计算机视觉和模式识别 Final Project

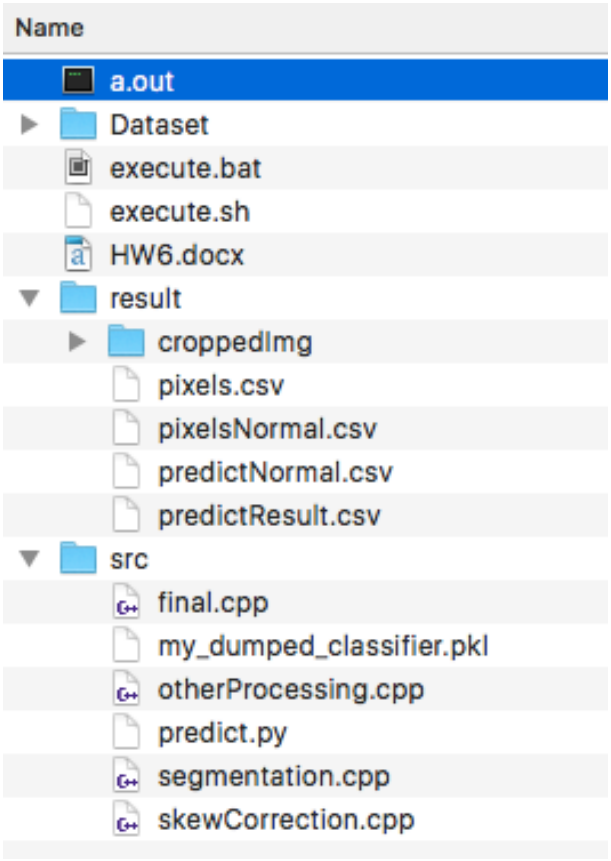
13331231
孙圣
计应 2 班

一、使用说明

通过 sh execute.sh 或 execute.bat 直接编译运行即可（需要安装 opencv,python3,numpy,scipy,scikit-learn）

测试环境：MAC OSX 10.11

目录结构：



1. 其中 a.out 和 execute.exe(还未编译产生)为 C++部分的可执行程序；
2. Dataset 中保存着被测试的三张图片；
3. execute.bat 和 execute.sh 为直接运行所有程序的脚本程序；
4. result 文件夹中保存着结果：
    4.1. croppedImg 目录下保存着所有的截取的数字图像；
    4.2. pixels.csv 保存着图片的 id 和展开的像素值；
    4.3. predictResult.csv 保存着预测的值；
5. src 文件夹中保存着所有源文件以及训练好的 SVM 模型。

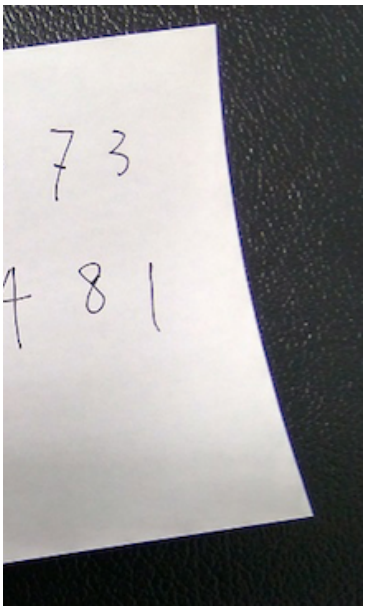
二、实验过程

1. 图像的矫正(skewCorrection.cpp)

这次实验是把前几次的作业综合起来的实验。因此，最重要的是保证前几次的代码能够重用而不至于出现太多的 BUG。

首先利用的是作业三中的 A4 纸矫正的代码。直接对 3 张图片进行测试发现，图片 2 和图片 3 都能够成功矫正，而图片 1 不能，原因是：图片 1 中存在一个明显的红黑分界线，在利用边缘检测和霍夫变换时，会检测到该分界线：

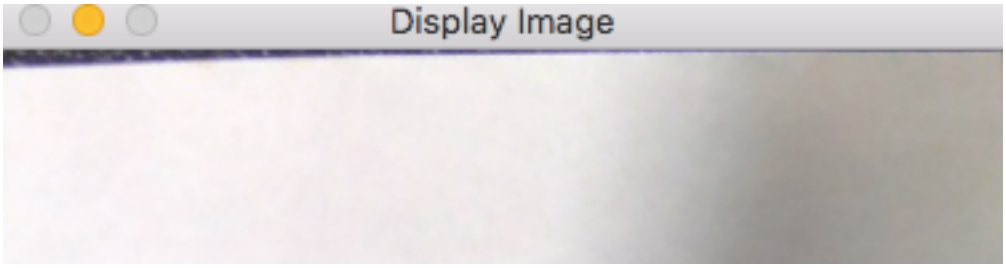
所以一开始的想法是在霍夫变换时，提高相应的阈值，使得这条额外的边不会被检测到。但是这样做会带来额外的问题，导致右侧的边无法检测到。右侧的边由于有所弯曲，并不是在一个完美的直线上，因此当提高霍夫变换的阈值时，该边缘由于包含的点较少，而被排除在外：



