《计算机视觉》

实验文档

（实验八）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	16 软件工程（数字媒体）
学 生 姓 名 ：	黎汛言
学　　号 ：	16340109
时　　间 ：	2018　年　01　月　07　日

实验八 ：期末综合项目

一. 实验目的

1. 综合本学期学到的知识，完成A4纸的矫正、手写数字的提取、分割和识别等一系列任务。

二. 实验内容

1. 采用边缘检测或者图像分割的方法获取图像的边缘，并计算图像的四个角点，完成图像矫正；
2. 采用图像分割（二值化）的方法，获取图像中的手写字符，输出二值化结果；
3. 针对二值化图像，对Y方向投影切割出各个行的图像，例如学号切成单个的图像，手机号和身份证号也如此。根据上面的结果，针对行图像（如学号图像），对X方向做投影切割，切割出单个字符；
4. 针对单个切割好的字符，进行分类识别。

三. 实验环境

Windows 10 64位、C++、Python 3.7

四. 实验过程

1. 采用边缘检测或者图像分割的方法获取图像的边缘，并计算图像的四个角点，完成图像矫正；(Step1.cpp)

   Step1的主要调用过程如下：

```
// 加载图片
cout << "Loading image '" << infilename << "'." << endl;
CImg<unsigned char> src(infilename);

// Canny边缘检测
Canny cny(src, sigma, tlow, thigh);
CImg<unsigned char> edge = cny.edge_detection();

// 霍夫变换
Hough hough(edge, houghThreshold, range);
vector<Point> intersections = hough.line_detection();

// A4纸矫正
Warping warping(intersections, src);
warping.getPointPair();
warping.calculateHomography();
CImg<unsigned char> warpingResult = warping.warp();

// 输出结果
cout << "Saving warped image to 'Result/Step1/" << outputname << endl;
char result[30];
sprintf(result, "Result\\Step1\\%s", outputname);
warpingResult.save(result);
```

下面是每一步的具体内容：

i. 边缘检测（Canny.h、Canny.cpp）

使用Canny算子，检测输入图像的边缘，得到边缘图像。类Canny定义如下：

```
class Canny {
    private:
        int w, h;                       /* The dimensions of the image.               */
        CImg<unsigned char> src,        /* The input image                            */
                            image,      /* The grey scale image                       */
                            edge,       /* The output edge image                      */
                            nms;        /* Points that are local maximal magnitude.   */
        CImg<short int> smoothedim,     /* The image after gaussian smoothing.        */
                        delta_x,        /* The first devivative image, x-direction.   */
                        delta_y,        /* The first derivative image, y-direction.   */
                        magnitude;      /* The magnitude of the gadient image.        */
        float sigma,                    /* Standard deviation of the gaussian kernel. */
              tlow,                     /* Fraction of the high threshold in hysteresis.*/
              thigh;                    /* High hysteresis threshold control. The actual
                                           threshold is the (100 * thigh) percentage
                                           point in the histogram of the magnitude of
                                           the gradient image that passes non-maximal
                                           suppression.                               */
        void to_grey_scale();                       /* Convert the image to grey scale.       */
        void gaussian_smooth();                     /* Perform gaussian smoothing on the
                                                       image using the input standard deviation.  */
        void derrivative_x_y();                     /* Compute the first derivative in the
                                                       x and y directions.                    */
        void magnitude_x_y();                       /* Compute the magnitude of the gradient. */
        void non_max_supp();                        /* Perform non-maximal suppression.       */
        void apply_hysteresis();                    /* Use hysteresis to mark the edge pixels.*/
        void optimize();                            /* Delete edges that are shorter than 20 px.  */

        void make_gaussian_kernel(float**, int*);   /* Create a one dimensional gaussian kernel.  */
        void follow_edges(int, int, int);           /* A recursive routine that traces edgs along
                                                       all paths whose magnitude values remain above
                                                       some specifyable lower threshhold.     */
        bool connect(int, int, int);                /* A recursive routine that finds whether an edge
                                                       point should be preserved.             */
    public:
        Canny(CImg<unsigned char>, float, float, float);    /* The constructor. */
        CImg<unsigned char> edge_detection();               /* Start the edge detection process. */
};
```

Canny算子的实现方法与作业2相同，可参见代码Canny.cpp。

ii. 边缘直线拟合（Hough.h、Hough.cpp）

使用霍夫变换，拟合A4纸边缘的直线，并求出直线的交点，得到A4纸4个角点的坐标。类Hough定义如下：

```
class Hough {
    private:
        int w, h;                               /* The dimensions of the image.     */

        CImg<unsigned char> edge;               /* The input image                  */
        CImg<short> hough;                      /* The hough space                  */
        vector<Point> peaks;                    /* Peak points in hough space       */
        int houghThreshold;
        int range;
        vector<Point> intersections;            /* Intersections of lines           */

        void hough_transform();
        void find_peak_points();
        void draw_intersections();
    public:
        Hough(CImg<unsigned char> , int, int);  /* The constructor. */
        vector<Point> line_detection();         /* Start the edge detection process. */
};
```

霍夫变换的实现方法与作业3相同，可参见代码Hough.cpp。

iii. A4纸矫正（Warping.h、Warping.cpp）

根据霍夫变换求得的角点坐标，可以求出从标准A4纸平面变换到原图像所对应的单应矩阵H。利用单应矩阵H，从标准A4纸平面做反向映射，可以将原图像中的A4纸内容矫正到标准A4纸图像中。

首先定义结构体pointPair，表示存在映射关系的点对：

```
struct pointPair {
    Point a;
    Point b;
    pointPair(Point _a, Point _b) {
        a = _a;
        b = _b;
    }
};
```

定义类Warping，封装Warping的操作：

```
class Warping {
private:
    vector<pointPair> pointPairs;    // 原图像和目标图像互相对应的点
    vector<Point> intersections;     // A4纸角点
    CImg<unsigned char> src;         // 原图像
    CImg<unsigned char> dst;         // 目标图像
    CImg<float> H;                   // 变换所用到的单应矩阵
public:
    Warping(vector<Point> _point, CImg<unsigned char> _src);
    void getPointPair();
    void calculateHomography();
    CImg<unsigned char> warp();
};
```

