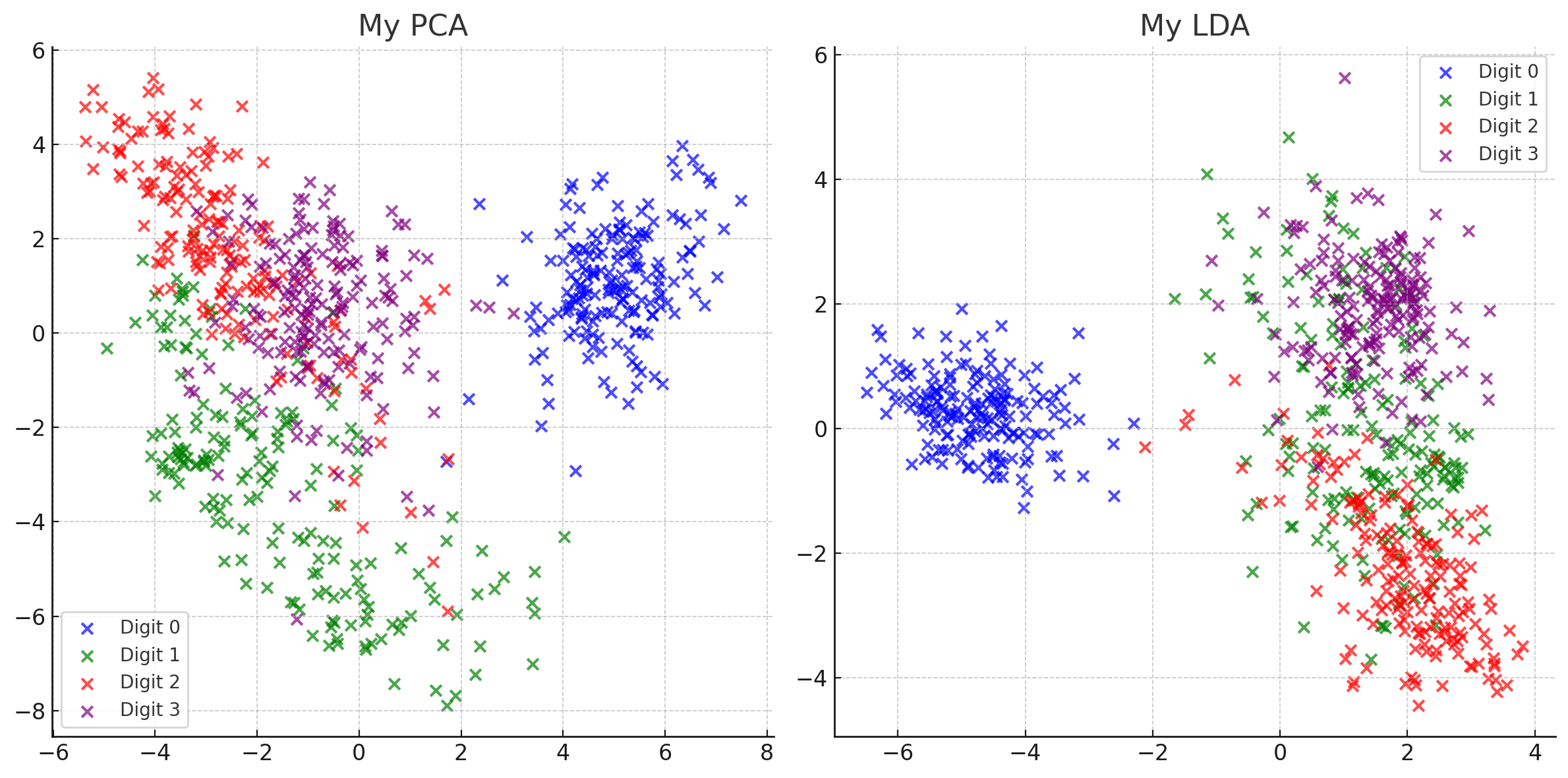
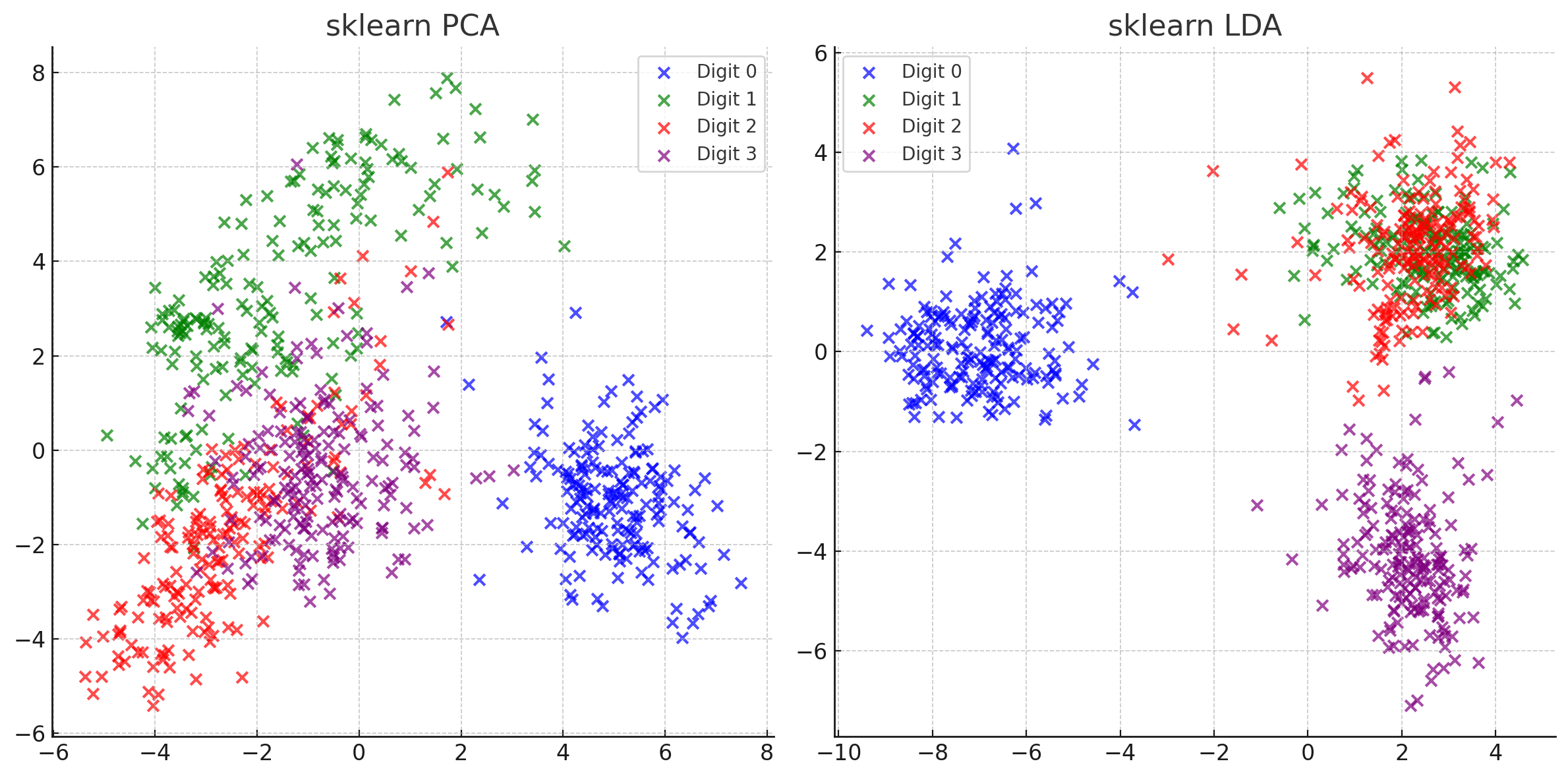
**2. 数值稳定性：**我的实现可能没有处理一些数值稳定性问题，如奇异矩阵（当类内散度矩阵非满秩时）。Scikit-learn的实现通过添加正则化项来确保逆矩阵的计算是稳定的。

**3. 收敛条件和迭代次数：**Scikit-learn的LDA可能使用了不同的收敛判定标准或迭代控制，使得算法在达到较低损失时更快停止。

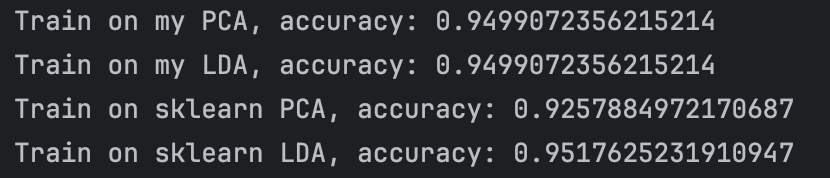
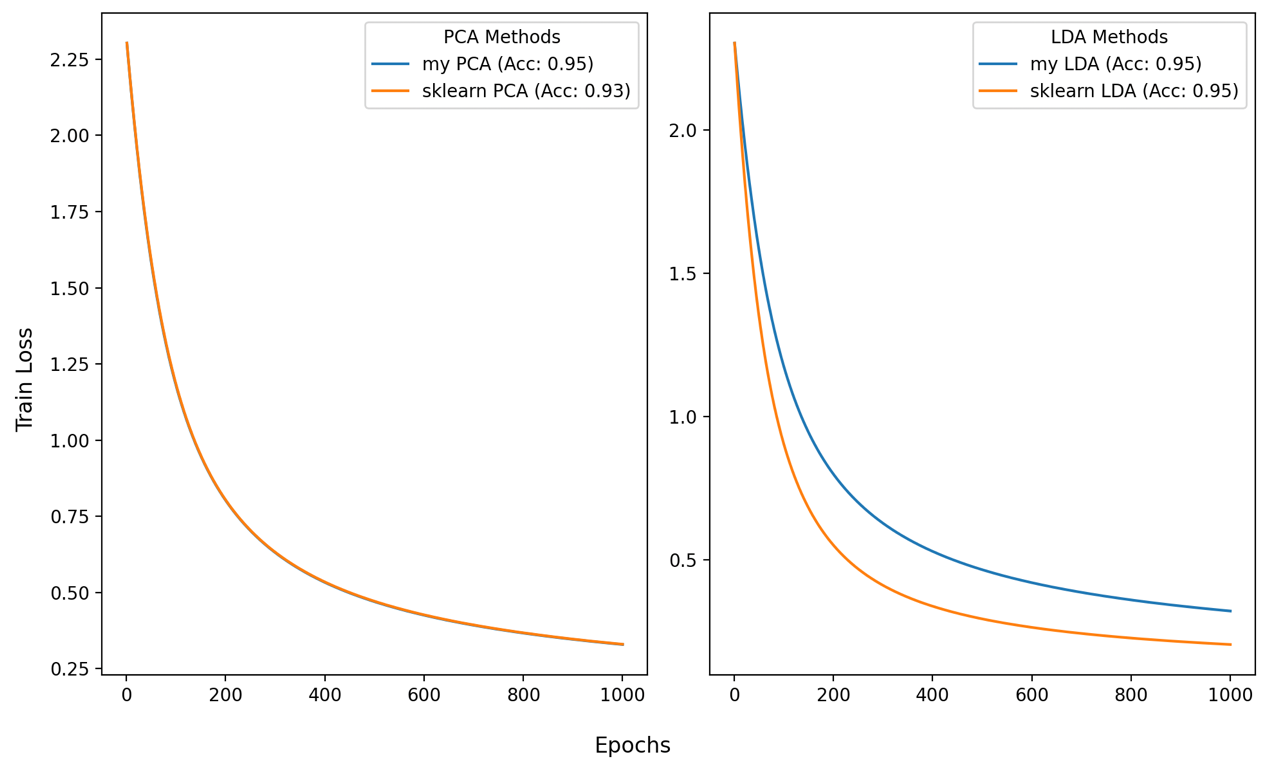
**4. 类间和类内散度矩阵的计算：**可能存在于类间散度和类内散度的计算差异，这直接影响LDA的效果和损失的下降速度。

****

**第三部分**

**** ****

在PCA方面，我自己实现的算法和sklearn库当中的PCA算法对MNIST数据集的分类能力较为接近，在LDA方面，可以看到虽然我自己实现的LDA算法展现出强大的分类能力，但是跟sklearn库中的LDA算法相比，还是有一些差距。sklearn可能在内部使用了更精细的数学运算或优化算法，例如更有效的协方差矩阵计算和特征向量提取，可能导致其结果更为紧凑和区分度更高。



从图中我们可以看到，尽管更换到了MNIST数据集，我们自己实现的算法在模型预测准确度方面和sklearn库中的算法的准确率依然很接近，在收敛速度方面，两个PCA算法的收敛速度大致相近，仅有LDA算法的收敛速度有一些差别。

1. **代码附录**

**train\_man.py**

import pandas as pd  
import matplotlib.pyplot as plt  
import numpy as np  
  
  
# 自定义的LogisticRegression类  
class LogisticRegression:  
 def \_\_init\_\_(self, learning\_rate=0.01, n\_iter=1000):  
 self.learning\_rate = learning\_rate  
 self.n\_iter = n\_iter  
 self.weights = None  
 self.bias = None  
 self.history = []  
  
 def \_sigmoid(self, z):  
 return 1 / (1 + np.exp(-z))  
  
 def \_compute\_loss(self, y, y\_pred):  
 m = y.shape[0]  
 return -np.sum(y \* np.log(y\_pred) + (1 - y) \* np.log(1 - y\_pred)) / m  
  
 def fit(self, X, y):  
 n\_samples, n\_features = X.shape  
 self.weights = np.zeros(n\_features)  
 self.bias = 0  
  
 for \_ in range(self.n\_iter):  
 linear\_model = np.dot(X, self.weights) + self.bias  
 y\_pred = self.\_sigmoid(linear\_model)  
 errors = y - y\_pred  
 dw = np.dot(X.T, errors) / n\_samples  
 db = np.sum(errors) / n\_samples  
 self.weights += self.learning\_rate \* dw  
 self.bias += self.learning\_rate \* db  
  
 # 计算并记录损失，以字典形式存储  
 loss = self.\_compute\_loss(y, y\_pred)  
 self.history.append({'train loss': loss})  
  
 def predict\_proba(self, X):  
 linear\_output = np.dot(X, self.weights) + self.bias  
 return self.\_sigmoid(linear\_output)  
  
 def predict(self, X):  
 probabilities = self.predict\_proba(X)  
 return np.where(probabilities >= 0.5, 1, 0)  
  
# 自定义的accuracy函数  
def accuracy(y\_true, y\_pred):  
 length=len(y\_true)  
 return sum(y\_true == y\_pred) / length  
  
  
# 自定义的PCA类  
class my\_PCA:  
 def \_\_init\_\_(self, n\_components):  
 self.n\_components = n\_components  
 self.components = None  
 self.mean = None  
  
 def fit\_transform(self, X):  
 # 计算数据的平均值，并中心化数据  
 self.mean = np.mean(X, axis=0)  
 X\_centered = X - self.mean  
  
 # 计算协方差矩阵  
 covariance\_matrix = np.cov(X\_centered, rowvar=False)  
  
 # 计算协方差矩阵的特征值和特征向量  
 eigenvalues, eigenvectors = np.linalg.eigh(covariance\_matrix)  
  
 # 对特征值进行排序，并选取最大的n\_components个特征向量  
 sorted\_indices = np.argsort(eigenvalues)[::-1]  
 selected\_eigenvectors = eigenvectors[:, sorted\_indices][:, :self.n\_components]  
  
 # 将数据投影到选定的特征向量上  
 self.components = selected\_eigenvectors.T  
 X\_reduced = X\_centered.dot(self.components.T)  
 return X\_reduced  
  
  
  
class my\_LinearDiscriminantAnalysis:  
 def \_\_init\_\_(self):  
 self.means\_ = None  
 self.scalings\_ = None  
  
 def fit\_transform(self, X, y):  
 # 计算每个类的平均值  
 class\_labels = np.unique(y)  
 mean\_overall = np.mean(X, axis=0)  
 S\_W = np.zeros((X.shape[1], X.shape[1]))  
 S\_B = np.zeros((X.shape[1], X.shape[1]))  
  
 for label in class\_labels:  
 X\_c = X[y == label]  
 mean\_c = np.mean(X\_c, axis=0)  
 S\_W += np.cov(X\_c, rowvar=False) \* (X\_c.shape[0] - 1)  
 mean\_diff = (mean\_c - mean\_overall).reshape(-1, 1)  
 S\_B += X\_c.shape[0] \* (mean\_diff @ mean\_diff.T)  
  
 # S\_W为类内散度矩阵，S\_B为类间散度矩阵  
 S\_W\_inv = np.linalg.inv(S\_W)  
 eigvals, eigvecs = np.linalg.eigh(S\_W\_inv @ S\_B)  
  
 # 排序并选取特征向量  
 sorted\_indices = np.argsort(eigvals)[::-1]  
 self.scalings\_ = eigvecs[:, sorted\_indices]  
  
 # 投影数据  
 X\_transformed = X @ self.scalings\_  
 return X\_transformed  
  
  
def remove\_outliers(df, n=1.5):  
 *"""Remove outliers from a dataframe based on the IQR method."""* for col in df.columns[:-1]: # 假设最后一列是标签，不进行处理  
 Q1 = df[col].quantile(0.25)  
 Q3 = df[col].quantile(0.75)  
 IQR = Q3 - Q1  
 filter = (df[col] >= (Q1 - n \* IQR)) & (df[col] <= (Q3 + n \* IQR))  
 df = df.loc[filter]  
 df.reset\_index(drop=True, inplace=True)  
 return df  
  
  
def standardize\_features(df):  
 *"""Standardize features to have zero mean and unit variance."""* features = df.iloc[:, :-1]  
 standardized\_features = (features - features.mean()) / features.std()  
 df.iloc[:, :-1] = standardized\_features  
 return df  
  
  
def train\_test\_split(X, y):  
 *"""手动实现的简单的训练测试数据集分割函数。  
  
 参数:  
 X -- 特征数据集  
 y -- 标签数据集  
  
 返回:  
 X\_train, X\_test, y\_train, y\_test -- 分割后的训练和测试数据集  
 """* # 确保数据和标签的长度一致  
 assert len(X) == len(y), "特征和标签的长度必须相同。"  
  
 # 计算测试集的大小（30%）  
 test\_size = int(len(X) \* 0.3)  
  
 # 生成随机的索引  
 indices = np.arange(len(X))  
 np.random.shuffle(indices)  
  
 # 根据随机索引划分训练集和测试集  
 X\_test = X[indices[:test\_size]]  
 y\_test = y[indices[:test\_size]]  
 X\_train = X[indices[test\_size:]]  
 y\_train = y[indices[test\_size:]]  
  
 return X\_train, X\_test, y\_train, y\_test  
  
  
n\_components = 9 # 保留9个主成分  
data = {}  
  
# 对原始数据集进行合并处理，红酒和白酒数据集合并之后增加一列category，表示酒的类别，白酒为0，红酒为1  
df\_wine\_white = pd.read\_csv(r'data/winequality-white.csv', header=0,  
 sep=';').drop\_duplicates().assign(category=0)  
df\_wine\_red = pd.read\_csv(r'data/winequality-red.csv', header=0,  
 sep=';').drop\_duplicates().assign(category=1)  
df\_wine = pd.concat([df\_wine\_white, df\_wine\_red], axis=0)  
  
# 移除异常值  
df\_wine = remove\_outliers(df\_wine)  
  
# 标准化特征  
df\_wine = standardize\_features(df\_wine)  
  
# 提取特征和标签  
X = df\_wine.iloc[:, :-1].to\_numpy()  
y = df\_wine.iloc[:, -1].to\_numpy()  
  
  
data['my PCA'] = my\_PCA(n\_components=n\_components).fit\_transform(X)  
data['my LDA'] = my\_LinearDiscriminantAnalysis().fit\_transform(X, y)  
data['original\_data\_PCA'] = X.copy()  
data['original\_data\_LDA'] = df\_wine.to\_numpy()  
  
  
def plot\_result(data):  
 # 设置图表大小和子图布局  
 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 6))  
 fig.supxlabel("Epochs")  
 fig.supylabel("Train Loss")  
  
 # 颜色列表  
 colors = ['blue', 'green', 'red', 'orange', 'brown']  
  
 # 遍历数据集，分别训练使用PCA和其他方法处理的数据  
 color\_index = 0 # 初始化颜色索引  
 for label in data:  
 # 初始化模型并划分数据  
 model = LogisticRegression()  
 X\_train, X\_test, y\_train, y\_test = train\_test\_split(data[label], y)  
 model.fit(X\_train, y\_train)  
  
