

**课程实验报告**

**题目： 机器学习实验报告**

**课程名称： 机器学习**

**专业班级： CS1701**

**学 号： U201714501**

**姓 名： 熊逸钦**

**指导教师： 李玉华**

**报告日期： 2020年5月12日**

**计算机科学与技术学院**

# 1实验一 KNN算法实现

## 1.1实验目的与要求

（1）了解KNN算法原理；

（2）熟悉Python语言用法；

（3）了解MINIST数据集的使用；

（4）自己动手实现KNN算法。

## 1.2实验内容

利用Python语言自己动手实现KNN算法，数据集使用MINIST训练集。

KNN结果展示方式：

* 输入若干测试图片，输出对应每张图片的k近邻；
* 绘制knn算法的训练misclassification rate曲线，并做出分析

## 1.3实验方案

（1）读入数据集

本次任务中使用的是MINIST数据集，其训练集包含60000个条目，测试集包含10000个条目，而且其使用了特殊的二进制文件格式来存放数据：对于labels标签而言，使用idx1-ubyte格式；对于images图片而言，使用idx3-ubyte格式。下面以训练集为例对这两个格式进行简单介绍：

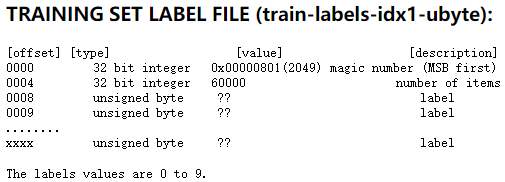
1. idx1-ubyte格式

图1-1 idx1-ubyte格式

上图为idx1-ubyte文件的格式，offset描述其在文件中偏移字节数，需要注意的是文件中的整数都以MSB（最高有效位）优先的格式进行存储。文件开头的两个32位整数分别是魔数和数据集的条目数。从offset=0008开始的每一个字节都代表着一个标签，表示对应的图片所描述的实际数字。

根据上面所描述的文件格式，可以利用python的struct包提供的unpack\_from函数对数据进行读取，对标头两个整数进行翻转的操作只需要在函数的第一个参数使用”>ii”进行大端格式转换即可。

具体代码实现如下：

def idx1\_to\_labels(idx1\_filename):

# 读取二进制数据

bin\_data = open(idx1\_filename, 'rb').read()

# 解析文件头信息，依次为魔数和数据集的条目数

offset = 0

fmt\_header = '>ii'

magic\_number, num\_images = struct.unpack\_from(fmt\_header, bin\_data, offset)

# 解析数据集

offset += struct.calcsize(fmt\_header)

fmt\_image = '>B'

labels = np.empty(num\_images)

for i in range(num\_images):

labels[i] = struct.unpack\_from(fmt\_image, bin\_data, offset)[0]

offset += struct.calcsize(fmt\_image)

return labels

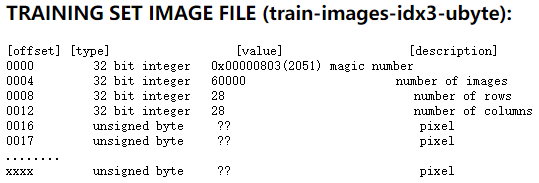
2. idx3-ubyte格式

图1-2 idx3-ubyte格式

上图为idx3-ubyte文件的格式，offset描述其在文件中偏移字节数，需要注意的是文件中的整数同样也都以MSB（最高有效位）优先的格式进行存储。文件开头的四个32位整数分别是魔数、数据集的条目数、图片像素行数、图片像素列数。从offset=0008开始的每一个字节都代表着一个像素，在这个例子中，每28\*28个像素就表示了一张图片。

根据上面所描述的文件格式，可以利用python的struct包提供的unpack\_from函数对数据进行读取，对标头四个整数进行翻转的操作只需要在函数的第一个参数使用”>iiii”进行大端格式转换即可。

具体代码实现如下：

def idx3\_to\_images(idx3\_filename):

# 读取二进制数据

bin\_data = open(idx3\_filename, 'rb').read()

# 解析文件头信息，魔数、条目数、图片像素行数、图片像素列数

offset = 0

fmt\_header = '>iiii'

magic\_number, num\_images, num\_rows, num\_cols = struct.unpack\_from(fmt\_header, bin\_data, offset)

