作业七验报告及测试文档

数据科学与计算机学院 梁育诚 16340133

一、实验内容

第一部分：

输入图像：普通 A4 打印纸，上面可能有手写笔迹或者打印内容，但是拍照时可能角度不正。（数据采用作业 3 的数据）。

输出：已矫正好的标准普通 A4 纸（长宽比为 A4 纸的比例），并裁掉无用的其他内容，只保留完整 A4 纸张。

只能采用图像分割的办法获取 A4 纸的边缘，并对 A4 纸做矫正，并对比前面用 Canny 算子获取图像边缘的算法，分析两者的优劣。

第二部分：

用 Adaboost 实现手写体数字的检测器；然后针对测试数据进行测试（含手写输入图片）

二、实验原理

第一部分：

图像分割是指将数字图像细分为多个图像子区域（亚像素）的过程。图像分割的目的是简化或者改变图像的表达形式，通过将图像分成互不重叠的区域，使得我们能够提取出感兴趣的目标。在整个计算机视觉处理的过程中，图像分割处于中层视觉地位，它是目标表达的基础，使得更高层的图像分析和理解成为可能。分割的方式有多种，这里我用到了 Otsu 阈值法。

Otsu 法按图像的灰度特性将图像分为背景和前景（目标）两部分，背景和目标之间的类间方差越大，说明构成整个图像的两部分的差别越大。因此，使得类间方差最大的分割意味着错分概率最小，也说明该分割越准确。

Otsu 法的基本步骤如下：

1. 记 t 为目标与背景的分割阈值，这个是我们要求的。目标点数占图像比例为 w0，平均灰度为 u0；背景点数占图像比例为 w1，平均灰度为 u1。则图像的总平均灰度 u = w0*u0+w1*u1。

2. 从最小灰度值到最大灰度值遍历 t，我们要找到一个 t，使得 $g = w_0 \times (u_0 - u)^2 + w_1 \times (u_1 - u)^2$ 最大。此时 t 就称为该分割的最佳阈值。

3. 应用时为了计算方便，使用的是等价公式 $g = w_0 \times w_1 \times (u_0 - u_1)^2$。

完成图像分割后，我们就可以进行边缘检测了。图像分割将目标很明显地分割出来，因此检测边缘的时候，只需要对图像求梯度。这里用到了 sobel 算子对图像求梯度。最后求出的每个像素点的梯度（包括 x 方向和 y 方向）。然后在梯度大于一定阈值的地方就是边缘了。在技术上，sobel 算子是离散型的差分算子，用于求图像亮度的梯度近似值，因此它是基于一阶导数的算子。它主要的好处就是对像素周围的位置进行了加权计算。

接着是检测 A4 纸的边缘。这里用了之前实现的霍夫变换，原来不再赘述。

最后是 A4 纸矫正。A4 纸的矫正实际上是一个透视变换，是一块区域到另一

块区域的变换。透视变换能够将变形的 A4 纸恢复到正常形状的 A4 纸，主要难点在于求解出透视变换的变换矩阵，这是计算部分的内容，在后面介绍。

第二部分：

Adaboost 算法是一种典型的 Boosting 算法，其核心思想很简单，针对同一个训练集构建不同的弱分类器，然后按照一定方法来将这些弱分类器集合起来，构成一个强分类器。训练数据有两个，一个是图片数据，一个是与之对应的标签数据。每一轮学习，根据每个样本的分类正确与否，来更新每个样本的权重，正确的样本权重减少，错误的样本权重增加。最后就能够得到一个具有不错识别能力的强分类器了。

三、实验过程

第一部分：

1. 首先实现图像分割。使用 Otsu 阈值法，需要先将图片进行灰度化处理，然后获取图像的灰度直方图，根据灰度直方图的分布找出最佳分割阈值。

```
11    CImg<float> img((root + to_string(i) + ".bmp").c_str());
12    CImg<float> grayImg(img._width, img._height, 1, 1);
13
14    // Change to Grey Image
15    cimg_forXY(img, x, y) {
16      grayImg(x, y, 0, 0) = 0.299 * img(x, y, 0, 0) + 0.587 * img(x, y, 0, 1) + 0.114 * img(x, y, 0, 2);
17    }
```

```
28  // Compute the histogram of the image
29  void Segmentation::getHistogram() {
30    cimg_forXY(img, x, y) {
31      histogram[(int)img(x, y)]++;
32    }
33  }
```

```
35  // Do segmentation
36  void Segmentation::segment() {
37    int totalPixels = img._width * img._height;
38    int gray = 0, grayBackground = 0, grayFrontground = 0;
39    float wBackground = 0, wFrontground = 0;
40    float g_best = 0;
41    float g = 0;
42    int numBackground = 0, numFrontground = 0;
43
44    getHistogram();
45    for (int i = 0; i < 256; i++) {
46      gray += i * histogram[i];
47    }
48
49    for (int i = 0; i < 256; i++) {
50      grayBackground += histogram[i] * i; // 背景总灰度值
51      grayFrontground = gray - grayBackground; // 前景总灰度值
52      numBackground += histogram[i]; // 背景pixel数量
53      numFrontground = totalPixels - numBackground; // 前景pixel数量
54      wBackground = numBackground * 1.0 / totalPixels; // 背景pixel占比
55      wFrontground = 1 - wBackground; // 前景pixel占比
56
57      // 无法区分前景和背景
58      if (wBackground == 0 || wFrontground == 0)
59        continue;
```

```
    // 找出最佳阈值
    g = wBackground * wFrontground * (grayBackground / numBackground - grayFrontground / numFrontground)
          * (grayBackground / numBackground - grayFrontground / numFrontground);
    if (g > g_best) {
      threshold = i;
      g_best = g;
    }
  }
  cout << "Threshold: " << threshold << endl;
}
```

2. 得到最佳分割阈值后，遍历整个图像，将小于阈值的像素设为 255（白色），大于阈值的像素设为 0（黑色）。

```
// Get Binary Image
void Segmentation::binarization(string path) {
  result = CImg<float>(img._width, img._height, 1);
  cimg_forXY(img, x, y) {
    if (img(x, y) > threshold) {
      result(x, y, 0, 0) = 0;
    }
    else {
      result(x, y, 0, 0) = 255;
    }
  }
}
```