 # 输出训练结果的准确率  
 accuracy\_val = accuracy(y\_test, model.predict(X\_test))  
 print(f'Train on {label}, accuracy: {accuracy\_val}')  
  
 # 获取训练过程中的损失历史  
 history = [x['train loss'] for x in model.history]  
  
 # 选择合适的子图来绘制历史数据  
 ax = ax1 if 'PCA' in label else ax2  
 ax.plot(range(1, len(history) + 1), history, color=colors[color\_index % len(colors)], label=f'{label} (Acc: {accuracy\_val:.2f})')  
 color\_index += 1 # 更新颜色索引  
  
 # 设置图例和图形布局  
 ax1.legend(title='PCA Methods')  
 ax2.legend(title='LDA Methods')  
 fig.tight\_layout()  
 plt.show()  
  
  
plot\_result(data)

**train\_sklearn.py**

import pandas as pd  
import matplotlib.pyplot as plt  
from sklearn.decomposition import PCA  
from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis  
import numpy as np  
  
  
# 自定义的LogisticRegression类  
class LogisticRegression:  
 def \_\_init\_\_(self, learning\_rate=0.01, n\_iter=1000):  
 self.learning\_rate = learning\_rate  
 self.n\_iter = n\_iter  
 self.weights = None  
 self.bias = None  
 self.history = []  
  
 def \_sigmoid(self, z):  
 return 1 / (1 + np.exp(-z))  
  
 def \_compute\_loss(self, y, y\_pred):  
 m = y.shape[0]  
 return -np.sum(y \* np.log(y\_pred) + (1 - y) \* np.log(1 - y\_pred)) / m  
  
 def fit(self, X, y):  
 n\_samples, n\_features = X.shape  
 self.weights = np.zeros(n\_features)  
 self.bias = 0  
  
 for \_ in range(self.n\_iter):  
 linear\_model = np.dot(X, self.weights) + self.bias  
 y\_pred = self.\_sigmoid(linear\_model)  
 errors = y - y\_pred  
 dw = np.dot(X.T, errors) / n\_samples  
 db = np.sum(errors) / n\_samples  
 self.weights += self.learning\_rate \* dw  
 self.bias += self.learning\_rate \* db  
  
 # 计算并记录损失，以字典形式存储  
 loss = self.\_compute\_loss(y, y\_pred)  
 self.history.append({'train loss': loss})  
  
 def predict\_proba(self, X):  
 linear\_output = np.dot(X, self.weights) + self.bias  
 return self.\_sigmoid(linear\_output)  
  
 def predict(self, X):  
 probabilities = self.predict\_proba(X)  
 return np.where(probabilities >= 0.5, 1, 0)  
  
# 自定义的accuracy函数  
def accuracy(y\_true, y\_pred):  
 length=len(y\_true)  
 return sum(y\_true == y\_pred) / length  
  
  
# 自定义的PCA类  
class my\_PCA:  
 def \_\_init\_\_(self, n\_components):  
 self.n\_components = n\_components  
 self.components = None  
 self.mean = None  
  
 def fit\_transform(self, X):  
 # 计算数据的平均值，并中心化数据  
 self.mean = np.mean(X, axis=0)  
 X\_centered = X - self.mean  
  
 # 计算协方差矩阵  
 covariance\_matrix = np.cov(X\_centered, rowvar=False)  
  
 # 计算协方差矩阵的特征值和特征向量  
 eigenvalues, eigenvectors = np.linalg.eigh(covariance\_matrix)  
  
 # 对特征值进行排序，并选取最大的n\_components个特征向量  
 sorted\_indices = np.argsort(eigenvalues)[::-1]  
 selected\_eigenvectors = eigenvectors[:, sorted\_indices][:, :self.n\_components]  
  
 # 将数据投影到选定的特征向量上  
 self.components = selected\_eigenvectors.T  
 X\_reduced = X\_centered.dot(self.components.T)  
 return X\_reduced  
  
  
# 请注意，这个代码实现是基础的，并没有包括一些 sklearn 实现中的优化和稳健性处理，  
# 例如对奇异类内散度矩阵的处理。在实际应用中，如果需要考虑计算的稳定性和性能，  
# 建议使用 sklearn.discriminant\_analysis.LinearDiscriminantAnalysis。  
class my\_LinearDiscriminantAnalysis:  
 def \_\_init\_\_(self):  
 self.means\_ = None  
 self.scalings\_ = None  
  
 def fit\_transform(self, X, y):  
 # 计算每个类的平均值  
 class\_labels = np.unique(y)  
 mean\_overall = np.mean(X, axis=0)  
 S\_W = np.zeros((X.shape[1], X.shape[1]))  
 S\_B = np.zeros((X.shape[1], X.shape[1]))  
  
 for label in class\_labels:  
 X\_c = X[y == label]  
 mean\_c = np.mean(X\_c, axis=0)  
 S\_W += np.cov(X\_c, rowvar=False) \* (X\_c.shape[0] - 1)  
 mean\_diff = (mean\_c - mean\_overall).reshape(-1, 1)  
 S\_B += X\_c.shape[0] \* (mean\_diff @ mean\_diff.T)  
  
 # S\_W为类内散度矩阵，S\_B为类间散度矩阵  
 S\_W\_inv = np.linalg.inv(S\_W)  
 eigvals, eigvecs = np.linalg.eigh(S\_W\_inv @ S\_B)  
  
 # 排序并选取特征向量  
 sorted\_indices = np.argsort(eigvals)[::-1]  
 self.scalings\_ = eigvecs[:, sorted\_indices]  
  
 # 投影数据  
 X\_transformed = X @ self.scalings\_  
 return X\_transformed  
  
  
def remove\_outliers(df, n=1.5):  
 *"""Remove outliers from a dataframe based on the IQR method."""* for col in df.columns[:-1]: # 假设最后一列是标签，不进行处理  
 Q1 = df[col].quantile(0.25)  
 Q3 = df[col].quantile(0.75)  
 IQR = Q3 - Q1  
 filter = (df[col] >= (Q1 - n \* IQR)) & (df[col] <= (Q3 + n \* IQR))  
 df = df.loc[filter]  
 df.reset\_index(drop=True, inplace=True)  
 return df  
  
  
def standardize\_features(df):  
 *"""Standardize features to have zero mean and unit variance."""* features = df.iloc[:, :-1]  
 standardized\_features = (features - features.mean()) / features.std()  
 df.iloc[:, :-1] = standardized\_features  
 return df  
  
  
def train\_test\_split(X, y):  
 *"""手动实现的简单的训练测试数据集分割函数。  
  
 参数:  
 X -- 特征数据集  
 y -- 标签数据集  
  
 返回:  
 X\_train, X\_test, y\_train, y\_test -- 分割后的训练和测试数据集  
 """* # 确保数据和标签的长度一致  
 assert len(X) == len(y), "特征和标签的长度必须相同。"  
  
 # 计算测试集的大小（30%）  
 test\_size = int(len(X) \* 0.3)  
  
 # 生成随机的索引  
 indices = np.arange(len(X))  
 np.random.shuffle(indices)  
  
 # 根据随机索引划分训练集和测试集  
 X\_test = X[indices[:test\_size]]  
 y\_test = y[indices[:test\_size]]  
 X\_train = X[indices[test\_size:]]  
 y\_train = y[indices[test\_size:]]  
  
 return X\_train, X\_test, y\_train, y\_test  
  
  
n\_components = 9 # 保留9个主成分  
data = {}  
  
# 对原始数据集进行合并处理，红酒和白酒数据集合并之后增加一列category，表示酒的类别，白酒为0，红酒为1  
df\_wine\_white = pd.read\_csv(r'data/winequality-white.csv', header=0,  
 sep=';').drop\_duplicates().assign(category=0)  
df\_wine\_red = pd.read\_csv(r'data/winequality-red.csv', header=0,  
 sep=';').drop\_duplicates().assign(category=1)  
df\_wine = pd.concat([df\_wine\_white, df\_wine\_red], axis=0)  
  
# 移除异常值  
df\_wine = remove\_outliers(df\_wine)  
  
# 标准化特征  
df\_wine = standardize\_features(df\_wine)  
  
# 提取特征和标签  
X = df\_wine.iloc[:, :-1].to\_numpy()  
y = df\_wine.iloc[:, -1].to\_numpy()  
  
  
data['my PCA'] = my\_PCA(n\_components=n\_components).fit\_transform(X)  
data['my LDA'] = my\_LinearDiscriminantAnalysis().fit\_transform(X, y)  
data['sklearn PCA'] = PCA(n\_components=n\_components).fit\_transform(X)  
data['sklearn LDA'] = LinearDiscriminantAnalysis().fit\_transform(X, y)  
  
  
def plot\_result(data):  
 # 设置图表大小和子图布局  
 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 6))  
 fig.supxlabel("Epochs")  
 fig.supylabel("Train Loss")  
  
 # 颜色列表  
 colors = ['blue', 'green', 'red', 'orange', 'brown']  
  
 # 遍历数据集，分别训练使用PCA和其他方法处理的数据  
 color\_index = 0 # 初始化颜色索引  
 for label in data:  
 # 初始化模型并划分数据  
 model = LogisticRegression()  
 X\_train, X\_test, y\_train, y\_test = train\_test\_split(data[label], y)  
 model.fit(X\_train, y\_train)  
  
 # 输出训练结果的准确率  
 accuracy\_val = accuracy(y\_test, model.predict(X\_test))  
 print(f'Train on {label}, accuracy: {accuracy\_val}')  
  
 # 获取训练过程中的损失历史  
 history = [x['train loss'] for x in model.history]  
  
 # 选择合适的子图来绘制历史数据  
 ax = ax1 if 'PCA' in label else ax2  
 ax.plot(range(1, len(history) + 1), history, color=colors[color\_index % len(colors)], label=f'{label} (Acc: {accuracy\_val:.2f})')  
 color\_index += 1 # 更新颜色索引  
  
 # 设置图例和图形布局  
 ax1.legend(title='PCA Methods')  
 ax2.legend(title='LDA Methods')  
 fig.tight\_layout()  
 plt.show()  
  
  
plot\_result(data)

**train\_mnist.py**

# 引用了来自sklearn的数据集mnist  
import pandas as pd  
from sklearn.datasets import load\_digits  
import matplotlib.pyplot as plt  
from sklearn.decomposition import PCA  
from sklearn.linear\_model import LogisticRegression  
import numpy as np  
from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis, StandardScaler  
  
  
# 自定义的LogisticRegression类  
class my\_LogisticRegression:  
 def \_\_init\_\_(self, learning\_rate=0.01, n\_iter=1000, decay=0.0, reg\_lambda=0.01):  
 *"""  
 多类逻辑回归模型初始化。  
 :param lr: 学习率  
 :param epochs: 迭代次数  
 :param decay: 学习率衰减  
 :param reg\_lambda: 正则化参数  
 """* self.learning\_rate = learning\_rate  
 self.n\_iter = n\_iter  
 self.decay = decay  
 self.reg\_lambda = reg\_lambda  
 self.weights = None  
 self.bias = None  
 self.history = []  
  
 def fit(self, X, y):  
 *"""  
 拟合多类逻辑回归模型。  
 :param X: 特征数据  
 :param y: 标签数据  
 """* num\_samples, num\_features = X.shape  
 num\_classes = np.max(y) + 1  
 self.weights = np.zeros((num\_features, num\_classes))  
 self.bias = np.zeros(num\_classes)  
 y\_encoded = self.\_one\_hot\_encode(y, num\_classes)  
  
 for epoch in range(self.n\_iter):  
 model = np.dot(X, self.weights) + self.bias  
 y\_pred = self.\_softmax(model)  
 loss = self.\_compute\_loss(y\_encoded, y\_pred)  
 self.history.append({'train loss': loss})  
  
 grad\_weights = np.dot(X.T, (y\_pred - y\_encoded)) / num\_samples + self.reg\_lambda \* self.weights  
 grad\_bias = np.mean(y\_pred - y\_encoded, axis=0)  
  
 self.weights -= self.learning\_rate \* grad\_weights  
 self.bias -= self.learning\_rate \* grad\_bias  
  
 self.learning\_rate \*= (1. / (1. + self.decay \* epoch))  
  
 def predict(self, X):  
 *"""  
 预测给定数据的类别标签。  
 :param X: 特征数据  
 """* model = np.dot(X, self.weights) + self.bias  
 y\_preds = self.\_softmax(model)  
 return np.argmax(y\_preds, axis=1)  
  
 def \_softmax(self, z):  
 e = np.exp(z - np.max(z, axis=1, keepdims=True))  
 return e / np.sum(e, axis=1, keepdims=True)  
  
 def \_compute\_loss(self, y, y\_pred):  
 epsilon = 1e-10  
 y\_pred = np.clip(y\_pred, epsilon, 1.0 - epsilon)  
 return -np.mean(np.sum(y \* np.log(y\_pred), axis=1))  
  
 def \_one\_hot\_encode(self, y, num\_classes):  
 return np.eye(num\_classes)[y].astype(float)  
  
  
# 自定义的accuracy函数  
def my\_accuracy(y\_true, y\_pred):  
 length=len(y\_true)  
 return sum(y\_true == y\_pred) / length  
  
  
# 自定义的PCA类  
class my\_PCA:  
 def \_\_init\_\_(self, n\_components):  
 self.n\_components = n\_components  
 self.components = None  
 self.mean = None  
  
 def fit\_transform(self, X):  
 # 计算数据的平均值，并中心化数据  
 self.mean = np.mean(X, axis=0)  
 X\_centered = X - self.mean  
  
 # 计算协方差矩阵  
 covariance\_matrix = np.cov(X\_centered, rowvar=False)  
  
 # 计算协方差矩阵的特征值和特征向量  
 eigenvalues, eigenvectors = np.linalg.eigh(covariance\_matrix)  
  
 # 对特征值进行排序  
 sorted\_indices = np.argsort(eigenvalues)[::-1]  
 sorted\_eigenvalues = eigenvalues[sorted\_indices]  
 sorted\_eigenvectors = eigenvectors[:, sorted\_indices]  
  
 # 如果n\_components是浮点数，解释为方差的百分比  
 if isinstance(self.n\_components, float):  
 total\_variance = np.sum(sorted\_eigenvalues)  
 variance\_ratio = sorted\_eigenvalues / total\_variance  
 cumulative\_variance = np.cumsum(variance\_ratio)  
 # 找到累积贡献率至少为n\_components的特征数  
 num\_components = np.where(cumulative\_variance >= self.n\_components)[0][0] + 1  
 self.components = sorted\_eigenvectors[:, :num\_components]  
 else:  
 # 否则，直接选取前n\_components个特征向量  
 self.components = sorted\_eigenvectors[:, :self.n\_components]  
  
 # 将数据投影到选定的特征向量上  
 X\_reduced = X\_centered.dot(self.components)  
 return X\_reduced  
  
  
class my\_LinearDiscriminantAnalysis:  
 def \_\_init\_\_(self):  
 self.means\_ = None  
 self.scalings\_ = None  
  
 def fit\_transform(self, X, y):  
 # 计算每个类的平均值  
 class\_labels = np.unique(y)  
 mean\_overall = np.mean(X, axis=0)  
 S\_W = np.zeros((X.shape[1], X.shape[1]))  
 S\_B = np.zeros((X.shape[1], X.shape[1]))  
  
 for label in class\_labels:  
 X\_c = X[y == label]  
 mean\_c = np.mean(X\_c, axis=0)  
 S\_W += np.cov(X\_c, rowvar=False) \* (X\_c.shape[0] - 1)  
 mean\_diff = (mean\_c - mean\_overall).reshape(-1, 1)  
 S\_B += X\_c.shape[0] \* (mean\_diff @ mean\_diff.T)  
  
 # 添加正则化以避免奇异矩阵  
 lambda\_ = 1e-5 # 正则化系数  
 S\_W += lambda\_ \* np.eye(S\_W.shape[0])  
  
 # S\_W为类内散度矩阵，S\_B为类间散度矩阵  
 S\_W\_inv = np.linalg.inv(S\_W)  
 eigvals, eigvecs = np.linalg.eigh(S\_W\_inv @ S\_B)  
  
 # 排序并选取特征向量  
 sorted\_indices = np.argsort(eigvals)[::-1]  
 self.scalings\_ = eigvecs[:, sorted\_indices]  
  
 # 投影数据  
 X\_transformed = X @ self.scalings\_  
 return X\_transformed  
  
# 示例使用  
# lda = my\_LinearDiscriminantAnalysis()  
# X\_transformed = lda.fit\_transform(X, y)  
  
  
def remove\_outliers(df, n=1.5):  
 *"""Remove outliers from a dataframe based on the IQR method."""* for col in df.columns[:-1]: # 假设最后一列是标签，不进行处理  
 Q1 = df[col].quantile(0.25)  
 Q3 = df[col].quantile(0.75)  
 IQR = Q3 - Q1  
 filter = (df[col] >= (Q1 - n \* IQR)) & (df[col] <= (Q3 + n \* IQR))  
 df = df.loc[filter]  
 df.reset\_index(drop=True, inplace=True)  
 return df  
  
  
def standardize\_features(df):  
 *"""Standardize features to have zero mean and unit variance."""* features = df.iloc[:, :-1]  
 standardized\_features = (features - features.mean()) / features.std()  
 df.iloc[:, :-1] = standardized\_features  
 return df  
  
  
def train\_test\_split(X, y):  
 *"""手动实现的简单的训练测试数据集分割函数。  
  
 参数:  
 X -- 特征数据集  
 y -- 标签数据集  
  
 返回:  
 X\_train, X\_test, y\_train, y\_test -- 分割后的训练和测试数据集  
 """* # 确保数据和标签的长度一致  
 assert len(X) == len(y), "特征和标签的长度必须相同。"  
  
 # 计算测试集的大小（30%）  
 test\_size = int(len(X) \* 0.3)  
  
 # 生成随机的索引  
 indices = np.arange(len(X))  
 np.random.shuffle(indices)  
  
 # 根据随机索引划分训练集和测试集  
 X\_test = X[indices[:test\_size]]  
 y\_test = y[indices[:test\_size]]  
 X\_train = X[indices[test\_size:]]  
 y\_train = y[indices[test\_size:]]  
  
 return X\_train, X\_test, y\_train, y\_test  
  
  
n\_components = 0.9  
data = {}  
scaler = StandardScaler()  
  
# 提取特征和标签  
digits = load\_digits()  
X = scaler.fit\_transform(digits.data)  
y = digits.target  
  
  
data['my PCA'] = my\_PCA(n\_components=n\_components).fit\_transform(X)  
data['my LDA'] = my\_LinearDiscriminantAnalysis().fit\_transform(X, y)  
data['sklearn PCA'] = PCA(n\_components=n\_components).fit\_transform(X)  
data['sklearn LDA'] = LinearDiscriminantAnalysis().fit\_transform(X, y)  
  
  
def plot\_result(data):  
 # 设置图表大小和子图布局  
 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 6))  
 fig.supxlabel("Epochs")  
 fig.supylabel("Train Loss")  
  
 # 遍历数据集，分别训练使用PCA和其他方法处理的数据  
 for label in data:  
 # 初始化模型并划分数据  
  
 model = my\_LogisticRegression()  
 X\_train, X\_test, y\_train, y\_test = train\_test\_split(data[label], y)  
 model.fit(X\_train, y\_train)  
  
 # 输出训练结果的准确率  
 accuracy\_val = my\_accuracy(y\_test, model.predict(X\_test))  
 print(f'Train on {label}, accuracy: {accuracy\_val}')  
  
 # 获取训练过程中的损失历史  
 history = [x['train loss'] for x in model.history]  
  
 # 选择合适的子图来绘制历史数据  
 ax = ax1 if 'PCA' in label else ax2  
 ax.plot(range(1, len(history) + 1), history, label=f'{label} (Acc: {accuracy\_val:.2f})')  
  
 # 设置图例和图形布局  
 ax1.legend(title='PCA Methods')  
 ax2.legend(title='LDA Methods')  
 fig.tight\_layout()  
 plt.show()  
  
  
plot\_result(data)

