ĐẠI HỌC QUỐC GIA TP.HCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

MÔN HỌC: THỐNG KÊ MÁY TÍNH VÀ ỨNG DỤNG

Báo cáo Project 1:

Logistic Regression Model For MNIST Dataset

Nhóm sinh viên

Nguyễn Huy Hải – 18120023

Phạm Công Minh – 18120058

Nguyễn Thanh Tùng – 18120104

Nguyễn Thị Hồng Nhung – 18120498

Mục lục

1 Phân công công việc và mức độ hoàn thành của mỗi thành viên nhóm **3**

2 Các thư viện được sử dụng trong bài **4**

3 Tìm hiểu về tập dữ liệu MNIST **5**
3.1 Tổng quan về tập dữ liệu 5
3.2 Tải và quan sát dữ liệu 5
3.2.1 Tải dữ liệu 5
3.2.2 Hiển thị và quan sát dữ liệu 6

4 Tiền xử lý dữ liệu **7**
4.1 Xoay ảnh 7
4.2 Lọc giá trị nhiễu 8
4.2.1 Xóa biên kết hợp canh giữa cho hình 8
4.2.2 Xóa tất cả dòng cột chứa toàn giá trị 0 9
4.3 Hiện tượng đa cộng tuyến (Multicollinearity) và cách giải quyết 10
4.3.1 Đa cộng tuyến là gì? 10
4.3.2 Xác định dấu hiệu xảy ra hiện tưởng đa cộng tuyến 10
4.3.3 Hướng giải quyết hiện tượng đa cộng tuyến 12
4.3.4 Principal Component Analysis (PCA) và ứng dụng vào mô hình 13
4.4 Polynomial Features 14
4.4.1 Polynomial Features là gì? 14
4.4.2 Ứng dụng Polynomial Features vào tập dữ liệu . . . 15
4.5 Standard Scaler 16

5 Xây dựng mô hình Logistic Regression **16**
5.1 Nhắc lại đầu vào và đầu ra của mô hình 16
5.2 Cấu trúc và cách thiết kế của mô hình Logistic Regression 17
5.2.1 Tổng quan về mô hình 17
5.2.2 Sigmoid Function 17
5.2.3 Cross-entropy Loss Function 18
5.2.4 Gradient Descent 19
5.2.5 Regularization 21
5.2.6 Multinomial Logistic Regression 21
5.3 Hàm LogisticRegression() trong thư viện sklearn và ứng dụng vào bài toán 23
5.3.1 Các tham số quan trọng 23

5.3.2 Lựa chọn giá trị các tham số phù hợp cho bài toán 25

6 Các bước thử nghiệm để tìm mô hình tốt nhất và đánh giá 25
6.1 Các bước thử nghiệm 25
6.1.1 Thử nghiệm 1: Huấn luyện mô hình không xử lí dữ liệu 26
6.1.2 Thử nghiệm 2: Huấn luyện mô hình sau khi canh giữa ảnh và cắt biên 27
6.1.3 Thử nghiệm 3: Huấn luyện mô hình sau khi loại bỏ các dòng, cột bằng 0 và chỉnh kích thước lên lại (28, 28) 27
6.1.4 Thử nghiệm 4: Huấn luyện mô hình sau khi PCA . 28
6.1.5 Thử nghiệm 5: Kết hợp thử nghiệm 3 với 4 28
6.1.6 Thử nghiệm 6: Huấn luyện mô hình như ở thử nghiệm 4 sử dụng ảnh đã cắt hết dòng, cột bằng 0 nhưng sử dụng thêm Polynomial Features 29
6.1.7 Thử nghiệm 7: Sử dụng Data Augmentation kết hợp thử nghiệm 6 29
6.2 Tổng kết mô hình và xây dựng Pipeline hoàn chỉnh của mô hình 30
6.3 Đánh giá mức độ hoàn thiện và độ chính xác của mô hình 31

7 Thử nghiệm thực tế 35
7.1 Giới thiệu phần mềm và cách sử dụng 35
7.2 Các bước xử lý cụ thể của chương trình 38
7.2.1 Class Paint() 38
7.2.2 Class ProcessImage() 39

8 Tài liệu tham khảo. 42

1 Phân công công việc và mức độ hoàn thành của mỗi thành viên nhóm

- **Nguyễn Huy Hải - 18120023**
  - Tổng hợp các file và viết báo cáo.

  Đánh giá: 100%
- **Phạm Công Minh - 18120058**
  - Thực hiện các bước tiền xử lý.

  Đánh giá: 100%
- **Nguyễn Thanh Tùng - 18120104**
  - Xây dựng phần mềm để kiểm thử mô hình.

  Đánh giá: 100%
- **Nguyễn Thị Hồng Nhung - 18120498**
  - Xây dựng mô hình *Logistic Regression* và làm *slide*.

  Đánh giá: 100%

2 Các thư viện được sử dụng trong bài

- numpy (version 1.19.2)
- cv2 (version 4.5.2 - Tải thư viện openCV)
- matplotlib.pyplot (version 3.3.2)
- pandas (version 1.1.3)
- time
- tkinter (Tải thư viện)
- PIL (import Image, ImageDraw)
- pickle
- imutils
- os
- Các thư viện và hàm trong Sklearn (version 0.24.2):
  - datasets (import fetch_openml)
  - linear_model (import LogisticRegression)
  - model_selection (import train_test_split)
  - preprocessing (import StandardScaler, PolynomialFeatures)
  - decomposition (import PCA)
  - pipeline (import Pipeline)
  - metrics (import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score, ConfusionMatrixDisplay)

3 Tìm hiểu về tập dữ liệu MNIST

3.1 Tổng quan về tập dữ liệu

- MNIST là tập dữ liệu về các chữ số viết tay.
- Tập dữ liệu được chia làm 2 phần: *training* set và *test set*.
- *Training set* gồm có 60000 mẫu. *Test set* gồm có 10000 mẫu.
- Mỗi mẫu có đặc điểm như sau:
  - Mỗi mẫu gồm có một bức ảnh và nhãn tương ứng đi kèm.
  - Bức ảnh trong mẫu là ảnh đen trắng (*grayscale*).
  - Bức ảnh có kích thước 28 × 28 điểm ảnh (*pixel*).
  - Mỗi điểm ảnh là một con số có giá trị 0 đến 255.
  - Chính giữa bức ảnh sẽ có hình ảnh về một chữ số nào đó trong các chữ số từ 0 đến 9.
  - Nhãn của bức ảnh sẽ có giá trị từ 0 đến 9, giá trị đó sẽ tương ứng với chữ số viết tay xuất hiện trong bức ảnh.

3.2 Tải và quan sát dữ liệu

3.2.1 Tải dữ liệu

Ta dùng hàm **fetch_openml()** từ thư viện **sklearn.datasets** để tải bộ dữ liệu **mnist_784**.

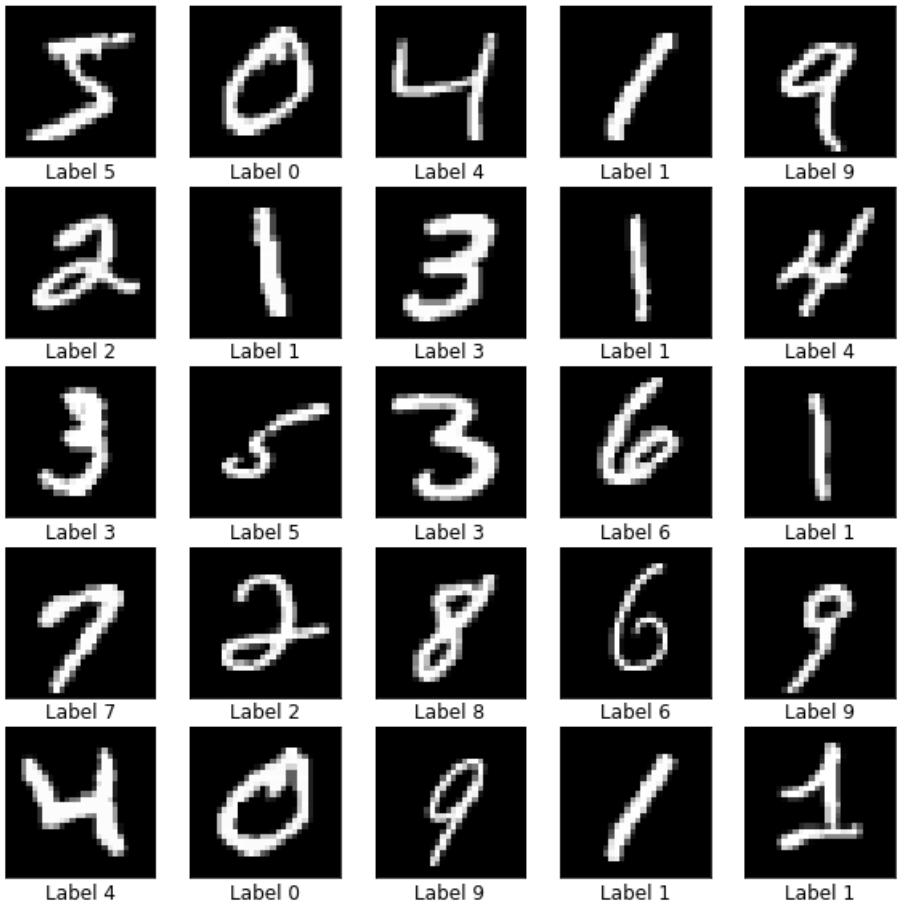
```
X, y = fetch_openml('mnist_784', version=1, return_X_y=True, as_frame=False)
```

Bộ dữ liệu này sẽ không chia thành *training set* và *test set*, mà sẽ chung lại thành một bộ duy nhất gồm có 70,000 mẫu. **X** được tổ chức thành 784 cột và 70,000 dòng; với các cột lần lượt là *pixel1, ..., pixel784* là giá trị các điểm ảnh trong từng bức ảnh. Và **y** là tập các giá trị nhãn (*class*) của các ảnh trong tập dữ liệu. Sau đó ta mới phân chia tập **X** theo tỉ lệ
train : test = 6 : 1.

3.2.2 Hiển thị và quan sát dữ liệu

Ở bước này ta sẽ lấy 25 dòng dữ liệu đầu tương đương với 25 hình ảnh đầu tiên trong dữ liệu ra để quan sát. Để làm được điều này, đầu tiên ta sẽ thay đổi *shape* của mỗi dữ liệu từ $(784, 1)$ thành $(28, 28)$ rồi chuyển màu của dữ liệu sang *gray* như sau:

```
fig, ax = plt.subplots(5,5, figsize=(10, 10))
for i in range(5):
  for j in range(5):
    img = X[i*5+j].reshape((28, 28))
    fig_img = ax[i,j].imshow(img, cmap='gray')
    ax[i,j].set_title("Label {}".format(y[i*5+j]), y=-0.2)
    fig_img.axes.get_xaxis().set_visible(False)
    fig_img.axes.get_yaxis().set_visible(False)
```



Quan sát tập dữ liệu vẽ tay ở trên ta thấy các nét vẽ ở mỗi hình đậm nhạt khác nhau; có những cặp số có nhiều nét tương đồng, khó phân biệt như $1-7$ hay $4-9$, ... và có lúc cùng một số nhưng lại được vẽ theo cách khác nhau tiêu biểu như số 1 ở dữ liệu thứ 24 và 25. Từ những nhận định trên, chúng ta cần phải thực hiện các bước tiền xử lý kĩ càng (sẽ được trình bày ở phần sau) và một mô hình đủ mạnh để phân loại và nhận dạng chữ số viết tay trên tập **MNIST** trên. Ở đây ta sẽ sử dụng mô hình *Logistic Regression* để thực hiện nhiệm vụ này!

4 Tiền xử lý dữ liệu

Như đã trình bày ở trên thì tiền xử lý dữ liệu (*Preprocessing*) là bước vô cùng quan trọng trong quá trình xây dựng tất cả mô hình, và đặc biệt trong bài toán này. Do đó dưới đây sẽ là các bước tiền xử lý dữ liệu cần thiết cho tập dữ liệu. Trong phần này ta chỉ đề cập đến lý thuyết và các vấn đề xoay quanh của các phương pháp tiền xử lý được áp dụng, cũng như cách áp dụng chúng vào tập dữ liệu của chúng ta. Còn về phần đánh giá hiệu suất cũng như đóng góp của các quá trình này vào mô hình sẽ được đề cập kĩ hơn ở phần xây dựng mô hình.

4.1 Xoay ảnh

Xoay ảnh với 1 góc bất kỳ quanh 1 điểm (ở đây là điểm của ảnh) là một phương pháp tăng cường dữ liệu giúp các thuật toán học được tốt hơn.