3. 然后对图像计算梯度。利用 sobel 算子求出该图像在 x 方向和 y 方向上的梯度。这里可以用 CImg 库提供的 get_convolve 函数实现（其实就是卷积）。然后对于梯度空间，同样使用一个阈值进行过滤，梯度高于该阈值的点就认为是边缘，梯度低于该阈值的点就认为是背景。，因为边缘的一边是黑色，一边是白色，所以我们认为只有边缘的点梯度才是最大的。如果以 0 为分界，则会包含过多的噪声，因此这里的阈值手动设为 30。

```
// Detect Edge
void Segmentation::getEdge() {
  // using sobel to compute gradient
  CImg<float> sobelx(3, 3, 1, 1, 0);
  CImg<float> sobely(3, 3, 1, 1, 0);
  // Initialize sobel
  sobelx(0, 0) = -1, sobely(0, 0) = 1;
  sobelx(0, 1) = 0, sobely(0, 1) = 2;
  sobelx(0, 2) = 1, sobely(0, 2) = 1;
  sobelx(1, 0) = -2, sobely(1, 0) = 0;
  sobelx(1, 1) = 0, sobely(1, 1) = 0;
  sobelx(1, 2) = 2, sobely(1, 2) = 0;
  sobelx(2, 0) = -1, sobely(2, 0) = -1;
  sobelx(2, 1) = 0, sobely(2, 1) = -2;
  sobelx(2, 2) = 1, sobely(2, 2) = -1;

  CImg<float> gradient_x = result;
  gradient_x = gradient_x.get_convolve(sobelx);
  CImg<float> gradient_y = result;
  gradient_y = gradient_y.get_convolve(sobely);

  cimg_forXY(result, i, j) {
    double grad = sqrt(gradient_x(i, j) * gradient_x(i, j) + gradient_y(i, j) * gradient_y(i, j));
    if (grad > gradient_threshold) {
      result(i, j) = EDGE;
    }
    else {
      result(i, j) = NOEDGE;
    }
  }
  result.display("edge", false);
}
```

4. 然后就是基于图像分割后的图像进行 A4 纸检测，使用霍夫变换，代码用回之前作业自己写的，效果不错。修改部分参数，邻域大小设置为 60，因为不需要在原图显示检测出来的直线，因此可以删减部分代码。注意画线的时候要重新用另一个 CImg，原来的 CImg 是灰度图，无法画彩色的线，不然会越界报错。

```cpp
// Initialize Hough Space
void Segmentation::initHoughSpace() {
  int maxp = (int)sqrt(pow(result._width / 2, 2) + pow(result._height / 2, 2));
  hough_space = CImg<float>(maxp, 360);
  hough_space.fill(0);

  cimg_forXY(result, x, y) {
    int value = (int)result(x, y);
    int p;
    // Only care the edge points
    if (value == EDGE) {
      // transform to polar coordinate (p, i) => (length, angle)
      int x0 = x - rows / 2;
      int y0 = columns / 2 - y;
      // Calculate all lines gone through a point(all angles)
      for (int i = 0; i < 360; i++) {
        p = x0 * coss[i] + y0 * sins[i];
        // Voting
        if (p >= 0 && p < maxp) {
          hough_space(p, i)++;
        }
      }
    }
  }
}
```

```cpp
// Detect Line from houghspace
void Segmentation::lineDetect() {
  cout << "Begin detect" << endl;
  int wid = hough_space._width;
  int hei = hough_space._height;
  int maxVotes;
  for (int i = 0; i < wid; i += PART / 2) {
    for (int j = 0; j < hei; j += PART / 2) {
      maxVotes = getPartMax(hough_space, i, j);
      for (int x = i; (x < i + PART) && (x < wid); x++) {
        for (int y = j; (y < j + PART) && (y < hei); y++) {
          if ((int)hough_space(x, y) < maxVotes) {
            hough_space(x, y) = 0;
          }
        }
      }
    }
  }
  cout << "finish detect" << endl;
  //hough_space.display("detected", false);
  // save the bright points in houghspace
  cimg_forXY(hough_space, x, y) {
    if ((int)hough_space(x, y) != 0) {
      lines.push_back(make_pair(x, y));
      lineCount.push_back((int)hough_space(x, y));
    }
  }
}
```

```cpp
// Find local maximum
int Segmentation::getPartMax(CImg<float>& img, int &x, int &y) {
  int wid = (x + PART > img._width) ? img._width : x + PART;
  int hei = (y + PART > img._height) ? img._height : y + PART;
  int maxVotes = 0;
  for (int i = x; i < wid; i++) {
    for (int j = y; j < hei; j++) {
      maxVotes = ((int)img(i, j) > maxVotes) ? ((int)img(i, j)) : maxVotes;
    }
  }
  return maxVotes;
}
```

```
// draw the outline of A4
void Segmentation::drawLines() {
  cout << "Begin drawLines" << endl;
  int maxLength;
  maxLength = sqrt(pow(rows / 2, 2) + pow(columns / 2, 2)); // width of houghspace

  showEdge = CImg<float>(rows, columns, 1, 1, 0); // Initialize showEdge
  afterHough = CImg<float>(rows, columns, 1, 3, 0); // Initialize output image with original image

  sortedLineCount = lineCount; // Initialize the vector

  // Sort lines by its votes, descending
  sort(sortedLineCount.begin(), sortedLineCount.end(), greater<int>());

  // Show the number of decteced size
  cout << "Size: " << sortedLineCount.size() << endl;

  // Record the parameters of the lines
  vector<pair<int, int>> points;
  for (int i = 0; i < 4; i++) {
    int weight = sortedLineCount[i];
    int indexInLines; // Index in lines vector
    vector<int>::iterator it;
    it = find(lineCount.begin(), lineCount.end(), weight);
    indexInLines = it - lineCount.begin();
    points.push_back(lines[indexInLines]);
  }
```

```
  // Draw lines in output image
  cout << endl << endl;
  for (int i = 0; i < points.size(); i++) {
    int p = points[i].first;
    int angle = points[i].second;
    double k;
    double b;
    if (sins[angle] == 0) {
      b = p / coss[angle];
      cout << "Line " << i << ": x = " << b << endl;
    }
    else {
      k = -(coss[angle] * 1.0 / sins[angle]);
      b = p * 1.0 / sins[angle];
      cout << "Line " << i << ": y = " << k << " * x + " << b << endl;
    }

    cout << afterHough._width << " " << afterHough._height << endl;
    // dye the points that satisfy the equation with BLUE
    cimg_forXY(showEdge, x, y) {
      int x0 = x - rows / 2;
      int y0 = columns / 2 - y;
      int p_ = x0 * coss[angle] + y0 * sins[angle];
```

```
        if (p == p_) {
          showEdge(x, y) += 255.0 / 2;
          // line is BLUE
          afterHough(x, y, 0) = 0;
          afterHough(x, y, 1) = 0;
          afterHough(x, y, 2) = 255;
        }
      }
    }
    cout << endl << endl;
    //afterHough.display("afterHough", false);
    cout << "End of drawLines" << endl;
}
```

