華中科技大學

课程实验报告

题目：KNN 算法实践

课程名称：机器学习

专业班级：CS1703

学　　号：U201714670

姓　　名：范唯

指导教师：李玉华

报告日期：2020 年 5 月 10 日

计算机科学与技术学院

目录

1.实验一
- 1.1 实验目的与要求
- 1.2 实验内容
- 1.3 实验方案
  - 1.3.1 读取MNIST数据集
  - 1.3.2 KNN训练算法
  - 1.3.3 KNN测试算法(K近邻图像)
  - 1.3.4 KNN测试算法(misclassification rate)
  - 1.3.5 参数设置
- 1.4 实验结果
  - 1.4.1 输出K近邻图像
  - 1.4.2 misclassification rate曲线
- 1.5 HUST登陆界面验证码识别
  - 1.5.1实现过程
  - 1.5.2 测试 结果分析

1.实验一

1.1 实验目的与要求

- 熟悉python编程语言
- 掌握knn算法实现过程
- 输入若干测试图片，输出对应每张图片k近邻的图片
- 绘制knn算法的训练misclassification rate曲线，并做出分析
- 自由发挥项目——**华中科技大学登陆界面验证码识别**

1.2 实验内容

- 使用python实现knn算法，并可以在输入图片后得出K近邻居的图片。
- 动态设置不同的k值，根据不同k值计算出对应的misclassification rate，并绘制出变化曲线。
- 自由发挥项目——**华中科技大学登陆界面验证码识别**

1.3 实验方案

整个实验过程分为五个步骤:

- 读取MNIST数据集
- KNN训练算法实现
- KNN测试算法(输出图像)
- KNN测试算法(输出misclassification rate)
- 参数设置(运行)

1.3.1 读取MNIST数据集

整个实验使用numpy与matplotlib库来实现

MINIST的训练集imgNum * rows * cols 一共有60000*28*28个像素值

MINIST的测试集imgNum * rows * cols 一共有10000*28*28个像素值

这里使用`numpy`中的`nbarray`类来保存图片像素数据 以及 图片label数据

并且使用二进制打开文档，读取至缓冲区。注意的是文件的前四个整数不是像素数据，而是图片文件的基础属性。

具体实现过程参考如下代码以及注释

```
# 读取图片
def read_image(file_name):
    #先用二进制方式把文件都读进来
    file_handle=open(file_name,"rb")  #以二进制打开文档
    file_content=file_handle.read()   #读取到缓冲区中

    offset=0
    head = struct.unpack_from('>IIII', file_content, offset)  # 取前4个整数，返回一个元组
    offset += struct.calcsize('>IIII')
    imgNum = head[1]  #图片数
    rows = head[2]   #宽度
    cols = head[3]  #高度

    images=np.empty((imgNum , 784))#empty，是它所常见的数组内的所有元素均为空，没有实际意义，它是创建数组最快的方法
    image_size=rows*cols#单个图片的大小
    fmt='>' + str(image_size) + 'B'#单个图片的format

    for i in range(imgNum):
        images[i] = np.array(struct.unpack_from(fmt, file_content, offset))
        offset += struct.calcsize(fmt)
    return images #返回图片像素数据array

# 读取标签
```

```
def read_label(file_name):
    file_handle = open(file_name, "rb")  # 以二进制打开文档
    file_content = file_handle.read()  # 读取到缓冲区中
    head = struct.unpack_from('>II', file_content, 0)  # 取前2个整数，返回一个元组
    offset = struct.calcsize('>II')
    labelNum = head[1]  # label数
    bitsString = '>' + str(labelNum) + 'B'  # fmt格式：'>47040000B'
    label = struct.unpack_from(bitsString, file_content, offset)  # 取data数据，
返回一个元组
    return np.array(label) # 返回图片打标array
```