所以并不能通过简单的参数调节来解决这个问题。因此又考虑通过去重的方法解决。但是，以前所写的去重主要是负责去除与纸张明显不平行的线段。但在这里却并不适用，因为无关的边缘恰好与纸张的边缘平行，因此要利用其它特点。

考虑到这张图片的特性，即无关的边缘与纸张边缘距离相近，因此可以设置一个阈值来将近距离的边缘排除。在作业 3 中，考虑过排除相隔距离较远的边缘线段，因此只需要复用那段代码，将相应的参数修改一下即可。

完成了以上修改，对于图片一的矫正基本完成。还有一点需要注意的是，由于图片的边缘并不是一条完美的直线，因此矫正完成后很有可能在边缘处参杂了许多无关像素点，这会给之后的切割带来很大的麻烦，因此要去除：



方法就是通过把边缘处割去几个像素点：

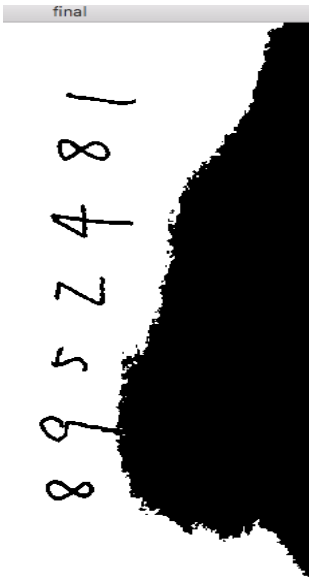
```
Mat boundaryRemovedImg(croppedImg, Rect(8, 8, croppedImg.cols - 16, croppedImg.rows - 16));
```

2. 图像切割(segmentation.cpp)

对于图像的切割主要分为两部分，第一部分为竖直切割，先将纸张切割呈条带状，这里主要使用的方法是在 x 轴计算频率直方图然后分割。

2.1 stripSegmentation：

首先要做的是将图片二值化，一开始考虑使用作业 5 中完成的 OTSU，但是发现 OTSU 并不适合这个场景,对图片 1 进行 OTSU 发现结果很不理想，右侧一大片区域被设为黑色：



主要原因为光线问题，在拍摄的时候那部分已经很暗，因此使用 OTSU 就会把它划为黑色的部分。

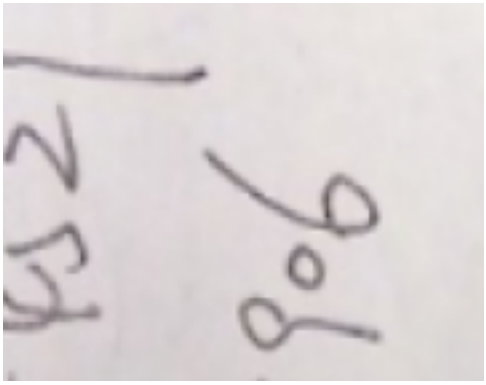
后来改用 opencv 提供的 adaptiveThreshold()方法，其中最后两个参数需要不断调节测试，同时将黑白反转，与 MNIST 中的数据一致：

```
adaptiveThreshold(grayImg, binarizedImg, 255, ADAPTIVE_THRESH_GAUSSIAN_C, THRESH_BINARY_INV, 35, 27);
```

之后就要统计每一列中白色的像素点的数目，统计结果大致如下：

```
MacBook-Air-3:Final Project sunsheng$ sh execute.sh
=====Frequency for Each Col=====
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 2 3 3 6 12 20 25 29 27 25 22 20 23 28 37 31 29 21 19 21
25 27 25 24 23 22 22 46 52 57 41 43 40 35 35 31 33 43 40 35 34 35 33 44 38 28 7
7 7 9 8 7 10 10 14 13 11 9 6 6 7 6 5 3 3 3 3 3 3 3 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7 9 8 6 10 15 19 12 16 22 20 26 4
6 56 44 37 34 26 31 38 28 26 29 30 29 28 28 29 29 25 30 38 56 49 22 20 20 26 20
16 16 15 14 14 18 38 40 17 15 18 14 9 9 9 8 8 7 8 9 10 9 9 9 9 6 5 6 6 5 3 3 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0
```

