手写数字识别项目报告

雷颖

目录

1 需求分析

文本是人类最重要的信息来源之一，我们的生活中充满了各种形形色色的文字符号；然而这些文字往往是我们眼睛可以看到的，而非文本信息，当我们想要分享或者录入这些信息，往往只能人工判断这些信息是什么，然后将其转为文本信息，这样虽然准确率高，但是却十分浪费人力、精力、时间。

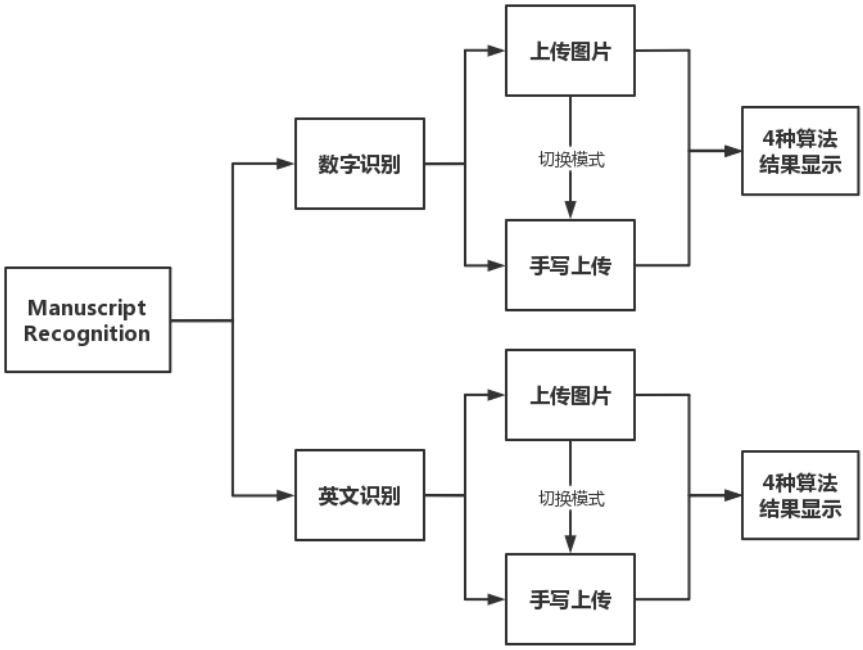
随着科技的发展，识别文本的需求变得越来越大；作为学生的我们，时常苦恼如何将老师板书上的公式、数字写入文档中，往往我们需要花费大量精力去仔细比对是否打错；作为银行的工作人员，也时常花费很多时间去比对银行票据、工商报表、财务报表、统计报表等各种表格系统.....这就使得识别文本变得十分重要。

2 功能介绍

产品形式为基于 Bootstrap+flask 框架下搭建的 web 网页，名称叫”Manuscript Recognition”。

2.1 功能总览

本产品支持两种类型的手写识别：数字识别、英文识别；支持上传图片、手写上传两种模式；并且采用四种算法识别，最后将识别结果显示在页面上。



2.2 上传数字图片

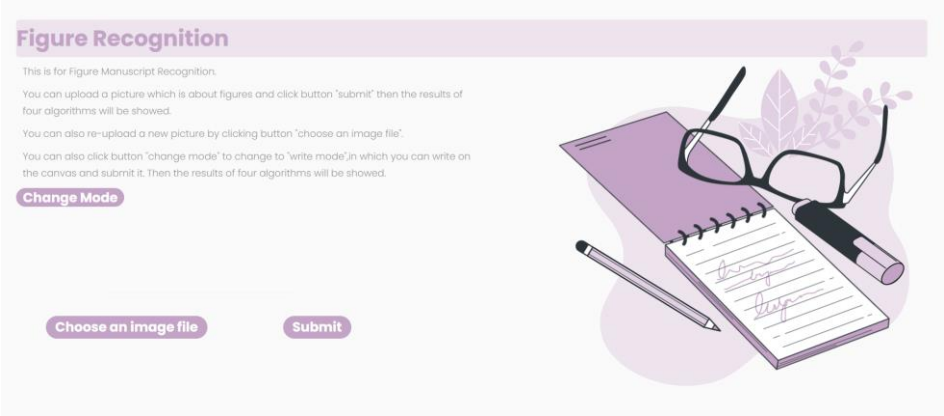
在上传数字图片页面，我们首先是写了一段文字介绍，简短地介绍了一下本产品如何上传图片并提交；上传图片和提交的按钮同样采取了文字提示的方法，这样可以让用户更加清楚如何操作，减少误操作的可能。

除此以外我们的按钮都设置了悬浮(hover)特效，使得当鼠标放在按钮上时，将其大小放大 1.1 倍，这样用户就可以知道，这是可以点击的，点击了是有用处的；同样也提示了操作，使用户更加清楚操作。