**实验二 KNN分类任务实验**

1. **问题描述**

**概述**

利用 KNN 算法对 Iris 鸢尾花数据集中的测试集进行分类。

**数据说明**

鸢尾花数据集内包含的 3 类分别为山鸢尾（Iris-setosa）、变色鸢尾（Iris-versicolor）和维吉尼亚鸢尾（Iris-virginica），共 150 条记录，每类各 50 个数据，每条记录都有4 项特征：花萼长度、花萼宽度、花瓣长度、花瓣宽度。标签 0、1、2 分别表示山鸢尾、变色鸢尾、维吉尼亚鸢尾。

数据集已被划分为训练集、验证集和测试集，分别存储于 data 文件夹下的 train.csv，val.csv 和 test\_data.csv 文件。其中，train.csv 和 val.csv 包括 data 和 label 字段，分别存储着特征 𝑋 ∈ 𝑅𝑁×𝑑 和标记 𝑌 ∈ 𝑅𝑁×1，*N* 是样例数量，*d* = 4 为特征维度，每个样例的标记 𝑦 ∈ {0,1,2}。test\_data.csv 仅包含 data 字段。

**实验内容**

1. 利用欧式距离作为KNN 算法的度量函数，对测试集进行分类。实验报告中，要求在验证集上分析近邻数 *K* 对KNN 算法分类精度的影响。
2. 利用马氏距离作为KNN 算法的度量函数，对测试集进行分类。在马氏距离中，*M* 为

半正定矩阵，正交基*A* 使得 𝑀 = 𝐴𝐴𝑇 成立。

1. 基于 MindSpore 平台实现算法，对相同的数据集进行训练，并与不使用 MindSpore 平台实现的算法对比结果（包括但不限于准确率、算法迭代收敛次数等指标），并分析结果中出现差异的可能原因，给出使用 MindSpore 的心得和建议。

四、（加分项）使用 MindSpore、sklearn 等平台提供的相似任务数据集（例如，其他的分

类任务数据集）测试自己独立实现的算法，并与 MindSpore、sklearn 等平台上的官方实现算法进行对比，进一步分析差异及其成因。

1. **实现步骤与流程**

**第一部分**

**一、数据预处理**

（1）去除重复数据

与其他模型不同，数据集中的重复数据对KNN模型的影响非常大，重复数据的存在会对分类结果产生显著影响。

如果重复的数据属于同一个类别，可能会导致该类别在多数投票中的权重过大，从而导致模型对该类别的偏好增强。这种情况在类别不平衡的数据集中尤其明显，可能会导致分类器过于偏向于多数类。

重复的数据实质上是在过度强调数据集中某些特定的点，这可能导致模型在训练数据上表现良好，但在未见过的心数据集上表现不佳。KNN算法容易收到这种影响，因为他依赖于训练数据的局部性质。

重复数据点的增多会在距离计算中占据更大的权重，这可能会扭曲实际的最近邻关系。例如，如果一个数据点在数据集中重复多次，那么它更有可能成为最近邻，即使其他更有代表性的数据点距离测试点更近。

重复数据的存在增加了存储需求和计算成本，尤其是在处理大数据集时。KNN算法本身在大数据集上就可能变得计算密集，重复数据的存在会进一步加重这一问题。

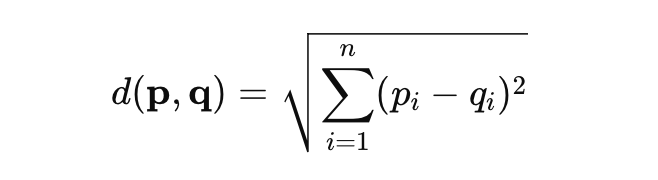
在本次实验用到的Iris数据集中，没有出现重复数据，因此不需要进行去重的工作。

（2）数据标准化

KNN（K-最近邻）算法是一种基于距离的分类算法，它的性能和准确性在很大程度上依赖于如何度量数据点之间的距离。因此，在使用KNN算法之前进行数据标准化是至关重要的，特别是当数据集中的特征具有不同的量级和单位时。

**二、欧氏距离计算**

欧氏距离（Euclidean distance）是在多维空间中两点之间的直线距离，它是最常用的距离度量之一。



**三、KNN具体实现**

K-最近邻（KNN）是一种基于实例的学习算法，常用于分类和回归问题。其工作原理非常简单直观，主要通过测量不同特征值之间的距离来进行预测。

**1. 数据准备**

收集数据：KNN算法首先需要一个已经标记的数据集，这个数据集将作为训练集，用于预测新数据点的分类。

特征选择：选择合适的特征对算法的性能至关重要，因为特征直接影响到后续的距离计算和分类的准确性。

数据预处理：包括标准化或归一化数据，确保不同特征之间的比较是公平的，特别是当各特征的度量单位或量级差异较大时。

**2. 距离度量**

选择距离公式：常用的距离度量包括欧氏距离、曼哈顿距离和闵可夫斯基距离等。根据数据的类型和特点选择最合适的距离度量标准。

计算距离：对于分类问题中的每一个未知样本，计算它与训练集中每一个样本之间的距离。

**3. 寻找最近邻**

确定K值：K值表示邻近点的数量。这是KNN算法中的一个重要参数，它决定了模型的复杂性。

寻找最近的K个邻居：根据计算出的距离，选择距离最近的K个训练样本，这些样本的已知响应值将用来预测新样本的分类。

**4. 决策规则**

分类决策：在分类任务中，常采用多数投票法，即在K个最近邻居中多数类别的标签被赋予未知样本。也可以基于距离权重进行加权投票，即距离较近的邻居对结果的贡献更大。

回归决策：在回归任务中，通常采用平均法，即取K个最近邻居的目标值的平均值作为预测值。同样可以进行距离加权。

**5. 模型评估**

交叉验证：使用交叉验证（如k-fold交叉验证）来评估KNN模型的准确性，这有助于验证模型的泛化能力。

性能调优：调整K值和其他参数，根据验证结果进行模型的优化，以达到更好的分类或预测性能。

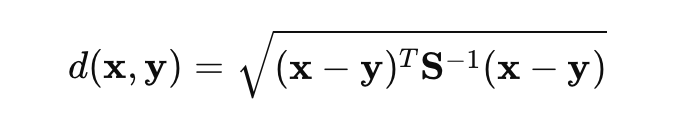
**6. 预测新数据**

应用模型：将经过训练和调优的KNN模型用于新的数据集进行预测。

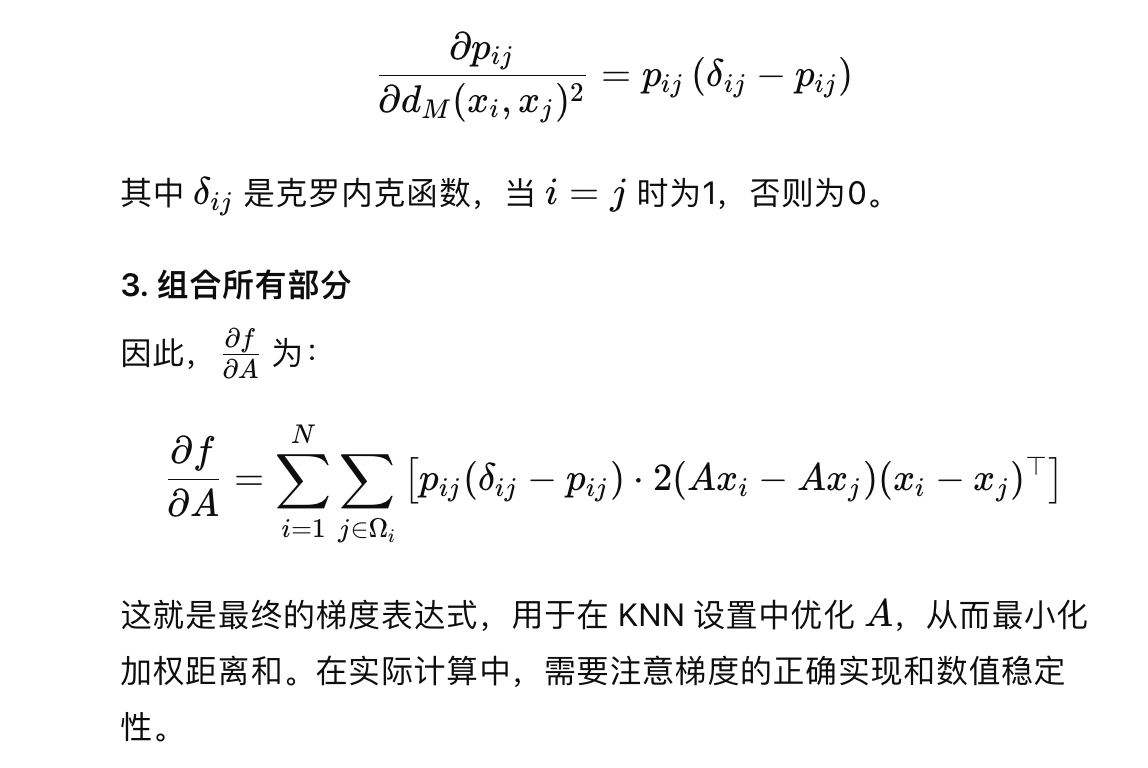
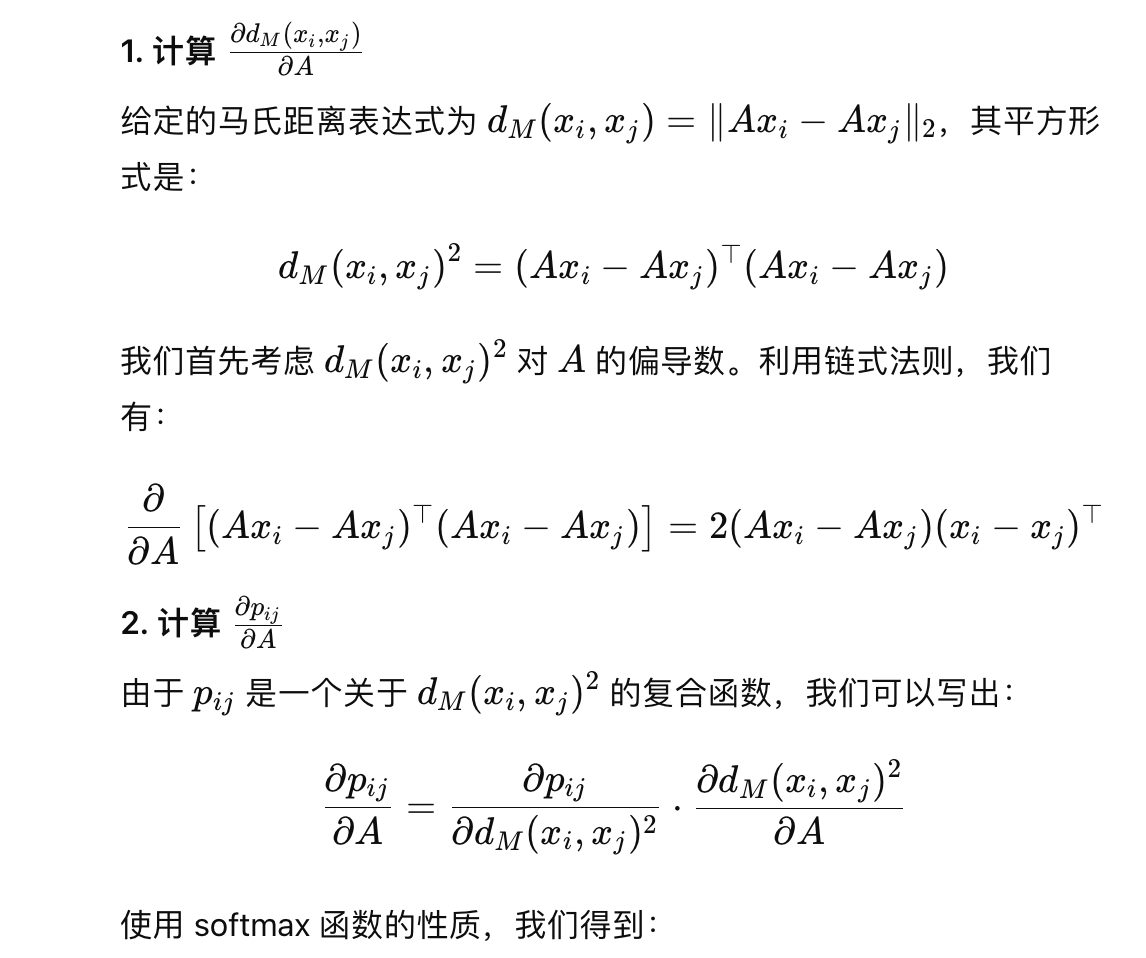
**第二部分**

1. **马氏距离计算**

马氏距离（Mahalanobis Distance）是一种有效的距离度量，用于考虑数据集中各特征之间的相关性。这种距离度量可以看作是对欧式距离的一种推广，特别适用于特征间存在相关性的情况。马氏距离的计算考虑了数据的协方差结构，对于不同的数据尺度具有自然的“尺度不变性”。



1. **公式推导**

****

**第三部分**

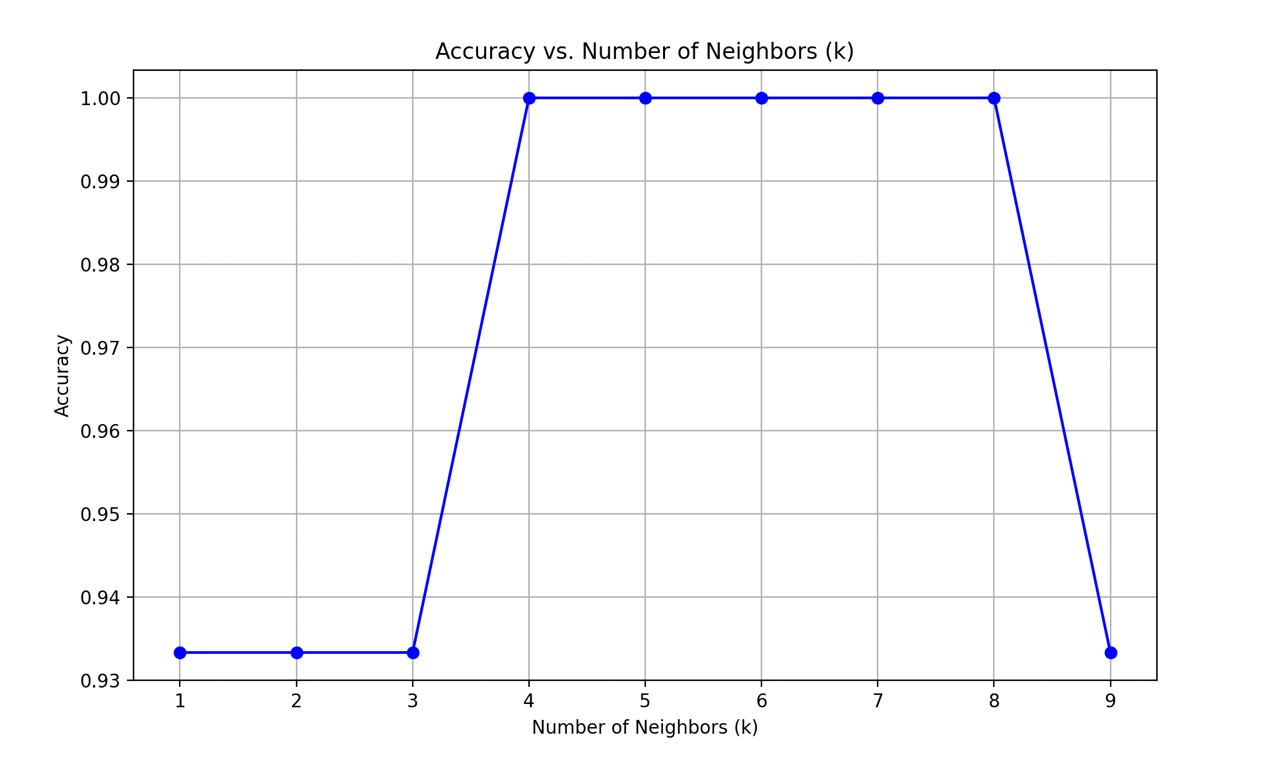
依旧是在Iris数据集上，将自己实现的两种KNN算法（欧氏距离和马氏距离）与sklearn中提供的KNN算法进行对比。

**第四部分**

我们采用sklearn库内置的一些用于KNN的数据集进行测试，在本次实验中，我们采用了Wine数据集和Breast Cancer Wisconsin数据集进行测试。

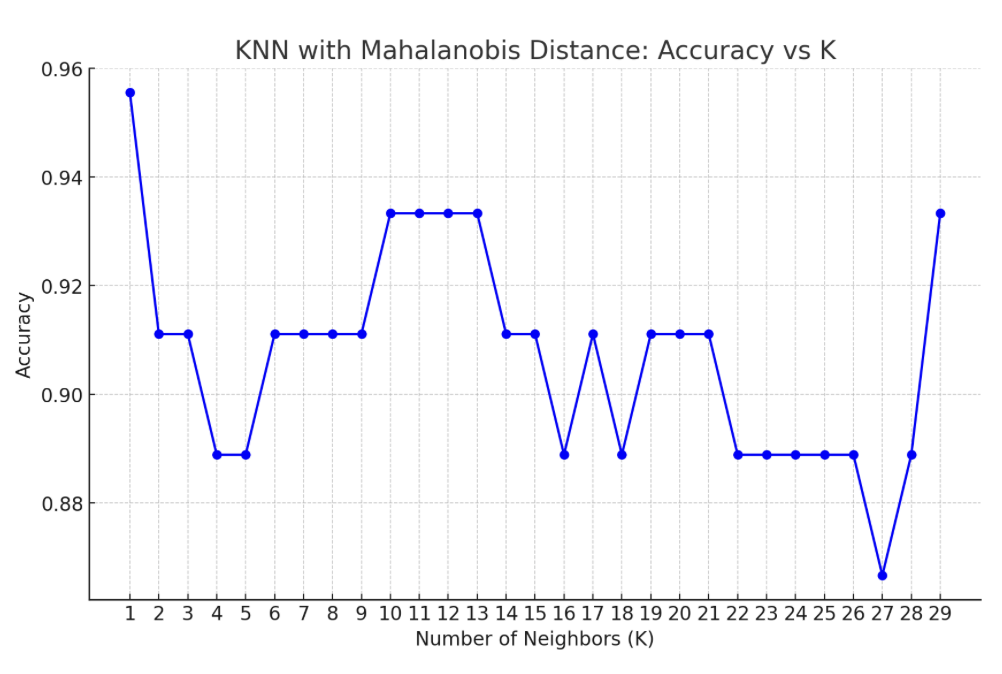
**3、实验结果与分析**

**第一部分**

****

根据结果可知，最为理想的k值为6或者7。在这个k值上，验证集上的正确率达到了100%。

**第二部分**

****

当采用马氏距离进行KNN计算时，由于马氏距离考虑了数据的整体统计特性（如协方差），这可能使得距离度量在不同区域的数据点间表现出较大的差异性。因此我们发现准确率随k取值的波动明显大于欧式距离，光从图形上来看，k=1的时候似乎具有最好的预测能力，但是当k=1时，模型很可能出现了过拟合，这会导致模型在新的数据集上的泛化能力下降，因此综合考虑各种因素，我们选取k=3为最佳k值。

