2023-2024 第二学期
大数据及数据挖掘 课程作业

组序号：第八组				
组员（学号）：121072021018 丁锦源				
121072021015 潘佳烨 121072021013 陈宝明				
	第一题	第二题	第三题	
文档成绩				
	20 分	10 分	5 分	
客观成绩				
	20 分	20 分	25 分	
总成绩				

问题一：在 10,000,000 个样本点中聚类，聚为 1,000,000 类

1.1 数据介绍

1. 数据文件：Task1-聚类数据.zip，其中含有 10,000,000 个样本，每个样本为一行，第一列为样本
号，剩余 64 列为样本特征。
2. 教师持有该数据的聚类参考标准。

1.2 任务介绍

将以上数据聚为 1,000,000 个类，每类 10 个样本，并按照样本号顺序输出所聚类别号，类别号取值范围为[1,1000000]。

1.3 问题求解

1.方案介绍：

首先，考虑到数据量庞大，为了能够更好的提取到有用的特征，我们的方法首先进行 PCA 降维至 16 维（前面降到 8 维发现效果不好）。

其次，我们随机从所有样本中采样 10,000 个样本（选 100,000 发现跑不动），并使用 KMeans 算法对其进行初步聚类得到 1000 个初步聚类中心。使用得到的 1,000 个初步聚类中心对整个数据集进行分类预测。

再者，对于每个类别内的样本进行细分聚类。第一，对于每个类别的个数我们将其调整到 10 的倍数，将多出来的样本放进一个临时集合里。第二，对于这 1000 个类别中的每一个类别，我们将他们再进行细聚类，分别将其聚类为（该类样本数除以 10）类，算出（该类样本数除以 10）个聚类中心，并从这些聚类中心中采样距离最近的 10 个样本归为一类。

最后我们将临时集合里的样本再进行最后的聚类。这样每一个类别都有 10 个样本。

总之，就是大类-小类两级的套圈层次化 kmeans 聚类算法啦。首先对全体样本进行粗聚类得到大类,然后再对每个大类内部进行细分聚类,最终得到每个样本的类别标签。这种方法可以有效地处理大规模数据集,提高聚类效率。

2. 代码部分：

2.1 调用的库

```
import numpy as np
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
```

```
import pandas as pd
import time
from sklearn.metrics.pairwise import euclidean_distances
from time import sleep
```

2.2 读取 CSV 文件"seqkmer.csv",获取数据集。数据集包含样本 ID 和特征矩阵。

```
# 读取数据
df = pd.read_csv("seqkmer.csv")
data = df.values

# 提取样本号和特征
sample_ids = data[:, 0]
X = data[:, 1:]
```

2.3 将数据从原始高维空间降到 16 维空间。

```
print("开始 PCA 降维...")
start_time = time.time()
pca = PCA(n_components=16)
X_pca = pca.fit_transform(X)
print(f"PCA 降维完成，用时: {time.time() - start_time:.2f}秒")
# 保存 PCA 结果到文件
np.save('pca_results.npy', X_pca)
print("PCA 结果已保存到文件。")
```

2.4 随机选取 10,000 个样本,使用 KMeans 算法对这部分样本进行初步聚类,得到 1,000 个初步聚类中心，需要注意的是这些类别的样本数量并不相同。

```
# 第一步：随机采样 10,000 个点进行初步聚类
print("开始初步聚类...")
start_time = time.time()
n_samples_for_initial_clustering = 10000
indices = np.random.choice(X_pca.shape[0],
n_samples_for_initial_clustering, replace=False)
X_sample = X_pca[indices]

# 初步聚类得到 1000 个粗聚类中心
kmeans_initial = KMeans(n_clusters=1000, n_init=10,random_state=42)
labels_initial = kmeans_initial.fit_predict(X_sample)
centers_initial = kmeans_initial.cluster_centers_
print(f"初步聚类完成，用时: {time.time() - start_time:.2f}秒")
```