Với dữ liệu ảnh số MNIST, như đã hiển thị mẫu ở trên, ta thấy có những ảnh bị nghiêng một góc nhỏ và việc chỉnh lại có vẻ là không khả thi do việc xác định thế nào là nghiêng hay ko ngiêng và chỉnh lại như thế nào cho đúng rất khó khăn và chưa chắc đã chính xác. Nên giải pháp đề ra là tạo thêm các mẫu bằng cách xoay các mẫu gốc một góc nhỏ sẽ giúp cho mô hình nhận dạng được các ảnh số bị nghiêng.

Ở đây ta sẽ xây dựng một hàm xoay ảnh với *OpenCV*:

```
def rotate_image(X, h, w, angle):
    image = X.reshape(h, w)
    center = (w / 2, h / 2)
    M = cv.getRotationMatrix2D(center, angle, 1.0)
    rotated_image = cv.warpAffine(image, M, (h, w))
    rotated_image =  threshold(rotated_image.reshape(-1), 100, 40)
    return rotated_image
```

Các bước để thực hiện việc xoay ảnh bao gồm:

- Bước 1: Tạo ma trận xoay (dòng 3) với tâm quay (*center*), góc quay (*angle*) và tỷ lệ ảnh trước và sau khi xoay (ở đây luôn cho là 1).

- Bước 2: Áp dụng phép biến đổi *Affine* cho ma trận điểm ảnh (*pixel*) với ma trận biến đổi là ma trận xoay vừa tìm được ở Bước 1. Kết quả là ma trận điểm ảnh đã xoay của ảnh đầu vào (chiều dương của quá trình xoay là ngược chiều kim đồng hồ).

- Bước 3: Do ảnh sau khi xoay bị mờ hơn so với ảnh gốc nên dùng hàm **sharpening** để làm ảnh rõ lại. Hàm này tăng giá trị các điểm ảnh và bỏ các điểm ảnh có giá trị nhỏ đi (các điểm ở biên của số).

Sau đây ta sẽ xem xét kết quả của việc xoay ảnh qua một ví dụ lấy từ tập *MNIST*:

4.2 Lọc giá trị nhiễu

Khi quan sát một số mẫu, một số điểm ảnh có thể không có ý nghĩa trong việc phân loại. Chẳng hạn, các điểm ảnh ở biên luôn mang giá trị 0. Do đó ta sẽ thực hiện việc loại bỏ các điểm ảnh nói trên để dữ liệu còn lại sẽ mang nhiều ý nghĩa hơn, phù hợp cho việc phân loại. Để làm được điều này ta sẽ xem xét hai hướng xử lý sau.

4.2.1 Xóa biên kết hợp canh giữa cho hình

Ở đây đầu tiên chúng ta sẽ thực hiện việc canh giữa cho tất cả các hình trước, để thực hiện được việc này đầu tiên ta sẽ tìm hình chữ nhật nhỏ hơn, bao sát hình vẽ (tức là ở bên ngoài sẽ chỉ còn các điểm 0). Sau đó dựa vào tọa độ 4 góc của hình chữ nhật này ta tìm được chính xác tọa độ tâm của mỗi bức hình. Từ đó việc dịch chuyển hình vẽ ở mỗi bức hình về vị trí trung tâm là đơn giản.

Sau khi dịch chuyển ta chỉ cần xóa biên sao cho mỗi hình có biên cho tất cả các hình. Thu được tập dữ liệu mới với tất cả các hình đều có kích thước là 20×20 và tổng cộng đã xóa $26,880,000$ điểm ảnh khỏi toàn tập dữ liệu. Tuy nhiên có thể thấy cách làm này không hoàn toàn tối ưu, ta xem thử một hình được trích ra từ tập *MNIST*.

Nhìn vào hình trên ta để ý thấy có một chấm trắng ở trước số 3, đây là điểm liệu nhiễu. Có nhiều hình ảnh cũng mang những điểm nhiễu như thế này. Và khi thực hiện thuật toán như trên thì ta chỉ có thể cắt tối đa tới trước dấu chấm trắng đó trong khi ở bên trong vẫn còn rất nhiều hàng, cột chứa toàn giá trị 0 không mà ý nghĩa cho việc dự đoán. Kèm theo đó, có thể có những trường hợp hi hữu là giữa các hình sẽ có nét đứt, ta cũng muốn bỏ những điểm đứt quãng đó đi để hình ảnh được liền nét thuận tiện cho mô hình sau này hơn. Từ đó ta đề xuất một cách xử lý khác tốt hơn ở phần tiếp theo.

4.2.2 Xóa tất cả dòng cột chứa toàn giá trị 0

Với cách này ta sẽ đơn giản xóa bỏ đi những dòng, cột mà chỉ chứa toàn số 0 (bao gồm các dòng, cột ở sát biên và ở xen giữa hình vẽ vì nó chỉ khiến cho các nét vẽ cách xa nhau). Sau khi xóa, ta sẽ thực hiện việc điều chỉnh kích thước dữ liệu cho mỗi ảnh trở lại thành $(28, 28)$ cho phù hợp với đầu vào mô hình, bằng hàm **resize** từ thư viện **cv2**.

```
class Image_Filter(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.sum_delete_pixel = 0
    def fit(self, X, y=None):
        return self
    def transform(self, X, y=None):
        X_adj = []
        delete_pixel = np.zeros((X.shape[0],1), dtype = int)
        for i in range(len(X)):
            img = X[i].reshape((28, 28))
            # Xoa dong, cot trong
            index_col = np.any(img != 0, axis=0)
            img = img[:, index_col]
            index_row = np.any(img != 0, axis=1)
            img = img[index_row]
            data = cv.resize(img, dsize=(28, 28), interpolation=cv.INTER_CUBIC)
            data = data.reshape(X.shape[1])
            X_adj.append(data)
            delete_pixel[i] = 784 - img.shape[0]*img.shape[1]
        X_adj=np.array(X_adj)
        X_adj = X_adj.reshape(X_adj.shape[0],-1)
        self.sum_delete_pixel = np.sum(delete_pixel)
        return X_adj
```

Sau khi thực hiện việc lọc các dòng cột kể trên, ta đã loại bỏ 33346053 điểm ảnh ra khỏi toàn bộ tập dữ liệu ban đầu (trước khi đưa các bức ảnh trở lại kích thước cũ). Theo đánh giá bằng quan sát, ta thấy có vẻ như cách giải quyết này đã khắc phục được một phần những hạn chế của phương pháp xử lý trước, đó là theo quan điểm chủ quan. Nhưng để thật sự đánh

giá được phương pháp nào hiệu quả hơn, ta phải xem khi nó áp dụng vào mô hình có thể cải thiện được độ chính xác như thế nào. Lúc đó mới thực sự ra quyết định là nên chọn và bỏ cách xử lý nào. Điều này sẽ được thực hiện ở các bước thử nghiệm ở mục **6**.

4.3 Hiện tượng đa cộng tuyến (Multicollinearity) và cách giải quyết

4.3.1 Đa cộng tuyến là gì?

Đa cộng tuyến (*Multicollinearity*) xảy ra khi hai hoặc nhiều biến độc lập có tương quan cao với nhau trong mô hình hồi quy. Điều này có nghĩa là một biến độc lập có thể được dự đoán từ một (hoặc nhiều) biến độc lập khác trong mô hình hồi quy. Đa cộng tuyến thực sự gây ảnh hưởng đến mô hình hồi quy vì chúng ta sẽ không phân biệt được sự tác động riêng lẻ từ các biến độc lập lên biến phụ thuộc (hoặc có thể chính là lớp của mô hình). Đa cộng tuyến có thể không ảnh hưởng nhiều đến độ chính xác của mô hình, nhưng nó làm giảm độ tin cậy trong việc xác định sự ảnh hưởng của các đặc tính riêng lẻ trong mô hình của chúng ta.

4.3.2 Xác định dấu hiệu xảy ra hiện tưởng đa cộng tuyến

Để nhận biết liệu rằng tập dữ liệu với các thuộc tính hiện có của chúng ta có xảy ra hiện tượng đa cộng tuyến hay không thì có rất nhiều cách, nhưng ở đây ta sẽ trình bày cách xác định phổ biến nhất, đó là **VIF**.

VIF xác định độ tương quan giữa các biến độc lập. Nó được dự đoán bằng cách lấy một biến và tính hồi quy của nó so với mọi biến khác. Hoặc có thể nói đơn giản **VIF score** của một biến độc lập thể hiện mức độ được giải thích của biến đó bằng các biến độc lập khác.

Giá trị R^2 được xác định để tìm hiểu xem một biến độc lập được các biến độc lập khác mô tả tốt như thế nào. Giá trị R^2 cao có nghĩa là biến có tương quan cao với các biến khác. Từ đó ta sẽ tính được **VIF** dựa theo R^2 như công thức sau:

$$\textbf{VIF} = \frac{1}{1 - R^2}$$

Vì $R^2 \in [0; 1]$ nên $\textbf{VIF} \in [1; +\infty)$. Tức là với một biến độc lập cụ thể trong dữ liệu, **VIF** càng cao thì mức độ xảy ra hiện tượng đa cộng tuyến trên biến đó càng cao.

Để thực hiện việc tính **VIF score** cho toàn bộ dữ liệu cụ thể của chúng ta, ta sẽ xây dựng hàm **calc_vif** sau:

```
from statsmodels.stats.outliers_influence import variance_inflation_factor

def calc_vif(X):

    # Calculating VIF
    vif = pd.DataFrame()
    vif["variables"] = X.columns
    vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]

    return(vif)
```

Theo đúng nguyên tắc, chúng ta sẽ dùng hàm trên để tính toán giá trị **VIF score** cho toàn bộ tập dữ liệu rồi xét xem thuộc tính nào có **VIF score** cao nhất thì ta sẽ xóa bỏ. Rồi cập nhật lại **VIF score** cho các thuộc tính còn lại. Ta sẽ thực hiện lặp đi lặp lại các bước trên cho tới khi **VIF score** của các thuộc tính còn lại đều không còn cao (cụ thể không quá 2 là tốt nhất) để chắc chắn rằng không còn hiện tượng đa cộng tuyến trong tập dữ liệu.

Tuy nhiên ở đây do kích thước của tập dữ liệu quá lớn. Cụ thể với 784 cột tương ứng là 784 *pixel* của mỗi ảnh, ta phải đi tính hồi quy của lần lượt mỗi cột thuộc tính đối với 783 cột còn lại để có được giá trị **VIF** rồi lại lặp lại bước này rất nhiều lần, đòi hỏi chi phí về thời gian quá lớn. Thêm vào đó tập dữ liệu có hẳn 60,000 dòng tương ứng với 60,000 bức ảnh thuộc tập *train* nên việc tính toán như trên là bất khả thi. Vì vậy ta sẽ thực hiện thử nghiệm này trên một mẫu nhỏ hơn, tức là chỉ trích 1000 dòng đầu của dữ liệu, để khám phá xem mức độ phục thuộc của các cột thuộc tính này với nhau cụ thể ra sao:

	variables	VIF
0	0	inf
1	1	inf
2	2	inf
3	3	inf
4	4	inf
...	...	...
779	779	inf
780	780	inf
781	781	inf
782	782	inf
783	783	inf