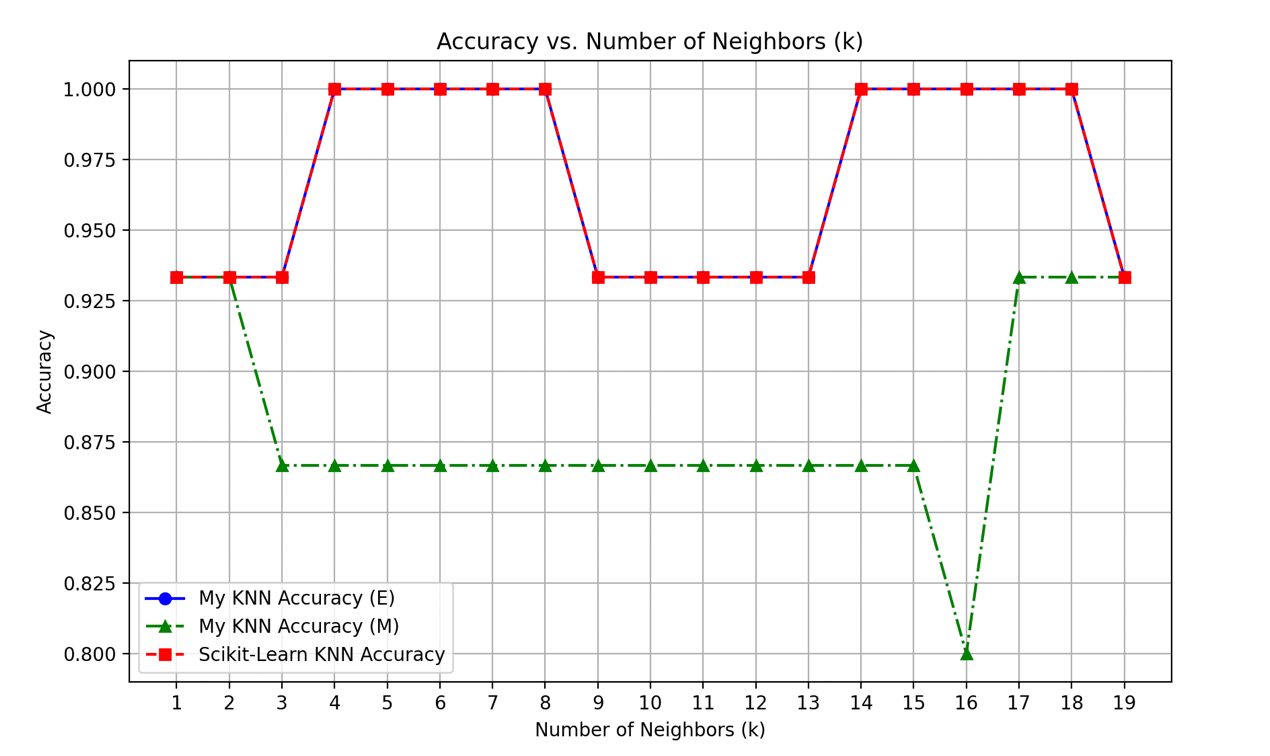
同时，要解决马氏距离计算的波动问题，我们可以尝试增加样本大小或使用更鲁棒的统计方法来估计协方差矩阵，以提高马氏距离的稳定性和准确性，或者使用交叉验证等技术来评估不同k值的性能，以确定最佳的k值设置。

**第三部分**

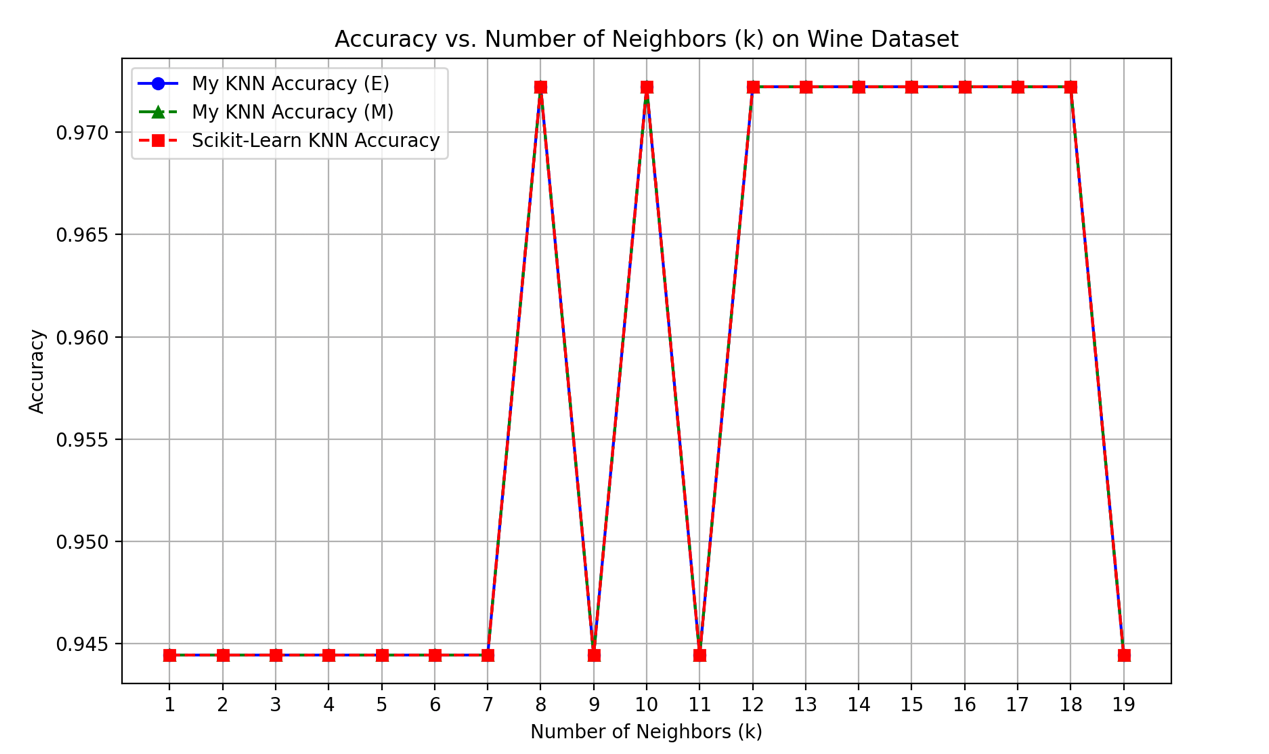
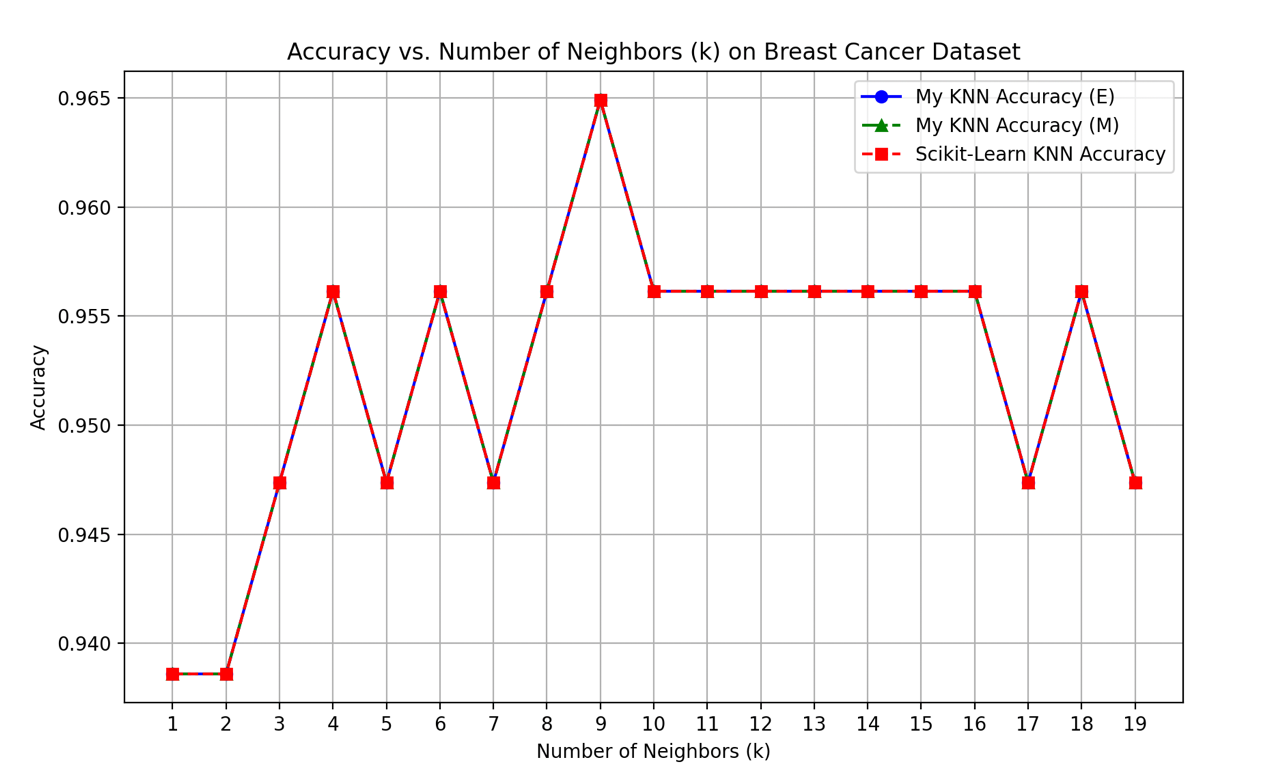
在下图中，在Iris数据集上，对三种算法进行了一个比较，可以观察到，使用欧式距离计算的KNN和sklearn库的KNN算法取得了一致的结果，这意味着我们基于欧式距离的KNN算法是非常有效的。

而采用马氏距离的KNN算法的效果则要差一点，这是马氏距离因为考虑了数据的协方差结构，适用于特征间存在相关性的数据。然而，如果协方差矩阵估计不准确，或者数据特征并不强烈相关，使用马氏距离可能不会带来预期的性能提升，反而因为误导性的特征权重而降低准确率。

某些数据集可能本身不适合使用马氏距离。例如，如果所有特征几乎独立，或者特征间的线性关系不明显，则马氏距离的优势无法发挥，甚至可能因为错误的假设（即特征间存在相关性）而导致性能下降。

****

**第四部分**

****

我们尝试将模型应用在别的模型上，从图中的结果来看，所有的结果都是和sklearn库中的KNN算法相吻合的，这意味着我们的KNN算法是非常有效的。

其中，在Wine数据集上，最佳的k取值为8，在Breast Cancer Wisconsin数据集上，最佳k取值为9。自己实现的两种KNN算法都取得了非常好的效果。

**4、代码附录**

**train\_Euclidean.py**

import pandas as pd  
import numpy as np  
from sklearn.metrics import accuracy\_score  
from sklearn.preprocessing import StandardScaler  
import matplotlib.pyplot as plt  
  
  
class KNeighborsClassifier:  
 def \_\_init\_\_(self, n\_neighbors=3):  
 self.n\_neighbors = n\_neighbors  
 self.X\_train = None  
 self.y\_train = None  
  
 def fit(self, X, y):  
 *"""存储训练数据"""* self.X\_train = X  
 self.y\_train = y  
  
 def predict(self, X):  
 *"""对提供的数据进行预测"""* y\_pred = [self.\_predict(x) for x in X]  
 return np.array(y\_pred)  
  
 def \_predict(self, x):  
 *"""对单个数据点进行预测"""* # 计算距离  
 distances = np.sqrt(np.sum((self.X\_train - x) \*\* 2, axis=1))  
 # 获取最近的 k 个点的索引  
 k\_indices = np.argsort(distances)[:self.n\_neighbors]  
 # 获取这些点的类别  
 k\_nearest\_labels = self.y\_train[k\_indices]  
 # 使用多数投票法确定预测类别，这里我们不使用 Counter，改用 np.bincount  
 label = np.argmax(np.bincount(k\_nearest\_labels))  
 return label  
  
  
# 加载数据  
train\_data = pd.read\_csv('./data/train.csv')  
val\_data = pd.read\_csv('./data/val.csv')  
test\_data = pd.read\_csv('./data/test\_data.csv')  
  
# 准备数据  
X\_train = train\_data.iloc[:, :-1].values  
y\_train = train\_data.iloc[:, -1].values  
X\_val = val\_data.iloc[:, :-1].values  
y\_val = val\_data.iloc[:, -1].values  
X\_test = test\_data.iloc[:, :].values # 假设测试数据没有标签  
  
# 数据标准化  
scaler = StandardScaler()  
X\_train = scaler.fit\_transform(X\_train)  
X\_val = scaler.transform(X\_val)  
X\_test = scaler.transform(X\_test)  
  
accuracies = []  
  
for k\_value in range(1,10):  
  
 # 创建 KNN 模型  
 knn = KNeighborsClassifier(n\_neighbors=k\_value)  
  
 # 训练模型  
 knn.fit(X\_train, y\_train)  
  
 # 在验证集上评估模型  
 y\_pred\_val = knn.predict(X\_val)  
 accuracies.append(accuracy\_score(y\_val, y\_pred\_val))  
  
  
def plot\_accuracy\_vs\_k(k\_values, accuracies):  
 *"""  
 绘制不同k值对应的准确率。  
  
 参数:  
 k\_values : list 或 numpy array  
 KNN 模型中尝试的 k 值列表。  
 accuracies : list 或 numpy array  
 每个 k 值对应的准确率。  
 """* plt.figure(figsize=(10, 6)) # 设置图像大小  
 plt.plot(k\_values, accuracies, marker='o', linestyle='-', color='b') # 绘制准确率曲线  
 plt.title('Accuracy vs. Number of Neighbors (k)') # 图像标题  
 plt.xlabel('Number of Neighbors (k)') # x轴标签  
 plt.ylabel('Accuracy') # y轴标签  
 plt.xticks(k\_values) # 设置x轴的刻度为给定的 k 值  
 plt.grid(True) # 显示网格  
 plt.show() # 显示图像  
  
  
k\_values = np.arange(1, 10) # k 从 1 到 47  
plot\_accuracy\_vs\_k(k\_values, accuracies)

**train\_Mahalanobis.py**

import pandas as pd  
import numpy as np  
from sklearn.metrics import accuracy\_score  
from sklearn.preprocessing import StandardScaler  
import matplotlib.pyplot as plt  
  
  
class KNeighborsClassifier:  
 def \_\_init\_\_(self, n\_neighbors=3):  
 self.n\_neighbors = n\_neighbors  
 self.X\_train = None  
 self.y\_train = None  
 self.inv\_cov\_matrix = None # 马氏距离所需的协方差矩阵的逆  
  
 def fit(self, X, y):  
 *"""存储训练数据并计算协方差矩阵的逆"""* self.X\_train = X  
 self.y\_train = y  
 # 计算协方差矩阵及其逆  
 cov\_matrix = np.cov(X, rowvar=False)  
 self.inv\_cov\_matrix = np.linalg.inv(cov\_matrix)  
  
 def predict(self, X):  
 *"""对提供的数据进行预测"""* y\_pred = [self.\_predict(x) for x in X]  
 return np.array(y\_pred)  
  
 def \_predict(self, x):  
 *"""对单个数据点使用马氏距离进行预测"""* # 计算马氏距离  
 distances = [np.sqrt((x - x\_train).T @ self.inv\_cov\_matrix @ (x - x\_train)) for x\_train in self.X\_train]  
 # 获取最近的 k 个点的索引  
 k\_indices = np.argsort(distances)[:self.n\_neighbors]  
 # 获取这些点的类别  
 k\_nearest\_labels = self.y\_train[k\_indices]  
 # 使用多数投票法确定预测类别  
 label = np.argmax(np.bincount(k\_nearest\_labels))  
 return label  
  
  
# 加载数据  
train\_data = pd.read\_csv('./data/train.csv')  
val\_data = pd.read\_csv('./data/val.csv')  
test\_data = pd.read\_csv('./data/test\_data.csv')  
  
# 准备数据  
X\_train = train\_data.iloc[:, :-1].values  
y\_train = train\_data.iloc[:, -1].values  
X\_val = val\_data.iloc[:, :-1].values  
y\_val = val\_data.iloc[:, -1].values  
X\_test = test\_data.iloc[:, :].values # 假设测试数据没有标签  
  
# 数据标准化  
scaler = StandardScaler()  
X\_train = scaler.fit\_transform(X\_train)  
X\_val = scaler.transform(X\_val)  
X\_test = scaler.transform(X\_test)  
  
accuracies=[]  
  
for k\_value in range(1,20):  
  
 # 创建 KNN 模型  
 knn = KNeighborsClassifier(n\_neighbors=k\_value)  
  
 # 训练模型  
 knn.fit(X\_train, y\_train)  
  
 # 在验证集上评估模型  
 y\_pred\_val = knn.predict(X\_val)  
 accuracies.append(accuracy\_score(y\_val, y\_pred\_val))  
  
  
def plot\_accuracy\_vs\_k(k\_values, accuracies):  
 *"""  
 绘制不同k值对应的准确率。  
  
 参数:  
 k\_values : list 或 numpy array  
 KNN 模型中尝试的 k 值列表。  
 accuracies : list 或 numpy array  
 每个 k 值对应的准确率。  
 """* plt.figure(figsize=(20, 6)) # 设置图像大小  
 plt.plot(k\_values, accuracies, marker='o', linestyle='-', color='b') # 绘制准确率曲线  
 plt.title('Accuracy vs. Number of Neighbors (k)') # 图像标题  
 plt.xlabel('Number of Neighbors (k)') # x轴标签  
 plt.ylabel('Accuracy') # y轴标签  
 plt.xticks(k\_values) # 设置x轴的刻度为给定的 k 值  
 plt.grid(True) # 显示网格  
 plt.show() # 显示图像  
  
  
k\_values = np.arange(1, 20) # k 从 1 到 47  
plot\_accuracy\_vs\_k(k\_values, accuracies)

**train\_sklearn.py**

import pandas as pd  
import numpy as np  
from sklearn.metrics import accuracy\_score  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.preprocessing import StandardScaler  
import matplotlib.pyplot as plt  
  
  
class E\_KNeighborsClassifier:  
 def \_\_init\_\_(self, n\_neighbors=3):  
 self.n\_neighbors = n\_neighbors  
 self.X\_train = None  
 self.y\_train = None  
  
 def fit(self, X, y):  
 *"""存储训练数据"""* self.X\_train = X  
 self.y\_train = y  
  
 def predict(self, X):  
 *"""对提供的数据进行预测"""* y\_pred = [self.\_predict(x) for x in X]  
 return np.array(y\_pred)  
  
 def \_predict(self, x):  
 *"""对单个数据点进行预测"""* # 计算距离  
 distances = np.sqrt(np.sum((self.X\_train - x) \*\* 2, axis=1))  
 # 获取最近的 k 个点的索引  
 k\_indices = np.argsort(distances)[:self.n\_neighbors]  
 # 获取这些点的类别  
 k\_nearest\_labels = self.y\_train[k\_indices]  
 # 使用多数投票法确定预测类别，这里我们不使用 Counter，改用 np.bincount  
 label = np.argmax(np.bincount(k\_nearest\_labels))  
 return label  
  
  
class M\_KNeighborsClassifier:  
 def \_\_init\_\_(self, n\_neighbors=3):  
 self.n\_neighbors = n\_neighbors  
 self.X\_train = None  
 self.y\_train = None  
 self.inv\_cov\_matrix = None # 马氏距离所需的协方差矩阵的逆  
  
 def fit(self, X, y):  
 *"""存储训练数据并计算协方差矩阵的逆"""* self.X\_train = X  
 self.y\_train = y  
 # 计算协方差矩阵及其逆  
 cov\_matrix = np.cov(X, rowvar=False)  
 self.inv\_cov\_matrix = np.linalg.inv(cov\_matrix)  
  
 def predict(self, X):  
 *"""对提供的数据进行预测"""* y\_pred = [self.\_predict(x) for x in X]  
 return np.array(y\_pred)  
  
 def \_predict(self, x):  
 *"""对单个数据点使用马氏距离进行预测"""* # 计算马氏距离  
 distances = [np.sqrt((x - x\_train).T @ self.inv\_cov\_matrix @ (x - x\_train)) for x\_train in self.X\_train]  
 # 获取最近的 k 个点的索引  
 k\_indices = np.argsort(distances)[:self.n\_neighbors]  
 # 获取这些点的类别  
 k\_nearest\_labels = self.y\_train[k\_indices]  
 # 使用多数投票法确定预测类别  
 label = np.argmax(np.bincount(k\_nearest\_labels))  
 return label  
  
  
# 加载数据  
train\_data = pd.read\_csv('./data/train.csv')  
val\_data = pd.read\_csv('./data/val.csv')  
test\_data = pd.read\_csv('./data/test\_data.csv')  
  
# 准备数据  
X\_train = train\_data.iloc[:, :-1].values  
y\_train = train\_data.iloc[:, -1].values  
X\_val = val\_data.iloc[:, :-1].values  
y\_val = val\_data.iloc[:, -1].values  
X\_test = test\_data.iloc[:, :].values # 假设测试数据没有标签  
  
# 数据标准化  
scaler = StandardScaler()  
X\_train = scaler.fit\_transform(X\_train)  
X\_val = scaler.transform(X\_val)  
X\_test = scaler.transform(X\_test)  
  
my\_E\_accuracies = []  
my\_M\_accuracies = []  
sk\_accuracies = []  
  
for k\_value in range(1,20):  
  