在getPointPair()函数中，对A4纸的4个角点进行排序。这一步是为了保证映射后A4纸方向的正确性(在本项目中不太重要)。具体代码参见Warping.cpp。

在calculateHomography()函数中，利用得到的点对，计算单应矩阵。代码与作业6中图像拼接过程使用到的代码相同，主要过程是，利用4对点的坐标，形成两个矩阵，然后使用CImg的solve函数求解出单应矩阵H。在这个过程中，同时还将A4纸的角点坐标输出到一个文件中，从而方便在后面生成excel文件。实现如下：

```
void Warping::calculateHomography() {
    int size = pointPairs.size();
    CImg<float> x(1, 8, 1, 1, 0); // 单应矩阵的参数向量
    CImg<float> b(1, 2 * size, 1, 1, 0);
    CImg<float> A(8, 2 * size, 1, 1, 0);
    ofstream out;
    out.open("corner.txt", std::ios::app);

    cout << "Warping the image." << endl;
    cout << "   The mappings of corners from a standard A4 paper to the image are: " << endl;
    for (int i = 0; i < pointPairs.size(); i++) {
        float x1 = pointPairs[i].a.a; float y1 = pointPairs[i].a.b;
        float x_1 = pointPairs[i].b.a; float y_1 = pointPairs[i].b.b;
        A(0, i * 2) = x1;
        A(1, i * 2) = y1;
        A(2, i * 2) = 1;
        A(6, i * 2) = -(x1 * x_1);
        A(7, i * 2) = -(y1 * x_1);
        A(3, i * 2 + 1) = x1;
        A(4, i * 2 + 1) = y1;
        A(5, i * 2 + 1) = 1;
        A(6, i * 2 + 1) = -(x1 * y_1);
        A(7, i * 2 + 1) = -(y1 * y_1);
        b(i * 2) = x_1; b(i * 2 + 1) = y_1;
        cout << "   (" << x1 << ", " << y1 << ") -> (" << x_1 << ", " << y_1 << ")" << endl;

        if (out.is_open()) {
            out << " " << x_1 << ", " << y_1;
        }
    }
    if (out.is_open()) {
        out << endl;
        out.close();
    }
    // 对于Ax = b，求解x
    x = b.solve(A);
    cimg_forY(x, y) {
        H(y) = x(0, y);
    }
    H(2, 2) = 1;
}
```

得到单应矩阵后，从标准A4纸上反向映射到原图像，求得对应的坐标，完成A4纸的矫正：

```
CImg<unsigned char> Warping::warp() {
    cimg_forXY(dst, x, y) {
        CImg<float> xy1(1, 3, 1, 1, 1);
        xy1(0) = x;
        xy1(1) = y;
        CImg<float> target = H * xy1;    // 反向映射
        target /= target(2);
        int target_x = (int)target(0);
        int target_y = (int)target(1);
        dst(x, y, 0) = src(target_x, target_y, 0);
        dst(x, y, 1) = src(target_x, target_y, 1);
        dst(x, y, 2) = src(target_x, target_y, 2);
    }
    return dst;
}
```

2. 采用图像分割（二值化）的方法，获取图像中的手写字符，输出二值化结果；(Step2.cpp)

Step2的主要调用过程如下：

```
// 加载图片
cout << "Loading image '" << infilename << "'." << endl;
CImg<unsigned char> src(infilename);
CImg<unsigned char> image(src.width(), src.height(), 1, 1, 0);

// 转灰度图
cimg_forXY(src, x, y) {
    int r = src(x, y, 0);
    int g = src(x, y, 1);
    int b = src(x, y, 2);
    int grey = (r * 30 + g * 59 + b * 11 + 50) / 100;
    image(x, y) = grey;
}

// 二值化
OSTU ostu = OSTU(image, edge);
CImg<unsigned char> binaryResult = ostu.ostu();

// 输出结果
cout << "Saving binary image to 'Result/Step2/" << outputname << endl;
char result[30];
sprintf(result, "Result\\Step2\\%s", outputname);
binaryResult.save(result);
```

下面是二值化的具体实现（OSTU.h、OSTU.cpp）：

i. 在OSTU类中封装图像二值化的全部操作。定义如下：

```
class OSTU {
private:
    CImg<unsigned char> img;    // 待分割的图像
    int hist[256];              // 直方图
    int threshold;              // 分割的阈值
    int edge;                   // 图像边缘宽度
public:
    OSTU(CImg<unsigned char> _img, int _edge);
    CImg<unsigned char> ostu(); // 确定每一块的阈值和是否进行分割
    CImg<unsigned char> divide(CImg<unsigned char> crop, int threshold); // 分割
};
```

ii. 在构造函数中，初始化图像和边缘宽度，将边缘置为黑色：

```
OSTU::OSTU(CImg<unsigned char> _img, int _edge) {
    img = _img;
    edge = _edge;
    // 将边缘置0
    cimg_forXY(img, x, y) {
        if (x < edge || y < edge ||
            x >= img.width() - edge || y >= img.height() - edge) {
            img(x, y) = 0;
        }
    }
}
```

iii. 在ostu()函数中对图像进行分块，计算局部阈值，进行局部二值化。

由于本项目中的测试图片存在光线不均匀的情况，所以不能简单的采用全局阈值的方法来分割图像。我采用了局部阈值的方法，将图像划为200×200的小块，在每一块中使用OSTU算法计算分割阈值。考虑到我们不需要对图像的背

