# 实验名称：A4 纸矫正与手写数字识别

## 实验要求：

**A4 纸矫正**：只能使用图像分割的办法获取 A4 纸的边缘。

**手写数字识别**：检测器使用 Adaboost 算法，可使用 Opencv 库

## 实验步骤：

**A4 纸矫正：**

1. 输入普通 A4 打印纸，使用图像分割的方法获取边缘。
2. 使用霍夫变换得到四个角点。
3. 对 A4 纸进行矫正，保留完整 A4 纸张。

**手写数字识别：**

1. 使用 Adaboost 算法实现手写数字检测器，用 MNIST 数据集进行训练、测试。
2. 自己在 A4 纸上书写单个数字进行识别。

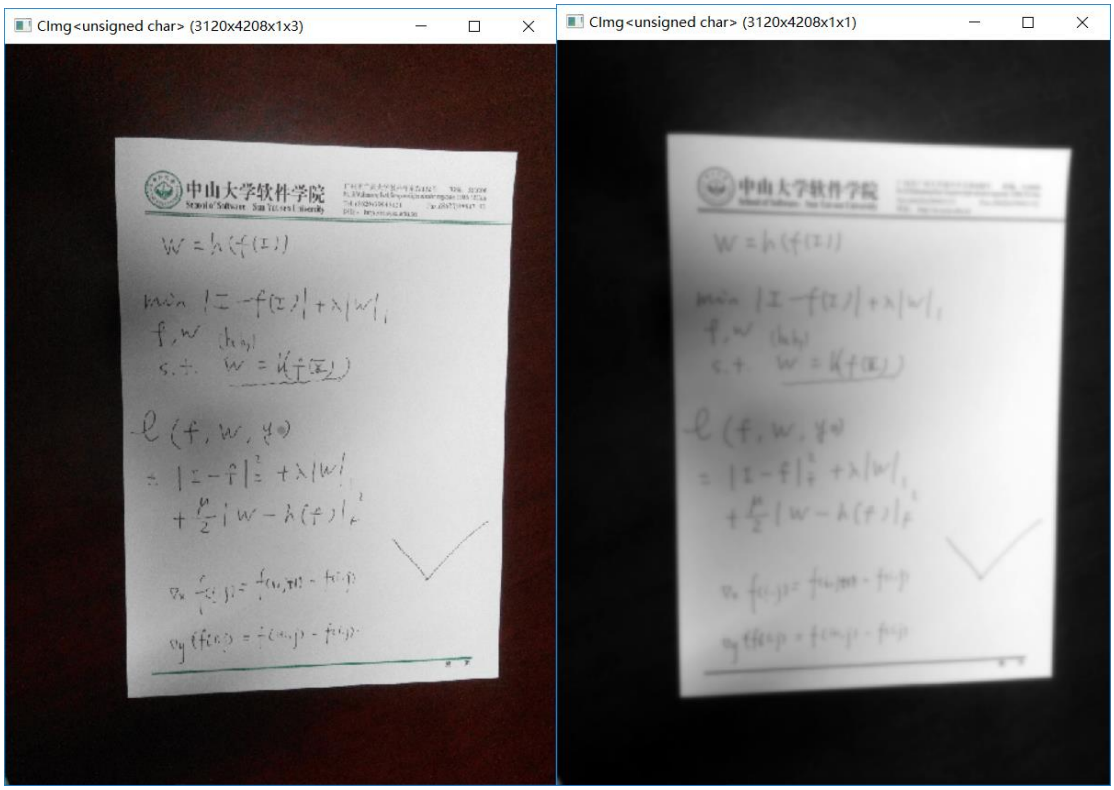
## 实验过程：

### A4 纸矫正

拿到一张普通 A4 打印纸，上面可能有手写的笔记或者打印内容，而且拍照角度可能是倾斜的，这时就需要对照片做 A4 纸矫正来得到完整的 A4 纸的内容。

首先需要对图像进行分割以获取边缘，这里选择的是区域生长的方法。不过要由区域生长得到正确的结果，首先要对图像进行预处理：转灰度图并高斯模糊，这是因为有的图像上有许多干扰的内容——A4 纸上的文字或者 A4 纸外的背景，进行高斯模糊可以使得这些干扰因素变得平滑，更易于进行区域生长。

预处理后的结果：




预处理之后就可以进行区域生长了，区域生长是一种比较古老的图像分割算法，其基本思想是将具有相似性质的像素集合起来构成区域，具体先对每个需要分割的区域找一个种子像素作为生长的起点，然后将种子像素周围邻域中与种子像素具有相同或相似性质的像素（根据某种事先确定的生长或相似准则来判定）合并到种子像素所在的区域中，将这些新像素当作新的种子像素继续进行上面的过程，直到再没有满足条件的像素可以被包括进来，这样一个区域就生长完成。

为了得到正确的结果，需要给区域生长设定几个条件：

1. 初始点（种子点的选取）
2. 生长准则
3. 终止条件

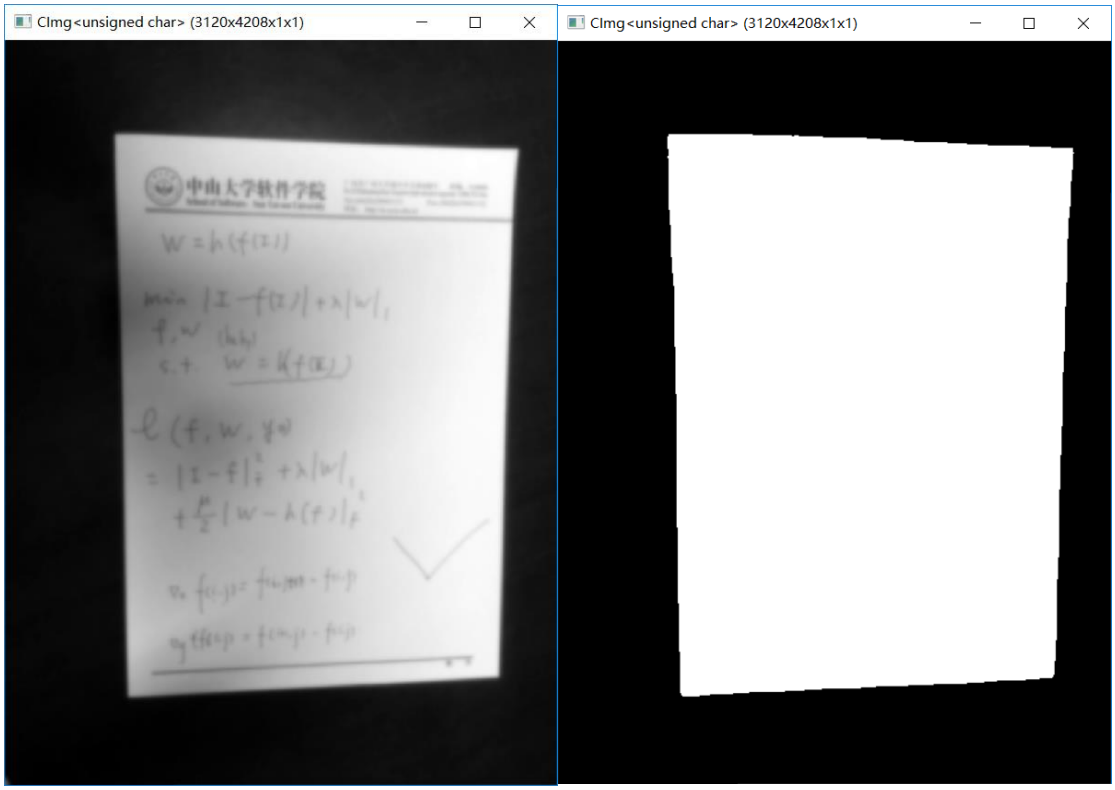
因为需要矫正的 A4 纸处于图像中心，所以初始种子点就选在图像的中心位置；在种子点处进行八邻域扩展，生长准则为灰度值大于某个值 T，此时视为相似并将该点加入准备处理的种子点队列中；终止条件为队列清空。