可以发现连续 0 的位置就是没有文字的位置，也是我们首选的切割点。但是对第 3 张图测试发现，情况并不是这么理想。因为图中第二列和第三列之间并没有空白：



相应的像素统计信息如下，红色框住的部分应该是分割点，但是并不是全为 0 的，而是连续的 2：

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 6 9 6 14 21 27 28 32 35 43 42 42 3
7 34 33 38 38 37 35 32 37 44 56 51 42 48 41 40 58 45 30 31 31 31 30 24 22 18 10
8 3 3 3 3 3 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7 10 12 29 3
7 50 47 40 47 45 36 31 32 26 29 34 32 32 34 37 36 35 40 45 44 34 32 28 19 15 17
17 16 13 9 12 6 4 4 2 2 2 2 2 2 3 3 3 3 3 3 3 4 4 3 4 4 4 4 4 3 3 3 3 5 17 25
21 18 16 12 16 20 22 21 24 25 26 29 28 33 35 32 31 35 35 44 45 39 42 37 37 39 41
 48 30 20 20 14 6 2 2 3 3 3 3 3 3 3 3 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

因此，下一步考虑统计像素低于某一阈值(本次实验中设为 3)连续出现的次数。对于图 3，有以下结果，红色框框住的区域就是上面所提到的两列数字有重叠的区域：

```
=====Continuous Low Times=====
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 5 6
 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 5 6 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 1
5 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 9
5 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 11
6 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 13
6 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 15
6 157
```

由此可见，我们只需要将 0->非 0 数字转变或者非 0 数字->0 转变的点作为切割点即可。对图 3 进行统计可得：

```
=====Dividing Point=====
22 67 96 134 141 206
```

之后就要进行切割。还有一个优化的地方：按照之前提到的方法寻找切割点，会导致数字的某些像素就在图片的边缘，与 MNIST 数据集相差较大，因此要补充一定的像素作为边界：

```
grayImg(Rect(pos[i * 2] - extendedBoundary, 0,
    pos[i * 2 + 1] - pos[i * 2] + extendedBoundary, grayImg.rows)).copyTo(croppedImg[i]);
```

最后就是对竖直的图像进行旋转，转换成水平的图像。

2.2 characterSegmentation：

这一部分与前一部分相比难度更大，因为在图片 2 和图片 3 中，有部分数字挨的很近，到时按照投影的方法切割十分困难。因此这部分主要采用的是 opencv 中提供的 findContours()方法：

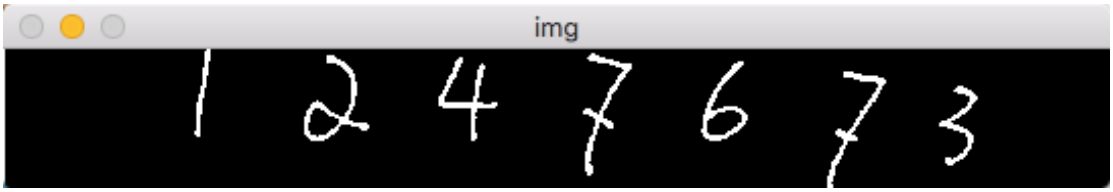
```
findContours(stripImg, contours, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE, Point(0, 0));
```

同时为了使得把数字作为一个整体识别出来而不是分为几个子区域，要对图片进行膨胀，增大白色的像素点：

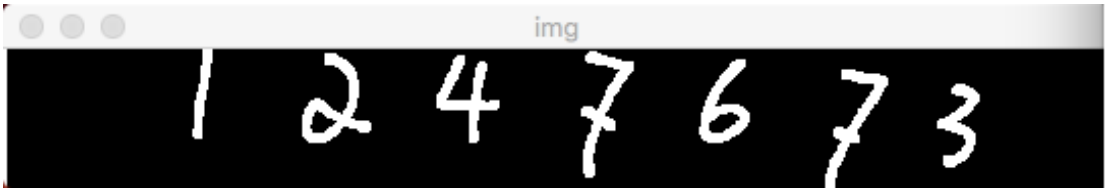
```
int size = 1;
Mat element = getStructuringElement(MORPH_RECT,
                                    Size(2 * size + 1, 2 * size + 1),
                                    Point(size, size));
dilate(stripImg, stripImg, element);
```

结果如下：

膨胀前：



膨胀后：

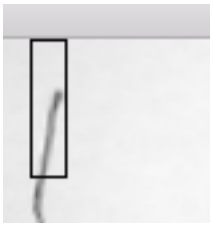


findContours()函数返回了一系列的点作为轮廓，首先我们要将其变为矩形，因此调用 boundingRect()方法。之后要对矩形进行筛选，如果太宽或者太窄都不可能是包含了数字的矩形，应该排除：

```
Rect bounding = boundingRect(contours[j]);
// http://docs.opencv.org/2.4/modules/core/doc/drawing_functions.html
// If the width of the region is too big or small, the region will never contain a digit.
if ( bounding.width > widthHigher || bounding.width < widthLower ) {
    continue;
}

boundingVector.push_back(bounding);
```

