基于 K 近邻的手写数字字母识别

摘要：本次实验是基于 K 近邻（K Nearest Neighbors，简称 KNN）算法的手写数字字母识别，主要目的是辨认出生活中复杂多样的手写字体。实验的主要方法，也就是 K 近邻法，是一种常用的基本分类和回归方法。我们同时处理了 KNN 的两种实现方案：暴力法与 KD 树（K Dimensional Tree，简称 KD 树），对比了它们在时间开销上的区别。实验结果显示，在运用 KNN 进行手写数字字母识别时，如同时使用 KD 树进行加速，可以在较为满意的时间开销下获得良好的识别正确率。

关键词：K 近邻；KD 树；手写识别

1 引言

字体识别处理是从自然世界获取信息的重要方法。光学字符识别技术也是计算机视觉领域的一大研究课题。传统的识别技术把文本进行字符切割，然后进行识别，识别的文本主要是背景简单、字体大小一样的高质量文本，在识别方面具有一定的局限性。在自然场景下，由于场景复杂多变，文本大小不一致，识别难度显著提高。因而传统字体识别只能处理出版物等上的规范字体，对于形状各异、更复杂的手写字体却无能为力。

随着信息化技术的发展，机器学习方法的出现使识别复杂多样的手写字体成为可能。本文中我们使用 KNN 算法来进行手写数字字母识别的实验，以加深对 KNN 算法和字体识别处理过程的理解。

2 本文工作

K 近邻法是一种基本的分类与回归方法。用于分类的 K 近邻法的输入为带标记的与不带标记的实例的特征向量，可表示特征空间上的点；输出为原不带标记的实例的类别，可以处理多分类问题。K 近邻法的基本思路是，对每一个实例，根据其特征空间上 K 个近邻的训练实例的类别，通过多数表决等方式进行预测。

一般来说，K 近邻法模型由距离度量准则、分类决策规则与 K 值的选择这三个基本要素决定。[1]

特征空间中两个点的距离的远近反映两个实例间的相似程度大小。一般，距离度量准则有欧式距离、曼哈顿距离等。它们综合考虑特征的每一个维度，但是在维度单位不同时，需要先进行加权或归一化处理。对于这个图像识别问题，特征实例为规范化的二值图像，每一个维度对应图像某一位置的像素，维度意义一致，不需要额外处理，直接选用欧式距离。

K 值的选择则是一个值得考虑的要素。一个较小的 K 值意味着只在一个较小的邻域中进行预测。如果 K 值过小，预测结果会严重地受到邻域内的噪声影响，些许噪声也许就会导致预测出错。换句话说，容易产生过拟合。但是如果 K 值过大，距离较远的实例点也会对预测产生影响，邻域内更有价值的信息被淹没，导致模型变得过于简单而且不敏感。我们的实验中将进行交叉验证以选择最优的 K 值。[2]

分类决策规则通常采用多数表决制，邻域内的有标记特征实例点各自投票一个类别（即它自己所属的类别），预测结果为得票最多的类别。多数表决制还要求 K 值最后选取奇数，以尽量避免几个类别票数相同的尴尬境地。

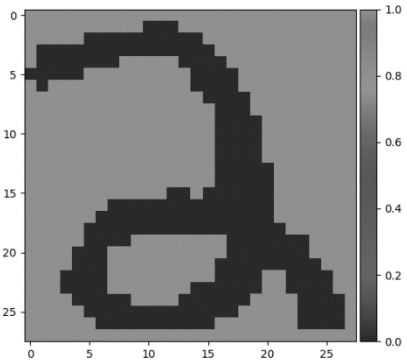
实际分类过程中，我们将实例处理成 784 行的列向量，并将其随机打乱，取其中 20%的实例用于测试。对于每一个测试实例，我们要算出距其前 K 近的实例。如果进行排序，则至少会带来 O(NlogN)的时间复杂度（其中 N 为实例总数）；但是，由于 K 一般远小于 N，我们可以通过构造一个大小为 K 的最大堆，遍历实例的距离并动态更新来实现 O(N)的时间复杂度。而测试实例数量为 0.2N，因此总的时间复杂度还是 O(N^2)级的。

