

# 计算机视觉实验文档

梁俊华 16340129 数据科学与计算机学院

## 第一部分：图像分割与 A4 纸矫正

### 一. 实验要求

输入：普通 A4 打印纸，上面可能有手写笔记或者打印内容，但是拍照时可能角度不正。

输出：已经矫正好的标准普通 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。

### 二. 实验原理

图像分割采用的 OTSU 方法进行分割，并通过图像的梯度进行检测边缘，然后再利用霍夫变换求出 A4 纸的四个顶点的坐标，最后通过仿射变换将 A4 纸的四个顶点映射到完整的矩形图案中，完成 A4 纸的矫正。

OTSU 法也称为阈值法，是日本学者大津于 1979 年提出，是一种自适应的阈值确定的方法。OTSU 法按图像的灰度特性将图像分成背景和目标两部分，背景和目标之间的类间方差越大，说明构成图像的两部分的差别越大。当部分目标错分为背景或部分背景错分为目标都会导致两部分差别变小。因此，使类间方差最大的分割意味着错分概率最小。

OTSU 法进行图像分割的步骤如下所示：

1. 将输入图像转换为灰度图，并统计生成灰度直方图，并生成概率密度函数
2. 选取最佳阈值，记  $t$  为前景与背景的分割阈值，从最小灰度值到最大灰度值遍历  $t$ ，当  $t$  使值  $g = w_0 \times (u_0 - u)^2 + w_1 \times (u_1 - u)^2$  最大时，即为分割的最佳阈值。
3. 遍历图像的每个像素，如果灰度值大于阈值，则置为 0，否则置为 255，从而实现图像的分割。

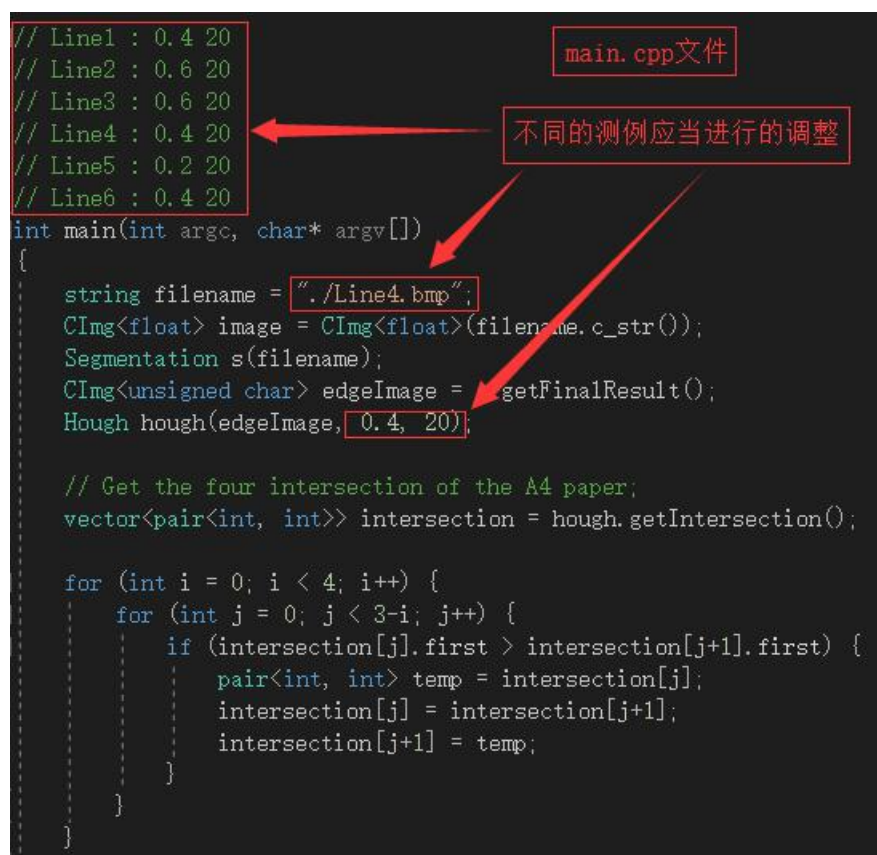
实现图像的分割后，可以利用梯度差进行简单的边缘检测，因为图像分割后，图像之间具有明显的区域界面，可以计算图像当前的梯度，当梯度的值非零时，

