Final Project 实验报告及测试文档

数据科学与计算机学院 梁育诚 16340133

一、实验内容

普通 A4 打印纸，上面有手写的如下信息：

1. 学号
2. 手机号
3. 身份证号

所有输入格式一致，数字号码不粘连，但是拍照时可能角度不正。

输出：

1. 根据标准流程输出每一个主要步骤的结果，包括 A4 纸张的矫正结果，行数据（包括学号、手机号、身份证号）的切割，单个字符的切割结果。
2. 对上面的 A4 纸的四个角、学号、手机号、身份证号进行识别，识别记过保存到 Excel 表格，对于手写体数字字符的训练数据可以使用 MNIST。

二、实验步骤及原理

这次是综合项目，综合运用前面学到的各种知识进行手写体数字识别。过程如下：

1. 采用图像分割的方法获取 A4 纸图像的边缘，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。

4. 根据上面第三步的结果，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。

5. 对单个字符的图像进行优化，做扩张和腐蚀。

6. 针对单个切割好的字符，进行分类识别。

三、实验过程

1. 首先使用图像分割的方法，从图片中提取出 A4 纸，并根据 A4 纸的四个角点对其进行矫正。图像分割中使用的是 OSTU 阈值法，代码用的是第七次作业，基本不需要额外修改就可以直接使用。

图像分割关键代码：

```cpp
// Do segmentation
void Segmentation::segment() {
  int totalPixels = img._width * img._height;
  int gray = 0, grayBackground = 0, grayFrontground = 0;
  float wBackground = 0, wFrontground = 0;
  float g_best = 0;
  float g = 0;
  int numBackground = 0, numFrontground = 0;

  getHistogram();
  for (int i = 0; i < 256; i++) {
    gray += i * histogram[i];
  }

  for (int i = 0; i < 256; i++) {
    grayBackground += histogram[i] * i; // 背景总灰度值
    grayFrontground = gray - grayBackground; // 前景总灰度值
    numBackground += histogram[i]; // 背景pixel数量
    numFrontground = totalPixels - numBackground; // 前景pixel数量
    wBackground = numBackground * 1.0 / totalPixels; // 背景pixel占比
    wFrontground = 1 - wBackground; // 前景pixel占比

    // 无法区分前景和背景
    if (wBackground == 0 || wFrontground == 0)
      continue;
```

```cpp
    // 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;
}
```

```cpp
// 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;
  }
  */
}
```

2. 然后对矫正后的 A4 纸图片进行边缘检测，目的是检测出图片中的数字。如果直接使用图像分割（二值化）的方法，受光照和背景影响较大，很难提取出清晰的数字部分。这里也不需要完整的使用 canny，因为我们的目的是提取出清晰的边缘，过滤掉有干扰的背景和光照部分，因此只需要对图片求梯度，梯度值大于一定阈值的点我们则认为是边缘点（黑色），而其余点则认为是背景点（白色）。这里求梯度用的是 sobel 算子。

```
// Detect Edge
CImg<float> ImageSplit::getEdge(CImg<float> img) {
  edgeImg = CImg<float>(img._width, img._height, 1, 1, 0);
  // 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 = img;
  gradient_x = gradient_x.get_convolve(sobelx);
  CImg<float> gradient_y = img;
  gradient_y = gradient_y.get_convolve(sobely);

  cimg_forXY(img, i, j) {
    double grad = sqrt(gradient_x(i, j) * gradient_x(i, j) + gradient_y(i, j) * gradient_y(i, j));
    if (grad > 110) {
      img(i, j, 0) = 0;
      img(i, j, 1) = 0;
      img(i, j, 2) = 0;
    }
    else {
      img(i, j, 0) = 255;
      img(i, j, 1) = 255;
      img(i, j, 2) = 255;
    }
  }
  //img.display("edge", false);
  return img;
}
```