景区域进行分割，我还计算了每一块中前景占所有像素的比例，只有当比例比较小时（<=0.08），才认为块内存在数字，需要分割，否则认为整个块都属于背景。具体实现如下：

```
CImg<unsigned char> OSTU::ostu() {
    // 将图像分成200×200的块，逐块进行分割
    for (int x0 = edge; x0 < img.width() - edge; x0 += 200) {
        for (int y0 = edge; y0 < img.height() - edge; y0 += 200) {
            int x1 = x0 + 199 < img.width() - edge ? x0 + 199 : img.width() - edge - 1;
            int y1 = y0 + 199 < img.height() - edge ? y0 + 199 : img.height() - edge - 1;
            CImg<unsigned char> crop = img.get_crop(x0, y0, x1, y1); // 图像块
            for (int i = 0; i < 256; i++) { // 初始化直方图
                hist[i] = 0;
            }
            cimg_forXY(crop, x, y) { // 统计直方图
                hist[(int)crop(x, y)]++;
            }
            double maxVariance = 0; // 当前最大方差
            int pixelNum = crop.width() * crop.height();    // 像素数量
            // 遍历所有灰度值，找到使类间方差最大的灰度值作为分割阈值（OSTU）
            for (int i = 0; i < 256; i++) {
                int lowNum = 0, lowValue = 0, highNum = 0, highValue = 0;
                double w0, w1, u0, u1, g;
                for (int j = 0; j <= i; j++) {
                    lowNum += hist[j];
                    lowValue += j * hist[j];
                }
                for (int j = i + 1; j < 256; j++) {
                    highNum += hist[j];
                    highValue += j * hist[j];
                }
                w0 = (double)lowNum / (double)pixelNum;     // 前景像素占总像素的比例
                w1 = (double)highNum / (double)pixelNum;    // 背景像素占总像素的比例
                u0 = (double)lowValue / (double)lowNum;     // 前景平均灰度
                u1 = (double)highValue / (double)highNum;   // 背景平均灰度
                g = w0 * w1 * pow((u0 - u1), 2);            // 类间方差
                if (g > maxVariance) {
                    maxVariance = g;
                    threshold = i;
                }
            }
            int lowNum = 0;
            double w0;
            for (int i = 0; i <= threshold; i++) {
                lowNum += hist[i];
            }
            w0 = (double)lowNum / (double)pixelNum; // 前景像素占总像素的比例
            if (w0 <= 0.08) { // 前景占比较小，分割
                crop = divide(crop, threshold);
            }
            else {  // 否则不分割，全部置0
                cimg_forXY(crop, x, y) {
                    crop(x, y) = 0;
                }
            }
            // 将处理后的块放回原图中
            for (int i = x0; i <= x1; i++) {
                for (int j = y0; j <= y1; j++) {
                    img(i, j) = crop(i - x0, j - y0);
                }
            }
        }
    }
    return img;
}
```

iv. 在divide()函数中，分割图像块的前景和背景。

```
CImg<unsigned char> OSTU::divide(CImg<unsigned char> crop, int threshold) {
    // 二值化
    cimg_forXY(crop, x, y) {
        if (crop(x, y) <= threshold) {
            crop(x, y) = 255;
        }
        else {
            crop(x, y) = 0;
        }
    }
    return crop;
}
```

3. 针对二值化图像，对Y方向投影切割出各个行的图像，例如学号切成单个的图像，手机号和身份证号也如此。根据上面的结果，针对行图像（如学号图像），对X方向做投影切割，切割出单个字符；(Step3.cpp)

Step3的主要调用过程如下：

```
// 读取图像
cout << "Loading image '" << infilename << "'." << endl;
CImg<unsigned char> image(infilename);

// 切割数字
Cutting cutting(image, outputname);
// 切割行
cutting.divideToLines();
// 切割列
cutting.divideToSingleNumber();
// 保存切割出的单个数字
cutting.save();
```

下面是切割的具体实现（Cutting.h、Cutting.cpp）：

i. 在Cutting类中完成数字切割的全部操作。定义如下：

```
class Cutting {
private:
    const char* outputname;     // 输出文件夹的名称
    CImg<float> histY, histX;   // Y方向和X方向的直方图
    CImg<unsigned char> binary; // 输入的二值化图像
    vector<int> cutY;           // Y方向的切割坐标集合
    vector<CImg<unsigned char> > rows;  // 切割出的行图像集合
    vector<vector<CImg<unsigned char> > > numbers;  // 切割出的单个数字集合
public:
    Cutting(CImg<unsigned char> img, const char* filename) {
        binary = img;
        outputname = filename;
    }
    void divideToLines();           // 切割行
    void divideToSingleNumber();    // 切割列
    void save();                    // 保存所有数字图像
};
```

ii. 在divideToLines()函数中，将图像切割成一行行的数字图像。

首先，对图像进行Y方向投影，得到Y方向的直方图，并得到直方图的各个拐点坐标：

```
void Cutting::divideToLines() {
    histY.assign(1, binary.height(), 1, 1, 0);  // 初始化Y方向的直方图
    vector<int> inflection; // 直方图的拐点集合
    inflection.push_back(0);

    // 对图像在Y方向做投影，得到Y方向的直方图
    cimg_forY(histY, y) {
        cimg_forX(binary, x) {
            if (binary(x, y) == 255) {
                histY(y)++;
            }
        }
        // 将直方图中波谷的开始和结束点加入拐点集合
        if (y > 0) {
            if (histY(y) > 0 && histY(y - 1) == 0) {
                inflection.push_back(y - 1);
            }
            else if (histY(y) == 0 && histY(y - 1) > 0) {
                inflection.push_back(y);
            }
        }

    }
    inflection.push_back(binary.height() - 1);
```

然后，计算两两拐点的中点，得到分割的Y坐标：

```
// 计算两个拐点坐标的平均值，得到分割的Y坐标
if (inflection.size() > 2) {
    for (int i = 0; i < inflection.size() - 1; i += 2) {
        int y = (inflection[i] + inflection[i + 1]) / 2;
        cutY.push_back(y);
    }
}
```

