南開大學
Nankai University

计算机学院

机器学习实验报告

基于 KNN 的手写数字识别实验

姓名：王旭

学号：2312166

专业：计算机科学与技术

2025 年 9 月 27 日

目录

1 实验名称

基于 kNN 的手写数字识别实验

2 实验目的

1. 实现 k 近邻（k-Nearest Neighbor, kNN）分类器；
2. 掌握留一法（Leave-One-Out, LOO）交叉验证流程；
3. 对比自实现结果与 Weka 内置 kNN 精度；
4. 学会用图表展示实验过程与结论。

3 数据集下载与快速检查

3.1 下载地址（官网可访问）

- 官网: http://archive.ics.uci.edu/ml/datasets/Semeion+Handwritten+Digit

3.2 数据集说明

解压后得到 semeion.data.txt（或 semeion.data），纯文本，空格分隔：

- 前 256 列 = 16×16 手写图像展开成 0/1 像素；
- 后 10 列 = 独热编码标签，第 i 列为 1 表示数字 i-1（如第 1 列为 1 表示数字 0）。

3.3 快速检查（Python 3 代码，实验前运行）

快速检查代码

```python
import numpy as np
import matplotlib.pyplot as plt #注意依赖库的安装

raw = np.loadtxt('semeion.data.txt')
X, y = raw[:, :256], raw[:, 256:]
y = np.argmax(y, axis=1)          # 独热 → 0~9 整数
print('样本数:', len(X), '像素数:', X.shape[1])

# 随机画 6 张图
for i in range(6):
    plt.subplot(2,3,i+1)
    plt.imshow(X[i].reshape(16,16), cmap='gray')
    plt.title(y[i]); plt.axis('off')
plt.tight_layout(); plt.show()
```

运行结果:

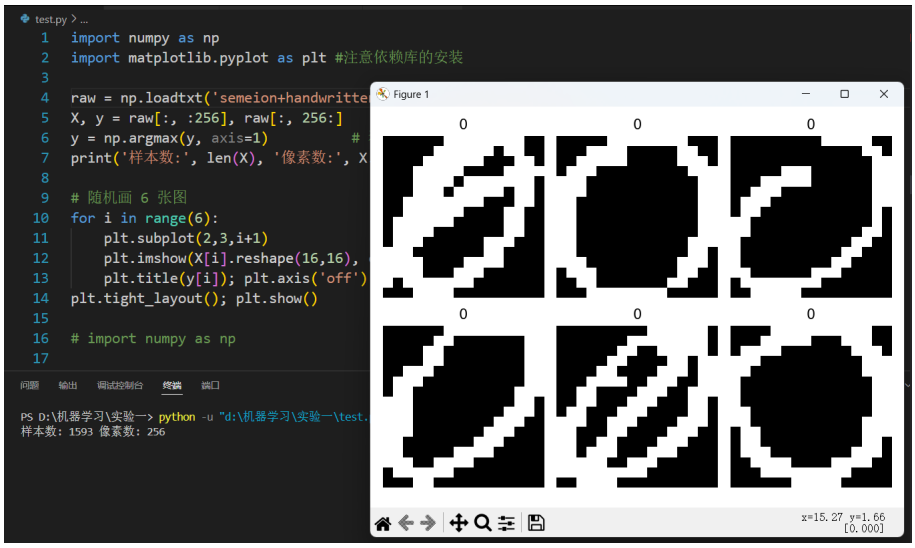


图 3.1: 快速检查代码运行结果

可以看到运行后看到 6 张清晰数字'0' 的手写图像，文件路径及环境配置均正确。

4 基础任务：自写 kNN + LOO

4.1 算法思路

因为我们要做的整体目标是 KNN+LOO，我们逐步分析，将目标分步来解决。

我们首先要明确的是数据集的格式以及含义，这样我们才能在此基础上对数据做出正确的计算与处理。经过我们前面" 快速检查代码" 部分的运行结果我们可以看出，数据样本数为 1593 即 1593 行，每一行为一个样本，像素数为 256，即对于每一行来说，前 256 列一起组成了一个手写数字图像。而后 10 列表示的是该数字表示的是数字几，即" 标签"。由于手写数字的范围是 09, 因此这 10 列表示的方式是采用独热编码的方式，元素为 1 的索引代表的就是数字图像对应的阿拉伯数字。比如第 [0] 列为 1，则代表的就是数字 0。