5. 检测出 A4 纸的四个角点。因为之前用的是邻域的方法，所以在靠近角点位置会有很多个点都满足情况。之前的作业只需要画出效果，因此这样做没有问题，但是这次需要获取出 A4 纸非常准确的四个点，不能少也不能多。思考了一下之后发现只需要在原来的基础上加一点条件就好了，判断在当前点与集合中的点的距离，如果该距离小于某一个值，那么我就认为他们是聚在一起的，也就不可能是两个不同的角点了，就不添加进去。这个距离我设置了 10，足够检测出绝大多数的 A4 纸了。还有一个问题就是，后面需要用到这四个点进行透视变换，

但是这个集合是无序的，而变换的时候需要四个点是按左上->右上->左下->右下顺序排列的。因此在最后还需要对点集合进行排序，先对 y 排序，然后分为两部分，前两个为一部分，后两个为一部分，每部分再对 x 进行排序，最后的结果就是有序的。

```
// Get crosspoint
void Segmentation::getCrossPoints() {
  cout << "Begin drawPoints" << endl;
  unsigned char green[3] = {0, 255, 0};
  for (int x = 0; x < rows; x++) {
    for (int y = 0; y < columns; y++) {
      int area[4];
      area[0] = (int)showEdge(x, y);
      area[1] = (int)showEdge(x + 1, y);
      area[2] = (int)showEdge(x, y + 1);
      area[3] = (int)showEdge(x + 1, y + 1);
      // Enough point, indicates a crosspoint!
      if (area[0] + area[1] + area[2] + area[3] >= 255*3/2) {
        bool flag = true;
        for (int i = 0; i < intersections.size(); i++) {
          double dis = pow((x-intersections[i].x), 2) + pow((y-intersections[i].y), 2);
          dis = sqrt(dis);
          if (dis <= 10) {
            flag = false;
            break;
          }
        }
        if (flag) {
          intersections.push_back(Point(x, y, 0));
        }
        afterHough.draw_circle(x, y, 5, green);
      }
    }
```

```
    }
  }
  cout << "End of drawPoints" << endl;
  afterHough.display("drawPoints", false);

  // 对四个角点排序
  // 先对y排序
  sort(intersections.begin(), intersections.end());
  // 对x分别排序
  if (intersections[0].x > intersections[1].x) {
    Point p = intersections[0];
    intersections[0] = intersections[1];
    intersections[1] = p;
  }
  if (intersections[2].x > intersections[3].x) {
    Point p = intersections[2];
    intersections[2] = intersections[3];
    intersections[3] = p;
  }
  /*
  for (int i = 0; i < intersections.size(); i++) {
    cout << intersections[i].x << " " << intersections[i].y << endl;
  }
  */
}
```