接下来，分割出每一行，统计行内白色像素的个数，以确定该行是否存在数字：

```
    vector<int> tempCutY;
    for (int i = 1; i < cutY.size(); i++) {
        int rowHeight = cutY[i] - cutY[i - 1];
        CImg<unsigned char> row(binary.width(), rowHeight, 1, 1, 0);
        int num = 0;
        // 分离出每一行的图像
        cimg_forXY(row, x, y) {
            row(x, y) = binary(x, cutY[i - 1] + y + 1);
            if (row(x, y) == 255)
                num++;
        }
        // 当行中白色像素的数量大于0.001的比例时，才认为该行有效
        if ((float)num / ((float)binary.width() * (float)rowHeight) > 0.001) {
            rows.push_back(row);
            tempCutY.push_back(cutY[i - 1]);
        }
    }
    cutY.clear();
    for (int i = 0; i < tempCutY.size(); i++) {
        cutY.push_back(tempCutY[i]);
    }
}
```

iii. 在divideToSingleNumber()函数中，将行图像切割成一个个数字。

切割方法与切割行类似。对行图像进行X方向的投影，找到直方图拐点，计算拐点的中点，进行切割。切割后计算列中白色像素的比例，以确定列中是否包含数字。

下面说明列分割与行分割的不同之处。

要切割出单个数字，需要求出数字的坐标范围，生成包裹数字的图像块：

```
int minX = number.width() - 1, minY = number.height() - 1,
    maxX = 0, maxY = 0;
cimg_forXY(number, x, y) {
    if (number(x, y) == 255) {
        if (x < minX) minX = x;
        if (x > maxX) maxX = x;
        if (y < minY) minY = y;
        if (y > maxY) maxY = y;
    }
}
int width = maxX - minX + 1;
int height = maxY - minY + 1;
// 求出包裹数字的图像
CImg<unsigned char> cutNum(width, height, 1, 1, 0);
cimg_forXY(cutNum, x, y) {
    cutNum(x, y) = number(x + minX, y + minY);
}
```

由于MNIST图像都是正方形图像，为了提高后续识别的准确率，需要将数字图像补成正方形，并且数字与图像边缘之间存在一定的空隙（edge）：

```
// 生成数字的正方形图像
CImg<unsigned char> squareNum;
// 用黑色将图像补成正方形
if (width < height) {
    float edge = height * 0.2;
    int gap = (height - width) / 2 + (int)edge;
    minX -= gap;
    maxX += gap;
    width = maxX - minX + 1;
    height += (int)edge * 2;
    squareNum.assign(width, height, 1, 1, 0);
    cimg_forXY(cutNum, x, y) {
        squareNum(x + gap, y + (int)edge) = cutNum(x, y);
    }
}
else if (width > height) {
    float edge = width * 0.2;
    int gap = (width - height) / 2 + (int)edge;
    minY -= gap;
    maxY += gap;
    height = maxY - minY + 1;
    width += (int)edge * 2;
    squareNum.assign(width, height, 1, 1, 0);
    cimg_forXY(cutNum, x, y) {
        squareNum(x + (int)edge, y + gap) = cutNum(x, y);
    }
}
else {
    squareNum = cutNum;
}
```

最后，为了提高识别准确率，需要对数字进行膨胀操作，避免数字过细或者断

裂：

```
// 根据图像的大小，对数字进行膨胀，避免数字断裂和过细
int expand = (squareNum.width() / 100 + 1) * 2; // 膨胀次数
for (int k = 0; k < expand; k++) {
    CImg<unsigned char> temp(squareNum.width(), squareNum.height(), 1, 1, 0);
    // 向8邻域膨胀
    cimg_forXY(squareNum, x, y) {
        if (squareNum(x, y) == 255) {
            temp(x, y) = 255;
            if (x > 0 && y > 0) {
                temp(x - 1, y - 1) = 255;
                temp(x - 1, y) = 255;
                temp(x, y - 1) = 255;
            }
            if (x < temp.width() - 1 && y < temp.height() - 1) {
                temp(x, y + 1) = 255;
                temp(x + 1, y) = 255;
                temp(x + 1, y + 1) = 255;
            }
            if (x > 0 && y < temp.height() - 1) {
                temp(x - 1, y + 1) = 255;
            }
            if (x < temp.height() - 1 && y > 0) {
                temp(x + 1, y - 1) = 255;
            }
        }
    }
    squareNum = temp;
}
rowNumbers.push_back(squareNum); // 将图像加入到行中的数字集合
tempCutX.push_back(cutX[j - 1]);
```

每一行切割完成后，将行的数字集合加入到行的集合中。最后，numbers中包含了所有的数字图像。

```
numbers.push_back(rowNumbers);  // 将行数字集合加入到总数字集合
```

iv. 在save()函数中，将分割出的单个数字图像保存到指定的路径中。

在保存图像的同时，使用一个txt文件记录输出目录中所有图像的文件名，使用另一个txt文件存储所有输出目录的路径。

```cpp
void Cutting::save() {
    char outputDir[100];
    sprintf(outputDir, "Result/Step3/%s", outputname);
    // 新建输出路径
    if (_access(outputDir, 0) == -1)
        _mkdir(outputDir);
    cout << "Saving cut images to '" << outputDir << "'." << endl;
    ofstream out, imageList;
    char dir[36];
    sprintf(dir, "Result\\Step3\\%s\\imageList.txt", outputname);
    imageList.open(dir); // 使用一个文件，记录该目录下所有图像的文件名
    for(int i = 0; i < numbers.size(); i++) {
        for(int j = 0; j < numbers[i].size(); j++) {
            // 保存图像
            char result[100];
            sprintf(result, "Result\\Step3\\%s\\row%d_col%d.jpg", outputname, i, j);
            numbers[i][j].save(result);
            if (imageList.is_open()) {
                imageList << "row" << i << "_col" << j << ".jpg" << " ";
            }
        }
        if (imageList.is_open()) {
            imageList << endl;
        }
    }
    if (imageList.is_open()) {
        imageList.close();
    }
    out.open("imageDir.txt", std::ios::app); // 使用一个文件，记录图像保存的路径
    if (out.is_open()) {
        out << "Result\\Step3\\" << outputname << endl;
        out.close();
    }
}
```