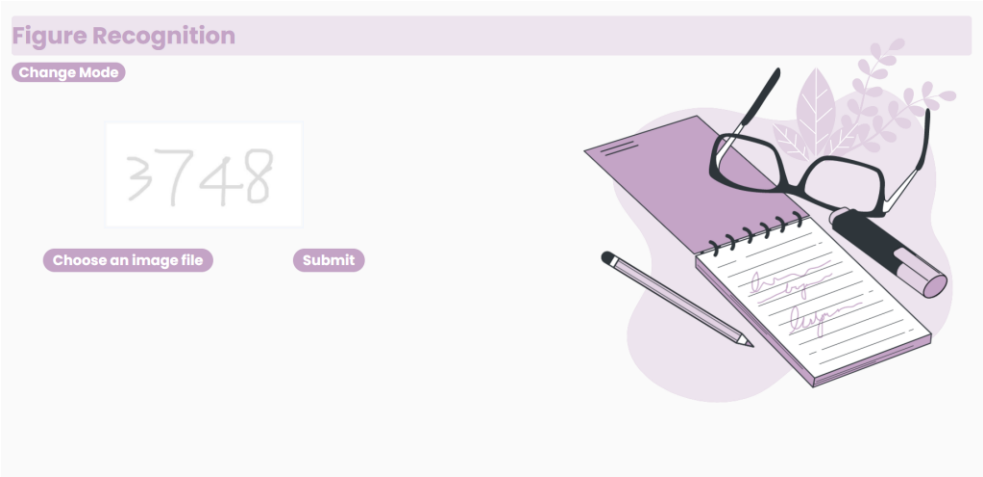
上传图片的实现是通过<input type=" file" />标签实现的，提交操作也是通过<input/>标签实现的，主要原理是通过 change 函数感知到这两个的元素的操作，然后通过 ajax 传到后端处理，然后后端将结果返回给前端显示。上传图片的实现是通过将图片传给后端，后端将图片保存在服务器中，然后后端将保存的地址传回前端，前端将元素的 src 设置为传回来的地址，这样就实现了图片的预览。

提交的实现是通过将图片的地址传给后端，后端根据地址取出图片，然后进行一系列操

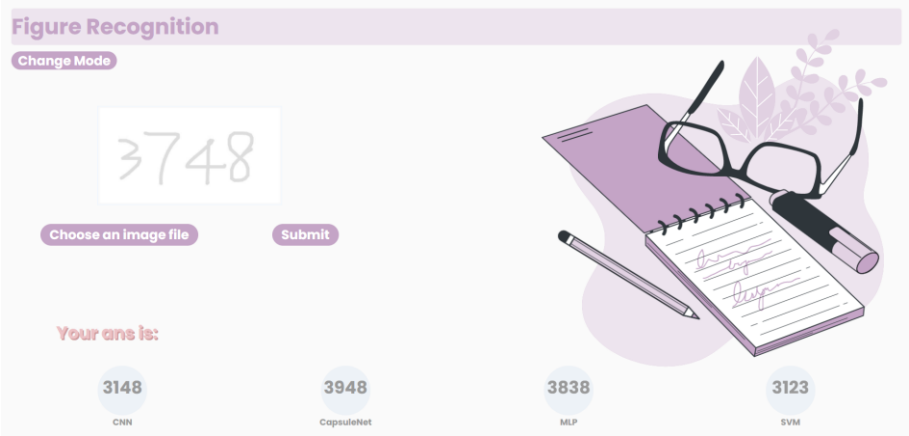
作识别出数字，然后将结果数组传回前端，前端此时就将显示结果的元素的 style.display 置为”flex”即可展示结果。页面如下：



图片预览：(这里上传图片后就将前面的介绍不显示，为图片预览和结果显示留足空间)



结果显示：

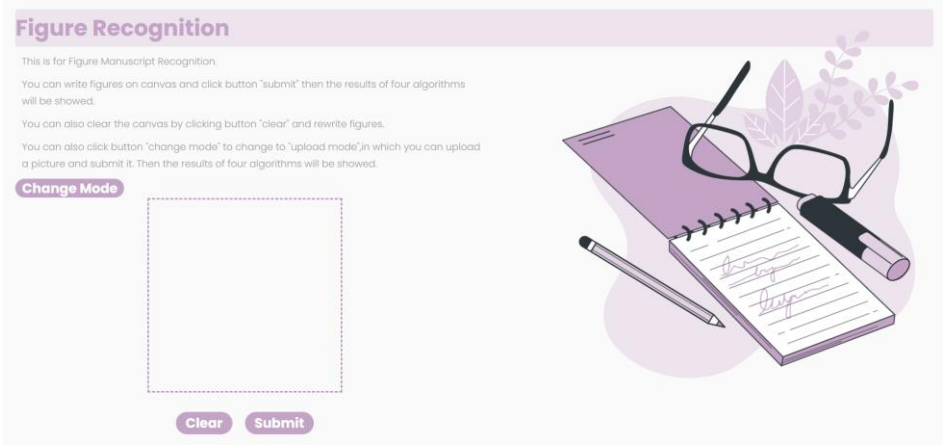


2.3 手写数字模式

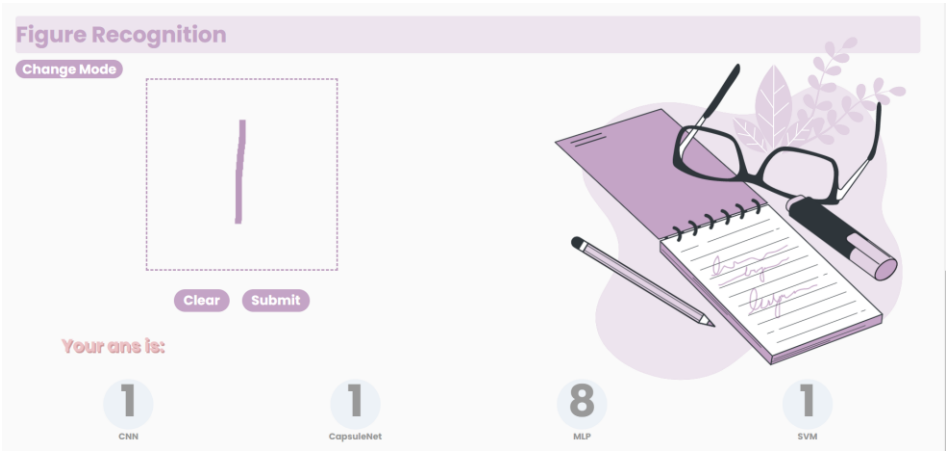
在手写数字页面，我们同样先写了一段文字介绍，简短地介绍了一下如何手写数字然后提交。这里我们通过一个 change mode 的按钮可以从上传图片模式切换到手写数字模式下，同样也可以切换回上传图片模式。