 # 创建 KNN 模型  
 my\_E\_knn = E\_KNeighborsClassifier(n\_neighbors=k\_value)  
 my\_M\_knn = M\_KNeighborsClassifier(n\_neighbors=k\_value)  
 sk\_knn = KNeighborsClassifier(n\_neighbors=k\_value)  
 # 训练模型  
 my\_E\_knn.fit(X\_train, y\_train)  
 my\_M\_knn.fit(X\_train, y\_train)  
 sk\_knn.fit(X\_train, y\_train)  
 # 在验证集上评估模型  
 y\_pred\_val\_E = my\_E\_knn.predict(X\_val)  
 y\_pred\_val\_M = my\_M\_knn.predict(X\_val)  
 y\_pred\_val\_sk = sk\_knn.predict(X\_val)  
 my\_E\_accuracies.append(accuracy\_score(y\_val, y\_pred\_val\_E))  
 my\_M\_accuracies.append(accuracy\_score(y\_val, y\_pred\_val\_M))  
 sk\_accuracies.append(accuracy\_score(y\_val, y\_pred\_val\_sk))  
  
  
def plot\_accuracy\_vs\_k(k\_values, my\_E\_accuracies, my\_M\_accuracies, sk\_accuracies):  
 *"""  
 绘制不同k值对应的三组准确率。  
  
 参数:  
 k\_values : list 或 numpy array  
 KNN 模型中尝试的 k 值列表。  
 my\_E\_accuracies : list 或 numpy array  
 使用自定义算法 E 实现的 KNN 模型的每个 k 值对应的准确率。  
 my\_M\_accuracies : list 或 numpy array  
 使用自定义算法 M 实现的 KNN 模型的每个 k 值对应的准确率。  
 sk\_accuracies : list 或 numpy array  
 使用scikit-learn实现的 KNN 模型的每个 k 值对应的准确率。  
 """* plt.figure(figsize=(10, 6)) # 设置图像大小  
 plt.plot(k\_values, my\_E\_accuracies, marker='o', linestyle='-', color='b', label='My KNN Accuracy (E)') # 绘制自定义KNN (E) 的准确率曲线  
 plt.plot(k\_values, my\_M\_accuracies, marker='^', linestyle='-.', color='g', label='My KNN Accuracy (M)') # 绘制自定义KNN (M) 的准确率曲线  
 plt.plot(k\_values, sk\_accuracies, marker='s', linestyle='--', color='r', label='Scikit-Learn KNN Accuracy') # 绘制scikit-learn KNN的准确率曲线  
 plt.title('Accuracy vs. Number of Neighbors (k)') # 图像标题  
 plt.xlabel('Number of Neighbors (k)') # x轴标签  
 plt.ylabel('Accuracy') # y轴标签  
 plt.xticks(k\_values) # 设置x轴的刻度为给定的 k 值  
 plt.grid(True) # 显示网格  
 plt.legend() # 显示图例  
 plt.show() # 显示图像  
  
  
  
k\_values = np.arange(1, 20) # k 从 1 到 20  
plot\_accuracy\_vs\_k(k\_values, my\_E\_accuracies, my\_M\_accuracies, sk\_accuracies)

**train\_breast\_cancer.py**

import numpy as np  
import pandas as pd  
from sklearn.datasets import load\_breast\_cancer  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import accuracy\_score  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.preprocessing import StandardScaler  
import matplotlib.pyplot as plt  
  
class E\_KNeighborsClassifier:  
 def \_\_init\_\_(self, n\_neighbors=3):  
 self.n\_neighbors = n\_neighbors  
 self.X\_train = None  
 self.y\_train = None  
  
 def fit(self, X, y):  
 self.X\_train = X  
 self.y\_train = y  
  
 def predict(self, X):  
 y\_pred = [self.\_predict(x) for x in X]  
 return np.array(y\_pred)  
  
 def \_predict(self, x):  
 distances = np.sqrt(np.sum((self.X\_train - x) \*\* 2, axis=1))  
 k\_indices = np.argsort(distances)[:self.n\_neighbors]  
 k\_nearest\_labels = self.y\_train[k\_indices]  
 return np.argmax(np.bincount(k\_nearest\_labels))  
  
class M\_KNeighborsClassifier:  
 def \_\_init\_\_(self, n\_neighbors=3):  
 self.n\_neighbors = n\_neighbors  
 self.X\_train = None  
 self.y\_train = None  
  
 def fit(self, X, y):  
 self.X\_train = X  
 self.y\_train = y  
  
 def predict(self, X):  
 y\_pred = [self.\_predict(x) for x in X]  
 return np.array(y\_pred)  
  
 def \_predict(self, x):  
 distances = np.sqrt(np.sum((self.X\_train - x) \*\* 2, axis=1))  
 k\_indices = np.argsort(distances)[:self.n\_neighbors]  
 k\_nearest\_labels = self.y\_train[k\_indices]  
 return np.argmax(np.bincount(k\_nearest\_labels))  
  
# 加载乳腺癌数据集  
data = load\_breast\_cancer()  
X, y = data.data, data.target  
  
# 划分数据集  
X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
  
# 数据标准化  
scaler = StandardScaler()  
X\_train = scaler.fit\_transform(X\_train)  
X\_val = scaler.transform(X\_val)  
  
my\_E\_accuracies = []  
my\_M\_accuracies = []  
sk\_accuracies = []  
  
for k\_value in range(1, 20):  
 my\_E\_knn = E\_KNeighborsClassifier(n\_neighbors=k\_value)  
 my\_M\_knn = M\_KNeighborsClassifier(n\_neighbors=k\_value)  
 sk\_knn = KNeighborsClassifier(n\_neighbors=k\_value)  
  
 my\_E\_knn.fit(X\_train, y\_train)  
 my\_M\_knn.fit(X\_train, y\_train)  
 sk\_knn.fit(X\_train, y\_train)  
  
 y\_pred\_val\_E = my\_E\_knn.predict(X\_val)  
 y\_pred\_val\_M = my\_M\_knn.predict(X\_val)  
 y\_pred\_val\_sk = sk\_knn.predict(X\_val)  
  
 my\_E\_accuracies.append(accuracy\_score(y\_val, y\_pred\_val\_E))  
 my\_M\_accuracies.append(accuracy\_score(y\_val, y\_pred\_val\_M))  
 sk\_accuracies.append(accuracy\_score(y\_val, y\_pred\_val\_sk))  
  
def plot\_accuracy\_vs\_k(k\_values, my\_E\_accuracies, my\_M\_accuracies, sk\_accuracies):  
 plt.figure(figsize=(10, 6))  
 plt.plot(k\_values, my\_E\_accuracies, marker='o', linestyle='-', color='b', label='My KNN Accuracy (E)')  
 plt.plot(k\_values, my\_M\_accuracies, marker='^', linestyle='-.', color='g', label='My KNN Accuracy (M)')  
 plt.plot(k\_values, sk\_accuracies, marker='s', linestyle='--', color='r', label='Scikit-Learn KNN Accuracy')  
 plt.title('Accuracy vs. Number of Neighbors (k) on Breast Cancer Dataset')  
 plt.xlabel('Number of Neighbors (k)')  
 plt.ylabel('Accuracy')  
 plt.xticks(k\_values)  
 plt.grid(True)  
 plt.legend()  
 plt.show()  
  
  
k\_values = np.arange(1, 20)  
plot\_accuracy\_vs\_k(k\_values, my\_E\_accuracies, my\_M\_accuracies, sk\_accuracies)

**train\_wine.py**

import numpy as np  
import pandas as pd  
from sklearn.datasets import load\_wine  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import accuracy\_score  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.preprocessing import StandardScaler  
import matplotlib.pyplot as plt  
  
class E\_KNeighborsClassifier:  
 def \_\_init\_\_(self, n\_neighbors=3):  
 self.n\_neighbors = n\_neighbors  
 self.X\_train = None  
 self.y\_train = None  
  
 def fit(self, X, y):  
 self.X\_train = X  
 self.y\_train = y  
  
 def predict(self, X):  
 y\_pred = [self.\_predict(x) for x in X]  
 return np.array(y\_pred)  
  
 def \_predict(self, x):  
 distances = np.sqrt(np.sum((self.X\_train - x) \*\* 2, axis=1))  
 k\_indices = np.argsort(distances)[:self.n\_neighbors]  
 k\_nearest\_labels = self.y\_train[k\_indices]  
 return np.argmax(np.bincount(k\_nearest\_labels))  
  
class M\_KNeighborsClassifier:  
 def \_\_init\_\_(self, n\_neighbors=3):  
 self.n\_neighbors = n\_neighbors  
 self.X\_train = None  
 self.y\_train = None  
  
 def fit(self, X, y):  
 self.X\_train = X  
 self.y\_train = y  
  
 def predict(self, X):  
 y\_pred = [self.\_predict(x) for x in X]  
 return np.array(y\_pred)  
  
 def \_predict(self, x):  
 distances = np.sqrt(np.sum((self.X\_train - x) \*\* 2, axis=1))  
 k\_indices = np.argsort(distances)[:self.n\_neighbors]  
 k\_nearest\_labels = self.y\_train[k\_indices]  
 return np.argmax(np.bincount(k\_nearest\_labels))  
  
  
# 加载葡萄酒数据集  
data = load\_wine()  
X, y = data.data, data.target  
  
# 划分数据集  
X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
  
# 数据标准化  
scaler = StandardScaler()  
X\_train = scaler.fit\_transform(X\_train)  
X\_val = scaler.transform(X\_val)  
  
my\_E\_accuracies = []  
my\_M\_accuracies = []  
sk\_accuracies = []  
  
for k\_value in range(1, 20):  
 my\_E\_knn = E\_KNeighborsClassifier(n\_neighbors=k\_value)  
 my\_M\_knn = M\_KNeighborsClassifier(n\_neighbors=k\_value)  
 sk\_knn = KNeighborsClassifier(n\_neighbors=k\_value)  
  
 my\_E\_knn.fit(X\_train, y\_train)  
 my\_M\_knn.fit(X\_train, y\_train)  
 sk\_knn.fit(X\_train, y\_train)  
  
 y\_pred\_val\_E = my\_E\_knn.predict(X\_val)  
 y\_pred\_val\_M = my\_M\_knn.predict(X\_val)  
 y\_pred\_val\_sk = sk\_knn.predict(X\_val)  
  
 my\_E\_accuracies.append(accuracy\_score(y\_val, y\_pred\_val\_E))  
 my\_M\_accuracies.append(accuracy\_score(y\_val, y\_pred\_val\_M))  
 sk\_accuracies.append(accuracy\_score(y\_val, y\_pred\_val\_sk))  
  
def plot\_accuracy\_vs\_k(k\_values, my\_E\_accuracies, my\_M\_accuracies, sk\_accuracies):  
 plt.figure(figsize=(10, 6))  
 plt.plot(k\_values, my\_E\_accuracies, marker='o', linestyle='-', color='b', label='My KNN Accuracy (E)')  
 plt.plot(k\_values, my\_M\_accuracies, marker='^', linestyle='-.', color='g', label='My KNN Accuracy (M)')  
 plt.plot(k\_values, sk\_accuracies, marker='s', linestyle='--', color='r', label='Scikit-Learn KNN Accuracy')  
 plt.title('Accuracy vs. Number of Neighbors (k) on Wine Dataset')  
 plt.xlabel('Number of Neighbors (k)')  
 plt.ylabel('Accuracy')  
 plt.xticks(k\_values)  
 plt.grid(True)  
 plt.legend()  
 plt.show()  
  
  
k\_values = np.arange(1, 20)  
plot\_accuracy\_vs\_k(k\_values, my\_E\_accuracies, my\_M\_accuracies, sk\_accuracies)

**实验三 神经网络**

1. **问题描述**

**概述**

利用神经网络算法，对CIFAR-10 数据集中给定的测试集进行分类。

**数据说明**

CIFAR-10是一个描述自然图像的数据集，一共包含10个类别的彩色图片：飞机（airplane）、汽车（automobile）、鸟类（bird）、猫（cat）、鹿（deer）、狗（dog）、蛙类（frog）、马（horse）、船（ship）和卡车（truck）。每个图片的尺寸为 32 ×32 ，每个类别有 6000 个图像，数据集中一共有50000张训练图片和10000张测试图片 。

**实验内容**

一、基于神经网络模型及BP 算法，根据训练集中的数据对所设计的神经网络模型进行训练，随后对给定的打乱的测试集中的数据进行分类。

二、基于MindSpore平台实现算法，对相同的数据集进行训练，并与不使用MindSpore平台实现的算法对比结果（包括但不限于准确率、算法迭代收敛次数等指标），并分析结果中出现差异的可能原因，给出使用 MindSpore 的心得和建议。

三、（加分项）使用 MindSpore、sklearn 等平台提供的相似任务数据集（例如，其他的分类任务数据集）测试自己独立实现的算法，并与 MindSpore、sklearn 等平台上的官方实现算法进行对比，进一步分析差异及其成因。

1. **实现步骤与流程**

**第一部分**

一、数据预处理

Cifar10数据集的数据是彩色图片，因此具有RGB三通道，每个通道的数值范围为0～255，我们需要对数值进行标准化，因此对原始数据除以255将图片的所有像素值压缩到0～1之间。此外，Cifar10的训练集还分成了5个batch，因此需要定义函数去将5个训练集合并到一个数据集中去。

二、神经网络的构建

在构建神经网络的过程中，一开始尝试使用单隐层神经网络去完成Cifar10数据集的分类，但是在尝试之后发现，模型预测的准确率始终不理想，考虑到Cifar10数据集的复杂性，我们采用了更复杂的神经网络的结构，使用了三个隐藏层。

在程序中MultiLayerNN类定义了一个具有多层全连接层的神经网络。它的结构如下：

**1. 输入层：**

输入数据的尺寸由input\_size指定。这是每个输入数据样本的特征数量。

**2. 第一层（隐藏层）：**

有512个神经元。

权重矩阵w1的维度为input\_size x 512，用于将输入数据从input\_size特征转换到512个特征。

偏置向量b1的维度为512。

使用ReLU激活函数。

**3. 第二层（隐藏层）：**

有256个神经元。

权重矩阵w2的维度为`512 x 256`。

偏置向量b2的维度为256。

使用ReLU激活函数。

**4. 第三层（隐藏层）：**

有128个神经元。

权重矩阵w3的维度为256 x 128。

偏置向量b3的维度为128。

使用ReLU激活函数。

**5. 输出层：**

有10个神经元，通常对应于分类任务中的10个类别。

权重矩阵`w4`的维度为`128 x 10`。

偏置向量`b4`的维度为10。

使用Softmax激活函数，以输出概率分布，每个神经元的输出代表属于对应类别的概率。

三、学习率

学习率是神经网络训练中的一个重要超参数，决定了模型参数（例如权重和偏置）的更新速度。它在梯度下降及其变种算法中起着关键作用，因为它控制了梯度在每次迭代中的步长。学习率对神经网络训练有多方面的影响，包括以下几点：

**影响模型收敛**

**收敛速度**：较高的学习率可以加快训练过程，因为权重的更新步长较大。这可以帮助模型快速接近目标函数的最小值。

**过快收敛或振荡**：如果学习率过高，可能会导致模型在最小值附近来回振荡，难以达到稳定的收敛状态。这可能导致模型性能不稳定。

**过慢收敛**：较低的学习率可能导致模型收敛缓慢，训练时间过长，甚至无法收敛到最佳状态。

**影响模型性能**

**模型精度**：适当的学习率可以使模型达到最佳性能。如果学习率过高，可能会错过目标函数的最小值，从而导致模型性能不佳。

**过拟合或欠拟合**：学习率过低可能导致模型欠拟合，因为更新步长太小，无法充分学习数据特征。学习率过高可能导致模型过拟合，因为权重更新幅度过大，容易受训练数据噪音影响。

**影响模型稳定性**

**梯度爆炸或梯度消失**：较高的学习率可能导致梯度爆炸，因更新步长太大，导致梯度不断增大。较低的学习率可能导致梯度消失，因更新步长过小，梯度不断减小，最终趋于零。

**训练中断**：学习率过高可能导致模型在训练过程中不稳定，甚至出现训练中断的情况。

**学习率调整策略**

**恒定学习率**：在整个训练过程中保持固定的学习率。这种方法简单，但可能需要多次尝试才能找到合适的值。

**学习率衰减**：随着训练的进行，逐渐降低学习率。这样可以帮助模型在训练后期更好地收敛。

**自适应学习率**：如 Adam、RMSprop 等优化算法会根据梯度的变化调整学习率，确保训练过程更加稳定。

学习率的选择是神经网络训练中的关键环节，需要根据具体任务、数据集和网络结构进行调整。正确的学习率可以显著提高训练速度、模型性能和稳定性，而不合适的学习率可能导致模型训练失败或性能不佳。

四、批大小

批大小（batch size）是深度学习训练过程中一个重要的超参数，它指的是每次训练时所使用的样本数量。在训练过程中，数据集会被分割成若干个批次，每个批次包含一定数量的样本，而批大小就是每个批次中包含的样本数量。

批大小的选择会影响训练的速度和模型的泛化能力，通常来说，批大小越大，训练过程中所需的内存和计算资源就越多，但训练速度通常会更快；而批大小越小，训练过程中的噪声会增加，但模型对于训练数据的梯度估计会更准确，有助于模型的泛化能力。

常见的批大小选择通常在 8 到 256 之间，具体的选择取决于数据集的大小、模型的复杂度、计算资源的限制等因素。在实践中，通常通过尝试不同的批大小，并根据模型的性能来选择最优的批大小。

值得注意的是，在训练过程中，每个批次的梯度更新只是对整个训练集的一个估计，而不是对整个数据集的准确梯度。因此，批大小的选择也会影响梯度更新的稳定性和模型的收敛速度。

1. 迭代次数

学习周期（epoch）是深度学习中一个重要的概念，它指的是将整个训练数据集在神经网络上进行一次完整的前向传播和反向传播的过程。在一个 epoch 中，所有的训练样本都被送入网络进行训练，并且根据损失函数计算出梯度，然后使用优化算法（如随机梯度下降）来更新网络的参数。

在每个 epoch 中，训练数据集会被划分为若干个批次（batch），每个批次包含一定数量的样本。神经网络在每个批次上进行一次前向传播和反向传播，然后更新参数。当所有批次都处理完毕后，一个 epoch 完成。

学习周期的选择通常取决于训练数据集的大小和复杂性，以及神经网络的架构和优化算法。通常情况下，一个 epoch 包含的迭代次数越多，模型的训练效果越好，但同时也会增加训练的时间成本。

一般来说，训练模型的过程是通过多个 epoch 来完成的，直到模型收敛到一个满意的状态为止。通常会监控训练过程中的指标（如损失函数的值或者验证集上的性能），并在模型停止改善或者出现过拟合时停止训练。

1. 优化器

在本实验中，我们手动实现了Adam优化器，Adam（Adaptive Moment Estimation）优化器是一种用于深度学习应用中的广泛使用的优化算法，它结合了两种扩展的随机梯度下降算法：动量（Momentum）和RMSprop。Adam优化器通过计算梯度的一阶矩估计（即均值）和二阶矩估计（即未中心化的方差），能够适应每个参数的更新步长，使其更加高效和稳定。

**Adam的主要特点包括：**

**1. 自适应学习率：**

Adam自动调整每个参数的学习率，基于估计的一阶和二阶矩。这意味着参数更新的大小可以根据参数的重要性动态调整。

**2. 动量：**

Adam利用梯度的指数移动平均（EMA），来帮助指导其优化路径，这有助于加速SGD在相关方向上的收敛速度并抑制振荡。

**3. 二阶矩估计：**

通过计算梯度的平方的指数移动平均，Adam调整每个参数的学习率，这有助于处理非平稳目标的优化和非均匀的梯度（即某些参数比其他参数具有更大或更小的更新）。

**Adam的更新规则：**

Adam优化器的参数更新可以通过以下步骤进行：

**1. 计算梯度：**计算当前参数相对于损失函数的梯度。

**2. 更新一阶矩估计（均值）**

**3. 更新二阶矩估计（未中心化的方差）**

**4. 修正一阶矩估计和二阶矩估计**

**5. 参数更新**

这种更新方法使得Adam在实际应用中，尤其是在处理非凸优化问题时，比简单的随机梯度下降方法表现更为优异。

**第二部分**

一、数据预处理

采用mindspore中的Cifar10Dataset函数来加载数据集，使用库中的map函数对数据进行转换，map函数将数据进行标准化和压缩，同时，还需要用到HWC2CHW函数对数据的维度进行重排，原始数据的通道数放在最后面，在转换之后，通道数被放置到最前面。