一种加速方案是使用辅助数据结构 KD 树。将有标记的实例按某个维度的值排序，选择中位数划分一个超平面，对实例构建节点挂载到树上，循环选择维度直到所有实例都变成树上的节点。这个 KD 树在后续每次预测时都能用到。对于每一个测试实例，选择当前节点要求的维度进行比较，根据结果向左子树或右子树继续查询；到达叶子节点时即可得到一个相对较近的实例点，但是它未必就是最终结果的其中之一，根本原因在于离测试实例更近的点也可能在超平面的另一侧；因此还需沿着查询路径回溯，利用沿途的兄弟节点更新结果。

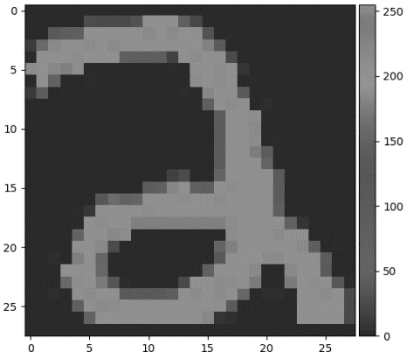
3 实验结果与分析

实现算法前，我们首先要对图像进行预处理，因为不是每一个像素都能反映图像的特征。第一步是进行放缩或者裁剪，以保证图像中字母或数字所占比例相同，同时使得图像大小一致以确保特征实例具有相同的维度。接下来我们随机查看几个图像不难发现，原图像有 255 个灰度级，且在字母或数字形状之外还有一些噪声点。鉴于我们关心的特征仅包括数字或字母的形状而无关颜色深浅，过多的灰度级会拖慢距离计算的速度，噪声点则会降低预测的准确度。为此，我们要采取一些特征提取的手段。这里尝试了阈值处理和边缘检测。

阈值处理是取图像各个像素点灰度值的均值为阈值，然后将灰度值低于该阈值的像素置为 0，高于的置为 1。边缘检测则是用检测算子（例如 Sobel）卷积原图像，生成一张反映边缘信息的二值图像。



图一 阈值处理后的图像

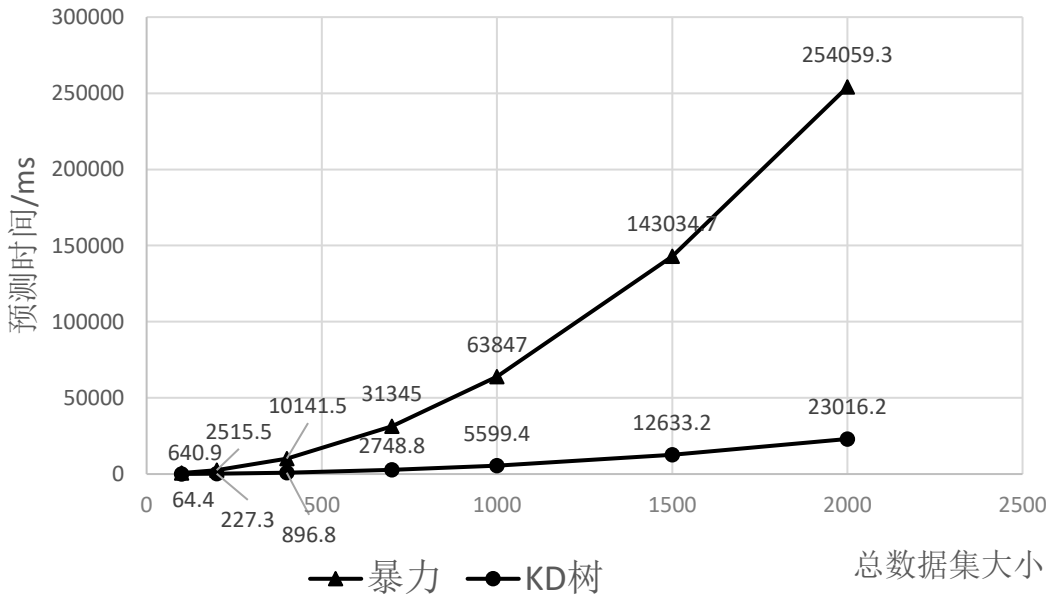


图二 原图像

如图所示是阈值处理后的图像与原图像的对比，不难发现，阈值处理简化了图像，突出了特征。后续的实验也显示，阈值处理的图像在预测中精度表现更佳。

KD 树仅仅是一种加速手段，不改变 K 近邻的思想，理论上不影响准确率。我们对于不同规模的数据集进行对比实验，以研究 KD 树的加速效果。始终使用数据集中 20%的实例用作测试。结果如下图所示。