实现代码如下：

```
queue<point> seeds;
while(!seeds.empty()) {
    currentSeed = seeds.front();
    seeds.pop();
    x = currentSeed.x;
    y = currentSeed.y;
     // 遍历八邻域
    for (int i = -1; i < 2; ++i) {
        for (int j = -1; j < 2; ++j) {
```

```
            if(isInsideImage(x + i, y + j)){
                // 判断是否已经生长过
                if(grown(x + i, y + j) != 255){
                    // 生长条件的判断
                    if(gray(x + i, y + j) >= T){
                        // 新种子点入队
                        seeds.push(point(x + i, y + j));
                        // 将结果图像中对应坐标的灰度值设置为 255
                        grown(x + i, y + j) = 255;
                    }
                }
            }
        }
    }
}
```

区域生长结果:



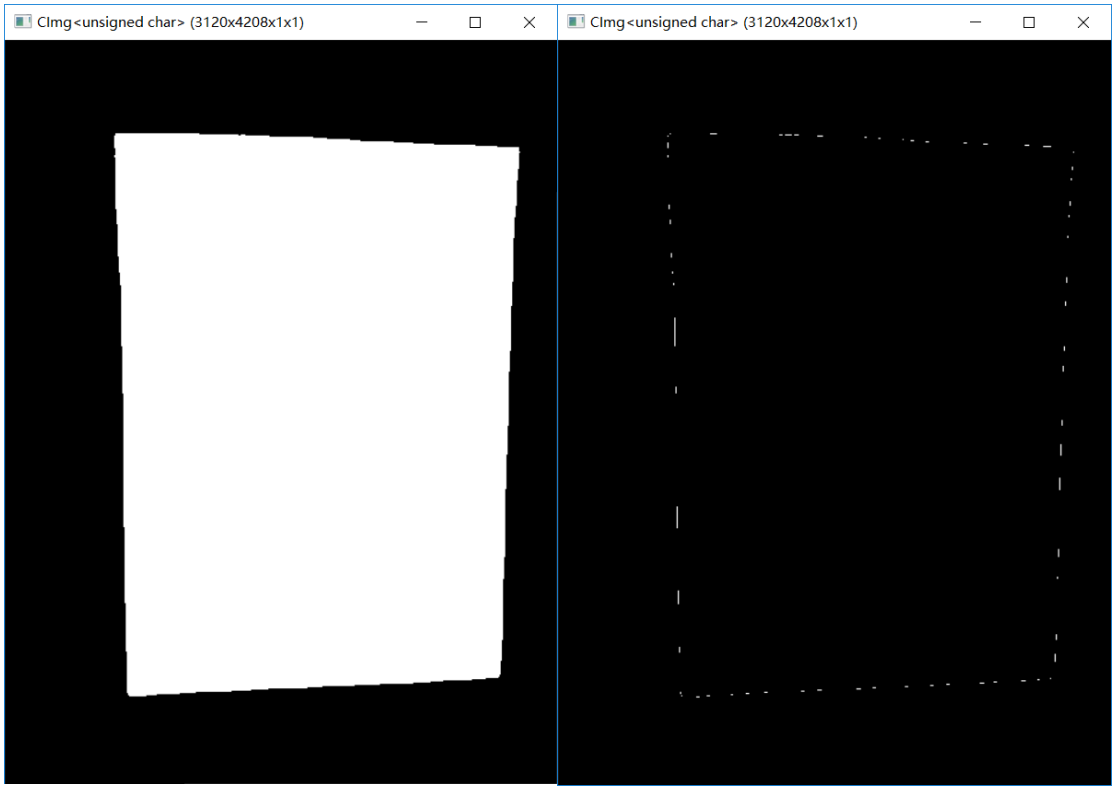

可以看到,区域生长的结果去除了其他的干扰因素,只留下唯一需要处理的 A4 纸部分,下一步要做的就是使用霍夫变换来得到 A4 纸的边缘(直线方程),进而得到四个角点。而通过之前的学习可以知道,霍夫变换的本质是对图像上的许多点进行投票,判断它们是否共线。那么这里就还需要进行一步处理才能够进行霍夫变换,可以看到区域生长的结果是整张 A4 纸,这一步处理就是只保留 A4 纸的边缘,去掉中间的多余部分。如果不进行这一步处理,

霍夫变换所花费的时间就会很久（对过多无用的点进行投票），最后得到的直线方程也不一定是边缘对应的直线方程。

提取 A4 纸的边缘同样需要进行八邻域搜索，我的方法是当且仅当一个像素的值为 255 且其八邻域中既有黑点也有白点时将其视为边缘点，最后只将这些边缘点的像素值设为 255，其他所有点的像素值都设为 0。代码如下：

```
cimg_forXY(grown, x, y){
    bool has_black_nei = false, has_white_nei = false;
    for (int i = -1; i < 2; ++i) {
        for (int j = -1; j < 2; ++j) {
            if(i == 0 && j == 0) continue;
            if(isInsideImage(x + i, y + j)){
                if(grown(x + i, y + j) == 255)
                    has_white_nei = true;
                if(grown(x + i, y + j) == 0)
                    has_black_nei = true;
            }
        }
    }
    if(grown(x, y) == 255 && has_black_nei && has_white_nei)
        result(x, y) = 255;
}
```