# 解析数据集

image\_size = num\_rows \* num\_cols

offset += struct.calcsize(fmt\_header)

fmt\_image = '>' + str(image\_size) + 'B'

images = np.empty((num\_images, num\_rows, num\_cols))

for i in range(num\_images):

images[i] = np.array(

struct.unpack\_from(fmt\_image, bin\_data, offset)

).reshape((num\_rows, num\_cols))

offset += struct.calcsize(fmt\_image)

return images

（2）KNN算法的具体实现

KNN算法的最主要思想就是找出与给定数据点距离最近的K个邻居，将给定的数据点的标签设置为这K个邻居中出现次数最多的一个标签。

给出针对手写数字识别问题的KNN算法的流程图，如下图1-3所示：

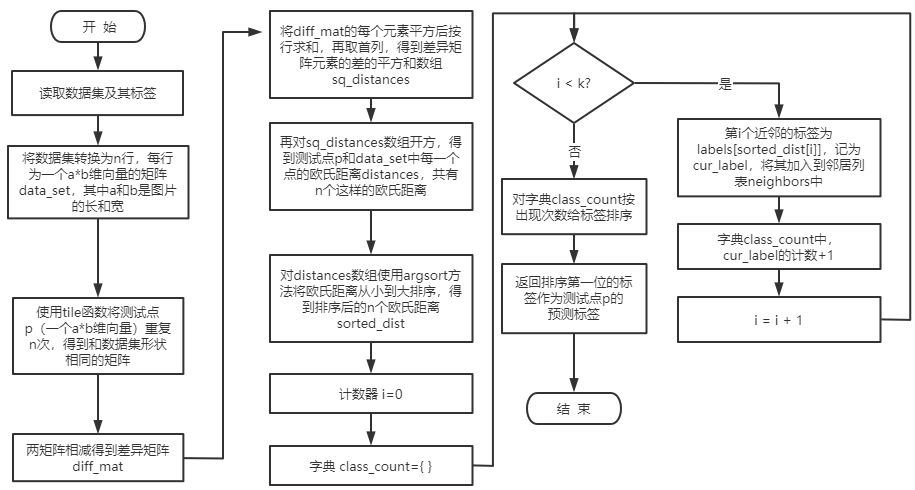


图1-3 流程图

具体的实现思路是：由于图片的分辨率是28\*28的，因此一张图片的像素总数为784，这样就可以使用一个784维的向量来描述一张图片，假设训练集有60000张图片，那么测试集可以用一个形状为(60000, 784)的矩阵来描述，对于每一个测试图片，使用numpy的tile函数让其重复60000次，也构成一个形状为(60000, 784)的矩阵，对这两个矩阵做减法得到差异矩阵，再将差异矩阵中的每一个数平方后按行求和再开方，即先求平方和再开方（欧氏距离）。就可以得到该测试数据点与训练集的60000个数据点的距离。将这个距离从小到大排序，排序的结果为数据集中数据点的序号形成的列表。统计前K个邻居的标签出现的数目，最后将出现数目最大的标签作为测试数据点的标签，完成分类。

KNN过程的代码如下：

def knn(in\_x, data\_set, labels, k):

# 训练样本个数

data\_set\_size = data\_set.shape[0]

# tile让in\_x重复data\_set\_size次，再与data\_set相减得到diff\_mat

diff\_mat = np.tile(in\_x, (data\_set\_size, 1)) - data\_set

# 将diff\_mat平方后横向求和得到sq\_distances（即Σ(ai-bi)^2）

#再开方得到in\_x和data\_set的各数据点的欧氏距离

sq\_diff\_mat = diff\_mat \*\* 2

sq\_distances = sq\_diff\_mat.sum(axis=1)

distances = sq\_distances \*\* 0.5

# 将距离从小到大排序，排序结果为数据集中数据点的序号形成的列表

sorted\_dist = distances.argsort()

# 字典存储不同标签出现的次数

class\_count = {}

# 记录k个邻居

neighbors = []

for i in range(k):

