華中科技大學

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

TRAINING SET LABEL FILE (train-labels-idx1-ubyte):

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
.....			
xxxx	unsigned byte	??	label

The labels values are 0 to 9.

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

TRAINING SET IMAGE FILE (train-images-idx3-ubyte):

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	60000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
.....			
xxxx	unsigned byte	??	pixel

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：

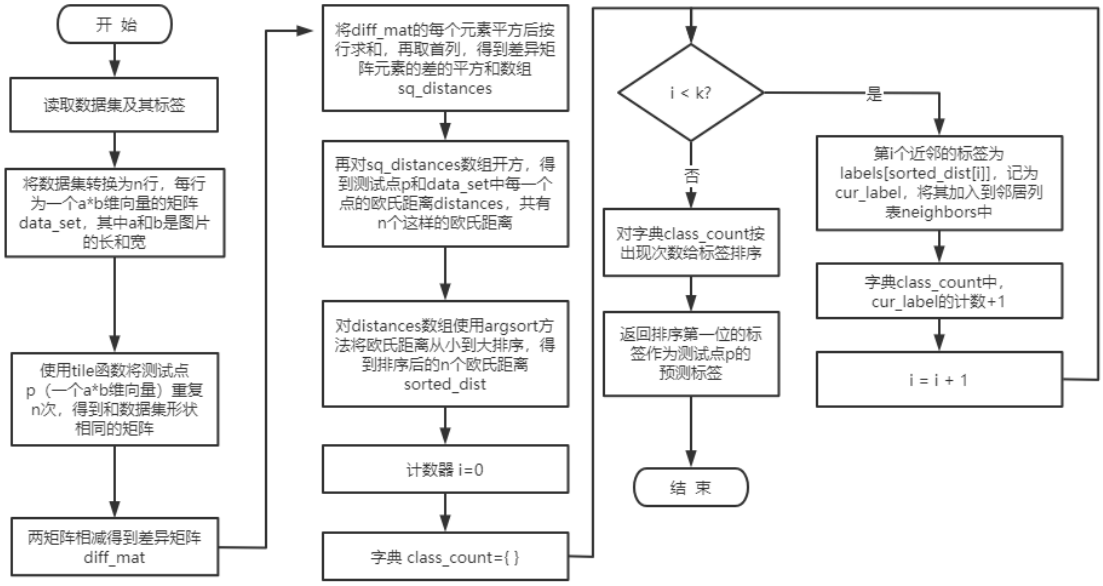