6. 然后根据提取到的 A4 纸四个角点的坐标，将 A4 纸作 warping，矫正为一张正的 A4 纸图像。变换的部分参考网上一些快速算法即可，大多都是利用矩阵的性质减少运算量，但其实求解的方程是一样的。

```
// 计算透视变换矩阵
vector<CImg<float>> Segmentation::computeTransformMatrix(CImg<float> a4) {
  vector<Point> destRectPoints;
  Point tempPoint1(0, 0, 0);
  Point tempPoint2(a4._width - 1, 0, 0);
  Point tempPoint3(0, a4._height - 1, 0);
  Point tempPoint4(a4._width - 1, a4._height - 1, 0);
  destRectPoints.push_back(tempPoint1);
  destRectPoints.push_back(tempPoint2);
  destRectPoints.push_back(tempPoint3);
  destRectPoints.push_back(tempPoint4);

  CImg<float> y1(1, 3, 1, 1, 0), y2(1, 3, 1, 1, 0), y3(1, 3, 1, 1, 0), y4(1, 3, 1, 1, 0);
  CImg<float> c1(1, 3, 1, 1, 0), c2(1, 3, 1, 1, 0), c3(1, 3, 1, 1, 0), c4(1, 3, 1, 1, 0);
  CImg<float> A1(3, 3, 1, 1, 1), A2(3, 3, 1, 1, 1);

  for (int i = 0; i < 3; i++) {
    A1(0, i) = destRectPoints[i].x; A1(1, i) = destRectPoints[i].y;
    A2(0, i) = destRectPoints[3-i].x; A2(1, i) = destRectPoints[3-i].y;

    y1(0, i) = intersections[i].x; y2(0, i) = intersections[i].y;
    y3(0, i) = intersections[3-i].x; y4(0, i) = intersections[3-i].y;
  }
  c1 = y1.solve(A1); c2 = y2.solve(A1);
  c3 = y3.solve(A2); c4 = y4.solve(A2);

  CImg<float> temptransform1(3, 3, 1, 1, 0), temptransform2(3, 3, 1, 1, 0);
```

```
  for (int i = 0; i < 3; i++) {
    temptransform1(i, 0) = c1(0, i);
    temptransform1(i, 1) = c2(0, i);

    temptransform2(i, 0) = c3(0, i);
    temptransform2(i, 1) = c4(0, i);
  }
  temptransform1(0, 2) = 0; temptransform1(1, 2) = 0; temptransform1(2, 2) = 1;
  temptransform2(0, 2) = 0; temptransform2(1, 2) = 0; temptransform2(2, 2) = 1;
  vector<CImg<float> > temptransform;
  temptransform.push_back(temptransform1);
  temptransform.push_back(temptransform2);
  return temptransform;
}
```

```
// Perspective Transformation
void Segmentation::warping(CImg<float> srcImg) {
  a4 = CImg<float>(srcImg._width, srcImg._height, 1, 3, 0);

  vector<CImg<float> > transform;
  transform = computeTransformMatrix(a4); // 计算变换矩阵

  CImg<float> y(1, 2, 1, 1, 0);
  CImg<float> c(1, 2, 1, 1, 0);
  CImg<float> A(2, 2, 1, 1, 1);
  A(0, 0) = 0;
  A(0, 1) = a4._width - 1;
  y(0, 0) = a4._height - 1;
  y(0, 1) = 0;
  c = y.solve(A);

  CImg<float> temp1(1, 3, 1, 1, 1), temp2(1, 3, 1, 1, 1);
  cimg_forXY(a4, i, j) {
    temp1(0, 0) = i;
    temp1(0, 1) = j;

    double inner_procuct = i * c(0, 0) - j + c(0, 1);
    temp2 = inner_procuct >= 0 ? transform[0] * temp1 : transform[1] * temp1;
    temp2(0, 0) = temp2(0, 0) < 0 ? 0 : (temp2(0, 0) > x_max ? x_max : temp2(0, 0));
    temp2(0, 1) = temp2(0, 1) < 0 ? 0 : (temp2(0, 1) > y_max ? y_max : temp2(0, 1));
```

```
      a4(i, j, 0) = srcImg(temp2(0, 0), temp2(0, 1), 0);
      a4(i, j, 1) = srcImg(temp2(0, 0), temp2(0, 1), 1);
      a4(i, j, 2) = srcImg(temp2(0, 0), temp2(0, 1), 2);
    }
    a4.display("a4", false);
  }
```

第二部分：

1. 训练模型。Python 中有 sklearn 库提供了 AdaBoost 算法的分类器，我们只需要确定参数，然后训练出模型就可以了。数据提取自 MNIST 数据集，这里用

了 tensorflow 来导入数据，它可以自动提取缺失的数据，在解析的时候也比较方便。数据分为训练数据和测试数据，每个数据又分为图片和标签。使用训练数据训练模型，然后用测试数据对模型进行测试，得出一个准确率。因为训练一个模型用时较长，因此在训练后要保存模型，便于后面的测试。参数主要是弱分类器的数量、学习率、分类算法等，这里我参考了网上的一些例子，分类器数量设在 200-400，学习率为 0.05，分类算法是 SAMME.R。

```python
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.externals import joblib
import tensorflow.examples.tutorials.mnist.input_data as input_data
import time
import os
from datetime import datetime

def train_model(clf_rf, save_dir):
  StartTime = time.clock()
  # Training
  clf_rf.fit(batch_x,batch_y)

  EndTime = time.clock()

  # Save Model
  joblib.dump(clf_rf, save_dir)

  print('Total time %.2f s' % (EndTime - StartTime))


```

```python
if __name__ == '__main__':
  data_dir = '../MNIST_data/'
  mnist = input_data.read_data_sets(data_dir,one_hot=False)
  batch_size = 50000
  batch_x,batch_y = mnist.train.next_batch(batch_size)
  test_x = mnist.test.images[:10000]
  test_y = mnist.test.labels[:10000]

  print("start AdaBoosting")
  save_dir = './model/clf_rf.m'
  i = 400

  if(os.path.isfile(save_dir)):
    clf_rf = joblib.load(save_dir)
  else:
    clf_rf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=5, min_samples_split=5, min_samples_leaf=5)
                            n_estimators=i, learning_rate=0.05, algorithm='SAMME.R')
    train_model(clf_rf, save_dir)

  y_pred_rf = clf_rf.predict(test_x)
  print("Result: %d", y_pred_rf)
  acc_rf = accuracy_score(test_y,y_pred_rf)
  print("%s n_estimators = %d, random forest accuracy:%f" % (datetime.now(), i, acc_rf))

```

2. 测试手写数据。读入一张手写数字的图片，然后使用训练出来的模型对这个数据进行检测，查看结果是否正确。读入图片需要转换成 MNIST 数据，这里是上网找了教程做的格式转换。

```python
MNIST_SIZE = 28

def translate(image_path):
    #读入图片并变成灰色
    img = io.imread(image_path, as_grey=True)
    #缩小到28*28
    translated_img = transform.resize(img, (MNIST_SIZE, MNIST_SIZE))
    #变成1*784的一维数组
    flatten_img = np.reshape(translated_img, 784)
    #mnist数据集中1代表黑，0代表白
    result = np.array([flatten_img])
    #返回该图的所代表的向量
    return result

def train_model(clf_rf, save_dir):
    StartTime = time.clock()
    # Training
    clf_rf.fit(batch_x,batch_y)

    # Save Model
    joblib.dump(clf_rf, save_dir)

    EndTime = time.clock()
    print('Total time %.2f s' % (EndTime - StartTime))
```

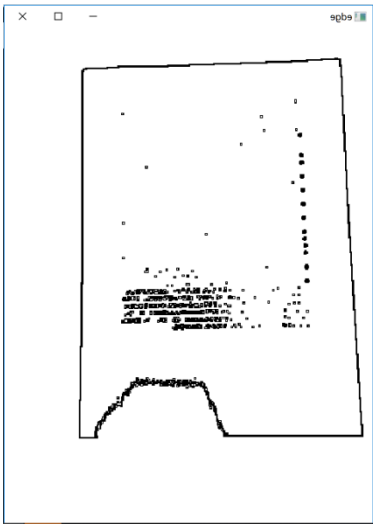
```python
if __name__ == '__main__':
    mnist = translate("./five.jpg");
    batch_size = 50000

    print("start AdaBoosting")
    save_dir = './test/clf_rf.m'
    i = 10

    if(os.path.isfile(save_dir)):
        clf_rf = joblib.load(save_dir)
    else:
        clf_rf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=5, min_samples_split=5, min_samples_leaf=5)
        train_model(clf_rf, save_dir)

    y_pred_rf = clf_rf.predict(mnist)
    print("Result: %d", y_pred_rf)
```

