

基于特征提取与 Xgboost 的 MNIST 手写数字识别

数据挖掘报告暨 UserGuide

中央财经大学 林应昕

目 录

摘要	3
1. 准备工作.....	4
1.1. 导入各种库.....	4
1.2. 定义一个读入 MNIST 数据集的函数 <code>load_mnist()</code>	4
1.3. 读入 Train Set	5
1.4. 展示原始图片	5
1.5. 以灰度图的形式展示	6
2. 图片数据预处理与特征提取	7
2.1. 人眼所能看到的图片 v.s 机器“看到”的图片	7
2.2. 正式的“二值化”处理	11
2.3. 全部需要提取的特征.....	13
2.4. 提取 on 像素的数量	13
2.5. 获得所有 on 像素的 X 坐标.....	14
2.6. 获得 on 像素 X 坐标均值和方差	15
2.7. 获得所有 on 像素的 Y 坐标	17
2.8. 获得 on 像素 Y 坐标均值和方差.....	19
2.9. 获取 on 像素 X 坐标与 Y 坐标相关系数.....	20
2.10. on 像素 X 坐标平方乘以 Y 坐标的平均	22
2.11. 最小矩形框相关特征的提取.....	24
2.12. on 像素边缘数的获取.....	28
2.13. 特征提取	32
3. 划分测试集与训练集	34
3.1. 样本划分函数	34
3.2. 划分	34
4. 特征描述与分析	35
4.1. 训练集特征的描述性统计	35
4.2. 特征间相关性	35
4.3. 特征剔除	37
5. 模型训练与逐步调参	37
5.1. 设置初始模型	37
5.2. 训练初始模型，并查看准确率	38
5.3. 调整惩罚项 γ	38
5.4. 调整学习率	42
5.5. 调整参数 Lambda.....	42
5.6. 绘制最优模型的训练曲线	43

5.7. 绘制混淆矩阵	45
6.总结	47

摘要

本报告基于 MNIST 手写数字图片数据集，调用 numpy 库提取相关特征，进而训练 Xgboost 手写数字识别模型。经过逐步调参后，最优模型在测试集上的准确率可达 88%。

关键字：图像识别 Xgboost 特征提取 MNIST

1. 准备工作

1.1. 导入各种库

In[1]:

```
import os
import struct
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
from copy import deepcopy
import pandas as pd
warnings.filterwarnings("ignore")
get_ipython().run_line_magic('matplotlib', 'inline')
```

1.2. 定义一个读入 MNIST 数据集的函数 load_mnist()

每一张 MNIST 手写数字图片实质上一个 28×28 的矩阵。通过 load_mnist() 函数，我们可以把 Train Set 中（共有 60000 张图片）的每张图片拉直为 1 个长度为 $28 \times 28 = 784$ 的行向量，并依次沿竖直方向进行堆叠；也即，最终我们读入的是一个 60000×784 的大矩阵。

In[3]:

```
def load_mnist(path, kind='train'):

    """Load MNIST data from `path`"""
    labels_path = os.path.join(path, '%s-labels-idx1-ubyte' % kind)
    images_path = os.path.join(path, '%s-images-idx3-ubyte' % kind)
    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II', lbpath.read(8))
```

```
labels = np.fromfile(lbpath, dtype=np.uint8)
with open(images_path, 'rb') as imgpath:
    magic, num, rows, cols = struct.unpack('>IIII', imgpath.read(16))
    images = np.fromfile(imgpath,
                         dtype=np.uint8).reshape(len(labels), 784)

return images, labels
```

1.3. 读入 Train Set

```
# n: 样例个数; p: 每张图的像素点总数
```

```
# In[4]:
```

```
file_path = "C:\\\\Users\\\\Lenovo\\\\1 MY_Code\\\\My Data for 期末\\\\MNIST\\\\raw"
X_train, y_train = load_mnist(file_path)
n, p = X_train.shape
```

1.4. 展示原始图片

```
# In[5]:
```

```
sns.set()
fig = plt.figure(figsize=(14, 6))
plt.subplots_adjust(wspace=0.03, hspace=0.00001)
for i in range(10):
    plt.subplot(2, 5, i+1)
    img = X_train[y_train == i][0].reshape(28, 28)
    plt.imshow(img ,
               #cmap='Greys',
               interpolation='nearest'
               )
    #plt.axis('off')
    plt.xticks([])
    plt.yticks([])
plt.show()
```



Fig.1 原始图片

1.5. 以灰度图的形式展示

下面，我们以灰度图的形式，展示 Train Set 中不同数字的第 1 个样例（详见 Fig. 2）。

In[6]:

```
sns.set_style("whitegrid")
fig = plt.figure(figsize=(14, 6))
plt.subplots_adjust(wspace=0.03, hspace=0.00001)
for i in range(10):
    plt.subplot(2,5,i+1)
    img = X_train[y_train == i][0].reshape(28, 28)
    plt.imshow(img ,
               cmap='Greys',
               #interpolation='nearest',
               )
    #plt.axis('off')
    plt.xticks([])
    plt.yticks([])
plt.show()
```



Fig.2 MNIST 图片的灰度图形式

由 Fig.1 和 Fig.2 可知，每 1 张原始 MNIST 图片并非二值图，其每个像素点的数值在 0 ~ 255 之间。因此，在彩图（Fig.1）和灰度图（Fig.2）模式下，不同像素点的颜色会有所不同。

2. 图片数据预处理与特征提取

2.1. 人眼所能看到的图片 v.s 机器“看到”的图片

在对图片进行二值化前，需剔除部分数值较小的像素点；而不能只是简单地把所有像素值 >0 的像素点都取为 255。下面说明具体理由。

2.1.1. 人眼能看到的图片

In[8]:

```
sns.set()
fig = plt.figure(figsize=(10,10))
plt.subplots_adjust(wspace=0.001, hspace=0.001)
for i in range(49):
    plt.subplot(7,7,i+1)
    img = X_train[i].reshape(28, 28)
    plt.imshow(img ,
               #cmap='Greys')
```

```
#plt.axis('off')
plt.xticks([])
plt.yticks([])
plt.suptitle("Panel(A) Human View", fontsize=18, y=0.92)
plt.show()
```



Fig.3 原始图片 (人眼可见)

2.1.2. 简单二值化后，机器“看见”的图片

所谓简单“二值化”，即是把每一张图片拉直形成的行向量中，所有非 0 像素点都设为 255。下图中，我们对简单二值化的效果进行可视化（Fig.4，只展示前 49 个样例的二值化结果）。

```
# In[9]:
```

'''简单的二值化处理'''

```
x_train = deepcopy(X_train)
x_train[x_train != 0] = 255
```

```
# In[10]:
```

'''画图'''

```
fig = plt.figure(figsize=(10,10))
plt.subplots_adjust(wspace=0.001, hspace=0.001)

for i in range(49):
    plt.subplot(7, 7, i+1)
    img = x_train[i].reshape(28, 28)
    plt.imshow(img ,
               #cmap='Greys',
               #interpolation='nearest'
               )
    #plt.axis('off')
    plt.xticks([])
    plt.yticks([])

plt.suptitle("Panel(B) Computer View", fontsize=18, y=0.92)
plt.show()
```



Fig.4 简单二值化后的 MNIST 图片

对比 Fig.3 和 Fig.4，不难看出，经过简单的二值化处理后，不同阿拉伯数字对应的 MNIST 图片的特征变得更为模糊不清。例如，第 3 行、第 4 列的数字“8”看起来更像是数字“1”。这说明其原有的特征被“弱化”了，可能不利于后续的模型训练与识别。因此，我们需改用其它的二值化方法。

2.2. 正式的“二值化”处理

在每 1 张图片中，我们把像素值<全部像素值均值的像素点变为 0；其他则全部变为 255。这一方法充分考虑了不同图片间的“异质性”，也即，不同人写字的轻重可能各不相同。下面是具体的 Python 代码和处理结果（详见 Fig.5，只展示前 49 个样例的二值化结果）。

In[281]:

'''二值化处理'''

```
xx_train = deepcopy(X_train)
threshold_mat = np.repeat(np.mean(xx_train, axis=1).reshape(n, 1),
                         p, axis=1)
xx_train[xx_train <= threshold_mat] = 0
xx_train[xx_train != 0] = 255
```

In[55]:

'''画图'''

```
fig = plt.figure(figsize=(10,10))
plt.subplots_adjust(wspace=0.001, hspace=0.001)
for i in range(49):
    plt.subplot(7,7,i+1)
    img = xx_train[i].reshape(28, 28)
    plt.imshow(img ,
               #cmap='Greys',
               #interpolation='nearest'
               )
    #plt.axis('off')
    plt.xticks([])
    plt.yticks([])
plt.suptitle("Panel(C) After Transfer", fontsize=18, y=0.92)
plt.show()
```



Fig.5 正式二值化后的 MNIST 图片

对比 Fig.4 和 Fig.5，不难发现，Fig.5 中图片特征的保留效果更好。因此，我们用 Fig.5 对应的二值化方法进行后续操作。

2.3. 全部需要提取的特征

参考马老师 PPT 上的“手写字母识别任务”中的特征提取方式，我们对每张 MNIST 图片提取以下特征（共计 17 个，详见 Table1）。

Table.1 需要提取的各种特征

编号	名称	特征中文名称	提取所用到的 Python 包
1	ON	二值化后，on 像素点 (=255) 的数量	Numpy
2	x_mean	on 像素 X 坐标均值	Numpy
3	x_std	on 像素 X 坐标标准差	Numpy
4	y_mean	on 像素 Y 坐标均值	Numpy
5	y_std	on 像素 Y 坐标标准差	Numpy
6	Pearson	on 像素 X 坐标与 Y 坐标相关系数	Numpy
7	x2_y	on 像素 X 坐标平方乘以 Y 坐标的平均	Numpy
8	y2_x	on 像素 X 坐标乘以 Y 坐标平方的平均	Numpy
9	x	最小外接矩形中心的 X 坐标	OpenCV2
10	y	最小外接矩形中心的 Y 坐标	OpenCV2
11	W	最小外接矩形的宽度	OpenCV2
12	H	最小外接矩形的高度	OpenCV2
13	Angel	最小外接矩形的倾斜角度	OpenCV2
14	L	从左扫描 on 像素边缘数	Numpy
15	R	从右扫描 on 像素边缘数	Numpy
16	Up	从上扫描 on 像素边缘数	Numpy
17	Down	从下扫描 on 像素边缘数	Numpy