3. 边缘检测后，就对图片进行二值化处理。因为后来发现效果不佳才加入了

第二步的边缘检测，而第二步实际上也是做了一个分割，所以这一步有点多余。但可以在这里对图片的边框部分进行一个简单的处理。因为 A4 纸的矫正也不是完全地正确，会有边的部分被认为是边缘点，因此我们直接设置一个 gap，在这个 gap 中的部分都直接认为是背景部分，这样就能排除纸张边缘的干扰。

```
// 图像二值化处理
CImg<float> ImageSplit::convertToBinaryImg(CImg<float> warpingResult) {
  binaryImg = CImg<float>(warpingResult._width, warpingResult._height, 1, 1, 0);
  cimg_forXY(binaryImg, x, y) {
    int intensity = warpingResult(x, y, 0);
    //先去掉黑色边缘
    if (x <= BoundaryRemoveGap || y <= BoundaryRemoveGap
        || x >= warpingResult._width - BoundaryRemoveGap || y >= warpingResult._height - BoundaryRemoveGap) {
      binaryImg(x, y, 0) = 255; // 白色
    }
    else {
      if (intensity < BinaryGap)   // 小于阈值认为是边缘
        binaryImg(x, y, 0) = 0;
      else
        binaryImg(x, y, 0) = 255;
    }
  }
  return binaryImg;
}
```

4. 在 Y 方向上做投影。求出图片在 Y 方向上的灰度直方图，因为数字是一行一行的，因此灰度直方图会有许多峰和谷。我们需要找到谷的位置，将其作为切割的点。比较好的做法是，求出每一个峰的上界与下界，再求每两个峰之间谷，这种做法得出来的效果比较好。然后就根据切割点，对图片进行切割，得到的是每一行的图片（barImage）。

```
void ImageSplit::findDividingLine() {
  int lineColor[3]{255, 0, 0};
  histogramImg = CImg<float>(binaryImg._width, binaryImg._height, 1, 3, 255);
  dividingImg = binaryImg;
  vector<int> inflectionPoints; // 拐点
  cimg_forY(histogramImg, y) {
    int blackPixel = 0;
    // 统计黑点个数
    cimg_forX(binaryImg, x) {
      if (binaryImg(x, y, 0) == 0)
        blackPixel++;
      /*
      if (y < binaryImg._height-2 && binaryImg(x, y + 2, 0) == 0)
        blackPixel++;
        */
    }
    cimg_forX(histogramImg, x) {
      if (x < blackPixel) {
        histogramImg(x, y, 0) = 0;
        histogramImg(x, y, 1) = 0;
        histogramImg(x, y, 2) = 0;
      }
    }
```

```
    // 求Y方向直方图，谷的最少黑色像素个数为0
    // 判断是否为拐点
    if (y > 0) {
      // 下界
      if (blackPixel <= 0 && histogramImg(0, y - 1, 0) == 0) {
        int yTemp = inflectionPoints[inflectionPoints.size() - 1];
        int count = 0;
        for (int i = yTemp; i < y; i++) {
          if (histogramImg(0, i, 0) == 0) {
            count++;
          }
        }

        // 统计两条线之间的黑点个数
        if (count >= 25 && (y - yTemp) > 1) {
          inflectionPoints.push_back(y);
          histogramImg.draw_line(0, y, histogramImg._width - 1, y, lineColor);
        }
      }
```

```
      // 上界
      else if (blackPixel > 0 && histogramImg(0, y - 1, 0) != 0) {
        if (inflectionPoints.size() >= 1) {
          int yTemp = inflectionPoints[inflectionPoints.size() - 1];
          int count = 0;
          for (int i = yTemp; i < y; i++) {
            if (histogramImg(0, i, 0) == 0) {
              count++;
            }
          }
          // 统计两条线之间的黑点个数
          if (count == 0 && (y-1 - yTemp) > 2) {
            inflectionPoints.push_back(y - 1);