2.5 使用得到的 1,000 个初步聚类中心,对整个数据集进行分类预测,得到每个样本的初步类别标签。将初步聚类的结果保存到"coarse_clustering_results.csv"文件中,并构建一个字典"cluster_to_sample_ids",记录每个类别对应的样本 ID 列表。

```
# 使用 KMeans 模型对所有样本进行预测（分配）
print("开始分配样本到粗聚类中心...")
start_time = time.time()
labels = kmeans_initial.predict(X_pca)  # 使用 KMeans 模型预测所有样本的类别
print(f"样本分配完成，用时: {time.time() - start_time:.2f}秒")

# 保存结果
print("保存粗聚类结果...")
start_time = time.time()
# 创建一个新的 DataFrame，包含样本号和类别号
result_df = pd.DataFrame({
    'SampleID': sample_ids,  # 样本号
    'ClusterID': labels  # 类别号
})
# 保存 DataFrame 到 CSV 文件
result_df.to_csv("coarse_clustering_results.csv", index=False)
print(f"保存粗聚类结果完成，用时: {time.time() - start_time:.2f}秒")

# 创建一个字典来保存每个类别号及其对应的样本 ID 列表
cluster_to_sample_ids = {}

# 使用 groupby 按 ClusterID 分组，并迭代组来填充字典
for cluster_id, group in result_df.groupby('ClusterID'):
    cluster_to_sample_ids[cluster_id] = group['SampleID'].tolist()


print("保存类别到样本 ID 的映射完成。")
```

2.6 定义一个名为"kmeans_with_top_samples"的函数,用于对每个大类内的样本进行细分聚类。对于初步聚类得到的 1000 个类中的每一个类别的所有样本来说，我们去除一些样本，使得每个类别的样本数量为 10 的倍数，被去除的样本被添加进临时数组 shengyu 中。处理完之后，我们将每个类的样本进行（样本数//10)聚类，找出（样本数//10)个聚类中心，并将距离这些聚类中心最近的 10 个样本分配给该类别。这样我们得到了每个类别都只有 10 个样本的聚类。（记住，若大于 10,000，则选取每个大类前 10,000 个样本，若小于 10,000 个则全部样本）。使用 KMeans 算法进行聚类,得到每个大类内部的细分类别。

```
def kmeans_with_top_samples(X_pca, n_clusters,
sample_idlist,leibiehao):
    #执行 KMeans 聚类
    kmeans = KMeans(n_clusters=n_clusters, n_init=10,random_state=42)
    X_pca=[X_pca[i-1] for i in sample_idlist]
    labels = kmeans.fit_predict(X_pca)
    centroids = kmeans.cluster_centers_
```

```
    # 初始化结果数组
    result = np.zeros((len(sample_idlist), 2), dtype=object)
    result[:, 0] = sample_idlist

    # 分配每个聚类中心的最近 10 个样本，并打印进度
    assigned_samples = set()
    for i in range(n_clusters):
        # 计算到当前聚类中心的距离
        distances = euclidean_distances(X_pca, [centroids[i]])
        distances = distances.flatten()

        # 模拟处理时间以便看到进度更新
        #sleep(0.1)  # 在实际应用中，这里不会有 sleep

        # 获取距离最小的 10 个样本的索引（如果可用）
        top_indices = np.argsort(distances)[:10]

        # 将这些样本分配给当前类别
        for idx in top_indices:
            if idx not in assigned_samples:
                result[idx, 1] = i
                assigned_samples.add(idx)

                # 将剩余的样本分配到“剩余”类别（假设为 n_clusters）
    for idx, label in enumerate(result[:, 1]):
        if label == 0:  # 假设我们不使用 0 作为实际类别
            result[idx, 1] = n_clusters  # 剩余类别可以是 n_clusters
或任何其他标识符
    for i in range(len(result)):
        result[i][1]+=leibiehao
    return result

zuizhong=[]
shengyu=[]
leibiehao=0
# 打印字典以检查其内容
for cluster_id, sample_ids_list in cluster_to_sample_ids.items():
    a=len(sample_ids_list)
    if a>10000:
        caiyangshu = 10000
        n = 1000
        temp = sample_ids_list[:caiyangshu]
        shengyu.extend(sample_ids_list[caiyangshu:])
```

```
        result = kmeans_with_top_samples(X_pca, n, temp, leibiehao)
    else:
        caiyangshu=a-a%10
        n=caiyangshu//10
        temp=sample_ids_list[:caiyangshu]
        shengyu.extend(sample_ids_list[caiyangshu:])
        result=kmeans_with_top_samples(X_pca, n, temp,leibiehao)
    leibiehao+=n
    zuizhong.extend(result)
    print(f"已完成{leibiehao}")
```

2.7 对于前面暂时保存的剩余样本,也使用"kmeans_with_top_samples"函数进行细分聚类,结果追加到"zuizhong"列表中。

```
result=kmeans_with_top_samples(X_pca, len(shengyu)//10,
shengyu,leibiehao)
print(f"已完成{leibiehao+len(shengyu)//10}")
zuizhong.extend(result)

print(len(zuizhong))
#print(len(shengyu_temp))
```