2.4. 提取 on 像素的数量

on_num()函数接收训练集或测试集（即，摊成一个行向量、且经过堆叠后的全部图片矩阵），并返回一个 1 维 array。array 的长度为 data 中图片（即，样例）的数量。

```
# In[237]:
```

```
def on_num(data):
    return np.count_nonzero(data, axis=1)
```

2.5. 获得所有 on 像素的 X 坐标

`xtick_on()`函数用于接收训练集 or 测试集；然后，返回一个和训练集 or 测试集等大小的矩阵，每一行的每个、非 0 元素对应 该位置的 on 像素的 X 坐标。

In[14]:

```
def xtick_on(data, m=28):

    '''data: 训练集 or 测试集;
    n: 样例数量;
    m: 表示每张图片是 m * m 的矩阵'''

    if np.ndim(data) == 1:
        n = 1
    n, _ = data.shape
    result = np.ravel(np.repeat(np.arange(1, m + 1).reshape(1, m),
                                repeats=m,
                                axis=0)) # axis=1 时, 有大坑
    result = np.repeat(result.reshape(1, m * m),
                       repeats=n,
                       axis=0)
    result = (data != 0) * result
    return np.float64(result)
# 得转成float, 否则下面的代码会报错: cannot convert float NaN to integer
```

下面，我们计算 Trian Set 中前 30 张图片全部 on 像素的 X 坐标，然后用 heatmap 进行可视化。以此简单验证下是否计算错误。

In[15]:

```
fig = plt.figure(figsize=(40,20))
sns.heatmap(xtick_on(xx_train[:30]),
            annot=False, # 热力图不显示具体数字
            yticklabels=y_train[:30])
plt.yticks(fontsize=25)
```

```

plt.ylabel("Label of Sample", fontsize=30)
plt.xlabel("Index of Pixel", fontsize=30, y=0)

plt.show()

```

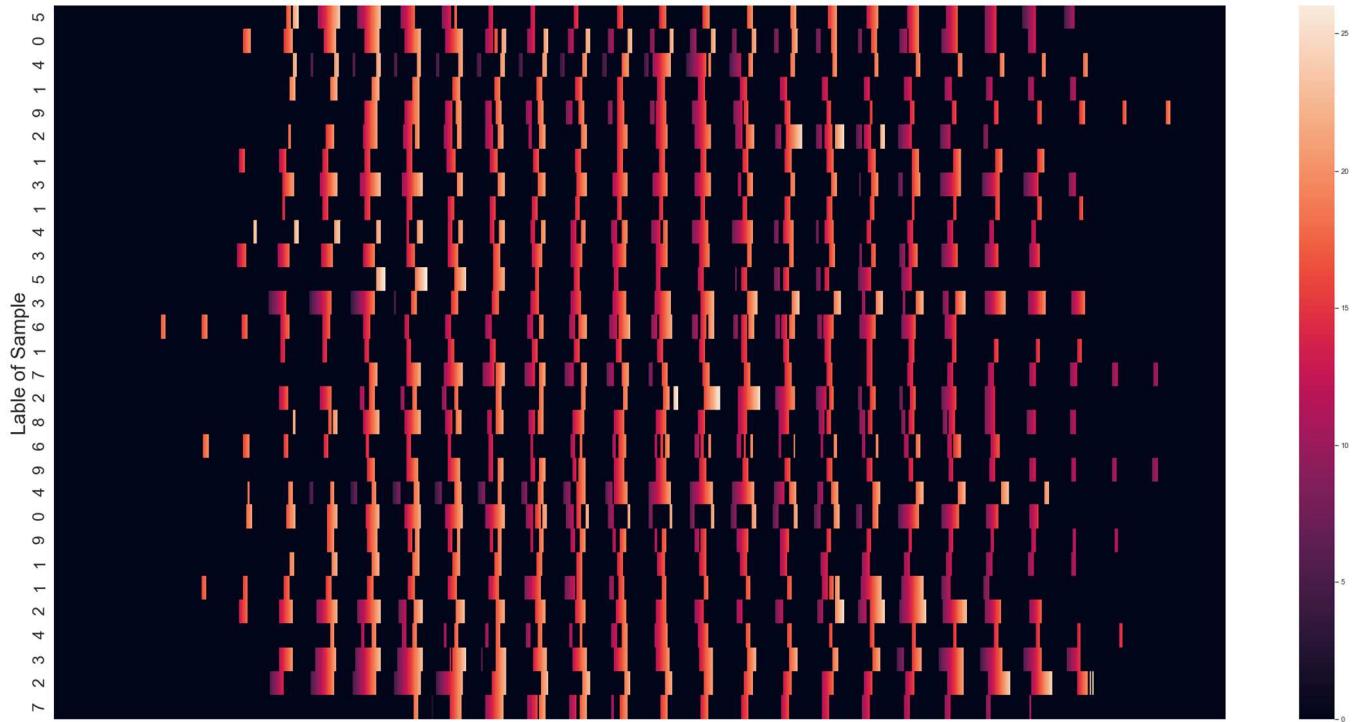


Fig.6 on 像素 X 坐标的可视化（前 30 张图片）

Fig.6 的每 1 行为：每一张图片在拉直后，各个像素点的 X 坐标值。Label of Sample 则表示这一行样例的标签。例如，第 1 行的 Label of Sample 为“5”，也便表示该行所对应的图片的标签是阿拉伯数字“5”。

观察 Fig.6 可知，heatmap 中色块呈现出“色温逐渐增加→骤降为 0→再次逐渐增加→……”的周期性变化。这与每一个 MNIST 图片矩阵中，每一行 on 像素的 X 坐标逐渐递增的规律相一致。因此，可以大致确认没有出现计算错误。

2.6. 获得 on 像素 X 坐标均值和方差

分别定义了 2 个函数 `x_mean_on()` 和 `x_std_on()`，它们都接收训练集 or 测试集，然后各自返

回一个长度为样例数量的 array，里面分别存储有各个样例 on 像素的 X 坐标均值和标准差。

In[16]:

```
def x_mean_on(data, m=28):  
  
    '''获得 on 像素 X 坐标均值，需调用前面的 xtick_on()函数'''  
  
    result = xtick_on(data, m=m)  
    result[result == 0] = np.nan # 把0 值全部转为NaN, 方便使用  
    np.nanmean()  
  
    return np.nanmean(result, axis=1)
```

In[18]:

```
def x_std_on(data, m=28):  
  
    '''获得 on 像素 X 坐标的标准差，需调用前面的 xtick_on()函数'''  
  
    result = xtick_on(data, m=m)  
    result[result == 0] = np.nan # 把0 值全部转为NaN, 方便使用  
    np.nanmean()  
  
    return np.nanstd(result, axis=1)
```

绘图展示 Train Set 中特征 x_std 和 x_mean 的分布和相关性（详见 Fig.7）。可以看出，这 2 个特征相关性较弱，不存在较大的信息冗余，因此有利于后续的模型训练。

In[17]:

```
'''先拼接为 DataFrame'''  
  
df_temp = pd.DataFrame(np.c_[x_mean_on(xx_train), x_std_on(xx_train),  
                           np.int64(y_train)],
```

```
columns=['x_mean', 'x_std', 'Label'])
```

'''绘图'''

```
sns.set_style('whitegrid')
sns.pairplot(data=df_temp, hue='Label')
plt.show()
```

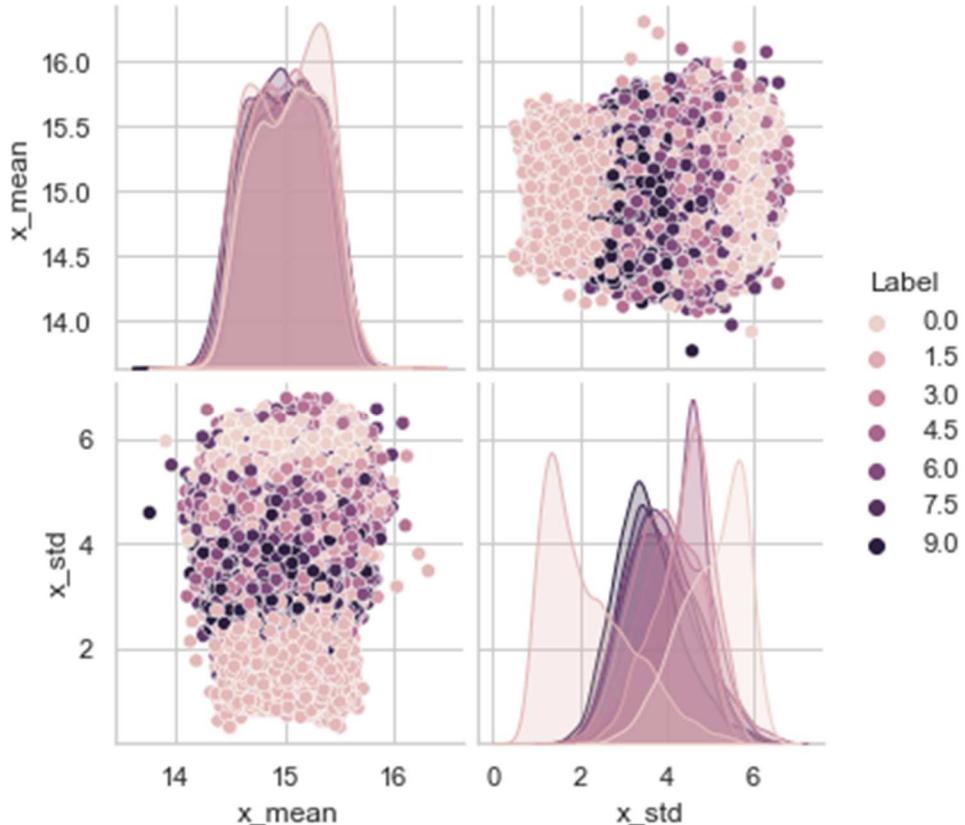


Fig.7 特征的分布与相关性

2.7. 获得所有 on 像素的 Y 坐标

`ytick_on()`函数用于接收训练集 or 测试集；然后，返回一个和训练集 or 测试集等大小的矩阵，每一行的每个、非 0 元素对应 该位置的 on 像素的 Y 坐标。

```
# In[20]:
```

```
def ytick_on(data, m=28):
```

```

'''data: 训练集 or 测试集;
n: 样例数量;
m: 表示每张图片是 m * m 的矩阵.'''
if np.ndim(data) == 1:
    n = 1
    n, _ = data.shape
    result = np.repeat(np.arange(m, 0, -1).reshape(1, m),
                       repeats=m,
                       axis=1) # If m=3, 则当axis=1时, 刚好是:
[[1,1,1,2,2,2,3,3,3]]
    result = np.repeat(result.reshape(1, m * m),
                       repeats=n,
                       axis=0)
    result = (data != 0) * result
return np.float64(result) # 转成float