提取边缘结果（CImg 显示问题，实际为连续的线）：



接下来就可以进行霍夫变换了，识别直线是最简单的霍夫变换，它将一条直线从：

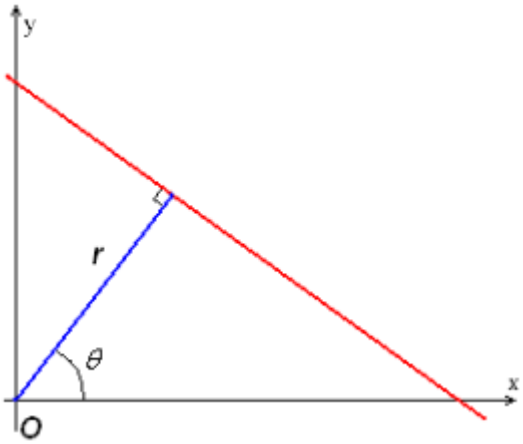
$$y = m_0 x + b_0$$

（视为（m, b）参数空间）

转到参数空间（r，θ）参数空间中（Hesse normal form）：

$$r = x\cos\theta + y\sin\theta$$

直线可表示为：

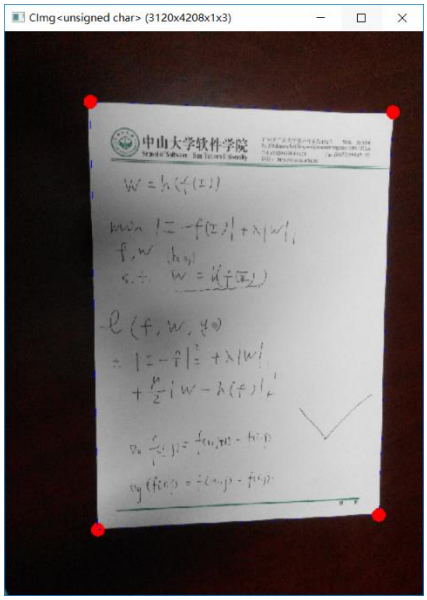



此时 r 是原点到直线的距离，θ是 r 和 x 轴的夹角。给定一个点，通过该点的所有直线的参数的集合会在（r，θ）平面上形成一个三角函数，因此判断边缘点是否共线的问题就转为判断一堆曲线是否在（r，θ）平面上相交于同一点的问题。

霍夫变换在前几次作业中已经实现，这里直接使用。对刚刚得到的边缘图像进行霍夫变换，得到的直线方程：

```
Line: y = -28.6363x + 83325
Line: y = -0.0524078x + 3749.14
Line: y = 57.29x + -35754.4
Line: y = 0.0349208x + 498.304
```

霍夫变换提取的角点：



由霍夫变换得到了 A4 纸的四个角点坐标，最后就到了 A4 纸的矫正部分。矫正部分使用了单应性矩阵，而要进行单应性变换必须先明确点与点的对应关系。通过霍夫变换得到的点的顺序是不确定的，这里必须对这些点进行排序。

将 A4 纸的四个角点 A、B、C、D 对应为左上、右上、右下、左下。则首先可以对得到的角点集合根据 y 坐标排序，y 坐标最小的一定为 A、B 中的一个（CImg 中 y 坐标越小对应像素点越靠上）：

```
sort(points.begin(), points.end(), [](pair<int, int> &a, pair<int, int>
&b){
    return a.second < b.second;
});
```

再对剩余的三个点求它们与第一个得到的点的距离。距离最短的点为 A、B 中的另一个：

```
int temp = 0;
for(int i = 1; i < points.size(); ++i){
    double dis = sqrt((points[0].first - points[1].first) *
(points[0].first - points[1].first) + (points[0].second -
points[1].second) * (points[0].second - points[1].second));
    if(dis < min){
        min = (dis < min) ? dis : min;
        temp = i;
    }
}
swap(points[1], points[temp]);
```

剩下的两个点则为 C、D。再根据点的 x 坐标值就可以明确 A、B、C、D 了：

```
if(points[0].first > points[1].first){
    swap(points[0], points[1]);
}
if(points[2].first < points[3].first){
    swap(points[2], points[3]);
}
```

A、B、C、D 对应于矫正后的坐标：

```
dst.push_back(make_pair(0, 0));
dst.push_back(make_pair(width - 1, 0));
dst.push_back(make_pair(width - 1, height - 1));
dst.push_back(make_pair(0, height - 1));
```

得到四对对应角点后就可以求解得到单应性矩阵的 8 个自由度（用到了之前作业 ransac

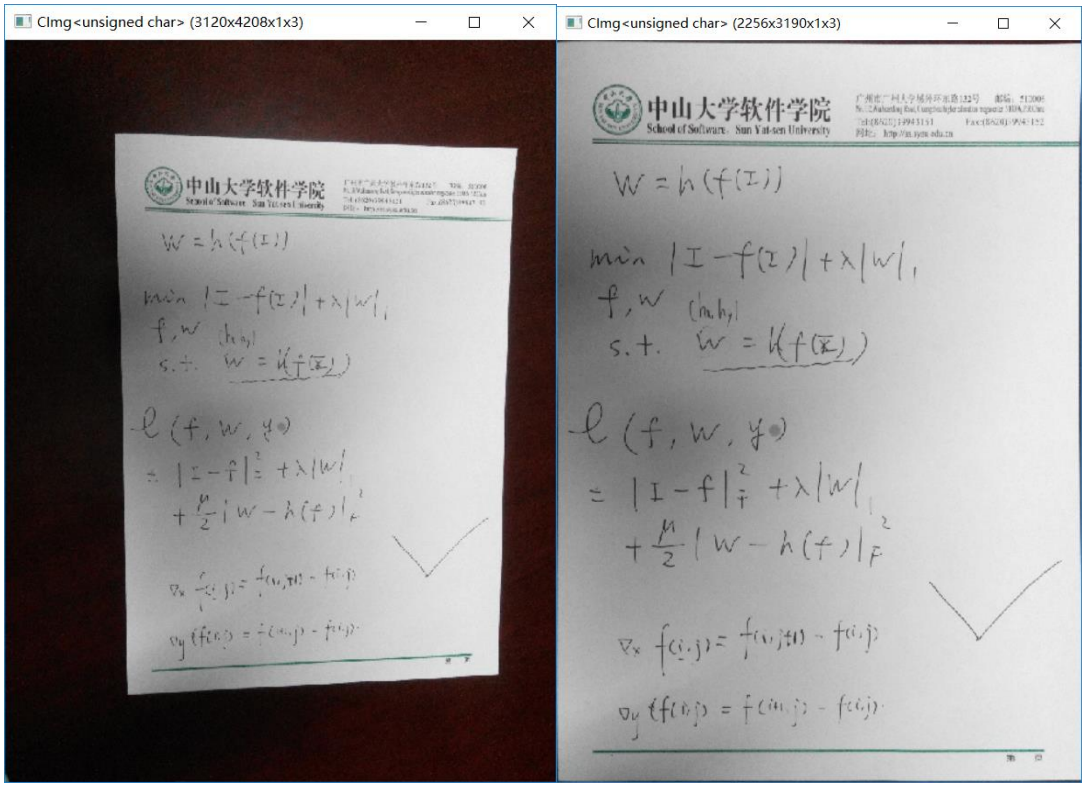
时单应性矩阵的计算)：

```
Homoparam Correct::getHomography(){
    CImg<double> A(4, 4, 1, 1, 0);
    CImg<double> b(1, 4, 1, 1, 0);
    //calc Ax = b
    for(int i = 0; i < 4; ++i){
        A(0, i) = dst[i].first;
        A(1, i) = dst[i].second;
        A(2, i) = dst[i].first * dst[i].second;
        A(3, i) = 1;
        b(0, i) = points[i].first;
    }
    CImg<double> x1 = b.get_solve(A);
    for(int i = 0; i < 4; ++i){
        b(0, i) = points[i].second;
    }
    CImg<double> x2 = b.get_solve(A);
    return Homoparam(x1(0,0), x1(0,1), x1(0,2), x1(0,3), x2(0,0),
x2(0,1), x2(0,2), x2(0,3));
}
```