手写的实现是通过<canvas>标签实现的，通过识别鼠标在 canvas 上的坐标来绘制鼠标的移动路径。当鼠标按下移动时，识别鼠标在 canvas 坐标，然后通过 canvas.getContext(‘2d’).lineTo(x,y) 实现绘制。清空的实现是通过 canvas.getContext(‘2d’).clearRect()函数实现的。

上传的实现是通过将 canvas.toDataURL()函数将画布转换为 base64_str 的图片（base64_str 是 canvas.toDataurl 转换而来的一种图片编码形式），然后通过 ajax 将该图片传给后端，后端将该图片解码然后保存在服务器中，然后进行一系列操作识别出数字(这里我们没有像上传图片一样将图片返回前端，而是直接处理，这是因为 canvas 已经显示在前端了，没必要再传回图片预览），然后将结果数组传回前端，前端此时就将显示结果的元素的 style.display 置为”flex”即可展示结果。



手写后提交：

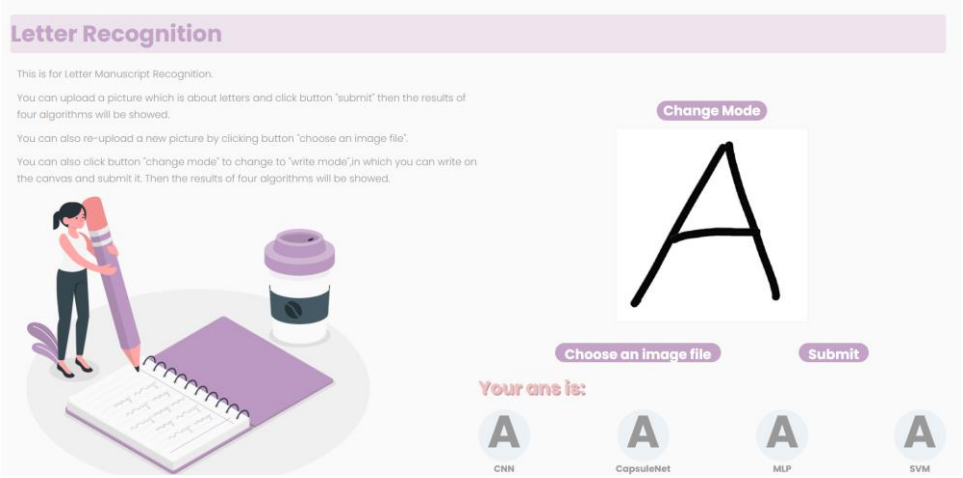


2.4 上传英文图片

这里实现原理和上述上传数字图片一样，因此不过多赘述。