```

下面，我们计算 Train Set 中前 30 张图片全部 on 像素的 Y 坐标，然后用 heatmap 进行可视化（详见 Fig.8）。以此简单验证下是否存在计算错误。

In[277]:

```

fig = plt.figure(figsize=(40,20))

sns.heatmap(ytick_on(xx_train[:30]),
             annot=False,
             yticklabels=y_train[:30])

plt.yticks(fontsize=25)
plt.ylabel("Label of Sample", fontsize=30)
plt.xlabel("Index of Pixel", fontsize=30, y=0)

plt.show()

```

Fig.8 的每 1 行为：每一张图片在拉直后，各个像素点的 Y 坐标值。Label of Sample 则表示这一行样例的标签。例如，第 1 行的 Label of Sample 为“5”，也便表示该行所对应的图片的标签是阿拉伯数字“5”。

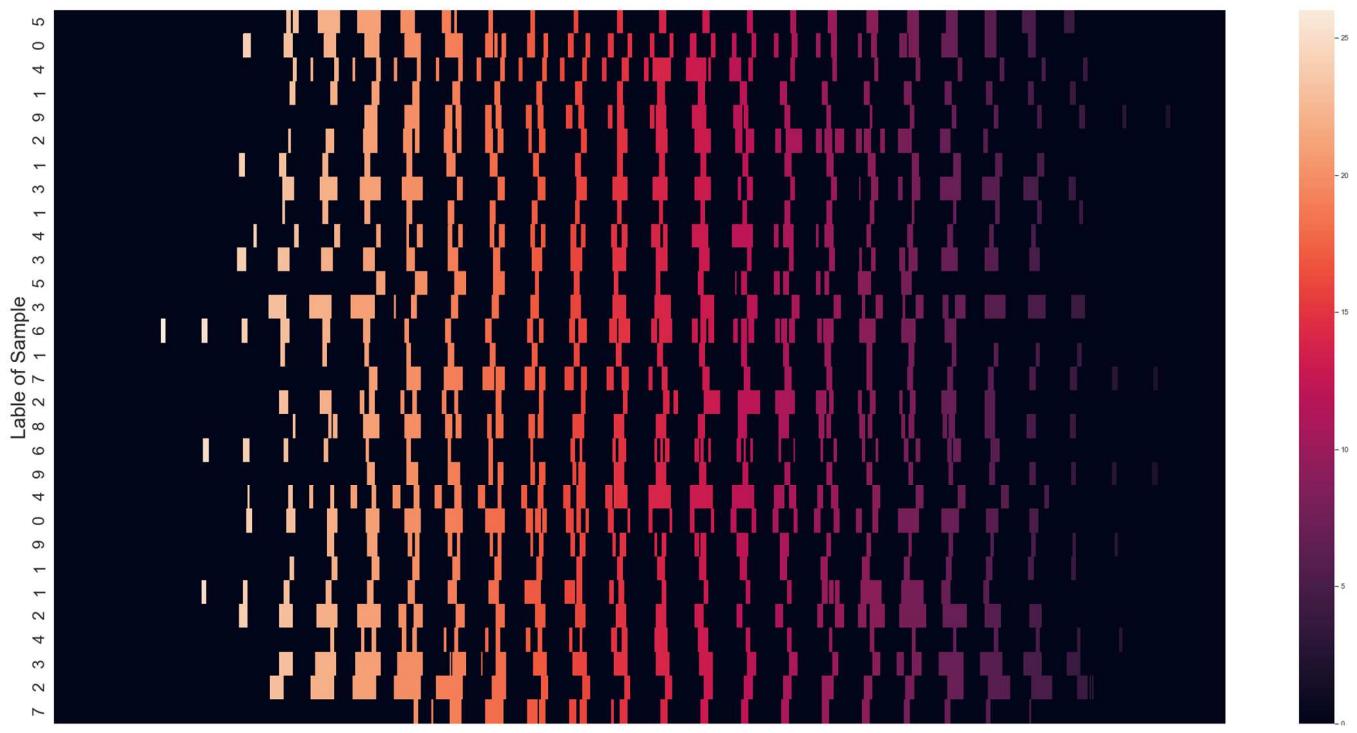


Fig.8 on 像素 Y 坐标的可视化 (前 30 张图片)

从左往右观察 Fig.8 可知, heatmap 中色块呈现出“色温保持不变→骤降为 0→回升至 1 个大于 0 略小一点的数值上→保持不变→……”的周期性变化。这与每 1 个 MNIST 图片矩阵中, 每一行 on 像素的 Y 坐标保持不变的规律相一致。因此, 可以大致确认没有出现计算错误。

2.8. 获得 on 像素 Y 坐标均值和方差

分别定义了 2 个函数 `y_mean_on()` 和 `y_std_on()`, 它们都接收训练集 or 测试集, 然后各自返回一个长度为样例数量的 array, 里面分别存储有各个样例 on 像素的 X 坐标均值和标准差。这 2 个函数的工作原理和 `x_mean_on()` 和 `x_std_on()` 类似, 因而在此不再赘述。

下面, 我们绘图展示 Train Set 中特征 `y_std` 和 `y_mean` 的分布和相关性 (详见 Fig.9)。不难看出, 这 2 个特征相关性较弱, 不存在较大的信息冗余, 因此有利于后续的模型训练。

In[277]:

```
fig = plt.figure(figsize=(40,20))
```

```

sns.heatmap(ytick_on(xx_train[:30]),
            annot=False,
            yticklabels=y_train[:30])

plt.yticks(fontsize=25)
plt.ylabel("Label of Sample", fontsize=30)
plt.xlabel("Index of Pixel", fontsize=30, y=0)

plt.show()

```

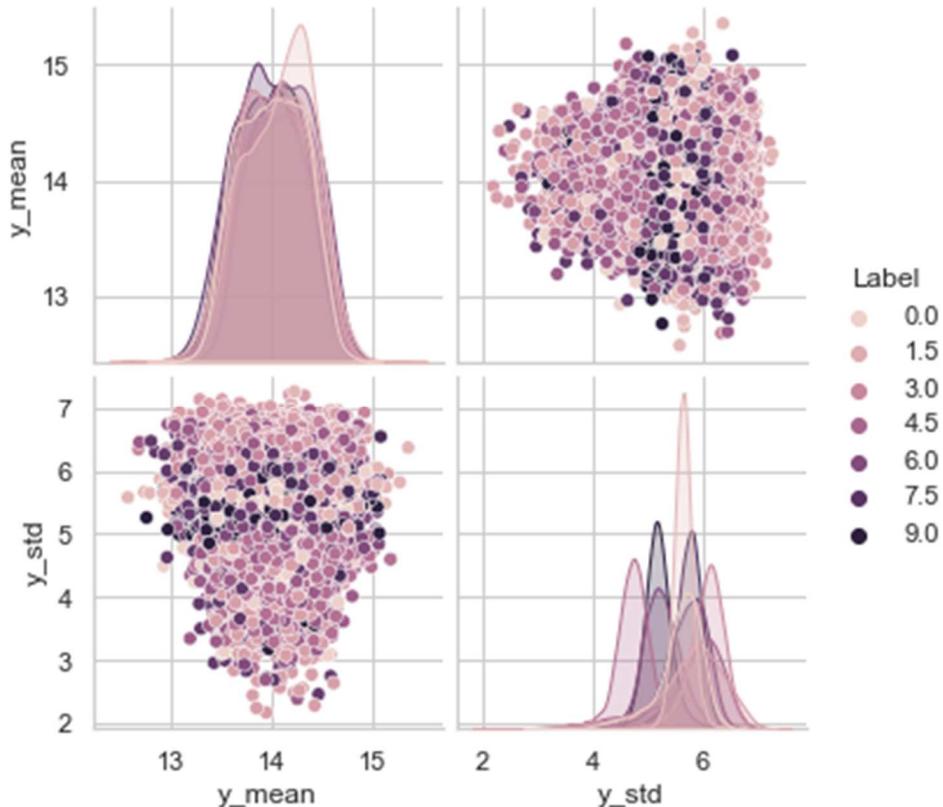


Fig.9 特征的分布与相关性

2.9. 获取 on 像素 X 坐标与 Y 坐标相关系数

定义了一个 `pearson()` 函数，该函数需调用前面的 `xtick_on()` 和 `ytick_on()` 函数。`pearson()` 接收训练集或测试集，并返回一个 1 维 array。array 的长度为 data 中图片的数量。

In[23]:

```
def pearson(data, m=28):
    n, _ = data.shape # n: data 内的样例个数

    x = xtick_on(data, m=m) # x: X 坐标
    y = ytick_on(data, m=m) # y: Y 坐标

    x[x == 0.0] = np.nan
    y[y == 0.0] = np.nan

    c_x = x - x_mean_on(data).reshape(n, 1) # 中心化
    c_y = y - y_mean_on(data).reshape(n, 1) # 这里 NaN 不会参与运算

    p1 = np.nansum(c_x * c_y, axis=1) # p1: Pearson 的分子出来了; p1 是一个1维数组
    p2 = (np.nansum(c_x ** 2, axis=1) * np.nansum(c_y ** 2, axis=1)) ** 0.5
    # p2: Pearson 的分母; p2 亦是一个1维数组

    return p1 / p2
```

下面，我们计算 Train Set 中不同阿拉伯数字所对应的图片的 on 像素的 X、Y 坐标相关系数值；然后，使用 boxplot 进行可视化（详见 Fig.10）。以此简单验证下是否存在计算错误。