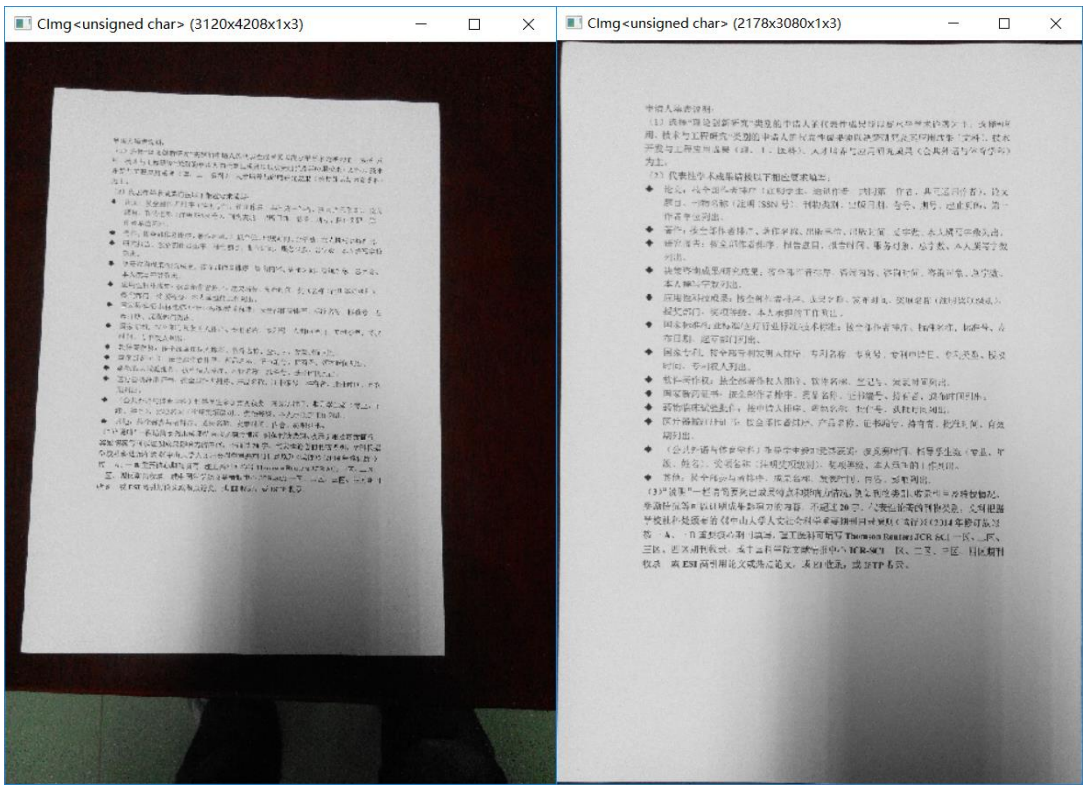
而后对图像上的每个像素点：由 8 个自由度计算得到新的像素点位置，并进行 bilinear 插值获取像素值：

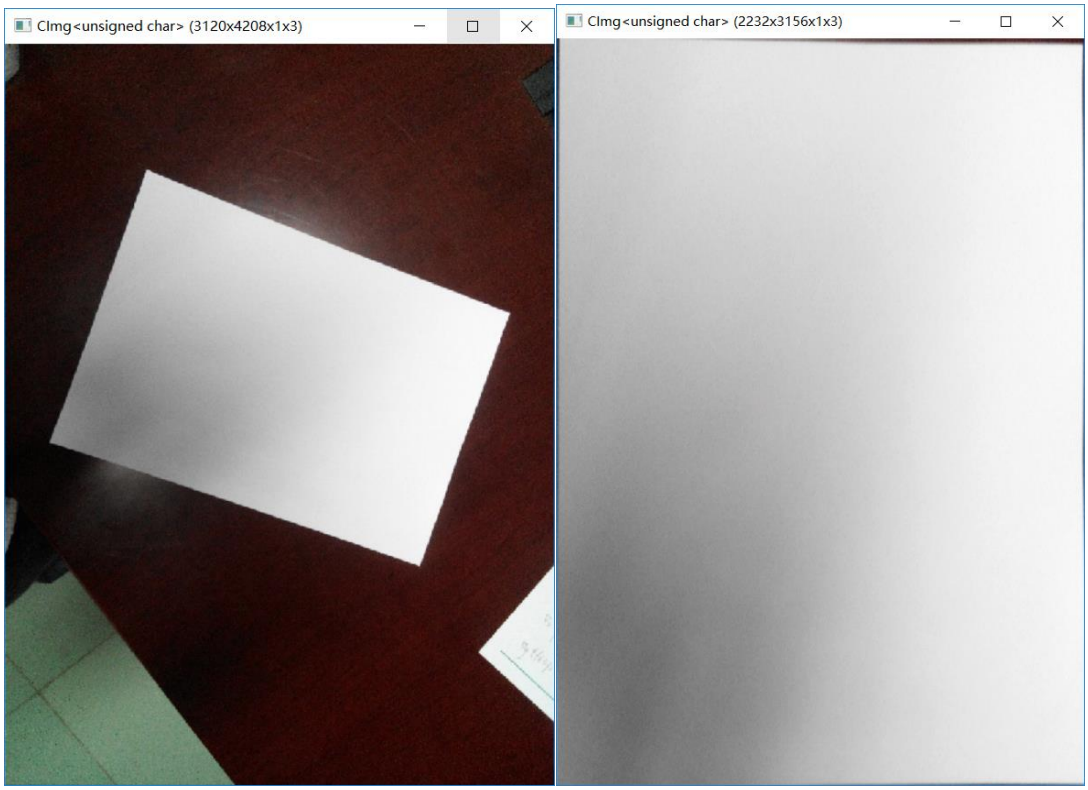
```
void Correct::bilinear(int x, int y, double tx, double ty){
    int x_pos = floor(tx);
    double x_u = tx - x_pos;
    int xb = (x_pos < img.width() - 1) ? x_pos + 1 : x_pos;

    int y_pos = floor(ty);
     double y_v = ty - y_pos;
    int yb = (y_pos < img.height() - 1) ? y_pos + 1 : y_pos;

    for(int i = 0; i < 3; ++i){
        double P1 = img(x_pos, y_pos, i) * (1 - x_u) + img(xb, y_pos, i) *
x_u;
        double P2 = img(x_pos, yb, i) * (1 - x_u) + img(xb, yb, i) * x_u;
        res(x, y, i) = P1 * (1 - y_v) + P2 * y_v;
    }
}
```