因此我们在代码中首先对数据集进行读取。

对数据集的读取及简要处理

```
raw = np.loadtxt('semeion+handwritten+digital/semeion.data.txt')
X, y = raw[:, :256], raw[:, 256:]
y = np.argmax(y, axis=1)          # 独热 → 0~9 整数
```

对 X，保留每一行的前 256 列表示手写数字图像样本的特征向量作为 X。

对 y，将每一行的后 10 列的独热编码形式 (10 维向量) 转为 1 维向量即一个整数，代表标签。更加直接。

以上明确了数据集的格式及含义，我们下面来对数据集进行 KNN 的处理。

4.1.1 KNN 处理思路

KNN 的理论层面的过程是，对于一个测试样本 x_test，要对其进行 KNN 分类，我们要首先计算测试样本 x_test 与现有训练样本 X_train 中的所有样本 x_train 的距离，然后从其中选出距离最小的 K 个训练样本 x_train_i，然后根据这 K 个训练样本 x_train_i 的标签，选出这些标签中占最多数量的标签作为对测试样本 x_test 标签的分类预测 y_{pred}，这样就得到了预测结果。

对应到本次的数据集中，我们数据集中的所有样本都是 256 维向量，因此我们要计算的样本间的距离就是欧氏距离 (向量之间的距离)$d_{\text{欧式距离}}(\mathbf{x_1}, \mathbf{x_2}) = \sqrt{\sum_{i=1}^{N}(x_i - y_i)^2}$，向量各维度元素差的平方求和，最后再开根号。对于这个求距离的过程，我思考了两种方式:

1. 单写一个函数 Oudistance(x_1,x_2)，来求两个样本间的距离。然后再利用 for 循环 1593 次对所有样本进行循环调用 Oudistance(x_1,x_2)，以求出测试样本与所有样本间的距离。

2. 直接利用 numpy 对向量的处理方式，让整体样本集 X_{train} 直接减去测试样本 x_{test} 得到一个向量，然后直接对这个向量平方求和再开根号，不用 for 循环 1593 次了。

我最初想到的是第一种方式，因为这个方法比较直接，也符合 C++ 思维方式。但是当我通过这种方式来实现之后，(结果肯定是正确无误的)，发现运行速度非常慢，平均对于一种 K 值，需要运行将近 20s 才能进行一次完整的 LOO 验证。因此我开始思考有没有更高效快速的方式，于是采用了这种 Numpy 对向量的高效处理方式，经过改进后发现运行速度大大提高，平均一种 K 值运行 6s 即可完成。这种方法更加符合 Python 的思维方式，也让我认识到我对 python 的使用还需要进一步提高。在后面的“具体实现代码”部分，我会贴出两种方式的对比。

在求出测试样本与所有训练样本的距离后，再对该距离数组进行从小到大排序，选出前 k 个样本的索引，再通过索引来得到这 k 个最近样本的标签 y_i，然后通过找到最多出现的标签 (采用桶排序的方式) 来确定测试样本的标签 y_{pred}。特别注意的是，我们还需要进行一下平票处理。当有多个相同标签出现次数最多时，从这些标签中随机选择一个标签作为 y_{pred}。

以上就是 KNN 的具体实现思路。

4.1.2 LOO 处理思路

对 LOO 的处理思路其实就比较简单了。因为 LOO 从理论方面，就是对数据集中的所有样本，每次选择一个样本作为测试样本，剩下的样本作为训练样本，求出一个预测结果后与实际结果进行对比求出一个预测正确率，经过所有样本都当过一次测试样本后，以求得的平均正确率作为模型正确率。