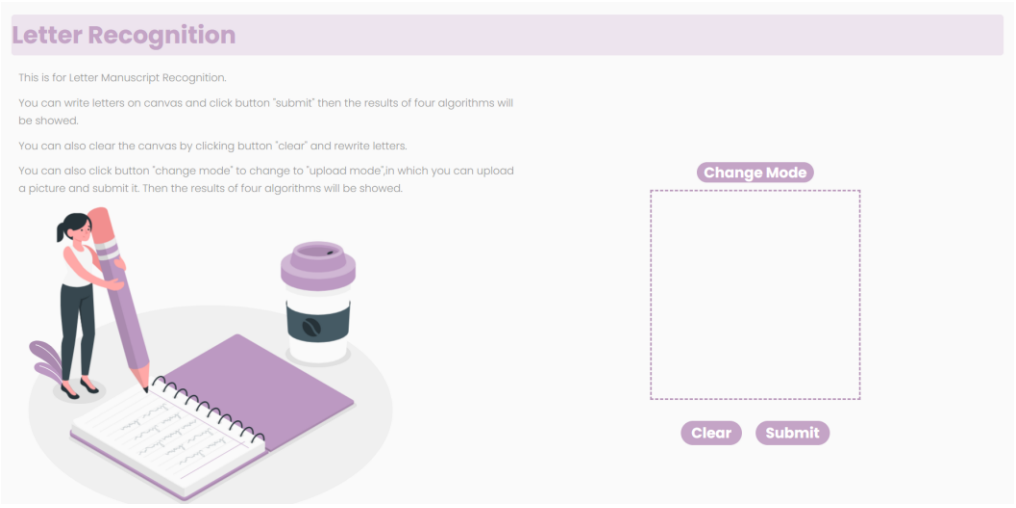
图片预览：

结果显示：

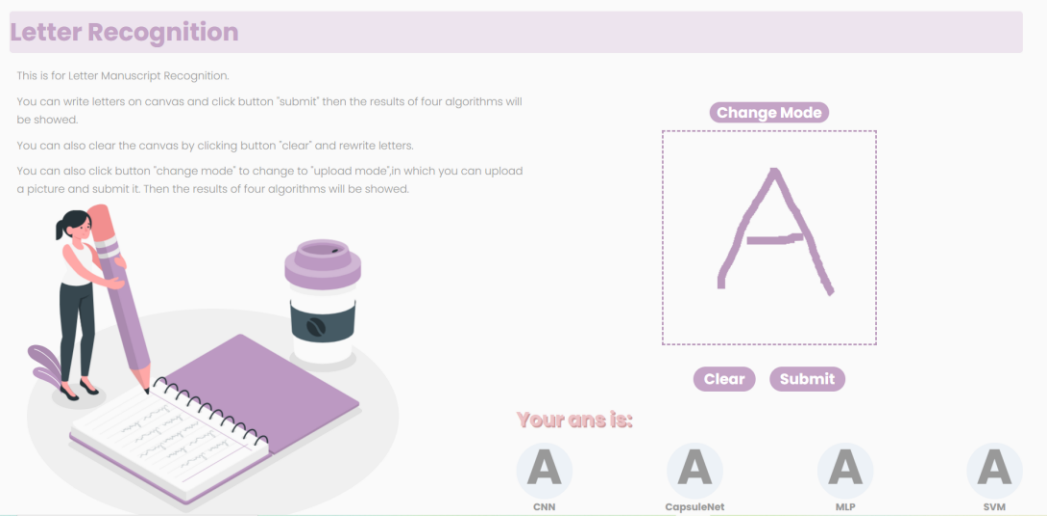


2.5 手写英文模式

这里实现原理和上述手写数字上传一样，因此不过多赘述。



手写后提交：



2.6 简易操作

本产品支持多种交互，操作容易；提供 sidebar 方便快速切换，多种按钮提供不同操作，并且系统完备，除了必备功能以外，还增加了系统介绍和成员介绍页面。

系统介绍：在首页和识别页面之间我们防止了本产品的特色介绍：

首页：数字 123 和英文 ABC 的图标作为 button，当点击其中某个按钮便可下滑到对应的识别页面。同样所有的 button 都增加了悬浮特效。

点击本网页左上角可弹出 sidebar，可快速切换到各页面：

3 UI 设计

3.1 配色方案

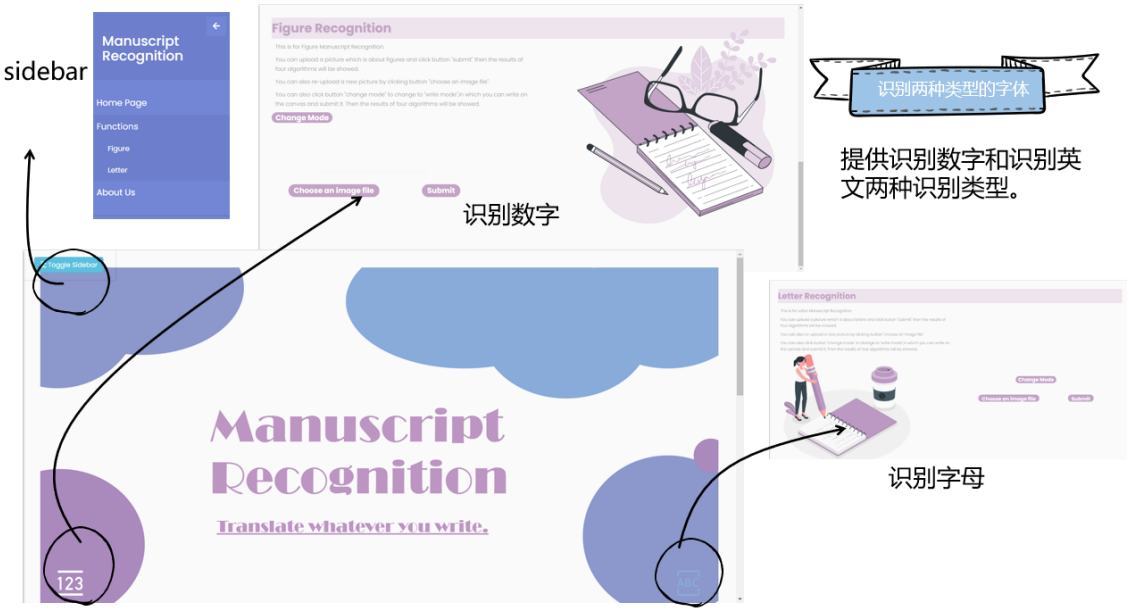
本产品配色采取相似色配色原则：相似色是指色轮上相邻的三种颜色，在色环中呈 45°角，相对来说颜色比较接近会给人一种舒服的感觉。本产品的主色调确定为紫色，辅色为粉色和蓝色，主要配色方案如下：