In[287]:

```
temp_df = pd.DataFrame(np.c_[pearson(xx_train),
                             y_train],
                        columns=['Pearson', 'Label'])
temp_df.Label = np.int0(temp_df.Label)
temp_df
```

'''画图验证下 Pearson 系数有没有算错'''

In[288]:

```

sns.set()
plt.figure(figsize=(10,9))
sns.boxplot(x="Label", y="Pearson", data=temp_df, palette="rocket_r")
plt.show()

```

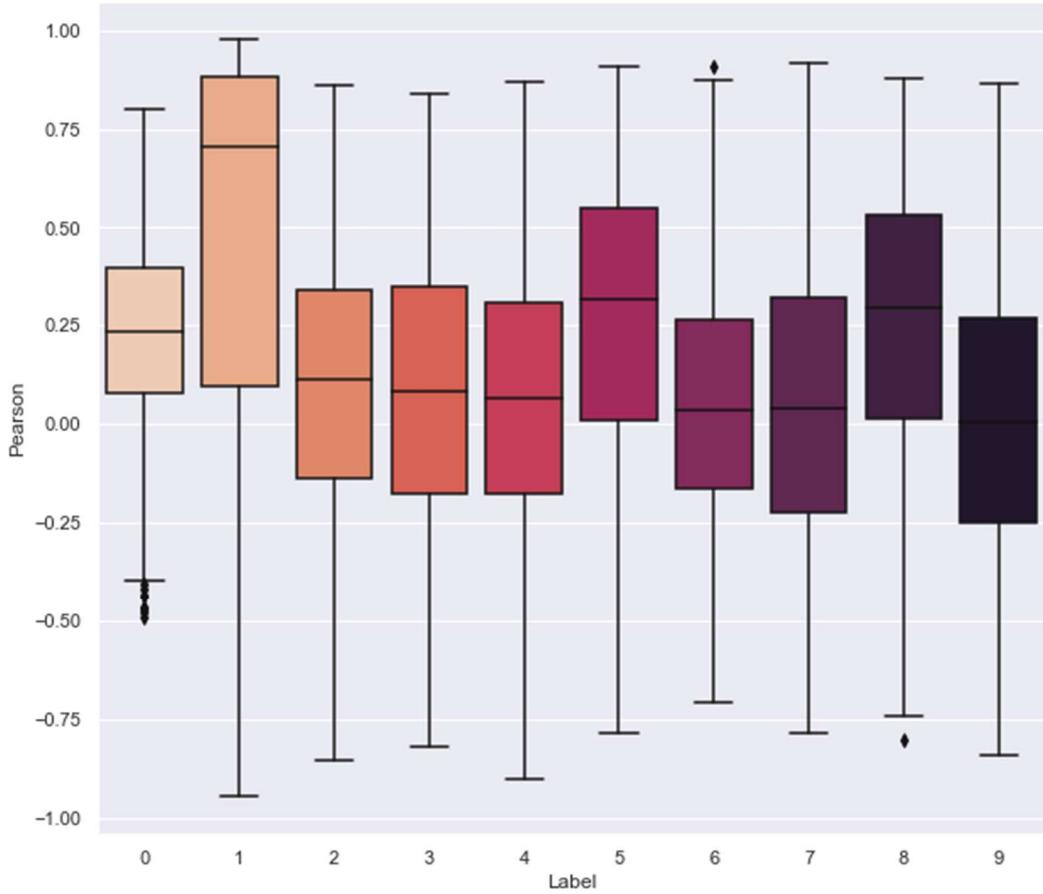


Fig.10 特征的分布与相关性

由 Fig.10 可知, 当 Label=1 时, 图片的 on 像素 X、Y 坐标相关系数多为正值 (我这里规定的 Y 坐标是越往上数值越大, 跟传统的做法不太一样)。结合前面 Fig.4 可知, MNIST 图片中的数字“1”大多向右上方倾斜, 因此, 可以大致确认没有出现计算错误。

2.10. on 像素 X 坐标平方乘以 Y 坐标的平均

分别定义了一个 `x2_y()` 和 `y2_x()` 函数, 分别用于计算 on 像素 X 坐标平方乘以 Y 坐标的平

均、on 像素 X 坐标乘以 Y 坐标平方的平均。它们都需调用前面的 `xtick_on()` 和 `ytick_on()` 函数。

In[24]:

```
'''计算 on 像素 X 坐标平方乘以 Y 坐标的平均'''
```

```
def x2_y(data, m=28):  
  
    x = xtick_on(data, m=m)  
    y = ytick_on(data, m=m)  
  
    x[x == 0.] = np.nan  
    y[y == 0.] = np.nan  
  
    x2 = x ** 2  
    result = np.nanmean(x2 * y, axis=1)  
    return result
```

In[25]:

```
'''计算 on 像素 Y 坐标平方乘以 X 坐标的平均'''
```

```
def y2_x(data, m=28):  
  
    x = xtick_on(data, m=m)  
    y = ytick_on(data, m=m)  
  
    x[x == 0.] = np.nan  
    y[y == 0.] = np.nan  
  
    y2 = y ** 2  
    result = np.nanmean(y2 * x, axis=1)  
    return result
```

2.11. 最小矩形框相关特征的提取

定义函数 square_()，用于计算 Train Set 中，每张图像的最大外接矩形和最小外接矩形的一些特征（包括最小外接矩形中心的 X 坐标、中心的 Y 坐标、宽度、高度以及倾斜角度）。

2.11.1. 导入 openCV2 包

```
# In[26]:
```

```
import cv2
```

2.11.2. 定义特征提取函数 square_()

```
# In[252]:
```

```
def square_(data, plot=False, m=28):  
  
    if np.ndim(data) == 1:  
        n = 1  
    else:  
        n, _ = data.shape  
  
    result_1 = []  
    result_2 = []  
    result_3 = []  
    box = None  
  
    for i_ in range(n):  
  
        thresh = xx_train[i_].reshape(m, m)  
  
        contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL,  
cv2.CHAIN_APPROX_SIMPLE)  
  
        for c in contours:  
            # 找到边界坐标
```

```

x, y, w, h = cv2.boundingRect(c) # 计算点集最外面的矩形边界

# 找面积最小的矩形
rect = cv2.minAreaRect(c)

if plot == True:
    # 得到最小矩形的坐标
    box = cv2.boxPoints(rect)
    # 标准化坐标到整数
    box = np.int0(box)

xy, hw, ratio = rect
x, y = xy
h, w = hw

rect = [x, y, h, w, ratio]

result_1.append((x, y, w, h))
result_2.append(rect)
result_3.append(box)

return result_1, np.vstack(result_2), result_3

```

2.11.3. 可视化最大外接矩形

In[141]:

```

temp, _, _ = square_(xx_train[:50], plot=True)
sns.set()
#sns.set_style("whitegrid")
fig = plt.figure(figsize=(10,10))
plt.subplots_adjust(wspace=0.001, hspace=0.001)
for i in range(49):
    x, y, w, h = temp[i]
    img = (X_train[i].reshape(28, 28) != 0) * 100
    img[x, y] = 255
    img[x + w, y] = 255

```

```
img[x, y + h] = 255  
img[x + w, y + h] = 255  
  
plt.subplot(7,7,i+1)  
plt.imshow(img )  
#plt.axis('off')  
plt.xticks([])  
plt.yticks([])  
#plt.suptitle("Panel(A) Human View",fontsize=18, y=0.92)  
plt.show()
```



Fig.11 最大外接矩形（前 49 张图片）

2.11.4. 可视化最小外接矩形

对前 49 张图片的最小外接矩形进行可视化如 Fig.12 所示。每张子图中，色温较高的 4 个像素点即为最小外接矩形的 4 个顶点。

In[142]:

```
sns.set()
#sns.set_style("whitegrid")
fig = plt.figure(figsize=(10,10))
plt.subplots_adjust(wspace=0.001, hspace=0.001)

for i in range(49):
    box = temp[i]
    img = (X_train[i].reshape(28, 28) != 0) * 100
    for j in range(4):
        x, y = tuple(box[j])
        img[x-1, y-1] = 255

    plt.subplot(7,7,i+1)

    plt.imshow(img ,
               #cmap='Greys',
               #interpolation='nearest'
               )
    #plt.axis('off')

    plt.xticks([])
    plt.yticks([])

#plt.suptitle("Panel(A) Human View", fontsize=18, y=0.92)

plt.show()
```



Fig.12 最小外接矩形（前 49 张图片）

2.12. on 像素边缘数的获取

分别定义 right_scan()、left_scan()、up_scan()、down_scan()这 4 个函数，用于计算：从右、从左、从上、从下扫描 on 像素的边缘数。这 4 个函数接收的参数都为图片矩阵；主要的返回值为

on 像素边缘数，且都为 float 类型。

2.12.1. 函数代码

由于上述 4 个函数的工作原理类似，因而，在此我们只列示 up_scan()的 Python 代码。

In[182]:

```
def up_scan(img):
    img = (img != 0)
    m, _ = img.shape

    # 图片矩阵整体上移，最下侧用0填充
    img_move_up = np.vstack([img[1:, :], np.zeros((1, m))])
    ans = ((img + img_move_up) == 1) * (img==0)

    num = np.sum(ans != 0)
    return num, ans
```

2.12.2. 绘图观察扫描效果

In[212]:

'''画图检查效果-1'''

```
image = xx_train[1].reshape(28, 28)
num, flag = up_scan(image)
image[image != 0] = 150
image = image + (flag != 0) * 255
```

In[213]:

```
fig = plt.figure(figsize=(24,20))
sns.heatmap(image, annot=True)
# plt.axis('off')
plt.xticks([])
```

```
plt.yticks([])
plt.show()
```