得到了一系列的矩形之后，要根据 x 轴的坐标进行排序，保证数字是从左往右依次排列的。之后就是对图像进行切割。这里要注意的是切割时不能按照给出的高进行切割，因为 findContours 返回的区域并不能完整的包括所要的数字，例如：

因此考虑将截取的高设为原图像的高：

```
Mat tmpImg;
srcImg[i](Rect(boundingVector[j].x, 0, boundingVector[j].width, srcImg[i].rows)).copyTo(tmpImg);
partialVector.push_back(tmpImg);
```

将切割好的图片放入 vector 内返回。至此，对数字的切割基本完成，剩下的就是进行相应的处理，并完成预测。

3. 图像的再处理(skewCorrection.cpp)

首先，要给刚才截取的图片添加边界。对于大部分的图片，宽远远比高要小。当之后要缩小到 28*28 的图像时，会带来很多的问题。所以就给图片填充纯黑色的边界：

```
if ( width < height ) {
    copyMakeBorder(dstImg, dstImg, 0, 0, (height - width) / 2, (height - width) / 2,
        BORDER_CONSTANT, Scalar(0, 0, 0));
    partialRetImg.push_back(dstImg);
} else {
    partialRetImg.push_back(srcImg[i][j]);
}
```

将图片缩小之后要做的就是风格上的处理，使得图片尽可能的接近 MNIST 数据集中提供的图片，提高正确率。有几点可以尝试的，第一，因为 MNIST 中的数字都很粗，所以可以利用膨胀来加粗；第二，可以对图像进行平滑处理等等。

4.数据的保存(final.cpp)

在训练之前，首先要把图片的数据都保存起来。首先利用 imwrite()方法，将图片写入到 result 目录下的 croppedImg 文件夹下。命名规范为 ImageID ID ID.jpg，三个 ID 分别为图片 ID，条带 ID 和位置 ID。

之后利用文件流的方法将 ID 和展开的像素值写入到 pixels.csv 中，以供 python 读取。

5.训练与预测(predict.py)

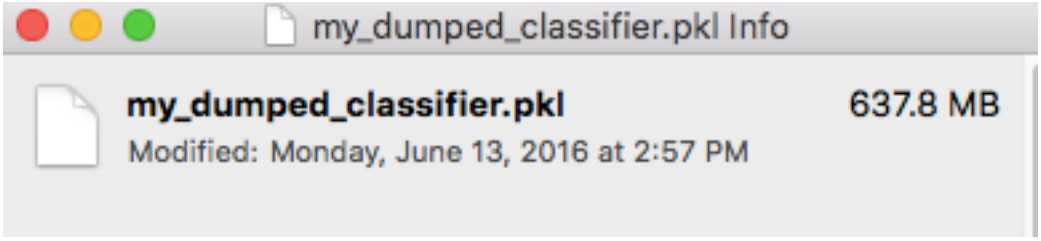
这部分的代码主要是依赖于作业 6 中的实现。首先利用 numpy 提供的 genfromtxt()方法读取之前存下来的图片数据。之后进行归一化。

这部分代码最大的改进地方在于，将训练的模型持久化的保存下来。这里利用了 pickle.dump()和 pickle.load()方法。这样就可以避免每次预测时都需要花费时间重新训练：

```
else:
    with open('src/my_dumped_classifier.pkl', 'rb') as f:
        clf = pickle.load(f)

predict = clf.predict(testImg).reshape(numberOfTestData, 1)
```