3.2 名片设计

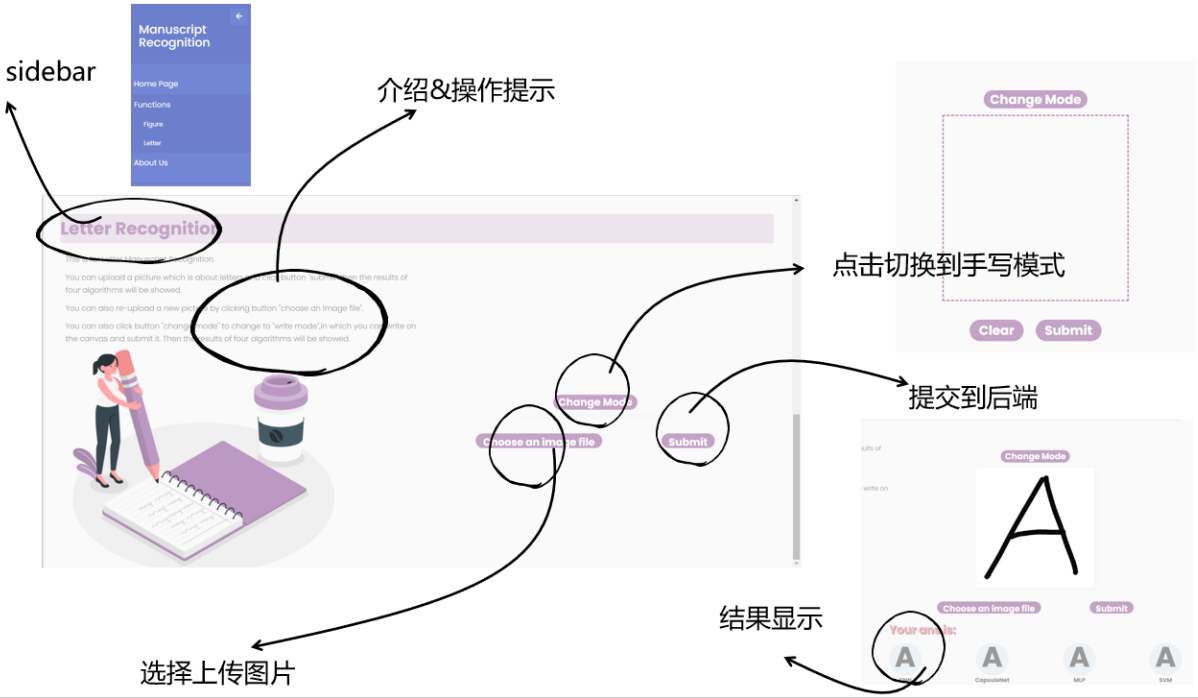
本产品的名片设计/海报设计如下，本产品的名称和标语正中名片中间，表明了我们本产品是什么，“ Translate whatever you write”精简凝练地介绍了本产品的作用。

3.3 逻辑结构

主界面逻辑结构如下：点击左上角的蓝绿色按钮，可以弹出 sidebar，可以点击 sidebar 的按钮快速切换到别的页面（识别数字页面、识别字母页面、关于我们页面）。主页面下方还有两个 icon，一个是识别数字的图标，一个是识别字母的图标；分别点击图标，页面将下滑并展示对应页面。



主要功能界面逻辑：（以识别英文为例，前文已讲述过具体操作实现，故略过）



团队介绍页面逻辑：增加了一个返回主页面的按钮，点击该按钮可以返回主页面。当然也可以通过点击 sidebar 随意跳转到你想跳转的地方去。

4 模型部分

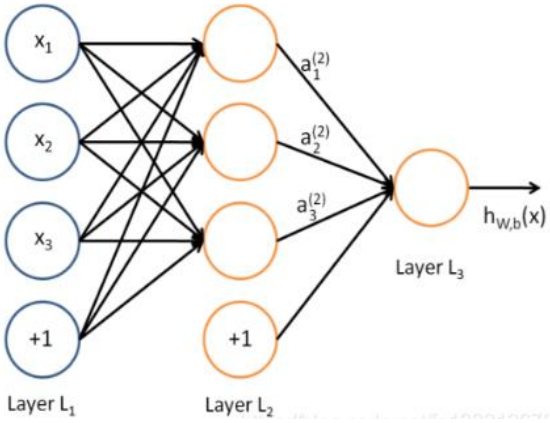
本项目基于 MNIST、EMNIST 和 The Chars74K 数据集，使用机器学习、深度学习和字符分割算法来实现对手写数字、字母的识别。

4.1 算法介绍

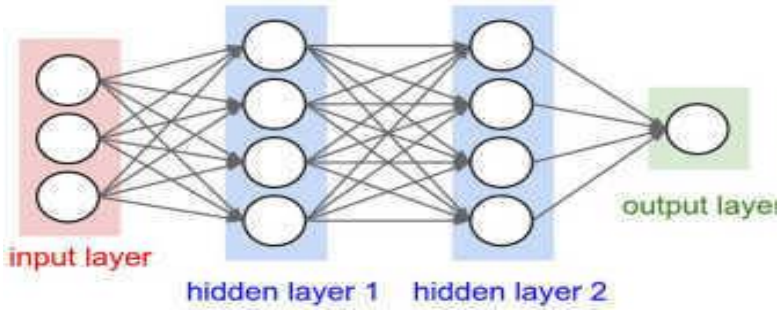
算法主要分为分类算法和字符分割算法。首先是分类算法的介绍，我们采用了机器学习和深度学习两种方法，其中机器学习选取了 SVM，Decision Tree，Random Forest 和 KNN 算法，深度学习算法选取了 MLP，CNN，和 CapsuleNet，如下图所示。

下面分别对每一种算法进行简要的介绍。

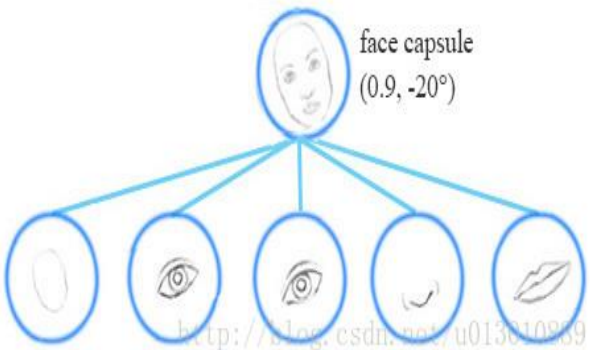
- **MLP**：多层感知机，是最简单的神经网络，其网络结构包含输入层、隐含层和输出层；