1.3.2 KNN训练算法

KNN训练中我们采取的是欧式距离

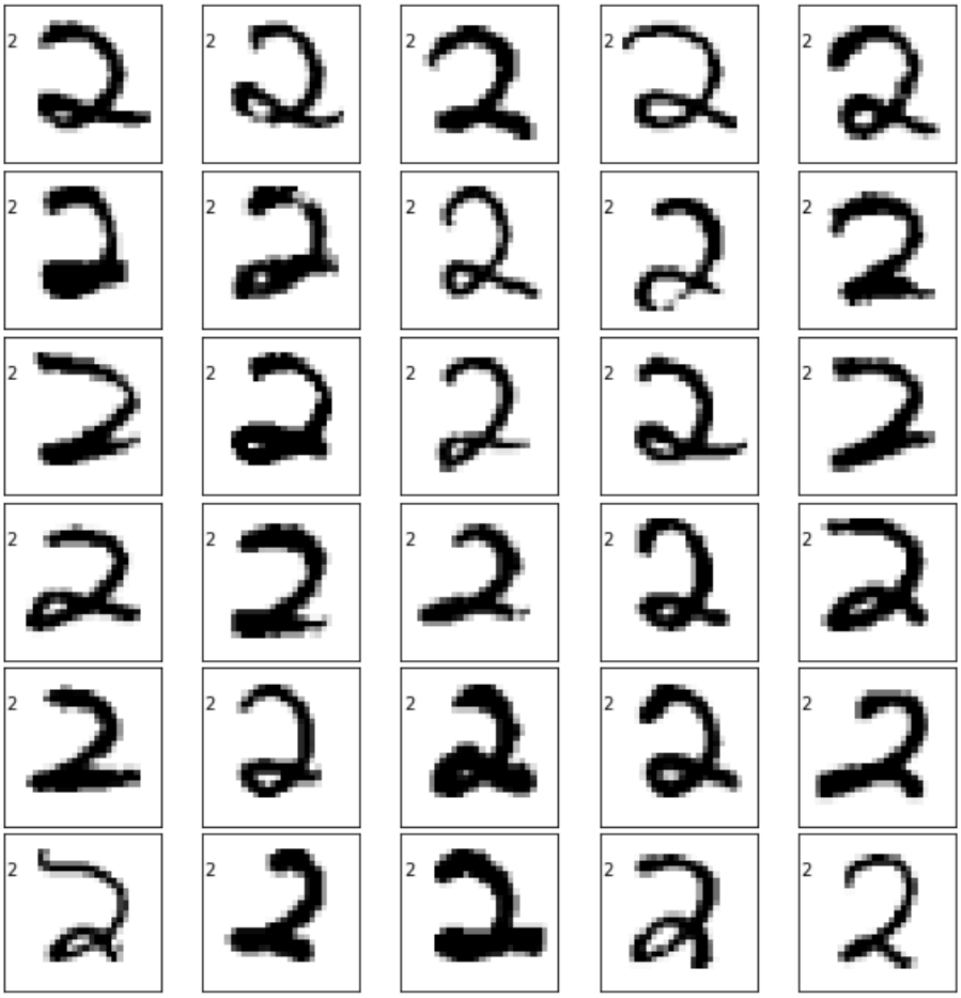
- images.shape[0]表示的是读取矩阵第一维度的长度，代表行数
- 将测试的图片复制60k次。再reshape成784行*60000列
- 每个元素平方、矩阵每行相加、合并成一行并开方
- 返回从小到大的索引数组，这个数组中包含的即是与测试图片相似图片的排行。

```
# KNN算法
def KNN(test_data, images, labels, k):
    # images.shape[0]表示的是读取矩阵第一维度的长度，代表行数60k
    dataSetSize = images.shape[0]  # 60k
    # tile函数在行上重复dataSetSize次，在列上重复1次
    distance1 = tile(test_data, (dataSetSize)).reshape((dataSetSize,784))-
images
    # 每个元素平方
    distance2 = distance1**2
    # 矩阵每行相加
    distance3 = distance2.sum(axis=1) # 合并成一行
    # 开方
    distances4 = distance3**0.5
    # 欧氏距离计算结束
    # 返回从小到大排序的索引
    sortedDistIndicies = distances4.argsort()
    classCount=np.zeros((10), np.int32) #10是代表10个类别
    for i in range(k): # 统计排在前k名的预测结果
        ## 找到图片对应的数据
        voteIlabel = labels[sortedDistIndicies[i]]
        ## 把这个Count + 1
        classCount[voteIlabel] += 1
    return np.argmax(classCount),sortedDistIndicies[0:k]
```