784 rows × 2 columns

Sau khi thực hiện quá trình tính toán và in kết quả, có vẻ như tất cả các giá trị **VIF** của các cột thuộc tính đều trả về là vô cùng (*Inf*). Điều này cũng dễ hiểu, ta có thể quan sát từ bảng tính **Correlation** sau:

```
corr_X =pd.DataFrame(X_adj) .corr()
corr_X[corr_X>0.8]
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1.000000	0.922304	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	0.922304	1.000000	0.941053	0.824617	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	0.941053	1.000000	0.940177	0.813005	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	0.824617	0.940177	1.000000	0.939697	0.810584	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	0.813005	0.939697	1.000000	0.945637	0.819343	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
779	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
780	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
781	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
782	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
783	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

784 rows × 784 columns

Từ bảng trên (đã lọc ra, chỉ giữ lại các giá trị **correlation** > 0.8), ta thấy được có vẻ các ô ở cạnh nhau (khi *reshape* lại kích cỡ $(28, 28)$) thì có mối liên quan mật thiết với nhau. Mà dữ liệu về hình ảnh này lại được xếp như một ma trận có kích thước $(28, 28)$, do đó tất cả các cột thuộc tính (hay là các điểm ảnh) có mối quan hệ hồi quy chặt chẽ với nhau, dẫn đến hiện tượng trên. Ta cùng xem xét cách để giải quyết hiện tượng đa cộng tuyến trên tập dữ liệu của chúng ta.

4.3.3 Hướng giải quyết hiện tượng đa cộng tuyến

Như đã trình bày ở trên, do đặc điểm tập dữ liệu **MNIST** rất lớn nên việc thực hiện phương pháp phổ thông dựa vào tính toán **VIF** là bất khả thi. Thêm vào đó vì tất cả các giá trị **VIF** của các cột thuộc tính ban đầu đều là *Inf* nên rất khó để lựa chọn việc bỏ và giữ thuộc tính nào. Do đó ta cần tìm kiếm một thuật toán tối ưu hơn, với tốc độ xử lý cũng như kết quả thu được tốt hơn so với phương pháp truyền thống. Một trong các phương pháp quen thuộc và dễ tiếp cận được đề xuất ở đây là một trong những thuật toán giảm chiều dữ liệu: *Principal Component Analysis (PCA)*.

4.3.4 Principal Component Analysis (PCA) và ứng dụng vào mô hình

a. Principal Component Analysis (PCA) là gì?

PCA là thuật toán làm giảm chiều dữ liệu - chuyển dữ liệu n chiều sang dữ liệu m chiều (với $m < n$) mà vẫn giữ được nhiều thông tin nhất có thể. Ý tưởng của thuật toán là biểu diễn các *vector* dữ liệu ban đầu trong một hệ cơ sở mới mà trong hệ cơ sở mới đó, tầm quan trọng giữa các thành phần là khác nhau rõ rệt, thì chúng ta có thể bỏ qua những thành phần ít quan trọng nhất. *PCA* còn được gọi là phương pháp tối đa tổng phương sai được giữ lại. Vậy ta có thể coi tổng các phương sai được giữ lại là lượng thông tin được giữ lại, với phương sai càng lớn, tức dữ liệu có độ phân tán cao, thể hiện lượng thông tin càng lớn.

b. Ứng dụng PCA vào tập dữ liệu

Ta dùng hàm **PCA()** từ thư viện **sklearn.decomposition** với các tham số quan trọng như sau:

- **n_components**:
  - Nếu **n_components** là 1 số nguyên thì **n_components** chính là số chiều còn lại của dữ liệu sau khi qua thuật toán *PCA*.
  - Nếu 0 < **n_components** < 1: chọn số lượng thành phần sao cho lượng phương sai còn lại lớn hơn hoặc bằng phần trăm được chỉ định bởi **n_components**. Hay nói cách khác là phần trăm lượng thông tin được giữ lại phải lớn hơn hoặc bằng giá trị được chỉ định.
- **svd_solver** là tham số để chọn các thuật giải cho mô hình *PCA*
- **random_state** là một giá trị số nguyên để kiểm soát trình tạo số ngẫu nhiên cho thuật toán khi **svd_solver**= 'arpack' hoặc 'randomized'

Ở đây ta sẽ tạo mô hình *PCA* với hai tham số truyền vào là:

- **n_components** = 70.
- **svd_solver** = "randomized".
- **random_state** = 0

Cách lựa chọn các tham số như trên sẽ được giải thích ở phần các bước thử nghiệm. Sau khi truyền tập dữ liệu qua mô hình *PCA* ta được tập dữ liệu mới với **VIF score** như sau:

	variables	VIF
0	0	1.057503
1	1	1.076554
2	2	1.068828
3	3	1.065656
4	4	1.057531
...	...	...
65	65	1.100008
66	66	1.045983
67	67	1.057367
68	68	1.068895
69	69	1.086676

70 rows × 2 columns

Như trên hình ta có thể thấy sau khi đi qua mô hình thuật toán *PCA* vừa được xây dựng, số cột thuộc tính của dữ liệu đã giảm xuống như mong muốn từ 784 còn 70 thuộc tính, và để ý thấy dữ liệu không còn xảy ra hiện tượng đa cộng tuyến nữa. Điều này được giải thích là trong quá trình giảm chiều của thuật toán, các thuộc tính liên quan gần với nhau đã được gom lại. Vậy là ta đã giải quyết hoàn toàn vấn đề đa cộng tuyến trong tập dữ liệu.

Ngoại trừ tác dụng được nêu ở trên, việc áp dụng thuật toán *PCA* vào tập dữ liệu của chúng ta cũng góp phần không nhỏ vào việc triển khai thuật toán được đề cập tiếp theo đây: *Polynomial Features*. Mà dựa vào đó ta có thể tăng độ chính xác của mô hình lên rất nhiều. Cụ thể sẽ được trình bày chi tiết ở phần bên dưới.

4.4 Polynomial Features

4.4.1 Polynomial Features là gì?

Polynomial Features là một dạng của *feature engineering* – quá tình tạo ra các thuộc tính mới dựa trên các thuộc tính có sẵn. Thuật toán này tạo ra một tập các thuộc tính mới dựa trên các số hạng có thể có của một đa thức bậc n với các biến chính là các thuộc tính của dữ liệu.

Ví dụ: Với đa thức 2 biến x, y có bậc là 2 thì các số hạng có thể có là: $1, x, y, xy, x^2, y^2$, trong thuật toán này số 1 sẽ được gọi là *bias*.

Vậy với dữ liệu có m thuộc tính sau khi áp dụng *Polynomial Features* với bậc n thì số lượng thuộc tính sẽ là $\binom{m+n}{n}$ (có tính thuộc tính *bias*). Với m lớn và bậc n từ 3 trở lên thì số lượng thuộc tính tăng lên rất nhiều.

Ví dụ với trường hợp 30 thuộc tính, số lượng thuộc tính sau khi qua *Polynomial Features*:

- $n = 2$: 496 thuộc tính
- $n = 3$: 5456 thuộc tính
- $n = 4$: 46376 thuộc tính

Chính vì vậy, như đã đề cập ở trên, đối với dữ liệu bài toán của chúng ta, chính nhờ vào việc đã đưa dữ liệu qua thuật toán *PCA* để giảm chiều dữ liệu (nhằm giảm tối đa số lượng thuộc tính có thể nhưng vẫn dữ lại những đặc trưng quan trọng nhất) thì ta có thể thực hiện thuật toán *Polynomial Features* ngay sau đó mà không gặp phải vấn đề chi phí về thời gian và bộ nhớ quá lớn không đáp ứng đủ.

4.4.2 Ứng dụng Polynomial Features vào tập dữ liệu

Ta dùng hàm **PolynomialFeatures()** từ thư viện **sklearn.preprocessing** với các tham số quan trọng như sau:

- **degree**: Bậc của đa thức, mặc định là 2.
- **include_bias**: mặc định là *True* – thêm thuộc tính *bias* – cột với dữ liệu là 1 cho tất cả các mẫu.
- **interaction_only**: mặc định là *False*. Nếu *True* thì thuật toán chỉ tạo ra các thuộc tính là tập hợp các thuộc tính gốc nhân với nhau mà không có mũ – các tổ hợp k thuộc tính gốc với k = 1..degree.

Ở đây ta chỉ tạo mô hình *Polynomial Features* đơn giản với một tham số truyền vào là: **include_bias** = *True*. Sau khi đưa dữ liệu đã được xử lý ở bước trước (*PCA*) đi qua mô hình thuật toán *Polynomial Features* ta được tập dữ liệu mới với số lượng thuộc tính tăng lên rất nhiều, kèm theo đó là độ chính xác của mô hình cũng được cải thiện đáng kể, sẽ được đề cập ở phần sau.

4.5 Standard Scaler

Đây là bước chuẩn hóa dữ liệu sao cho phân phối của từng thuộc tính sẽ có giá trị trung bình là 0 và độ lệch chuẩn là 1. Nhiều thành phần trong các hàm mục tiêu của các mô hình này giả định rằng tất cả các thuộc tính của dữ liệu có phân bố xoay quanh 0 và có cùng độ lệch chuẩn vì vậy ta cần cho dữ liệu qua *Standard Scaler* trước khi đưa vào huấn luyện.

Cách chuẩn hóa một giá trị x của một thuộc tính như sau:

$$z = \frac{x - u}{s}$$

với u và s là trung bình và độ lệch chuẩn của thuộc tính đó.

Kết quả của quá trình xử lý này sẽ giúp cho nhiều thuật toán quan trọng trong Máy học sử dụng kĩ thuật *Gradient Descent* hội tụ nhanh. Nhờ đó giúp giảm bớt thời gian thực hiện của các thuật toán và tăng độ chính xác.

Ở bước này chúng ta đơn giản gọi hàm **StandardScaler()** từ thư viện **sklearn.preprocessing** để chuẩn hóa dữ liệu, là bước tiền xử lý cuối cùng trước khi đưa vào mô hình.

5 Xây dựng mô hình Logistic Regression

5.1 Nhắc lại đầu vào và đầu ra của mô hình

Như đã trình bày trong quá trình tiền xử lý, dữ liệu thô của chúng ta sẽ là tập dữ liệu *X_ train* gồm $60,000$ dòng và 784 cột. Sau đó qua lần lượt các quá trình tiền xử lý sau:

- Canh giữa kết hợp xóa biên hoặc xóa tất cả dòng cột chứa toàn giá trị 0.
- Đưa qua mô hình thuật toán *PCA*.
- Đưa qua mô hình thuật toán *Polynomial Features*.
- Đưa qua hàm chuẩn hóa *Standard Scaler*.

Tất cả những quyết định cũng như tham số của các hàm của những thuật toán trên sẽ được xác định ở thử nghiệm cuối cùng tại mục **6**. Sau khi qua lần lượt tất cả các bước trên, dữ liệu cuối cùng thu được sẽ là dữ liệu đầu vào của mô hình. Đầu ra của mô hình sẽ là xác suất tại mỗi lớp của tập dữ liệu được đưa vào.

5.2 Cấu trúc và cách thiết kế của mô hình Logistic Regression

5.2.1 Tổng quan về mô hình

Logistic Regression (Hồi quy *logistic*), mặc dù có tên gọi như vậy nhưng đây lại là một mô hình tuyến tính để phân loại chứ không phải là hồi quy. *Logistic Regression* còn được gọi là hồi quy *logit*, phân loại *entropy* cực đại (*MaxEnt*) hoặc bộ phân loại *log*-tuyến tính (*log-linear*). Trong mô hình này, các xác suất mô tả các kết quả có thể có của một thử nghiệm đơn lẻ được mô hình hóa bằng cách sử dụng một hàm *logistic*.

Đầu ra dự đoán của mô hình losgictic regression sẽ có dạng:

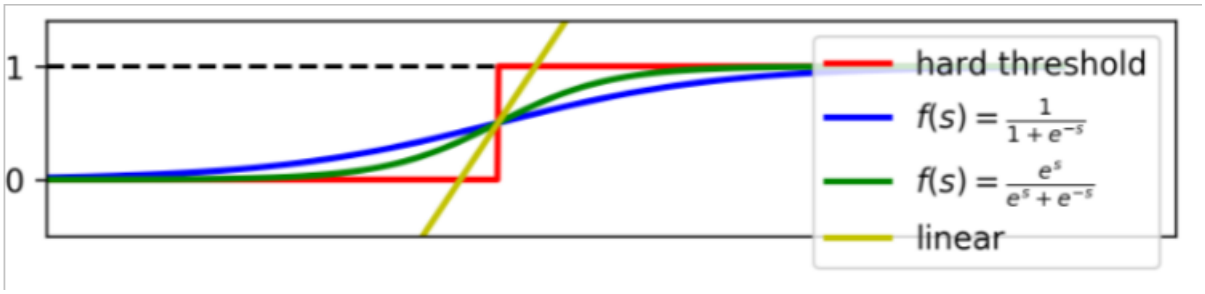
- $P(y = 1|x) = \theta(w^T x)$
- $P(y = 0|x) = 1 - \theta(w^T x)$

Trong đó x là dữ liệu mở rộng với $x_0 = 1$ được thêm vào để thuận tiện cho việc tính toán, θ được gọi là *Acticvation Function*.

Hồi quy *logistic* được triển khai bằng hàm **LogisticRegression()** trong thư viện **sklearn**. Hàm này có thể phù hợp cả cho các loại như nhị phân (*binary*), One-vs-Rest và hồi quy đa thức (*Multinomial Logistic Regression*) với các tùy chọn l_1, l_2 hoặc *Elastic-Net regularization*. Đối với tập dữ liệu *MNIST* này, cụ thể chúng ta sẽ sử dụng hồi quy đa thức để giải bài toán phân loại trên. Sau đây là các cơ sở lý thuyết cũng như cấu trúc và cách thiết kế cụ thể của mô hình *Logistic Regression*.

5.2.2 Sigmoid Function

Một số activation cho mô hình tuyến tính được cho trong hình dưới đây:



Để phù hợp với mục đích của mô hình *Losgictic Regression* các *activation* cần thỏa các tính chất sau:

- Là hàm số liên tục nhận giá trị thực, bị chặn trong khoảng $(0, 1)$.
- Nếu coi điểm có tung độ là $\frac{1}{2}$ làm điểm phân chia thì các điểm càng xa điểm này về phía bên trái có giá trị càng gần 0. Ngược lại, các điểm càng xa điểm này về phía phải có giá trị càng gần 1.
- Mượt (smooth) nên có đạo hàm mọi nơi, có thể thuận lợi trong việc tối ưu hóa.

Trong số các hàm số có 3 tính chất nói trên thì hàm *sigmoid*:

$$f(s) = \frac{1}{1 + e^{-s}} = \sigma(s)$$

Được sử dụng nhiều nhất, vì nó bị chặn trong khoảng (0,1). Thêm vào đó:

$$\lim_{x \to -\infty} \sigma(s) = 0; \lim_{x \to +\infty} \sigma(s) = 1$$

Đặc biệt hơn nữa:

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

Chính vì công thức đạo hàm đơn giản này mà hàm *sigmoid* được sử dụng rộng rãi. Ngoài ra nó còn có tính chất sau:

$$1 - \sigma(x) = \sigma(-x)$$

Vì vậy ta có thể tính $P(y = 0|x) = \sigma(-wx)$.

5.2.3 Cross-entropy Loss Function

Cross-entropy Loss Function là hàm mất mát được sử dụng trong *Logistic Regression*. Ta sẽ tìm hiểu cách xây dựng hàm này.

Với từng điểm dữ liệu huấn luyện (đã biết đầu ra y), ta có:

$$P(y_i = 1|x_i; w) = f(w^T x_i)$$

$$P(y_i = 0|x_i; w) = 1 - f(w^T x_i)$$

Mục đích của chúng ta là tìm các hệ số w sao cho $f(w^T x_i)$ càng gần với 1 càng tốt với các điểm thuộc lớp 1 và càng gần với 0 càng tốt với các điểm

thuộc lớp 0. Ký hiệu $z_i = f(w^T x_i)$ và viết gộp lại hai biểu thức ở trên ta có:

$$P(y_i|x_i; w) = z_i^{y_i}(1 - z_i)^{1-y_i}$$

Xét toàn bộ tập huấn luyện với $X = [x_1, x_2, ..., x_N] \in R^{d \times N}$ và
$y = [y_1, y_2, ..., y_N]$, cần tìm w để biểu thức $P(y|X; w)$ đạt giá trị lớn nhất. Hàm số này còn được gọi là *Likelihood Function.*

Giả sử thêm rằng các điểm dữ liệu được sinh ra một cách ngẫu nhiên, độc lập với nhau, ta có thể viết:

$$P(y|X; w) = \prod_{i=1}^{N} P(y_i|x_i; w) = \prod_{i=1}^{N} z_i^{y_i}(1 - z_i)^{1-y_i}$$

Trực tiếp tối ưu hàm số này theo w nhìn qua có vẻ không đơn giản. Hơn nữa, khi N lớn, tích của N số nhỏ hơn 1 có thể dẫn tới sai số trong tính toán. Một phương pháp thường được sử dụng đó là lấy *logarit* tự nhiên (cơ số e) của *likelihood function*, biến phép nhân thành phép cộng để tránh việc số quá nhỏ. Sau đó lấy ngược dấu để được 1 hàm và coi nó là hàm mất mát (*loss function*). Lúc này bài toán tìm giá trị lớn nhất của *likelihood function* trở thành bài toán tìm giá trị nhỏ nhất của hàm mất mát (hàm này còn được gọi là *negative log likelihood*).