这里采用的模型是 SVM 的模型，因为模型占用的空间较小，只有 64KB，而使用了随机森林的 adaboost 模型却有 600M 左右：



因此尽管 SVM 模型在训练集上的正确率只有 92%，不如 adaboost 的 97%，考虑到空间因素，还是采用 SVM 模型。

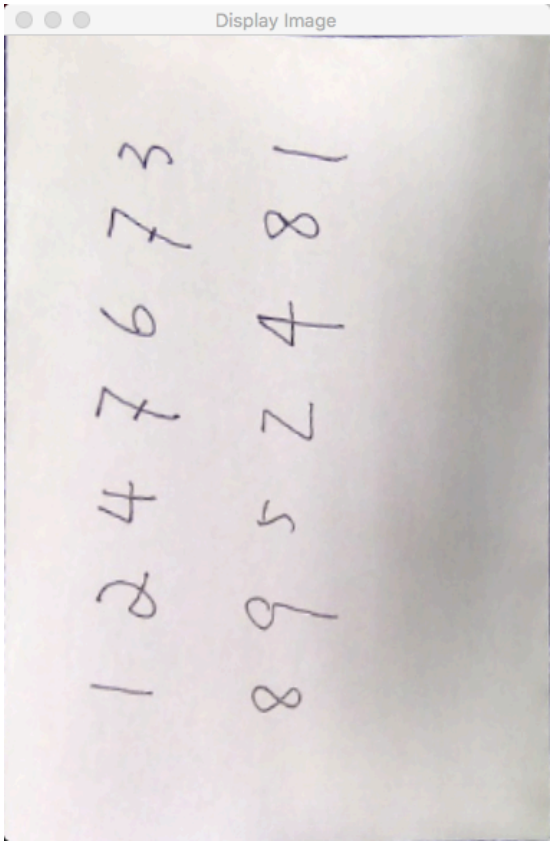
最后将预测的数据写入到文件中并在屏幕中输出。

至此所有步骤都成功完成，效果分析见实验结果部分。

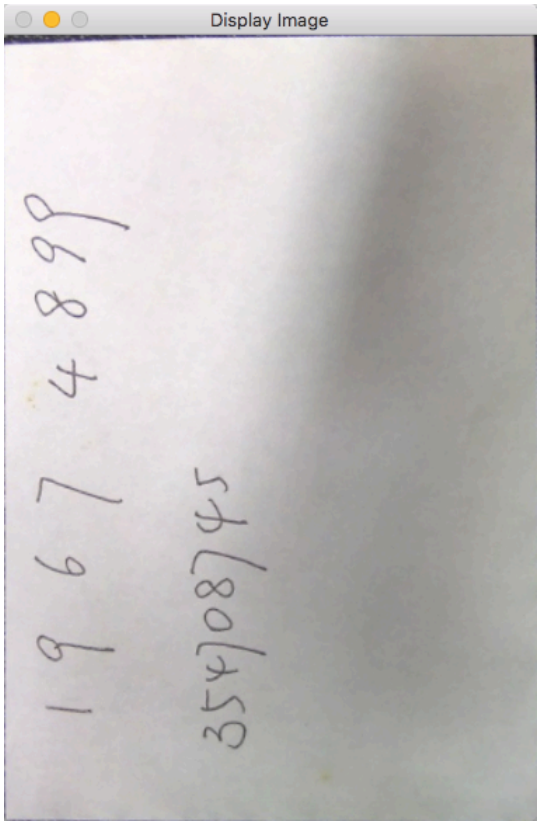
三、实现结果：

矫正结果：

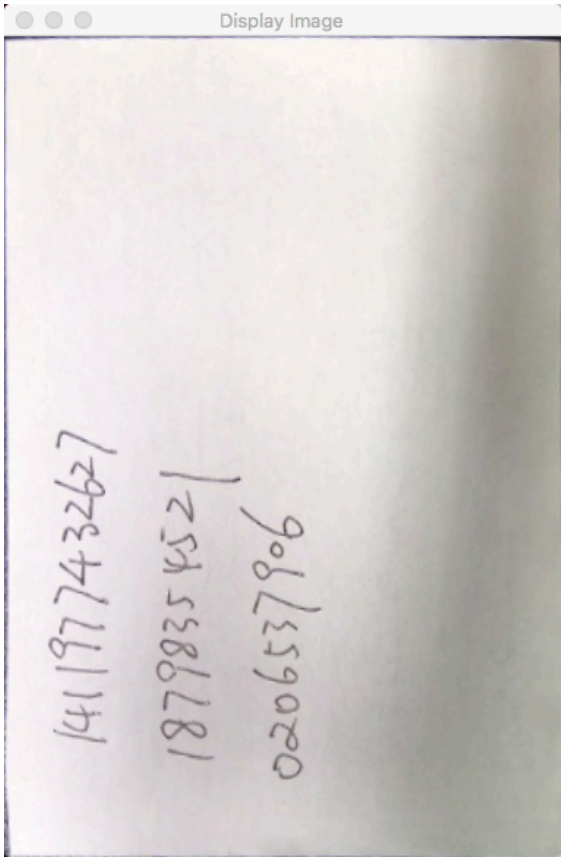
1.



2.