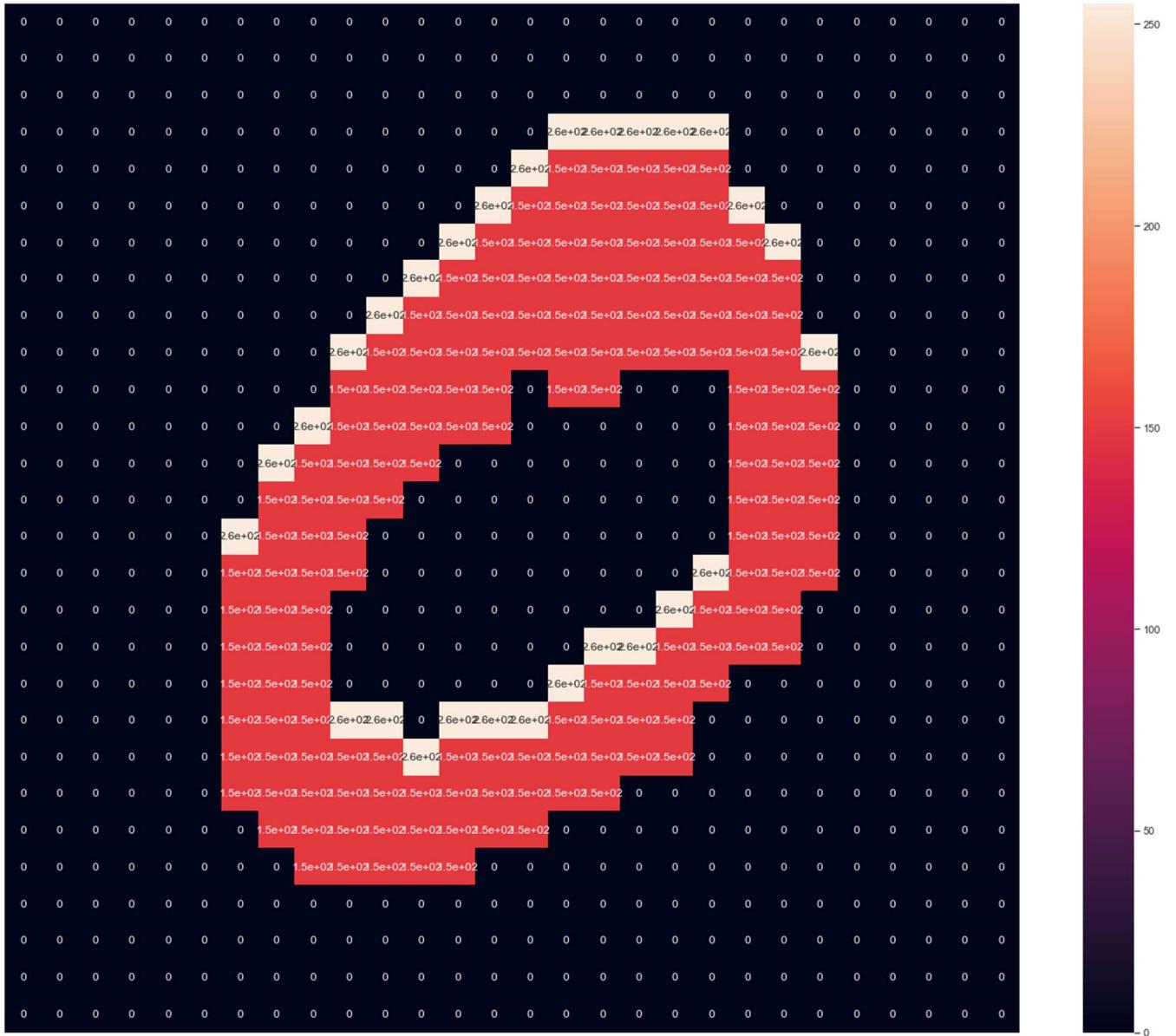


Fig.13 从上扫描 on 像素边缘数 (第 2 张图片)

Fig.13 中，色温最高的像素点即为扫描得到的 on 像素边缘数。不难看到，虽然存在一定的误差，但扫描效果尚可。

```
# In[215]:
```

```
'''画图检查效果-2'''
```

```
fig = plt.figure(figsize=(20, 18))

plt.subplots_adjust(wspace=0.01, hspace=0.01)

for i in range(16):

    image = xx_train[i].reshape(28, 28)
    num, flag = up_scan(image)
    image[image != 0] = 150
    image = image + (flag != 0) * 255

    plt.subplot(4, 4, i + 1)

    sns.heatmap(image, annot=False)

    #plt.axis('off')
    plt.xticks([])
    plt.yticks([])
    plt.grid()

plt.show()
```

Fig.14 对 Train Set 中前 49 张图片的扫描结果进行了可视化。同样地，色温最高的像素点即为扫描得到的 on 像素边缘数。

不难看到，在不同 Label 的图片中，扫描结果虽然都存在着一定的误差，但总体而言扫描效果尚可。



Fig.13 从上扫描 on 像素边缘数 (前 49 张图片)

2.13. 特征提取

定义一个 extract() 函数，extract() 集成了前面所有的单个特征获取函数，用于对数据集中全部样例的特征进行提取。若接收的 data 为一个含有 60000 条样例的数据集，则返回一个 60000×17 的矩阵，每一列对应一个特征。

In[295]:

```
def extract(data, m=28):
```

```
n, _ = data.shape
```

'''首先，是一些不用循环就能获得的特征的提取'''

```
on_num_ = on_num(data) # ON 像素的数量
```

```
x_mean_on_ = x_mean_on(data) # X 坐标均值
```

```
x_std_on_ = x_std_on(data) # X 坐标方差
```

```
y_mean_on_ = y_mean_on(data) # Y 坐标均值
```

```
y_std_on_ = y_std_on(data) # Y 坐标方差
```

```
pearson_ = pearson(data) # on 像素X坐标与Y坐标相关系数
```

```
x2_y_ = x2_y(data) # on 像素X坐标平方乘以Y坐标的平均
```

```
y2_x_ = y2_x(data) # on 像素X坐标乘以Y坐标平方的平均
```

```
featrue_class_1 = np.c_[on_num_, x_mean_on_, x_std_on_, y_mean_on_,  
y_std_on_,  
pearson_, x2_y_, y2_x_]
```

'''其次，是一些需要循环才能获得的特征的提取'''

```
# 从左往右依次为：矩形框的 中心(x, y)、高、宽、倾斜角度，共计5个特征  
_, min_rectangle, _ = square_(data)
```

```
featrue_class_3 = [[ right_scan(data[i_].reshape(m, m))[0],  
left_scan(data[i_].reshape(m, m))[0],  
up_scan(data[i_].reshape(m, m))[0],
```

```

        down_scan(data[i_].reshape(m, m))[0] ]
    for i_ in range(n)]
featrue_class_3 = np.vstack(featrue_class_3)

return np.hstack([featrue_class_1, min_rectangle, featrue_class_3])

```

3. 划分测试集与训练集

3.1. 样本划分函数

考虑到时间有限，我们只选取 MNIST 数据集的 Train Set 中的前 12000 张图片用于模型训练与测试。下面，我们定义了 1 个训练集和测试集划分函数，按照 $\frac{8}{2}$ 的比例，将这 12000 张图片划分为训练集和测试集。

In[390]:

```

def split_train_test(data,test_ratio):
    # 设置随机数种子，保证每次生成的结果都是一样的

    np.random.seed(42)

    # permutation 随机生成 0-Len(data) 随机序列
    shuffled_indices = np.random.permutation(len(data))
    # test_ratio 为测试集所占的百分比
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]

    return data.iloc[train_indices],data.iloc[test_indices]

```

3.2. 划分

In[391]:

```

train_set,test_set = split_train_test(train, 0.20)
train = np.array(train_set)[:, :-1]
test = np.array(test_set)[:, :-1]
y_train = np.array(train_set)[:, -1]
y_test = np.array(test_set)[:, -1]

```

4. 特征描述与分析

4.1. 训练集特征的描述性统计

Table.2 特征描述性统计

VARIABLES	(1) N	(2) mean	(3) std	(4) min	(5) max	(6) skewness	(7) kurtosis
ON	9,000	133.8	34.96	40	279	0.254	3.011
x_mean	9,000	15.01	0.318	14.10	16.08	-0.0296	2.136
x_std	9,000	3.883	1.053	0.660	6.711	-0.402	3.122
y_mean	9,000	14.02	0.332	12.85	15.08	-0.0369	2.383
y_std	9,000	5.498	0.563	2.450	7.214	-0.573	3.785
Pearson	9,000	0.185	0.371	-0.940	0.975	-0.0841	2.415
x2_y	9,000	3,503	313.7	2,629	4,710	0.238	2.797
y2_x	9,000	3,516	291.6	2,550	4,551	0.0422	2.836
x	9,000	13.84	1.063	4	22.50	-0.409	12.08
y	9,000	13.97	1.329	1.500	18.50	-1.336	13.72
H	9,000	15.09	4.948	0	26.87	-0.783	2.670
W	9,000	17.22	4.871	0	26.87	-0.704	3.165
Angel	9,000	55.66	31.84	-90	180	-0.188	1.681
R	9,000	25.61	4.545	12	44	0.367	2.595
Up	9,000	25.45	8.569	4	53	-0.230	2.881

4.2. 特征间相关性

使用 heatmap 对 17 个不同特征之间的 Pearson 相关系数进行可视化（详见 Fig.14）。可以看出，特征 L 和 R、Up 和 Down 之间相关系数接近于 1.0，说明存在信息冗余。