1.3.3 KNN测试算法(K近邻图像)

按照实验要求，需要输出与测试图像最相似的k个图像，这里采用的是`plt`的图片拼接和显示功能

预测结果为： 2 实际结果为： 2



实现过程如下：

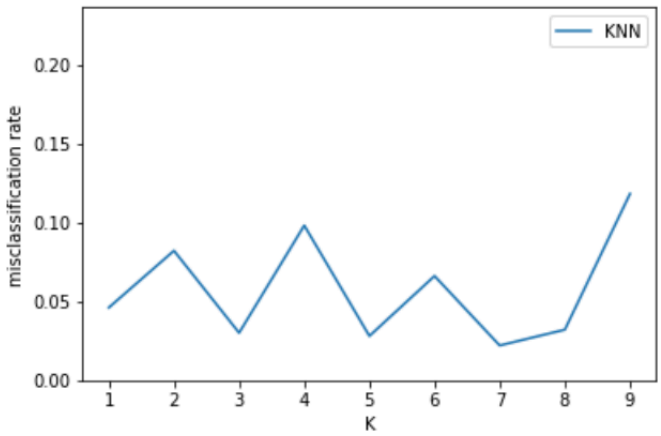
```
def KNN_TEST_PRINT(train_x,test_x,train_y,test_y,test_id,k):
    result,sortlist = KNN(test_x[test_id], train_x, train_y, k)
    print('预测结果为:',result,'实际结果为:',test_y[test_id])
    ## 打印图片
    fig=plt.figure(figsize=(8,8))
    fig.subplots_adjust(left=0,right=1,bottom=0,top=1,hspace=0.05,wspace=0.05)
    for i in range(len(sortlist)):
        images = np.reshape(train_x[sortlist[i]], [28,28])
        ax=fig.add_subplot(6,5,i+1,xticks=[],yticks=[])
        ax.imshow(images,cmap=plt.cm.binary,interpolation='nearest')
        ax.text(0,7,str(train_y[sortlist[i]]))
    plt.show()
    plt.pause(5)##停留5秒
    plt.close(fig)##关闭图片
```

1.3.4 KNN测试算法(misclassification rate)

按照实验要求，我们需要动态的根据`k`的值来计算出不同的misclassfication rate并绘制出变化曲线。

所以需要设计一个函数来根据输入k来返回不同的error_rate。还可以根据要求自定义返回值。例如：

- 正确预测率
- 错误预测率
- 错误预测的图片数量




```
def KNN_TEST(train_x,test_x,train_y,test_y,k):
    testNum = test_x.shape[0]
    print('测试图片数量:',testNum)
    errorCount = 0  # 判断错误的个数
    #error = []
    #error_image = []
    errorCount = 0  # 判断错误的个数
    for i in range(testNum):
        result,sortlist = KNN(test_x[i], train_x, train_y, k)
        # print('返回的结果是: %s, 真实结果是: %s' % (result, test_y[i]))
        if result != test_y[i]:
            errorCount += 1.0
            # 如果mnist验证集的标签和本身标签不一样，则出错
            ## print('返回的结果是: %s, 真实结果是: %s' % (result, test_y[i]))
            ## 为了更加直观，只打印出错误的预测结果
            #error.append(test_y[i])
            #error_image.append(sortlist)
    error_rate = errorCount / float(testNum)  # 计算出错率
    #acc = 1.0 - error_rate
    return error_rate
```