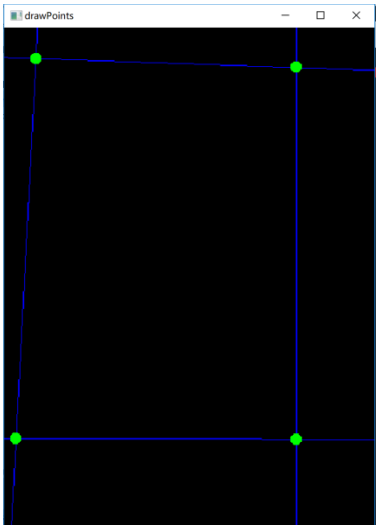
四、测试与分析

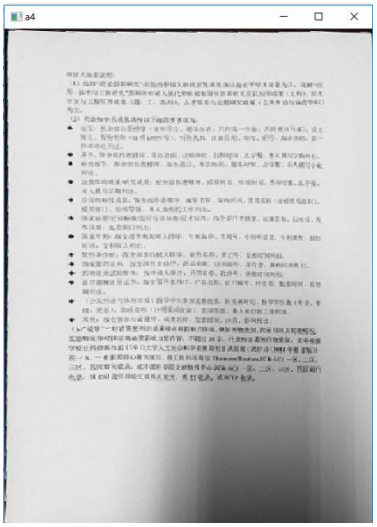
第一部分：

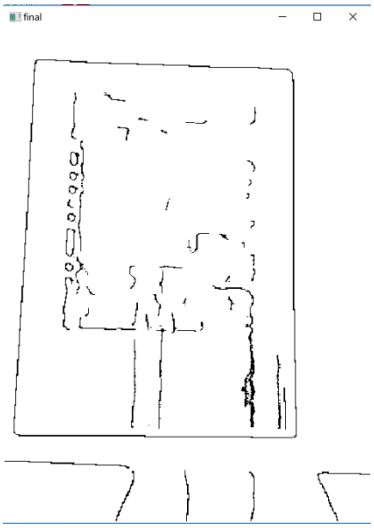
说明：测试数据使用作业三的 A4 纸数据，每张图片展示图像分割后提取边缘后的图、霍夫变换后检测出的图和最后矫正后的 A4 纸图。最后再给出一张使用 canny 算子获取边缘的效果图进行对比。