对于本数据集来说，就是用 for 循环对数据集 X 做 1593 次循环处理，每次处理选择 X[i] 作为测试样本 x_{test}，剩下的 1592 个样本作为整体训练集 X_{train}。然后调用 KNN 函数求得对 X[i] 的预测结果标签，与实际 X[i] 的标签 y[i] 对比，相同则预测正确数 acc+1，否则不加，这样一轮循环结束。当 1593 次循环完成后，求得最终预测正确率为 acc/N，即为 LOO 验证结果正确率。

4.2 具体实现代码

4.2.1 对整个数据集的 KNN+LOO 处理 (1593 个样本，不分训练集与测试集)

对整个数据集 (1593 个样本) 的 KNN+LOO 处理

```
import numpy as np
import time
## import ... 安装所需要的依赖库
```

```
4
5  # def Oudistance(x1,x2):
6  #     return np.sqrt(np.sum((x1 - x2) ** 2))
7
8  def KNN(X_train,Y_train,x_test,k):
9      # distances = [Oudistance(x_train,x_test) for x_train in X_train]
10     distances=np.sqrt(np.sum((X_train-x_test)**2,axis=1))
11     nearest_k_indexs=np.argsort(distances)[:k]
12     nearest_k_labels = Y_train[nearest_k_indexs]
13     counts = np.bincount(nearest_k_labels)
14     maxnum = np.max(counts)
15     if( np.sum(counts==maxnum) >1 ):
16         maxindexs = np.where(counts==maxnum)[0]
17         vote = np.random.choice(maxindexs)
18     else:
19         vote = np.argmax(counts)
20     return vote
21
22 def loo_eval(X, y, k):
23     print(f"开始在整个数据集semeion.data上进行LOO评估,k={k}")
24     #计算时间
25     time_start = time.time()
26     N=len(X)
27     correct=0
28     for i in range(N):
29         X_train = np.delete(X,i,0)
30         Y_train = np.delete(y,i)
31         x_test = X[i]
32         y_true = y[i]
33         y_pred = KNN(X_train,Y_train,x_test,k)
34         if y_pred == y_true:
35             correct += 1
36     print("LOO评估完成,耗时: %.2f 秒" % (time.time() - time_start))
37     acc = correct / N
38     return acc
39
40
41 # 主流程
42 raw = np.loadtxt('semeion+handwritten+digital/semeion.data.txt')
43 X, y = raw[:, :256], np.argmax(raw[:, 256:], 1)
44
45 for k in [1, 3, 5]:
46     acc = loo_eval(X, y, k)
47     print(f'k={k}  LOO 准确率 = {acc:.4f}')
```

以上代码就是基于我在“算法思路”部分所实现的 KNN+LOO 代码，其中目前的 KNN 处理方式是采取的“KNN 思路”中提到的第二种思路，采用 Numpy 对向量处理的方式进行求样本间欧氏距离，即

第二种方式处理样本间距离 (Numpy 高效)

```
distances=np.sqrt(np.sum((X_train-x_test)**2,axis=1))
```

而提到的第一种方式则是代码中注释掉的部分，这是我在改进前我的实现想法。我手动写了一个处理两个样本间距离的函数 Oudistance(x1,x2)，并在 KNN 函数中通过 for 循环的方式循环调用 1593 次 Oudistance(x1,x2) 函数，这样的实现是正确的，但是效率是很慢的。

第一种方式处理样本间距离 (效率较低)

```
def Oudistance(x1,x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))

distances = [Oudistance(x_train,x_test) for x_train in X_train]
```