图三 暴力法与KD树法运行时间对比

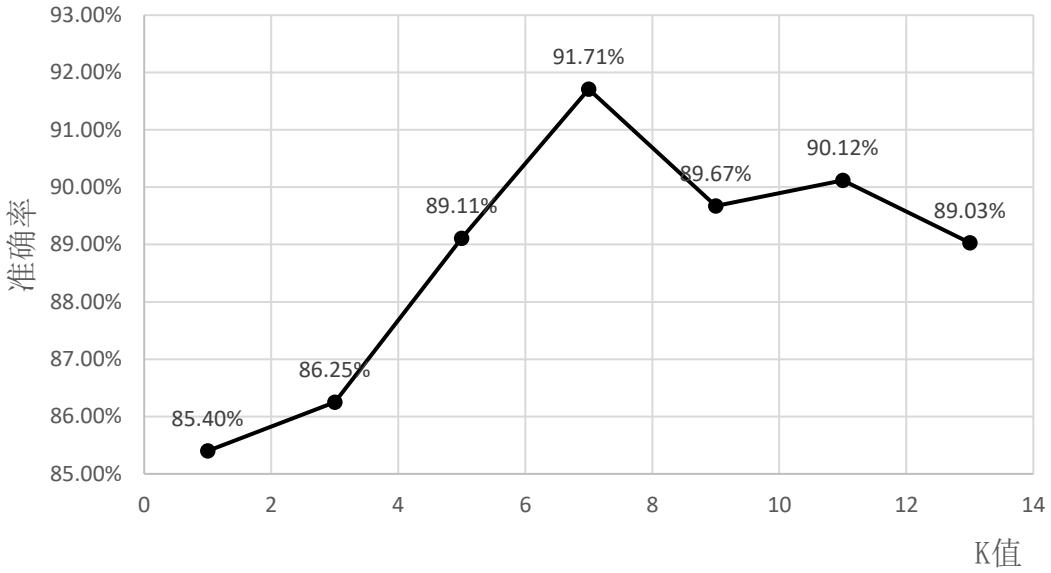


显然，随着数据规模的增加，暴力法的时间开销近似呈平方趋势增长，而 KD 树的优化效果也很明显。

而为了找到最佳的 K 值，我们有必要进行多次测试，挑选准确率最好的一次。但是在整体数据集上进行测试很容易导致过拟合，为此我们采用 N 折交叉验证，

将整个数据集均分为 N 份，然后用第一份数据做预测得出准确率 a1，接着换用第二份……一共进行 N 次，取平均得到最终的准确率。改变 K 值重复测试得到：

图四 准确率与K值的关系



显然，K 值过小时，噪声会严重影响预测的准确率；而随着 K 值增大，不那么相似的实例也会加入投票从而拉低准确率；在本实验中，取 K 值为 7 时预测效果最佳。

4 总结

在本次实验中我们使用 KNN 进行手写数字字母的识别。为了获得更好的效果，在预处理阶段对图像使用了阈值化。为了比较不同实现方案的时间开销，我们进行了暴力法与 KD 树法的对比实验，并验证了后者的显著加速效果。KNN 中 K 值的选择是一个重要要素，过大或过小都会降低最终准确率，为此我们使用交叉验证法，通过多次实验，找到了对应已知数据集最优的 K 值。

但是，实际场景可能比实验中的数据集还要复杂，比如手写字体只是图像中的一部分时，还要先使用图像分割等技术提取字体。换句话说，实际工程需要更多的预处理。另外，就本实验中所用的方法 KNN 而言，对于每一个待预测实例，都需要计算所有有标记实例，同时实例必须全部加载到内存中，时空复杂度都较大。在数据量很大时可能需要考虑支持向量机、神经网络等方法。

