**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。

1. 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.")

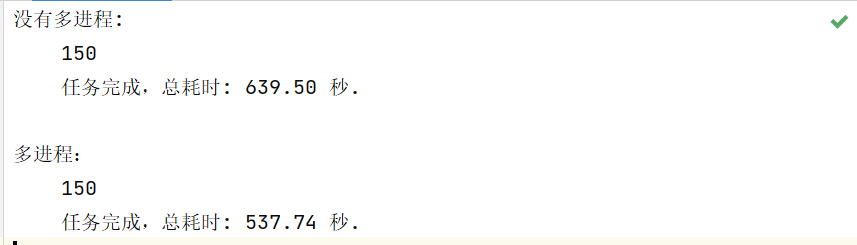
2.5.2方法二：多进程

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。



# 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)

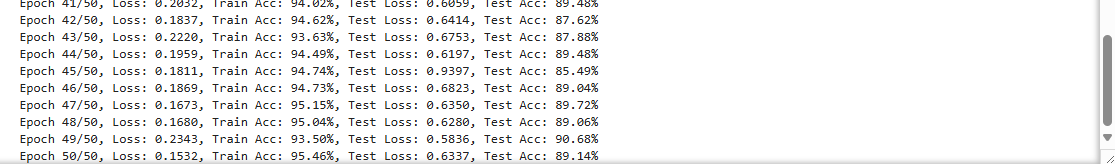
2.4定义交叉熵损失函数与Adam优化器，使用的初始学习率为0.01。在训练过程中,我们将使用交叉熵损失函数来计算模型输出与真实标签之间的差异,并使用 Adam 优化器来更新模型参数,以最小化这个损失函数。

*# 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。

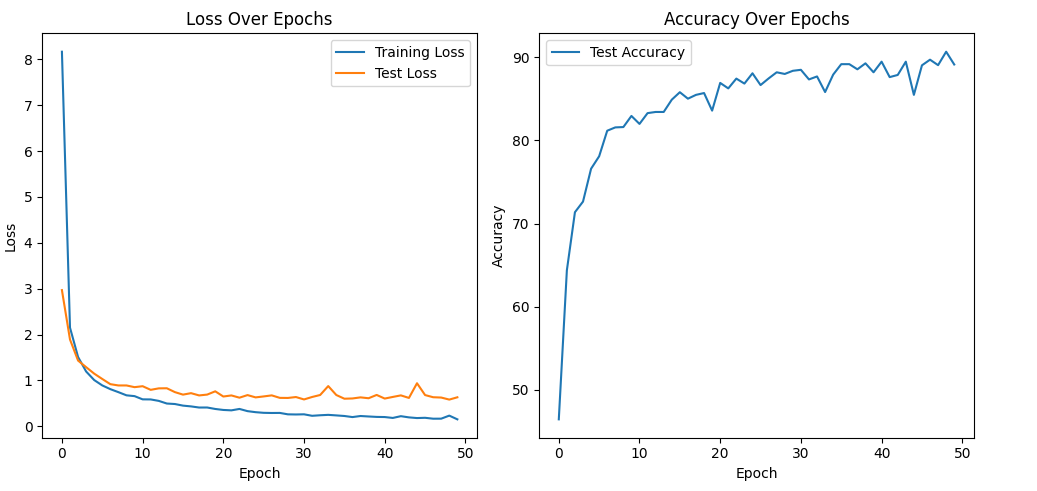
*# 初始化用于记录损失和精度的列表*train\_losses = []  
test\_losses = []  
test\_accuracies = []  
num\_epochs = 50  
*# 训练循环*for epoch in range(num\_epochs):  
 model.train() *# 设置为训练模式* running\_loss = 0.0  
 corrects = 0  
 total = 0  
  
 for inputs, labels in train\_loader:  
 *# 前向传播* outputs = model(inputs)  
 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。



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')