Kaggle 竞赛——“使图像易于访问”实验报告

队员一：黄弈骁 学号：PB20000131
队员二：郑滕飞 学号：PB20000296

2023 年 8 月 1 日

（全文共 16995 个字）

目录

一、赛事背景

1.1 赛事背景

世界上数以百万计的学生具有视力障碍，他们几乎没有办法直接阅读 STEM 领域的各种传统纸质教材。虽然目前我们有许多途径使得书面文字易于访问，但是对于图表、照片等材料，使其易于访问将十分困难且需要大量的资源。目前，将这些图表数据转化为可使用的文字材料一般是人工完成的，虽然这种方法可以生产可访问的材料，但它既昂贵又耗时，并且必须经过几轮质量检查。“Benetech - Making Graphs Accessible”使图像易于访问竞赛，就是希望针对这一问题，设计一个很好的机器学习模型，使其能够自动地将出现在教材中的图表（如条形图、折线图、散点图等）转化为易于视力障碍者访问的文字材料。

1.2 赛事链接

https://www.kaggle.com/competitions/benetech-making-graphs-accessible

二、问题定义

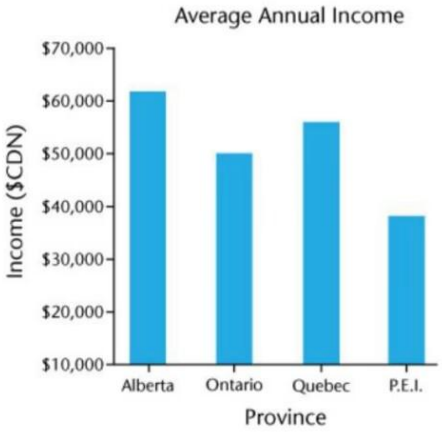
2.1 任务目标

比赛的任务目标是解析五种常见图表中的数据信息，包括横向条形图、纵向条形图（含直方图）、点图、折线图、散点图。首先我们应当使用某些机器学习算法，正确地分类这五种图表。除散点图之外，我们的目标，是输入一张图表图片，输出其坐标轴的所有标签及对应的数据值。而对于散点图而言，我们的目标是输出图中每一个散点对应的横纵坐标轴取值。以下给出一些例子：

①纵向条形图：

x,Alberta;Ontario;Quebec;P.E.I.,vertical_bar

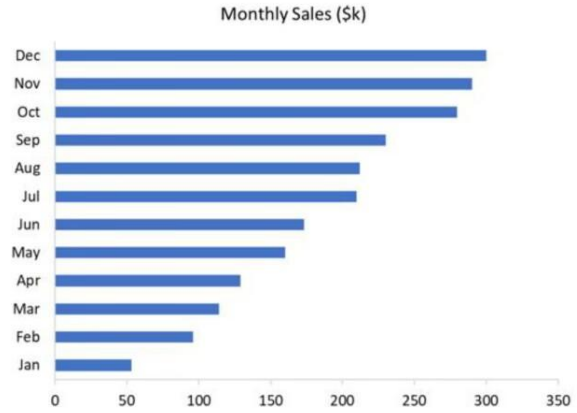
y,62023;50355;56288;38621,vertical_bar



②横向条形图：

x,299;289;279;229;211;209;173;159;128;113;95;53,horizontal_bar

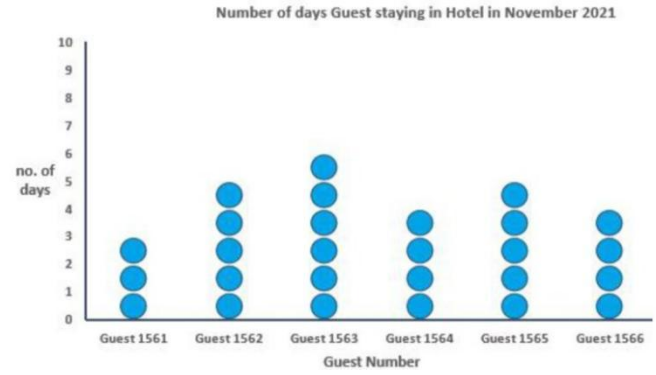
y,Dec;Nov;Oct;Sep;Aug;Jul;Jun;May;Apr;Mar;Feb;Jan,horizontal_bar



③点图：

x,Guest 1561;Guest 1562;Guest 1563;Guest 1564;Guest 1565;Guest 1566,dot

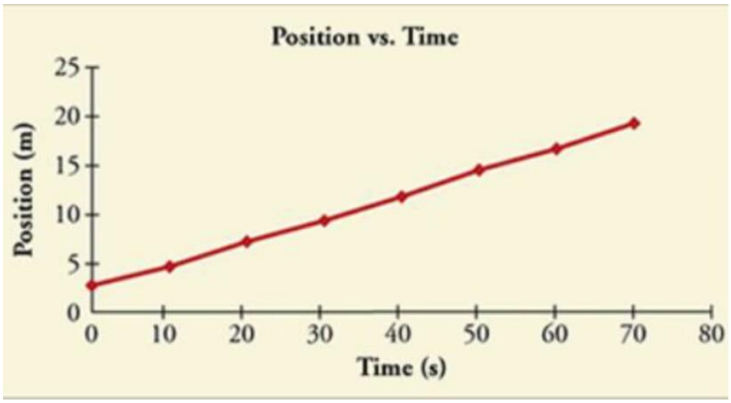
y,3;5;6;4;5;4,dot



④折线图：

x,0;10;20;30;40;50;60;70,line

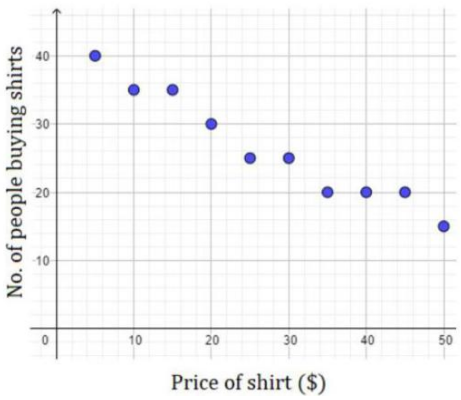
y,2.85532;4.85973;7.02666;9.41515;11.72693;14.44788;16.70328;19.50852,line



⑤散点图：

x,5;10;15;20;25;30;35;40;45;50,scatter

y,40;35;35;30;25;25;20;20;20;15,scatter



2.2 特殊要求

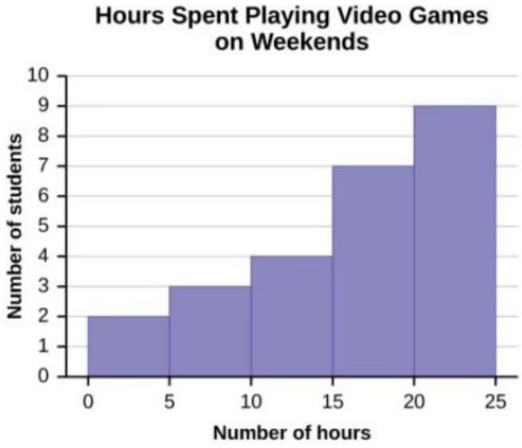
由于现实中图表的种类纷繁多样，因此对于一些特殊情况，我们有许多特殊的输出要求。识别并满足这些特殊要求是个非常困难的挑战，因为存在特殊要求的图表仅在总量中占一小部分，如果识别错误，那么将干扰无特殊要求图表的识别精度，具体要求如下：

①条形图有可能为直方图，这种情况下需要多识别一个横坐标结果：

（这是唯一一种 x 轴值比 y 轴多的情况）

x,0;5;10;15;20;25,vertical_bar

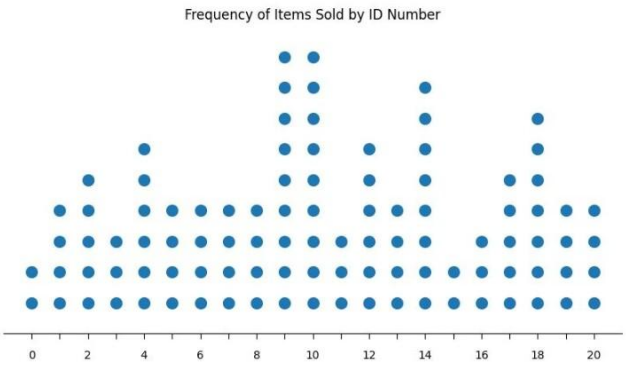
y,2.0;3.0;4.0;7.0;9.0,vertical_bar



②点图有可能没有 y 轴，此时纵坐标应点的数量：

x,0;1;2;3;4;5;6;7;8;9;10;11;12;13;14;15;16;17;18;19;20,dot

y,2;4;5;3;6;4;4;4;4;9;9;3;6;4;8;2;3;5;7;4;4,dot

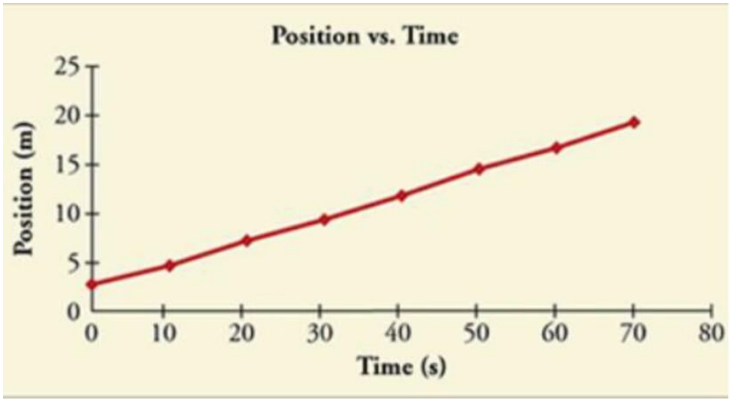


③若折线图 x 轴点在图中没有对应的点，则不输出这个点：

（如下图，我们不输出 x=80 的这个点）

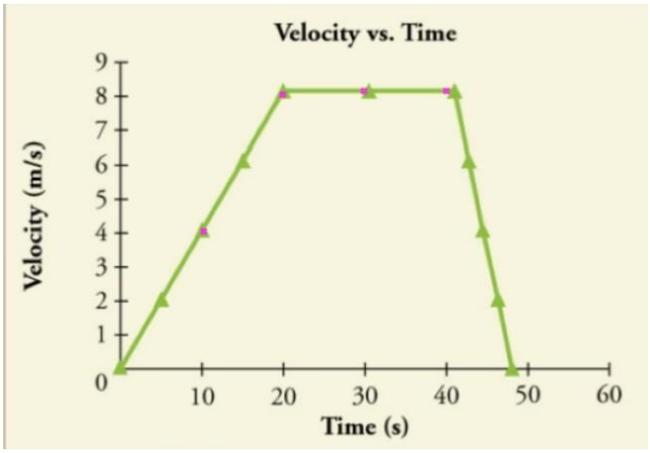
x,0;10;20;30;40;50;60;70,line

y,2.85532;4.85973;7.02666;9.41515;11.72693;14.44788;16.70328;19.50852,line



④如果 x 轴 y 轴在坐标原点共用了一个标签，那么忽略这个标签：

（即下图不包含标签 x=0，y=0）



⑤含百分号的数字应当被标记为数字本身，如 35%应标记为 35。

2.3 提交要求

提交 Kaggle 竞赛往往有专门的代码提交要求，要求如下：

①GPU 运行时间小于等于 9 小时；

②不允许连接互联网：这个要求直接影响到我们在每次提交时都得在代码中主动安装 Kaggle Notebook 并未自动安装的库（具体方法是将这些库下载到本地，再上传至数据集，最后在代码中手动用!pip 安装），这对我们的实验运行是非常耗时的；

③可以使用外部公开且免费的数据集和预训练模型；

2.4 评价标准

本次比赛的评价标准是非常严苛的，有许多情况会导致 Scoring Error，而有许多情况会直接导致某个样例得分为 0，想得到非 0 评分是相当困难的，评价标准如下：

I）整体要求：

①整份文件中的行数必须完全正确，否则 Scoring Error；

②每行的格式必须严格按照要求，否则 Scoring Error；

③文件输出中关于特殊字符，尤其是单引号’要特殊处理，否则 Scoring Error；

II）样例得分要求：

①每个样例的分类必须完全正确，否则直接得 0 分；

（这意味着分类准确率直接影响最终结果）

②每个样例的横纵标签数量必须完全正确，否则直接得 0 分；

（这将是一个很恐怖的事情，例如散点图必须将点的数量判断得完全正确，如果有粘连在一起的点没分辨出来将直接判为 0 分，另外标签数量也必须完全识别正确，否则也会被直接判为 0 分，这个是最困难的事情，后文中我们会详细介绍这一挑战）