二、神经网络的构建

引用了Mindspore Neural Network (nn) 模块：用于构建网络层（如Dense, ReLU）和设置损失函数及优化器；Mindspore Operations (ops) 模块：提供底层操作如value\_and\_grad（计算函数值和梯度）和depend（控制操作的依赖关系），这是训练过程中自动微分和参数更新的核心；使用create\_tuple\_iterator方法遍历数据集，这是Mindspore中处理数据集的一种方式，能够有效地批量处理数据。Network类充分利用了Mindspore的功能来构建、训练和评估一个神经网络，这表明了Mindspore在机器学习和深度学习领域的实用性和灵活性。

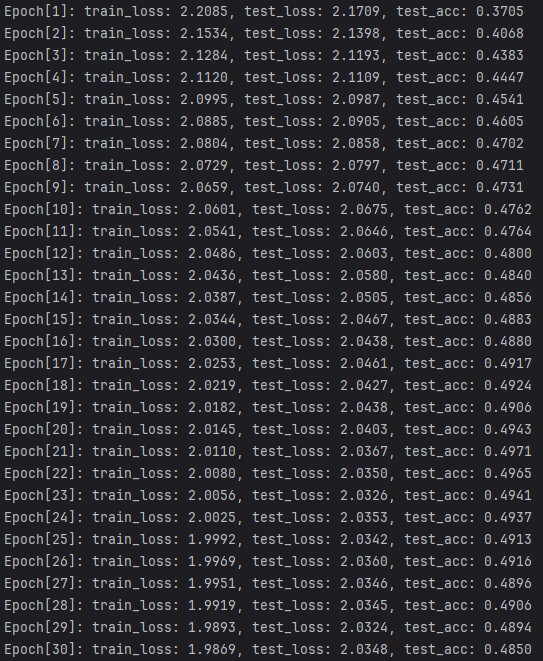
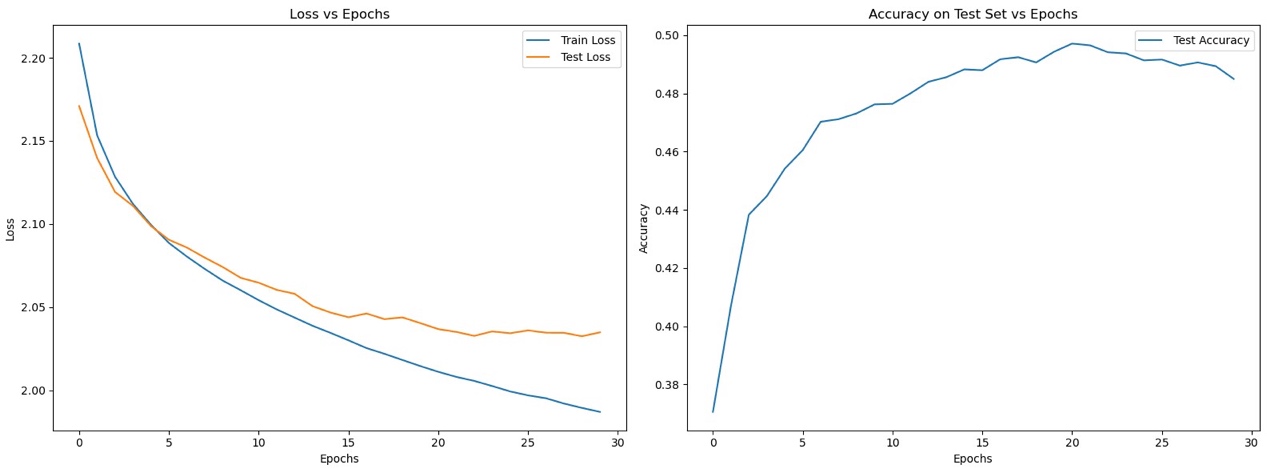
**第三部分**

一、数据预处理

我们在MNIST数据集上去测试mindspore和自己实现的神经网络，在数据预处理方面，需要注意的是MNIST数据集中的图片都是灰度图，不存在RGB三个通道，在调用mindspore的程序中，我们采用MnistDataset函数来加载MNIST数据集。

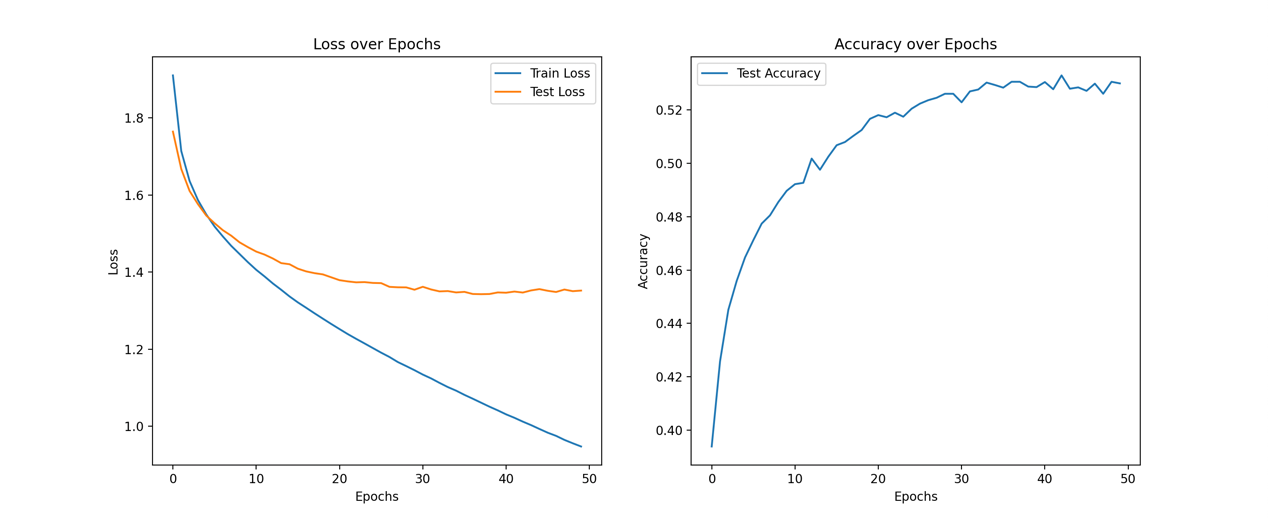
1. **实验结果与分析**

**第一部分**

****

从图中可以看出，由于Cifar10的数据集复杂度比较高，模型预测的准确率最高49.71%，，当训练轮数大于20轮之后，模型出现了较为明显的过拟合现象，test\_loss的曲线上升，模型预测的准确率出现下降，因此，模型在学习轮次20左右的时候已经达到了最佳水平。

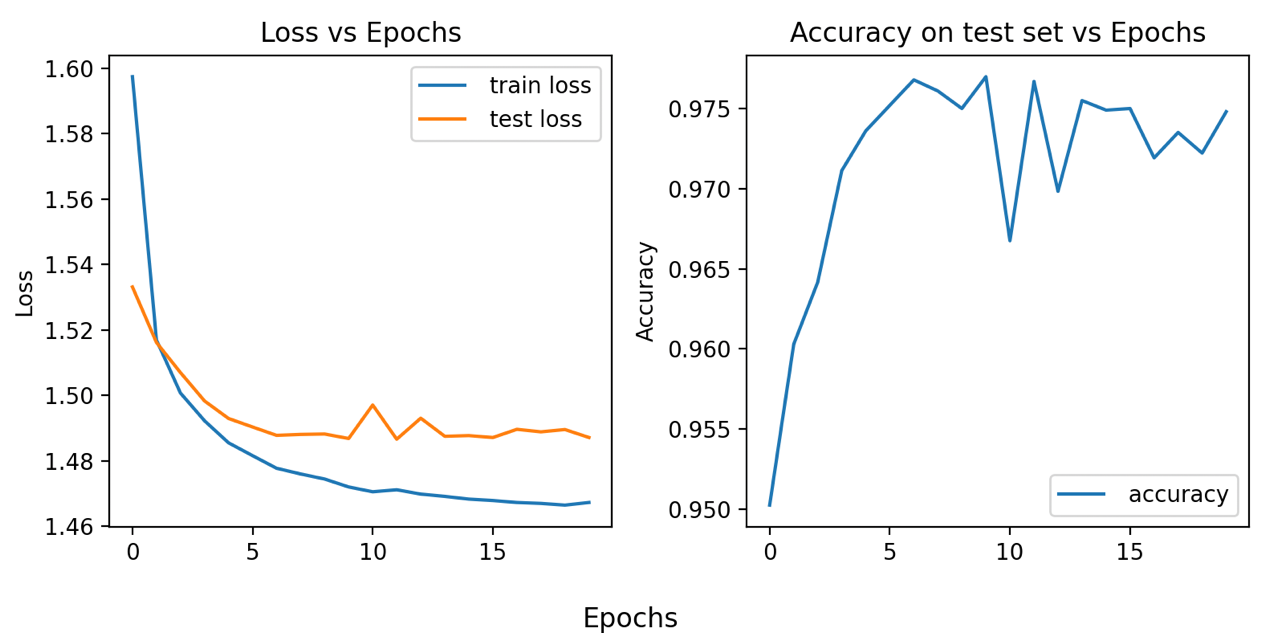
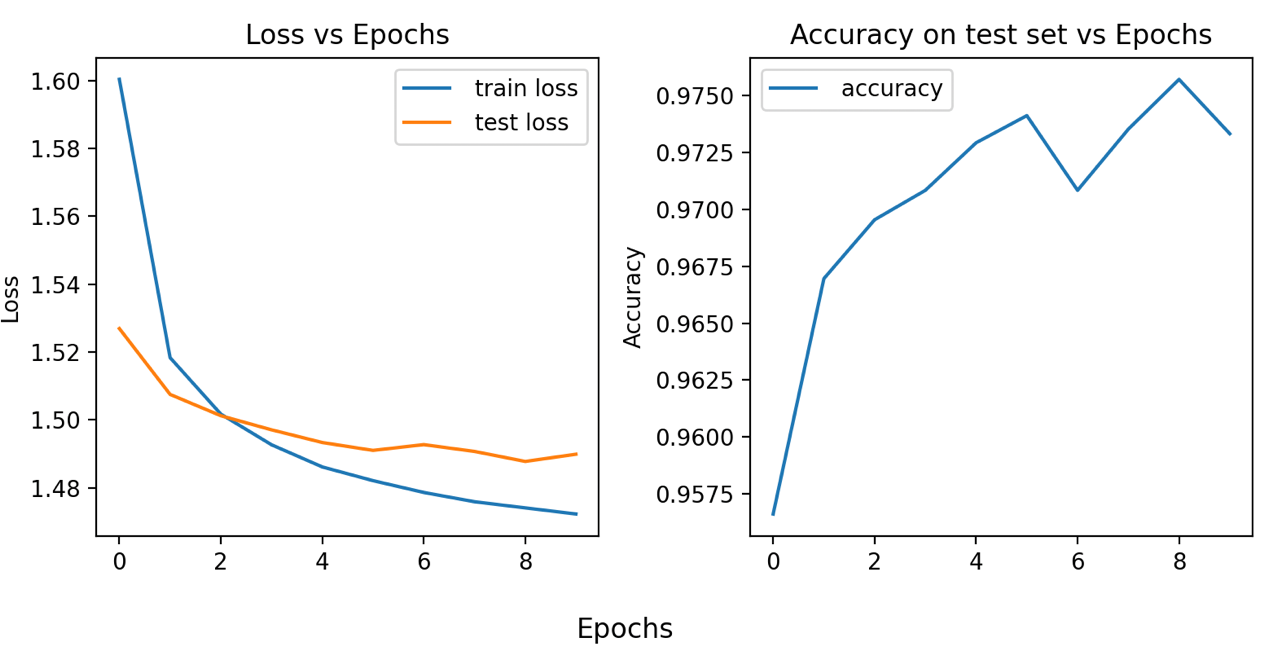
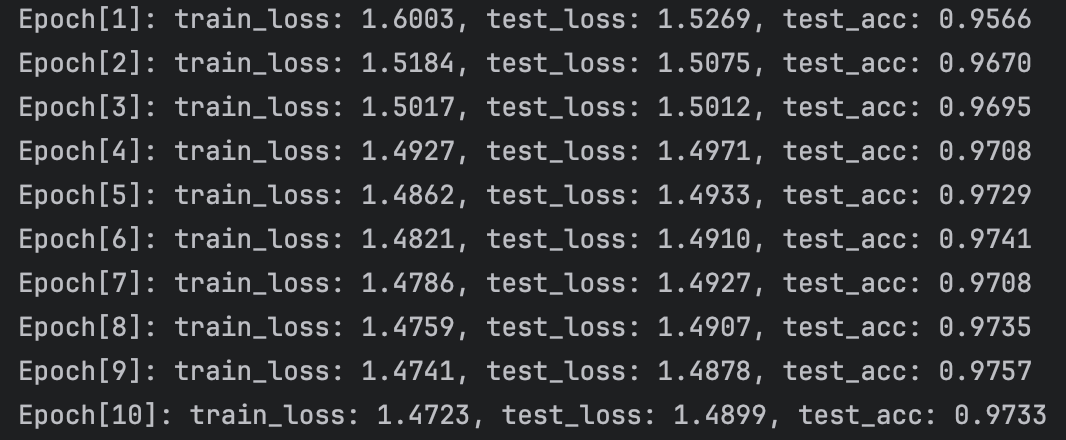
**第二部分**

****

可以看到训练轮次达到40轮的时候，模型出现了过拟合，最高准确率为53.30%。取得了和自己实现的算法相近的预测准确率。

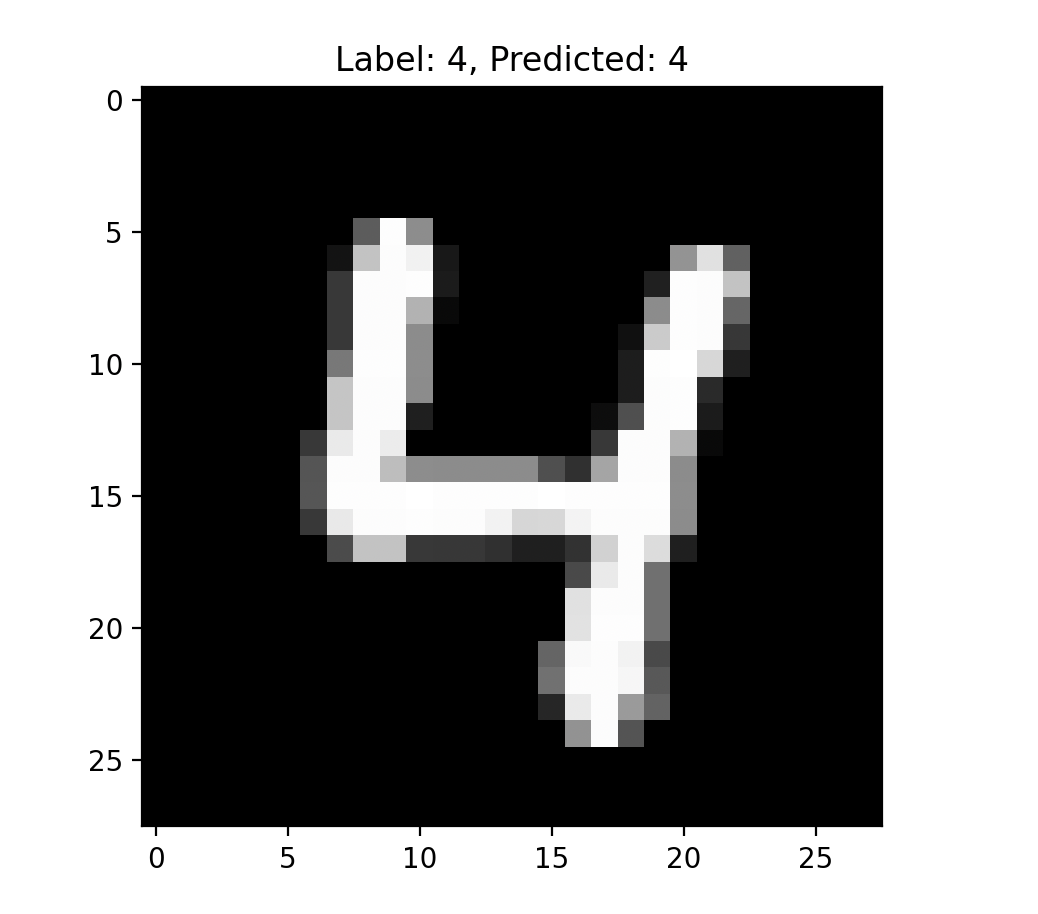
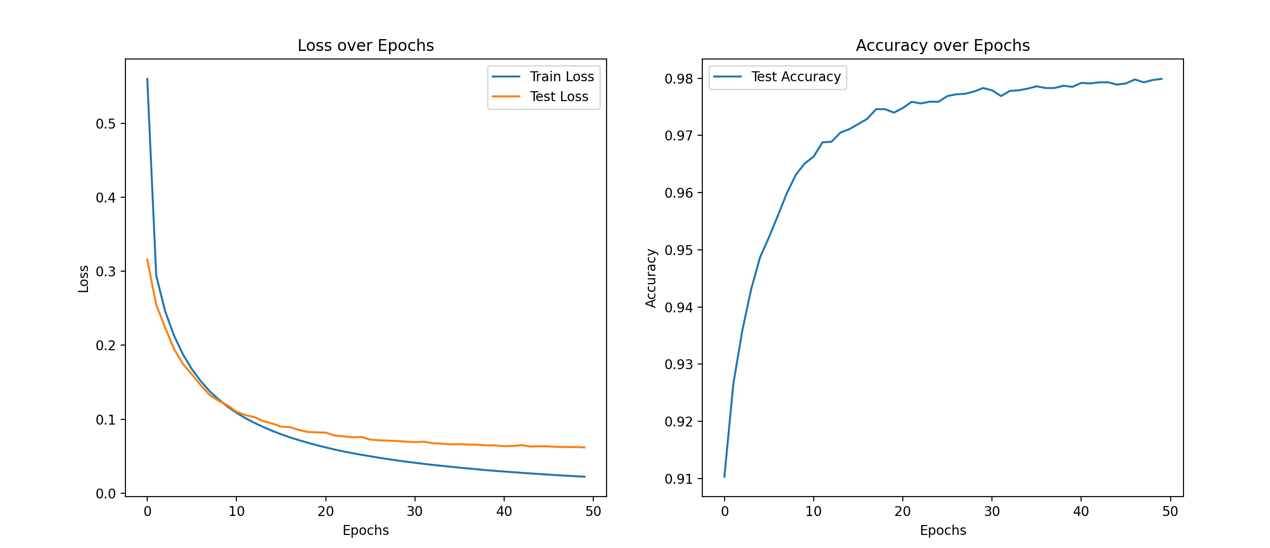
**第三部分**

1、自己实现的算法

****

可以看到在MNIST数据集上，因为数据集复杂度降低的原因，我们自己实现的神经网络在该数据集上非常有效，甚至取得了比在Cifar10数据集上好得多的效果，训练得到的准确率最高达到了97.57%。同时我们可以看到，学习迭代次数在10次的时候就已经能取得较好的学习效果，过多的迭代次数会造成模型的过拟合。

2、mindspore实现的算法



基于mindspore的算法在MNIST数据集上也取得了非常好的效果，最高准确率达到了97.99%。

1. **MindSpore学习使用心得体会**

MindSpore封装了许多人工智能常见的模型，主要用于计算机视觉方面的计算，但是在

KNN和PCA和LDA这种数据处理方面，没有集成现成的算法，在神经网络方面，MindSpore的使用方式与sklearn类似，在计算方面有比较明显的优化，在相同数据集上的计算速度明显比自己实现的算法训练速度快。缺点是生态较差，遇到问题时，在网上难以查询到对症的解决方案。

1. **代码附录**

**train\_man\_cifar.py**

import numpy as np  
import matplotlib.pyplot as plt  
  
  
def cifar\_transform(images):  
 return images.astype(np.float32) / 255.0  
  
  
def plot\_history(\*args, \*\*kwargs):  
 *"""  
 Plot training and testing loss, and testing accuracy from training histories.  
  
 This function can accept multiple training histories, provided either as positional  
 arguments or as keyword arguments, but not both simultaneously.  
  