            histogramImg.draw_line(0, y - 1, histogramImg._width - 1, y - 1, lineColor);
          }
        }
        else {
          inflectionPoints.push_back(y - 1);
          histogramImg.draw_line(0, y - 1, histogramImg._width - 1, y - 1, lineColor);
        }
      }
    }
  }
  histogramImg.display("histogramImg Point");
  dividePoints.push_back(Point1(0, -1));
  // 两拐点中间做分割
  if (inflectionPoints.size() > 2) {
    for (int i = 1; i < inflectionPoints.size() - 1; i = i + 2) {
      int dividePoint = (inflectionPoints[i] + inflectionPoints[i + 1]) / 2;
      dividePoints.push_back(Point1(0, dividePoint));
      histogramImg.draw_line(0, dividePoint, histogramImg._width - 1, dividePoint, lineColor);
    }
  }
  histogramImg.display("histogramImg Line");
  dividePoints.push_back(Point1(0, binaryImg._height - 1));
}
```

```
// 根据行分割线划分图片
void ImageSplit::divideIntoBarItemImg() {
  vector<Point1> tempDivideLinePointSet;
  for (int i = 1; i < dividePoints.size(); i++) {
    int barHeight = dividePoints[i].y - dividePoints[i - 1].y;
    int blackPixel = 0;
    // 初始化切割图
    CImg<float> barItemImg = CImg<float>(binaryImg._width, barHeight, 1, 1, 0);
    // 复制对应部分
    cimg_forXY(barItemImg, x, y) {
      barItemImg(x, y, 0) = binaryImg(x, dividePoints[i - 1].y + 1 + y, 0);
      if (barItemImg(x, y, 0) == 0)
        blackPixel++;
    }

    double blackPercent = (double)blackPixel / (double)(binaryImg._width * barHeight);
    // 只有当黑色像素个数超过图像大小一定比例0.005时，才可视作有数字
    if (blackPercent > 0.005) {
      barItemImg.display(("barItemImg" + to_string(i)).c_str());
      string fileName = "./result/row/row";
      fileName = fileName + to_string(i) + ".bmp";
      barItemImg.save(fileName.c_str());
```

```
      // 横向切割
      vector<int> dividePosXset = getDivideLineXofSubImage(barItemImg);
      vector<CImg<float>> rowItemImgSet = getRowItemImgSet(barItemImg, dividePosXset);
      for (int j = 0; j < rowItemImgSet.size(); j++) {
        subImageSet.push_back(rowItemImgSet[j]);
        tempDivideLinePointSet.push_back(Point1(dividePosXset[j], dividePoints[i - 1].y));
      }
      if (i > 1) {
        histogramImg.draw_line(0, dividePoints[i - 1].y,
          histogramImg._width - 1, dividePoints[i - 1].y, lineColor);
        dividingImg.draw_line(0, dividePoints[i - 1].y,
          histogramImg._width - 1, dividePoints[i - 1].y, lineColor);
      }
      // 绘制竖线
      for (int j = 1; j < dividePosXset.size() - 1; j++) {
        dividingImg.draw_line(dividePosXset[j], dividePoints[i - 1].y,
          dividePosXset[j], dividePoints[i].y, lineColor);
      }
    }
  }
  dividingImg.display("dividingImg");
  dividePoints.clear();
  for (int i = 0; i < tempDivideLinePointSet.size(); i++) {
    dividePoints.push_back(tempDivideLinePointSet[i]);
  }
}
```