III）合规样例评分标准：

①总分计算：对于每一行满足 II 中规则的结果，我们根据其应当属于分类型数据还是数值型数据，对其套用对应的距离计算公式，再通过一个公共的函数进行标准化，得到一个 0-1 的数值作为该行的分数。对所有行的分数进行平均即得到总分。

②标准化函数：

$$\sigma(x) = 2 - \frac{2}{1+e^{-x}}$$

③数值型数据：我们使用标准化的 RMSE：

$$\mathrm{NRMSE} = \sigma\left(\frac{\mathrm{RMSE}(y,\hat{y})}{\mathrm{RMSE}(y,\bar{y})}\right)$$

④分类型（非数值型）数据：我们使用标准化的 Levenshtein 距离：

$$\mathrm{NLev} = \sigma\left(\frac{\sum_i \mathrm{Lev}(y_i,\hat{y}_i)}{\sum_i \mathrm{length}(y_i)}\right)$$

三、训练数据集

3.1 数据集简介

本竞赛提供的官方数据集含有 60578 张图表与它们对应的较完整的数据信息。该数据集提供的数据信息十分完整，但从数据分布上看，这是一个非常差的数据集。下面将对这两个方面分别描述。

3.2 数据集数据分布

该数据集的数据分布非常差，这非常灾难性地影响了我们的工作。首先数据来源包括机器生成的和真实数据，机器生成的有 59460 张，真实数据仅有 1118 张，而最终的评判是单纯基于真实数据上的，因此训练集上的准确度很难代表真实数据上的准确度。

另外，训练数据中的五种分类比例极度失衡，甚至存在某些种类数据完全缺失的情况。这对我们的模型同样有非常灾难性的影响。

完整数量分布如下：

来源\类型	点图	横向条形	纵向条形	折线图	散点图	总和
真实数据	0	73	457	423	165	1118
机器生成	5131	0	18732	24519	11078	59460
总和	5131	73	19189	24942	11243	60578

3.3 数据集数据信息

该数据集的每张图表都提供了非常详细的数据信息，其主要信息如下：
①数据来源②图表类型③坐标轴矩形框④每个文本位置⑤每个文本内容⑥每个文本作用⑦每个坐标轴标签位置⑧每个坐标轴标签类型⑨最终输出结果。

四、困难与挑战

（这个竞赛是一个非常困难的竞赛，原因是有特别多难以处理的困难和挑战需要识别且处理，因此特别以此节列举本次竞赛的难点）

4.1 由赛事设计及 Kaggle 平台带来的困难与挑战

①离线要求：比赛要求提交的 Notebook 是不可以联网的，这将使得我们在每次提交时都得在代码中主动安装 Kaggle Notebook 并未自动安装的库（具体方法是将这些库下载到本地，再上传至数据集，最后在代码中手动用!pip 安装），这对我们的实验运行是非常耗时的（每次运行安装库都要 20-30 分钟）；

②Kaggle 自带 GPU 的时间限制问题：我们必须要在 Kaggle 自带的环境中进行测试，否则无法检测出许多 Bug（可能是许多库的版本问题，导致许多在我们自己电脑或者实验室环境中可以运行的东西，在 Kaggle 平台上提交就会报错）。而我们的程序运行时间为 1h+，且需要使用 GPU，而 Kaggle 平台的 GPU 每个用户每周只能使用 30h，这对我们的测试是远远不够的。

③Kaggle 平台的命令行限制：由于 Kaggle 平台的命令行功能过度有限，如只能操控少量文件，这就使得在手动安装库时遇到了非常多的阻碍（这里就不列举了）；另外，这也使得我们训练大量模型时无法使用 Kaggle 平台调试，因此我们只能使用自己实验室的服务器进行模型预训练。

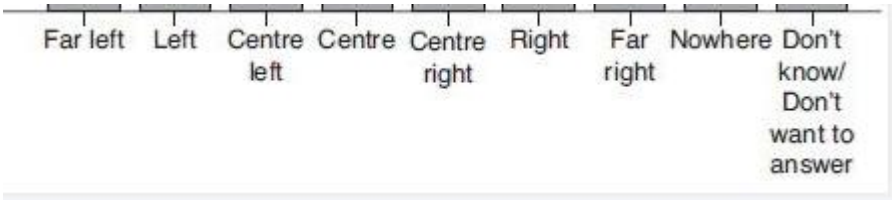
4.2 由数据集带来的困难与挑战

①真实数据与生成数据比例过于悬殊：数据集的数据来源包括机器生成的和真实数据，机器生成的有 59460 张，真实数据仅有 1118 张，而最终的评判是单纯基于 6000 多个真实数据上的，生成数据的效果显著优于真实数据，因此训练集上的准确度很难代表真实数据上的准确度。

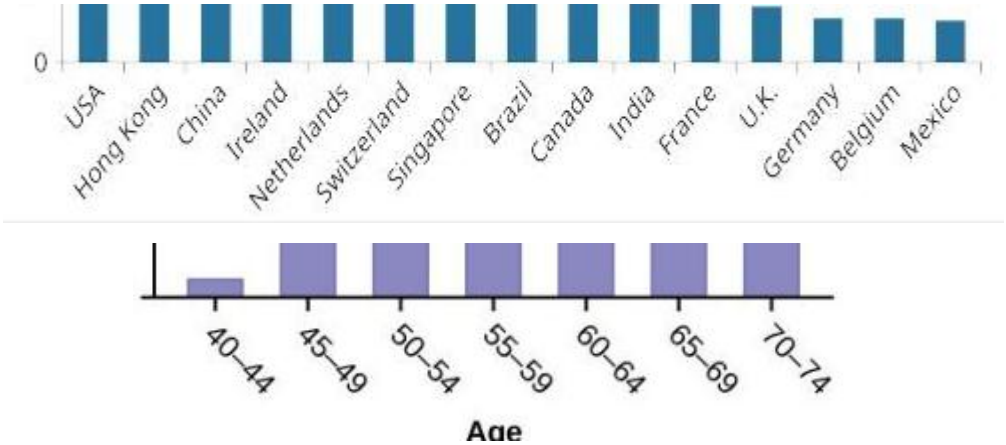
②训练数据中的五种分类比例极度失衡，甚至存在某些种类数据完全缺失的情况：这个比例已经在 3.2 节中给出，这里不再赘述，这对实验的测试影响是灾难性的。

4.3 由图表的标签特性带来的困难与挑战

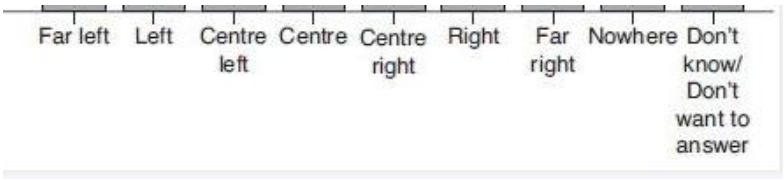
①文字多行问题：这个问题基本出现在正的字体方向上，坐标轴标签无法用一行完整写出，因此分成了好几行。OCR 的识别结果仅仅能识别一个个文字框的位置，无法识别他们是否属于同一个标签，因此需要对识别出的结果进行专门的组合。但每个标签的行数可能不同，每行的长度甚至中心很可能不一致，这个会导致组合起来产生问题，例图如下：

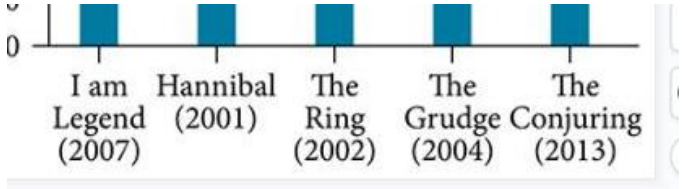


②字体斜度问题：极大一部分图表中的文字不是正的，这部分字体斜度可能是旋转了 0、45、90、135、180、225、270、315 度后的结果。一般的 OCR 模型都只能解决 0 度的情况或者勉强能解决 90、180、270 度的情况。因此，我们需要对图像反复旋转后进行识别，这就需要解决坐标转换问题包括如何旋转图片，并将旋转图片后识别出的坐标还原回原本图片中的坐标，例图如下：

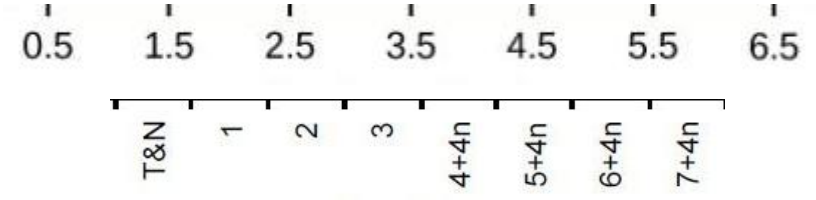


③字体间距问题：分为两大类，同标签内距离太大可能识别为两个标签，不同标签间距离太少会识别为同一标签。由此造成的错误对于这个图片的结果会有灭顶之灾（标签数量不对该样例直接判为 0 分）。如下图，最右侧的 Nowhere 与 Don't 会被识别为一个标签，而 want to 又会被识别为两个不同的标签；另外一张图的标签间距就更加小了：



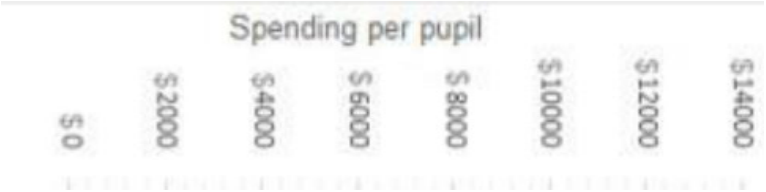


④特殊符号问题：文字中有一些非英文字符，或者有一些-_'".之类的符号，对这些微小的符号识别的效果可能偏差，虽然少量符号的偏差可能对得分影响较小，但如果是小数点、负号等负号，这就对最后的结果插值影响巨大了：



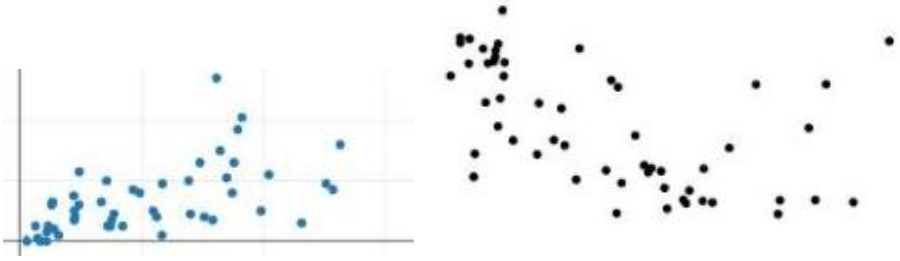
⑤图片模糊问题：有不止一小部分的图表人眼都看不出来标签的内容是什么，这样的样例对 OCR 模型来说也是很难识别的，例如：

⑥数值型数据的数值化问题：许多数值型数据需要进行插值，但数值型数据中有一些非数值元素，需要先去除掉这些元素，化为数值型数据，例如$%等：

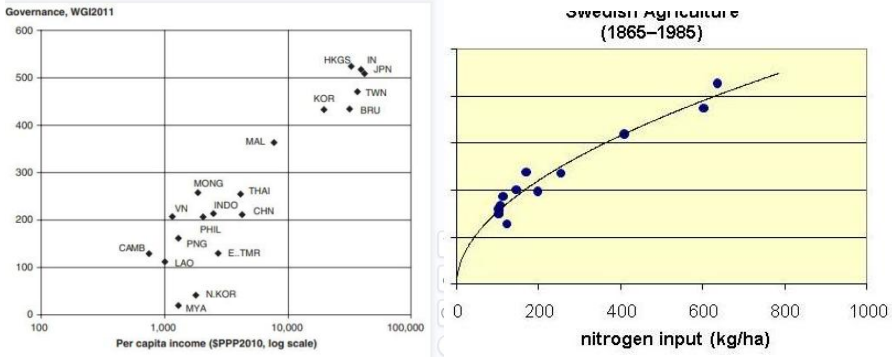


4.4 由图表的图形特性带来的困难与挑战

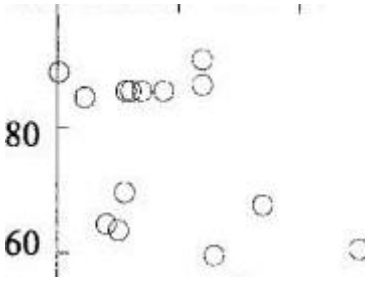
①散点图的点粘连问题：由于需要完全准确识别标签数量，否则判为零分，因此，对于散点图而言，我们需要对点的数量完全判断正确，这是非常困难的。更灾难的是，许多散点图的点质量非常差，甚至出现的多点粘连的情况，这将严重影响散点图判定的准确度：