$$-logP(y|X; w) = -\sum_{i=1}^{N}(y_i log z_i + (1 - y_i) log(1 - z_i))$$

Biểu thức vế phải có tên gọi là *cross entropy*, thường được sử dụng để đo khoảng cách giữa hai phân phối. Trong bài toán này, một phân phối là đầu ra y của tập huấn luyện với xác suất chỉ có 0 hoặc 1, phân phối còn lại được tính theo mô hình *logistic regression.* Khoảng cách giữa 2 phân phối này nhỏ có vẻ như đồng nghĩa với việc 2 phân phối đó rất gần nhau.

Với toàn bộ điểm dữ liệu, hàm mất mát trong *losgictic regression* sẽ là như sau:

$$L_{CE}(\hat{y}, y) = -[y log\sigma(wx) + (1 - y) log(1 - \sigma(wx))]$$

5.2.4 Gradient Descent

Mục đích của *Gradient Descent* là để tìm ra các trọng số tối ưu, cực tiểu hóa hàm mất mát. Ta xem hàm mất mát là một hàm số với các biến là

w. Nhìn chung, việc tìm cực tiểu toàn cục của các hàm mất mát trong *Machine Learning* là rất phức tạp, thậm chí là bất khả thi. Thay vào đó, người ta thường cố gắng tìm các điểm cực tiểu cục bộ, và ở một mức độ nào đó, coi đó là nghiệm cần tìm của bài toán. *Gradient descent* sẽ tìm điểm cực tiểu cục bộ của hàm số bằng cách xuất phát từ một điểm và tìm hướng đi (của các biến) mà theo đó giá trị của hàm số sẽ giảm dần (càng tiến gần đến điểm cực tiểu) và di chuyển theo hướng đó cho đến khi đến một điểm cực tiểu (có thể là cực tiểu cục bộ), tức là đến khi đạo hàm gần với 0.

Thuật toán *Gradient Descent* sẽ trả lời câu hỏi hướng đi nào là đúng bằng cách tính đạo hàm của hàm lỗi. Đạo hàm của một hàm số nhiều biến tại một điểm sẽ là một *vector* chỉ hướng tăng của một hàm số tại điểm đó. Vậy chỉ cần cho các trọng số hiện tại (vị trí của điểm) di chuyển theo hướng ngược lại.

Vậy để tìm cực trị cho một hàm $f(\theta)$, cho trước một điểm dự đoán θ_0. Sau đó, ở vòng lặp thứ t, quy tắc cập nhật là:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta f(\theta_t)$$

Trong đó ∇ là một số dương được gọi là *learning rate* (tốc độ học), càng lớn tức là sự thay đổi của θ sau mỗi vòng lặp càng lớn.

Sau đây ta sẽ nói về thuật toán *Gradient* cho *Logistic Regression*. Có một điều quan trọng là trong *losgictic regression*, hàm mất mát là một hàm lồi (là hàm số chỉ có cực trị) vì vậy không có các giá trị cực tiểu cục bộ để thuật toán mắc phải trước khi đến được cực tiểu toàn cục.

Trong mô hình *Logistic regression*, θ chính là các trọng số w, $f(\theta)$ sẽ là hàm mất mát *cross entropy*. Với đạo hàm của hàm mất mát trong *logistic* sẽ như sau:

$$\nabla_{\boldsymbol{\theta}} L(f(x;\boldsymbol{\theta}),y)) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\boldsymbol{\theta}),y) \\ \frac{\partial}{\partial w_2} L(f(x;\boldsymbol{\theta}),y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x;\boldsymbol{\theta}),y) \end{bmatrix}$$

Thì công thức cuối cùng cho cập nhật θ sẽ là:

$$\theta_{t+1} = \theta_t - \eta \nabla L(f(x;\theta),y)$$

Trong đó các đạo hàm riêng của hàm mất mát sẽ có dạng:

$$\frac{\partial L_{CE}(\hat{y}, y)}{\partial w_j} = [\sigma(wx) - y]x_j$$

5.2.5 Regularization

Đầu tiên ta sẽ nói về vấn đề *Overfitting*. Đây là hiện tượng mô hình tìm được quá khớp với dữ liệu *training*. Việc quá khớp này có thể dẫn đến việc dự đoán nhầm nhiễu, và chất lượng mô hình không còn tốt trên dữ liệu test nữa. Về cơ bản, *overfitting* xảy ra khi mô hình quá phức tạp để mô phỏng *training data*. Điều này đặc biệt xảy ra khi lượng dữ liệu *training* quá nhỏ trong khi độ phức tạp của mô hình quá cao.

Trong các bài toán *Machine Learning*, lượng tham số cần xác định thường lớn hơn nhiều, và khoảng giá trị của mỗi tham số cũng rộng hơn nhiều, chưa kể đến việc có những tham số có thể là số thực. Như vậy, việc chỉ xây dựng một mô hình thôi cũng là đã rất phức tạp rồi. Có một cách giúp số mô hình cần huấn luyện giảm đi nhiều, thậm chí chỉ một mô hình. Cách này có tên gọi chung là *regularization*.

Kỹ thuật *regularization* phổ biến nhất là thêm vào hàm mất mát một số hạng nữa. Số hạng này thường dùng để đánh giá độ phức tạp của mô hình. Số hạng này càng lớn, thì mô hình càng phức tạp. Hàm mất mát mới này thường được gọi là *regularized loss function*, thường được định nghĩa như sau:

$$J_{reg}(\theta) = J(\theta) + \lambda R(\theta)$$

Trong đó θ chính là các trọng số w trong mô hình *Logistic regression*, $J(\theta)$ sẽ là hàm mất mát *cross entropy*.

Việc tối thiểu *regularized loss function*, nói một cách tương đối, đồng nghĩa với việc tối thiểu cả *loss function* và số hạng *regularization*. Ta dùng cụm "nói một cách tương đối" vì nghiệm của bài toán tối ưu *loss function* và *regularized loss function* là khác nhau. Chúng ta vẫn mong muốn rằng sự khác nhau này là nhỏ, vì vậy tham số *regularization (regularizaton parameter)* λ thường được chọn là một số nhỏ để biểu thức *regularization* không làm giảm quá nhiều chất lượng của nghiệm.

5.2.6 Multinomial Logistic Regression

Các bài toán *classification* thực tế thường có rất nhiều *classes (multi-class)*, các *binary classifiers* mặc dù có thể áp dụng cho các bài toán *multi-class*,

chúng vẫn có những hạn chế nhất định. Với *binary classifiers*, kỹ thuật được sử dụng nhiều nhất là *one-vs-rest* có một hạn chế về tổng các xác suất. Một phương pháp mở rộng của *Logistic Regression* giúp khắc phục hạn chế trên là *Multinomial logistic regression* (hay còn gọi là *Softmax Regression*).

Với 1 *vector* k chiều z, $z = [z_1, z_2, ...z_k]$ kết quả hàm *softmax* sẽ là một *vector* k chiều được định nghĩa như sau:

$$softmax(z_i) = \frac{exp(z_i)}{\sum_{j=1}^{k} exp(z_j)}, 1 \leq i \leq k$$

Hoặc:

$$softmax(z) = \left[\frac{exp(z_1)}{\sum_{j=1}^{k} exp(z_j)}, \frac{exp(z_2)}{\sum_{j=1}^{k} exp(z_j)}, ..., \frac{exp(z_k)}{\sum_{j=1}^{k} exp(z_j)} \right]$$

Mẫu số $\sum_{j=1}^{k} exp(z_j)$ trong công thức của hàm *softmax* giúp chuẩn hóa các giá trị của kết quả về xác suất, tức tổng các giá trị sẽ bằng 1.

Về vấn đề học trong *Multinomial Logistic Regression*. Cũng như hàm *sigmoid*, đầu vào của hàm *softmax* trong *Multinomial Logistic Regression* sẽ là tích vô hướng giữa trọng số w và *vector* đầu vào x. Nhưng ở đây ta cần các k bộ trọng số w khác nhau cho k lớp. w_c sẽ là bộ trọng số tương ứng với lớp c. Hàm likelihood với mô hình có k lớp:

$$p(y = c|x) = \frac{exp(w_c x)}{\sum_{j=1}^{k} exp(w_j x)}$$

Cũng như hàm mất mát trong *Logistic Regression* với 2 lớp sẽ có dạng:

$$L_{CE}(\hat{y}, y) = -logp(y|x) = -[y log\hat{y} + (1 - y) log(1 - \hat{y})]$$

Hàm mất mát trong *Multinomial Logistic Regression* với k lớp sẽ như sau:

$$L_{CE}(\hat{y}, y) = -\sum_{k=1}^{K} y_k log\hat{y}_k = -\sum_{k=1}^{K} y_k log\hat{p}(y = k|x)$$

Trong *Multinomial Logistic Regression*, y không còn là một giá trị 0 hoặc 1 nữa mà là một *vector* với giá trị 1 ở lớp đúng và giá trị 0 ở các lớp còn lại, hàm mất mát sẽ được viết lại như sau:

$$
\begin{aligned}
L_{\text{CE}}(\hat{y}, y) &= -\sum_{k=1}^{K} \mathbb{1}\{y = k\} \log \hat{p}(y = k|x) \\
&= -\sum_{k=1}^{K} \mathbb{1}\{y = k\} \log \frac{\exp(w_k \cdot x + b_k)}{\sum_{j=1}^{K} \exp(w_j \cdot x + b_j)}
\end{aligned}
$$

Do đó *cross-entropy loss* chỉ đơn giản là hàm *log* của xác xuất đầu ra tương ứng với lớp chính xác, và do đó chúng ta cũng có thể gọi đây là hàm *negative log likelihood loss*:

$$
\begin{aligned}
L_{CE}(\hat{y}, y) &= -log\hat{y}_k \\
&= -log\frac{exp(w_k x + b_k)}{\sum_{j=1}^{K} exp(w_j x + b_j)}
\end{aligned}
$$

Với k là lớp chính xác.

Đạo hàm riêng cho một mẫu của hàm lỗi khá giống với các đạo hàm riêng trong *binary Logistic Regression*:

$$
\begin{aligned}
\frac{\partial L_{\text{CE}}}{\partial w_{k,i}} &= -(\mathbb{1}\{y = k\} - p(y = k|x))x_i \\
&= -\left(\mathbb{1}\{y = k\} - \frac{\exp(w_k \cdot x + b_k)}{\sum_{j=1}^{K} \exp(w_j \cdot x + b_j)}\right) x_i
\end{aligned}
$$

5.3 Hàm LogisticRegression() trong thư viện sklearn và ứng dụng vào bài toán

Đối với bài toán này, sau khi đã thực hiện các bước tiền xử lý như đã trình bày ở trên, ta sẽ chỉ việc đưa dữ liệu cuối cùng vào hàm **LogisticRegression()** trong thư viện **sklearn** để huấn luyện và đưa ra kết quả cuối cùng. Điều quan trọng ở đây là chọn giá trị cho các tham số của mô hình để phù hợp nhất với tập dữ liệu và cho ra mô hình tốt nhất với độ chính xác cao.

Ta sẽ đi tìm hiểu những tham số quan trọng được cho là cần thiết đối với bài toán này, và cách chọn giá trị cho các tham số đó.

5.3.1 Các tham số quan trọng

a. Penalty

Tham số này để tránh hiện tượng *overfit*, có các giá trị là {'l1', 'l2', 'elasticnet', 'none'}.

b. Solver

Tham số này được dùng để tối ưu hóa, có các giá trị là {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}. Các trường hợp sử dụng của các loại thuật toán này như sau:

- Đối với các tập dữ liệu nhỏ, 'liblinear' là một lựa chọn tốt, trong khi 'sag' và 'saga' nhanh hơn đối với các tập lớn.
- Đối với các trường hợp cần phân loại nhiều lớp, chỉ có 'newton-cg', 'sag', 'saga' và 'lbfgs' xử lý tốt. 'liblinear' phù hợp hơn với phương pháp *one-versus-rest*.
- 'Newton-cg', 'lbfgs', 'sag' và 'saga' chi xử lý L2 hoặc không không có *penalty*.
- 'Liblinear' và 'saga' cũng xử lý cho cả trường hợp L1.
- 'saga' cũng hỗ trợ 'elasticnet' penalty
- 'liblinear' không hỗ trợ cho trường hợp penalty='none'.

c. Tol và max_iter

Tol được coi là tiêu chí để dừng thuật toán, mặc định là 10^{-4}. Khi hàm lỗi *Cross-entropy* đạt được giá trị nhỏ hơn *'tol'* thì thuật toán được coi là hội tụ và sẽ dừng lại. Tham số này ta sẽ chọn giá trị là 0.01.

Đi kèm với *'tol'* là *'max_iter'*, đây là số vòng lặp tối đa, mặc định là 100. Ta có thể hiểu có những trường hợp bài toán mà thuật toán không thể hội tụ, hoặc hội tụ rất chậm, lúc này tham số này là cần thiết để dừng thuật toán và tạm chấp nhận kết quả tốt nhất đạt được cho tới lúc đó.

d. Multi_class

Đây là tham số để chọn loại mô hình *Logistic Regression* phù hợp cho bài toán, có danh sách tùy chọn là {'auto', 'ovr', 'multinomial'}.

e. Random_state

Được sử dụng khi *solver == 'sag', 'saga' hoặc 'liblinear'* để xáo trộn dữ liệu, mặc định là *None*.

5.3.2 Lựa chọn giá trị các tham số phù hợp cho bài toán

Như đã trình bày về các tham số ở trên, và đây là quyết định để chọn các giá trị phù hợp cho các tham số đó qua quá trình thử nghiệm:

- **Penalty** = 'l2': Sẽ được giải thích ở phần thử nghiệm mô hình.
- **Solver** = 'saga': Đây là thuật toán tương tự và cải tiến của 'sag' (viết tắt của *Stochastic Average Gradient*). Được mô tả bằng công thức:

$$w^{(k)} = w^{(k-1)} - t_k.(g_{i_k}^{(k)} - g_{i_k}^{(k-1)} + \frac{1}{n}\sum_{i=1}^{n} g_i^{(k-1)})$$

- **Tol** = 0.01 và **max_iter** = 100 (mặc định): Trong quá trình thử nghiệm thì tham số **tol** = 0.01 có vẻ là phù hợp nhất. Tuy nếu hạ giá trị thấp hơn có thể đạt được độ chính xác cao hơn, nhưng là không đáng kể, thậm chí là thuật toán không hội tụ. Còn nếu chỉnh cho tham số **max_iter** cao hơn, tức số vòng lặp nhiều hơn, thì cũng chưa chắc thuật toán đã hội tụ, thêm vào đó còn rất tốn thời gian thực thi, có thể lên tới vài tiếng đồng hồ.
- **Multi_class**: Ta sẽ để mặc định là 'auto', và dựa theo dữ liệu bài toán thì hàm sẽ tự động chuyển về 'multinomial', rất phù hợp cho việc giải quyết bài toán của ta.
- **random_state** = 0: Cố định một giá trị để đảm bảo tính khách quan trong việc quan sát và đánh giá mô hình.

6 Các bước thử nghiệm để tìm mô hình tốt nhất và đánh giá

6.1 Các bước thử nghiệm

Các thử nghiệm này nhằm xem xét và đánh giá độ hiệu quả của các phương pháp tiền xử lý trước khi xây dựng *Pipeline* hoàn chỉnh cho mô hình. Từ đó chọn lựa và sắp xếp các bước tiền xử lý hợp lý, cũng như chọn các bộ tham số phù hợp giúp mang lại kết quả tốt nhất cho mô hình cuối cùng.

Ở các bước thử nghiệm này chúng ta sẽ phân chia tập dữ liệu ban đầu thành hai tập mới theo tỉ lệ $train : test = 6 : 1$. Ta sẽ dùng hàm **train_test_split()** để thực hiện nhiệm vụ trên, với các tham số cần chú ý là:

- **test_size** = 1/7: Nhằm phân chia 2 tập theo tỉ lệ đã định trước.
- **random_state** = 0: Giúp cho việc chia tập ở mỗi lần luôn là cố định, đảm bảo tính khách quan trọng việc đánh giá và so sánh ở các bước thử nghiệm.
- **stratify** = y: Nhằm đảm bảo trong quá trình phân chia thì dữ liệu ở các lớp trong tập *train* là đều nhau.

Sau đây là lần lượt các bước thử nghiệm.

6.1.1 Thử nghiệm 1: Huấn luyện mô hình không xử lí dữ liệu

Ở thử nghiệm này ta sẽ lấy nguyên tập X và y như ban đầu khi tải tập *MNIST* về để phân chia dữ liệu và tiến hành đưa vào mô hình *Logistic Regression*, cụ thể như sau:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/7, random_state=0, stratify=y)
```