5. 对于每一张 barImage，在 X 方向上做投影，求出图片在 X 方向上的灰度直方图。与第四步类似，每一个数字在直方图中都对应一个峰，使用同样的方法找到谷的位置，也就是切割点所在的位置。根据切割点，对图片进行切割，获得单个数字的图片。这里还需要对图片颜色置反，即边缘点置为白色，背景点置为黑色，这是为了和 MNIST 数据集的数据格式保持一致。

```
// 根据X方向直方图判断真实的拐点
vector<int> ImageSplit::getInflectionPosXs(const CImg<float>& XHistogramImage) {
  vector<int> resultInflectionPosXs, tempInflectionPosXs;
  int totalDist = 0, dist = 0;
  // 查找拐点
  //XHistogramImage.display("XHistogramImage");
  // 置反，与MNIST数据一致
  cimg_forX(XHistogramImage, x) {
    if (x >= 1) {
      // 白转黑
      if (XHistogramImage(x, 0, 0) == 0 && XHistogramImage(x - 1, 0, 0) == 255)
        tempInflectionPosXs.push_back(x - 1);
      // 黑转白
      else if (XHistogramImage(x, 0, 0) == 255 && XHistogramImage(x - 1, 0, 0) == 0)
        tempInflectionPosXs.push_back(x);
    }
  }
  for (int i = 2; i < tempInflectionPosXs.size() - 1; i = i + 2) {
    int tempdist = tempInflectionPosXs[i] - tempInflectionPosXs[i - 1];
    if (tempdist <= 0)
      tempdist--;
    totalDist += tempdist;
  }
```

```
  // 计算间距平均距离
  dist += (tempInflectionPosXs.size() - 2) / 2;
  int avgDist = 0;
  if (dist != 0) {
    avgDist = totalDist / dist;
  }

  resultInflectionPosXs.push_back(tempInflectionPosXs[0]); //头
  // 当某个间距大于平均距离的一定倍数时，视为分割点所在间距
  for (int i = 2; i < tempInflectionPosXs.size() - 1; i = i + 2) {
    int dist = tempInflectionPosXs[i] - tempInflectionPosXs[i - 1];
    if (dist > avgDist * 4) {
      resultInflectionPosXs.push_back(tempInflectionPosXs[i - 1]);
      resultInflectionPosXs.push_back(tempInflectionPosXs[i]);
    }
  }
  resultInflectionPosXs.push_back(tempInflectionPosXs[tempInflectionPosXs.size() - 1]); //尾
  return resultInflectionPosXs;
}
```

```
// 获取每一行的每个数字的竖直分割线
vector<int> ImageSplit::getDivideLineXofSubImage(const CImg<float>& subImg) {
  // 先绘制X方向灰度直方图
  CImg<float> XHistogramImage = CImg<float>(subImg._width, subImg._height, 1, 3, 255);
  cimg_forX(subImg, x) {
    int blackPixel = 0;
    cimg_forY(subImg, y) {
      if (subImg(x, y, 0) == 0)
        blackPixel++;
    }
    // 对于每一列x，只有黑色像素多于一定值，才绘制在直方图上（确认是数字）
    // 求X方向直方图，谷的最少黑色像素个数
    if (blackPixel >= 4) {
      cimg_forY(subImg, y) {
        if (y < blackPixel) {
          XHistogramImage(x, y, 0) = 0;
          XHistogramImage(x, y, 1) = 0;
          XHistogramImage(x, y, 2) = 0;
        }
      }
    }
  }
```

```
  vector<int> InflectionPosXs = getInflectionPosXs(XHistogramImage);    //获取拐点
  for (int i = 0; i < InflectionPosXs.size(); i++) {
    XHistogramImage.draw_line(InflectionPosXs[i], 0, InflectionPosXs[i], XHistogramImage._height - 1, lineColor)
  }

  // 两拐点中间做分割
  vector<int> dividePosXs;
  dividePosXs.push_back(-1);
  if (InflectionPosXs.size() > 2) {
    for (int i = 1; i < InflectionPosXs.size() - 1; i = i + 2) {
      int divideLinePointX = (InflectionPosXs[i] + InflectionPosXs[i + 1]) / 2;
      dividePosXs.push_back(divideLinePointX);
    }
  }
  dividePosXs.push_back(XHistogramImage._width - 1);
  return dividePosXs;
}
```

```
// 分割行子图，得到单个数字图
vector<CImg<float>> ImageSplit::getRowItemImgSet(const CImg<float>& lineImg, vector<int> _dividePosXset) {
  vector<CImg<float>> result;

  for (int i = 1; i < _dividePosXset.size(); i++) {
    int rowItemWidth = _dividePosXset[i] - _dividePosXset[i - 1];
    // 初始化单个数字图
    CImg<float> rowItemImg = CImg<float>(rowItemWidth, lineImg._height, 1, 1, 0);
    // 复制对应部分
    cimg_forXY(rowItemImg, x, y) {
      rowItemImg(x, y, 0) = lineImg(x + _dividePosXset[i - 1] + 1, y, 0);
    }
    result.push_back(rowItemImg);
    //rowItemImg.display(("rowItemImg" + to_string(i)).c_str());

  }
  return result;
}
```