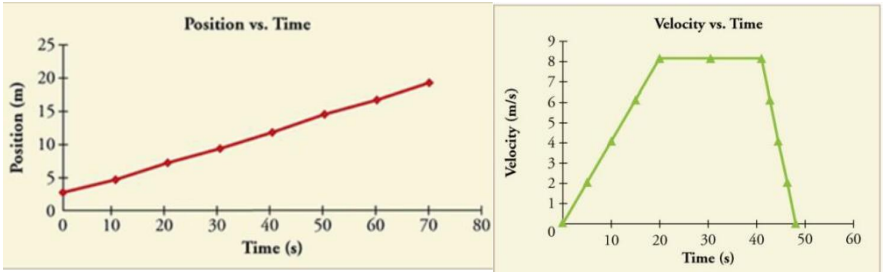
②散点图的干扰信息问题：散点图中经常不一定只有散点，有的时候会有拟合的线（这容易直接被分类为折线图），或者标记的数字，这将很可能被识别为散点；另外，我们还需要精准识别图形区域，否则标题、坐标轴标签的文字也可能被识别为散点，这些都将直接导致该样例零分：



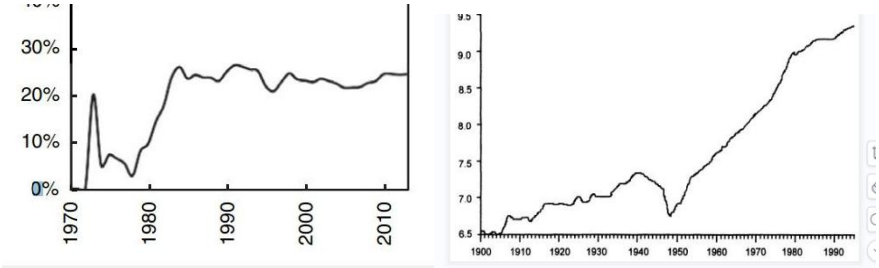
③散点图的点形状问题：散点图中的点不一定都为圆形，甚至可能是空心点，这将对我们判断什么是“散点”的泛化要求非常高：



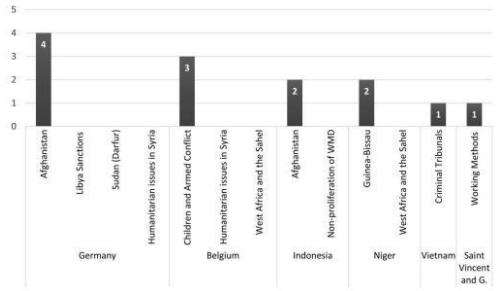
④折线图的无线问题：题目要求，若折线图 x 轴点在图中没有对应的点，则不输出这个点，而折线的宽度是很低的，因此我们需要判定标签对应位置是否有折线，如果判定错误导致标签数量错误，该样例将直接得到零分，这是非常危险的：



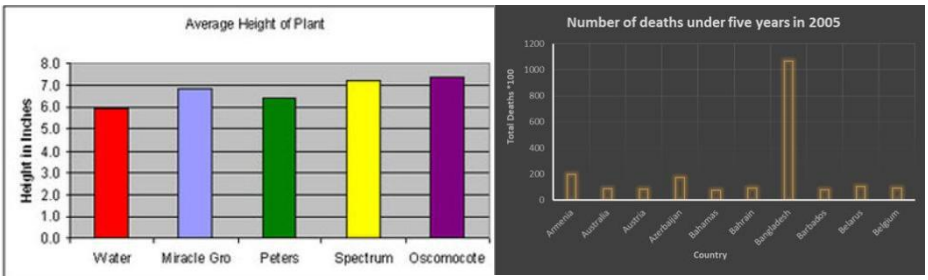
⑤折线图的线与坐标轴重合问题：许多折线图会有线与坐标轴重合，这样的图会导致我们容易判定为④中情况，这将使得我们标签数量不足，也会直接获得零分：



⑥条形图的无柱问题：条形图我们需要识别每个柱子的值，但许多条形图会出现空柱子，这对我们识别产生一定的阻碍，容易识别错误：



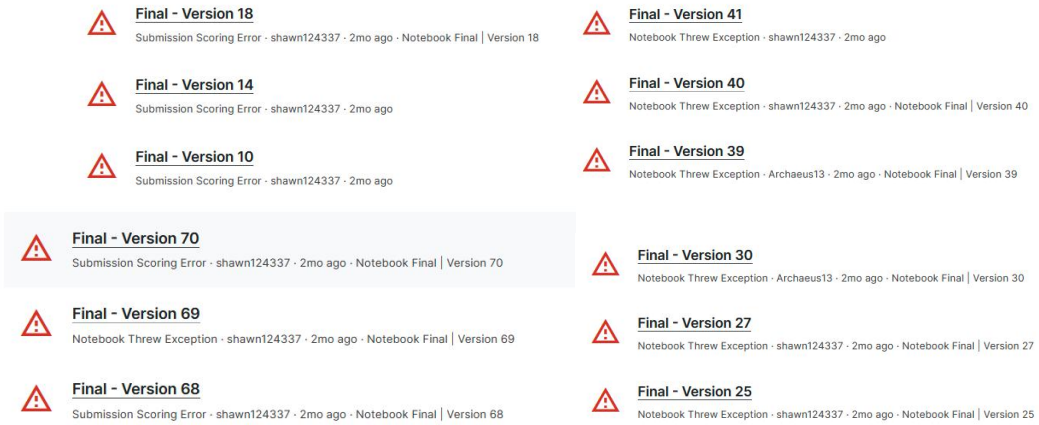
⑦图像的色调问题：许多图像不是白底黑柱，甚至是各种各样奇怪的颜色，或者空心柱，这对我们识别哪些是我们需要识别物体的色调有很大的影响：



4.5 由评价标准带来的困难与挑战

①提交文件的行数问题：由于如果输出文件的行数不正确，会直接得到 Scoring Error 并无法计分，因此程序的鲁棒性很重要，如果程序的鲁棒性不足，那么在经历测试集的困难环境下容易出现错误，很可能会遭遇训练集中未出现的异常，导致发生程序错误或者行数错误，

这浪费了我们非常非常多的时间，以下给出一小部分例子：



②错分类的零分问题：一旦分类错误，那么我们该样例就直接得到零分，因此，分类准确率会直接作为乘积因子影响最终准确率，因此分类准确率是很关键的。当然，我们的分类器最终达到了 98%的准确率，这个挑战就被成功克服了。

③错标签数量的零分问题：不管是 OCR 还是具体图形识别，都有大量情况会导致标签数量识别错误，具体见 4.3 中的 1-5 和 4.4 中的 1-7，这些都是非常致命的，因此如何尽可能识别对标签数量，是非常非常关键的。

正是在这许许多多的困难与挑战下，其错误率会不断累计，这就使得我们的工作变得极其困难而繁琐。

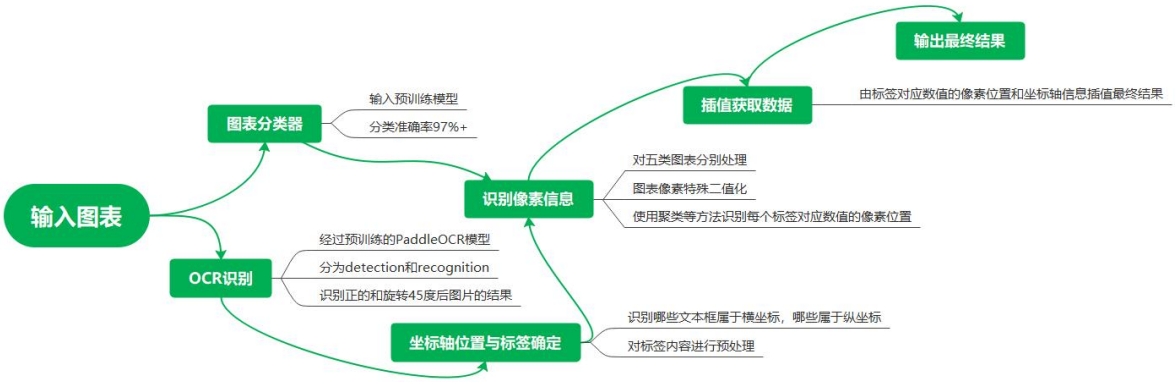
五、问题分析

这个竞赛的四个核心步骤是，分别是分类任务、OCR 任务、坐标轴定位任务和图形识别任务。首先我们需要通过训练准确度高的分类器，这样我们就可以分五类完成对应的图形识别任务；其次，我们需要使用或训练很好的 OCR 模型，以准确识别出图表中的每一个标签及它们对应的矩形框位置，尤其是条形图、点图和折线图横坐标的标签，如果没能识别准确，那么将直接导致标签数量错误，该样例直接记为零分，另外，纵坐标的标签也起到了插值定位的作用，我们应当尽可能识别准确，这能提高图形识别任务的准确度；接着，我们需要通过 OCR 的识别结果，判断哪些标签属于横坐标轴标签，哪些属于纵坐标轴标签，这也可能会导致标签数量错误；最后，我们要根据横纵坐标轴的结果及分类结果，分五类对图表进行识别处理（我们的主体方法是先对图表进行二值化，分辨出哪些是需识别的图形，再对二值化后的图形进行定位），最终得到每个图表的识别结果。

接下来我们会给出我们模型简要的流程图，并在后面几节对其中的每一个步骤进行详细地介绍：

六、模型流程图

根据第五节的问题分析，我们可以构建模型的大致流程图，我们将给出一份简要的流程图。简要流程图的内容是模型的最主体架构，其中还省略了非常多的实现细节和训练过程，以下给出该流程图：



七、训练集内容读取与图表输入

7.1 训练集数据结构 RawData

我们设计了一个专门用来解析训练集信息的类 RawData，以方便后续的测试与训练。这个类的功能是比较强大的，它可以自动对数据集进行读取前的切片，以此降低训练集的大小，降低各种训练和测试的时间（60000 多张照片的读取与测试要几十分钟）；另外，可以通过在这个类中追加需要的数据读取函数，完成 json 文件中指定内容的读取。以下给出这个类的部分基本函数：

```
get_all_train_filenames();
get_train_data(begin=0,end=None);
get_train_imagenames();
get_train_labels();
get_train_source();
```

我们反复使用了数据集进行包括分类器训练，OCR 的 detection 和 recognition 训练，从图表中识别数据的各种性能测试，还有整体代码的鲁棒性测试等等的各种各样的工作。这个解析数据集的数据结构还是对我们的工作起到了很大的帮助的。

7.2 最终模型的图表输入

对上述数据结构 RawData 进行简要修改，就可以得到一个输入测试集图表数据的类，完成最终模型的图标输入部分。

八、图表分类器与其预训练

8.1 图表分类器简介

我们使用带有残差链接的卷积神经网络对五种图表进行分类，并通过标签转换函数label_transform(label)将分类结果转化为对应的类别标签。图表分类器是经过两个半小时的预训练过后得到的最优的预训练模型，这极大程度上降低了我们最终模型的运行效率。最终，我们的图表分类器效果惊人，在验证集上达到了约 0.984 的准确率。

8.2 图表分类器的模型预训练过程

模型的预训练代码其实相当得长，我们这里只对其进行简要的描述：

①图表预处理：我们使用带有残差链接的卷积神经网络对五种图表进行分类，首先我们对图表进行预处理，我们采用了中心切割方法，切割图片中的中心部分，这个部分将不会带有坐标轴信息，这样就能更加暴露图表信息。事实证明这个方法的效果是惊人的，经过切割，我们最终分类器的效果有了显著的提升。

②数据结构构建：接着我们继承 torch.utils.data.Dataset 类并构建 DataLoader 数据加载器类，将训练数据集的 Dataset 子类按 4：1 的比例划分为训练集和验证集，完成预训练过程的数据结构建立。

③单层卷积神经网络的模型构建：对于每一层卷积神经网络 nor_cov，都是这样的结构，nn.Conv2d()实现单层卷积，nn.BatchNorm2d()实现标准化，nn.ReLU()作为激活函数，nn.Dropout(p=0.5)实现 drop out，由此构建了单层卷积神经网络。

④残差网络的模型构建：残差网络是由一系列残差块组成的。残差块分成两部分：直接映射部分和残差部分。直接映射部分就是返回数据本身，而残差部分，一般由两个或者三个单层卷积操作构成，其作用是减少网络中信息的丢失和损耗。残差网络 residual_block()同样由 torch.nn.Module 类完成，每层残差网络重复以下结构两次，{nn.Conv2d()实现单层卷积，nn.BatchNorm2d()实现标准化，nn.ReLU()作为激活函数}，最后用 nn.Dropout(p=0.5)实现 drop out，输出直接映射 x 和经过处理后的残差 x_之和，由此构建了单层残差网络。