4. 针对单个切割好的字符，进行分类识别。

使用Python的sklearn库，使用MNIST数据集，进行模型的训练和测试（classifier.py）。这次使用了随机森林分类器进行训练。

i. 初始化随机森林分类器，设定经过调校的最优参数。

```python
# 初始化随机森林模型
model = RandomForestClassifier(n_estimators=256, n_jobs=-1)
```

ii. 判断是否已存在训练好的分类器。若存在，则加载分类器；若不存在，则训练模型:

```python
modelDir = './MNIST/classifier'
if(os.path.isfile(modelDir)):
    # 加载模型
    model = joblib.load(modelDir)
else:
    # 训练模型
    trainModel(model, modelDir)
```

iii. 训练模型，以及使用MNIST的测试集，测试模型准确率的过程:

```python
def trainModel(model, modelDir):
    # 读取训练数据集
    print("Loading training set.")
    # 读取图像
    with open("./MNIST/train-images.idx3-ubyte", 'rb') as imagesFile:
        magicNumber, numberOfImages, rows, cols = struct.unpack('>IIII', imagesFile.read(16))
        trainImages = np.fromfile(imagesFile, dtype=np.uint8).reshape(numberOfImages, 28 * 28)
        trainImages = trainImages.astype('int32')
    # 读取标签
    with open("./MNIST/train-labels.idx1-ubyte", 'rb') as labelsFile:
        magicNumber, numberOfItems = struct.unpack('>II', labelsFile.read(8))
        trainLabels = np.fromfile(labelsFile, dtype=np.uint8)
        trainLabels = trainLabels.astype('int32')

    # 开始训练
    print("Start training.")
    model.fit(trainImages, trainLabels)
    print("Finish training.")

    # 保存模型
    print("Saving model.")
    joblib.dump(model, modelDir)
    model = joblib.load(modelDir)

    # 读取测试数据集
    print("Loading testing set.")
    # 读取图像
    with open("./MNIST/t10k-images.idx3-ubyte", 'rb') as imagesFile:
        magicNumber, numberOfImages, rows, cols = struct.unpack('>IIII', imagesFile.read(16))
        testImages = np.fromfile(imagesFile, dtype=np.uint8).reshape(numberOfImages, 28 * 28)
        testImages = testImages.astype('int32')
    # 读取标签
    with open("./MNIST/t10k-labels.idx1-ubyte", 'rb') as labelsFile:
        magicNumber, numberOfItems = struct.unpack('>II', labelsFile.read(8))
        testLabels = np.fromfile(labelsFile, dtype=np.uint8)
        testLabels = testLabels.astype('int32')

    # 预测测试数据集，计算准确率
    predictedLabels = list(model.predict(testImages))
    print("Accuracy:", metrics.accuracy_score(testLabels, predictedLabels))
```

iv. 得到模型后，加载Step3中分割出的手写数字图像，测试模型，得到预测结果，将结果输出到excel文件中。

首先初始化一些数组，用于存放数据：

```python
imageDir = [] # 手写数字图像的路径
corner = []   # A4纸的角点

# 输出到excel的列
filename = []
corner1 = []
corner2 = []
corner3 = []
corner4 = []
studentID = []
phone = []
ID = []
```

打开两个txt文件，获取图像路径和A4纸角点坐标：

```
# 获取图像路径
print('Reading images directory.')
f = open('./imageDir.txt')
imagedir = f.readline()
while imagedir:
    imagedir = imagedir.strip('\n')
    imageDir.append(imagedir)
    imagedir = f.readline()
f.close()

# 获取A4纸角点信息
f = open('./corner.txt')
c = f.readline()
while c:
    corner.append(c)
    c = f.readline()
for i in corner:
    info = i.split()
    filename.append(info[0])
    corner1.append(info[1] + ' ' + info[2])
    corner2.append(info[3] + ' ' + info[4])
    corner3.append(info[5] + ' ' + info[6])
    corner4.append(info[7] + ' ' + info[8])
```

根据路径信息，读取图像并预测：

```
# 从各个文件夹中读取图像
for dir in imageDir:
    images = []
    print('Reading images from directory: %s' % dir)
    f = open(dir + '/imageList.txt')
    row = f.readline()
    while row:
        row = row.strip('\n')
        row = row.split()
        images.append(row)
        row = f.readline()
    f.close()
    # 预测三行
    for x in range(0, 3):
        results = ""
        # 读取单个数字并预测
        for i in images[x]:
            imageName = dir + '/' + i
            img = Image.open(imageName)
            img = img.resize((28,28))
            arr = np.array(img)
            arr = arr[:,:]
            arr = arr.reshape(1, 784)
            res = int(model.predict(arr.copy()))
            results = results + str(res)
        if x == 0: # 预测学号
            print("Predicted student id:")
            studentID.append(results)
        elif x == 1: # 预测手机号
            print("Predicted phone number:")
            phone.append(results)
        else: # 预测身份证号
            print("Predicted ID:")
            ID.append(results)
        print(results)
```

最后输出结果到excel：

```
writer = pd.ExcelWriter('Result/Step4/predict.xlsx', engine='xlsxwriter')
df = pd.DataFrame({'文件名': filename, '角点1': corner1, '角点2': corner2,
                   '角点3': corner3, '角点4': corner4, '学号': studentID,
                   '手机号': phone, '身份证号': ID})
df.to_excel(writer, sheet_name='Sheet1', index=False)
writer.save()
writer.close()
print("Saved predicted results to 'Result/Step4/predict.xlsx'.")
input("Press any key to exit.")
```

五. 测试过程及结果

由于Part1做得不是太好，下面只展示Part2的测试过程以及结果。Part1和Part2的不同及优化过程会在第六部分说明。Windows运行main.bat可以运行全部测试。测试数据集：去除一些坏样本（写中文、涂改等）后，一共有76张图像。