 Each history should be a list of dictionaries, with each dictionary containing  
 'train\_loss', 'test\_loss', and 'test\_acc' keys.  
  
 Parameters:  
 - \*args: Positional arguments, each an iterable of dictionaries.  
 - \*\*kwargs: Keyword arguments, each a named iterable of dictionaries.  
  
 Raises:  
 RuntimeError: If both positional and keyword arguments are provided.  
 ValueError: If provided history items do not contain required keys.  
 """* if args and kwargs:  
 raise RuntimeError("Cannot specify positional and keyword arguments at the same time")  
  
 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6)) # Use subplots for cleaner code  
  
 data = zip([""] \* len(args), args) if args else kwargs.items()  
  
 for label, history in data:  
 try:  
 train\_losses = [x['train\_loss'] for x in history]  
 test\_losses = [x['test\_loss'] for x in history]  
 test\_accuracies = [x['test\_acc'] for x in history]  
 except KeyError as e:  
 raise ValueError(f"Missing key in history data: {e}")  
  
 ax1.plot(train\_losses, label=f'{label} Train Loss')  
 ax1.plot(test\_losses, label=f'{label} Test Loss')  
 ax2.plot(test\_accuracies, label=f'{label} Test Accuracy')  
  
 ax1.set\_title('Loss vs Epochs')  
 ax1.set\_xlabel('Epochs')  
 ax1.set\_ylabel('Loss')  
 ax1.legend()  
  
 ax2.set\_title('Accuracy on Test Set vs Epochs')  
 ax2.set\_xlabel('Epochs')  
 ax2.set\_ylabel('Accuracy')  
 ax2.legend()  
  
 plt.tight\_layout()  
 plt.show()  
  
  
def load\_cifar10\_batch(file):  
 with open(file, 'rb') as f:  
 data = np.fromfile(f, dtype=np.uint8)  
 labels = data[::3073]  
 images = data.reshape(-1, 3073)[:, 1:].reshape(-1, 3, 32, 32).transpose(0, 2, 3, 1)  
 return images, labels  
  
  
def load\_cifar10\_batches(files):  
 images\_list, labels\_list = [], []  
 for file in files:  
 images, labels = load\_cifar10\_batch(file)  
 images\_list.append(images)  
 labels\_list.append(labels)  
 images = np.concatenate(images\_list, axis=0)  
 labels = np.concatenate(labels\_list, axis=0)  
 return images, labels  
  
  
class MyDataLoader:  
 def \_\_init\_\_(self, data, batch\_size=1, shuffle=False):  
 *"""  
 Initialize the data loader with a dataset, batch size, and shuffle option.  
  
 Parameters:  
 data (tuple): Tuple containing two elements, the images and their corresponding labels.  
 batch\_size (int): Number of samples per batch.  
 shuffle (bool): Whether to shuffle the indices of the samples.  
 """* self.batch\_size = batch\_size  
 self.shuffle = shuffle  
 self.image, self.label = data  
 self.num\_samples = len(self.image)  
 self.indices = np.arange(self.num\_samples)  
 if self.shuffle:  
 np.random.shuffle(self.indices)  
  
 def \_\_iter\_\_(self):  
 *"""  
 Reset the current index and return the iterator.  
  
 Returns:  
 self: The instance itself.  
 """* self.current\_idx = 0  
 return self  
  
 def \_\_next\_\_(self):  
 *"""  
 Return the next batch of images and labels.  
  
 Raises:  
 StopIteration: If all batches have been fetched.  
  
 Returns:  
 tuple: A batch of images and their corresponding labels.  
 """* if self.current\_idx >= self.num\_samples:  
 raise StopIteration  
  
 end\_idx = self.current\_idx + self.batch\_size  
 indices = self.indices[self.current\_idx:end\_idx]  
 images = self.image[indices]  
 labels = self.label[indices]  
 self.current\_idx += self.batch\_size  
 return images, labels  
  
  
def one\_hot(y, num\_classes):  
 *"""  
 Convert an array of numerical labels to one-hot encoded format.  
  
 Parameters:  
 y (int, list, or np.ndarray): The label or array of labels to be one-hot encoded.  
 num\_classes (int): The total number of classes.  
  
 Returns:  
 np.ndarray: One-hot encoded array for the given labels.  
  
 Raises:  
 ValueError: If `num\_classes` is less than the maximum label in `y`.  
 TypeError: If `y` is not of a type that can be converted to an np.ndarray.  
 """* if not isinstance(y, (int, list, np.ndarray)):  
 raise TypeError("Labels must be an int, list, or np.ndarray.")  
  
 y\_array = np.array(y, dtype=int)  
 if np.max(y\_array) >= num\_classes:  
 raise ValueError("Maximum label in `y` should be less than `num\_classes`.")  
  
 one\_hot\_encoded = np.eye(num\_classes, dtype=float)[y\_array]  
 return one\_hot\_encoded  
  
  
def relu(x):  
 *"""  
 Apply the ReLU (Rectified Linear Unit) activation function to each element in the input array.  
  
 The ReLU function outputs the input directly if it is positive; otherwise, it outputs zero.  
  
 Parameters:  
 x (np.ndarray): A numpy array of any shape containing numeric data.  
  
 Returns:  
 np.ndarray: An array of the same shape as `x`, where each element is the result of applying the ReLU function.  
  
 Raises:  
 TypeError: If the input `x` is not a numpy array.  
 """* if not isinstance(x, np.ndarray):  
 raise TypeError("Input must be a numpy array.")  
  
 return np.maximum(0, x)  
  
  
def softmax(logits, axis=1):  
 *"""  
 Compute the softmax of a set of scores (logits).  
  
 Args:  
 logits (ndarray): Input array containing raw scores for each class.  
 axis (int): Axis over which to perform the softmax, typically the feature or class dimension.  
  
 Returns:  
 ndarray: Softmax probabilities which sum to 1 along the specified axis.  
 """* # Stabilize the logits by subtracting the maximum value within the axis to prevent overflow in exp  
 shift\_logits = logits - np.max(logits, axis=axis, keepdims=True)  
 exp\_logits = np.exp(shift\_logits)  
  
 # Normalize the exponential scores to get probabilities  
 softmax\_probs = exp\_logits / np.sum(exp\_logits, axis=axis, keepdims=True)  
 return softmax\_probs  
  
  
def cross\_entropy(predictions, labels):  
 # Apply softmax to the raw predictions to transform them into probabilities  
 predictions = softmax(predictions)  
  
 # Convert labels to one-hot encoded vectors based on the number of classes  
 labels = one\_hot(labels, predictions.shape[1])  
  
 # Small constant to prevent numerical issues in logarithm calculation  
 epsilon = 1e-10  
  
 # Clip predictions to avoid log(0) which results in -inf  
 predictions = np.clip(predictions, epsilon, 1. - epsilon)  
  
 # Calculate cross-entropy loss  
 cross\_entropy\_loss = -np.sum(labels \* np.log(predictions), axis=1)  
 return np.mean(cross\_entropy\_loss)  
  
  
class MultiLayerNN:  
 def \_\_init\_\_(self, input\_size, lr=1e-2):  
 self.input\_size = input\_size  
 self.learning\_rate = lr  
  
 self.w1 = np.random.randn(input\_size, 512) \* 0.01  
 self.b1 = np.zeros(512)  
 self.w2 = np.random.randn(512, 256) \* 0.01  
 self.b2 = np.zeros(256)  
 self.w3 = np.random.randn(256, 128) \* 0.01  
 self.b3 = np.zeros(128)  
 self.w4 = np.random.randn(128, 10) \* 0.01  
 self.b4 = np.zeros(10)  
  
 def forward(self, x):  
 x = x.reshape(-1, self.input\_size)  
 self.z1 = x @ self.w1 + self.b1  
 x = relu(self.z1)  
 self.z2 = x @ self.w2 + self.b2  
 x = relu(self.z2)  
 self.z3 = x @ self.w3 + self.b3  
 x = relu(self.z3)  
 self.z4 = x @ self.w4 + self.b4  
 return softmax(self.z4, axis=1)  
  
 def \_\_call\_\_(self, inputs):  
 return self.forward(inputs)  
  
 def parameters(self):  
 yield self.w1  
 yield self.b1  
 yield self.w2  
 yield self.b2  
  
 def named\_parameters(self):  
 params = {'w1': self.w1, 'b1': self.b1, 'w2': self.w2, 'b2': self.b2}  
 return params.items()  
  
 def \_train\_step(self, images, labels, optimizer=None):  
 batch\_size = images.shape[0]  
 # forward  
 outputs = self(images)  
  
 # calculate loss  
 loss = cross\_entropy(outputs, labels)  
  
 # backpropagation  
 delta\_4 = outputs - one\_hot(labels, num\_classes=10)  
 grad\_w4 = relu(self.z3).T @ delta\_4 / batch\_size  
 grad\_b4 = np.sum(delta\_4, axis=0) / batch\_size  
  
 delta\_3 = delta\_4 @ self.w4.T  
 delta\_3[self.z3 <= 0] = 0  
 grad\_w3 = relu(self.z2).T @ delta\_3 / batch\_size  
 grad\_b3 = np.sum(delta\_3, axis=0) / batch\_size  
  
 delta\_2 = delta\_3 @ self.w3.T  
 delta\_2[self.z2 <= 0] = 0  
 grad\_w2 = relu(self.z1).T @ delta\_2 / batch\_size  
 grad\_b2 = np.sum(delta\_2, axis=0) / batch\_size  
  
 delta\_1 = delta\_2 @ self.w2.T  
 delta\_1[self.z1 <= 0] = 0  
 grad\_w1 = images.reshape(-1, self.input\_size).T @ delta\_1 / batch\_size  
 grad\_b1 = np.sum(delta\_1, axis=0) / batch\_size  
  
 # parameters update  
 if optimizer is None:  
 self.w4 -= self.learning\_rate \* grad\_w4  
 self.b4 -= self.learning\_rate \* grad\_b4  
 self.w3 -= self.learning\_rate \* grad\_w3  
 self.b3 -= self.learning\_rate \* grad\_b3  
 self.w2 -= self.learning\_rate \* grad\_w2  
 self.b2 -= self.learning\_rate \* grad\_b2  
 self.w1 -= self.learning\_rate \* grad\_w1  
 self.b1 -= self.learning\_rate\* grad\_b1  
 else:  
 optimizer.step([grad\_w1, grad\_b1, grad\_w2, grad\_b2, grad\_w3, grad\_b3, grad\_w4, grad\_b4])  
  
 return loss  
  
 def train\_and\_evaluate(self, num\_epochs, train\_loader, eval\_loader):  
 training\_history = []  
 optimizer = Adam(self.parameters(), learning\_rate=self.learning\_rate)  
 for current\_epoch in range(num\_epochs):  
 batch\_train\_losses = []  
 for batch\_images, batch\_labels in train\_loader:  
 batch\_train\_losses.append(self.\_train\_step(batch\_images, batch\_labels, optimizer))  
  
 batch\_eval\_losses = []  
 batch\_eval\_accuracies = []  
 for batch\_images, batch\_labels in eval\_loader:  
 predictions = self(batch\_images)  
 batch\_eval\_losses.append(cross\_entropy(predictions, batch\_labels))  
 predicted\_labels = np.argmax(predictions, axis=1)  
 batch\_accuracy = np.mean(predicted\_labels == batch\_labels)  
 batch\_eval\_accuracies.append(batch\_accuracy)  
  
 training\_history.append({  
 'train\_loss': np.mean(batch\_train\_losses),  
 'test\_loss': np.mean(batch\_eval\_losses),  
 'test\_acc': np.mean(batch\_eval\_accuracies)  
 })  
 print(  
 f"Epoch[{current\_epoch + 1:d}]: train\_loss: {training\_history[-1]['train\_loss']:.4f}, test\_loss: {training\_history[-1]['test\_loss']:.4f}, test\_acc: {training\_history[-1]['test\_acc']:.4f}"  
 )  
 return training\_history  
  
  
class Adam():  
 def \_\_init\_\_(self, parameters, learning\_rate=1e-2, momentum\_factors=(0.9, 0.999), stability\_constant=1e-8):  
 self.parameters = list(parameters)  
 self.learning\_rate = learning\_rate  
 self.momentum\_factors = momentum\_factors  
 self.stability\_constant = stability\_constant  
 self.time\_step = 0  
 self.first\_moment = [np.zeros\_like(param) for param in self.parameters]  
 self.second\_moment = [np.zeros\_like(param) for param in self.parameters]  
  
 def step(self, gradients):  
 self.time\_step += 1  
 for index, param, grad in zip(range(len(gradients)), self.parameters, gradients):  
 self.first\_moment[index] = self.momentum\_factors[0] \* self.first\_moment[index] + (  
 1 - self.momentum\_factors[0]) \* grad  
 self.second\_moment[index] = self.momentum\_factors[1] \* self.second\_moment[index] + (  
 1 - self.momentum\_factors[1]) \* (grad \*\* 2)  
  
 corrected\_first\_moment = self.first\_moment[index] / (1 - self.momentum\_factors[0] \*\* self.time\_step)  
 corrected\_second\_moment = self.second\_moment[index] / (1 - self.momentum\_factors[1] \*\* self.time\_step)  
  
 param -= self.learning\_rate \* corrected\_first\_moment / (  
 np.sqrt(corrected\_second\_moment) + self.stability\_constant)  
  
  
batch\_size = 64  
epochs = 20  
  
  
train\_files = ['./data/cifar/train/data\_batch\_{}.bin'.format(i) for i in range(1, 6)]  
test\_file = './data/cifar/test/test\_batch.bin'  
  
train\_images, train\_labels = load\_cifar10\_batches(train\_files)  
train\_images, train\_labels = cifar\_transform(train\_images), train\_labels  
  
test\_images, test\_labels = load\_cifar10\_batch(test\_file)  
test\_images, test\_labels = cifar\_transform(test\_images), test\_labels  
  
train\_dl = MyDataLoader((train\_images, train\_labels), batch\_size, True)  
test\_dl = MyDataLoader((test\_images, test\_labels), batch\_size)  
  
model = MultiLayerNN(32 \* 32 \* 3)  
history = model.train\_and\_evaluate(epochs, train\_dl, test\_dl)  
plot\_history(history)

**train\_man\_mnist.py**

import numpy as np  
import matplotlib.pyplot as plt  
import os  
import struct  
  
  
def mnist\_transform(images):  
 return images.astype(np.float32) / 255.0  
  
  
def plot\_history(\*args, \*\*kwargs):  
 fig = plt.figure(figsize=(8, 4))  
 ax1 = fig.add\_subplot(1, 2, 1)  
 ax2 = fig.add\_subplot(1, 2, 2)  
 if len(args) > 0 and len(kwargs) > 0:  
 raise RuntimeError(  
 "cannot specify position args and keyword args at the same time")  
  
 data = zip([""] \* len(args), args) if len(args) > 0 else kwargs.items()  
 for label, his in data:  
 ax1.plot([x['train\_loss'] for x in his], label=label + ' train loss')  
 ax1.plot([x['test\_loss'] for x in his], label=label + ' test loss')  
 ax2.plot([x['test\_acc'] for x in his], label=label + ' accuracy')  
 ax2.set\_ylabel("Accuracy")  
 ax2.set\_title('Accuracy on test set vs Epochs')  
 ax1.set\_ylabel('Loss')  
 ax1.set\_title('Loss vs Epochs')  
 ax1.legend()  
 ax2.legend()  
 fig.supxlabel("Epochs")  
 fig.tight\_layout()  
 plt.show()  
  
  
def load\_mnist\_images(file\_path):  
 with open(file\_path, 'rb') as f:  
 magic, num, rows, cols = struct.unpack(">IIII", f.read(16))  
 images = np.frombuffer(f.read(), dtype=np.uint8).reshape(num, 784)  
 return images  
  
  
def load\_mnist\_labels(file\_path):  
 with open(file\_path, 'rb') as f:  
 magic, num = struct.unpack(">II", f.read(8))  
 labels = np.frombuffer(f.read(), dtype=np.uint8)  
 return labels  
  
  
import numpy as np  
  
  
class MyDataLoader:  
 def \_\_init\_\_(self, data, batch\_size=1, shuffle=False):  
 *"""  
 Initialize the data loader with a dataset, batch size, and shuffle option.  
  
 Parameters:  
 data (tuple): Tuple containing two elements, the images and their corresponding labels.  
 batch\_size (int): Number of samples per batch.  
 shuffle (bool): Whether to shuffle the indices of the samples.  
 """* self.batch\_size = batch\_size  
 self.shuffle = shuffle  
 self.image, self.label = data  
 self.num\_samples = len(self.image)  
 self.indices = np.arange(self.num\_samples)  
 if self.shuffle:  
 np.random.shuffle(self.indices)  
  
 def \_\_iter\_\_(self):  
 *"""  
 Reset the current index and return the iterator.  
  
 Returns:  
 self: The instance itself.  
 """* self.current\_idx = 0  
 return self  
  
 def \_\_next\_\_(self):  
 *"""  
 Return the next batch of images and labels.  
  
 Raises:  
 StopIteration: If all batches have been fetched.  
  
 Returns:  
 tuple: A batch of images and their corresponding labels.  
 """* if self.current\_idx >= self.num\_samples:  
 raise StopIteration  
  
 end\_idx = self.current\_idx + self.batch\_size  
 indices = self.indices[self.current\_idx:end\_idx]  
 images = self.image[indices]  
 labels = self.label[indices]  
 self.current\_idx += self.batch\_size  
 return images, labels  
  
  
def one\_hot(y, num\_classes):  
 *"""  
 Convert an array of numerical labels to one-hot encoded format.  
  
 Parameters:  
 y (int, list, or np.ndarray): The label or array of labels to be one-hot encoded.  
 num\_classes (int): The total number of classes.  
  
 Returns:  
 np.ndarray: One-hot encoded array for the given labels.  
  
 Raises:  
 ValueError: If `num\_classes` is less than the maximum label in `y`.  
 TypeError: If `y` is not of a type that can be converted to an np.ndarray.  
 """* if not isinstance(y, (int, list, np.ndarray)):  
 raise TypeError("Labels must be an int, list, or np.ndarray.")  
  
 y\_array = np.array(y, dtype=int)  
 if np.max(y\_array) >= num\_classes:  
 raise ValueError("Maximum label in `y` should be less than `num\_classes`.")  
  
 one\_hot\_encoded = np.eye(num\_classes, dtype=float)[y\_array]  
 return one\_hot\_encoded  
  
  
def relu(x):  
 *"""  
 Apply the ReLU (Rectified Linear Unit) activation function to each element in the input array.  
  
 The ReLU function outputs the input directly if it is positive; otherwise, it outputs zero.  
  
 Parameters:  
 x (np.ndarray): A numpy array of any shape containing numeric data.  
  
 Returns:  
 np.ndarray: An array of the same shape as `x`, where each element is the result of applying the ReLU function.  
  
 Raises:  
 TypeError: If the input `x` is not a numpy array.  
 """* if not isinstance(x, np.ndarray):  
 raise TypeError("Input must be a numpy array.")  
  
 return np.maximum(0, x)  
  
  
def softmax(logits, axis=1):  
 *"""  
 Compute the softmax of a set of scores (logits).  
  
 Args:  
 logits (ndarray): Input array containing raw scores for each class.  
 axis (int): Axis over which to perform the softmax, typically the feature or class dimension.  
  