⑤完整的分类器模型构建：完整的分类器模型由完整的卷积神经网络 CNN_net(nn.Module)完成，同样使用了 torch.nn.Module 类，我们构建了四层的网络结构，其中每一层使用 nor_cov()实现单层卷积，调用 residual_block()两次实现残差连接，nn.MaxPool2d()实现池化。最终我们用线性模型 nn.Linear()输出结果，由此构建了完整的分类器模型。

⑥模型训练：我们使用 train()函数对其进行模型训练，步骤如下：

I）设置设备（GPU/CPU）；

II）构建网络 net 为 CNN_net()；

III）构建优化器 optim.Adam(self.net.parameters(), lr=lr)；

IV）构建学习率衰减函数 ReduceLROnPlateau；

V）进行如下循环（次数由参数 epochs 决定）：

VI）从 dataloader 按 batch 提取数据，并用 optimizer 完成参数更新；

VII）计算训练集的交叉熵 loss；

VIII）计算验证集的交叉熵 loss 和准确率；

IX）完成学习率衰减；

X）根据早停策略判断是否停止训练；

XI）输出验证集最优结果对应的预训练模型参数 network_paramsX.pth。

8.3 最终的分类器模型训练超参数

①batch_size=64②num_workers=1③lr=0.001④epoch=40⑤device=’cuda’⑥wait=2,⑦depth=4⑧residual connection depth=2⑨normalization=True⑩drop out=True;⑪drop out p=0.5⑫learningrate decay=True⑬learningrate decay patience=1。

8.4 最终的分类器模型效果

最终，我们的模型经过 9185.9s，共 19epoch 的训练，最终达到了 0.9840443121693122 的验证集准确率，这个效果是相当好的，为后续工作给出了很好的准确率优势。

九、OCR 识别模型

9.1 OCR 模型简介

OCR 即光学字符识别，是指电子设备检查纸上打印的字符，通过检测暗、亮的模式确定其形状，然后用字符识别方法将形状翻译成计算机文字的过程。OCR 正是我们这次竞赛中最核心的部分，我们可以通过 OCR 模型检测图表中哪些位置有文字（detection 部分），并通过识别这些文字的内容（recognition 部分）以得到众多文本框，其中每个文本框对应一个文字位置和一个字符串。图表中最重要的信息正是这些字符串，通过判断哪些字符串属于横坐标轴，哪些字符串属于纵坐标轴，我们就可以得出应当输出标签的数量、位置及内容，并给予后续的识别插值提供相应的点值。当然，一旦 OCR 模型识别出错，如将粘连在一起的标签识别为同一个标签，就会直接导致标签数量错误以致该样例为零分，因此，OCR 模型的强度是最影响最终结果的，是本次竞赛最核心的部分。

9.2 OCR 模型的选择

自己制作 OCR 模型显然不如使用现成的可用工具包。经过一番调研，我们最终使用了百度的 PaddlePaddle 飞桨平台的 PaddleOCR 工具库作为最终的 OCR 模型。这个模型经实验室学长介绍，是一个效果相对比较好的 OCR 模型，此外，它还具有以下几个优势：

①能够识别除英文字符、数字字符外的许多标点符号；

②具有方向器，能够识别旋转过 90 度、180 度、270 度或略微旋转一定小角度的字符；

③模型体量较小，通用性强，速度较快；

④模型输出文本框四个点位置、内容以及识别的置信度，置信度可以起到一定帮助；

⑤自带预训练模型的同时，还提供了进一步预训练的功能，我们可以自己提供数据训练 OCR 模型的 detection 和 recognition，让模型更适用于我们的竞赛要求。

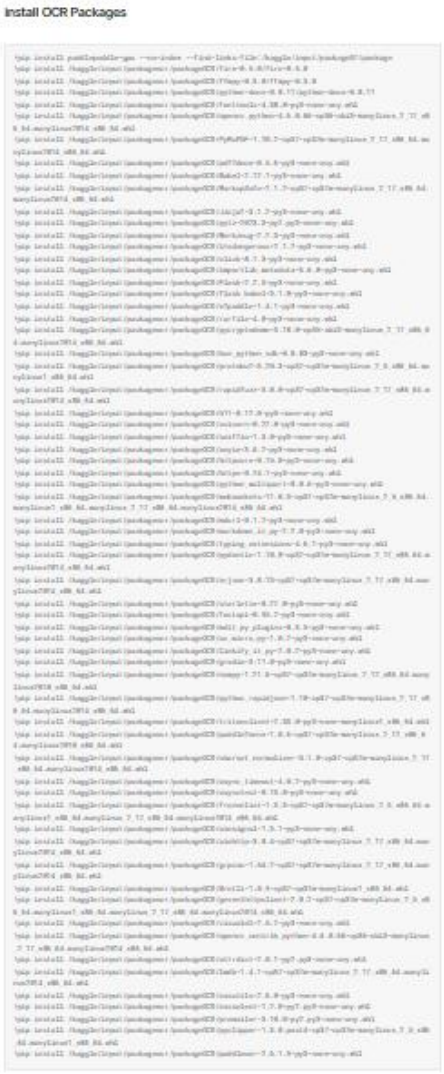
当然，从实践下来看，这个模型还是具有不少的缺点的，不过最终的标签识别数量的准确率按照评价标准我们自己测试下来为 0.62404，从最终结果来看，这个得分是相当高的了。

模型链接：https://gitee.com/paddlepaddle/PaddleOCR

9.3 OCR 模型的安装

由于最终提交的代码要求离线运行，且 Kaggle 平台的命令行功能过度有限，如只能操控少量文件，这就使得在手动安装库时遇到了非常多的阻碍（太多 bug 了），这耗费了我们非常多的时间。我们需要先在 Kaggle 平台上找到一个适用于 PaddleOCR 的版本（最终找到了一个 2020 年的 notebook），还要在这个版本中用!pip download 命令下载安装包到本地。接着我们需要将安装包上传到 kaggle 中，再手动对一个个包进行安装（自动安装总是会出 bug）。这个流程在每次运行代码时都要运行 2200s 左右，是非常非常耗时的。

以下是安装的代码，由此可以看到完成这个序列是多么困难：



9.4 OCR 模型的使用

经过痛苦的安装后，OCR 的使用还是比较容易的，我们只需要输入参数及预训练模型的位置，就可以得到一个 OCR 模型，将图片输入 OCR 模型中，即可返回 OCR 的结果。我们只要对 OCR 的结果进行数据结构的转换，就可以得到想要的输出结果了，构建 OCR 的代码如下：

```
import paddle
from paddleocr import PaddleOCR
ocr = PaddleOCR(use_angle_cls=True, lang="ch",drop_score=0,cls_model_dir='/kaggle/input/pretrain-old/.paddleocr/whl/cls/ch_ppocr_mobile_v2.0_cls_infer',det_model_dir='/kaggle/input/detection6/ch_ppocr_server_v2.0_det_infer',rec_model_dir='/kaggle/input/recognition4/ch_ppocr_server_v2.0_rec_infer',ocr_version='PP-OCRv2')
```

而对 OCR 的结果进行数据结构的转换的代码十分繁琐，这里就不展示了。

值得一提的是，置信度舍弃参数 drop_score 预设为 0.2，即将置信度小于 0.2 的标签自动舍去了，这个设置使得我们的模型标签识别数量显著降低。因此我们需要关闭这个舍弃功能，将 drop_score 设置为 0（我们测试过这个指标为众多值的情况，都不如直接设置为 0 来得好）。这个情况是比较晚发现的，这对我们是一个很大的提升。

9.5 OCR 模型的多方向识别

PaddleOCR 模型只能识别旋转过 90 度、180 度、270 度或略微旋转一定小角度的字符，而图表中有很大一部分字符是斜方向的，如旋转 45 度、315 度的，这些字符经过测试是较难直接识别出来的。因此我们需要对图表进行两次识别，第一次识别正的，第二次先对图表进行 45 度旋转，识别后通过坐标变换变回原本图表中的坐标，此时得到的矩形框是斜向的。图片旋转，坐标变换有一定的数学计算难度，代码如下：

```
def r45(img):
    rows, cols = img.shape[:2]
    M = cv2.getRotationMatrix2D((cols / 2, rows / 2), -45, 1)
    l = int((rows + cols) * 0.707106)
    M[0, 2] += (l - cols) * 0.5
    M[1, 2] += (l - rows) * 0.5
    return cv2.warpAffine(img, M, (l, l)), M[:, 2]

def recover(x1, y1, m):
    x=int(x1)
    y=int(y1)
    x -= int(m[0])
    y -= int(m[1])
    x, y = 0.707106 * (x+y), 0.707106 * (y-x)
    if x<0:x=0
    if y<0:y=0
    return int(x), int(y)
```

值得注意的是，有一些文字识别在图片的边缘，旋转回来后会产生坐标溢出，这个时候我们应当对上下界进行处理。（这个 bug 困扰了我们一段时间）

十、OCR 模型的预训练

10.1 OCR 模型的预训练

PaddleOCR 模型不仅为我们提供了现成的 OCR 模型，还为我们提供了进一步训练 OCR 模型的手段。PaddleOCR 提供了一个专门的 train.py 文件，可以直接使用如下代码 python3.9 tools/train.py -c configs/rec/PP-OCRv3/en_PP-OCRv3_rec.yml -o 进行模型训练，输出训练日志，并输出各断点的预训练模型。通过设置 yml 文件，可以设置训练过程中的各种参数，大概有七八十种参数可以调整。我们可以通过自己实验室的服务器，使用 GPU 直接进行离线训练，并在适当时候停止。最终使用如下代码 python3.9 tools/export_model.py -c /home/3067hyx/PaddleOCR/configs/rec/PP-OCRv3/en_PP-OCRv3_rec.yml -o 生成可使用的 OCR 模型，并上传 Kaggle 平台，即可离线使用。

但是，这个预训练工具的 bug 非常非常多，涉及到大量的 Linux 系统操作，我们大概花费了完整的 15 个小时才成功在实验室的服务器上完成了对它的搭建，这个过程是相当折磨的。另外，这个模型的训练会显著占用 GPU，且停止后不会自动释放内存，因此我们需要经常手动清理内存，十分折磨。

10.2 OCR 模型的预训练数据集

PaddleOCR 模型分为 detection 模型和 recognition 模型，它们是分开训练的，我们需要为他们分别制作对应的训练数据集。对于 detection 模型，我们的核心是为训练提供每个图表所有的文本框位置；对于 recognition 模型，我们的核心是提供每一个文本框的小文字图片，并提供他们的内容。制作这些预训练数据集都有专门的代码，由于逻辑较为简单，这里就不给出具体的代码了。

10.3 OCR 模型的预训练效果

经过大约 2h 的 recognition 训练和大约 48h 的 detection 训练，我们最终得到了多个版本的 recognition 和 detection。Kaggle 比赛允许提交两个文件作为最终比赛代码，最终我们提交了未训练过的 OCR 模型和已训练过的，已训练过后的 OCR 模型效果是好于未训练过的，虽然这个提升并不算太大，但也足以证明，这个预训练是颇具效果的。

十一、坐标轴位置与标签确定

11.1 坐标轴位置与标签确定方法

根据 OCR 的结果，我们会得到许多文本框位置及其内容，我们需要确定哪些标签属于横坐标轴，哪些标签属于纵坐标轴。我们采取的方法是对每个文本框的最顶部坐标进行聚类，以此判定哪个小区间的文本框数目最多，这样的文本框就是横坐标轴；而对每个文本框的最右部坐标进行聚类，以此判定哪个小区间的文本框数目最多，这样的文本框就是纵坐标轴。我们通过对未旋转过的 OCR 结果和已旋转过的 OCR 结果分别处理，取较多标签数量的结果作为最终坐标轴，识别横坐标轴代码如下，还是比较长的，纵坐标轴同理：

```
def row_detect(d=5,d45=10,ord=1):
    for i in range(0,len(TestData.__test_data)):
        list_ca=[]
        list_ca_st=[]
        for item in TestData.__test_data[i]['ocr'][0]:
            row_item=np.array([item[0][0][1],item[0][1][1],item[0][2][1],item[0][3][1]])
            min1=np.min(row_item)
            if min1>TestData.__test_data[i]['image'].shape[0]//2:
                min2 =np.sort(row_item)[1]
                row_item=np.array([min1,min2])
                state=0
                for ca in range(0,len(list_ca_st)):
                    if np.linalg.norm(row_item-list_ca_st[ca], ord=ord)<=d:
                        list_ca[ca].append(item)
                        state=1
                if state==0:
                    list_ca.append([item])
                    list_ca_st.append(row_item)
        axis_num=0;
        axis_t=[];
        for axis in list_ca:
            if len(axis)>axis_num:
                axis_t=axis
                axis_num=len(axis)
        list_ca=[]
        list_ca_st=[]
```

```
for item in TestData.__test_data[i]['ocr45'][0]:
    row_item=np.array([item[0][0][1],item[0][1][1],item[0][2][1],item[0][3][1]])
    min1=np.min(row_item)
    if min1>TestData.__test_data[i]['image'].shape[0]//2:
        min2 =np.sort(row_item)[1]
        row_item=np.array([min1,min2])
        state=0
        for ca in range(0,len(list_ca_st)):
            if np.linalg.norm(row_item-list_ca_st[ca], ord=ord)<=d45:
                list_ca[ca].append(item)
                state=1
        if state==0:
            list_ca.append([item])
            list_ca_st.append(row_item)
axis_num2=0;
axis_t2=[];
for axis in list_ca:
    if len(axis)>axis_num2:
        axis_t2=axis
        axis_num2=len(axis)

if axis_num2>axis_num:
    TestData.__test_data[i]['row_axis_num']=axis_num2
    TestData.__test_data[i]['row_axis']=axis_t2
else:
    TestData.__test_data[i]['row_axis_num']=axis_num
    TestData.__test_data[i]['row_axis']=axis_t
```