图 1-3 流程图

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：

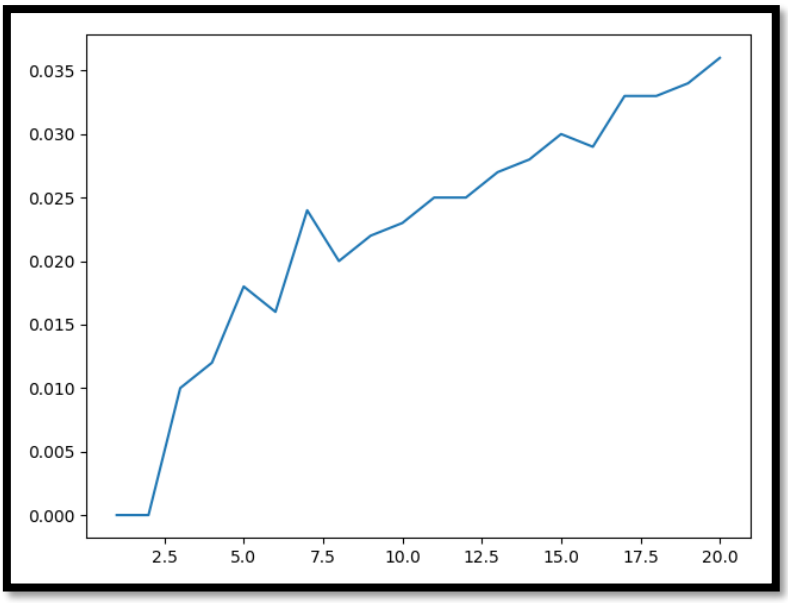

图 1-4 训练集错误率曲线

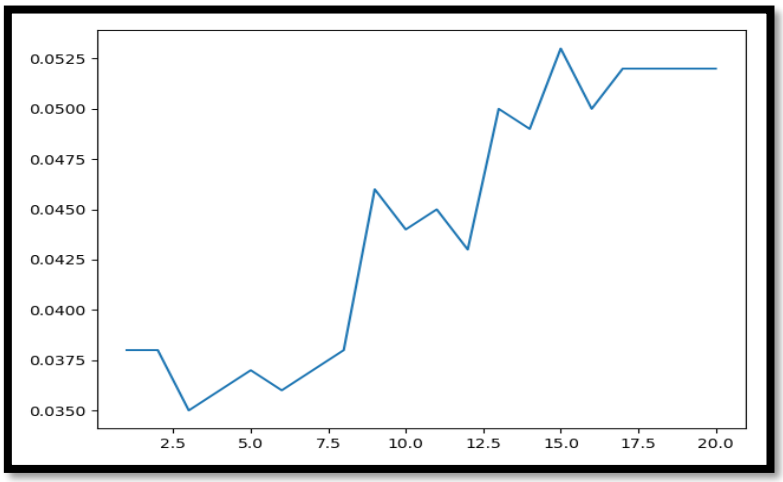

图 1-5 测试集错误率曲线

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：

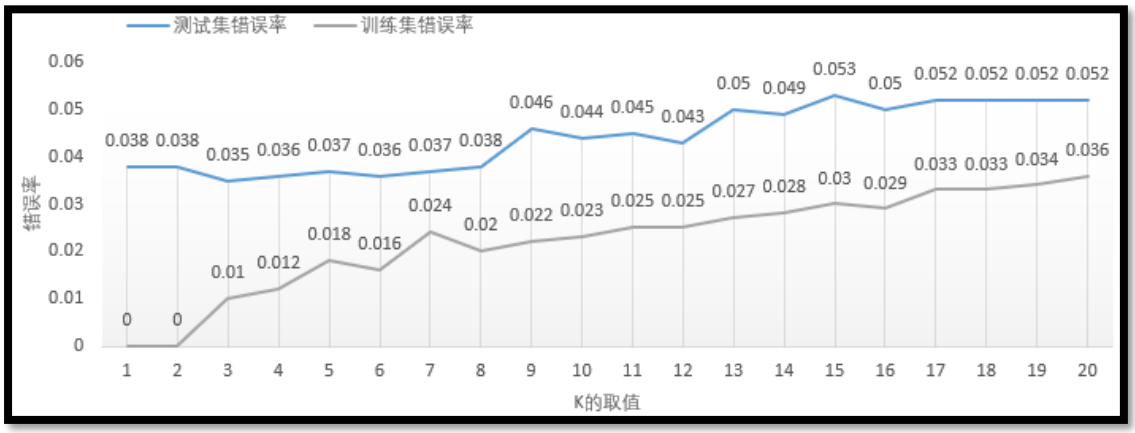

图 1-6 错误率曲线整合结果

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：

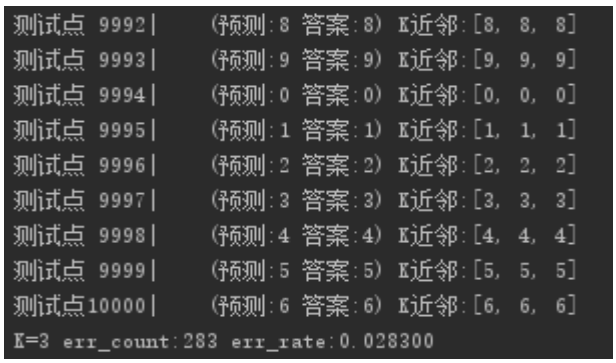

图 1-7 部分测试结果

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：

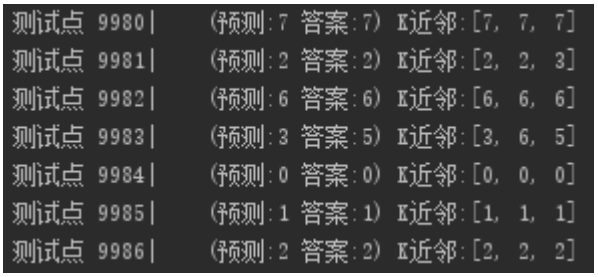

图 1-8 预测错误情况示例

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。