6. 图片膨胀与腐蚀。由于手写体数字一般都比较细，通过切割的方法提取出

来后非常难被识别，因此需要对图片进行基本的处理。膨胀与腐蚀针对的都是图片中的高亮部分，即白色部分。膨胀是将高亮部分向 XY 两个方向以不同的规则延伸，腐蚀是指将高亮部分按照一定规则去除。这两个操作的本质都是对目标点周围的像素做判断，然后决定是否碰撞或腐蚀。

```
// 膨胀
void ImageSplit::dilateImg(int barItemIndex) {
  // 膨胀Dilation -X-X-X-XYY方向
  CImg<float> answerXXY = CImg<float>(subImageSet[barItemIndex]._width,
                                      subImageSet[barItemIndex]._height, 1, 1, 0);
  cimg_forXY(subImageSet[barItemIndex], x, y) {
    answerXXY(x, y, 0) = getDilateXXY(subImageSet[barItemIndex], x, y);
  }

  // 膨胀Dilation -X-X-X-XYY方向
  CImg<float> answerXXY2 = CImg<float>(answerXXY._width, answerXXY._height, 1, 1, 0);
  cimg_forXY(answerXXY, x, y) {
    answerXXY2(x, y, 0) = getDilateXXY(answerXXY, x, y);
  }

  // 膨胀Dilation XY方向
  CImg<float> answerXY = CImg<float>(answerXXY2._width, answerXXY2._height, 1, 1, 0);
  cimg_forXY(answerXXY2, x, y) {
    answerXY(x, y, 0) = getDilateXY(answerXXY2, x, y);
  }

  cimg_forXY(subImageSet[barItemIndex], x, y) {
    subImageSet[barItemIndex](x, y, 0) = answerXY(x, y, 0);
  }
}
```

```
// XY方向的正扩张
int ImageSplit::getDilateXY(const CImg<float>& Img, int x, int y) {
  int intensity = Img(x, y, 0);
  if (intensity == 255) { // 若中间点为白色
    // X和Y都是一个单位
    for (int i = -1; i <= 1; i++) {
      for (int j = -1; j <= 1; j++) {
        // 保证不越界
        if (0 <= x + i && x + i < Img._width && 0 <= y + j && y + j < Img._height) {
          // 水平或垂直方向
          if (i != -1 && j != -1 || i != 1 && j != 1 || i != 1 && j != -1 || i != -1 && j != 1)
            if (Img(x + i, y + j, 0) == 0) {
              intensity = 0; // 扩张
              break;
            }
        }
      }
      if (intensity != 255) {
        break;
      }
    }
  }
  return intensity;
}
```

```cpp
// X方向2个单位的负扩张，Y方向1个单位的正扩张
int ImageSplit::getDilateXXY(const CImg<float>& Img, int x, int y) {
  int intensity = Img(x, y, 0);
  if (intensity == 255) {    //若中间点为白色
    int blackAccu = 0;
    // 一个单位
    for (int i = -1; i <= 1; i++) {
      if (0 <= y + i && y + i < Img._height) {    //竖直方向
        if (Img(x, y + i, 0) == 0)
          blackAccu++;
      }
    }
    // 两个单位
    for (int i = -2; i <= 2; i++) {
      if (0 <= x + i && x + i < Img._width) {     //水平方向
        if (Img(x + i, y, 0) == 0)
          blackAccu--;
      }
    }
    if (blackAccu > 0) {
      intensity = 0;  // 扩张
    }
  }
  return intensity;
}
```

```cpp
// 对单个数字图像做Y方向腐蚀操作
CImg<float> ImageSplit::eroseImg(CImg<float>& Img) {
  CImg<float> result = CImg<float>(Img._width, Img._height, 1, 1, 0);
  cimg_forXY(Img, x, y) {
    result(x, y, 0) = Img(x, y, 0);
    if (Img(x, y, 0) == 255) {
      // 上方
      if (y - 1 >= 0) {
        if (Img(x, y - 1, 0) == 0)
          result(x, y, 0) = 0;
      }
      // 下方
      if (y + 1 < Img._height) {
        if (Img(x, y + 1, 0) == 0)
          result(x, y, 0) = 0;
      }
    }
  }
  return result;
}
```

7. 区域连通。这一步是为了解决数字中多部分相互断裂的问题。标记算法是区域连通算法的一种，它对每一部分进行标记，然后不断地向以后标记添加区域或新增标记来实现。

```
// 连通区域标记算法
void ImageSplit::connectedRegionsTagging(int barItemIndex) {
  tagImg = CImg<float>(subImageSet[barItemIndex]._width, subImageSet[barItemIndex]._height, 1, 1, 0);
  tagAccumulate = -1;

  cimg_forX(subImageSet[barItemIndex], x)
    cimg_forY(subImageSet[barItemIndex], y) {
    // 第一行和第一列
    if (x == 0 || y == 0) {
      int intensity = subImageSet[barItemIndex](x, y, 0);
      if (intensity == 0) {
        addNewTag(x, y, barItemIndex);
      }
    }
    // 其余的行和列
    else {
      int intensity = subImageSet[barItemIndex](x, y, 0);
      if (intensity == 0) {
        // 检查正上、左上、左中、左下这四个邻点
        int minTag = INT_MAX; //最小的tag
        Point1 minTagPointPos(-1, -1);
        // 先找最小的标记
        findMinTag(x, y, minTag, minTagPointPos, barItemIndex);
```

```
        // 当正上、左上、左中、左下这四个邻点有黑色点时，与minTag合并；
        if (minTagPointPos.x != -1 && minTagPointPos.y != -1) {
          mergeTagImageAndList(x, y - 1, minTag, minTagPointPos, barItemIndex);
          for (int i = -1; i <= 1; i++) {
            if (y + i < subImageSet[barItemIndex]._height)
              mergeTagImageAndList(x - 1, y + i, minTag, minTagPointPos, barItemIndex);
          }
          // 当前位置
          tagImg(x, y, 0) = minTag;
          Point1 cPoint(x + dividePoints[barItemIndex].x + 1, y + dividePoints[barItemIndex].y + 1);
          pointPosListSet[minTag].push_back(cPoint);

        }
        // 否则，作为新类
        else {
          addNewTag(x, y, barItemIndex);
        }
      }
    }
  }
}
```