可以认为该点为边缘点。边缘检测完成后，通过实验 3 中已经实现的霍夫变换对 A4 纸张的四个顶点的坐标进行提取，最后通过图 Morphing 中实现的仿射变换，将 A4 纸进行矫正。

### 三. 实验测试

实验的测试环境为 Windows10 操作系统,使用 Visual Studio 2017 进行工程项目的编译。直接运行工程项目即可执行程序。

因为霍夫变换的检测存在参数调整的过程，因此在测试六张不同的 A4 纸样例时，应当对参数进行相应的调整，具体的说明如下图所示：



```
// Line1 : 0.4 20
// Line2 : 0.6 20
// Line3 : 0.6 20
// Line4 : 0.4 20
// Line5 : 0.2 20
// Line6 : 0.4 20
int main(int argc, char* argv[])
{
    string filename = ".\\Line4.bmp";
    CImg<float> image = CImg<float>(filename.c_str());
    Segmentation s(filename);
    CImg<unsigned char> edgeImage = s.getFinalResult();
    Hough hough(edgeImage, 0.4, 20);

    // Get the four intersection of the A4 paper;
    vector<pair<int, int>> intersection = hough.getIntersection();

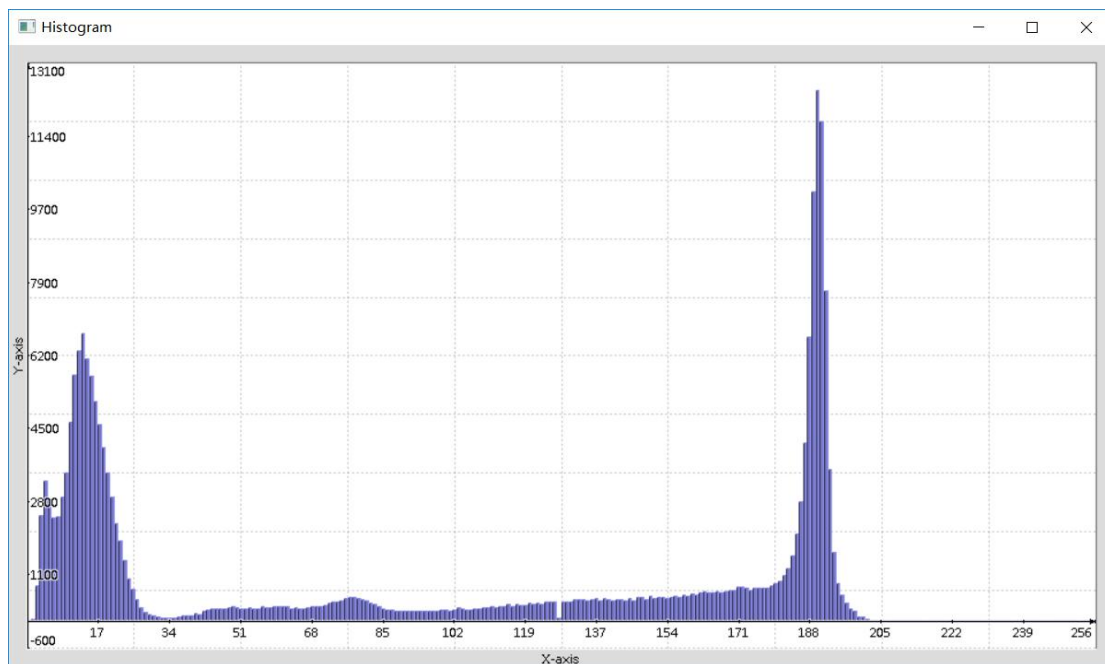
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 3-i; j++) {
            if (intersection[j].first > intersection[j+1].first) {
                pair<int, int> temp = intersection[j];
                intersection[j] = intersection[j+1];
                intersection[j+1] = temp;
            }
        }
    }
}
```