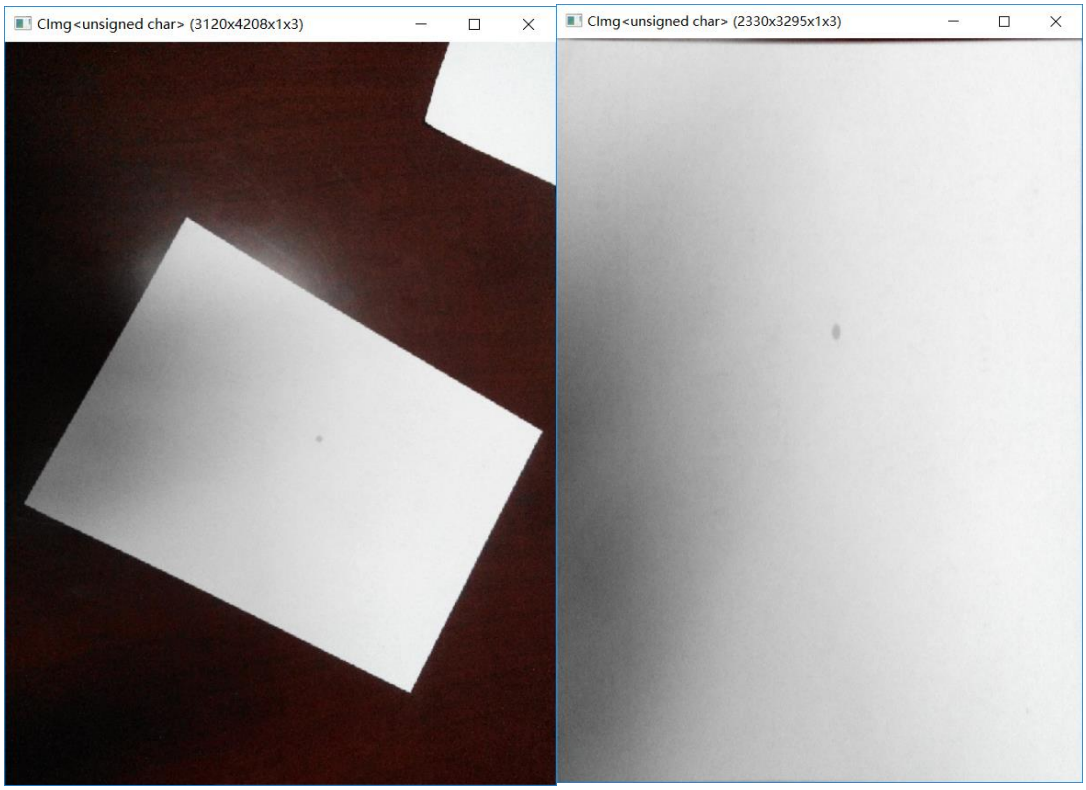
矫正结果:

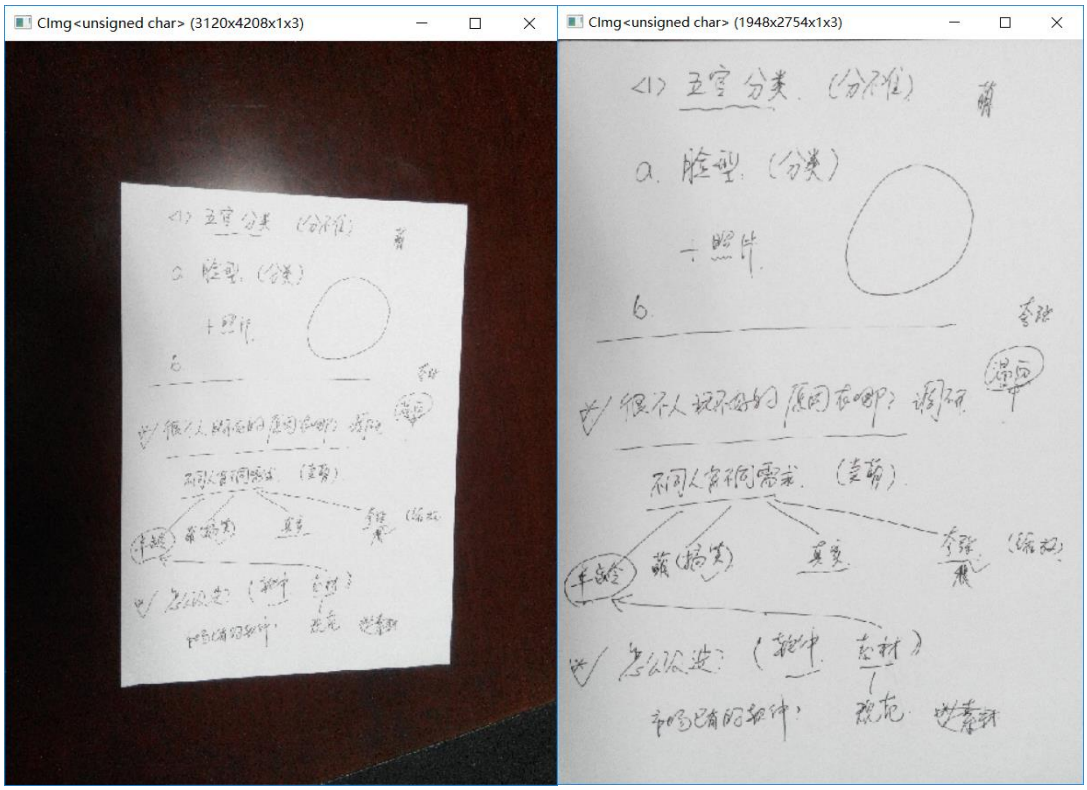


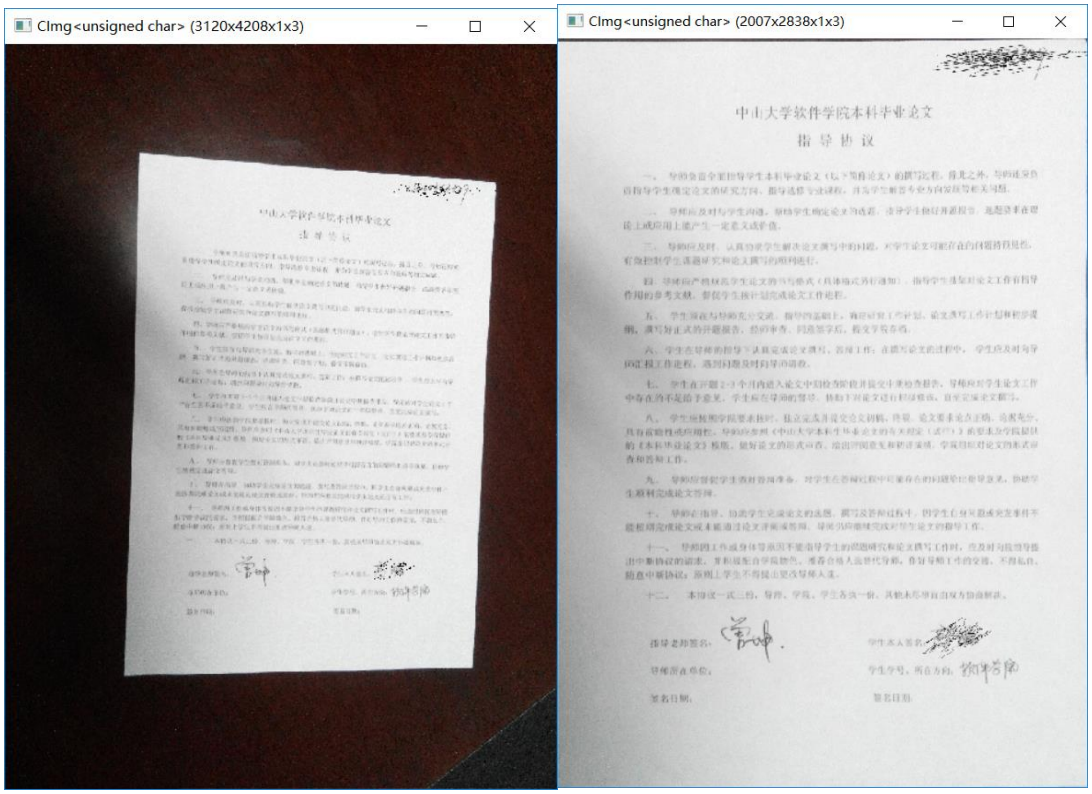

更多的矫正结果:




CImg<unsigned char> (3120x4208x1x3)
CImg<unsigned char> (2232x3156x1x3)


CImg<unsigned char> (3120x4208x1x3)
CImg<unsigned char> (2330x3295x1x3)




相对来说，区域生长法计算简单，对于 A4 纸这种较均匀的连通目标有较好的分割效果。但它需要人为设定初始种子点，而且如果是比较复杂的场景的分割或者对自然景物的分割，

区域生长法的效果就没那么好了（先验知识不足）。而之前使用的 Canny 算子对边缘的检测效果也不错，可以用来处理比较复杂的场景的分割。只是若有噪点过多的情况，canny 算子易受影响，所以需要预先降噪。

# 手写数字识别

此次实现的 Adaboost 手写数字检测器需要使用 MNIST 数据集来训练与测试，所以首先必须正确读取 MNIST 数据集的内容。MNIST 数据集包含训练集与测试集，其中每个集合中又分为图像文件与标签文件，其类型如下：

训练集标签：

```
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
```

训练集图像：

```
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
```

测试集的图像与标签的格式与上面的相同，其中 label 的取值为 0-9，每个图像的大小为 28×28。注意到图像与标签除了实际数据外还有一些 header 数据，这些数据决定了后面实际数据的项数，所以要先正确的读取这些 header。将 header 信息读取到结构体中：

标签：

```
struct MNISTLabelFileHeader {
    unsigned char magicNumber[4];
    unsigned char numberOfLabels[4];
};
```

图像：

```
struct MNISTImageFileHeader {
    unsigned char magicNumber[4];
    unsigned char numberOfImages[4];
    unsigned char numberOfRows[4];
    unsigned char numberOfCols[4];
};
```

实际读取时，首先根据读到的header来判断是否为标签或图像数据，再根据numberOfLabels/numberOfImages来读取剩下的信息，将剩下的信息读取到opencv的Mat数据类型中，以便后续处理：

```
cv::Mat readData(std::fstream& dataFile, int numberOfData, int
dataSizeInBytes) {
    Mat dataMat;
    if (dataFile.is_open()) {
        int dataSize = dataSizeInBytes * numberOfData;
        unsigned char *tmpData = new unsigned char[dataSize];
        dataFile.read((char *)tmpData, dataSize);
        dataMat = Mat(numberOfData, dataSizeInBytes, CV_8UC1,
tmpData).clone();
        delete[] tmpData;
        dataFile.close();
    }
    return dataMat;
}
```

经过读取，得到两个Mat：图像集与标签集，为了后面的训练，需要先将两个Mat的类型分别转为CV_32FC1与CV_32SC1，然后声明一个boosting model：

```
Ptr<Boost> model;
```

这里的Boost实际上就是AdaBoost，opencv提供了四种类型的boosting：

- CvBoost::DISCRETE（discrete AdaBoost）
- CvBoost::REAL（real AdaBoost）
- CvBoost::LOGIT（LogitBoost）
- CvBoost::GENTLE（gentle AdaBoost）

这些都是由原始AdaBoost变换而来的，其中real Adaboost与gentle Adaboost的效果最好，real Adaboost利用置信区间预测，在标签数据上有很好的性能。Gentle Adaboost对outlier data赋较小的值，所以在回归问题上效果很好。这里经过测试选择的是gentle Adaboost。

遗憾的是，Opencv提供的Boosting算法只支持二分类，但手写数字识别需要十个类：0到9，所以这里将训练样本“展开”，实际得到十种二分类器来进行分类：

```
Mat new_data(ntrain_samples * class_count, var_count + 1, CV_32FC1);
Mat new_responses(ntrain_samples * class_count, 1, CV_32SC1);
cout << "Unrolling the database..." << endl;
for (int i = 0; i < ntrain_samples; ++i) {
    const float *data_row = data.ptr<float>(i);
    for (int j = 0; j < class_count; ++j) {
        float *new_data_row = (float *)new_data.ptr<float>(i *
class_count + j);
        memcpy(new_data_row, data_row, var_count * sizeof(data_row[0]));
        new_data_row[var_count] = (float)j;
        new_responses.at<int>(i * class_count + j) =
responses.at<int>(i) == j + 0;
    }
}
```