# 获取当前标签并加入到邻居列表中

cur\_label = labels[sorted\_dist[i]]

neighbors.append(int(cur\_label))

# 统计标签出现次数

class\_count[cur\_label] = class\_count.get(cur\_label, 0) + 1

# 按‘出现次数’对标签字典进行排序，出现次数多的排前面

# operator.itemgetter选择‘出现次数’为排序依据

sorted\_class\_count = sorted(class\_count.items(), key=operator.itemgetter(1),

reverse=True)

return sorted\_class\_count[0][0], neighbors

（3）选取合适的K值

在MINIST官方网站中给出了测试集的组成，包括5000条NIST的训练集中的数据和5000条NIST的测试集中的数据，这里将取值在1到20之间（闭区间）的每个K进行一次KNN过程，训练集规模为60000，测试集规模为1500，对每个K的取值的分类错误率进行统计，并绘制错误率曲线，最终选取使得分类错误率最小的K值作为最终的K值。

核心代码如下：

# 参数设置，包括k的范围设置以及测试的数据集大小设置

conf\_k\_range = range(1, 21)

conf\_test\_begin = 0

conf\_test\_size = 1000

# 记录绘图的横纵坐标

mis\_classification\_rates = []

# 下面对各个k都进行一次测试

for k in conf\_k\_range:

# 错误计数

err\_count = 0.0

# 对测试集中的各个数据进行测试

for i in range(conf\_test\_begin, conf\_test\_size):

classifier\_result, neighbors = knn(test\_images[i],

train\_images, train\_labels, k)

print('(预测:%d 答案:%d)' %

(classifier\_result, test\_labels[i]), end='\t')

print('K近邻:', end='')

print(neighbors)

if classifier\_result != test\_labels[i]: err\_count += 1.0

# 计算误差率

err\_rate = err\_count / float(conf\_test\_size)

print('K=%d err\_count:%d err\_rate:%f' % (k, err\_count, err\_rate))

mis\_classification\_rates.append(err\_rate)

# 绘图

plt.plot(conf\_k\_range, mis\_classification\_rates)

plt.show()

# 取使得误差率最小的k值

print('选择K=%d' %

conf\_k\_range[

mis\_classification\_rates.index(min(mis\_classification\_rates))])

## 1.4实验结果

（1）数据集介绍

本次实验使用的是MINIST手写数字数据集，其中训练集包含60000个示例，而测试集为10000个示例。在MINIST数据集中所提供的数字图片都具有规格化的尺寸（28\*28的分辨率），并且数字居中出现在图像中。数据集的图片和标签使用特殊的IDX格式进行存储，对于文件格式的描述请见报告第1.3节第（1）部分。

（2）实验结果与分析

①选取合适的K值

对K从1到20（闭区间）分别进行训练集和测试集规模为1000的测试（由于笔记本电脑的性能有限，跑完整的10000个测试点需要花费大量的时间，因此适当简化为1000个测试点），并统计得到错误率曲线，这部分代码详见报告1.3节第（3）部分。