Sau đó ta sẽ thực hiện bước chuẩn hóa dữ liệu bằng hàm **StandardScaler()** như đã trình bày ở phần ý tưởng:

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Sau khi thực hiện chuẩn hóa dữ liệu này xong, ta sẽ đưa tập *train* vào huấn luyện rồi sau đó sẽ tính độ chính xác của mô hình trên tập *train* cũng như tập *test*. Ở đây ta cũng làm thêm một thử nghiệm nhỏ với tham số **penalty** của hàm **LogisticRegression()** như đã nói ở trên, với hai giá trị cần xem xét là 'l1' và 'l2'.

```
Train score with L1 penalty: 0.9183 Train score with L2 penalty: 0.9183
Test score with L1 penalty: 0.9143  Test score with L2 penalty: 0.9148
```

Ở đây ta có hai nhận xét:

- Độ chính xác của mô hình *Logistic Regression* cho tập dữ liệu *MNIST*, với bộ tham số đã chọn là tương đối cao cho dù chưa qua bất cứ bước tiền xử lý dữ liệu nào.
- Ta thấy độ chính xác trên tập *train* và *test* đối với 2 tham số 'l1' và 'l2' là gần như giống nhau, thậm chí 'l2' còn nhỉnh lên một chút không đáng kể. Thêm vào đó công thức của 'l2' dễ tính đạo hàm hơn nên thường được sử dụng phổ biến hơn. Do đó sau bước này ta sẽ sử dụng 'l2' là tham số cho mô hình trong suốt quá trình thử nghiệm cũng như cho mô hình ở bước xây dựng cuối cùng.

6.1.2 Thử nghiệm 2: Huấn luyện mô hình sau khi canh giữa ảnh và cắt biên

Ở đây sẽ chỉ khác một điểm là ta sẽ tách *X_ train* và *X_ test* từ tập dữ liệu mới là *X_ center*, là tập dữ liệu đã được xử lý qua quá trình canh giữa và cắt biên (được trình bày ở phần tiền xử lý mục **4.2.1**). *X_ center* có dạng (70000, 400) vì mỗi ảnh sau bước xử lý này chỉ còn kích thước là (20, 20). Sau đó ta cũng đưa dữ liệu qua bước chuẩn hóa như trên rồi đưa tập *train* vào hàm **LogisticRegression()** với các tham số đã định sẵn. Và cuối cùng là hiển thị độ chính xác trên hai tập *train* và *test* như vừa rồi.

```
Train score with L2 penalty: 0.9308
Test score with L2 penalty: 0.9213
```

Quan sát thấy qua bước tiền xử lý này thì độ chính xác trên hai tập *train* và *test* đã tăng nhưng không nhiều, chỉ trên dưới 1%. Điều này dễ hiểu vì độ chính xác đã cao sẵn rồi và đây còn là một bước xử lý quá đơn giản, kèm theo đó là những hạn chế đã được trình bày ở phần ý tưởng. Ta sẽ cùng đi đến phương pháp được coi là giúp cải thiện hơn hạn chế của phương pháp này.

6.1.3 Thử nghiệm 3: Huấn luyện mô hình sau khi loại bỏ các dòng, cột bằng 0 và chỉnh kích thước lên lại (28, 28)

Với phương pháp này ta sẽ sử dụng dữ liệu hai tập *train* và *test* sau khi đã qua hai bước loại bỏ các dòng, cột bằng 0 và chỉnh kích thước lên lại $(28, 28)$, cũng như qua bước chuẩn hóa dữ liệu như thông thường. Và bước cuối cùng là tương tự như 2 thử nghiệm trên không cần phải nhắc lại:

```
Train score with L2 penalty: 0.9476
Test score with L2 penalty: 0.9373
```

Như quan sát độ chính xác đạt được, có thể thấy với bước xử lý này thì mô hình của chúng ta đã tốt hơn rất nhiều, không ngoài dự kiến. Chính vì vậy ta sẽ giữ lại bước này coi như là một bước tiền xử lý chính cho mô hình cuối cùng được xây dựng. Và đương nhiên là sẽ bỏ đi phương pháp ở trên.

6.1.4 Thử nghiệm 4: Huấn luyện mô hình sau khi PCA

Như đã trình bày ở phần ý tưởng cho các bước tiền xử lý. Phương pháp này được coi là một trong những phương pháp đơn giản mà hiệu quả để giải quyết hiện tượng đa cộng tuyến. Và ta sẽ chọn nó để giải quyết hiện tượng này cho bài toán của chúng ta.

Về tham số:

- **n_components** = 600: Tạm thời ta sẽ để tham số này có giá trị 600, ta phải để số cột còn dưới 80% nhằm để mô hình mặc định cho tham số **svd_solver**=*'randomized'*, phù hợp và hoạt động tốt cho dữ liệu của ta.
- **random_state** = 0: Tham số này nên được để cố định.

Sau đó vẫn là các bước như cũ, và ta được kết quả:

```
Train score with L2 penalty: 0.9096
Test score with L2 penalty: 0.9080
```

Quan sát thấy độ chính xác trên cả hai tập không tăng thậm chí còn giảm đáng kể so với mô hình gốc (chưa qua bước tiền xử lý nào) được thử nghiệm ở Thử nghiệm 1. Tuy nhiên điều này cũng dễ hiểu là do khi qua giảm chiều thì lượng thông tin sẽ mất mát đi một chút. Và bước này chủ yếu là để loại bỏ hiện tượng đa cộng tuyến, tăng độ tin cậy của mô hình. Thêm vào đó việc giảm chiều bằng *PCA* giúp cho thời gian thực thi giảm đi đáng kể.