在进行训练前，还要设置一些boosting的参数：

```
model = Boost::create();
model->setBoostType(Boost::GENTLE);
model->setWeakCount(95);
model->setWeightTrimRate(0.95);
model->setMaxDepth(5);
model->setUseSurrogates(false);
model->setPriors(Mat(priors));
```

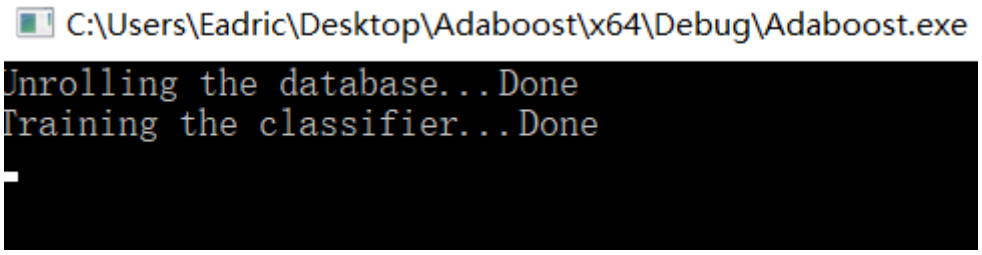
其中：

- setWeakCount：设置弱分类器的数量
- setWeightTrimRate：阈值，影响参与训练的样本，样本权重更新排序后（从小到大），从前面累计权重小于（1-weightTrimRate）的样本将不参与下一次训练，可加快计算速度
- setUseSurrogates：是否建立替代分裂点
- setPriors：先验类概率数组

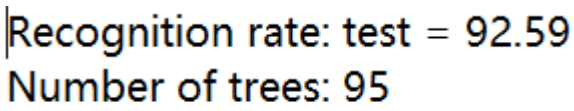
创建训练数据，并开始训练：

```
Ptr<TrainData> tdata = TrainData::create(new_data, ROW_SAMPLE,
new_responses, noArray(), noArray(), noArray(), var_type);
model->train(tdata);
```

训练：




这里设置了对训练集中所有60000个数据进行训练，且弱分类器设置为95个，训练时间比较慢，需要几十分钟。训练完后对测试集的数据进行测试，记录预测正确率：




预测准确率为92.59%，可见对测试数据的识别正确率还是比较高的。

接下来开始对在A4纸上手写的数字做识别，为此首先要使用图像分割的方法得到A4纸上书写的一个个数字，这里使用的是投影分割法。

首先对输入的图像在水平方向上投影（输入的图像已二值化），记录图像每一行的像素值为255的点的个数：

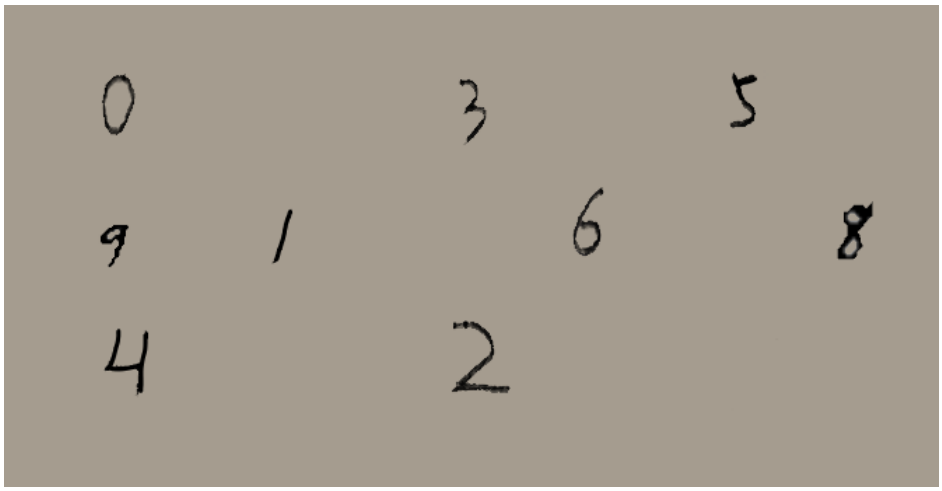
```
horizontal = new int[thres.height()]{ 0 };
for (int j = 0; j < thres.height(); ++j) {
    for (int i = 0; i < thres.width(); ++i) {
        if (thres(i, j) == 255) {
            ++horizontal[j];
        }
    }
}
```

然后在一张新的图像上根据之前每行的计数为像素点赋值，画出投影图：

```
bool spliting = false;
CImg<unsigned char> dis(thres.width(), thres.height(), 1, 1, 0);
for (int j = 0; j < thres.height(); ++j) {
    if (horizontal[j] > 0) {
        if (spliting == false) {
            spliting = true;
            to_split_hor.push_back(j - 5);
        }
        if (horizontal[j + 1] == 0) {
            to_split_hor.push_back(j + 5);
        }
    }
```

```
        else {
            spliting = false;
        }
        for (int i = 0; i < horizontal[j]; ++i) {
            dis(i, j) = 255;
        }
    }
```