main.cpp文件

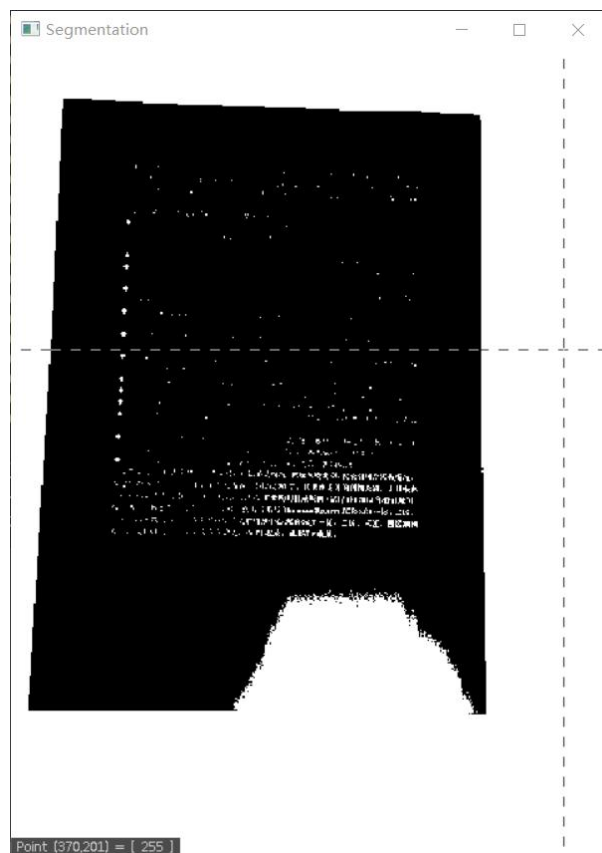
不同的测例应当进行的调整

因下面是对测例 Line1.bmp 的测试结果：

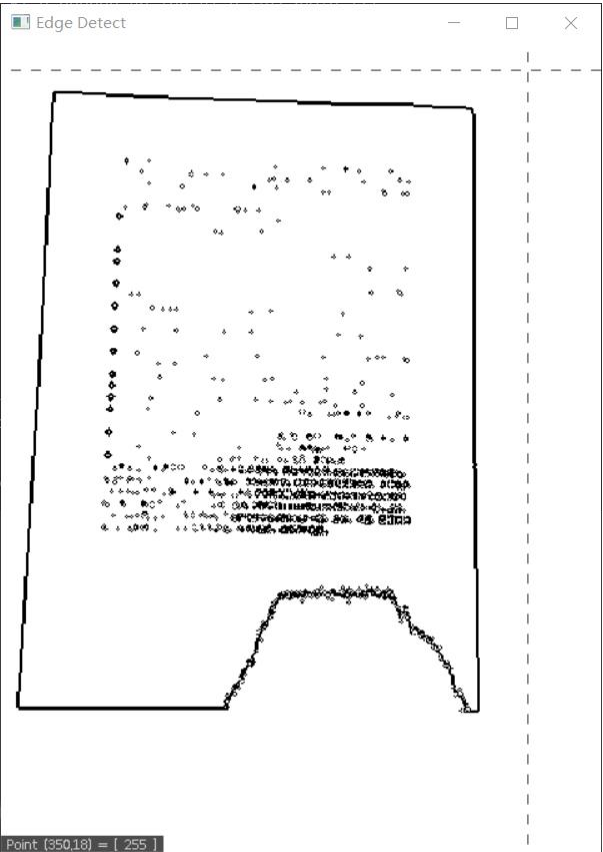
1. 灰度图统计结果如下图所示：



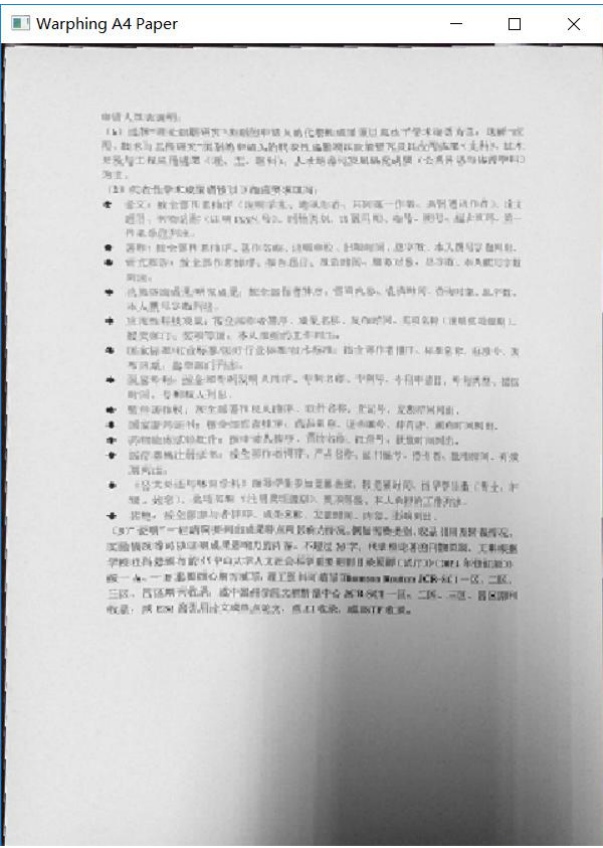