6.1.5 Thử nghiệm 5: Kết hợp thử nghiệm 3 với 4

Ở bước này ta sẽ kết hợp hai quá trình tiền xử lý ở hai thử nghiệm 3 và 4 lại để đánh giá độ hiệu quả. Đó sẽ gồm các bước là xóa tất cả dòng cột bằng 0, chuẩn hóa dữ liệu, đưa dữ liệu đã chuẩn hóa qua *PCA* và cuối cùng là đi vào mô hình. Ta được kết quả như sau:

```
Train score with L2 penalty: 0.9490
Test score with L2 penalty: 0.9369
```

Ta thấy với sự kết hợp này, nếu so sánh với việc chỉ dùng mỗi *PCA* thì kết quả đạt được là tốt hơn rất nhiều, điều này có vẻ là đến từ sự hiệu quả của bước xử lý ở Thử nghiệm 3. Và ngược lại nếu so với kết quả ở Thử nghiệm 3 thì sự thay đổi gần như là bằng không. Điều này tương đối dễ hiểu, đã được giải thích ở Thử nghiệm 4. Nhưng *PCA* thật sự quan trọng, không những làm giảm đáng kể thời gian thực thi, mà còn đóng vai trò quan trọng trong việc triển khai bước Thử nghiệm 6.

6.1.6 Thử nghiệm 6: Huấn luyện mô hình như ở thử nghiệm 4 sử dụng ảnh đã cắt hết dòng, cột bằng 0 nhưng sử dụng thêm Polynomial Features

Như đã trình bày ở phần ý tưởng của các bước tiền xử lý thì thuật toán *Polynomial Features* có vẻ sẽ đem lại độ hiệu quả rất lớn khi áp dụng vào mô hình của chúng ta. Tuy nhiên đi kèm với đó có một vấn đề là nếu áp dụng thuật toán này lên tập dữ liệu ban đầu (có 784 cột thuộc tính) với các tham số được nêu ở phần ý tưởng (đã làm giảm tối đa sự phức tạp của thuật toán), thì số thuộc tính của dữ liệu đầu ra sẽ là rất lớn. Và ta sẽ đạt giới hạn bộ nhớ khi thực thi thuật toán này. Chính vì vậy nhờ vào việc giảm chiều từ thuật toán *PCA* thì việc triển khai thuật toán này với tập dữ liệu của chúng ta trở nên khả thi.

Ở thực nghiệm này ta sẽ cho dữ liệu đi qua hết tất cả các bước tiền xử lý được chọn ở trên, với các tham số đã được xác định. Có một lưu ý là tham số **n_components** của thuật toán *PCA* ở đây phải được giảm xuống thật nhỏ để đảm bảo được yêu cầu của thuật toán *Polynomial Features* được trình bày ở trên. Qua quá trình thử nghiệm thì giá trị 70 được cho là thích hợp và đảm bảo được đủ nhu cầu nên từ bước này ta sẽ để 70 là giá trị sử dụng cho hàm **PCA()**. Ta có kết quả đạt được như sau:

```
Train score with L2 penalty: 0.9993
Test score with L2 penalty: 0.9820
```

Đến đây thì độ chính xác của mô hình đã được cải thiện rất nhiều so với lúc đầu. Điều này cho thấy sự hiệu quả của quá trình tiền xử lý dữ liệu (bao gồm tất cả các bước cho tới giờ) và đặc biệt là sự hiệu quả và phù hợp của thuật toán *Polynomial Features* đối với tập dữ liệu này. Cụ thể là độ chính xác trên tập *train* đã lên tới gần 100% và trên tập *test* là 98.2%.

Tới đây ta tự hỏi liệu còn cách nào để tăng độ hiệu quả của mô hình thêm nữa không? Khi mà độ chính xác đã tăng gần tới tối đa và việc tăng thêm là càng khó hơn. Điều này khiến ta quay lại nhìn vào dữ liệu ban đầu của bài toán, và tiến hành Thử nghiệm 7.

6.1.7 Thử nghiệm 7: Sử dụng Data Augmentation kết hợp thử nghiệm 6

Data Augmentation thật ra là cách gọi chung của các thuật toán tăng cường dữ liệu, một trong những cách hiệu quả để cải thiện mô hình học. Ở đây ta sẽ sử dụng một thuật toán phù hợp với kiểu dữ liệu là chữ số viết

tay của tập *MNIST*, đó là Xoay ảnh (Đã được trình bày ở phần ý tưởng các bước tiền xử lý). Bước này sẽ được đặt ở đầu tiên, tức là ta sẽ thực hiện xoay ảnh cho tập dữ liệu *train* với mỗi ảnh được xoay 5 lần. Vậy dữ liệu sau sẽ có tổng cộng là $300,000$ bức ảnh. Sau đó ta lại đưa vào tuần tự các bước tiền xử lý như ở bước Thử nghiệm 6 và quan sát kết quả:

```
Train score with L2 penalty: 0.9988
Test score with L2 penalty: 0.9849
```

Và đến đây ta có thể thấy mô hình đã cải thiện được độ chính xác rất đáng kể trên tập *test* (vì đã cao sẵn nên việc cao thêm nữa là rất khó so với lúc đầu). Trên tập *train* có giảm nhưng là gần như không đáng kể, điều này dễ hiểu là vì dữ liệu tập *train* đã nhân lên gấp 5 lần. Vậy có thể thấy bước xử lý này đem lại độ hiệu quả rất tốt.

Trên đây là tất cả các bước tiền xử lý đã được thử nghiệm và triển khai. Bây giờ chúng ta sẽ tổng hợp lại hết để được một mô hình hoàn chỉnh.

6.2 Tổng kết mô hình và xây dựng Pipeline hoàn chỉnh của mô hình

Đầu tiên chúng ta sẽ tổng kết lại lần lượt các bước tiền xử lý được chọn:

- Xoay ảnh
- Cắt ảnh (Xóa tất cả dòng cột chứa toàn số 0) và điều chỉnh lại kích cỡ thành $(28, 28)$
- Thuật toán *PCA*
- Thuật toán *Polynomial Features*
- Chuẩn hóa dữ liệu bằng thuật toán *Standard Scaler*

Và cuối cùng là đưa vào mô hình *Logistic Regression* để dự đoán.

Ta xây dựng một *Pipeline* hoàn chỉnh cho mô hình của chúng ta với các tham số đã được chọn phù hợp như sau:

```
pipeline = Pipeline([('image_filter', Image_Filter()),
                     ('pca', PCA(n_components= 70, svd_solver="randomized", random_state = 0)),
                     ('poly', PolynomialFeatures(include_bias = True, interaction_only= True)),
                     ('scaler', StandardScaler()),
                     ('classifier', LogisticRegression(penalty='l2', solver='saga', tol=0.01, random_state = 0))])
```

Ta xem xét một chút về dữ liệu đầu vào của *pipeline* cũng như là dữ liệu đầu vào của hàm **Logistic Regression()** ở bước cuối của *pipeline* (sau khi đã qua hết các bước tiền xử lý dữ liệu ở trước).

Đầu vào của *pipeline* sẽ là một mảng 2 chiều với các cột là số cột thuộc tính (bắt buuộc phải là $784 = 28 \times 28$). Ví dụ như khi truyền tập *X_ test* vào thì *X_ test* lúc đầu sẽ có kích thước là $(10000, 784)$. Sau đó qua lần lượt các bước trong *pipeline* thì dữ liệu sẽ biến đổi như sau:

- 'image_filter': Không thay đổi kích thước.
- 'pca': Kích thước sẽ còn là $(10000, 70)$.
- 'poly': Kích thước sẽ tăng lên thành $(10000, 2486)$.
- 'scaler': Kích thước không đổi.

Do đó khi tới bước 'classifier' thì kích thước của tập *X_ test* sẽ là $(10000, 2486)$. Vậy ta có thể kết luận dữ liệu trước khi vào hàm **Logistic Regression()** ở bước cuối trong *pipeline* sẽ có kích thước cột là 2486 còn số dòng dữ liệu chỉ là số bức ảnh sẽ được giữ nguyên như ban đầu truyền vào. Và đầu ra của mô hình hay nói cách khác là của *pipeline* sẽ là xác xuất trên từng lớp của tất cả các bức ảnh trong tập dữ liệu đưa vào.

Sau bước này chúng ta sẽ thực hiện lưu *pipeline* của mô hình lại thành một *file* có tên là **pipeline_LR.sav** để tiện dùng cho sau này.

6.3 Đánh giá mức độ hoàn thiện và độ chính xác của mô hình

Đầu tiên ta sẽ đánh giá mức độ hoàn thiện của mô hình bằng dữ liệu tập *train* và *test* ở bước Thử nghiệm 7. Độ chính xác của hai tập đã được đề cập ở trên. Ta sẽ đi xem các thông số khác cụ thể hơn cũng như bảng *Confusion Matrix*.

- Đối với tập *train*:

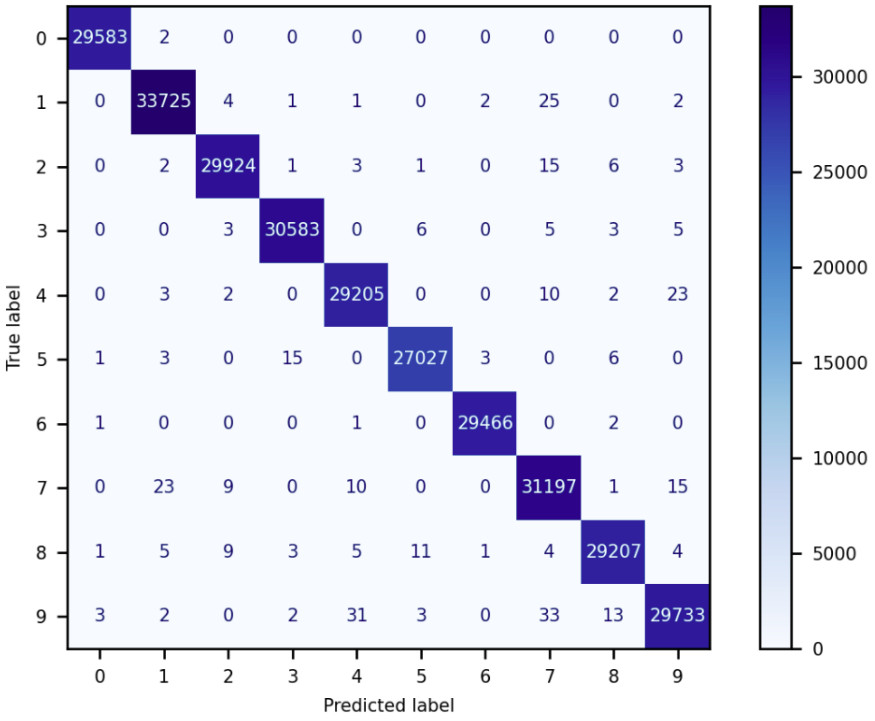
- Độ chính xác của mỗi lớp trong tập train

```
In [57]: acc_each_label = Confusion_Matrix.diagonal()/ np.sum(Confusion_Matrix, axis =0)*100
         acc_each_label

Out[57]: array([99.97972219, 99.88153413, 99.90985276, 99.92811632, 99.82567678,
                99.92236025, 99.97964169, 99.70596695, 99.8871409 , 99.82541548])
```

```
In [65]: print(classification_report(y_train, predict_train_y, labels=np.arange(10), digits=5))
```

	precision	recall	f1-score	support
0	0.99980	0.99993	0.99986	29585
1	0.99882	0.99896	0.99889	33760
2	0.99910	0.99897	0.99903	29955
3	0.99928	0.99928	0.99928	30605
4	0.99826	0.99863	0.99844	29245
5	0.99922	0.99897	0.99909	27055
6	0.99980	0.99986	0.99983	29470
7	0.99706	0.99814	0.99760	31255
8	0.99887	0.99853	0.99870	29250
9	0.99825	0.99708	0.99767	29820
micro avg	0.99883	0.99883	0.99883	300000
macro avg	0.99885	0.99884	0.99884	300000
weighted avg	0.99883	0.99883	0.99883	300000



- Đối với tập *test*:

- Độ chính xác của mỗi lớp trong tập test

```
In [59]: acc_each_label = Confusion_Matrix.diagonal()/ np.sum(Confusion_Matrix, axis =0)*100
         acc_each_label

Out[59]: array([99.69325153, 98.16112084, 98.00598205, 98.91625616, 98.46938776,
                98.6636971 , 98.38220425, 98.28408008, 98.01876955, 98.37892604])
```