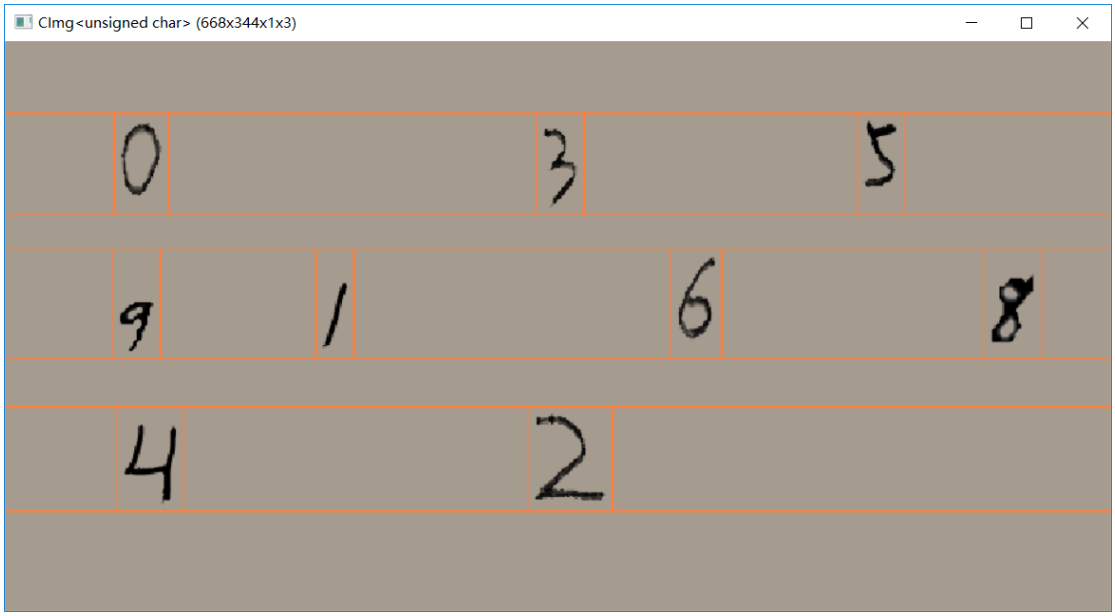
以这幅图来说：



其水平方向投影结果为：

可见投影结果分成了三个部分，代表数字只处在这三个区域，这样就可以将原图像分为三行。之后再以同样的方法对三行分别做垂直方向上的投影，

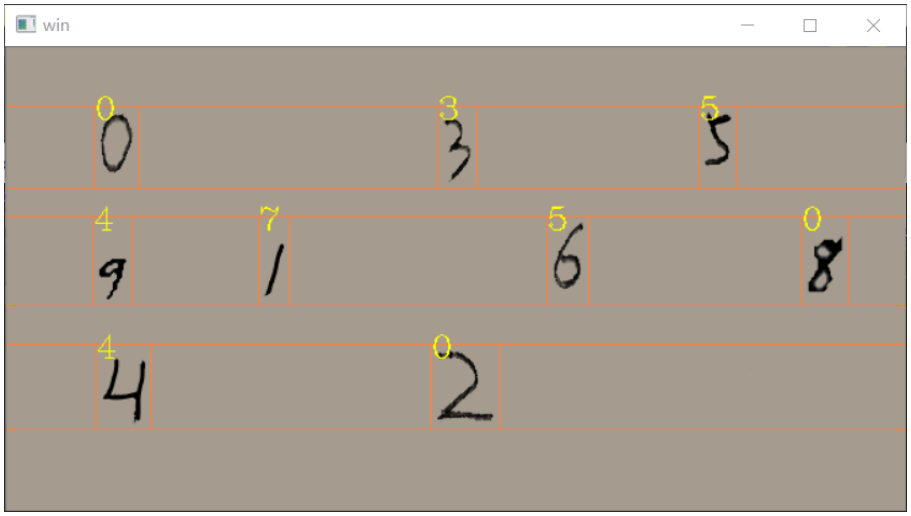
与图像上的数字是一一对应的。根据投影过程中记录的坐标信息，就可以将图像分成一个个的区域，得到所有单个的数字：



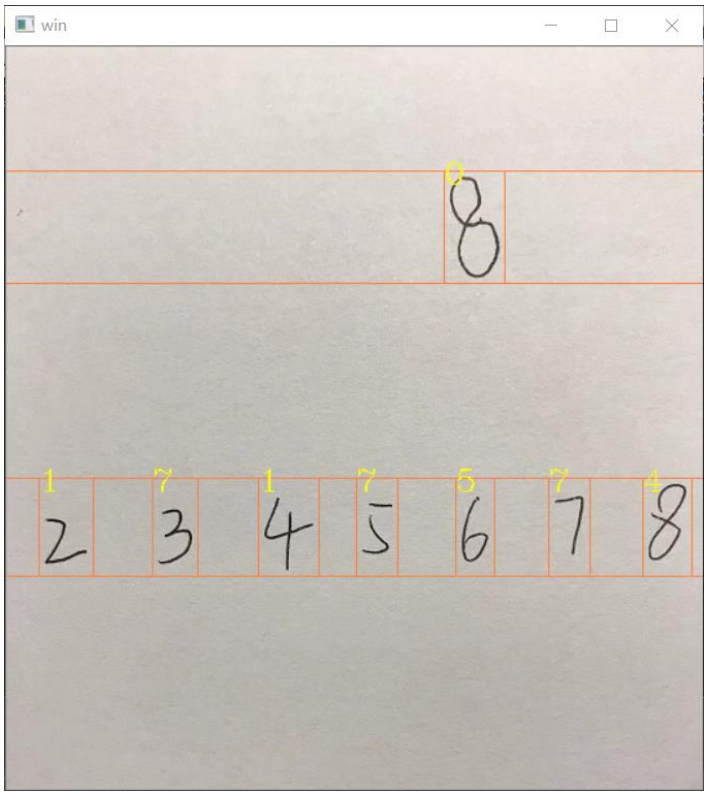

得到的单个数字：

当然，这些数字还不能直接用来测试，需要先将它们resize到28×28的大小（与训练时的图像数据相同），否则无法正确预测，resize后：

将得到的图像转为1×784的Mat格式，并进行预测，预测后将结果输出到图像上：

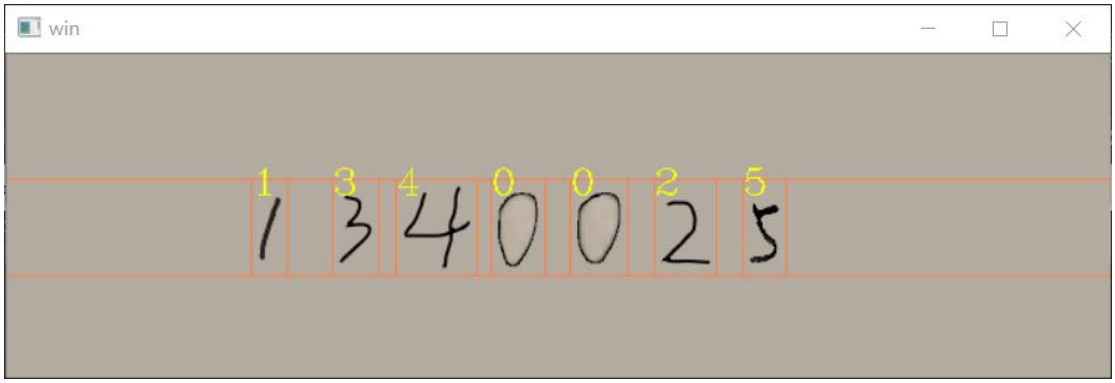


结果并不准确！再测试一次：




程序对A4纸上手写数字的预测结果并不正确，但对MNIST的测试集进行预测时，正确率却达到了92%，为什么会有如此大的差别呢？

将MNIST测试集的图像数据解析出来后，我发现它其中的手写数字有很多种风格，我模仿其中几个数字再一次进行手写并预测：




测试结论：由MNIST训练集训练得到的Adaboost分类器对MNIST数据集中的数据识别效果很好，但对普通手写数字的识别效果不尽人意。