MNIST Image Recognition Based in Xgboost algorithm and Features extraction

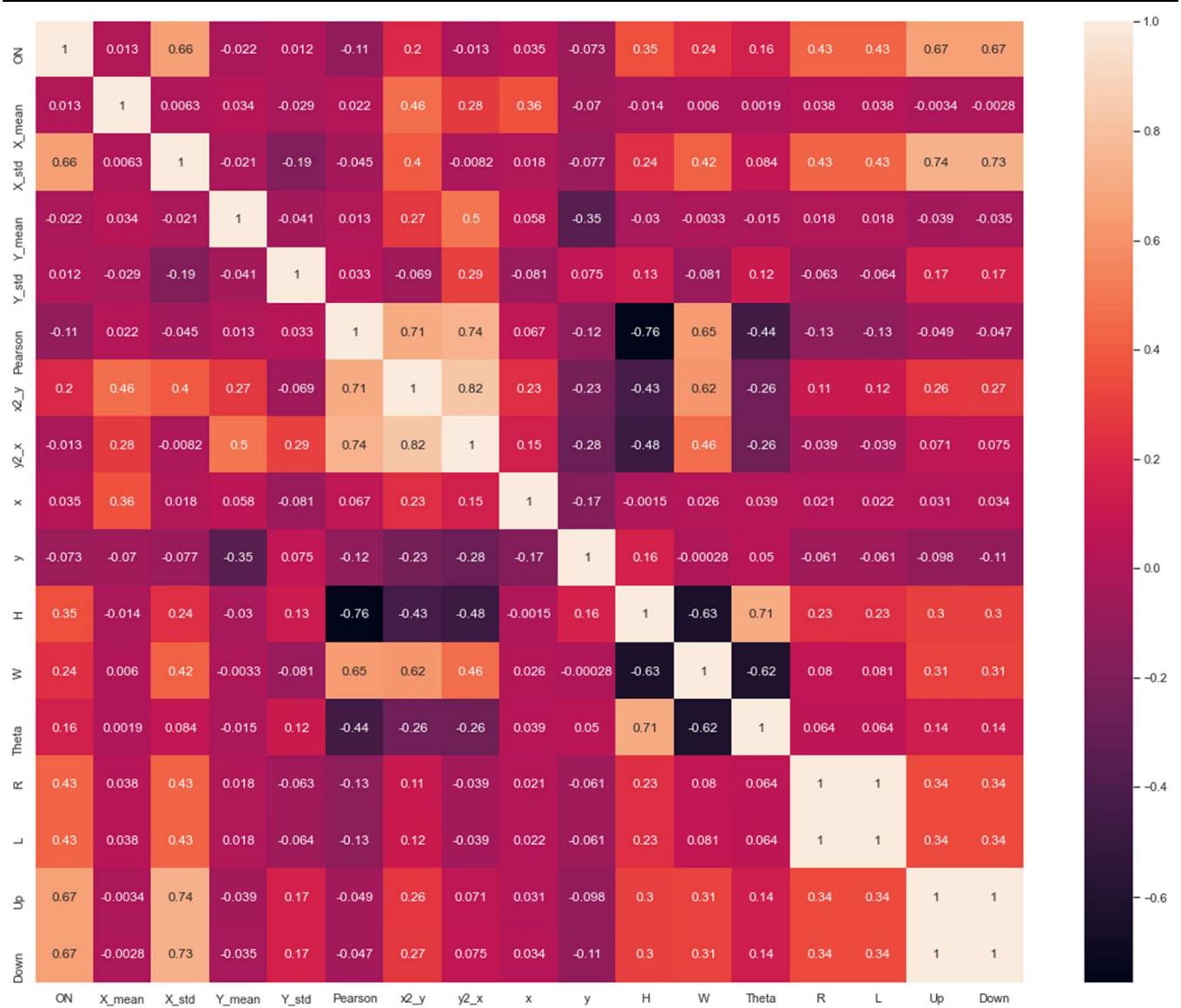


Fig.14 特征间相关系数矩阵 (训练集)

用散点图观察 L 和 R、Up 和 Down 之间的关系，发现这 2 对特征几乎相等（详见 Fig.15）。

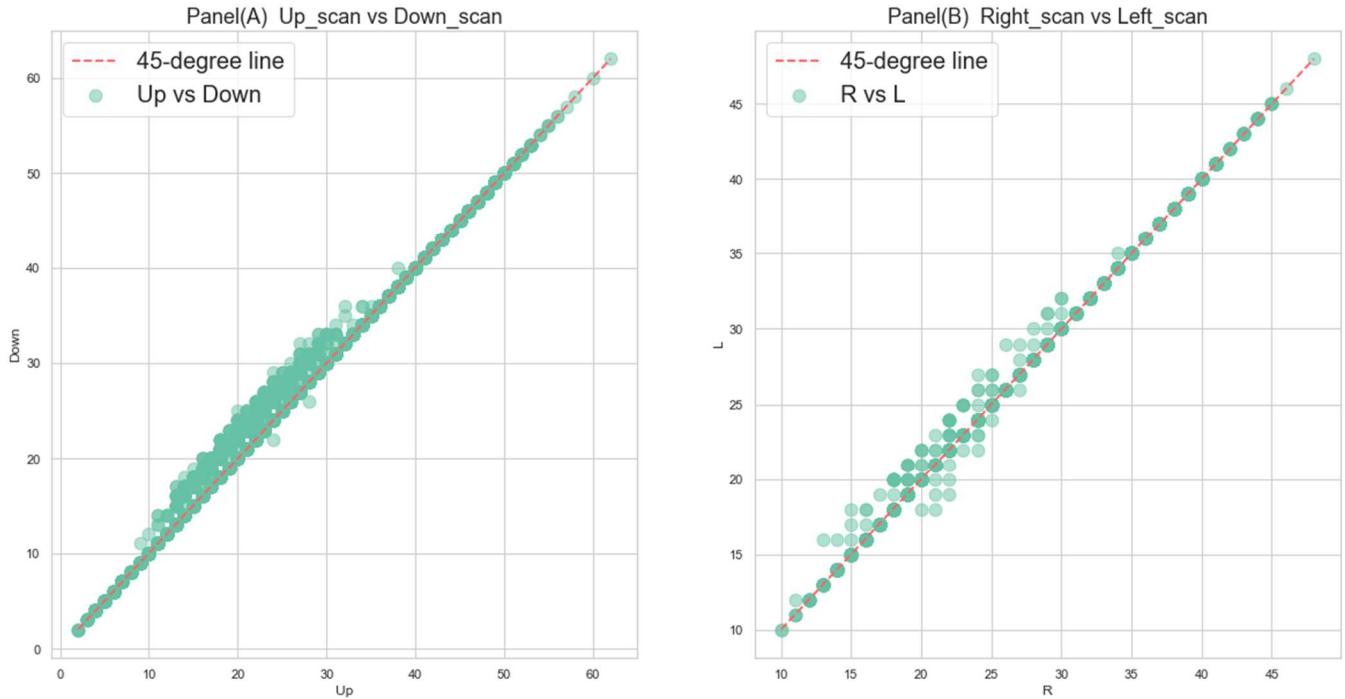


Fig.15 L 和 R、Up 和 Down 之间的数量关系（训练集）

4.3. 特征剔除

为避免过多冗余的信息影响模型训练，在 L 和 R、Up 和 Down 之间，我们只保留 R 和 Up 这 2 个特征。

5. 模型训练与逐步调参

下面，我们用 Xgboost 方法实现 MNIST 手写数字图片的分类。

5.1. 设置初始模型

```
# In[331]:
```

```
import xgboost as xgb
```

```
# In[392]:
```

```
param_dist = {'objective': 'multi:softmax',
              'n_estimators': 1000,
              'num_class': 10, # 分成的类数
              'eta': 0.01, # 学习率
              'alpha': 1, # 惩罚项参数: γ
              'lambda': 0.001, # 惩罚项参数: λ
              'use_label_encoder': False,
              #'metric': 'multiclass',
              'eval_metric': 'mlogloss'
              # 'metric': multiclass
              # 'subsample': 0.9
              }
# 建立一个初步的模型, 看一下效果怎么样。
xgb_model = xgb.XGBClassifier(**param_dist)
```

5.2. 训练初始模型，并查看准确率

```
# In[393]:
```

```
xgb_model.fit(train, y_train) # 拟合
xgb_result = xgb_model.predict(test, iteration_range = (0, 1000) )
np.mean(xgb_result == y_test)
```

经计算，初始模型的准确率为 0.8473333333333334。

5.3. 调整惩罚项 γ

使用 5 折交叉验证来调节模型的惩罚项参数 γ ，并以交叉验证误差作为最优模型的评判标准。
设置 50 个待选参数，范围从 $10^{-8} \sim 10^0$ 。

```
# In[396]:
```

'''待选参数'''

```
alphas = np.logspace(-8, 0, 50, base = 10)

y_train = np.int64(y_train)
y_test = np.int64(y_test)

dtrain = xgb.DMatrix(train, label=y_train)
dtest = xgb.DMatrix(test, label=y_test)

result_ls = []
for i in range(len(alphas)):

    param_dict = {'objective': 'multi:softmax',
                  'num_class': 10,
                  'eta': 0.01,
                  'alpha': alphas[i],
                  'lambda': 0.001,
                  'eval_metric': 'mlogloss'}
    xgb_cv = xgb.cv(param_dict, dtrain,
                     num_boost_round=1000,
                     nfold=5, stratified=True,
                     early_stopping_rounds=10, metrics='merror')

    temp = np.array(xgb_cv)
    result_ls.append(temp[-1])
    param_dic
    print(">>>> 第{}个参数测试完毕!".format(i + 1))
```

```

result1 = np.vstack(result_ls)
result1.shape

'''保存调参结果'''

error_mean = result1[:, 0]
error_std = result1[:, 1]
CV_std = result1[:, 3]
CV_mean = result1[:, 2]

'''绘图，展示调参过程'''

sns.set_style("whitegrid")
plt.figure(figsize=(18, 7))
plt.subplot(1, 2, 1)
plt.errorbar(alphas, CV_mean      #x, y 数据, 一一对应
             , yerr=CV_std           #y 误差范围
             , fmt="o"                #数据点标记
             , ms=5                  #数据点大小
             , mfc="red"              #数据点颜色
             , mec="r"                #数据点边缘颜色
             , ecolor="grey"           #误差棒颜色
             , elinewidth=3            #误差棒线宽
             , capsizes=4              #误差棒边界线长度
             , capthick=1              #误差棒边界线厚度
             )
plt.semilogx()
plt.axvline(alphas[np.argmin(CV_mean)], color = 'black', ls="--", lw=1.2)
plt.legend(['The Best Model', 'CV Mean Error'], fontsize=16)
plt.xlabel('Alpha', fontsize = 15)
plt.ylabel('Mean Error', fontsize = 15)
plt.title("Panel(A) Cross Validation", fontsize = 17)
plt.subplot(1, 2, 2)
plt.errorbar(alphas, error_mean    #x, y 数据, 一一对应
             , yerr=error_std          #y 误差范围
             , fmt="o"                 #数据点标记
             )