①

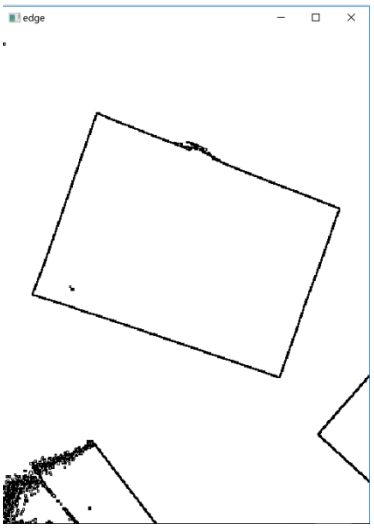


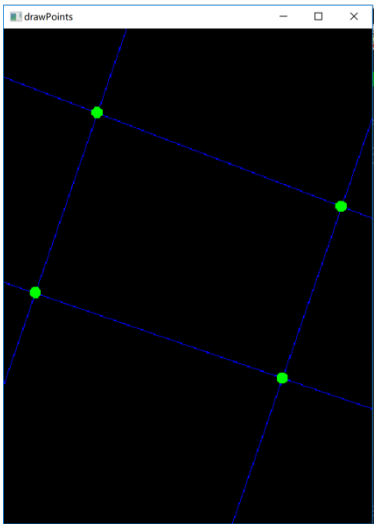




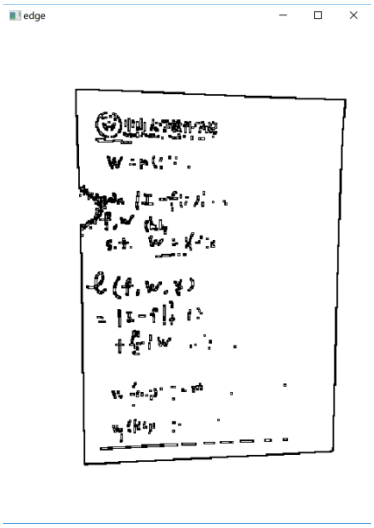


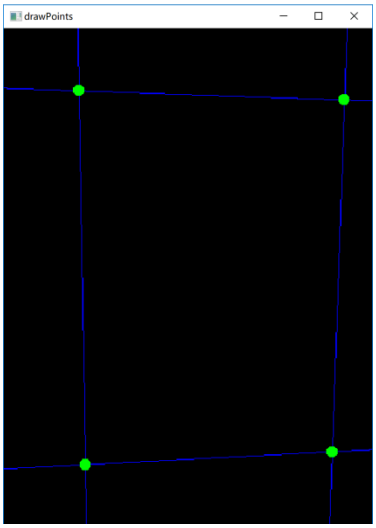
②

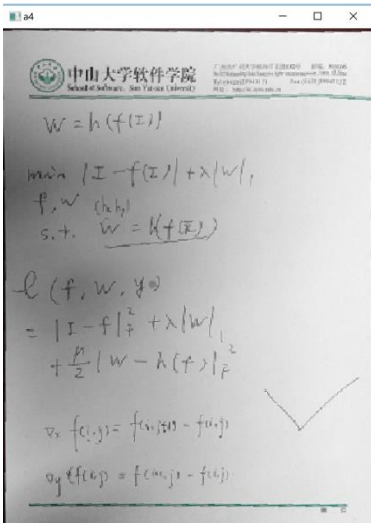


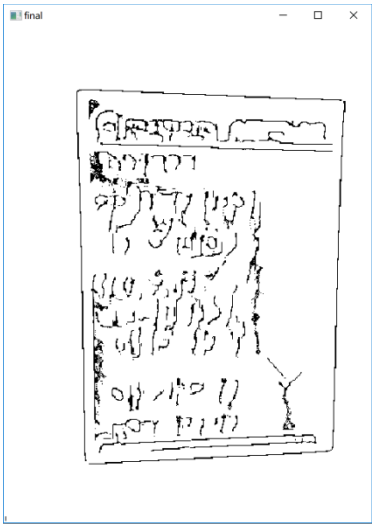


③

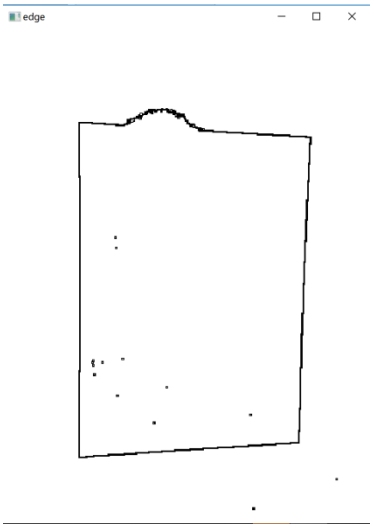


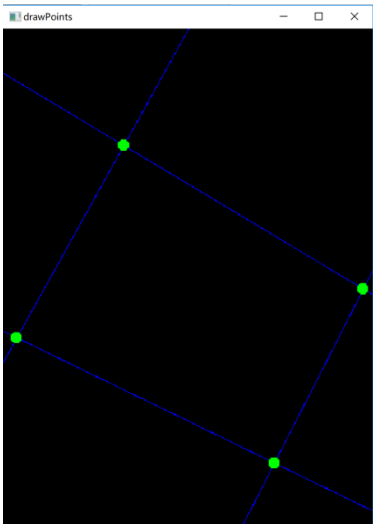


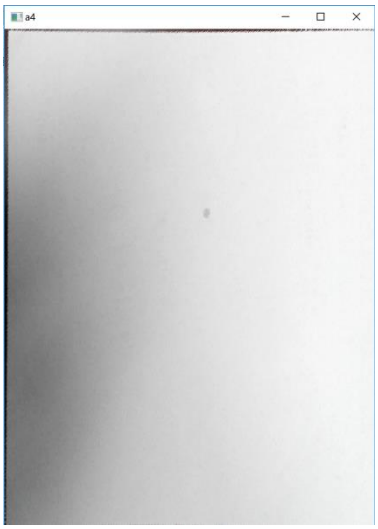


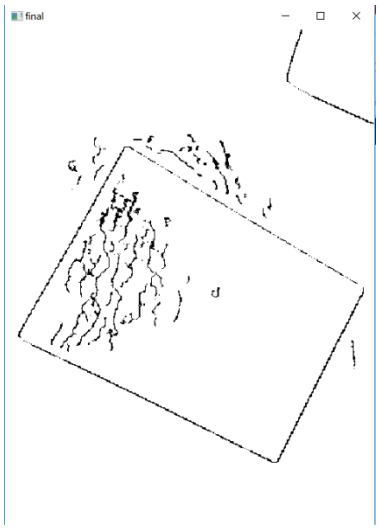


④

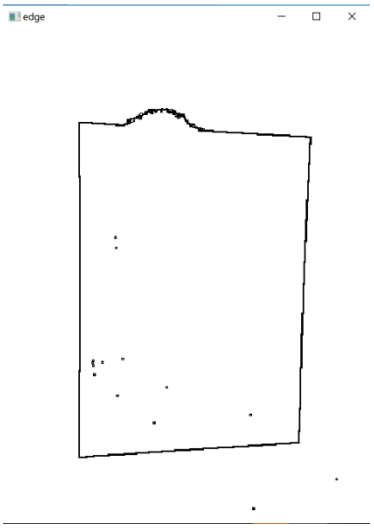


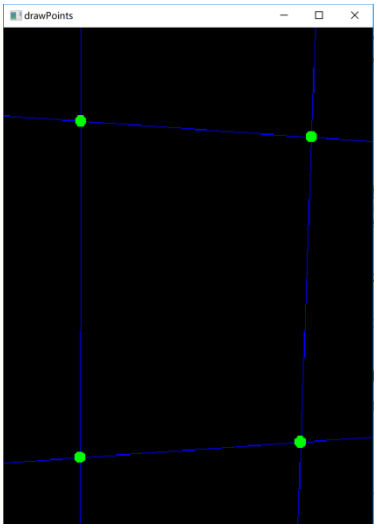


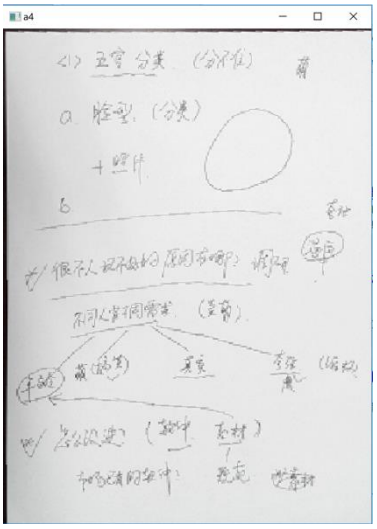


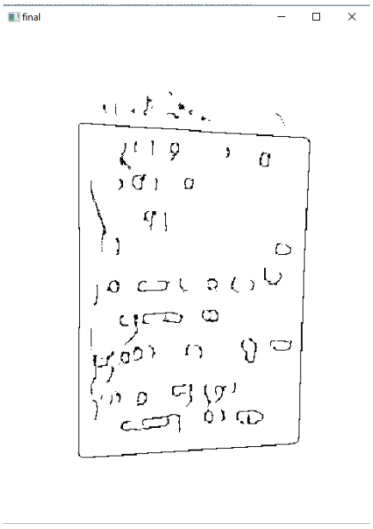


⑤

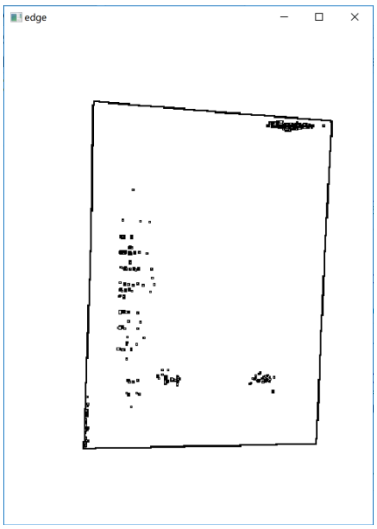


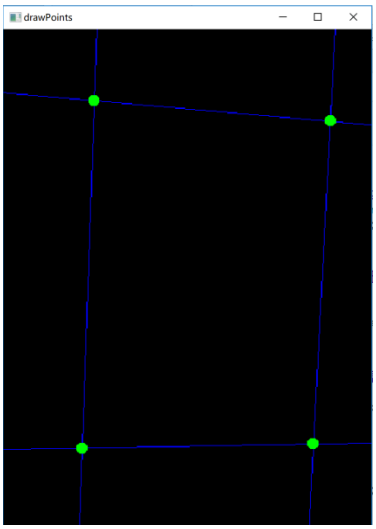


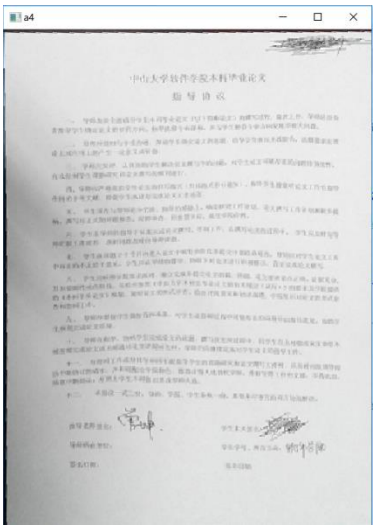


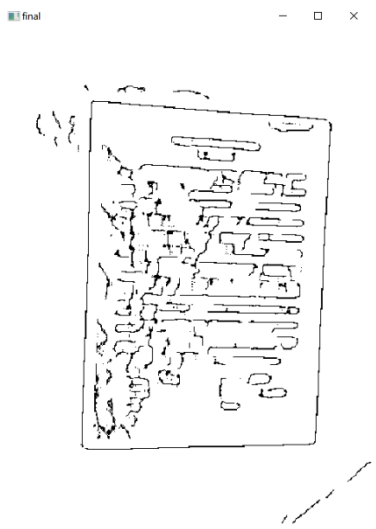


⑥









分析：从矫正的角度来看，算法是比较成功的，六张图片都能很好地将 A4 纸矫正输出，都很准确地检测到了 A4 纸的边缘，在丢弃无用信息的同时，很好地保留了有用信息。