1. 采用边缘检测或者图像分割的方法获取图像的边缘，并计算图像的四个角点，完成图像矫正；

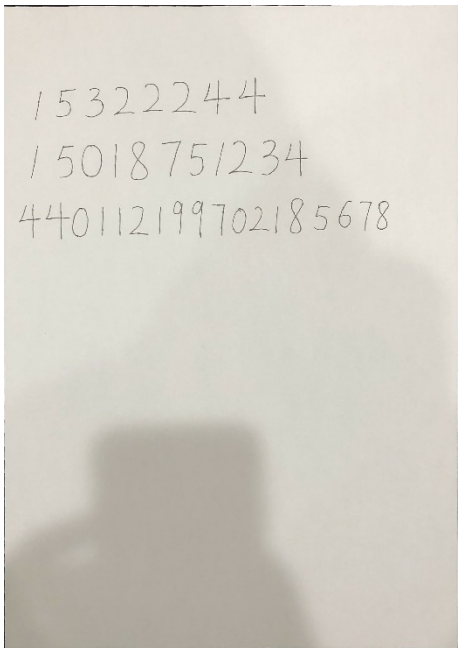
编译：`g++ Step1.cpp Canny.cpp Hough.cpp Warping.cpp -lgdi32 -o Step1`

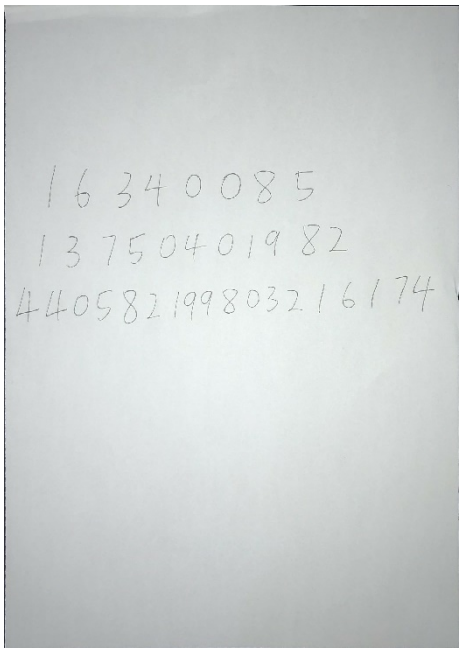
然后运行Step1.exe。由于矫正后的A4纸分辨率较高，单张图片的矫正过程大概需要8秒（不同的计算机可能有不同的结果）。降低矫正的分辨率可以提升速度。

```
D:\Xungerrrr\SYSU\大三上\计算机视觉\作业8 part2>.\Step1.exe
Loading image './ImageData/12330029.jpg'.
Starting Canny edge detection.
Converting the image to grey scale.
Smoothing the image using a gaussian kernel.
   Computing the gaussian smoothing kernel.
      The kernel has 11 elements.
      The filter coefficients are:
      kernel[0] = 0.008812
      kernel[1] = 0.027144
      kernel[2] = 0.065114
      kernel[3] = 0.121649
      kernel[4] = 0.176998
      kernel[5] = 0.200565
      kernel[6] = 0.176998
      kernel[7] = 0.121649
      kernel[8] = 0.065114
      kernel[9] = 0.027144
      kernel[10] = 0.008812
   Bluring the image in the X-direction.
   Bluring the image in the Y-direction.
Computing the X and Y first derivatives.
   Computing the X-direction derivative.
   Computing the Y-direction derivative.
Computing the magnitude of the gradient.
Doing the non-maximal suppression.
Doing hysteresis thresholding.
The input low and high fractions of 0.97 and 0.97 computed to
magnitude of the gradient threshold values of: 469 483
Deleting short edges.
Doing hough transformation.
Finding peak points.
   Finding with threshold: 200
Intersections are:
(98, 128)
(98, 1511)
(1061, 128)
(1085, 1494)
Warping the image.
   The mappings of corners from a standard A4 paper to the image are:
   (0, 0) -> (98, 128)
   (2479, 0) -> (1061, 128)
   (0, 3507) -> (98, 1511)
   (2479, 3507) -> (1085, 1494)
Saving warped image to 'Result/Step1/12330029.jpg
```

输出结果在Result/Step1/目录下。

部分输出结果如下：





2. 采用图像分割（二值化）的方法，获取图像中的手写字符，输出二值化结果；

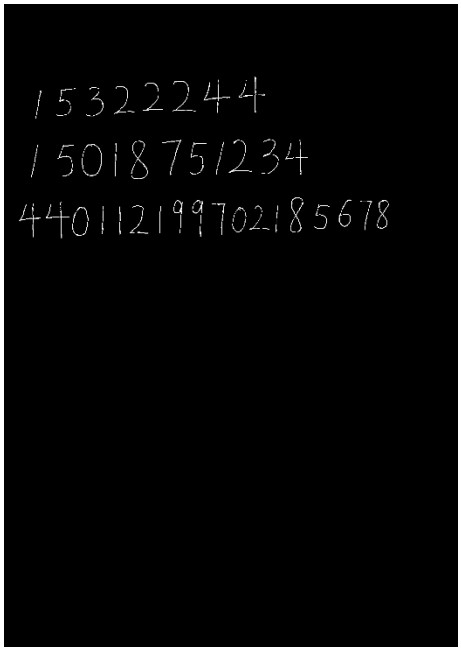
编译：
```
g++ Step2.cpp OSTU.cpp -lgdi32 -o Step2
```