8. 扩充边缘。获取单个数字的包围盒，在其边缘扩充一定的宽度，使得图片与 MNIST 数据集中的数据尽可能地相似，这样能提高识别率。这部分比较简单，与保存图片的代码写在了一起，这里就不放代码了。

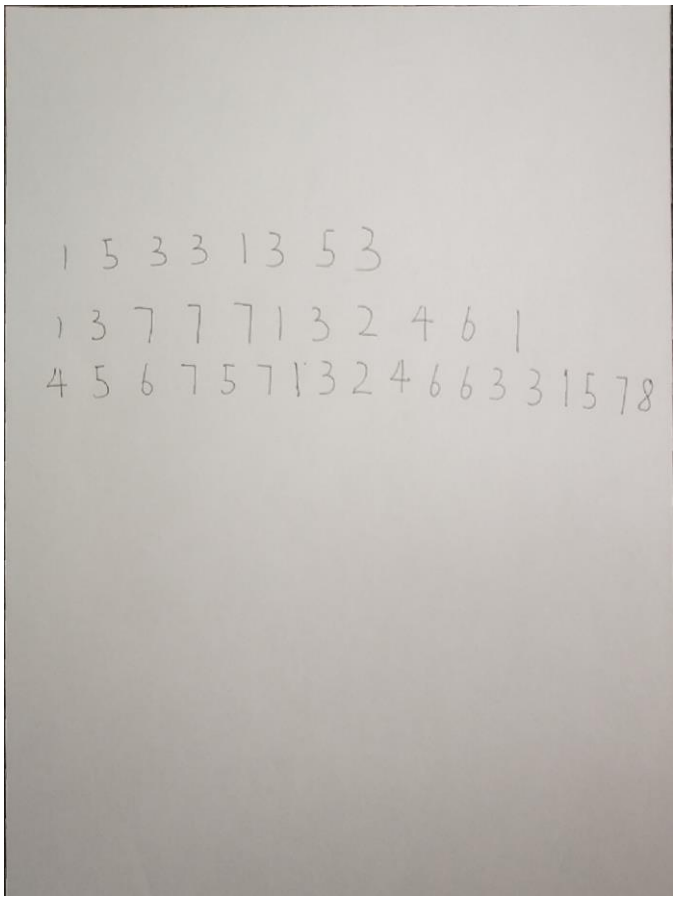
9. 训练数字识别模型，这里使用了 Adaboost 和 SVM 两个模型（在一个模型识别效果很差的时候换另一个模型尝试一下），使用 MNIST 数据集进行训练，得到模型后，将每张数字图片给模型识别，统计结果。

四、测试及分析

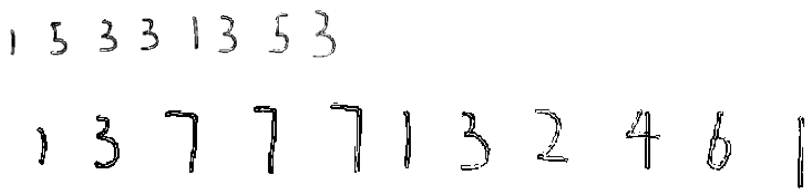
详细的测试结果在 Excel 表格中，第一部分的 10 个测试在【识别结果 1.xlsx】中，第二部分的测试在【识别结果 2.xlsx】中。每张图片切割后的单个数字都存放在【./result/SingleNumImg】对应的文件夹中。这里只选取个别例子来分析。