11.2 坐标轴位置与标签确定效果

经过这个方法判定的坐标轴，效果还是比较好的。经过我们的简要测算，在这一个步骤的准确率大约为 94.2%。出错的样例普遍具有这样的一些特征，包括①图表的标题被识别为了多个文本框，超过了横坐标轴的数量；②将图中的每个点识别为 0，将这些点识别为坐标轴；③纵坐标轴的最低标签与横坐标轴的标签平行，多识别进了一个文本框。这些问题其实都是比较容易解决的，我们最终只处理了①②两个问题，而问题③由于样例太少，在有限的时间内我们最终并未完成处理，但对结果的影响不算太大。

十二、从图表获取数据

12.1 从图表获取数据介绍

本部分中，我们主要介绍从图表中提取数据的过程。具体来说，我们对每张图片接收的输入有三部分：图表的类型、OCR 得到的图片中所有文字与它们的位置（由四个顶点标记的矩形）、以及推测出的属于两坐标轴的文字与它们的位置。而我们要求的输出，则是每张图片对应的数据，均以实数表示（文字类型的坐标轴已在上一部分被确定）。

12.2 总体处理方法

值得一提的是，从不同图表中获取数据有诸多深度学习方法，例如 Deplot、Matcha 模型等模型可以用于条形图、折线图的数据提取，而 YoloX 模型可以用于散点图的数据提取等。在前几名发布的题解中，也均运用了这些模型。但是，我们最终并没有采用深度学习方法，这是出于三方面考虑：

首先，由于我们并没有在这一领域运用深度学习模型的经验，前期的调试可能会花费大量时间，而在成功运行后，进行一次调整需要的训练周期也是很长的，在比赛流程较为紧张（由于选定比赛较早，比赛期限为 6.19）的情况下，训练不熟悉的模型并不是一个好的选择。

其次，我们对测试数据集是完全未知的。提供的训练图片中真实而非生成的图表极少，主办方也明确表示了最终评分所用的数据集的分布与训练图片中的真实图表分布不同。如果只根据提供的图片中的真实图表训练，很容易陷入过拟合，导致结果并不好。相比之下，传统方法反而具有更高的稳定性。最终成绩也证明了这个结果：在采用与训练图片中真实图表相同分布的测试集进行测试时，我们的排名仅为 536，但随着最终评分的测试集更改，大家的成绩都有大幅下降，我们由于下降较少，排名上升到了 142 位。

最后，传统方法比起深度学习方法更灵活，也更容易最大限度利用上一部分的结果。由于过程不存在黑箱，传统方法易于分析每一张图片的判断失误原因，进而做出对应的修改。

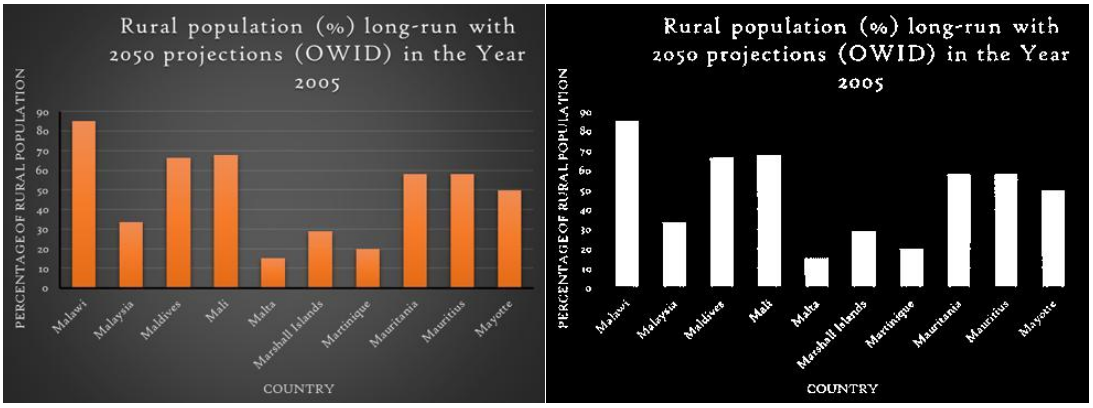
总体来说，对每一类图片提取数据的过程都是要经历像素到点与点到值，对不同图片来说，核心的操作差别是在像素到点的部分，于是，我们先对共同的处理进行介绍。

12.3 图片预处理

首先，为了进行数据提取，需要将任何图片都转化成便于提取数据的形式。由于图表中只需要区分“有内容“与”无内容“，色彩、深浅信息均不重要，对图片进行二值化是一个合理的方式。经测试，我们最终的二值化算法如下：

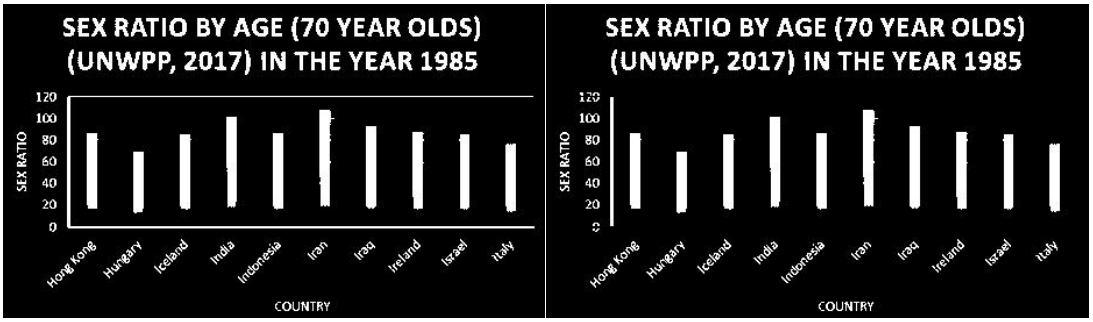
```python
def to_logical(img):
    N = 48
    img = np.mean(img, 2)
    rows, cols = img.shape[:2]
    rc = rows // 2; cc = cols // 2
    try:
        img_crop = img[(rc-N):(rc+N), (cc-N*2):(cc+N*2)]
    except:
        img_crop = img
    if np.max(img_crop) - np.min(img_crop) < 40:
        img_crop = img
    b = (np.max(img_crop) + np.min(img_crop) + np.mean(img_crop)) / 3
    A = img > b
    if np.sum(A) < rows * cols / 2:
        return A
    else:
        return ~A
```

效果示例如下（右侧为二值化后，以下左右对比右侧均为处理后）：



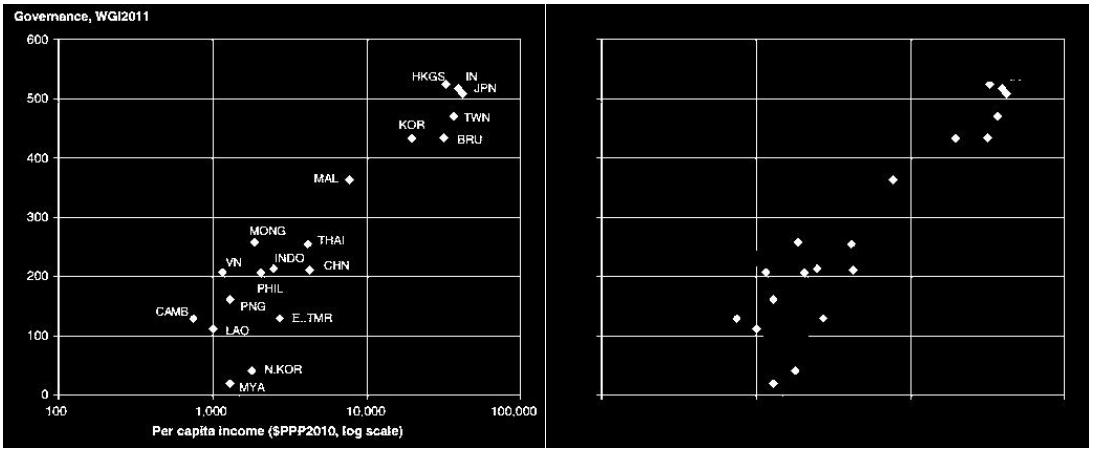
具体来说，我们先通过截取尽量保留图片的中心部分，若图片过小或截取部分中无明显颜色差别则以整个图片作为中心部分。接着，我们在中心部分中根据最大、最小与均值确定某个灰度阈值进行二值化。为了方便后续处理，我们假定整个图片中数据占比相对非数据较小，因此将较小的部分置为 1，其他为 0。

尽管二值化可以排除无效的大部分区域，却无法排除网格的影响。出于颜色的区分，网格的颜色一般较为鲜明，与数据同样二值化为有效。这时，需要通过算法排除网格。由于上一步的坐标轴判断能划分出数据区域，网格可以直接以“有效数据比例过高的行/列“进行判断，并按行列去除。不过，由于图表类型的差异，这个步骤对不同图表的判断标准有一定区别。例如，对垂直柱状图，就应允许很长的垂直有效数据。以下为效果示例：

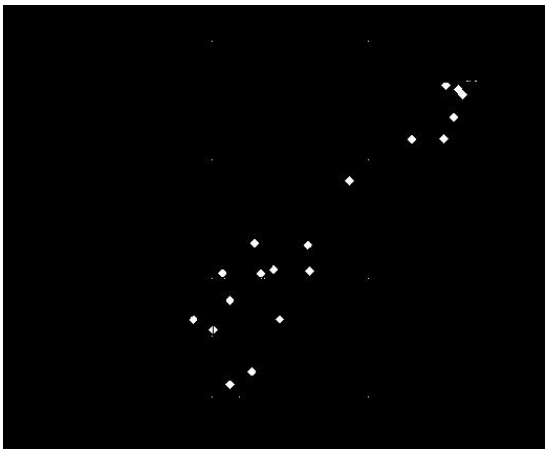


虽然这种处理并不足以完全消灭坐标轴，但在各类图表的判断过程中可以减少很多干扰，尤其对于折线与散点图。

除此之外，经过测试，还有一个非常有效的预处理，也即文字蒙版。具体来说，我们可以认为文字框所框出的区域均为无效区域。将这些区域去除后，图表会变得更加整洁，从而易于分析有效数据对应的像素，效果如下（为方便展示，此处未处理坐标轴）：



虽然本步骤的效果与OCR的准确度有很大关联，由于大部分情况下OCR的框定是准确的，采用蒙版可以显著改善效果。若进一步采用坐标轴删除，上方的图片可以最终成为：



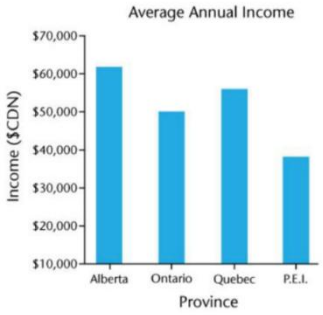
从这样预处理后的结果中提取数据点会远远易于原图。

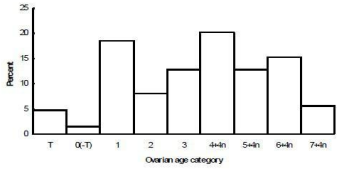
接下来，我们需要根据不同图片的特性，分别进行提取像素。这一步的输出应为有效数据的像素序列，如垂直柱状图就应输出每个立柱顶部所在的像素排成的列表。

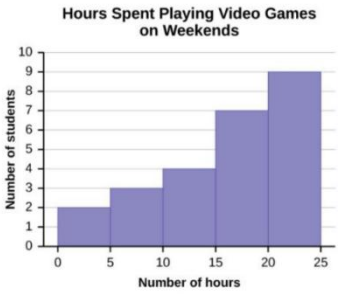
12.4 柱状图

虽然柱状图分为了水平柱状图与垂直柱状图两类，但总体的思路是一致的，水平柱状图的主要差别也仅仅是垂直柱状图处理方式中的 xy 互换，由此，我们以垂直柱状图为主介绍柱状图的处理。