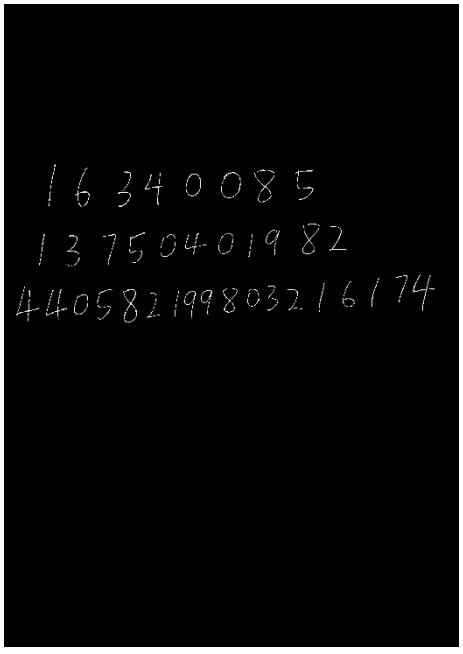
然后运行Step2.exe。每张图像大约需要处理1秒。

```
D:\Xungerrrr\SYSU\大三上\计算机视觉\作业8 part2>.\Step2.exe
Loading image './Result/Step1/12330029.jpg'.
Saving binary image to 'Result/Step2/12330029.jpg
Loading image './Result/Step1/12353045.jpg'.
Saving binary image to 'Result/Step2/12353045.jpg
```

输出结果在Result/Step2/目录下。

部分输出结果如下：





3. 针对二值化图像，对Y方向投影切割出各个行的图像，例如学号切成单个的图像，

手机号和身份证号也如此。根据上面的结果，针对行图像（如学号图像），对X方向做投影切割，切割出单个字符；

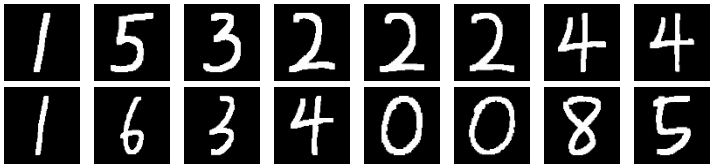
编译：`g++ Step3.cpp Cutting.cpp -lgdi32 -o Step3`

然后运行Step3.exe。这部分运行速度不是很快，经过观察，时间主要花费在保存图片的过程，而不是切割图片的过程，这说明切割的速度是没有问题的。还不是很清楚保存速度慢的具体原因。

```
D:\Xungerrrr\SYSU\大三上\计算机视觉\作业8 part2>.\Step3.exe
Loading image './Result/Step2/12330029.jpg'.
Saving cut images to 'Result/Step3/12330029'.
Loading image './Result/Step2/12353045.jpg'.
Saving cut images to 'Result/Step3/12353045'.
Loading image './Result/Step2/13331217.jpg'.
Saving cut images to 'Result/Step3/13331217'.
Loading image './Result/Step2/15322244.jpg'.
Saving cut images to 'Result/Step3/15322244'.
Loading image './Result/Step2/15331029.jpg'.
Saving cut images to 'Result/Step3/15331029'.
Loading image './Result/Step2/15331035.jpg'.
Saving cut images to 'Result/Step3/15331035'.
Loading image './Result/Step2/15331046.jpg'.
Saving cut images to 'Result/Step3/15331046'.
Loading image './Result/Step2/15331052.jpg'.
Saving cut images to 'Result/Step3/15331052'.
Loading image './Result/Step2/15331180.jpg'.
Saving cut images to 'Result/Step3/15331180'.
Loading image './Result/Step2/15331220.jpg'.
Saving cut images to 'Result/Step3/15331220'.
```

输出结果在Result/Step3/目录下。

部分输出结果如下：



4. 针对单个切割好的字符，进行分类识别。

运行py classifier.py。若没有模型文件，则会自动训练模型，然后用模型进行预测。

```
D:\Xungerrrr\SYSU\大三上\计算机视觉\作业8 part2>py classifier.py
Loading training set.
Start training.
Finish training.
Saving model.
Loading testing set.
Accuracy: 0.9708
Reading images directory.
Reading images from directory: Result\Step3\12330029
Predicted student id:
12330021
Predicted phone number:
18520124317
Predicted ID:
340826117801086610
```

预测结束后，会将结果输出到Result/Step4/目录下的predict.xlsx中。

六. 结果分析和优化

1. 边缘检测、边缘直线拟合和图像校正；

Part1：

一开始，采用OSTU单阈值法进行图像分割，提取边缘。该方法引入的噪声太多，导致霍夫变换很容易受到噪声的影响，不能完整地拟合出4条边缘。后来，采用Canny算子进行边缘检测，通过调整标准差、双阈值等参数，能够很方便地去除噪声，效果比较好，但是速度比OSTU稍慢。

Part2:

由于Part2的图像较多，我对代码进行了优化。首先，编写函数自动获取所有图像的文件名，避免了手动输入文件名的麻烦。然后，我对霍夫变换的代码进行了改进，使得当霍夫变换拟合出的边缘不等于4条时，会自动调整阈值大小，多次重复拟合，最终获得4条拟合的边缘，避免了因边缘数量不对导致Warping的错误，也避免了对每张图手动调参的麻烦。

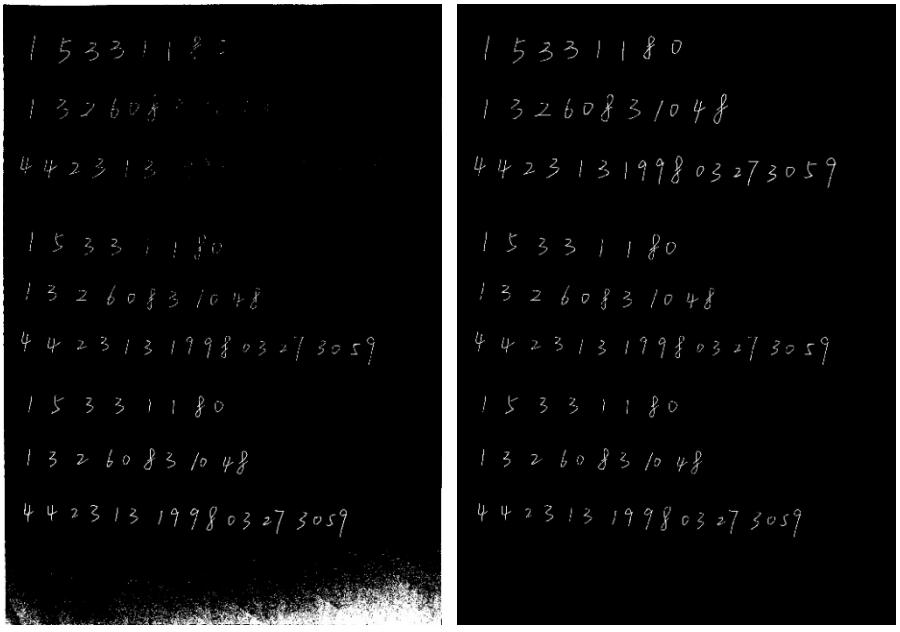
```
void Hough::find peak points() {
    while (peaks.size() != 4) {
        if (peaks.size() > 4) houghThreshold += 5;
        else if (peaks.size() != 0) houghThreshold -= 5;
        peaks.clear();
        cout << "   Finding with threshold: " << houghThreshold << endl;
```