1.3.5 参数设置

在KNN开始训练、测试前。需要传入训练集、测试集的数据路径。并通过`read_image`和`read_label`转换为ndarray类型的数据。

并且需要设置训练集图片数量、测试集图片数量、以及所选`k`值，例如：

- 训练图片数 trainNum = 6000
- 测试图片数 testNum = 100
- 距离最小的k个图片 k = 30

```
# 读取数据
train_x = read_image(train_image)  # train_image
test_x = read_image(test_image)  # test_image
train_y = read_label(train_label)  # train_label
test_y = read_label(test_label)  # test_label

train_Start = random.randint(0,60001-trainNum)# 随机选取训练集数据
train_End = train_Start+trainNum
train_i = train_x[train_Start:train_End,:] # 切割
train_l = train_y[train_Start:train_End]

test_Start = random.randint(0,10001-testNum) #随机选取测试集数据
test_End = test_Start+testNum
test_i = test_x[test_Start:test_End,:]  # 切割
test_l = test_y[test_Start:test_End]
```

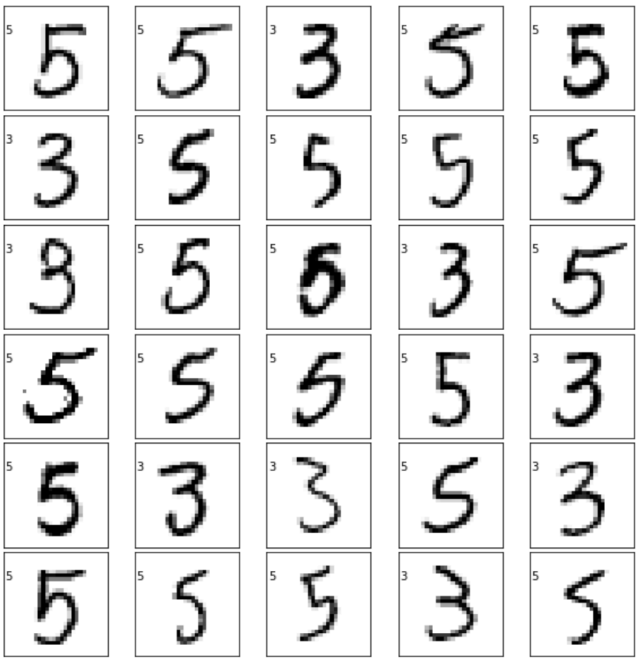
1.4 实验结果

1.4.1 输出K近邻图像

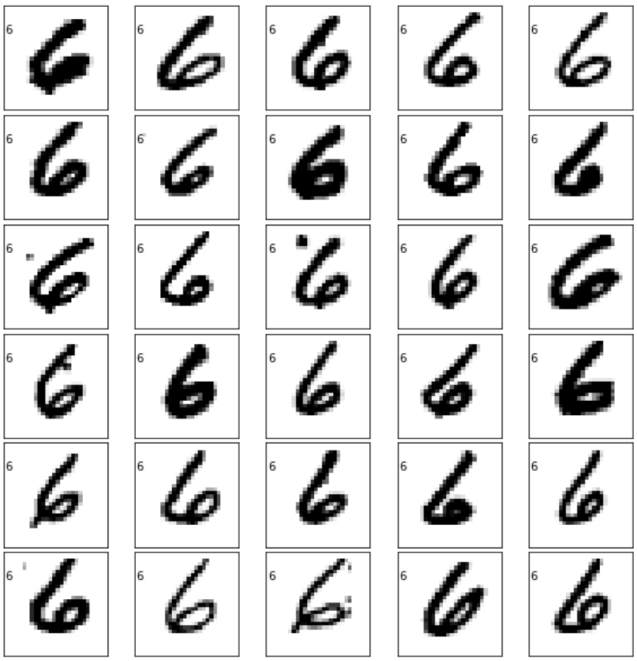
随机选取测试集中的一副图。并传入`KNN_TEST_PRINT`中。

其中为了结果的直观，我们将k设置为30

- 测试一：预测结果为: 5 实际结果为: 5



- 测试二： 预测结果为: 6 实际结果为: 6



- 测试三：预测结果为: 2 实际结果为: 2

综上可以看出knn的识别匹配效果达到了基本要求。

1.4.2 misclassification rate曲线