简单观察分析得到，柱状图基本可以分为三类：实心、空心与直方图，示例如下：

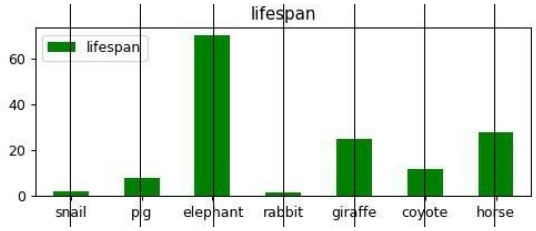


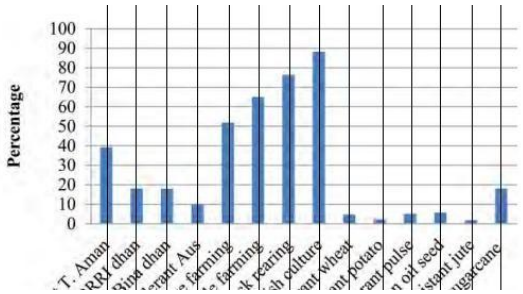




其中，直方图是最特殊的一类，因为其水平标签并不与垂直数据一一对应。然而，由于柱状图本身标签偏移等情况多，立柱个数无法代表垂直数据（因为可能存在无高度的立柱），经过实际测试，考虑直方图时的结果还不如直接不考虑（统计也可发现直方图比例极低），于是直接假设标签与立柱一一对应。

此外，由于标签可能倾斜，定位立柱的中心应以标签对应的矩形框上方两顶点的中点为准，如下图黑线：





在确定对应的列后，就必须确定立柱顶点所在的像素。由于立柱本身的特性，无论是实

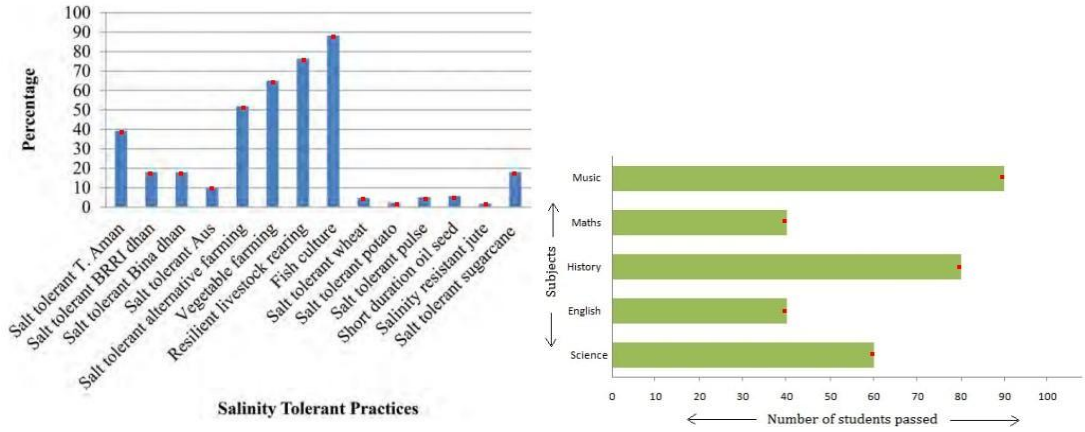
心还是空心，都应该考虑每个像素点相邻的方形区域，因此，在这一列，对每个像素点(r,c)周围的 img_1[r-1:r+2, c-5:c+5]进行求和，并记为 box 数组，这里 img_1 是已经二值化后的图片。

有了数组后，起初的想法是考虑这个数组中从上往下最大的增长序列，因为这很可能意味着进入立柱的部分，但经过实际测试，事实上不如简单方案：

```
m = np.max(box)
if m >= 8:
    for r in range(3, r_end):
        if box[r] > m * 0.9:
            points.append(r)
            break
else:
    points.append(r_end - 5)
```

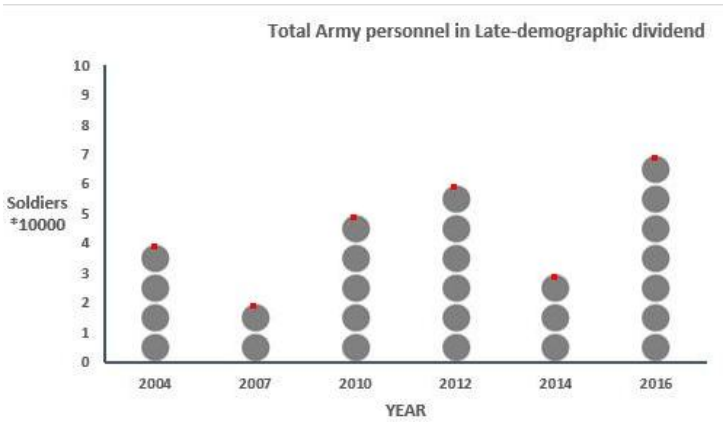
直接计算本列中最密集的区域，若最密集的 box 有效数据不足 8，则认为不存在立柱，按纵坐标最低点作为像素估计，否则，从上往下找到第一个附近有效数据相对较多的点，并将其作为立柱的顶点。

对水平柱状图，整体思路是完全相同的，只是具体参数有差别。最终找到的数据像素如下如红点：



12.5 点图

比起柱状图，点图的立柱更加不规则，但类似策略仍然可行，结果如下图：



不过，点图的关键处理在于，它只需要找到与确定的顶点最接近的 y 轴值，因为每一列

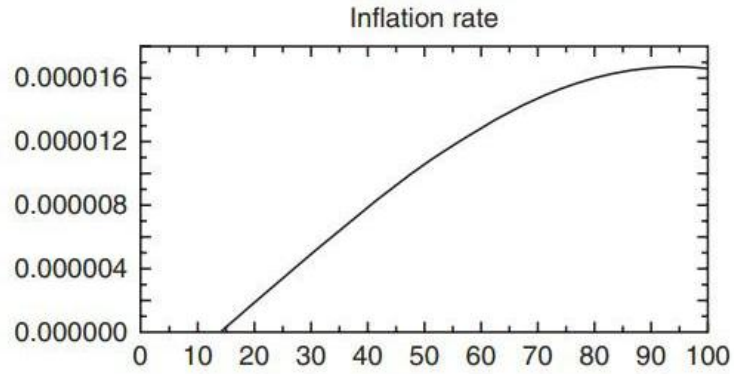
都必然处在离散的某个位置。由此可以直接写出（这里 base 代表像素，val 代表对应的值）：

```
def points_to_vals_dot(points, labels):
    # 点图只需要寻找最接近的标签的 y 值
    N = len(labels)
    base = np.zeros(N)
    val = np.zeros(N)
    for i in range(N):
        base[i] = (labels[i][1][1] + labels[i][2][1] + labels[i][3][1] + labels[i][4][1]) / 4
        val[i] = to_num(labels[i][0])
    for i in range(len(points)):
        points[i] = val[np.argmin(np.abs(base - points[i]))]
    return points
```

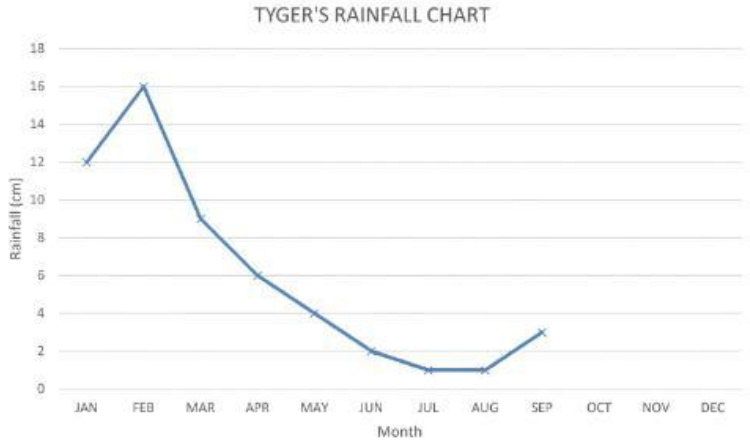
对其他类型的图片，需要利用插值才能确定值，因此放在之后的部分介绍共同的处理方式。对于上方的点图，输出即为[4, 2, 5, 6, 3, 7]。理论来说，即使无坐标轴，也可以通过数点或计算距离的方式得到点的个数，但这两种方法的稳定性都远不如直接寻找最高点，且给出的训练图片中完全没有无坐标轴的点图，因此直接用此更稳定的方式进行处理。

12.6 折线图

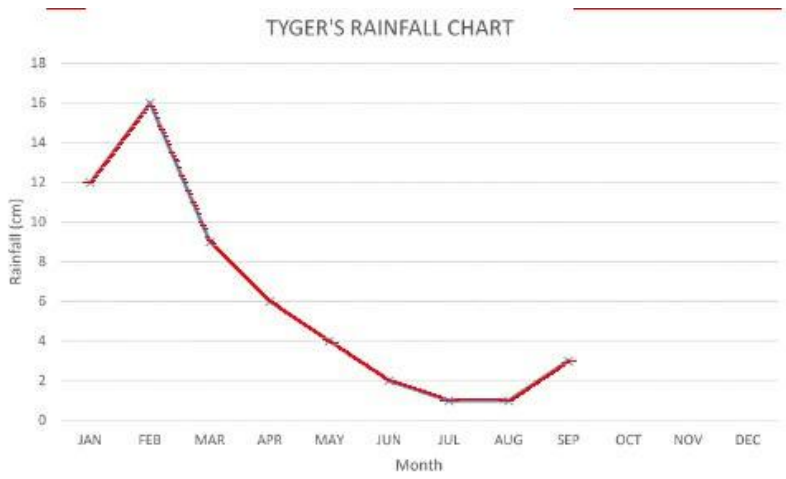
比起柱状图，折线图的处理要复杂很多。这是由于它的数据区域很少，且容易混淆。此外，折线图会出现标签上方无值的情况，此时这个标签不应该被计算，如下图中的 0 与 10 就不应有对应值：



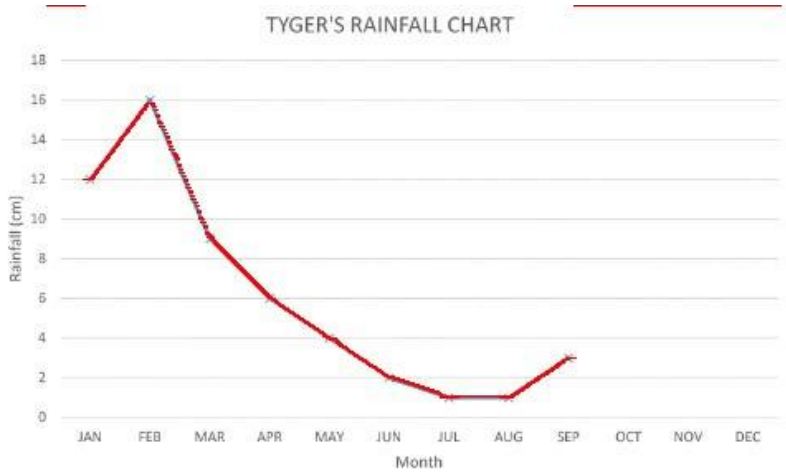
我们仍然返回一列与标签数量相等的像素数组，不过，若标签上方没有线，与柱状图填充最低点不同，应当直接返回 Nan，并在后续正确忽略。就上图的输出而言，前两个分量应为 Nan，此后才应正确出现值。为方便后续说明，考察下图：



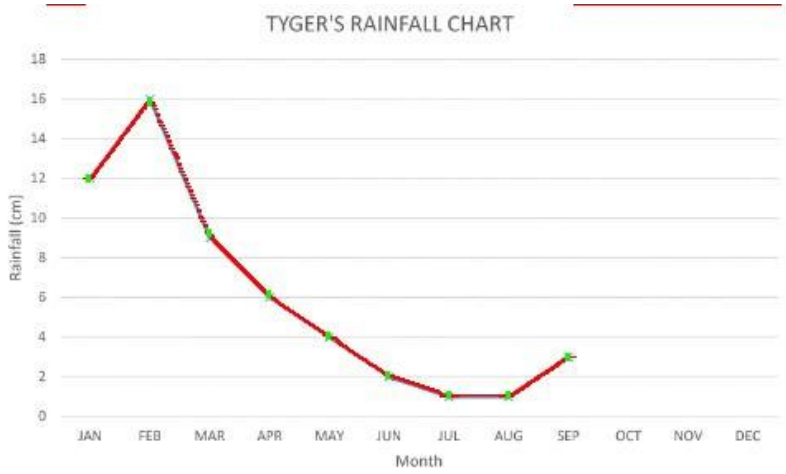
对折线图的处理中，我们采用了全局的思路，也即不只关注标签上方的部分，而是考虑整个图片中的折线。先找到数据区域每一列中最密集的部分，如下图中红色：