2. 二值化

Part1和Part2:

实验一开始，我尝试使用全局阈值进行二值化，发现容易受光照影响。后来，改用局部阈值的方法，效果要远优于前者。缺点是处理速度会变慢。

全局阈值（左）和局部阈值（右）对比：



3. 数字切割

Part1：

在完成Part1的切割部分时，我只是简单地找到包裹数字的最小区域，然后将其切割下来。经过观察，MNIST数据集的图像是正方形的，数字与图像边界存在一定的距离（margin），而且数字笔画比较粗。由于我在Part1切割出来的图像不具有这样的特点，因此识别率比较低（而模型在MNIST测试集中的准确率高达0.97）。

以下是Part1的15331180切割结果：

预测情况（比较红框中的数字，可以看到准确率非常低，和随机没有太大区别）：

	A	B	C	D	E	F	G	H
1	文件名	角点1	角点2	角点3	角点4	学号	手机号	身份证号
2	15331029.jpg	354, 575	2401, 504	453, 3419	2501, 3383	07415717	81417105171	740755077501775701
3	15331046.jpg	236, 547	2474, 468	341, 3563	2474, 3563	11551565	65427662175	616538871450261156
4	15331052.jpg	518, 484	2628, 781	101, 3455	2211, 3752	1755176	15557555755	552475177717151711
5	15331180.jpg	181, 441	2568, 441	239, 3768	2568, 3727	14771817	17757451754	1175172771757555757
6	15331344.jpg	171, 615	2409, 615	171, 3782	2409, 3743	17470774	07713027573	775575077477771774
7	15331347.jpg	902, 652	2767, 111	780, 3341	2119, 3926	32776577	35757847517	747214177774747267
8	15331348.jpg	2670, 36	3427, 1737	189, 1193	1076, 2935	14443444	84614711414	1777717777710777114
9	15331351.jpg	115, 633	2373, 633	171, 3844	2428, 3765	27471775	71717777777	774777777771717714
10	15331353.jpg	152, 541	2504, 500	210, 3866	2561, 3783	24453434	07777372750	4747473277466442472
11	15331364.jpg	171, 629	2322, 554	274, 3604	2374, 3568	87778721	87777777473	778788747777777747

Part2：

在Part2中，我对切割过程进行了改进：首先，将图像填补成具有一定margin的正方形图像。然后，根据图像的大小，对数字进行适当的膨胀，以达到加粗的目的。结果，识别的准确率有了显著的提高。

以下是Part2的15331180切割结果：

预测情况（准确率非常高，红框内学号都是100%准确，其他大部分数字都能够识别正确）：

	A	B	C	D	E	F	G	H
1	文件名	角点1	角点2	角点3	角点4	学号	手机号	身份证号
2	12330029.jpg	98, 128	1061, 128	98, 1511	1085, 1494	12330021	18520124317	340826117801086610
3	12353045.jpg	135, 204	1004, 158	135, 1420	1027, 1451	12353015	13521256111	211102111105151514
4	15322244.jpg	167, 167	1094, 184	120, 1519	1094, 1536	15322244	15018151234	4401121311021856618
5	15331029.jpg	354, 575	2401, 504	453, 3419	2501, 3383	15331029	13821813332	4417211116122230 12
6	15331035.jpg	200, 170	1024, 141	200, 1374	1087, 1343	15331035	13331153615	510228113255280826
7	15331046.jpg	236, 547	2474, 468	341, 3563	2474, 3563	15311086	13611154131	885101111810252118
8	15331052.jpg	518, 484	2628, 781	101, 3455	2211, 3752	15351052	15560563783	550426137112221012
9	15331180.jpg	181, 441	2568, 441	239, 3768	2568, 3727	15331180	15260831081	4823131111032230 57
10	15331220.jpg	34, 68	1036, 50	34, 1502	1061, 1484	15331220	15521220011	513102197611225 11
11	15331344.jpg	171, 615	2409, 615	171, 3782	2409, 3743	15331344	15521125538	350102133502237321
12	15331347.jpg	902, 652	2767, 1117	80, 3341	2119, 3926	15331391	18350182323	350105119105042118
13	15331351.jpg	115, 633	2373, 633	171, 3844	2428, 3765	15331351	13219214506	445202200001010058
14	15331353.jpg	152, 541	2504, 500	210, 3866	2561, 3783	15331333	13222132961	45625210324663315
15	15331364.jpg	171, 629	2322, 554	274, 3604	2374, 3568	15331361	13632552831	911281239503013135

此外，由于我只是简单地进行X和Y方向的投影切割，因此我的代码不能处理数字XY坐标交错的情况，如下图。连通区域算法应该是一种可行的解决办法。

4. 训练和识别

在作业7中，我使用了决策树分类器和AdaBoost分类器的组合，进行模型的训练和预测，准确率只有0.87左右，不是特别理想。

```
Loading training set.
Start training.
Finish training.
Saving model.
Loading testing set.
Accuracy: 0.8778
```

在本项目中，我使用了更适合MNIST数据集的随机森林分类器，n_estimators设为256，进行训练和测试。使用MNIST测试集进行测试，准确率能够达到0.97，远超之前的水平。

```
Loading training set.
Start training.
Finish training.
Saving model.
Loading testing set.
Accuracy: 0.9708
```