2. Line1.bmp 图像分割的结果如下所示：



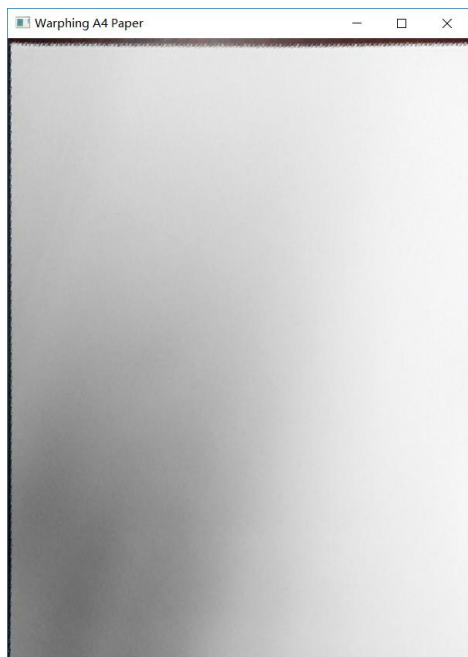
3. 执行边缘检测后的结果如下所示：



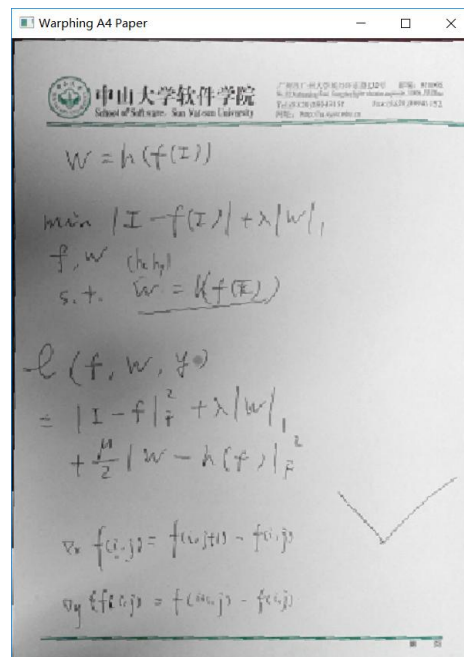
4. A4 纸矫正的最终结果如下所示：



Line2.bmp 到 Line3.bmp 的最终实验结果如下所示:



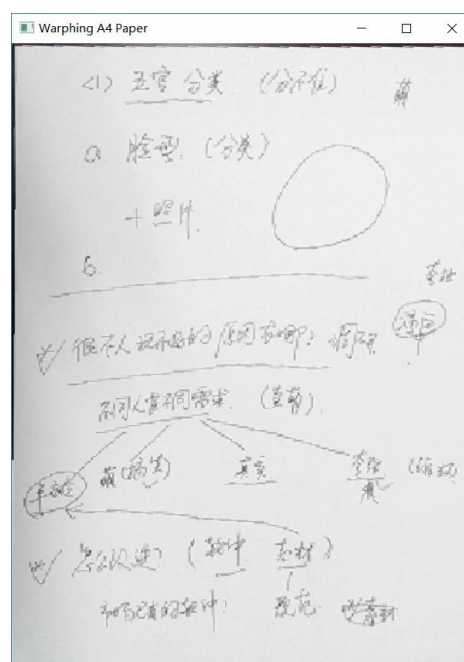
Line2.bmp



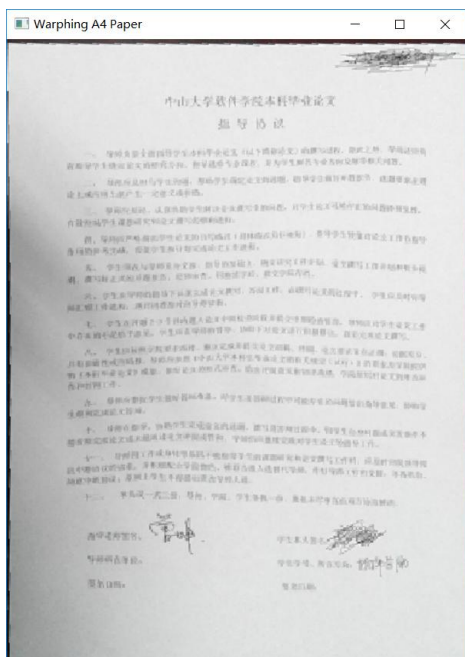
Line3.bmp



Line4.bmp



Line5.bmp

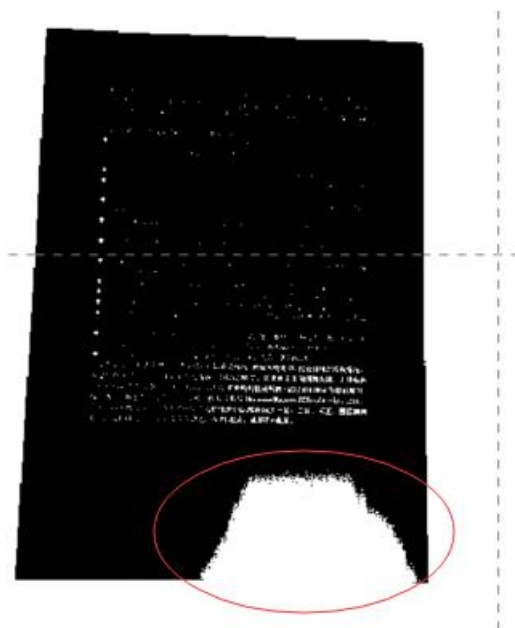


Line6.bmp

## 四. 实验分析

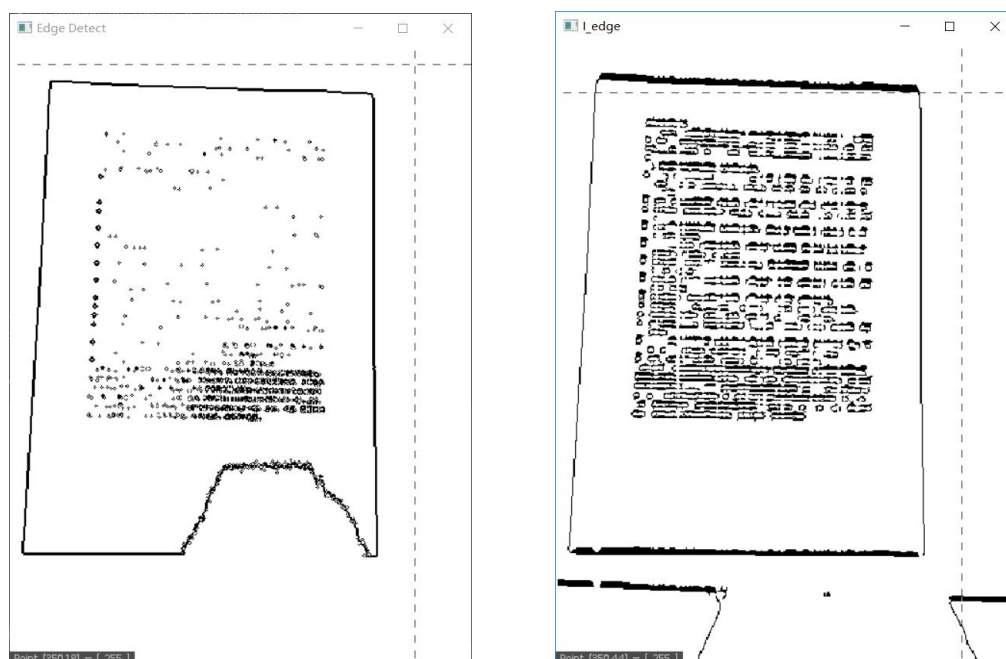
实验结果可以看出，矫正后的 A4 纸字迹并不是很清晰，因为存在尺寸变换，所以在不少测例上都存在瑕疵。改进的方法可以是缩小图片的尺寸，并找到合适的尺寸来进行实验，就可以了。