2.8 将所有细分聚类的结果汇总到一个列表"zuizhong"中,并按样本 ID 进行排序。

```
# 根据第一列（样本 ID）进行排序，获取排序后的索引
sorted_indices =np.array(zuizhong)[:, 0].argsort()

# 使用排序后的索引对数组进行排序

sorted_zuizhong =[]
for item in sorted_indices:
    sorted_zuizhong.append(zuizhong[int(item)])
```

2.9 最后,将"zuizhong"列表中的类别标签写入到"task1.out"文件中,每行一个整数,按样本 ID 排序。

```
# 打开文件以写入
with open('task1.out', 'w') as f:
    # 遍历排序后的数组的第二列（即样本类别）
    for category in np.array(sorted_zuizhong)[:, 1]:
        # 将类别（整数）写入文件，并在每个数字后添加一个换行符
        f.write(str(category) + '\n')

        # 现在 task1.out 文件应该包含了按样本 ID 顺序排列的样本类别，每
行一个整数
```

2.10 检查的时候发现题目是要类别号取值范围为[1,1000000]，而我们生成的是

[0,999999]因为跑一次要很久，所以就直接写了一个函数让他每个值加一，当然最后 task1_modified.out 直接重命名为 task1.out。

```
with open('task1.out', 'r') as file, open('task1_modified.out', 'w') as outfile:
    for line in file:
        num = int(line.strip())
        outfile.write(str(num + 1) + '\n')
```

问题二：垃圾邮箱地址检测

2.1 数据说明：

1. email*.txt，共 50 个文件，每个文件 50000000 个邮箱地址，这些都是垃圾邮件地址
2. check.txt，一个文件，内有 30000000 个邮箱地址，这些是待检测的邮件地址

2.2 任务介绍 ：

检查 check.txt 文件中的每一个邮箱地址，如果出现在 email*.txt 文件中，说明该邮件地址为垃圾邮件地 址，输出到文件 Task2.out，一个邮件地址占一行。

2.3 问题求解

1.方案介绍：
我们使用 python set()哈希的方式将 50，000，000 个查询利用 Python set()转化成集合以便快速查询。接着我们引入多进程程的方式对每一个 email*.txt 进行遍历，判断是否有邮件落在 set 中，如果有则将其写入文件。我们还做了对比实验，对比引入多进程的用时与未使用多进程的用时。
2.代码部分：
2.1 调用的库

```
import os
import time
from multiprocessing import Pool, cpu_count
```

2.2 定义 load_check_emails 函数。读取待检测邮件并加载成 python set(),set 的实现使用了哈希的方法，可以快速实现对元素的检索。

```
def load_check_emails():
    with open('check.txt', 'r') as file:
        #只使用前五位，可以表示 62*62*62*62*62*62 将近 570 亿个数
```

```
        check_emails = set(file.read().splitlines())
    return check_emails
```

2.3 定义 process_spam_file 函数。我们分别遍历每一个 email*.txt，逐行读取邮件地址。检查每个邮件地址是否在 check_emails 集合中，若有邮件地址在 set 中则记录下来，添加到 spam_emails_in_check 列表。

```
def process_spam_file(filename, check_emails):
    count=0
    spam_emails_in_check = []
    with open(f'emailaddress/{filename}', 'r') as file:
        for line in file:
            count+=1

            email = line.strip()
            if count%1000000==0:
                print(f"正在处理{filename},已完成{count}个")
                #print(email)

            if email in check_emails:
                print(1)
                spam_emails_in_check.append(email)
    return spam_emails_in_check
```

2.4 定义 write_spam_emails 函数。接收一个 spam_emails_in_check 列表作为输入，将列表中的所有邮件地址写入 Task2.out 文件

```
def write_spam_emails(spam_emails_in_check):
    with open('Task2.out', 'w') as file:
        for email in spam_emails_in_check:
            file.write(f"{email}\n")
```

2.5.1 方法一：单进程

```
def main():
    start_time = time.time()

    print("加载待检测的邮件地址...")
    check_emails = load_check_emails()

    filenames=[]
    for i in range(0,50):
        if i <=9:
            filenames.append(f"email0{i}.txt")
        else:
            filenames.append(f'email{i}.txt')
```

```
    results=[]
    for file in filenames:
        results.extend(process_spam_file(file, check_emails))


    print("将结果写入 Task2.out...")
    print(len(results))
    write_spam_emails(results)

    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"任务完成，总耗时: {elapsed_time:.2f} 秒.")
```