因为KNN的性能在高纬度时会下降。所以选取6k张训练集图片和500张测试集图片。将k设置为1～10之间。

来绘制misclassfication rate曲线。

代码如下

```
# 训练图片数
trainNum = 6000
```

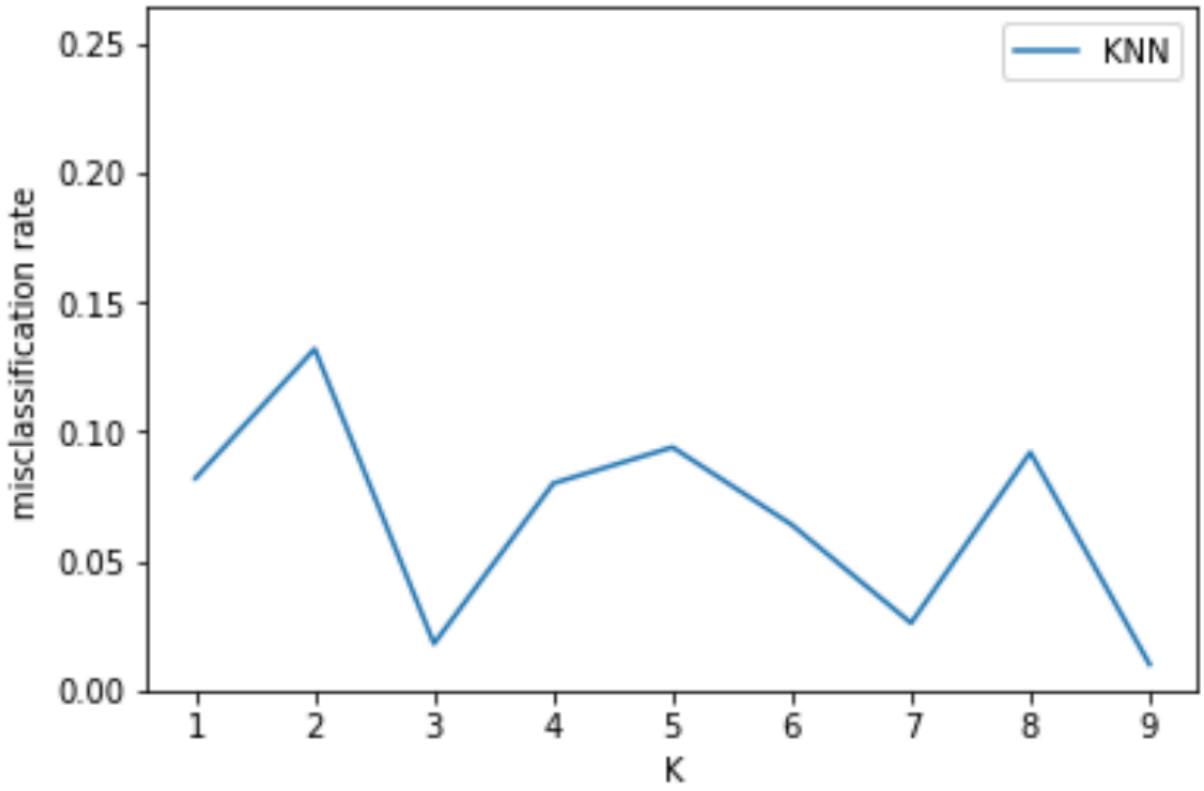
```
# 测试图片数
testNum = 500

x = np.arange(1, 10, 1)
y = []
for t in x:
    ##数据选择
    train_Start = random.randint(0,60001-trainNum)
    train_End = train_Start+trainNum
    train_i = train_x[train_Start:train_End,:]
    train_l = train_y[train_Start:train_End]
    test_Start = random.randint(0,10001-testNum)
    ## test_Start = 0
    test_End = test_Start+testNum
    test_i = test_x[test_Start:test_End,:]
    test_l = test_y[test_Start:test_End]
    print('\nNo.',t)
    print('训练集图片数量：',train_i.shape[0],'区间:',train_Start,'~',train_End)
    print('测试集图片数量：',test_i.shape[0],'区间:',test_Start,'~',test_End)
    misclassification_rate = KNN_TEST(train_i,test_i,train_l,test_l,t)
    print('K=',t,'accuracy:',1-misclassification_rate)
    y.append(misclassification_rate)
plt.plot(x, y, label='KNN')
plt.xlabel("K")
plt.ylabel("misclassification rate")
plt.ylim(0,2*max(y))
plt.legend()
plt.show()
```