其次，OTSU 图像分割，对于一些光照因素的影响比较敏感，如下图所示，光照比较强的部分，图片的内容被分割错误了。



可以尝试使用其他的一些图像分割的算法，如迭代法进行尝试，或者对噪音进行消除，目前由于技术原因，未对该问题进行更为深入的尝试，虽然对图像边缘的提取有很大的影响，但是对于直线检测的影响还是比较小。

最后将图像分割方法提取边缘和 Canny 算法提取边缘进行对比，首先观察下面两幅图像的边缘，左边是图像分割算法，右边是 Canny 算法：



从图中可以看出，图像分割进行边缘提取，所得到的边缘的缺少得比较多，而 canny 算法保留的边缘非常多，信息也非常丰富。图像分割提取边缘对于噪音是比较敏感的，因此在噪音比较强烈的右下部分，就缺失了一块信息，导致边缘提取错误。总得来说 canny 算法提取边缘的效果比图像分割要好很多。

## 第二部分：Adaboost 手写数字检测器

### 一. 实验要求

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。

测试说明：

1. Adaboost 训练和测试算法可以用 C++、Opencv 或者 Python 的相关代码；
2. 手写体的检测和识别相关的训练数据为：Dataset: <http://yann.lecun.com/exdb/mnist/>
3. 用户自己在 A4 纸上书写单个数字做识别。

## 二. 实验原理

实验原理为 Adaboost 算法, 是 Freund 和 Schapire 在 1996 年提出的一种自适应性算法, 其核心思想是针对同一个训练集训练不同的分类器 (弱分类器), 然后把这些若分类器集合起来, 构成一个更强的最终分类器 (强分类器)。

Adaboost 算法本身是通过改变数据的分布来实现的, 它根据每次训练集中每个样本的分类是否正确, 以及上次的总体分类的准确率, 来确定每个样本的权值。将修改过权值的新数据集送给下层分类器进行训练, 最后将每次得到的分类器最后融合起来, 作为最后的决策分类器。

本实验中, 手写数字识别使用 mnist 数据集进行训练, mnist 数据集中的手写体数字被处理为 784 维的向量, 作为输入的数据, 总共有 60000 组数据, 作为原始数据输入, 并使用决策分类树作为弱分类器, 进行 Adaboost 算法。具体的算法流程如下所示:

1. 输入样本数据  $(x_1, y_1), (x_2, y_2) \dots (x_m, y_m), x_i \in X, y_i \in \{-1, 1\}$
2. 初始化零时刻样本  $x_i$  的概率分布  $D_i = \frac{1}{m}, i = 1, 2, \dots, m$
3. T 为训练的最大循环次数, 则对  $t = 1, 2, \dots, T$  进行弱分类器的训练, 其中

$$h_t = \underset{h_j}{\operatorname{argmin}} \varepsilon_j, \varepsilon_j = \sum_{i=1}^m D_t(i) [y_i \neq h_j(x_i)]$$

上式表示若划分正确, 则不计入误差, 若所有元素都被正确划分, 则误差为 0, 若划分错误, 则计入误差, 弱分类器选取误差最小的, 然后计算适应值:

$$\alpha_t = \frac{1}{2} \ln \frac{1 - \varepsilon_t}{\varepsilon_t}$$

然后更新权值分布:

$$D_{i+1}(i) = \frac{D_t(i) e^{-\alpha_t y_i h_t(x_i)}}{Z_t}, Z_t = \sum_{i=1}^m D_t(i) e^{-\alpha_t y_i h_t(x_i)}$$

最后得到强分类器:

$$H(x) = \operatorname{sign} \left( \sum_{t=1}^T \alpha_t h_t(x) \right)$$



### 三. 实验测试

实验的测试环境为 Windows10 操作系统，使用 python3 的 sklearn 库进行实验。由于 sklearn 库封装了 Adaboost 的相关方法，因此只需要进行库函数的调用即可完成 Adaboost 算法对手写体的识别，其中关键代码如下所示：

Adaboost 模型的初始化：

```
adaboost = AdaBoostClassifier(DecisionTreeClassifier(max_depth=5, min_samples_split=5, min_samples_leaf=5), \
                               n_estimators=200, \
                               learning_rate=0.05, \
                               algorithm='SAMME.R')
```

模型训练的代码如下所示：

```
# Train the adaboost model.
print("Train Adaboost Model")
adaboost.fit(train_images, train_labels)
```