2.5.2 方法二：多进程

在处理大规模数据时（特别是当需要对数百万行数据进行操作时），单线程程序的性能会显得不足。我们需要检查一组垃圾邮件地址是否存在于大规模的邮件地址文件中，发现原始单线程程序效率低下于是我们通过引入多进程并行处理技术，发现显著提高了程序的性能。

```
def multiprocess_main():
    start_time = time.time()

    print("加载待检测的邮件地址...")
    check_emails = load_check_emails()

    filenames = [f'email{i if i > 9 else f"0{i}"}.txt' for i in range(50)]

    # 创建一个进程池，使用与 CPU 核心数相等的进程数
    with Pool(processes=cpu_count()) as pool:
        # 使用 Pool 的 map_async 或 starmap_async 方法并行处理文件
        # 因为我们需要传递两个参数给 process_spam_file，我们使用
starmap_async
        results_async = pool.starmap_async(process_spam_file,
[(filename, check_emails) for filename in filenames])

        # 获取并连接结果
        results = results_async.get()

        # 将结果展平，因为 starmap 返回的是一个列表的列表
        results = [email for sublist in results for email in sublist]

    print("将结果写入 Task2.out...")
    print(len(results))
    write_spam_emails(results)
```

```
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"任务完成，总耗时: {elapsed_time:.2f} 秒.")
if __name__ == "__main__":
    #main()
    multiprocess_main()
```

2.6 在搜索完成后，我们在 check.txt 中找出了 150 个垃圾邮件，其中单进程用时 639 秒，引入多进程后我们的用时是 537 秒。

```
没有多进程:
    150
    任务完成，总耗时: 639.50 秒.

多进程:
    150
    任务完成，总耗时: 537.74 秒.
```

问题三：分类应用，给出 300 类 DNA，进行分类预测

3.1 数据介绍

文件 Task3-分类数据.zip 提供以下三个数据集：
1. task.3.train.data.csv，训练数据特征文件，一行为一个样本，第一行为特征名称，第一列为训练样
本号，剩余 16 列为特征
2. task.3.train.label.csv，训练数据标签文件，一行为一个样本，第一行为特征名称，第一列为训练
样本号，第二列为样本标签即样本分类，其中样本号与 task.3.train.data.csv 的样本号相同的为同一
样本。
3. task.3.test.data.csv，测试数据特征文件，一行为一个样本，第一行为特征名称，第一列为测试样
本号，剩余 16 列为特征。
教师持有该测试数据的分类参考标准。

3.2 任务介绍

根据训练样本数据，进行训练模型，并对测试数据特征文件中每一个测试样本进行预测分类。

3.3 问题求解

1.方案介绍：首先通过读取和预处理数据,将原始的训练数据和标签转换为PyTorch 张量格式,并按照 8:2 的比例划分训练集和验证集（原本 7/3 发现 8/2 效果更好）。然后定义了一个简单的 MLP 模型,搭建了一个多层感知机网络。在训练过程中,使用交叉熵损失函数和 Adam 优化器进行 50 个 epoch 的迭代训练,同时记录训练集和验证集的损失及准确率。训练完成后,在验证集上评估模型的最终性能,并将训练过程中的损失和准确率可视化。最后,读取测试数据并使用训练好的模型进行预测,将预测结果输出到文件中。

2.代码部分：
2.1 代入必要库

```
import pandas as pd
import numpy as np
import torch
from torch.utils.data import TensorDataset, DataLoader
from torch import nn
from torch.optim import SGD,Adam
import matplotlib.pyplot as plt
```

2.2 数据预处理。使用 pd.read_csv() 函数读取训练数据和标签数据,并将其转换为PyTorch 张量。将数据划分为训练集和验证集,比例为 80%和 20%。创建 dataset 类和验证集的 DataLoader,用于批量加载数据。

```
# 使用 read_csv 函数读取 CSV 文件
data= pd.read_csv("task.3.train.data.csv").values[:,1:]
# 使用 read_csv 函数读取 CSV 文件
label = pd.read_csv("task.3.train.label.csv").values[:,-1].ravel()-1
# 转换为 PyTorch 张量
data_tensor = torch.tensor(data, dtype=torch.float32)
label_tensor = torch.tensor(label, dtype=torch.long)

# 假设我们想要 80%的数据作为训练集，20%作为验证集
train_size = int(0.8 * len(data_tensor))
test_size = len(data_tensor) - train_size

# 创建训练集和验证集
train_dataset, test_dataset =
torch.utils.data.random_split(TensorDataset(data_tensor,
```

```
label_tensor),

[train_size, test_size])

# 创建数据加载器
注意：我们小组原本在 16、32、64 中发现 64 效果最好既可以利用较大的并行计算优势,又不会占用过多的内存。
batch_size = 64  # 你可以根据需要调整这个值
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)
```