运行结果为：

```
No. 1
训练集图片数量： 6000 区间: 49303 ~ 55303
测试集图片数量： 500 区间: 9438 ~ 9938
测试图片数量: 500
K= 1 accuracy: 0.918

No. 2
训练集图片数量： 6000 区间: 38563 ~ 44563
测试集图片数量： 500 区间: 4042 ~ 4542
测试图片数量: 500
K= 2 accuracy: 0.868

No. 3
训练集图片数量： 6000 区间: 21625 ~ 27625
测试集图片数量： 500 区间: 8959 ~ 9459
测试图片数量: 500
K= 3 accuracy: 0.982
```

```
No. 4
训练集图片数量： 6000 区间：39939 ~ 45939
测试集图片数量： 500 区间：3489 ~ 3989
测试图片数量：500
K= 4 accuracy: 0.92

No. 5
训练集图片数量： 6000 区间：21587 ~ 27587
测试集图片数量： 500 区间：257 ~ 757
测试图片数量：500
K= 5 accuracy: 0.906

No. 6
训练集图片数量： 6000 区间：28045 ~ 34045
测试集图片数量： 500 区间：2604 ~ 3104
测试图片数量：500
K= 6 accuracy: 0.9359999999999999

No. 7
训练集图片数量： 6000 区间：43593 ~ 49593
测试集图片数量： 500 区间：6955 ~ 7455
测试图片数量：500
K= 7 accuracy: 0.974

No. 8
训练集图片数量： 6000 区间：9651 ~ 15651
测试集图片数量： 500 区间：9481 ~ 9981
测试图片数量：500
K= 8 accuracy: 0.908

No. 9
训练集图片数量： 6000 区间：34086 ~ 40086
测试集图片数量： 500 区间：8445 ~ 8945
测试图片数量：500
K= 9 accuracy: 0.99
```