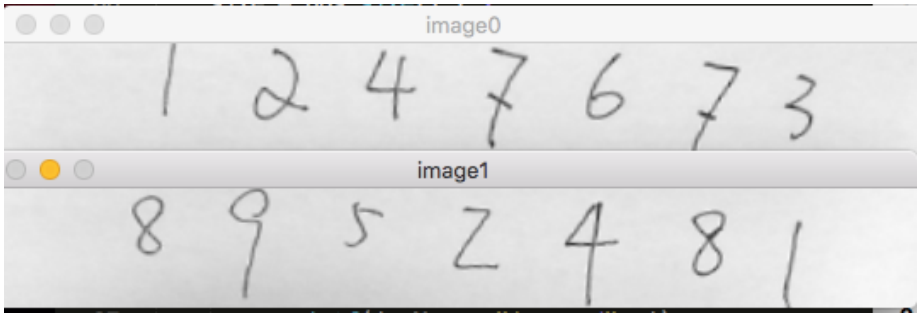


3.

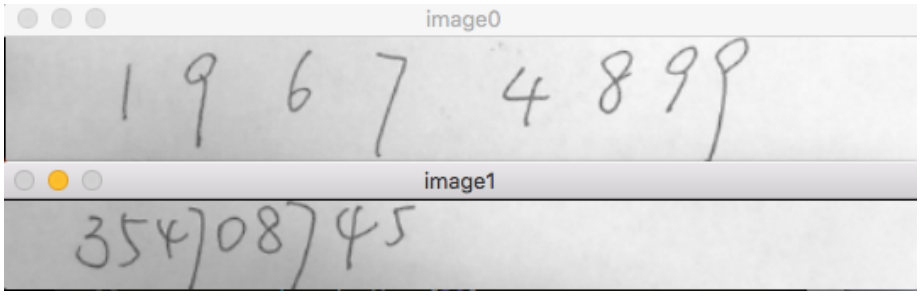


竖直切割：

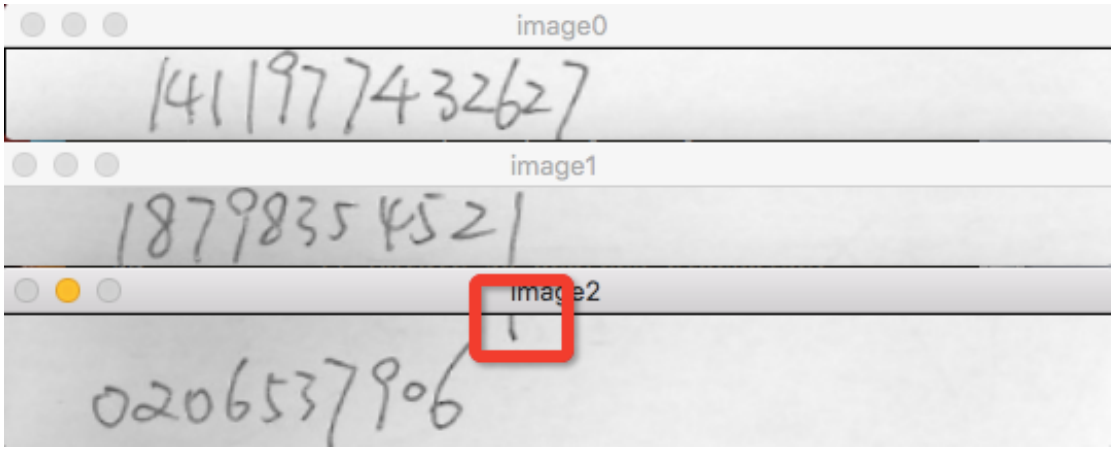
1.



2.