4.2.2 划分数据集后的 KNN+LOO 处理

划分数据集后的 KNN+LOO 处理

```
import numpy as np

def loo_eval(X, y, k):
    print(f"开始在训练集semeion_train上进行LOO评估,k={k}")
    N=len(X)
    correct=0
    for i in range(N):
        X_train = np.delete(X,i,0)
        Y_train = np.delete(y,i)
        x_test = X[i]
        y_true = y[i]
        y_pred = KNN(X_train,Y_train,x_test,k)
        if y_pred == y_true:
            correct += 1
    print("LOO评估完成")
    acc = correct / N
    return acc

def test_eval(X, y, k):
    print(f"开始在测试集semeion_test上进行评估,k={k}")
    N=len(X)
    correct=0
    for i in range(N):
        x_test = X[i]
        y_true = y[i]
        y_pred = KNN(X,y,x_test,k)
        if y_pred == y_true:
            correct += 1
    print("测试集评估完成")
    acc = correct / N
```

```
    return acc

# 主流程
raw = np.loadtxt('semeion+handwritten+digital/semeion.data.txt')
X, y = raw[:, :256], np.argmax(raw[:, 256:], 1)

train_raw = np.loadtxt('semeion+handwritten+digital/semeion_train.txt')
X_train,Y_train = train_raw[:, :256], np.argmax(train_raw[:, 256:], 1)

test_raw = np.loadtxt('semeion+handwritten+digital/semeion_test.txt')
X_test,Y_test = test_raw[:, :256], np.argmax(test_raw[:, 256:], 1)

print(f'训练集样本数: {len(X_train)}, 测试集样本数: {len(X_test)}')
print('----------------------------------------------------------------------')

for k in [1, 3, 5]:
    acc = loo_eval(X_train, Y_train, k)
    print(f'k={k}  LOO 准确率 = {acc:.4f}')

print('----------------------------------------------------------------------')

for k in [1, 3, 5]:
    acc = test_eval(X_test, Y_test, k)
    print(f'k={k}  测试集 准确率 = {acc:.4f}')
```

代码详见: machine-learning/实验一

划分数据集为训练集与测试集后，我将 KNN+LOO 在训练集上进行了运行验证，在测试集上进行了 KNN 测试。所以 KNN 函数与原来的对整个数据集的函数是一样的，我在上面的代码中略去。LOO 函数只是修改了调试信息，test_eval 函数与 LOO 函数是类似的，只不过不需要进行 LOO 除去一个样本的操作。

5 运行结果

5.1 对整个数据集的 KNN+LOO 处理 (1593 个样本，不分训练集与测试集)

5.1.1 方式一 (未优化版)

```
PS D:\机器学习\实验一> python -u "d:\机器学习\实验一\KNN+LOO_1593.py"
开始在整个数据集semeion.data上进行LOO评估,k=1
LOO评估完成,耗时: 21.00 秒
k=1  LOO 准确率 = 0.9159
开始在整个数据集semeion.data上进行LOO评估,k=3
LOO评估完成,耗时: 18.00 秒
k=3  LOO 准确率 = 0.9153
开始在整个数据集semeion.data上进行LOO评估,k=5
LOO评估完成,耗时: 17.34 秒
k=5  LOO 准确率 = 0.9140
PS D:\机器学习\实验一>
```

图 5.2: 未优化版

5.1.2 方式二 (Numpy 优化版)

```
PS D:\机器学习\实验一> python -u "d:\机器学习\实验一\KNN+LOO_1593.py"
开始在整个数据集semeion.data上进行LOO评估,k=1
LOO评估完成,耗时: 5.25 秒
k=1  LOO 准确率 = 0.9159
开始在整个数据集semeion.data上进行LOO评估,k=3
LOO评估完成,耗时: 5.22 秒
k=3  LOO 准确率 = 0.9134
开始在整个数据集semeion.data上进行LOO评估,k=5
LOO评估完成,耗时: 5.30 秒
k=5  LOO 准确率 = 0.9134
PS D:\机器学习\实验一>
```