misclassification rate曲线如下：

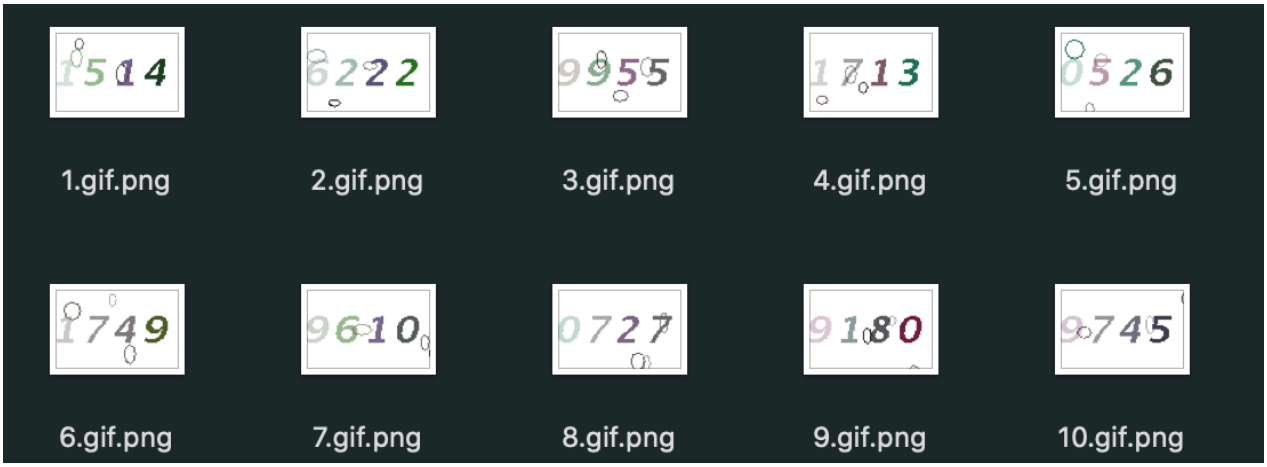


可以发现 K = 3时错误率最低，K=9时算法表现也不错。

1.5 HUST登陆界面验证码识别

1.5.1实现过程

1.爬取一定数量的验证码GIF。抽取每个GIF中的第二张图片作为完整验证码的PNG



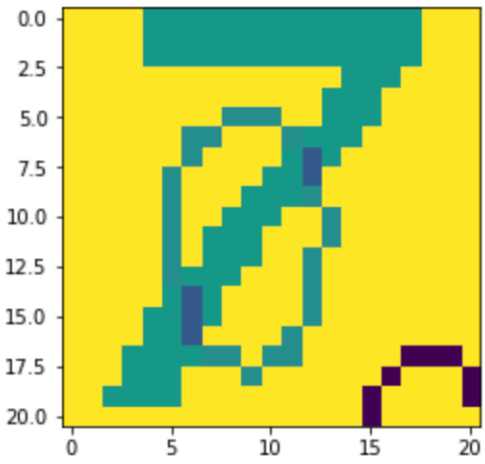

```
def GIF_toPNG(path,savepath):
    #print("GIF/%d.gif"%(i+1))
    img = Image.open(path)
    select = 0
    for frame in ImageSequence.Iterator(img):
        select+=1
        if(select==2):frame.save(savepath)
    code_img = Image.open(savepath)
    code_size = code_img.size
    h = code_size[0]
    w = code_size[1]
    # print("GIF_PNG/%d.PNG"%(i+1))
    print("RAW GIF:%s\n保存 PNG:%s"%(path,savepath))
```

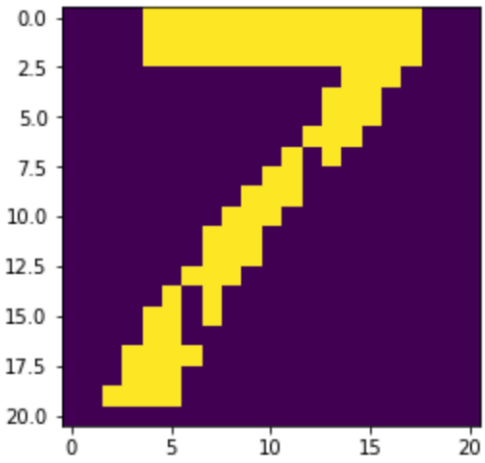
2.制作训练集与测试集、切分PNG为4个单独数字的PNG格式图像、并保存到GIF_SPLIT文件夹下。

值得注意的是验证码中的干扰噪声的像素块与非噪声的像素块的值不一样。所以可以根据这个特性以及干扰噪声出现频率低的特别把它筛选出来。

```
def GIF_SPLIT(PATH,SAVEPATH):
    png_path = (PATH)
    code_img = Image.open(png_path)
    for i in range(4):
        ## 切隔成4张数字
        left = 21*i
        low = 19
        right = 21*(i+1)
        up = 40
        split = (left,low,right,up)
        img_save = code_img.crop(split)
        plt.imshow(img_save)
        plt.show()
        # img_save.save("GIF/%d.png"%i)
        img_array = np.array(img_save)


        ## 统计噪声
        ## Counter dict sorted
        result = Counter({})
        noise = []
        for j in img_array:
            result+=Counter(j)
        result = dict(result)
        #print(result)
        s = sorted(result.values(),reverse = True)
        img_array[img_array==s[0]]=0
        #print(i)

        ## 将噪声和背景全部置0，其他的全部置为1
        for k,v in result.items():
            if(v != s[0] and v != s[1]):
                noise.append(k)
            if(v==s[0]):
                zero = k
            if(v==s[1]):
                val = k
        for n in noise:
            img_array[img_array == n] = 0
        img_array[img_array == zero] = 0
        img_array[img_array == val] = 255

        ## 打印去掉噪声后的图形
        im = Image.fromarray(img_array)
        plt.imshow(im)
        plt.show()
        im.save(SAVEPATH+"-%d.png"%i)
    #np.savetxt("GIF/%d.txt"%i,img_array,fmt =   "%4d",delimiter=',')
```