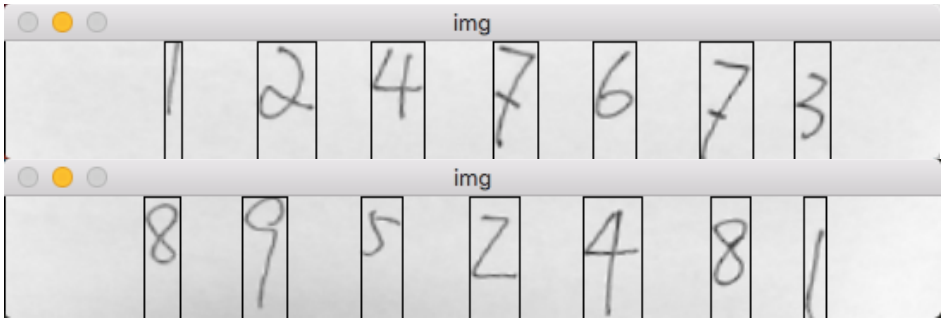


3.可见，由于重叠的问题，image2 中还是有 1 的部分痕迹：



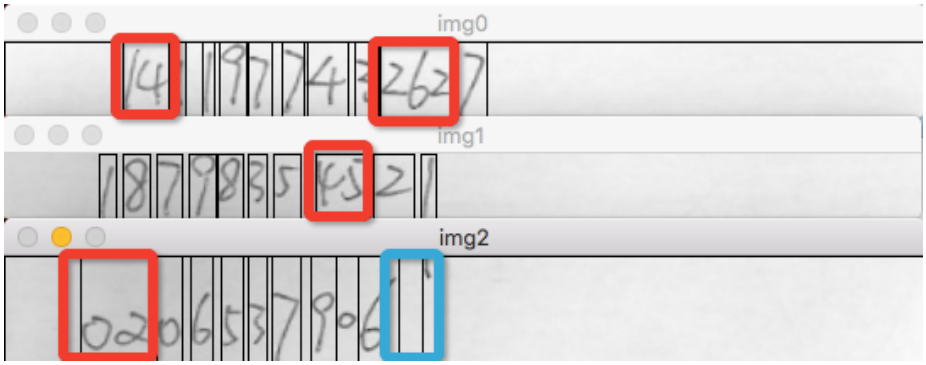
数字切割：

1.



2.

3.可见红色框住的部分都是因为数字挨的太近，导致无法分割成功。而蓝色框住的部分由于有噪声导致误判：



切割字符节选：

不正确切割字符：

预测结果一般情况（仅截取前两张图的数据）：

predictNormalFirst2.csv

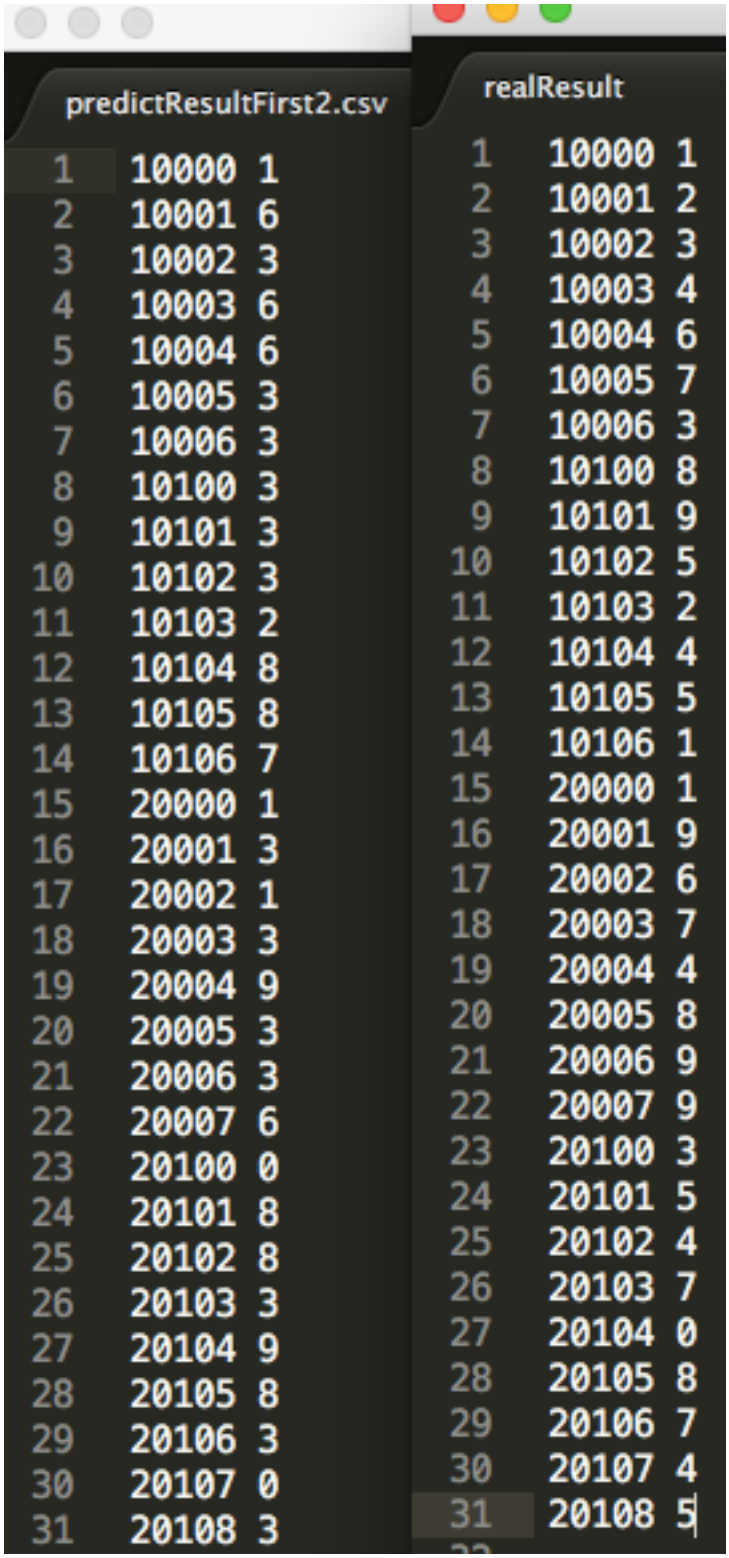
1	10000 5
2	10001 6
3	10002 3
4	10003 6
5	10004 4
6	10005 7
7	10006 1
8	10100 3
9	10101 3
10	10102 5
11	10103 2
12	10104 8
13	10105 5
14	10106 1
15	20000 1
16	20001 1
17	20002 5
18	20003 3
19	20004 1
20	20005 3
21	20006 5
22	20007 6
23	20100 7
24	20101 5
25	20102 1
26	20103 7
27	20104 5
28	20105 5
29	20106 7
30	20107 7
31	20108 1
32	