这里选用第一个测试数据作为例子：

A4 矫正后图片：

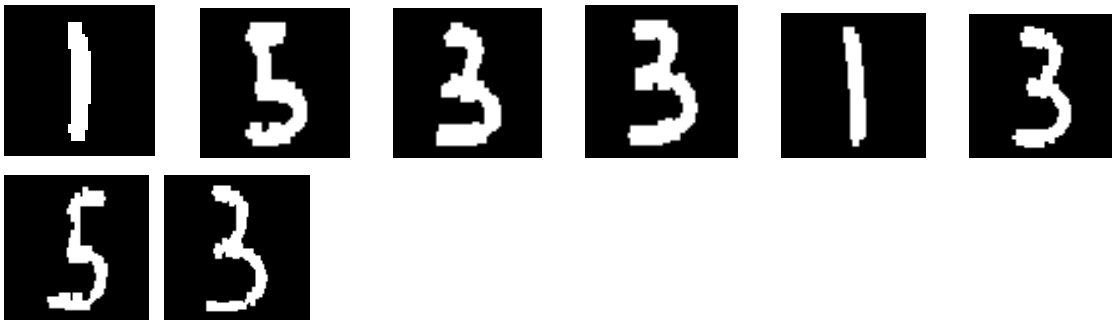

1 5 3 3 1 3 5 3
1 3 7 7 7 1 3 2 4 6 1
4 5 6 7 5 7 1 3 2 4 6 6 3 3 1 5 7 8

Y 方向切割后图片：

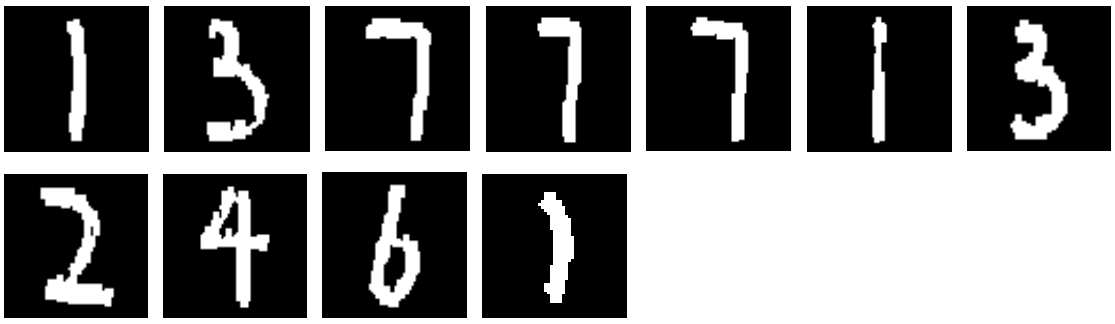

1 5 3 3 1 3 5 3
1 3 7 7 7 1 3 2 4 6 1

切割成单个数字，连通、膨胀并腐蚀：

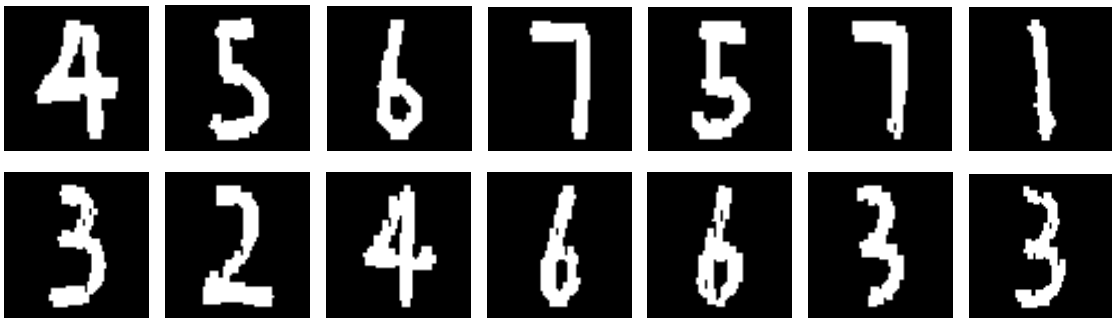
学号：



电话：



身份证：



分析：可以看出切割之后每个用于测试的单个数字都是比较清楚的，而且数字有一定宽度，轮廓比较清晰，边缘也留有一定空隙，与 MNIST 数据集非常相似

五、难点、思考、总结与体会

难点：1. 首先是二值化处理。A4 纸矫正是第七次作业的部分，直接用效果就不错。按照老师给的步骤，对矫正后的 A4 纸进行图像分割，但是发现效果很差。原因是，手写体的数字比较细，数字部分和背景部分的像素值差别不大，而且部分图片还有光照的干扰。因此我想这里需要检测一下数字的边缘。第一反应使用 canny，但是 canny 速度很慢，效果也不是很好。思考一段时间后，发现这里其实我只需要过滤掉背景，因为边缘点与背景最本质的区别就是他们的梯度会不同，所以这里我简单地求一个梯度就基本解决问题了。用了 canny 部分中求梯度的代码，在梯度上对图像进行二值化处理，梯度的阈值是自己慢慢调的，做完之后图片就干净很多了，数字边缘也会更加清晰。

2. 投影切割算法。投影切割算法是切割的基础，如果找的切割点不对，那么切割是肯定失败的。一开始是直接在灰度图中找“谷”，但是发现这样很难去找到一个普遍适用的规则去找到。如果规则是：灰度值为 0，那么两个峰之间可能会存在很多个灰度值为 0 的点，找哪个才是最好呢？后来变换了一个思路，我可以先找到所有的峰，对峰之间求一个平均值就是两个峰之间的“谷”了。当然找到峰的界也是困难的，因为一个峰中间也可能会有断裂，我们需要排除这种情况。我的基本做法是，统计当前点和上一个界之间的黑点数。对于下界而言，黑点数必须超过一定数量才认为是中间夹着一个峰；对于上界而言，黑点数必须少于一定数量才认为他们之间没有一个峰，具体的数值也是经过测试自己慢慢调的，但不是对于所有数据都适应。