图 5.3: Numpy 优化版

5.1.3 结果对比

K 值	耗时 (未优化版)	耗时 (Numpy 优化版)	LOO 准确率 (未优化版)	LOO 准确率 (优化版)
1	21.00	5.25	0.9159	0.9159
3	18.00	5.22	0.9153	0.9134
5	17.34	5.30	0.9140	0.9134

表 1: 两个版本运行结果对比

5.2 划分数据集后的 KNN+LOO 处理

```
PS D:\机器学习\实验一> python -u "d:\机器学习\实验一\KNN+LOO.py"
训练集样本数: 1293, 测试集样本数: 300
----------------------------------------------------------------------
开始在训练集semeion_train上进行LOO评估,k=1
LOO评估完成
k=1  LOO 准确率 = 0.9118
开始在训练集semeion_train上进行LOO评估,k=3
LOO评估完成
k=3  LOO 准确率 = 0.9080
开始在训练集semeion_train上进行LOO评估,k=5
LOO评估完成
k=5  LOO 准确率 = 0.9080
----------------------------------------------------------------------
开始在测试集semeion_test上进行评估,k=1
测试集评估完成
k=1  测试集 准确率 = 1.0000
开始在测试集semeion_test上进行评估,k=3
测试集评估完成
k=3  测试集 准确率 = 0.9367
开始在测试集semeion_test上进行评估,k=5
测试集评估完成
k=5  测试集 准确率 = 0.8967
PS D:\机器学习\实验一>
```

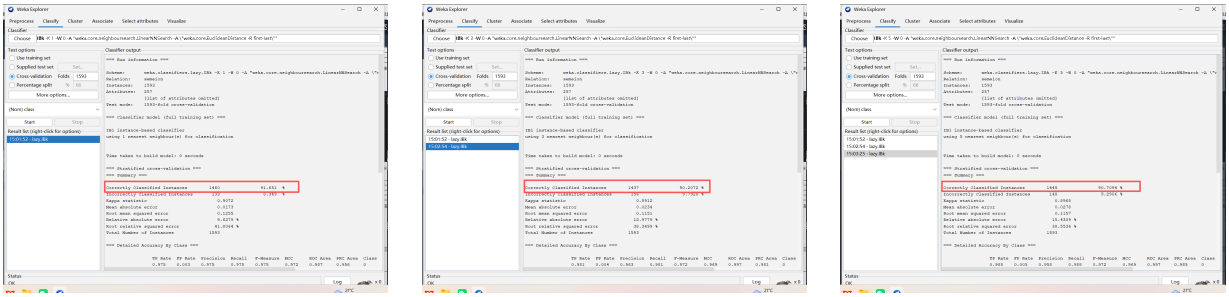
图 5.4: 划分数据集后的运行结果

K 值	训练集 LOO 准确率	测试集准确率
1	0.9118	1.0000
3	0.9080	0.9367
5	0.9080	0.8967

表 2: 划分数据集后结果

6 Weka 对比 (中级要求)

在使用 Weka 过程中，由于在选择 Open file 时，无法选择.txt 文件，因此我们借助 AI 编写 python 脚本将 semeion.data.txt 转为了 (Arff data files (*.arff)) 格式的 semeion.arff 文件，使得 Weka 能够处理。对于 K=1,3,5 情况下，结果如下:



(a) k=1 运行结果 (b) k=3 运行结果 (c) k=5 运行结果

图 6.5: Weka 在三种 k 值下的运行结果

K 值	自写 LOO 精度	Weka LOO 精度	差异（绝对值）	是否 >0.5%
1	0.9159	0.91651	0.061%	否
3	0.9134	0.902072	1.1328%	是
5	0.9134	0.907094	0.6306%	是

可以看到当 K=3,5 时差异值 >0.5%。可能的原因是:

Weka 中 KNN 相关算法（如 IBk）默认可能使用曼哈顿距离（L_1 距离），或自动对输入特征做了标准化（如 Z-Score 标准化、Min-Max 归一化），导致相同样本的“近邻”判断结果不同，最终 LOO（留一法）精度差异较大。