参考文献

[1]李航. 统计学习方法. 北京:清华大学出版社, 2012. P38
[2]周志华. 机器学习. 北京:清华大学出版社, 2016. P26

附录一 KNN 暴力法核心代码

```
# KNN with brute method
import pandas
import numpy as np
import random
import heapq as hp
import matplotlib.pyplot as plt
from skimage import io, filters

# load data from specified location
csv = pandas.read_csv('E://handwriting numbers and letters.csv')
li = [i for i in range(len(csv['data']))]
# shuffle the data
random.shuffle(li)
length = len(li)
print('li len: ' + str(len(li)))

# simplify the data and build the vectors
data = csv['data']
vectors = []
for i in li:
    img = np.array(eval(data[i]))
    thres = filters.threshold_otsu(img)
    vectors.append(((img <= thres) * 1).reshape(28 * 28, 1))
vectors = np.array(vectors)

# build the labels
labels = []
for i in li:
    labels.append(csv['target'][i])
```

```
# classify test samples
tests_size = length // 5
cnt = 0
for i in range(length - tests_size, length):
    heap = [(- 10 ** 10, '')] * 5
    hp.heapify(heap)
    for j in range(length - tests_size):
        distance = sum((vectors[j] - vectors[i]) ** 2)
        hp.heappushpop(heap, (-distance, labels[j]))

    # calculate the votes and predict the label
    m = {}
    for _, label in heap:
        if label not in m:
            m[label] = 1
        else:
            m[label] += 1
    print(m)
    inverse = [(value, key) for key, value in m.items()]
    label = max(inverse)[1]

    # compare the prediction with the real label
    if label == labels[i]:
        cnt += 1
        print(cnt)
# print the accuracy
print(cnt / tests_size)
```

附录二 KD 树法核心代码

```
# KNN with KD-Tree
import numpy as np
import heapq
import pandas
import random
```

```
from skimage import filters


class Node:
    def __init__(self, vector, dimension=0, left=None, right=None):
        self.vector = vector
        self.dimension = dimension
        self.left = left
        self.right = right

    def __cmp__(self, other):
        return -1

class KDTree:
    def __init__(self, vectors):
        # k dimensions
        l = len(vectors)

        def create(vectors, dimension):
            if len(vectors) == 0:
                return None
            # sort by current dimension
            vectors = sorted(vectors, key=lambda x: x[dimension])
            mid = len(vectors) // 2
            # split by the median
            e = vectors[mid]
            return Node(e, dimension,
                        create(vectors[:mid], (dimension + 1) % l),
                        create(vectors[mid + 1:], (dimension + 1) % l))

        self.root = create(vectors, 0)

    def nearest(self, x, k=1):
        # max heap
        heap = [(-np.inf, None)] * k
```

```
        def visit(node: Node):
            if node is not None:
                # cal the distance to the split point, i.e. the hyperplane
                dis = x[node.dimension] - node.vector[node.dimension]
                visit(node.left if dis < 0 else node.right)
                # cal the distance to the current nearest point
                cur_dis = np.linalg.norm(x - node.vector, 2)
                # push the minus distance to the heap
                heapq.heappushpop(heap, (-cur_dis, node.vector.tostring()))
                # compare the distance to the hyperplane with the min distance
                # if less, visit another node.
                if -(heap[0][0]) > abs(dis):
                    visit(node.right if dis < 0 else node.left)

        visit(self.root)
        return heap


# load data from specified location
csv = pandas.read_csv('E://handwriting numbers and letters.csv')
li = [i for i in range(len(csv['data']))]
random.shuffle(li)

off = []
for i in li:
    if csv['target'][i][-1].isdigit():
        off.append(i)
li = off
length = len(li)
print('li len: ' + str(len(li)))

# simplify the data and build the vectors
data = csv['data']
```

```
vectors = []
for i in li:
    img = np.array(eval(data[i]))
    thres = filters.threshold_otsu(img)
    vectors.append(((img <= thres) * 1).reshape(28 * 28, 1))
vectors = np.array(vectors)

# build the labels
labels = {}
for i in li:
    labels[vectors[i].tostring()] = csv['target'][i]
print(len(vectors[li[0]].tostring()))

tests_size = length // 5
cnt = 0

# build the kd-tree
tree = KDTree(vectors[:length - tests_size])

# classify test samples
for i in range(length - tests_size, length):
    heap = tree.nearest(vectors[i], 5)

    # calculate the votes and predict the label
    m = {}
    for _,vector in heap:
        label = labels[vector]
        if label not in m:
            m[label] = 1
        else:
            m[label] += 1
    print(m)
    inverse = [(value, key) for key, value in m.items()]
    label = max(inverse)[1]
```

```
        # compare the prediction with the real label
        if label == labels[vectors[i].tostring()]:
                cnt += 1
                print(cnt)
# print the accuracy
print(cnt / tests_size)
```