2.3 定义单层、隐藏层节点数为 128 的多层感知机模型。

```
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out

    # 初始化模型


input_size = data_tensor.shape[1]  # 特征的数量
hidden_size = 128  # 你可以根据需要调整这个值
num_classes = 300  # 分类的数量

model = MLP(input_size, hidden_size, num_classes)
```

2.4 定义交叉熵损失函数与 Adam 优化器，使用的初始学习率为 0.01。在训练过程中,我们将使用交叉熵损失函数来计算模型输出与真实标签之间的差异,并使用 Adam 优化器来更新模型参数,以最小化这个损失函数。

```
# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = Adam(model.parameters(), lr=0.001)  # 你可以根据需要调整学习率和动量
```

2.5 初始化用于记录损失和精度的列表。编写训练循环，我们进行了 50 个 epoch 的训练。首先，将模型设置为训练模式,并初始化训练集的损失和正确预测数。其次，遍历训练数据集,进行前向传播计算损失,反向传播计算梯度,并更新模型参数。同时统计训练集的损失和正确预测数。再者，计算训练集的平均损失和准确度,并记录到对应的列表中。接着，将模型设置为评估模式,并初始化测试集的损失和正确预测数。然后，不进行梯度计算地遍历测试数据集,进行前向传播并统计测试集的损失和正确预测数。之后，计算测试集的平均损失和准确度,并记录到对应的列表中。最后，打印当前 epoch 的训练和测试指标。通过这个训练循环,我们可以观察模型在训练集和测试集上的损失和准确度的变化情况,以评估模型的性能。最终训练完成后,可以使用这些记录的指标来分析模型的表现。

```
# 初始化用于记录损失和精度的列表
train_losses = []
test_losses = []
test_accuracies = []
num_epochs = 50
# 训练循环
for epoch in range(num_epochs):
    model.train()  # 设置为训练模式
    running_loss = 0.0
    corrects = 0
    total = 0

    for inputs, labels in train_loader:
        # 前向传播
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        # 反向传播和优化
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # 统计
        running_loss += loss.item() * inputs.size(0)
        _, preds = torch.max(outputs, 1)
        corrects += torch.sum(preds == labels.data)
        total += labels.size(0)

        # 计算训练集的损失和精度
    epoch_loss = running_loss / total
    epoch_acc = 100. * corrects.double() / total
    train_losses.append(epoch_loss)
```

```
        # 评估测试集
        model.eval()  # 设置为评估模式
        test_loss = 0.0
        correct = 0
        with torch.no_grad():  # 不需要计算梯度
            for inputs, labels in test_loader:
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                test_loss += loss.item() * inputs.size(0)
                _, preds = torch.max(outputs, 1)
                correct += torch.sum(preds == labels.data)

                # 计算测试集的损失和精度
        test_loss /= len(test_loader.dataset)
        test_acc = 100. * correct.double() / len(test_loader.dataset)
        test_losses.append(test_loss)
        test_accuracies.append(test_acc)

        print(
            f'Epoch {epoch + 1}/{num_epochs}, Loss: {epoch_loss:.4f}, Train
    Acc: {epoch_acc:.2f}%, Test Loss: {test_loss:.4f}, Test Acc:
    {test_acc:.2f}%')# 训练的进度 训练集平均损失值 训练集准确度 测试集平
    均损失值测试集准确度
```