```

```

        , ms=5                                #数据点大小
        , mfc="red"                            #数据点颜色
        , mec="r"                               #数据点边缘颜色
        , ecolor="grey"                          #误差棒颜色
        , elinewidth=3                           #误差棒线宽
        , capsize=4                             #误差棒边界线长度
        , capthick=1
    )

plt.semilogx()
plt.axvline(alphas[np.argmin(CV_mean)], color = 'black', ls="--", lw=1.2)
plt.legend(['The Best Model', 'Train Mean Error'], fontsize=16)
plt.xlabel('Alpha', fontsize = 15)
plt.ylabel('Mean Error', fontsize = 15)
plt.title("Panel(B) Trainning", fontsize = 17)

plt.show()

```

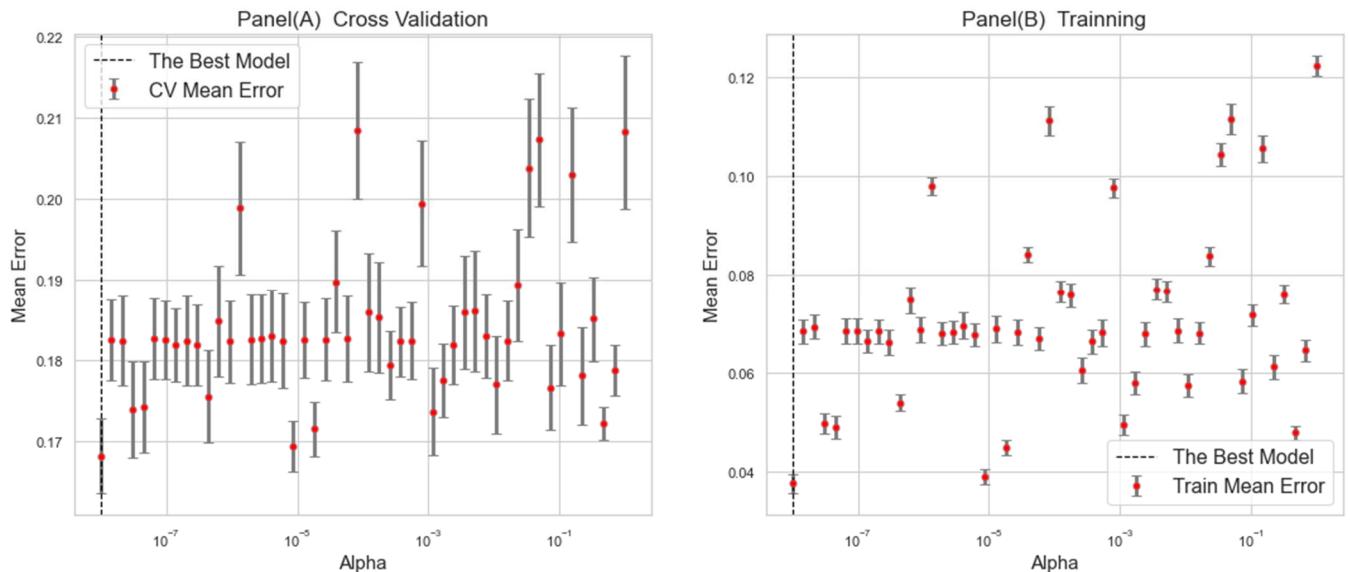


Fig.16 不同的参数 α (γ) 下，交叉验证误差与训练误差

由 Fig.16, 我们选择 $\gamma = 10^{-8}$ 作为最优参数。仅计算, 设置最优参数后, 模型在 Test Set 上的准确率上升至 0.8636666666666667。

5.4. 调整学习率

使用 5 折交叉验证来调节模型的学习率 η ，并以交叉验证误差作为最优模型的评判标准。设置 30 个待选参数，范围从 $10^{-4} \sim 10^0$ 。代码与上面的类似，故不再赘述。

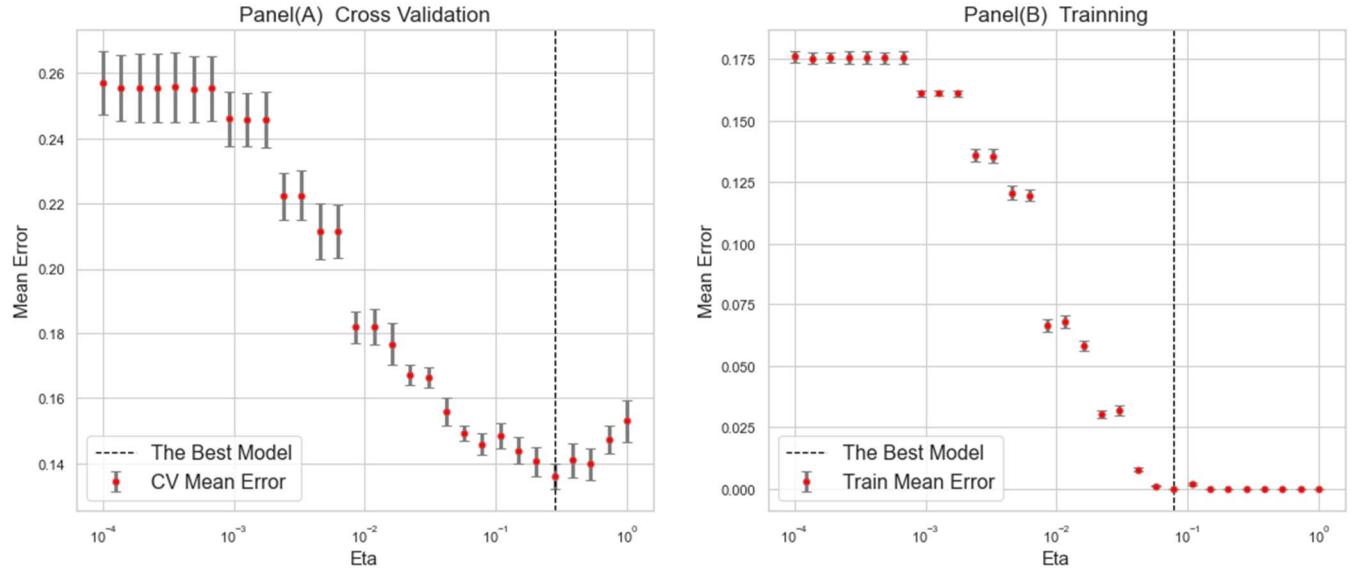


Fig.17 不同的参数 Eta (学习率) 下，交叉验证误差与训练误差

由 Fig.17，我们选择 $\eta = 0.0035564803062231283$ 作为最优参数。仅计算，设置最优的 η 后，模型在 Test Set 上的准确率上升至 0.867。

5.5. 调整参数 Lambda

接下来，使用 5 折交叉验证来调节模型的另一个惩罚项 λ ，并以交叉验证误差作为最优模型的评判标准。设置 50 个待选参数，范围从 $10^{-5} \sim 10^0$ 。代码与 5.2. 中类似，故不再赘述。

由 Fig.18，我们选择 $\lambda = 0.0035564803062231283$ 作为最优参数。不幸的是，仅计算，设置最优的后，模型在 Test Set 上的准确率下降为 0.8636666666666667。但下降幅度较小，所以可能是

测试集本身的原因。

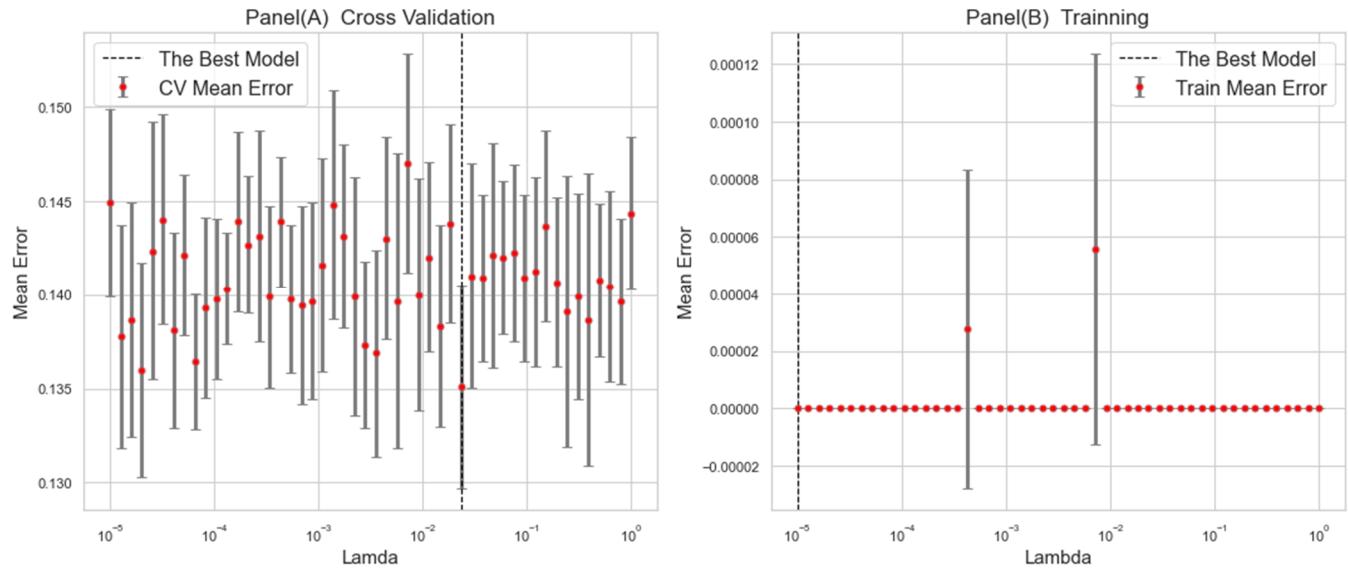


Fig.18 不同的参数 Lambda 下，交叉验证误差与训练误差

5.6. 绘制最优模型的训练曲线

下面，我们绘制 Xgboost 在最优参数组合下的训练曲线（基于 10 折交叉验证）。

In[411]:

```
xgb_cv = xgb.cv(param_dict, dtrain,
                  num_boost_round=1000,
                  nfold=10, stratified=True,
                  early_stopping_rounds=15, metrics='merror')
result4 = np.array(xgb_cv)
```

result4.shape

```
CV_mean = result4[:, 2]
CV_std = result4[:, 3]
error_mean = result4[:, 0]
error_std = result4[:, 1]

htick_ls = range(1, result4.shape[0] + 1)

plt.figure(figsize=(15, 8))

plt.plot(htick_ls, CV_mean,
          color="#FF6464", marker='o',
          markersize=5, label='Cross Validation Error')
plt.fill_between(htick_ls,
                 CV_mean + CV_std,
                 CV_mean - CV_std,
                 alpha=0.5, color="#FF6464")
plt.plot(htick_ls, error_mean,
          color='royalblue', linestyle='--',
          marker='s', markersize=5,
          label='Training Error')
plt.fill_between(htick_ls,
                 error_mean + error_std,
                 error_mean - error_std,
                 alpha=0.5, color='royalblue')

#plt.grid()

plt.xlabel('Number of training rounds', fontsize=16)
plt.ylabel('Error', fontsize=16)
plt.legend(loc='upper right', fontsize=18)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
#plt.ylim([0.6, 1.0])
plt.tight_layout()
plt.show()
```

```
plt.show()
```