对比使用 canny 提取边缘和图像分割提取边缘，我认为它们各有优劣。Canny 算子对于图像中的细节保留的较多，而图像分割对于目标的细节保留的较少。以本数据集为例，在有光线影响的情况下，canny 算子依然能够很好地提取到边缘，但使用图像分割则会出现缺失。之所以最后能够正确检测，是因为霍夫变换很好地投票出四条直线。因此我认为，对于目标与背景易于区分、对细节要求较少的数据，可以采用图像分割，而对于受光照影响较大、边缘细节丰富的数据，应该使用 canny 算法。

第二部分：

首先是训练模型，使用 MNIST 的测试数据集测试：

Estimators = 10

```
start AdaBoosting
Total time 63.17 s
Result: %d [7 2 1 ... 4 5 6]
2018-12-19 00:05:56.782150 n_estimators = 10, random forest accuracy:0.857400
```

Estimators = 20

```
start AdaBoosting
Total time 130.50 s
Result: %d [7 2 1 ... 4 5 6]
2018-12-19 00:10:16.815396 n_estimators = 20, random forest accuracy:0.883100
```

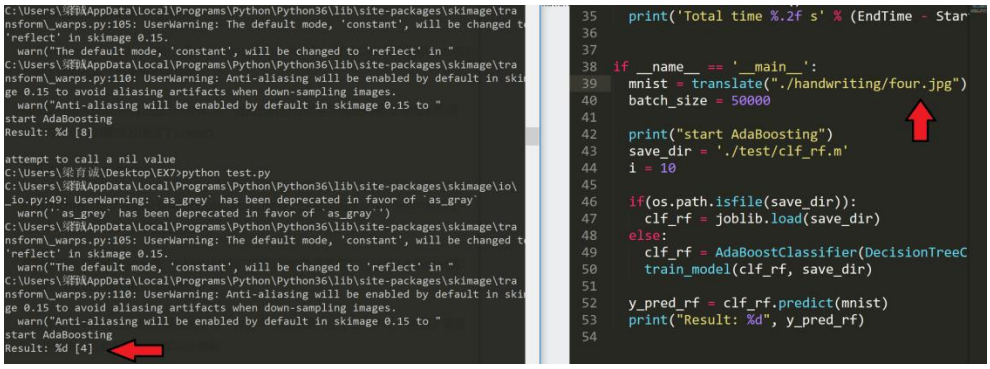
Estimators = 100

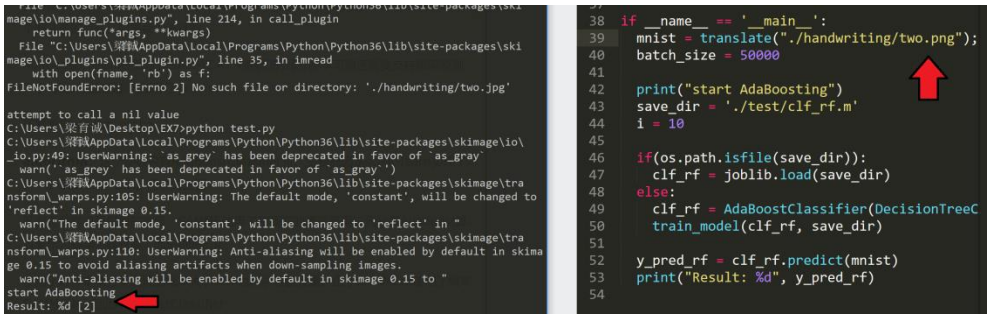
```
start AdaBoosting
Total time 676.07 s
Result: %d [7 2 1 ... 4 5 6]
2018-12-19 00:34:17.530971 n_estimators = 100, random forest accuracy:0.928000
```