- **CNN**：卷积神经网络，能够较好地处理图片这类具有空间信息的对象，其网络结构包含输入层、卷积层、池化层、激活层和全连接层；

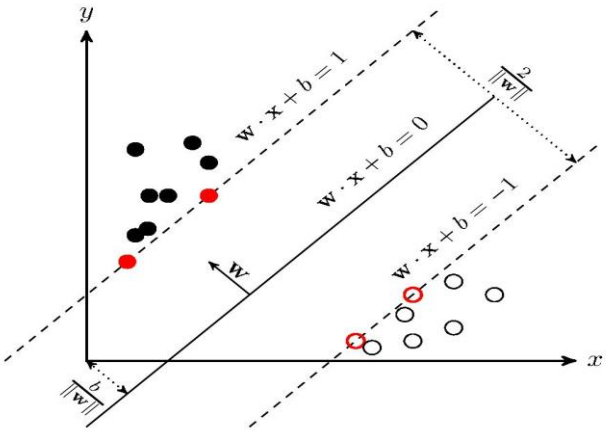


- **CapsuleNet**：胶囊网络，可以通过重构模块更好得处理空间位置间不同对象之间的关联，从而实现较 CNN 更好的识别效果；

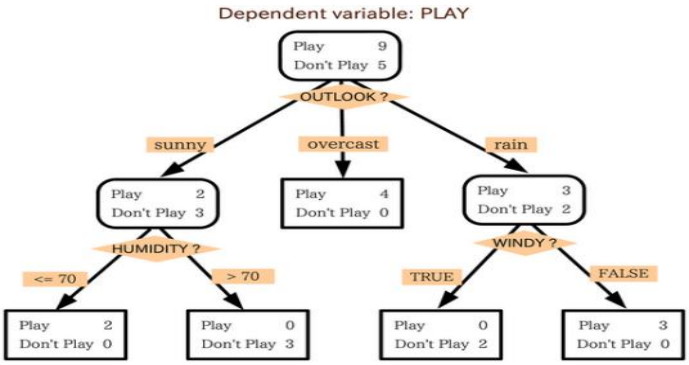


- **SVM**（Support Vector Machine）：支持向量机是一种线性分类器，基本原理是特征

空间上的间隔最大化，包括核技巧等操作；

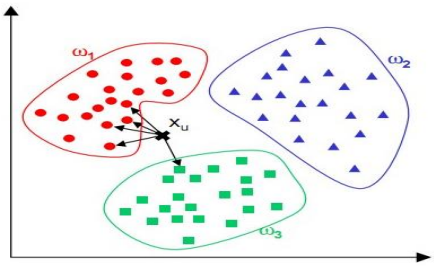


- **Decision Tree**：决策树即是一种分类算法也是一种回归算法，其叶子节点表示标签，非叶子节点表示评估条件。决策树的构建一般包括三大步骤，分别是特征选择、决策树生成和剪枝操作。

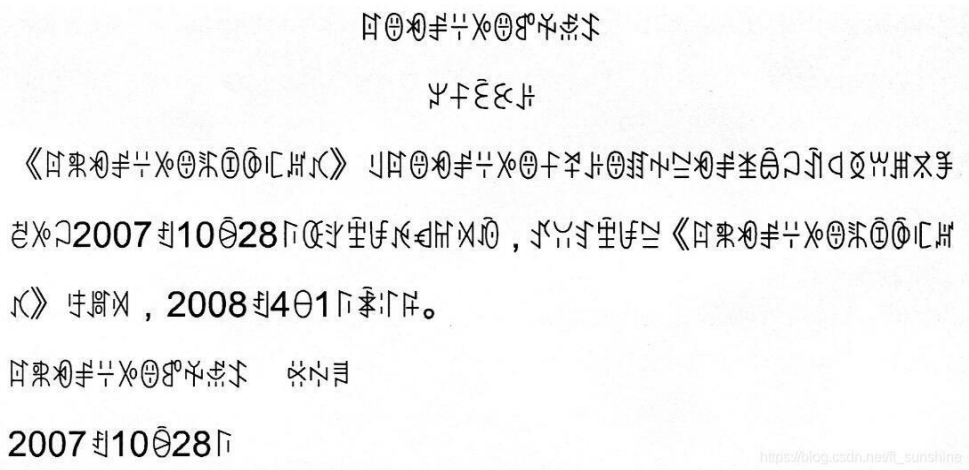


- **Random Forest**：随机森林是一种分类方法，其基本单元是决策树。本质上随机森林属于一种集成学习（Esemble Learning）方法；

- **KNN**（K-Nearest Neighbor）：K 近邻算法是一种分类算法，其基本原理是测试样本可以用它最接近的 k 个近邻来表示。



下面是分割算法的介绍。我们的字符分割算法主要采用“**投影+连通域**”的方法来实现。它大致的流程是：首先对图像进行二值化与腐蚀(断开印刷过程中可能存在的粘连)的操作，之后对预处理后的图片进行水平投影，得到图片中所有的文本行。然后分别对图片中的每一个文本行进行一系列操作，得到其连通域，然后再对连通域进行合并，针对合并错的部分进行切分操作（针对过宽的部分）。最后再对没有合并的部分高宽比很大的竖形字符进行单独的合并。进行完上述操作之后，就可以得到不错的分割效果。下图分别显示了分割前和分割后的效果。