经过五十轮训练后模型在测试集上的精度达到了 90%左右，在训练集上的精度达到了 95%左右。

```
Epoch 41/50, Loss: 0.2032, Train Acc: 94.02%, Test Loss: 0.6059, Test Acc: 89.48%
Epoch 42/50, Loss: 0.1837, Train Acc: 94.62%, Test Loss: 0.6414, Test Acc: 87.62%
Epoch 43/50, Loss: 0.2220, Train Acc: 93.63%, Test Loss: 0.6753, Test Acc: 87.88%
Epoch 44/50, Loss: 0.1959, Train Acc: 94.49%, Test Loss: 0.6197, Test Acc: 89.48%
Epoch 45/50, Loss: 0.1811, Train Acc: 94.74%, Test Loss: 0.9397, Test Acc: 85.49%
Epoch 46/50, Loss: 0.1869, Train Acc: 94.73%, Test Loss: 0.6823, Test Acc: 89.04%
Epoch 47/50, Loss: 0.1673, Train Acc: 95.15%, Test Loss: 0.6350, Test Acc: 89.72%
Epoch 48/50, Loss: 0.1680, Train Acc: 95.04%, Test Loss: 0.6280, Test Acc: 89.06%
Epoch 49/50, Loss: 0.2343, Train Acc: 93.50%, Test Loss: 0.5836, Test Acc: 90.68%
Epoch 50/50, Loss: 0.1532, Train Acc: 95.46%, Test Loss: 0.6337, Test Acc: 89.14%
```

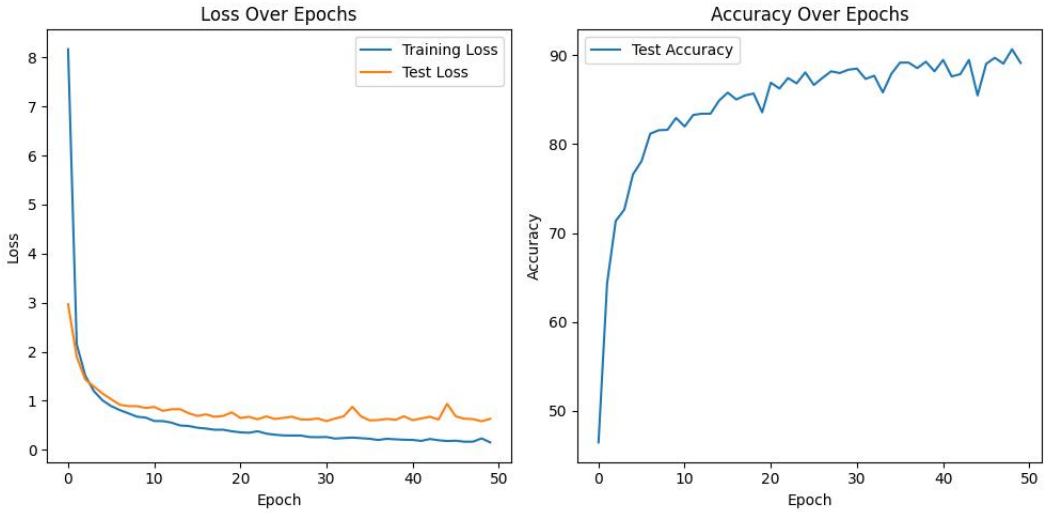
2.6 为了使结果增加可读性，我们使用 matplotlib.pyplot 库绘制训练损失、验证损失和验证准确率随 epoch 的变化情况。

```
from sy1 import *

# 可视化损失和精度
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
```

```
plt.plot(test_losses, label='Test Loss')
plt.title('Loss Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(test_accuracies, label='Test Accuracy')
plt.title('Accuracy Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.tight_layout()
plt.show()
```



2.7 因为模型训练效果还是不错的，所以可以训练最后的测试集啦。遍历测试数据,使用训练好的模型进行预测,并收集所有预测结果。将预测结果写入"Task3.out"文件中,每个样本一行。

```
from sy1 import *
# 使用 read_csv 函数读取 CSV 文件
test_data= pd.read_csv("task.3.test.data.csv").values[:,1:]
# 转换为 PyTorch 张量
data_tensor = torch.tensor(test_data, dtype=torch.float32)
predictions=[]
for item in data_tensor:
    out=model(item.unsqueeze(0))
    _, preds = torch.max(out, 1)
    #print(preds)
    predictions.append(int(preds))
# 打开文件以写入模式
with open('Task3.out', 'w') as f:
```

```
    for prediction in predictions:
        # 确保 prediction 是一个整数，并且在[1, 300]的范围内
        # 如果你的模型输出的是从 0 开始的索引，你可能需要+1 来匹配[1, 300]的范围
        prediction = max(1, min(prediction + 1, 300))  # 如果预测从 0 开始，则+1
        # 将预测结果写入文件，每个样本一行
        f.write(str(prediction) + '\n')
```