realResult

1	10000 1
2	10001 2
3	10002 3
4	10003 4
5	10004 6
6	10005 7
7	10006 3
8	10100 8
9	10101 9
10	10102 5
11	10103 2
12	10104 4
13	10105 5
14	10106 1
15	20000 1
16	20001 9
17	20002 6
18	20003 7
19	20004 4
20	20005 8
21	20006 9
22	20007 9
23	20100 3
24	20101 5
25	20102 4
26	20103 7
27	20104 0
28	20105 8
29	20106 7
30	20107 4
31	20108 5
32	

共有 31 张图片，预测正确 10 个，正确率 32%。

对图片进行膨胀操作的预测结果情况（仅截取前两张图的数据）：



predictResultFirst2.csv

```
1  10000 1
2  10001 6
3  10002 3
4  10003 6
5  10004 6
6  10005 3
7  10006 3
8  10100 3
9  10101 3
10 10102 3
11 10103 2
12 10104 8
13 10105 8
14 10106 7
15 20000 1
16 20001 3
17 20002 1
18 20003 3
19 20004 9
20 20005 3
21 20006 3
22 20007 6
23 20100 0
24 20101 8
25 20102 8
26 20103 3
27 20104 9
28 20105 8
29 20106 3
30 20107 0
31 20108 3
```

realResult

```
1  10000 1
2  10001 2
3  10002 3
4  10003 4
5  10004 6
6  10005 7
7  10006 3
8  10100 8
9  10101 9
10 10102 5
11 10103 2
12 10104 4
13 10105 5
14 10106 1
15 20000 1
16 20001 9
17 20002 6
18 20003 7
19 20004 4
20 20005 8
21 20006 9
22 20007 9
23 20100 3
24 20101 5
25 20102 4
26 20103 7
27 20104 0
28 20105 8
29 20106 7
30 20107 4
31 20108 5
```

共有 31 张图片，预测正确 7 个，正确率 22%，比不做膨胀处理的正确率低了 10%。

原因分析：

造成预测成功率较低的原因有以下几个方面：

1. 图像切割时的问题：

1.1. 图像无法完美切割，即部分像素点被排除在截取的图片之外，同时又有其他噪音加入到截取的图片中；

1.2. 图像切割时位置的问题，数字像素并不是集中在图像的中部，有一些集中在上半部分，而有一些又偏下；

2. 图像预测前处理的问题：

2.1. 图像缩小带来失真的问题；

2.2. 图像像素值与训练所用数据集内图像像素值差距过大的问题：尽管已经做到使得图片的样式和训练样本中图片的样式差不多，但是像素的值的分布还是有很大的差距的。训练集中的图像的值大多为 0 和 255，有少量的值是接近 0 或 255。但是截取出来的图片中，有部分像素值是 90-170 左右的中间值，这给之后的预测带来了很大的麻烦；

2.3. 数字粗细的问题，训练集中数字是相对较粗的，而我们截取出来的数字是相对较细的。但是经过一次核为 1 的膨胀之后，由于图像本身较小，整个数字的形态会发生巨大的变化，因此造成相应的问题；

3. 分类器的问题：

3.1. 由于分类器是运用 MNIST 数据集进行训练的，由于书写风格，环境等种种因素的影响，很难推广到其他手写体识别的预测集上；

综合考虑以上几个方面，我们可以提出几个方法来改进，提高预测的成功率：

1. 使用更好的切割方法，然后对图像进行平移等变换，使得数字位于图像的中间；

2. 对图像做更好的预处理，这方面不局限于膨胀等基本方法，也可以是其他相对高级的方法；

3. 利用推广性能更好的数据挖掘算法，训练出适用面更广的方法。