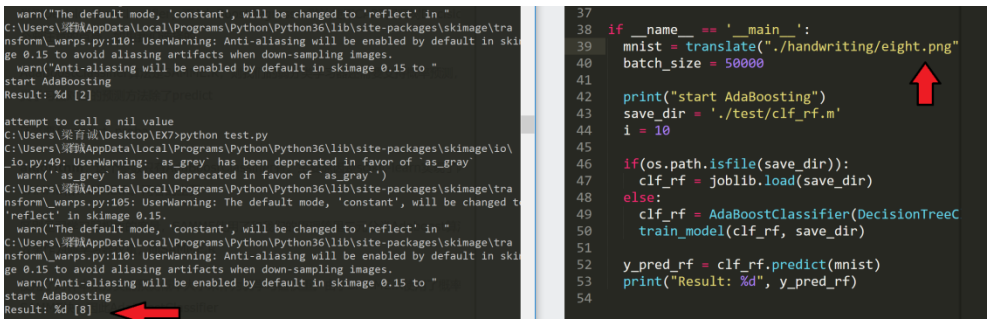
Estimators = 400（因为训练时间过长，之前已经训练好了，这里无训练时间，大概是 1 个小时）

```
start AdaBoosting
Result: %d [7 2 1 ... 4 5 6]
2018-12-18 23:32:58.469051 n_estimators = 400, random forest accuracy:0.935900
```

然后是自己输入手写的数字图片进行识别：







分析：弱分离器数量越多，模型准确率越高，当然耗时也就越长。训练模型后，使用 MNIST 数据集测试，最好的准确率接近 93.6%。然后输入自己手写的数字图片，基本能识别出数字，但是对于部分数字，或者写的不工整的，模型可能

会识别错误，但总体来说正确率还是很高的。

五、难点、思考、总结与体会

难点：1. 在做图像分割的时候，计算类间方差的公式经常报错，一直卡在公式那里，于是我想可能是做除法的时候除以 0 了。输出信息，发现确实是这样，检查算法没有错，最后发现原来是类型错误，除法的时候要使用小数，否则就会截取整数部分。

2. 确定梯度阈值。在计算梯度提取边缘的时候，一开始我判断的条件是以 0 为界限，不等于 0 的都认为是 EDGE。但是发现这样做会有很多点，噪声很大，因此我觉得要设置一个一定大小的阈值，用于去除噪声。通过输出数据，找到了一个大概的值 30，跑了几次效果还不错，就确定用这个了。

3. 霍夫变换。霍夫变换的代码是用回之前自己写的，但是在画线的时候就报错了，因为是在 cimg_forXY 中，因此我认为是图像越界问题，但是 x、y 都没有越界，只可能是图像色道问题。查看变量名后，发现在 Segmentation 整个类中我们操作的都是 main 传入的灰度图，因此不能有彩色。新增一个变量用于存储彩色图即可。

4. 透视变换。透视变换本身不难，之前做人脸变形已经研究过。这次难点在于如何拿到四个角点。一开始的问题是检测到过多的角点，这是因为我使用了邻域的方法来判断。解决办法是计算两点之间的距离，过小则认为他们是同一个点，这样实现后角点的数量就是正确的 4 个，但是顺序不对。我们需要按序排好才能应用透视变换，上网查了一下，其实很简单。先对 y 排序，然后分成两组（前两个一组，后两个一组），再对每组的 x 进行排序。

5. Adaboost 检测自己手写图片的时候，因为要将图片转化为 MNIST 类型，所以在网上找了一下代码。检测了几张发现结果全部错误，我认为模型应该是没问题的，可能是测试数据处理的不对。上网看了 MNIST 解析后，发现测试数据转化的时候不需要取反，因为 1 是黑，0 是白，修改过后就能正确识别大多数的数字了。

总结：这次实验用到了之前学过的知识——霍夫变换、边缘检测等，在写代码的过程中，也发现自己对于计算机视觉的一些概念都比较熟悉了，基本知道一些图像处理的思路与方法，使用边缘检测算子的时候也比较熟练地运用了库的函数。因为霍夫变换是自己写的代码，因此理解起来也没花多少时间，稍微修改一下就能用上，节省了很多时间。在透视变换上还是花了一点时间理解，但是网上找到了不错的资料，因此也比较快就写出代码了。至于 AdaBoost 算法，因为是第一次接触到了学习算法，在谷歌一搜基本都是 python 的版本，因此也就用了 python 来实现。经过这次实验，我大致知道了如何使用数据集去训练一个模型，然后再用这个模型来对测试数据进行检测。我们要在理解算法原理的基础上，对参数进行调试，从而得出一个比较好的模型。这次训练出来的模型准确率还算不错，对于大部分手写数字都能识别，但有时候还是会识别错误。总而言之，这次作业收获很大，既有对以前知识的回顾，也接触到了全新的东西。

六、参考资料

1. 四顶点校正透视变换的线性方程求解：

https://www.cnblogs.com/faith0217/articles/5027490.html
2. AdaBoost 实现 MNIST 手写体数字识别：
https://blog.csdn.net/Barry_J/article/details/81950336
3. 手写数字图片预处理：
https://blog.csdn.net/qq_40358998/article/details/79281936
4. sobel 算子：https://blog.csdn.net/Mahabharata_/article/details/69099136
5. 图像分割 OTSU：https://blog.csdn.net/liyuanbhu/article/details/49387483