模型预测的代码如下所示：

```
# Get the predict digit.
predict_digit = list(adaboost.predict(test_images))
```

测试只需要在 Adaboost 所在文件的目录下执行：Python Adaboost.py 即可进行代码编译和执行。

由于 Adaboost 的训练需要比较长的时间，因此已经预先将训练的结果保存在 ./mnist/adaboost.m 文件中，所以可以直接进行相关测试，如果希望能够体验一边训练的过程，可以将该文件删除，然后重新执行上述代码。

训练和测试的数据保存在 ./mnist 文件夹中，如下图所示，且读入这些文件所需要的代码为参考 mnist 数据形式读入的代码，在源代码注释中有所说明：

- t10k-images.idx3-ubyte
- t10k-labels.idx1-ubyte
- train-images.idx3-ubyte
- train-labels.idx1-ubyte

训练测试的结果如下所示，下图是最大迭代次数为 10 次时的结果：

```
Train Adaboost Model
Train Adaboost Model Complete!
The accuracy score is: 0.8623
The digit of this paper is: [7]
```

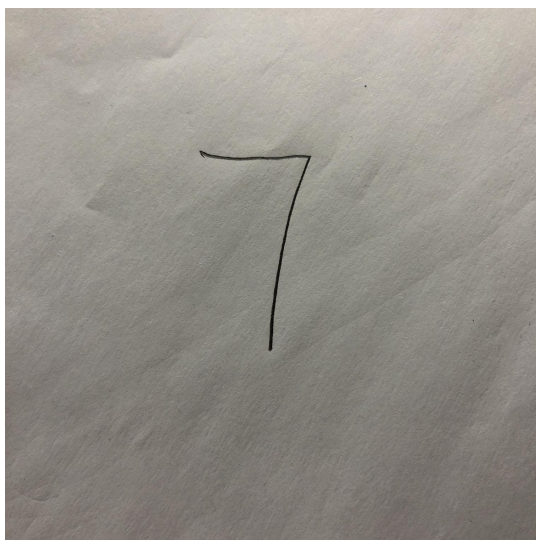
训练 10 次的准确率只有 0.8623

```
D:\大三上\2016级\计算机视觉\HW7
λ python Adaboost.py
Train Adaboost Model
Train Adaboost Model Complete!
The accuracy score is: 0.9357
```

训练 400 次的准确率提高到 0.9357

最后是手写体的测试，我在纸上写了如下的数字，测试时注意修改文件的路径，测试数据均保存在 mnist 文件夹中：

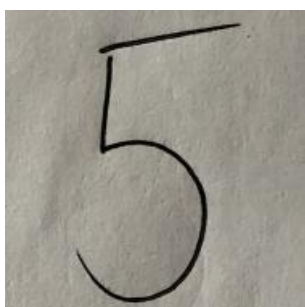
```
# Test the digit from A4 paper.
digit = digitPredictFromPaper(adaboost, "./mnist/7.bmp")
print("The digit of this paper is: {}".format(digit))
```



首先会在代码中会将该数字转换为 mnist 形式的 784 维向量，然后传入检测模型中进行检测。

检测的结果如下所示：

```
D:\大三上\2016级\计算机视觉\HW7
λ python Adaboost.py
The digit of this paper is: [7]
```



```
D:\大三上\2016级\计算机视觉\HW7
λ python Adaboost.py
The digit of this paper is: [5]
```



```
D:\大三上\2016级\计算机视觉\HW7
λ python Adaboost.py
The digit of this paper is: [3]
```

#### 四. 实验分析

从上述实验数据中可以看出，训练的过程需要花费大量的时间，且弱分类器的选择对于 **Adaboost** 算法的结果影响是比较大的。除了决策分类树作为弱分类器外，还可以利用神经网络的模型作为弱分类器。

统计了一下训练 200 次所需要的时间，大概需要 10 多分钟，因此训练的速率确实是很慢，但是准确率能够达到比较高，而且只需要训练一次，把模型保存下来，就能够直接在下一次运用。

用 python 实现 **Adaboost** 确实是比较简单，因为在 **sklearn** 库中都已经实现了相关算法，我觉得如果用 **C++** 实现的话，还是会花费很多功夫的，而且在处理 **A4** 纸输入数据的过程中，还是花费了大量时间把它转换为向量。