得到的结果如下图1-4和图1-5所示：

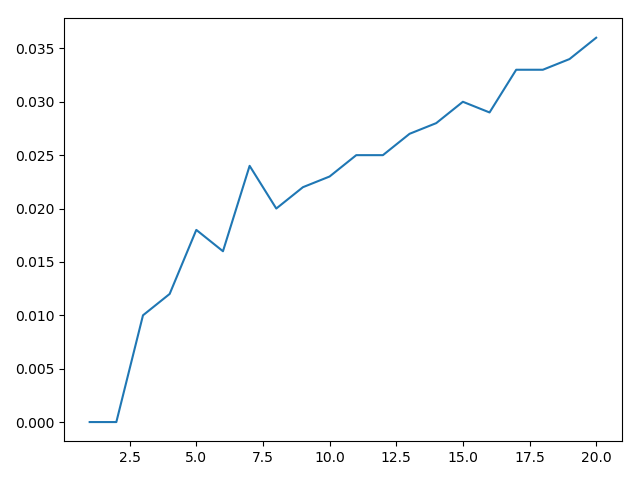


图1-4 训练集错误率曲线

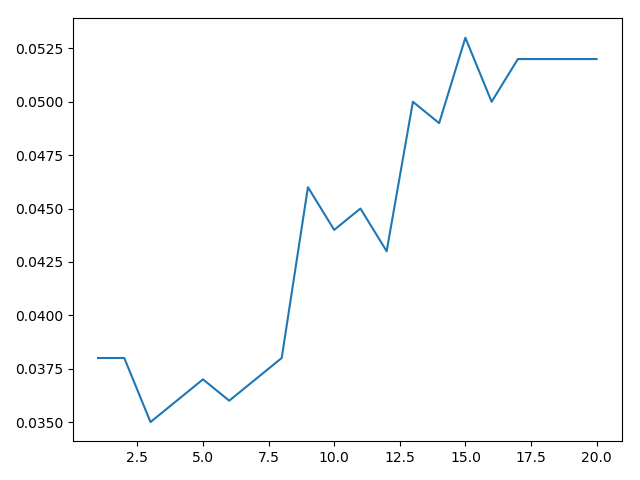


图1-5 测试集错误率曲线

根据图1-4和图1-5的内容，将曲线整合到同一张图中，如下图1-6所示：

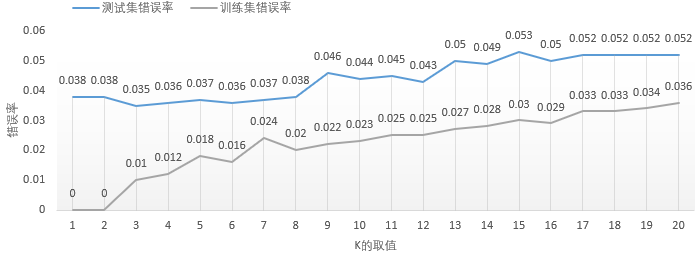


图1-6 错误率曲线整合结果

根据上图的错误率曲线，可以看到在K值取3时，测试集错误率为最低的0.035，即3.5%，而且当K取1至8时，测试集错误率曲线均保持在一个比较低的水平，而当K的取值大于8时，错误率提升比较明显，尤其在K取15时达到最高的0.053，即5.3%。综合分析可见，对于给定的这个数据集，K的较好取值为3。

②使用较好的K值进行测试

接下来选取①中得到的K值，即取K=3，并对完整的规模为10000的测试集在规模为60000的训练集上进行测试，输出部分测试结果的展示如图1-7：

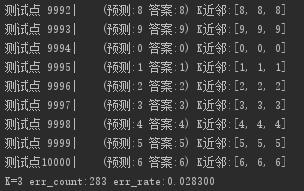


图1-7 部分测试结果

如上图所示，程序给出测试点的预测结果，并与测试集标签中存放的正确答案进行对比，并给出对应的K个近邻的情况。

可以看到，测试过程中预测正确率很高，在10000个测试点中，只有283个出现了错误，错误率为2.83%，即预测正确率为97.17%。但还是有预测错误的情况出现，如图1-8所示：

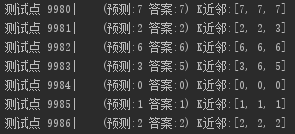


图1-8 预测错误情况示例

在图1-8中，以第9983个测试点为例，预测值为“3”，正确答案为“5”，测试点的K个近邻依次为[3, 6, 5]，由于“3”、“6”和“5”的出现次数均为1，按就近原则选取了“3”作为最终的预测结果。再以第9981个测试点为例，预测值为“2”，正确答案为“2”，测试点的K个近邻依次为[2, 2, 3]，由于“2”的出现次数为2次，“3”的出现次数仅为1次，因此根据KNN算法，选取“2”作为最终的预测结果。

除第9983个测试点之外，在图中的其他例子中均能正确识别输入的手写数字，这体现了KNN算法简单高效的特点。

不过KNN算法的缺点在实验中也得以体现，就是它的计算量太大，对于每一个等待分类的测试点，都需要计算它到全体已知的样本的距离，才能得到前K个近邻。KNN算法的复杂度为O(n·d)，其中n为样本规模，d为维度，在本次实验中训练集的n为60000，d为28\*28=784，对于每个测试点都要和60000个数据点进行欧氏距离的计算，所需的计算资源是比较大的。