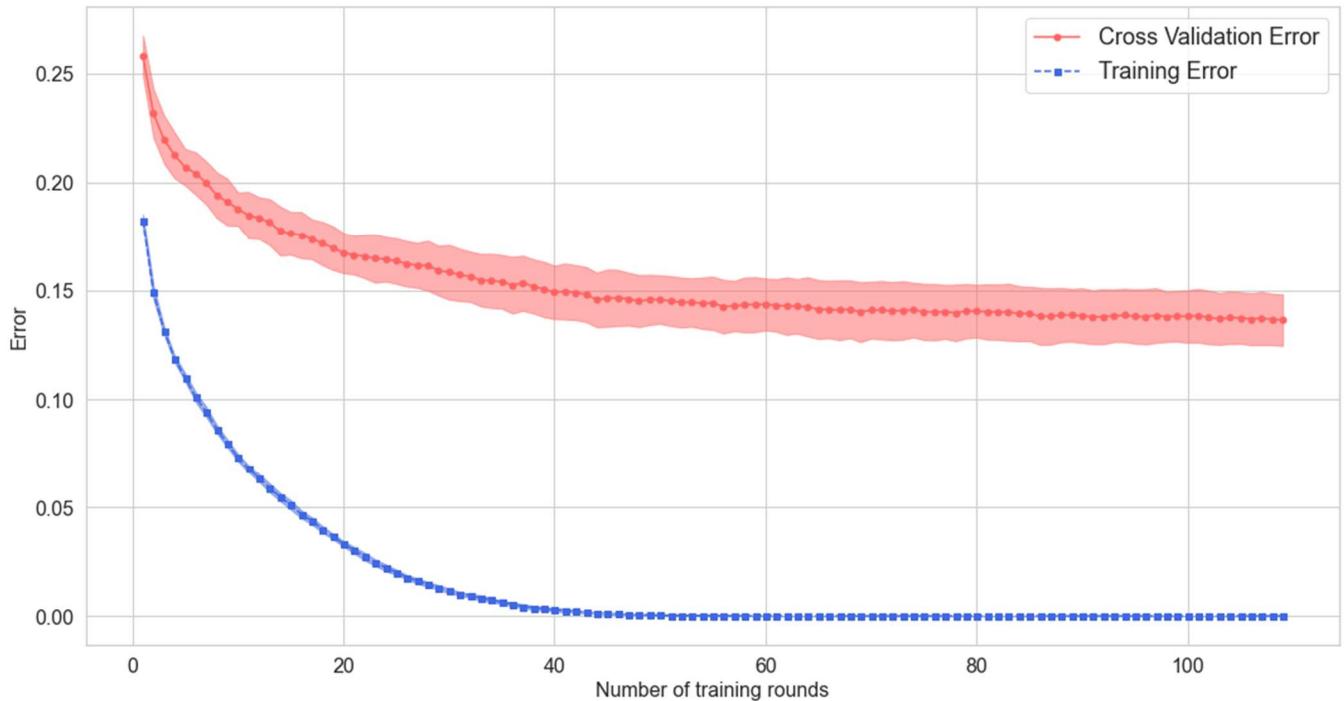


Fig.19 最优 Xgboost 的训练曲线 (CV=10, 阴影为误差)

由 Fig.19 可知，随着训练轮次的增加，训练集误差逐渐减小，直至降低为 0；交叉验证集误差也逐渐降低，最后稳定在 0.14 左右。另外，红线和蓝线之间的 Gap 较大，在模型稳定后依然有将近 0.15，这说明模型的方差较大。也有可能是由于训练集中的样例数量太少，以至于出现了“过拟合”现象。

5.7. 绘制混淆矩阵

下面，我们使用混淆矩阵来查看测试集中，不同手写数字对应的图片被准确或错误归类的情况（详见 Fig.20）。

```
# In[416]:
```

```
from sklearn.metrics import confusion_matrix
```

'''获取混淆矩阵'''

```
cm = confusion_matrix(y_true, y_pred)

plt.figure(figsize=(12, 8), dpi=100)
sns.heatmap(cm, cmap='YlGnBu', annot=True, fmt='.20g')
plt.ylabel('Actual label')
plt.xlabel('Predict label')
plt.show()
```

由 Fig.20 不难看出，测试集中的数字“0”、“1”几乎被完全正确预测。而数字“2”、“3”、“5”被错误预测的样例较多，后续可能需要进一步提取更多特征用于模型训练与预测。

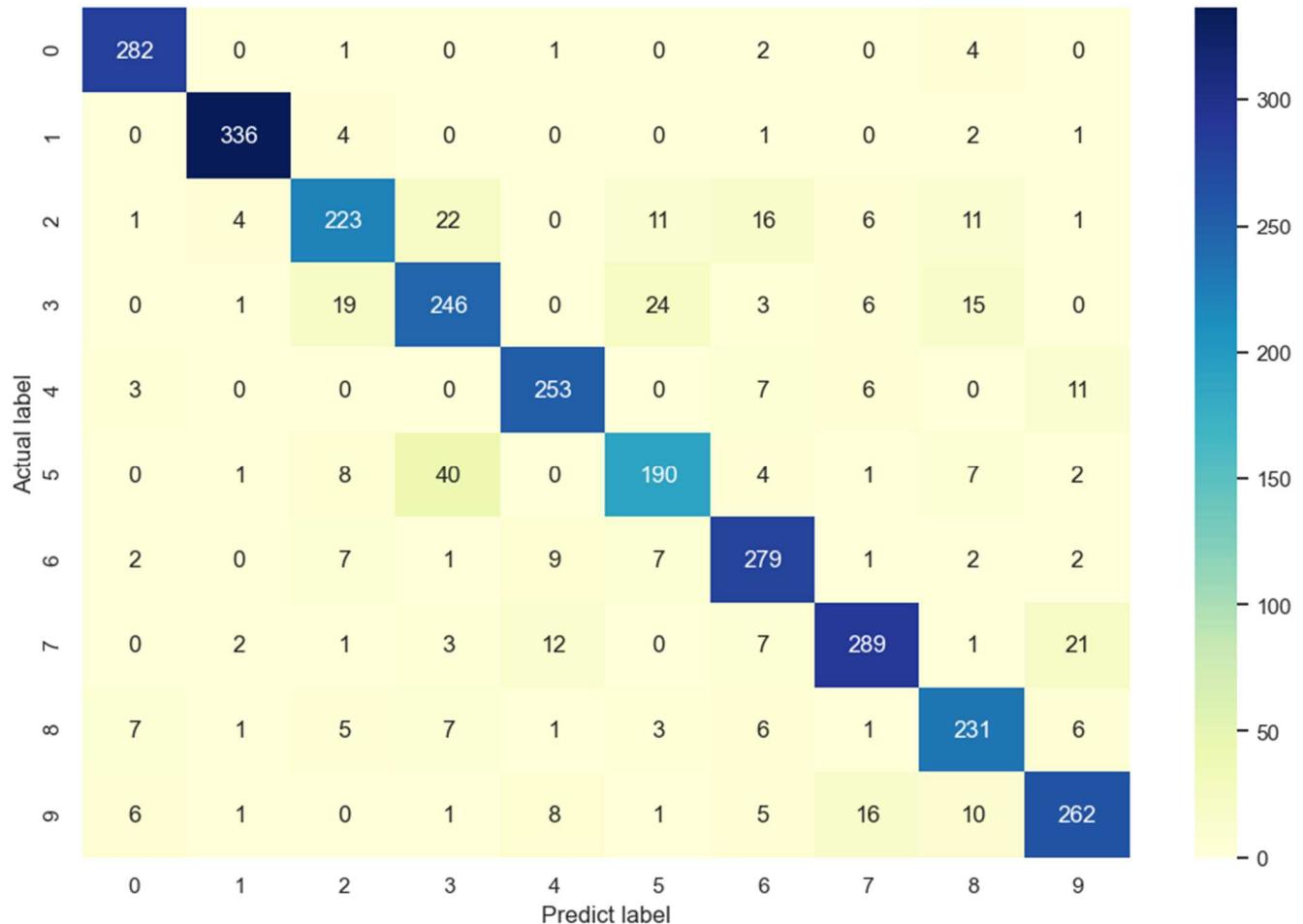


Fig.20 最优 Xgboost 在测试集上的混淆矩阵

6.总结

本报告基于 MNIST 手写数字图片数据集，调用 numpy、OpenCV2 等库提取相关特征，进而训练 Xgboost 手写数字识别模型。经过调参后，最优模型在测试集上的准确率将近有 88%。

纵观整个使用流程，最复杂的部分莫过于图片特征的提取。这恐怕也是卷积神经网络（CNN）等可以自动提取特征的机器学习算法在图像识别领域被大量运用的原因。

本实验还有以下可改进之处：(1) 对 on 像素点边缘数量的监测不够准确，导致从右侧扫描与从左侧扫描的边缘数量几乎相等（从上扫描和从下扫描亦存在类似的问题），从而使得可用特征从 17 个下降至 15 个，影响了模型对图片特征的学习。(2) 调参采取的是逐步调参的办法，可能陷入局部最优解。(3) 由于时间原因，没有对整个 MNIST Train Set 进行训练。这造成用于训练模型的训练集容量太小，可能导致模型的泛化能力不足。