接着，对其进一步进行连续化处理，结果如下图：



此处标记在最顶部意味着并没有在有效数据中找到，可以发现红线基本完整刻画了折线的部分。接下来，得到有折线部分的标签对应的点，如下图标记的绿点：

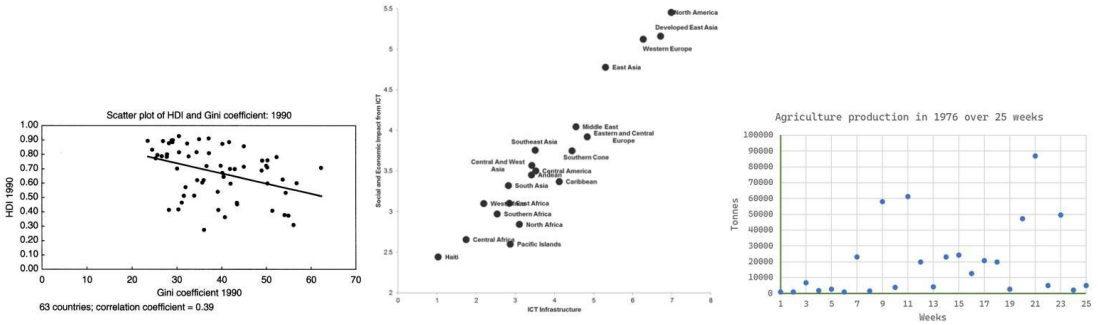


这样即可找到所有要求的像素，最终输出为[105, 59, 138, 175, 200, 224, 236, 236, 213, nan, nan, nan]，符合舍弃标签的要求。

在上述的基本流程下，我们还加入了一些细节操作，例如用周围平均填充中间的 Nan，对边缘标签适当扩大查找范围以确定是否真的无值等。在这些操作的综合控制下，对折线图的处理稳定性得到了有效的提升。

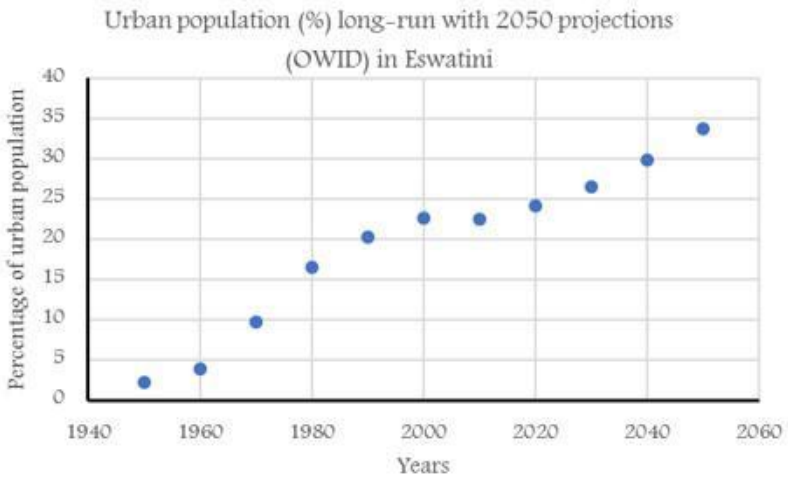
12.7 散点图

对散点图的处理是任务中最困难的一部分，这是由于散点图经常表现出极差的性状：

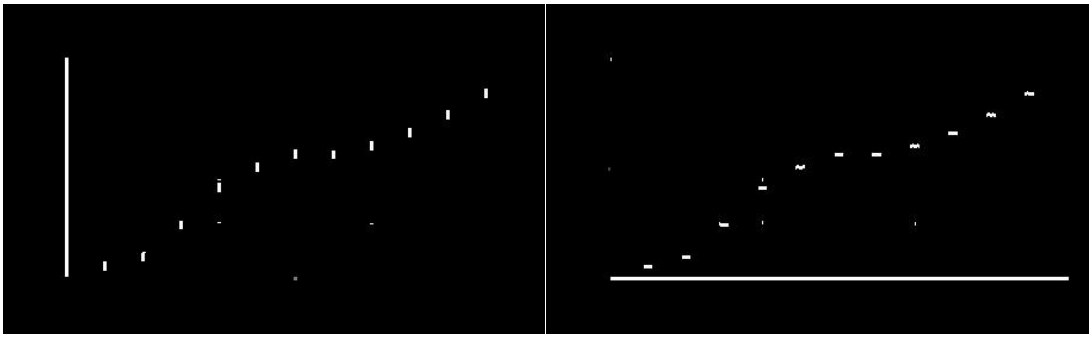


上方展示了图中的拟合线、文字、落在坐标轴上的散点，实际还有空心实心、不同形状、密集以致重合等复杂的情况。又由于比赛规则若无法正确统计个数则样例直接 0 分，首当其冲的是正确判断散点的个数。

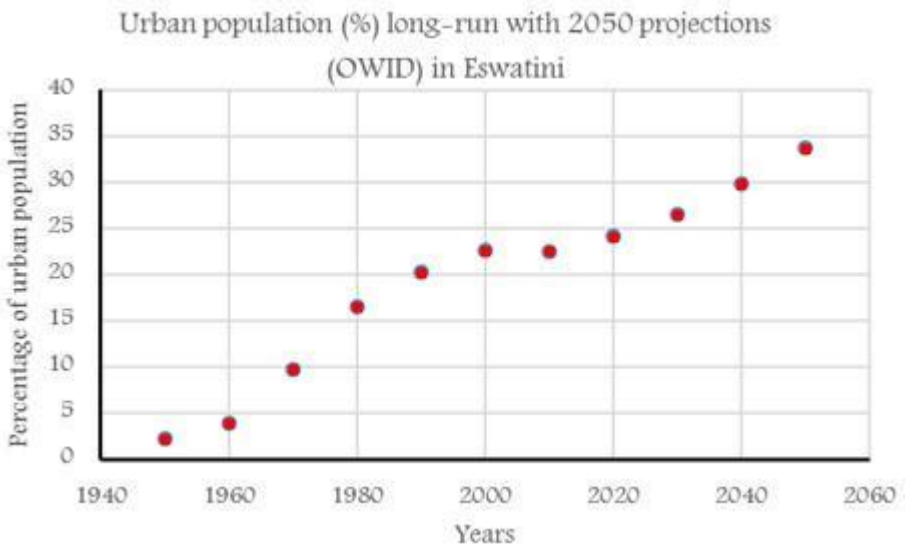
下面仍以简单例子出发介绍对散点图的处理流程：



首先，对每行考虑，寻找其中足够长的连续有效数据部分的中点，并全部标记，再对每列类似考虑，得到的结果如下方两张图：



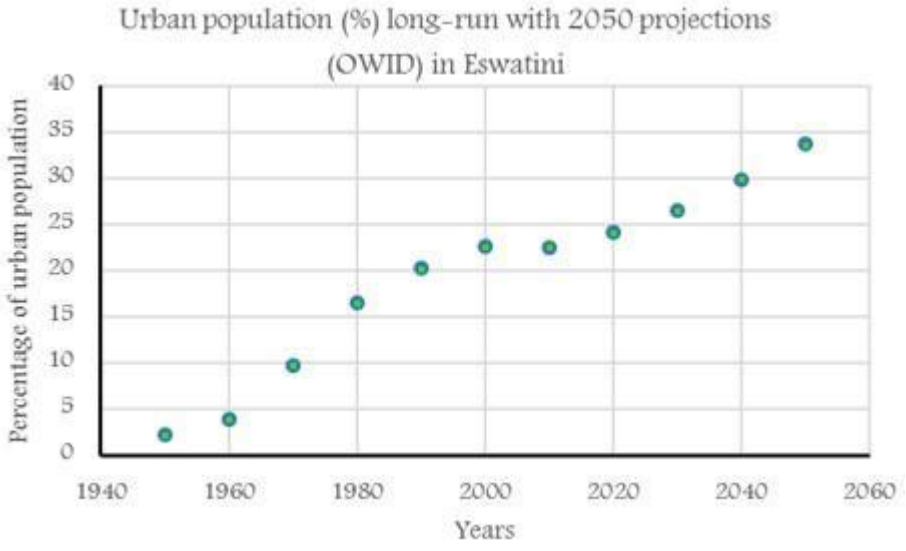
通过行中心与列中心的交叉，我们可以初步确定散点的中心备选，如下图红点（此处事实上有着较复杂的判断逻辑，如标记中点时同样标记了有效段的长度，若过长则很可能是未消除干净的坐标轴，过短则未必是有效的散点）：



但是，走到这步后，又出现了一个问题：为了不漏算，一般每个散点中都有较多的中心备选点，如何确定这些中心备选点如何合并？这就必须先估算散点的大小。

由于标记中点时也标记了有效段的长度，每个中心备选点的横竖都构成了一个类似十字的形状，根据两侧的长度即可以估算散点的直径。最终实践表面较好的估算方式是，取每个中心备选点对应十字较小边的最大值，并将其平方后乘比例进行面积的估算（估算面积是由于绝大部分散点都是圆形，计算半径较方便）。例如，上图最终估算出的半径为 3.7 像素。

最终，对于每个中心备选点，合并其半径以内的中心备选点（这里合并是指将新的位置记为所有被合并的点的均值），直到完全合并完成，最终得到的中心点如下图绿点：



此图片输出为[[75, 405], [92.67, 373], [108, 341], [119, 309], [126, 245], [126, 277], [137, 213], [154, 181], [185, 149], [212, 117], [220, 85]]，符合真实结果（注意这里第二个分量代表 x 值，第一个分量代表从上到下的 y 值）。

12.8 坐标轴与标签插值

由于接收输入包含 OCR 得到的坐标轴，很重要的步骤是从坐标轴的值中得到像素对应的真实值，并返回。注意到除了点图以外，剩下的四类图片都是以列表的形式得到所有 x 坐标或 y 坐标，只要进行对应转化即可，这意味着可以写一个坐标轴插值的函数统一处理，函数

接收 OCR 得到的坐标轴文字/位置，输入像素列表，输出值列表。

不过，在介绍此函数之前，坐标轴还有一个在预处理阶段已经利用了的作用：框定数据区域。虽然不同图表的细节有差别，但数据区域的框定基本是以 y 标签最低点向上、x 标签最左点向右的部分为准的。预先框定区域后，可以减少不少对无效标签的识别。

插值函数的基本思路是直接根据所有轴标签的像素 base 与值 val 进行线性拟合，并对整个输入列表应用拟合结果（若输入 nan，输出也会对应部分为 nan，恰好解决折线图的问题）：

```
N = len(val)
w = (N * np.dot(val, base) - np.sum(val) * np.sum(base)) / (N * np.dot(base, base) - np.sum(base) ** 2)
b = np.mean(val - w * base)
points = w * np.array(points) + b
return points
```

但是，考虑到 OCR 结果可能有偏差，必须有更多增强鲁棒性的举措，如文字转数字时考虑更多情况（这里涉及逗号可能表示小数点或单纯分隔符）：

```
def to_num(label : str):
    labeled = ""
    for word in label:
        if word in "-0123456789.":
            labeled += word
        elif word == "(":
            if label.find("(") == 0:
                labeled += "-"
        elif word == ",":
            if label.find(".") == -1:
                l = label.split(",")
                if len(l) == 2 and len(l[1]) != 3:
                    labeled += "."
        elif word == "b":
            labeled += "6"
        elif word == "B":
            labeled += "8"
        elif word == "K":
            labeled += "000"
    try:
        return float(labeled)
    except ValueError:
        return None
```

并且清理没有按顺序排列的标签值：

```
flag = True
while flag:
    flag = False
    for i in range(N):
        if i > 0 and i < N - 1:
            if val[i] > max(val[i-1], val[i+1]) or val[i] < min(val[i-1], val[i+1]):
                N -= 1
                val = np.delete(val, i)
                base = np.delete(base, i)
                flag = True
                break
```

由于只需要两个准确的标签就可以得到正确的线性拟合结果，此处的处理均以最大限度舍弃不良值为主要目的。

十三、生成输出

13.1 输出格式

经过上一部分后，对每类图片都已经完成了值列表的生成，于是整个代码的最后部分是对输出进行生成。主办方要求对每张图片的提交格式是：

图片名_x, x 轴标签, 图片类型

图片名_y, y 轴标签, 图片类型

的形式，且 x-y 的对应需要以一定的顺序进行排序。由于其他部分都已经在前面完成，这里需要额外处理的就是 x 轴标签 x_str 与 y 轴标签 y_str 的生成。