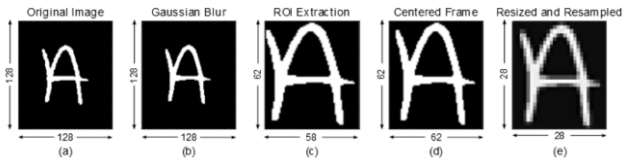


4.2 数据集简介

我们训练手写数字采用的是 MNIST 数据集，训练手写字母主要采用的是 EMNIST，为了提升模型的泛化性能，我们还加入了 The Chars74K 数据集。下面分别对这三种数据集进行介绍。

- **MNIST**：MNIST 数据集是一个包含十类数字（0~9）的数据集，总共包含 60000 项训练数据，10000 项测试数据，每张图像都是 28*28 像素的灰度图像。下面是 MNIST 数据样例图片。

- **EMNIST**：EMNIST 数据集是 26 类大小写字母的组合，每一个类别包括 4800 张训练图片，800 张测试图片。下面是 EMNIST 数据集的数据样例。



- The Chars74K：该数据集是英文字符和坎那达字符的组合，总共包含 74000 张图像，我们选取了其中的英文字符进行训练。下面是 The Chars74K 数据集英文字符的数据样例。

4.3 实践过程

我们的实践过程大概分为五大步骤，分别是导入必要的包、数据预处理、模型的构建、模型的训练和测试、以及模型的使用。下面以深度学习模型为例，分别对这些步骤进行说明。

- **导包**：导入必要的 Python 包，如下图所示；

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as Data
import torch.nn.functional as F
import torchvision
from torchvision import datasets,transforms
import warnings
import pickle
import numpy as np

warnings.filterwarnings("ignore")
```

- **数据**：数据预处理，以及 train loader 和 test loader 的构建，如下图所示；

1. Data preprocess

In [2]:
```
#导入训练数据
train_dataset = datasets.MNIST(root='./data/',                #数据集保存路径
                               train=True,                    #是否作为训练集
                               transform=transforms.ToTensor(), #数据如何处理，可以自己自定义
                               download=False)                #路径下没有的话，可以下载

#导入测试数据
test_dataset = datasets.MNIST(root='./data/',
                              train=False,
                              transform=transforms.ToTensor())
```

In [4]:
```
batch_size = 64

train_loader = torch.utils.data.DataLoader(dataset=train_dataset, #分批
                                           batch_size=batch_size,
                                           shuffle=True)          #随机分批

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
```

- **模型**：深度学习模型需要进行神经网络模型的构建，机器学习模型直接调用 sklearn 对应的库即可，如下图所示；

```
# 激活函数
def squash(inputs, axis=-1):
    norm = torch.norm(inputs, p=2, dim=axis, keepdim=True)
    scale = norm**2 / (1 + norm**2) / (norm + 1e-8)
    return scale * inputs

class PrimaryCapsule(nn.Module):
    def __init__(self, in_channels, out_channels, dim_caps, kernel_size, stride=1, padding=0):
        super(PrimaryCapsule, self).__init__()
        self.dim_caps = dim_caps
        self.conv2d = nn.Conv2d(in_channels, out_channels, kernel_size=kernel_size, stride=stride, padding=padding)

    def forward(self, x):
        outputs = self.conv2d(x)
        outputs = outputs.view(x.size(0), -1, self.dim_caps)
        return squash(outputs)

class DenseCapsule(nn.Module):
    def __init__(self, in_num_caps, in_dim_caps, out_num_caps, out_dim_caps, routings=3):
        super(DenseCapsule, self).__init__()
        self.in_num_caps = in_num_caps
        self.in_dim_caps = in_dim_caps
        self.out_num_caps = out_num_caps
        self.out_dim_caps = out_dim_caps
        self.routings = routings
        self.weight = nn.Parameter(0.01 * torch.randn(out_num_caps, in_num_caps, out_dim_caps, in_dim_caps))

    def forward(self, x):
        x_hat = torch.squeeze(torch.matmul(self.weight, x[:, None, :, :, None]), dim=-1)
        x_hat_detached = x_hat.detach()
        b = torch.zeros(x.size(0), self.out_num_caps, self.in_num_caps)

        assert self.routings > 0, 'The \'routings\' should be > 0.'
        for i in range(self.routings):
            c = F.softmax(b, dim=1)
            if i == self.routings - 1:
                outputs = squash(torch.sum(c[:, :, :, None] * x_hat, dim=-2, keepdim=True))
            else:
                outputs = squash(torch.sum(c[:, :, :, None] * x_hat_detached, dim=-2, keepdim=True))
                b = b + torch.sum(outputs * x_hat_detached, dim=-1)
        return torch.squeeze(outputs, dim=-2)

class CapsuleNet(nn.Module):
    def __init__(self, input_size, classes, routings):
        super(CapsuleNet, self).__init__()
        self.input_size = input_size
        self.classes = classes
        self.routings = routings
        self.conv1 = nn.Conv2d(
            input_size[0], 256, kernel_size=9, stride=1, padding=0)
        self.primarycaps = PrimaryCapsule(
            256, 256, 8, kernel_size=9, stride=2, padding=0)
        self.digitcaps = DenseCapsule(in_num_caps=32*6*6, in_dim_caps=8,
                                      out_num_caps=classes, out_dim_caps=16, routings=routings)
        self.relu = nn.ReLU()

    def forward(self, x, y=None):
        x = self.relu(self.conv1(x))
        x = self.primarycaps(x)
        x = self.digitcaps(x)
        length = x.norm(dim=-1)
        return length
```