3. 添加单个数字图像边缘。一开始切割到单个数字后，就直接拿去模型检测了，但是效果很差，后来上网查了一下，发现要添加包围部分。要尽量做到与 MNIST 测试集一样，然后为图片添加了四周的黑色边缘，识别效果就好一点了。

4. 过滤部分数据。在测试过程中，有些数据是不能使用的，可能是含有中文，可能是光线影响过大，也可能是字写歪了，这些其实都不是算法的问题，而是数据的问题，在测试的时候需要首先过滤掉这些不好的数据。在剩下的数据中再做测试。

总结：Final Project 综合运用了这个学期所学的知识，实现了手写数字的识别。主要有以下几个部分：A4 纸矫正、边缘检测、图像切割、膨胀与腐蚀、区域连通。整个项目的难度在于其步骤很多，每一步都需要做的比较好，最后的结果才能比较满意，一旦某几个步骤有误差，每一步累积起来，最后的效果就会很差。整个项目中用到了许多阈值，这些阈值有些是自适应的，有些是要通过多次数据测试自己调的，这其中也花了比较多的时间。我认为这次实验的结果比较满意，切割和识别的结果都比较好。可以优化的地方有：可以优化切割算法，以支持斜线的切割；使用 CNN 神经网络来进行数字识别，这类模型可能有更高的准确率。

总的来说我觉得整个项目的效果不错，在切割方面，对于同学们在课堂上手

写的数据集测试效果更佳。但是由于对深度学习的了解不够，可能训练的模型仍然存在较大问题，在某几个数字的识别上仍存在问题，导致整体的识别率不能提高。我认为我做的项目优点有两个：一是输入参数基本为 0，虽然部分特殊的数据需要调个别参数，但是对于正常的数据（无中文、数字没有连写）来说，基本是普遍使用的，不需要手动输入参数；二是速度很快，因为我没有使用 canny 的 nms 来提取边缘，因此整个算法的速度非常快。通过这个学期的学习，我对计算机视觉这个方面有了大致的了解。通过大量的动手机会，对基本的一些算法比较熟悉，并能够在最后一个项目中自由地运用。我相信日后我一定能在这个方面做出自己的成绩的！

六、参考资料

1. 膨胀与腐蚀：https://blog.csdn.net/poem_qianmo/article/details/23710721
2. 二值图像连通域标记：https://www.cnblogs.com/ronny/p/img_aly_01.html
3. 区域连通标记算法：https://blog.csdn.net/jiangxinyu/article/details/7999102