以折线图与散点图为例，最终的输出方法如下，这里 val 是上一部分得到的值列表，N 是 vals 列表的长度：

```
elif type == "line":
    for i in range(N):
        if not np.isnan(vals[i]):
            flag = False
            if xlabels[i][0] == "": xlabels[i][0] = "0"
            x_str += xlabels[i][0].replace(",", ".").replace(";", ".").replace('\"', "")  + ";"
            y_str += str(vals[i]) + ";"
elif type == "scatter":
    for i in range(N):
        flag = False
        if np.isnan(vals[i][0]) or vals[i][0] == np.inf or vals[i][0] == -np.inf:
            x_str += "0;"
        else:
            x_str += str(vals[i][0]) + ";"
        if np.isnan(vals[i][1]) or vals[i][1] == np.inf or vals[i][1] == -np.inf:
            y_str += "0;"
        else:
            y_str += str(vals[i][1]) + ";"
```

此处 flag 代表至少找到了有效的结果，否则需要用其他方法进行填充。对于折线图，只有 vals 对应部分不为 nan 的数据才需要被放进结果中，否则会直接忽略。在将 OCR 得到的 x 轴标签记录进结果时，也必须留意分隔符本身可能引起的格式问题。而对于散点图，则需要注意数据产生 nan 或 inf 时，需要用 0 进行填充才能进行正常评判。

由此，通过每张图两行进行写入的方式，即能得到符合提交要求的文档。

13.2 最终输出结果的自主评价工具

最后，虽然结果评判只需要主办方进行，为了优化我们的结果，我们也必须有自己的评价函数。运用主办方在笔记本中提供的 score_series，我们可以直接进行系列的比较，而类似上一部分中得到 x 和 y 对应的序列后，系列比较就变得简单了，类似下图：

```
if img_type == "vertical_bar":
    for xlabel in xlabels:
        x.append(xlabel[0])
    y = vals
x_real = []
y_real = []
for now in data['data-series']:
    if now['x'] is not None:
        x_real.append(now['x'])
    if now['y'] is not None:
        y_real.append(now['y'])
if img_type == "vertical_bar":
    sc[0].append(score_series(x_real, x))
    sc[0].append(score_series(y_real, y))
```

与上图展示的初值柱状图比较类似可以得到任何一类图片的比较方式，从而可以根据训练集数据进行分类评分，而在训练集中的真实图片的评分结果为（这里只有四项是由于真实图片中并没有任何点图）：

综合准确率 0.7297246032816991
水平条形图准确率 0.863293900061776 数量 457
垂直条形图准确率 0.964380631953434 数量 72
折线图准确率 0.6730265946717434 数量 423
散点图准确率 0.4027358453977667 数量 165

虽然经历了复杂的优化过程，对散点图极低的综合评分表明了传统方法仍然存在较大的局限性，不过，根据比赛中的观察可以证明，这个稳定结果已经较良好了。

十四、可能的进一步工作及处理情况

在中期我们已经得到比赛成绩之后，我们就认真总结并归纳出了如下可能的进一步工作，并记录在 Kaggle 的 notebook 中，我们在这个地方标记出这些问题是否已解决，以及这些解决后的结果是否有效：

（1）标签溢出问题：即识别出来的坐标轴标签数量大于实际坐标轴标签数量，此问题易于解决，且训练集因此准确度下降 6.7%：

a、将坐标轴识别为图中的点（原因是圈太多了，圈数大于坐标轴标签数）；

b、将坐标轴识别为标题（原因是标题被识别成了好几段，段数大于坐标轴标签数）；

c、将纵坐标最下方的点或纵轴标题的一小段识别入横坐标轴标签中，最为常见（原因是最下方的点的纵坐标和横坐标轴标签的纵坐标接近）；

d、将无字符的地方识别为奇怪的字符，置信度低，但会混入横坐标轴标签中；

e、坐标轴识别的容错度 d 太大，导致较近的别的字符串也被识别进去；

我们解决了其中的 a，b 两个问题，得分有所提升，但提升近似于无，c 问题处理起来难度大且会产生别的错误，因此未处理，d，e 两个问题的比重较小，也并未处理；

（2）横纵坐标标签多行问题，未处理；

（3）横坐标相近的黏在一起，或许可以根据比例切开，未处理；

（4）分类器训练可以沿中心截取 96 像素大小而不是 64，已完成；

（5）特殊符号问题

a、优化被杀掉字符如引号的输出，使其被允许；

b、将明显错误的特殊符号进行更合理的替换；

已经过简单处理，还需进一步处理，但提升不会太大；

（6）Exception 处理：或许有许多 exception 只是单纯是意外情况，发现并处理可能能增加准确性，已尽量完成；

（7）OCR 的训练及参数调整：

a、detection 的训练，暂定输入全部训练集，可调参数；

b、recognition 的训练，暂定 0-5000 而已，可调参数；

c、调用 ocr 时有许多参数可以研究；

已完成，效果还可以，参数还可以进一步调整；

（8）图片糊度问题，或许可以做些图像增强再使用 OCR，未处理；

（9）多角度 OCR，可以多 OCR 一些旋转角度，增加识别出的可能，已处理，是负优化；

（10）集成 OCR 模型，可以增添新的 OCR 同时输出结果取较好，未处理；

（11）特殊图表问题

a、直方图的标签处理问题，未处理；

b、折线图横轴标签上方无点问题，已尽力处理，需进一步优化；

c、其他的特殊情况（待补充）；

（12）纵轴的 45 度旋转识别，已处理，是负优化；

（13）横轴漏标签，或许可以依照距离进行补齐，未处理；

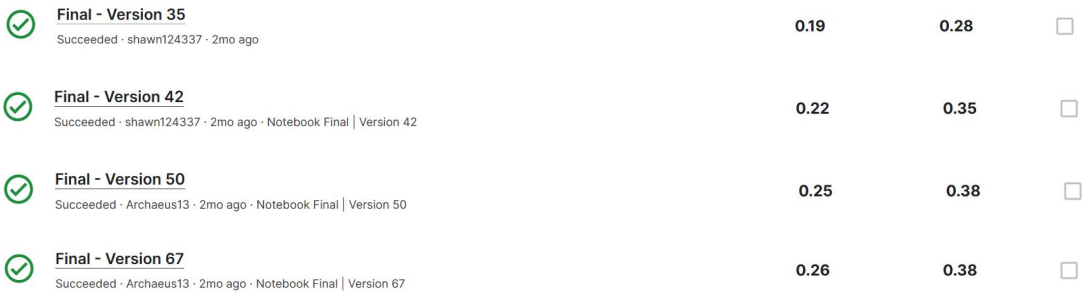
（14）横轴标签数量太少，会识别为别的东西，未处理；

（15）可以考虑多截取几个框分类进行投票，可能分类更准，未处理。

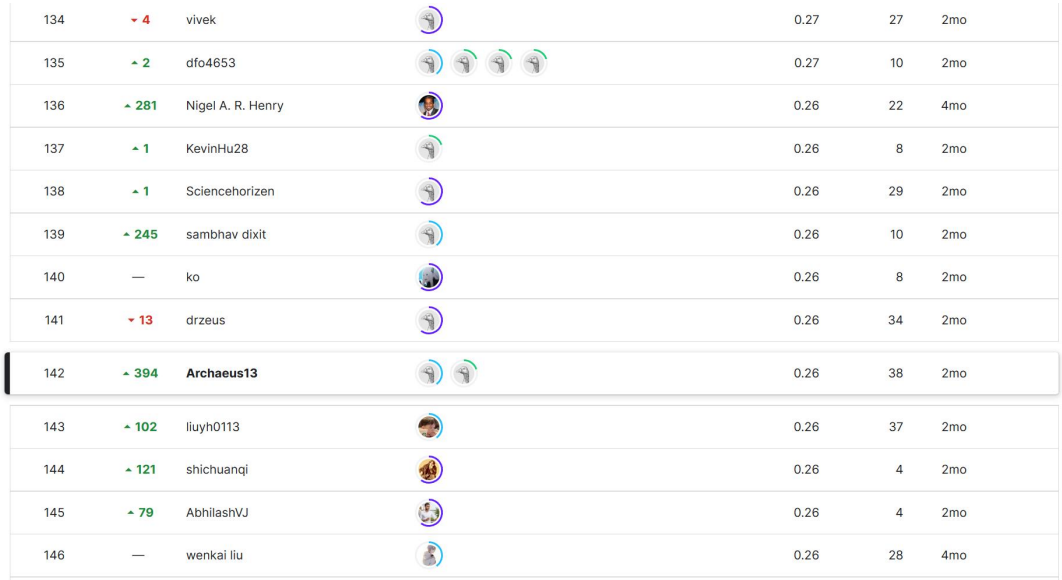
以上的进一步工作我们采取的是先对影响最大的工作进行解决的方法，到比赛截止，我们已经尽力完成了这些工作中我们能想到好办法又不繁琐的部分，也有了不小的提升。

十五、最终结果与成绩排名

我们总共进行了 38 次提交，对有效的提交，在公共测试集上的结果从 0.26 改进到了最高 0.38，而在最终测试集上（比赛结束前不可见）则是从 0.19 提升至了 0.26：



我们的最终排名为 142/609，也即 23% 左右的位置：



排行榜链接如下：

https://www.kaggle.com/competitions/benetech-making-graphs-accessible/leaderboard。

十六、比赛总结与心得体会

从最开始的选题到比赛完整结束，我们经历了颇多的波折。事实上，我们差点就选择了 Predict Student Performance from Game Play 项目，最后选中这个比赛的理由也颇简单：它是一个容易看到应用的、有趣的项目，并且不止需要在深度学习模型的角度考虑，还需要与各种过程结合。

我们较早就开始了比赛的各项内容准备和磋商，两个人之间也进行了大量的交流，从交流时间中可以看到，我们在这次比赛中是投入了大量且长线的时间的（我们基本没有怎么聊别的事情）：

比赛的过程中，将图片一步步转化成数据的过程也的确让我们感受了不少折磨：整体处理流程的复杂、数次的提交报错、模型导入的麻烦、图表的各种奇怪情况……或许这也是这个赛事参与人数相对不多的原因，它自身的门槛导致完成有效提交都已成为了一件困难的事（光是每次提交并评判结果就需要一小时余的时间）。事实上，我们为此投入了两周多的几乎全部时间，方才完成了这样一个完整的结果，而在此之中，我们大约花费了 20 个以上的完整小时在 debug 上，实际花费的时间相当之多。

尽管如此，我们从其中的每一步都能得到不错的收获，而这样的快乐是套模型调参数的“炼丹”过程中无论如何也体会不到的。此处简单列举一些经验与教训：

规则方面

- 在开始操作前应先确定特殊情况的规则，以方便考虑。
- 不要浪费前几天的提交次数，可以先作为提交格式的尝试。
- 代码多用 try except 以规避异常。
- 注意数据集的结构，尤其不同情况下分布的不同。
- 注意共享和提交时的设置选项。

关于模型

- 提前确认离线配置合适环境的方法。
- 现成模型可多进行比较，或用不同参数分别训练以选取较合适的。
- 可以引用预训练结果时可直接导入以控制运行时间。
- 尽量寻找 kaggle 外的训练与确认方法以节省平台 GPU。
- 单个模型可以在不同处理方法后复用。

经典算法

- 算法设计时需要考虑是否适合一般场景。
- 对于过于特殊的情况，可能忽略反而能有较好总体效果。
- 当鲁棒性容易出现问题时应考虑简单的方案。
- 大部分情况下经典算法耗时比起训练可以忽略，但仍应注意时空复杂度。
- 注意对图像的直观判断方式与抽象特征的权衡。

代码拆分

- 事先确定工作流程与主要负责部分。
- 做好接口约定以方便连接，尽快完成才能有充分时间研究整体。
- 接口提供尽量多的信息，易于充分利用。
- 自行设计对每个部分分别的评分以确认整体问题的主要来源。
- 优先对容易修改/测试周期短的可能优化方式进行检验。

遗憾的是，由于时限，我们还有不少可能的改进方向没有进行充分测试（具体见前文）。不过，从这样一个或许有些另类的深度学习赛事中，我们或多或少感受到了将深度学习融入研究的方式。一方面，对于分类、OCR 这样的深度学习强项与基础步骤，我们可以充分利用已有结果，用模型避免复杂的重复造轮子过程；另一方面，对更多细化的问题，例如比赛中的获取数据部分，用一些传统算法的思维进行模型设计，或是在模型结果上进行传统算法优化，都是可取的方法。

十七、附：队伍分工

黄弈骁在本次工作中主要负责的内容是前半部分内容，即本文七到十一节的部分，包括训练集与测试集内容读取与图表输入，分类器的搭建与预训练，OCR 模型的搭建与预训练，坐标轴位置与内容的识别，另外也负责整合了双方的工作；郑滕飞在本次工作中主要负责的内容是后半部分内容，即本文十二到十三节的部分，包括从图表读取数据（其中包含二值化，对五类图表的分类讨论处理，插值最终结果）及输出最终结果。其实双方也大量参与了对方负责的工作并在此期间进行了大量的交流，并最终共同完成了最后的这份比赛报告。

最后，感谢老师与助教一学期以来的辛苦付出，也祝深度学习课程越开越好！