```
#model = CNN()
#model = MLP()
model = CapsuleNet([1,28,28],10,3)
```

- **训练和测试**：利用预处理好的数据集对构建好的模型进行训练，并保留权重文件，以便后续的使用，如下图所示；

3.训练

```
optimizer = optim.Adam(model.parameters(),lr=1e-3)
criterion = nn.CrossEntropyLoss()
scheduler = optim.lr_scheduler.MultiStepLR(optimizer,milestones=[1,2,3,4,5,6,7,8,9], gamma=0.7)
```

```
from tqdm.notebook import tqdm

epoch = 5
total_step = 0

for epoch in range(epoch):
    for step, (b_x, b_y) in tqdm(enumerate(train_loader)):
        total_step += 1
        b_x = b_x
        b_y = b_y
        output = model(b_x)
        loss = criterion(output, b_y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if total_step % 100 == 0:
            model.eval()      #测试模式，关闭正则化
            correct = 0
            total = 0
            for x,y in tqdm(test_loader):
                #x, y = x.to(device), y.to(device)
                outputs = model(x)
                _, predicted = torch.max(outputs, 1)   #返回值和索引
                total += y.size(0)
                correct += (predicted == y).sum().item()
            accuracy = correct/total
            print('Epoch: ', epoch, 'Step: ', step, '| train loss: %.4f' % loss.data.cpu().numpy(), '| test accuracy: %.4f' % acc
    scheduler.step()
torch.save(model.state_dict(), 'weight/figure/cnn.pkl')#保存模型
```

- **使用**：使用模型进行字符识别，首先需要读入原始图像并进行一系列的数据预处理，然后进行字符的分割，再初始化模型的权重，最后使用模型进行单个字符的识别得到结果，如下图所示。

字符分割

In [98]:
```
segmentation_result = get_segmentation_result(img,img_input) # (x,y,w,h)
print(segmentation_result)
display(img_input) # 展示字符分割的效果
```

```
[[(114, 83, 6, 166), (352, 60, 150, 183)], [(63, 334, 138, 226), (348, 323, 161, 226)]]
```

分类

In [82]:
```
'''
方法一：CNN
'''
def MYCNN(segmentation_result,img):
    cnn = CNN()
    cnn.load_state_dict(torch.load('weight/figure/cnn.pkl'))
    cnn.eval()
    results = []
    for i in range(len(segmentation_result)):
        row_results = []
        for j in range(len(segmentation_result[i])):
            x, y, w, h = segmentation_result[i][j]
            tmp_img = img[y-20:y+h+20,x-20:x+w+20]
            display(tmp_img)
            tmp_img =cv2.resize(tmp_img,(28,28))
            tmp_img = tmp_img.reshape(1,28,28)
            tmp_img = torch.tensor(tmp_img).to(torch.float32)
            tmp_img = torch.unsqueeze(tmp_img,dim = 0)
            tmp_result = cnn(tmp_img)
            pred_y = torch.max(tmp_result,1)[1].data.numpy()[0]
            row_results.append(pred_y)
        results.append(row_results)
    return results
```

In [83]:
```
MYCNN(segmentation_result,img)
```

Out[83]:
```
[[1, 2], [3, 4]]
```

4.4 结果分析

分别使用手写数字和手写字母的数据集训练模型，在测试结果集中达到了如下表所示的准确率，可以看到**英文字符集的识别中有四个模型在测试集上达到了 85%以上的准确率**，分别是 CNN 88.30%，CapsuleNet 87.43%，SVM 90.83%，和 KNN 86.53%。综合在手写数字和手写字母集上的识别效果，最后我们选取了 CNN、MLP、CapsuleNet 和 SVM 这四种表现较好的模型作为最后使用的模型。

MODEL	FIGURES	LETTERS
CNN	99.07%	88.30%
MLP	98.25%	81.95%
CAPSULENET	98.96%	87.43%
SVM	97.92%	90.83%
RANDOM FOREST	94.64%	83.80%
DECISION TREE	87.65%	70.65%
KNN	96.88%	86.53%

5 小结

本项目通过原型设计、网站开发和模型实现等技术搭建出了整套手写数字/字母识别的使用流程，总体上具有界面美观、操作便捷等用户友好型的特色保障用户良好的使用体验，以及识别准确率较高、模型推理延时低等特点来优化产品性能。不过，总结下来，我们还在模型的识别准确率等方面存在改进的空间。

一方面模型的泛化性能有待提升，另一方面字符分割算法有待改进。首先，模型虽然在数据集上表现优异，但由于现实手写字符和数据集中的字符存在着一定的差异，造成了现实中识别准确率不高的现象。这一问题可以通过将用户手写的数字图片不断加入数据集不断重新训练的方法加以改进。其次，我们使用的字符分割算法是使用方框将字符外周包围起来，再统一变化尺寸到 28*28 像素，因此存在数字变形的问题，尤其是数字 1 这种狭长的数字，从而造成识别效果差的问题。这一问题可以考虑不改变数字原有尺寸的方法进行实现。