降噪之后的结果如下：





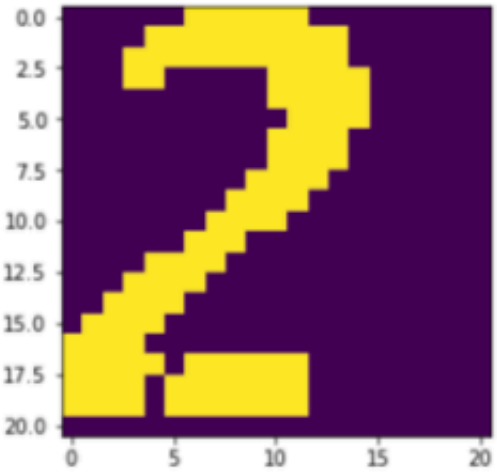
3.编写KNN算法

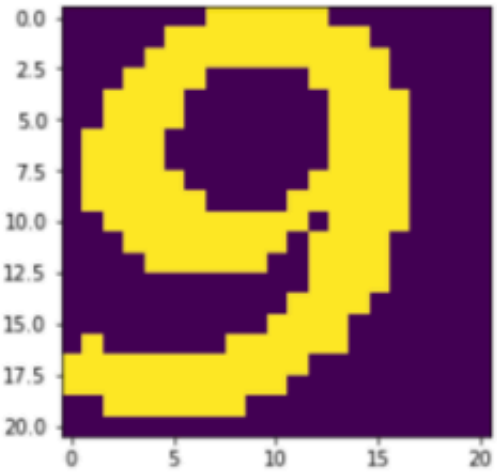
与MNIST识别的算法基本思路一致，详情请见源码包中的hust_code.ipynb

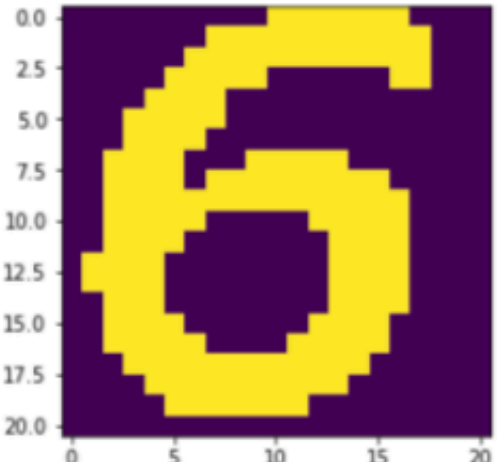
1.5.2 测试 结果分析

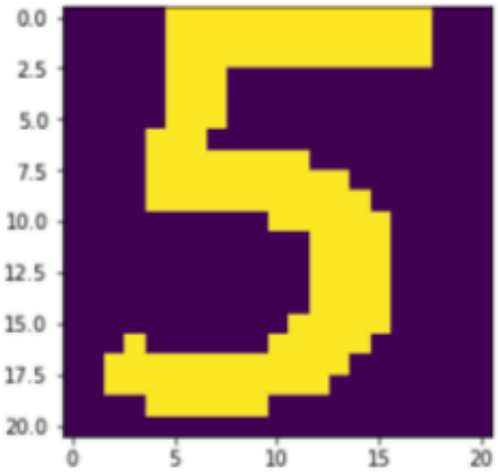
测试集中的二维码识别情况

选取未出现过在训练集中的图像进行测试，结果分析如下。每个数字匹配出最相似的三个数字图像


0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
5
10
15
20


0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
5
10
15
20


0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
5
10
15
20


0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
5
10
15
20