 Returns:  
 ndarray: Softmax probabilities which sum to 1 along the specified axis.  
 """* # Stabilize the logits by subtracting the maximum value within the axis to prevent overflow in exp  
 shift\_logits = logits - np.max(logits, axis=axis, keepdims=True)  
 exp\_logits = np.exp(shift\_logits)  
  
 # Normalize the exponential scores to get probabilities  
 softmax\_probs = exp\_logits / np.sum(exp\_logits, axis=axis, keepdims=True)  
 return softmax\_probs  
  
  
def cross\_entropy(predictions, labels):  
 # Apply softmax to the raw predictions to transform them into probabilities  
 predictions = softmax(predictions)  
  
 # Convert labels to one-hot encoded vectors based on the number of classes  
 labels = one\_hot(labels, predictions.shape[1])  
  
 # Small constant to prevent numerical issues in logarithm calculation  
 epsilon = 1e-10  
  
 # Clip predictions to avoid log(0) which results in -inf  
 predictions = np.clip(predictions, epsilon, 1. - epsilon)  
  
 # Calculate cross-entropy loss  
 cross\_entropy\_loss = -np.sum(labels \* np.log(predictions), axis=1)  
 return np.mean(cross\_entropy\_loss)  
  
  
class MultiLayerNN:  
 def \_\_init\_\_(self, input\_size, output\_size=10, lr=1e-3):  
 self.input\_size = input\_size  
 self.output\_size = output\_size  
 self.lr = lr  
 self.initialize\_weights()  
  
 def initialize\_weights(self):  
 self.w1 = np.random.randn(self.input\_size, 128) \* 0.01  
 self.b1 = np.zeros(128)  
 self.w2 = np.random.randn(128, 64) \* 0.01  
 self.b2 = np.zeros(64)  
 self.w3 = np.random.randn(64, self.output\_size) \* 0.01  
 self.b3 = np.zeros(self.output\_size)  
  
 def forward(self, x):  
 x = x.reshape(-1, self.input\_size)  
 self.z1 = x @ self.w1 + self.b1  
 x = relu(self.z1)  
 self.z2 = x @ self.w2 + self.b2  
 x = relu(self.z2)  
 self.z3 = x @ self.w3 + self.b3  
 return softmax(self.z3, axis=1)  
  
 def \_\_call\_\_(self, inputs):  
 return self.forward(inputs)  
  
 def parameters(self):  
 yield self.w1  
 yield self.b1  
 yield self.w2  
 yield self.b2  
  
 def named\_parameters(self):  
 params = {'w1': self.w1, 'b1': self.b1, 'w2': self.w2, 'b2': self.b2}  
 return params.items()  
  
 def \_train\_step(self, images, labels, optimizer=None):  
 batch\_size = images.shape[0]  
 # forward  
 outputs = self(images)  
  
 # calculate loss  
 loss = cross\_entropy(outputs, labels)  
  
 # backpropagation  
 delta\_3 = outputs - one\_hot(labels, num\_classes=10)  
 grad\_w3 = relu(self.z2).T @ delta\_3 / batch\_size  
 grad\_b3 = np.sum(delta\_3, axis=0) / batch\_size  
  
 delta\_2 = delta\_3 @ self.w3.T  
 delta\_2[self.z2 <= 0] = 0  
 grad\_w2 = relu(self.z1).T @ delta\_2 / batch\_size  
 grad\_b2 = np.sum(delta\_2, axis=0) / batch\_size  
  
 delta\_1 = delta\_2 @ self.w2.T  
 delta\_1[self.z1 <= 0] = 0  
 grad\_w1 = images.reshape(-1, self.input\_size).T @ delta\_1 / batch\_size  
 grad\_b1 = np.sum(delta\_1, axis=0) / batch\_size  
  
 # parameters update  
 if optimizer is None:  
 self.w3 -= self.lr \* grad\_w3  
 self.b3 -= self.lr \* grad\_b3  
 self.w2 -= self.lr \* grad\_w2  
 self.b2 -= self.lr \* grad\_b2  
 self.w1 -= self.lr \* grad\_w1  
 self.b1 -= self.lr \* grad\_b1  
 else:  
 optimizer.step([grad\_w1, grad\_b1, grad\_w2, grad\_b2, grad\_w3, grad\_b3])  
  
 return loss  
  
 def train\_and\_evaluate(self, epochs, train\_dl, test\_dl):  
 history = []  
 optim = Adam(self.parameters())  
 for epoch in range(epochs):  
 train\_losses = []  
 for images, labels in train\_dl:  
 train\_losses.append(self.\_train\_step(images, labels, optim))  
  
 test\_losses = []  
 test\_accs = []  
 for images, labels in test\_dl:  
 outputs = self(images)  
 test\_losses.append(cross\_entropy(outputs, labels))  
 preds = np.argmax(outputs, axis=1)  
 test\_accs.append(  
 np.array(np.sum(preds == labels).item() / len(preds)))  
  
 history.append({  
 'train\_loss': np.stack(train\_losses).mean().item(),  
 'test\_loss': np.stack(test\_losses).mean().item(),  
 'test\_acc': np.stack(test\_accs).mean()  
 })  
 print(  
 f"Epoch[{epoch + 1:d}]: train\_loss: {history[-1]['train\_loss']:.4f}, test\_loss: {history[-1]['test\_loss']:.4f}, test\_acc: {history[-1]['test\_acc']:.4f}"  
 )  
 return history  
  
  
class Adam():  
 def \_\_init\_\_(self, parameters, learning\_rate=1e-2, momentum\_factors=(0.9, 0.999), stability\_constant=1e-8):  
 self.parameters = list(parameters)  
 self.learning\_rate = learning\_rate  
 self.momentum\_factors = momentum\_factors  
 self.stability\_constant = stability\_constant  
 self.time\_step = 0  
 self.first\_moment = [np.zeros\_like(param) for param in self.parameters]  
 self.second\_moment = [np.zeros\_like(param) for param in self.parameters]  
  
 def step(self, gradients):  
 self.time\_step += 1  
 for index, param, grad in zip(range(len(gradients)), self.parameters, gradients):  
 self.first\_moment[index] = self.momentum\_factors[0] \* self.first\_moment[index] + (  
 1 - self.momentum\_factors[0]) \* grad  
 self.second\_moment[index] = self.momentum\_factors[1] \* self.second\_moment[index] + (  
 1 - self.momentum\_factors[1]) \* (grad \*\* 2)  
  
 corrected\_first\_moment = self.first\_moment[index] / (1 - self.momentum\_factors[0] \*\* self.time\_step)  
 corrected\_second\_moment = self.second\_moment[index] / (1 - self.momentum\_factors[1] \*\* self.time\_step)  
  
 param -= self.learning\_rate \* corrected\_first\_moment / (  
 np.sqrt(corrected\_second\_moment) + self.stability\_constant)  
  
  
batch\_size = 64  
epochs = 20  
  
  
train\_images = load\_mnist\_images('./data/mnist/train/train-images.idx3-ubyte')  
train\_labels = load\_mnist\_labels('./data/mnist/train/train-labels.idx1-ubyte')  
test\_images = load\_mnist\_images('./data/mnist/test/t10k-images.idx3-ubyte')  
test\_labels = load\_mnist\_labels('./data/mnist/test/t10k-labels.idx1-ubyte')  
  
train\_images = mnist\_transform(train\_images)  
test\_images = mnist\_transform(test\_images)  
  
train\_dl = MyDataLoader((train\_images, train\_labels), batch\_size, True)  
test\_dl = MyDataLoader((test\_images, test\_labels), batch\_size)  
  
model = MultiLayerNN(784)  
history = model.train\_and\_evaluate(epochs, train\_dl, test\_dl)  
plot\_history(history)

**train\_mindspore\_cifar.py**

import mindspore  
from mindspore import nn  
from mindspore import ops  
from mindspore import Tensor  
from mindspore.dataset import vision, transforms  
from mindspore.dataset import Cifar10Dataset  
import matplotlib.pyplot as plt  
  
  
def prepare\_dataset(dataset, batch\_size, mean=(0.4914, 0.4822, 0.4465), std=(0.2023, 0.1994, 0.2010)):  
 transformations = [  
 vision.Resize((32, 32)), # 确保图片是32x32的  
 vision.Rescale(1.0 / 255.0, 0),  
 vision.Normalize(mean=mean, std=std),  
 vision.HWC2CHW()  
 ]  
 dataset = dataset.map(operations=transformations, input\_columns="image")  
 dataset = dataset.map(operations=transforms.TypeCast(mindspore.int32), input\_columns="label")  
 dataset = dataset.batch(batch\_size)  
 return dataset  
  
  
class Network(nn.Cell):  
 def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, lr=2\*(1e-3)):  
 super().\_\_init\_\_()  
 self.flatten = nn.Flatten()  
 self.fc1 = nn.Dense(input\_size, hidden\_size)  
 self.relu = nn.ReLU()  
 self.fc2 = nn.Dense(hidden\_size, output\_size)  
 self.loss\_fn = nn.CrossEntropyLoss()  
 self.optimizer = nn.SGD(self.trainable\_params(), lr)  
  
 def construct(self, x):  
 x = self.flatten(x)  
 x = self.fc1(x)  
 x = self.relu(x)  
 x = self.fc2(x)  
 return x  
  
 def forward(self, data, label):  
 logits = self(data)  
 loss = self.loss\_fn(logits, label)  
 return loss, logits  
  
 def train\_and\_evaluate(self, total\_epochs, training\_dataset, validation\_dataset):  
 epoch\_history = []  
 for epoch in range(total\_epochs):  
 gradient\_function = ops.value\_and\_grad(self.forward,  
 None,  
 self.optimizer.parameters,  
 has\_aux=True)  
  
 self.set\_train(True)  
 total\_train\_batches = training\_dataset.get\_dataset\_size()  
 epoch\_train\_loss = 0  
 for inputs, targets in training\_dataset.create\_tuple\_iterator():  
 (batch\_loss, \_), gradients = gradient\_function(inputs, targets)  
 epoch\_train\_loss += ops.depend(batch\_loss, self.optimizer(gradients))  
  
 self.set\_train(False)  
 total\_validation\_batches = validation\_dataset.get\_dataset\_size()  
 total\_items, validation\_loss, correct\_predictions = 0, 0, 0  
 for inputs, targets in validation\_dataset.create\_tuple\_iterator():  
 predictions = self(inputs)  
 total\_items += len(inputs)  
 validation\_loss += self.loss\_fn(predictions, targets).asnumpy()  
 correct\_predictions += (predictions.argmax(1) == targets).asnumpy().sum()  
 epoch\_history.append({  
 'train\_loss': float(epoch\_train\_loss / total\_train\_batches),  
 'test\_loss': float(validation\_loss / total\_validation\_batches),  
 'test\_acc': float(correct\_predictions / total\_items)  
 })  
 print(  
 f"Epoch[{epoch + 1:d}]: train\_loss: {epoch\_history[-1]['train\_loss']:.4f}, test\_loss: {epoch\_history[-1]['test\_loss']:.4f}, test\_acc: {epoch\_history[-1]['test\_acc']:.4f}"  
 )  
 return epoch\_history  
  
  
def plot\_history(history):  
 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))  
 train\_losses = [x['train\_loss'] for x in history]  
 test\_losses = [x['test\_loss'] for x in history]  
 accuracies = [x['test\_acc'] for x in history]  
 ax1.plot(train\_losses, label='Train Loss')  
 ax1.plot(test\_losses, label='Test Loss')  
 ax2.plot(accuracies, label='Test Accuracy')  
 ax1.set\_xlabel('Epochs')  
 ax2.set\_xlabel('Epochs')  
 ax1.set\_ylabel('Loss')  
 ax2.set\_ylabel('Accuracy')  
 ax1.set\_title('Loss over Epochs')  
 ax2.set\_title('Accuracy over Epochs')  
 ax1.legend()  
 ax2.legend()  
 plt.show()  
  
  
def predict\_image(image\_tensor, model):  
 *"""Function to predict the label of a single image using the trained model."""* # Ensure the image tensor is in the right shape [batch\_size, channels, height, width]  
 image\_tensor = image\_tensor.expand\_dims(0) # Add batch dimension  
 # Forward pass to get logits  
 logits = model(image\_tensor)  
 # Get predicted label index  
 pred\_label\_index = logits.argmax(1).asnumpy()[0]  
 return pred\_label\_index  
  
  
# Create a function to convert image data into MindSpore tensor with correct format  
def process\_single\_image(image\_path):  
 image = plt.imread(image\_path) # Read the image from file  
 # Apply transformations  
 transforms = vision.Compose([  
 vision.Resize((32, 32)), # Resize to 32x32 if needed  
 vision.Rescale(1.0 / 255.0, 0),  
 vision.Normalize(mean=(0.4914, 0.4822, 0.4465), std=(0.2023, 0.1994, 0.2010)),  
 vision.HWC2CHW()  
 ])  
 image = transforms(image)  
 return Tensor(image, mindspore.float32)  
  
  
train\_dataset = Cifar10Dataset('data/cifar/train', usage='train', shuffle=True)  
test\_dataset = Cifar10Dataset('data/cifar/test', usage='test')  
  
train\_ds = prepare\_dataset(train\_dataset, 64)  
test\_ds = prepare\_dataset(test\_dataset, 64)  
  
  
model = Network(32 \* 32 \* 3, 256, 10)  
  
epochs = 50  
  
history = model.train\_and\_evaluate(epochs, train\_ds, test\_ds)  
  
plot\_history(history)

**train\_mindspore\_mnist.py**

import mindspore  
import random  
from mindspore import nn  
from mindspore import ops  
from mindspore import Tensor  
from mindspore.dataset import vision, transforms  
from mindspore.dataset import MnistDataset  
import matplotlib.pyplot as plt  
  
  
def prepare\_dataset(dataset, batch\_size, mean=0.1307, std=0.3081):  
 transformations = [  
 vision.Rescale(1.0 / 255.0, 0),  
 vision.Normalize(mean=(mean,), std=(std,)),  
 vision.HWC2CHW()  
 ]  
 dataset = dataset.map(operations=transformations, input\_columns="image")  
 dataset = dataset.map(operations=transforms.TypeCast(mindspore.int32), input\_columns="label")  
 dataset = dataset.batch(batch\_size)  
 return dataset  
  
  
class Network(nn.Cell):  
 def \_\_init\_\_(self,input\_size, hidden\_size, output\_size, lr=1e-2):  
 super().\_\_init\_\_()  
 self.flatten = nn.Flatten()  
 self.fc1 = nn.Dense(input\_size, hidden\_size)  
 self.relu = nn.ReLU()  
 self.fc2 = nn.Dense(hidden\_size, output\_size)  
 self.loss\_fn = nn.CrossEntropyLoss()  
 self.optimizer = nn.SGD(self.trainable\_params(), lr)  
  
 def construct(self, x):  
 x = self.flatten(x)  
 x = self.fc1(x)  
 x = self.relu(x)  
 x = self.fc2(x)  
 return x  
  
 def forward(self, data, label):  
 logits = self(data)  
 loss = self.loss\_fn(logits, label)  
 return loss, logits  
  
 def train\_and\_evaluate(self, total\_epochs, training\_dataset, validation\_dataset):  
 epoch\_history = []  
 for epoch in range(total\_epochs):  
 gradient\_function = ops.value\_and\_grad(self.forward,  
 None,  
 self.optimizer.parameters,  
 has\_aux=True)  
  
 self.set\_train(True)  
 total\_train\_batches = training\_dataset.get\_dataset\_size()  
 epoch\_train\_loss = 0  
 for inputs, targets in training\_dataset.create\_tuple\_iterator():  
 (batch\_loss, \_), gradients = gradient\_function(inputs, targets)  
 epoch\_train\_loss += ops.depend(batch\_loss, self.optimizer(gradients))  
  
 self.set\_train(False)  
 total\_validation\_batches = validation\_dataset.get\_dataset\_size()  
 total\_items, validation\_loss, correct\_predictions = 0, 0, 0  
 for inputs, targets in validation\_dataset.create\_tuple\_iterator():  
 predictions = self(inputs)  
 total\_items += len(inputs)  
 validation\_loss += self.loss\_fn(predictions, targets).asnumpy()  
 correct\_predictions += (predictions.argmax(1) == targets).asnumpy().sum()  
 epoch\_history.append({  
 'train\_loss': float(epoch\_train\_loss / total\_train\_batches),  
 'test\_loss': float(validation\_loss / total\_validation\_batches),  
 'test\_acc': float(correct\_predictions / total\_items)  
 })  
 print(  
 f"Epoch[{epoch + 1:d}]: train\_loss: {epoch\_history[-1]['train\_loss']:.4f}, test\_loss: {epoch\_history[-1]['test\_loss']:.4f}, test\_acc: {epoch\_history[-1]['test\_acc']:.4f}"  
 )  
 return epoch\_history  
  
  
def plot\_history(history):  
 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))  
 train\_losses = [x['train\_loss'] for x in history]  
 test\_losses = [x['test\_loss'] for x in history]  
 accuracies = [x['test\_acc'] for x in history]  
 ax1.plot(train\_losses, label='Train Loss')  
 ax1.plot(test\_losses, label='Test Loss')  
 ax2.plot(accuracies, label='Test Accuracy')  
 ax1.set\_xlabel('Epochs')  
 ax2.set\_xlabel('Epochs')  
 ax1.set\_ylabel('Loss')  
 ax2.set\_ylabel('Accuracy')  
 ax1.set\_title('Loss over Epochs')  
 ax2.set\_title('Accuracy over Epochs')  
 ax1.legend()  
 ax2.legend()  
 plt.show()  
  
  
def predict\_image(image\_tensor, model):  
 *"""Function to predict the label of a single image using the trained model."""* # Ensure the image tensor is in the right shape [batch\_size, channels, height, width]  
 image\_tensor = image\_tensor.expand\_dims(0) # Add batch dimension  
 # Forward pass to get logits  
 logits = model(image\_tensor)  
 # Get predicted label index  
 pred\_label\_index = logits.argmax(1).asnumpy()[0]  
 return pred\_label\_index  
  
  
# Create a function to convert image data into MindSpore tensor with correct format  
def process\_single\_image(image\_path):  
 *"""Process an image to be model-ready."""* image = plt.imread(image\_path) # Read the image from file  
 if image.ndim == 2: # grayscale image  
 image = np.expand\_dims(image, axis=-1)  
 # Apply transformations  
 transforms = vision.Compose([  
 vision.Resize((28, 28)),  
 vision.Rescale(1.0 / 255.0, 0),  
 vision.Normalize(mean=(0.1307,), std=(0.3081,)),  
 vision.HWC2CHW()  
 ])  
 image = transforms(image)  
 return Tensor(image, mindspore.float32)  
  
  
train\_dataset = MnistDataset('data/mnist/train', shuffle=True)  
test\_dataset = MnistDataset('data/mnist/test')  
test\_single = MnistDataset('data/mnist/test', shuffle=False)  
train\_ds = prepare\_dataset(train\_dataset, 64)  
test\_ds = prepare\_dataset(test\_dataset, 64)  
  
test\_sg = prepare\_dataset(test\_single, 1)  
  
model = Network(28 \* 28, 256, 10)  
  
epochs = 50  
  
history = model.train\_and\_evaluate(epochs, train\_ds, test\_ds)  
  
plot\_history(history)  
  
labels\_mapping = {  
 0: '0', 1: '1', 2: '2', 3: '3', 4: '4',  
 5: '5', 6: '6', 7: '7', 8: '8', 9: '9'  
}  
  
# 获取数据集中的一张图片及其标签，用于预测  
data\_iterator = test\_sg.create\_tuple\_iterator()  
num = random.randint(1,100)  
for i in range(num):  
 \_, \_ = next(data\_iterator)  
sample\_image, sample\_label = next(data\_iterator)  
sample\_image = sample\_image.asnumpy().squeeze() # Remove batch dimension and channel dimension for plotting  
  
# 使用model进行预测  
predicted\_index = predict\_image(Tensor(sample\_image, mindspore.float32), model)  
predicted\_label = labels\_mapping[predicted\_index]  
  
# 显示图片和预测结果  
plt.imshow(sample\_image, cmap='gray')  
plt.title(f'Label: {sample\_label.asnumpy()[0]}, Predicted: {predicted\_label}')  
plt.show()

**心得体会**

通过这次模式识别实验，我深刻体验到了数据降维技术和分类算法的重要性。在实验中，我采用了PCA和LDA两种降维方法，并结合逻辑回归分类器进行了实际的分类训练和测试。通过实验，我发现降维后的数据集在分类准确率上有了显著的提升，这验证了降维技术在提高数据处理效率和分类性能中的有效性。

在实验的过程中，我还学习到了如何利用MindSpore平台来实现这些算法。与传统方法相比，MindSpore平台提供了更加高效和便捷的方式来处理大规模数据。通过对比实验，我观察到在MindSpore平台上实现的算法与传统算法在准确率和迭代收敛次数上存在细微的差异，这可能与平台对算法优化和底层实现有关。

此外，我也尝试将自己实现的算法应用于不同的数据集，并与MindSpore及sklearn平台的官方算法进行了比较。这一过程不仅增强了我的编程技能，也让我更深入地理解了各种算法的内在机制及其适用场景。

总体来说，这次实验不仅增强了我的实际操作能力，也使我对模式识别的理论和应用有了更深的认识。未来，我期望能将这些知识应用到更复杂的实际问题中，以解决实际生活或工作中遇到的挑战。