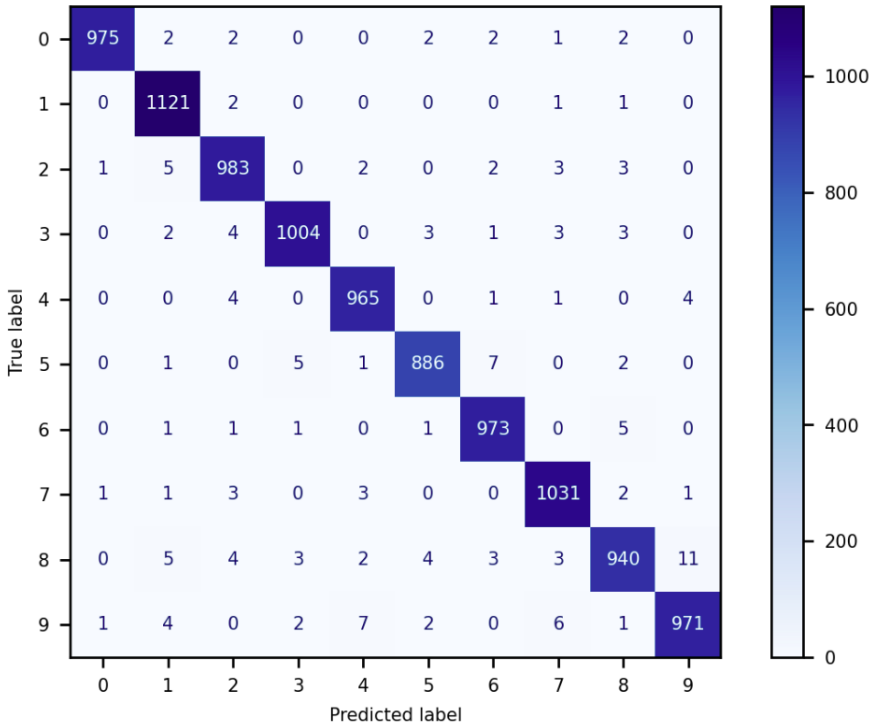
```
In [64]: from sklearn.metrics import classification_report
         import warnings
         warnings.simplefilter(action='ignore', category=FutureWarning)

         print(classification_report(y_test, predict_test_y, labels=np.arange(10), digits=5))
```

```
              precision    recall  f1-score   support

           0    0.99693   0.98884   0.99287       986
           1    0.98161   0.99644   0.98897      1125
           2    0.98006   0.98398   0.98202       999
           3    0.98916   0.98431   0.98673      1020
           4    0.98469   0.98974   0.98721       975
           5    0.98664   0.98226   0.98444       902
           6    0.98382   0.99084   0.98732       982
           7    0.98284   0.98944   0.98613      1042
           8    0.98019   0.96410   0.97208       975
           9    0.98379   0.97686   0.98031       994

   micro avg    0.98490   0.98490   0.98490     10000
   macro avg    0.98497   0.98468   0.98481     10000
weighted avg    0.98491   0.98490   0.98489     10000
```



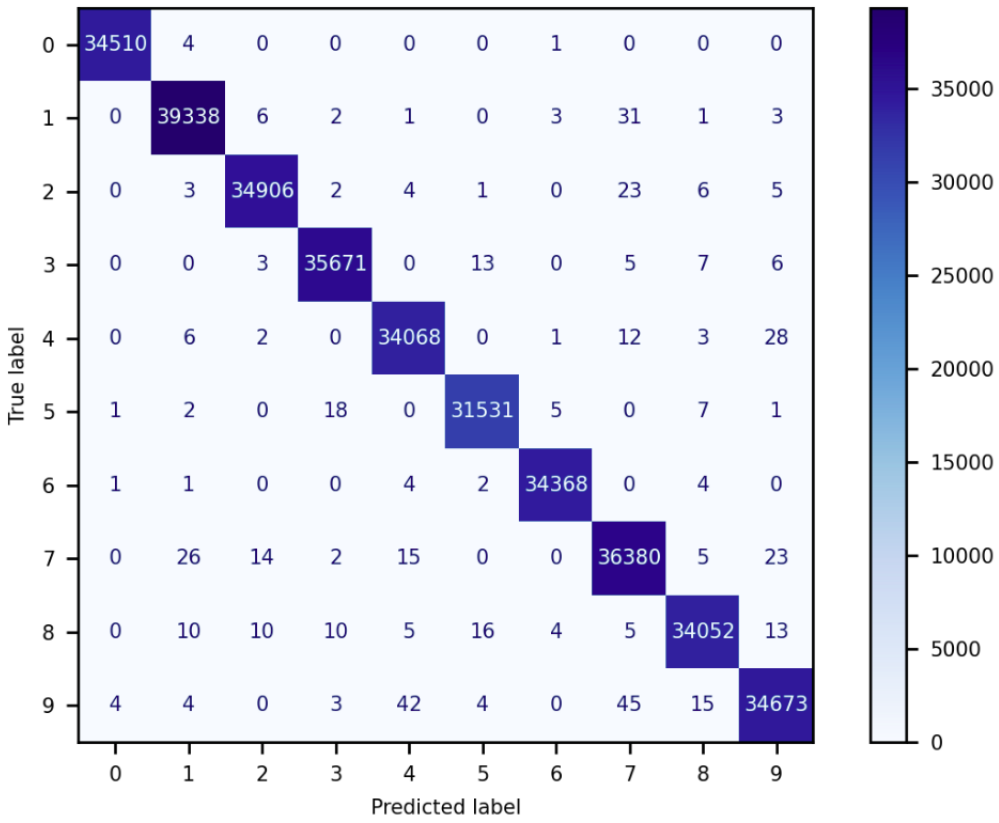
Và ở bước cuối cùng, sau khi đã xác định được cấu trúc của mô hình cần xây dựng như trên, thì ta sẽ gộp toàn bộ dữ liệu lại như ban đầu của tập *MNIST* rồi tiến hành huấn luyện để được mô hình hoàn chỉnh và tốt nhất để sử dụng cho sau này. Kết quả đạt đượcc với độ chính xác của mô hình là:

```
Train score with L2 penalty: 0.9986
```

Các thông số khác cũng như bảng *Confusion Matrix*:

```
print(classification_report(y, predict_y, target_names=label, digits=6))
```

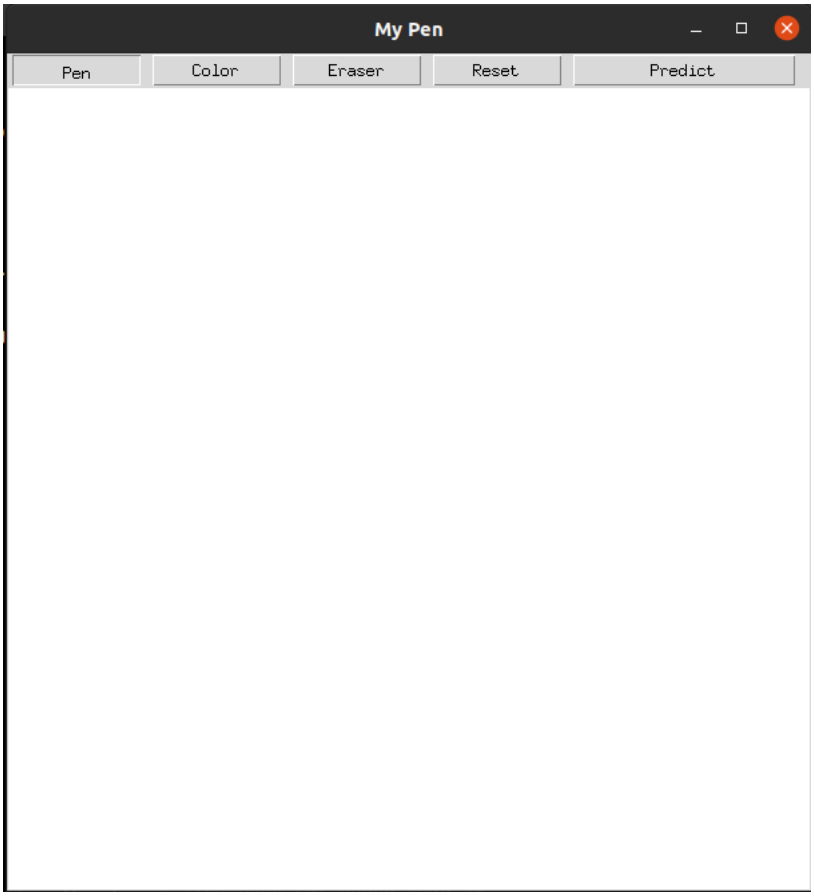
	precision	recall	f1-score	support
0	0.999826	0.999855	0.999841	34515
1	0.998578	0.998807	0.998693	39385
2	0.998998	0.998741	0.998870	34950
3	0.998964	0.999048	0.999006	35705
4	0.997920	0.998476	0.998198	34120
5	0.998860	0.998923	0.998891	31565
6	0.999593	0.999651	0.999622	34380
7	0.996685	0.997669	0.997177	36465
8	0.998592	0.997861	0.998226	34125
9	0.997727	0.996637	0.997182	34790
accuracy			0.998563	350000
macro avg	0.998574	0.998567	0.998570	350000
weighted avg	0.998563	0.998563	0.998563	350000



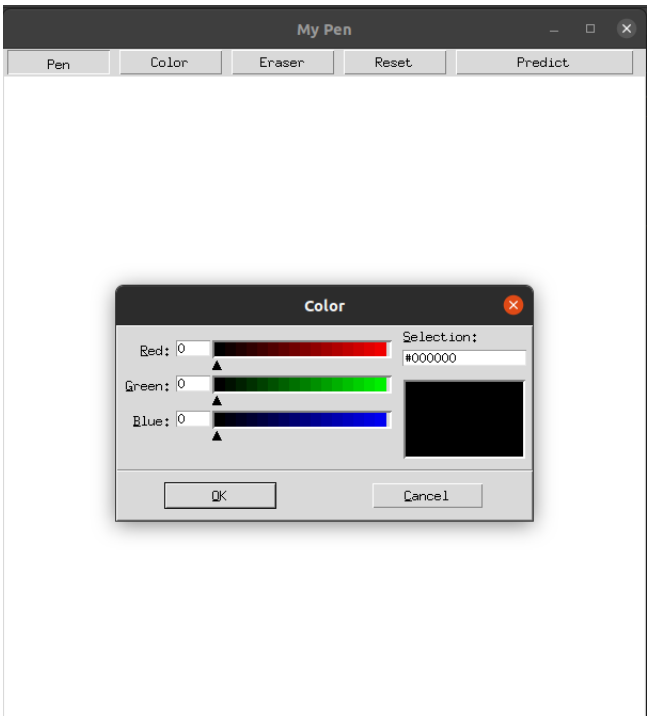
7 Thử nghiệm thực tế

7.1 Giới thiệu phần mềm và cách sử dụng

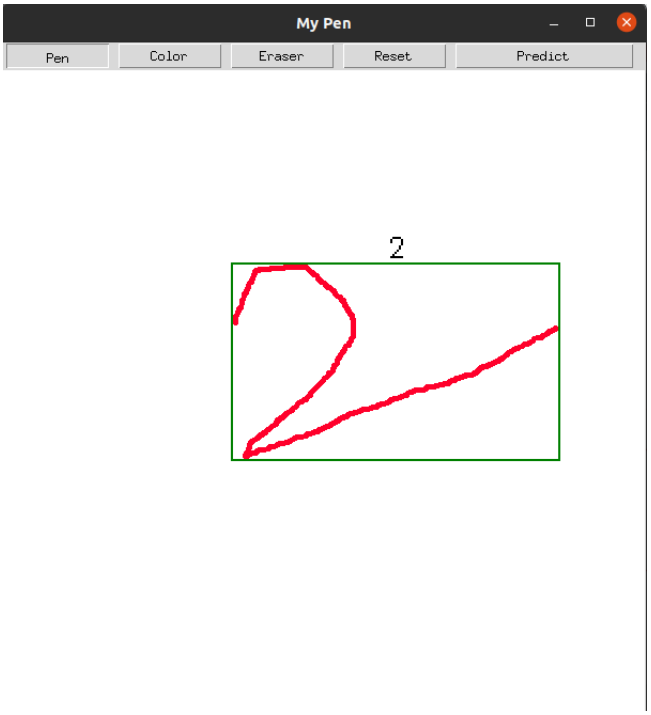
Trong phần này, ta sẽ xây dựng một phần mềm phục vụ nhu cầu có thể dùng chuột viết chữ số lên *Canvas* và dự đoán. Để phục vụ nhu cầu thực tế, ta sẽ nâng cấp tính năng của ứng dụng hơn so với yêu cầu, tức là cho phép người dùng có thể viết cùng lúc nhiều chữ số lên cửa sổ ứng dụng, và có thể dự đoán kết quả bất cứ lúc nào. Cụ thể chúng ta sẽ làm quen với ứng dụng qua những hình ảnh mô tả sau:



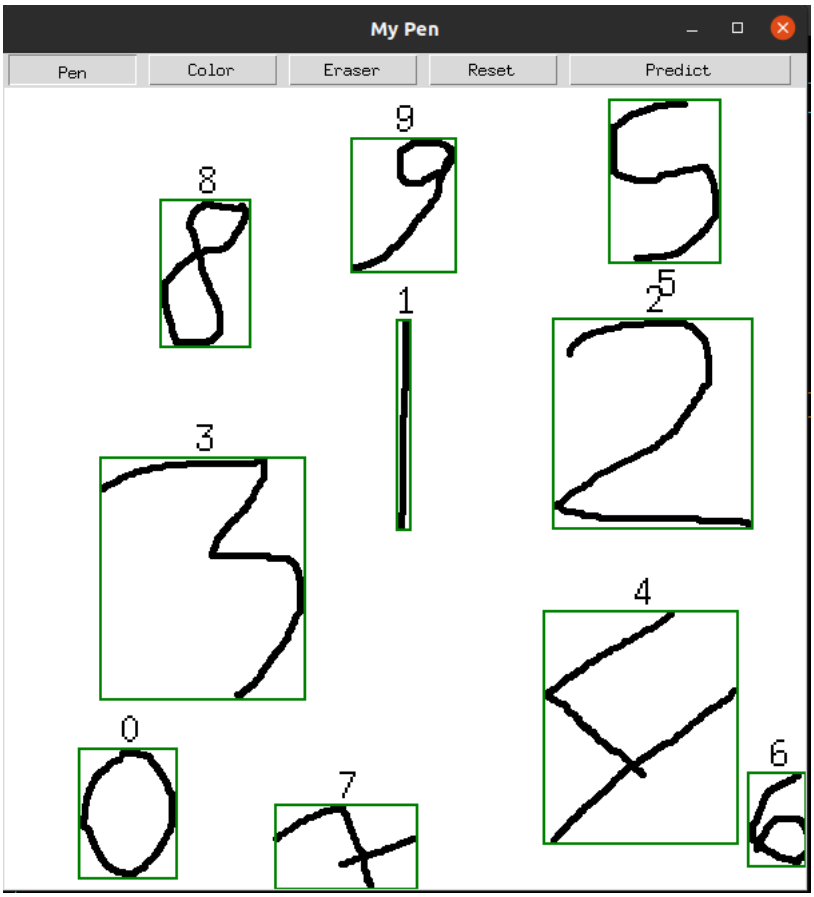
Ở giao diện khởi động của ứng dụng, ta sẽ thấy một cửa sổ đơn giản hiện lên với phần nền trắng là khu vực để vẽ hình, kèm theo ở trên là thanh công cụ chứa các nút lệnh hỗ trợ như: *Pen, Color, Eraser, Reset* và *Predict*. Có thể để ý là khi ta chọn một nút lệnh nào thì phần khung nút lệnh đó sẽ được đè lên (như hình) để tránh việc nhầm lẫn. Khi nhấn vào nút *Color* thì một bảng màu như sau sẽ hiện lên:



Đây là hệ thống màu *RGB* quen thuộc, người dùng sẽ chọn màu từ thích cho bút vẽ. Trong trường hợp vẽ nét không mong muốn, công cụ *Eraser* sẽ hỗ trợ việc tẩy xóa đi nét đó. Nút lệnh *Reset* để đưa phần mềm về với tình trạng như ban đầu lúc mới khởi động. Và cuối cùng sau khi vẽ xong các con số mong muốn ta chỉ việc bấm nút *Predict* để gắn nhãn con số đó, bằng việc sử dụng mô hình đã được xây dựng ở phần trước. Ta sẽ thử vẽ một con số và dự đoán:



Như ta thấy, ở đây với một số 2 được viết bằng nét bút màu đỏ và cách viết khác bình thường một chút, thì chương trình vẫn dễ dàng nhận biết được. Chương trình sẽ tự động kẻ một hình chữ nhật nhỏ nhất bao sát con số được vẽ và in một nhãn ngay phía trên hình chữ nhật đó, trong trường hợp hình chữ nhật nằm sát biên trên thì nhãn sẽ được in phía dưới. Bây giờ ta thử vẽ kết hợp một lúc nhiều số trên *canvans*, các số sẽ được vẽ nghiêng ngã và khó nhìn, để xem liệu chương trình có dự đoán và phân lớp chính xác được hay không.



Trên hình là đầy đủ các con số từ $0-9$ với cách nhiều kiểu viết gây khó khăn cho chương trình, nhưng với có vẻ như chương trình đã tách, nhận biết và phân loại các con số rất tốt khi các nhãn đã khớp hết với những con số được viết. Ở đây ta có thể nhận thấy chương trình đã làm tốt được những nhiệm vụ như: phân tách các con số trên cùng một bức hình, xử lý các trường hợp khó như số được viết sát biên, số viết nghiêng ngã; Và cho kết quả tốt nhất.

Ở phần tiếp theo sẽ trình bày cụ thể chương trình được thiết kế như thế nào để xử lý các con số, cũng như bước áp dụng mô hình được xây dựng ở phần trước vào chương trình như thế nào để ra được kết quả gắn nhãn cuối cùng.

7.2 Các bước xử lý cụ thể của chương trình

Chương trình sẽ gồm 4 *file* chính: *Main.py, Class_ Paint.py, Class_ ProcessImage.py* và *Function.py*. Trong đó sẽ có 2 *file* chứa 2 *class* chính là *class* **Paint()** để thực hiện việc tạo giao diện ứng dụng và hỗ trợ vẽ hình cũng như trả ra kết quả dự đoán cuối cùng của chương trình và *class* **ProcessImage()** để hỗ trợ việc cắt ghép các con số trên hình và các bước xử lý (bao gồm căn chỉnh và đổi màu) để dữ liệu phù hợp với đầu vào mô hình được xây dựng. *File Function.py* bao gồm hai hàm con nhằm hỗ trợ tốt cho việc xử lý của *class* **ProcessImage()**. Sau đây ta sẽ là cấu trúc sơ lược về 2 *class* chính của chương trình.

7.2.1 Class Paint()

Trong *class* này chủ yếu là *code* để thiết kế giao diện cũng như các thao tác trên giao diện (bao gồm các nút) của phần mềm, như đã mô tả ở trên. Ở đây chỉ có một hàm quan trọng dùng để dự đoán (gắn nhãn) các chữ số là hàm **use_predict(self)**.

```
def use_predict(self):
    self.activate_button(self.predict_button)
    self.predict_button.config(relief=RAISED)
    ######
    self.np_image = np.array(self.image)
    img_process = ProcessImage(self.np_image, mode=1)
    img_process.find_digits()
    pipeline = pickle.load(open('pipeline_LR.sav', 'rb'))
    for i in range(len(img_process.list_info)):
        info_img = img_process.list_info[i]
        x0 = info_img[0]
        y0 = info_img[1]
        x1 = info_img[2] + x0 + 1
        y1 = info_img[3] + y0 + 1
        self.c.create_rectangle(x0, y0, x1, y1, fill="", outline='green', width = 2)
        text = pipeline.predict([img_process.list_digits[i]])[0]
        if y0 - 15 > 5:
            self.c.create_text((x0+x1)/2-5, y0 - 15, anchor=W, font=("Purisa",20),text=text)
        else:
            self.c.create_text((x0+x1)/2-5, y1 + 15, anchor=W, font=("Purisa",20),text=text)
    self.activate_button(self.pen_button)
```

Ở hàm này, khi được kích hoạt (bằng cách nhấn nút *Predict* trên thanh công cụ), việc đầu tiên là chương trình sẽ đọc thông tin toàn bộ bức hình (chứa tất cả các chữ số) rồi chuyển qua thành dạng *numpy array*. Sau đó dữ liệu này sẽ được truyền vào *class* **ProcessImage()**, qua hàm **find_digits()** (sẽ được trình bày kĩ ở phần tiếp theo), để nhận lại dữ liệu cuối cùng là một biến chứa thông tin của tất cả các con số cần được dự đoán (gắn nhãn), đã qua tất cả các bước tiền xử lý để khớp với đầu vào mô hình của chúng ta.

Bước tiếp theo ta sẽ đọc dữ liệu của *pipeline* được xây dựng từ trước và lưu lại dưới dạng *file* bao gồm các bước tiền xử lý cũng như mô hình đã được huấn luyện sẵn. Bước cuối cùng, ta chỉ cần duyệt qua thông tin của lần lượt các con số rồi đẩy vào *pipeline* để thu lại nhãn phù hợp với con số đó. Sau đó thực hiện vẽ hình chữ nhật bao quanh số đó, và gắn nhãn trên hình cho số (như đã được mô tả ở phần giới thiệu ở trên).

7.2.2 Class ProcessImage()

Như đã trình bày ở trên, *class* này sẽ bao gồm các bước xử lý hình ảnh được vẽ cho phù hợp với đầu vào của mô hình trước khi đưa vào dự đoán. Ở *class* sau hàm khởi tạo, ta chỉ tạo một hàm duy nhất là **find_digits(self)** để tìm cũng như xử lý những con số trong hình.

```
gray = cv.cvtColor(self.image, cv.COLOR_BGR2GRAY)
blurred = cv.GaussianBlur(gray, (5, 5), 0)
edged = cv.Canny(blurred, 50, 200, 255)

cnts = cv.findContours(edged.copy(), cv.RETR_EXTERNAL,
|   |   |   |   cv.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)
```

Ở bước đầu ta sẽ thực hiện việc đổi màu cho bức hình cũng như màu viền của các con số để dễ dàng xác định khung chữ nhật bao những con số hơn. Sau đó sử dụng hàm **findContours()** của thư viện **cv2** để xác định các khung hình chữ nhật bao các con số trong hình và lưu thông tin vào biến **cnts**.

Bước tiếp theo và cũng là bước cuối cùng của hàm này là duyệt qua tất cả các hình chữ nhật chứa các hình vẽ đã được lưu thông tin trong biến **cnts** để lấy và xử lý các thông tin của con số nhằm tiến hành dự đoán.

```
self.list_info = [] # Danh sách thông tin của các con số trong ảnh gốc, mỗi phần tử là một list [x, y, w, h]
self.list_digits = [] # Danh sách ảnh được cắt ra

# Vòng lặp qua các phần tử được xác định trong ảnh gốc
for c in cnts:
    # Tạo hình chữ nhật bao quanh các vùng ảnh được vẽ
    (x, y, w, h) = cv.boundingRect(c)
    cv.rectangle(self.image, (x, y), (x + w+1, y + h+1), (0, 255, 0), 3)

    # Với mỗi ảnh xét xem liệu nó có phân biệt với các ảnh đã cắt hay không
    i = 0
    while i < len(self.list_digits):
        if CheckSeparated(self.list_info[i][0], self.list_info[i][1], self.list_info[i][2], self.list_info[i][3], x, y, w, h) == False:
            # Nếu nó trùng với một ảnh nào đã cắt nhưng kích thước lớn hơn thì sẽ quan trọng hơn, dùng nó thay thế ảnh cũ
            if w*h > self.list_info[i][2] * self.list_info[i][3]:
                split_img = AdjustImage(self.image, x, y, w, h)
                self.list_digits[i] = split_img
                self.list_info[i] = [x, y, w, h]
            break
        i += 1

    # Nếu phân biệt thì thêm nó như một ảnh mới vào danh sách
    if i == len(self.list_digits):
        split_img = AdjustImage(self.image, x, y, w, h)
        self.list_digits.append(split_img)
        self.list_info.append([x, y, w, h])
```

Trong mỗi lần lấy thông tin một hình chữ nhật ra ta sẽ thu thập 4 giá trị (x, y, w, h) là 2 tọa độ và chiều rộng, chiều cao của hình chữ nhật đó. Từ đó ta sẽ xem xét một trường hợp phát sinh. Đó là khi ta viết một con số nhiều khi sẽ lỡ tay để dư ra một nét rời không quan trọng, hoặc là con số sẽ bị tách ra 2 hoặc 3 nét khác nhau trong lúc vẽ (vì các nét vẽ nhiều khi sẽ không dính liền nhau). Trong trường hợp đó ta sẽ sử dụng hàm **CheckSeparated()** được triển khai trong *file Function.py* để kiểm tra liệu chúng có trùng nhau không. Trùng ở đây tức là khoảng cách của chúng quá nhỏ để coi là 2 cá thể riêng biệt, và khi đó chương trình sẽ nhận biết 2 nét vẽ trong hình chữ nhật này được hiểu là của cùng một con số, và sẽ chọn hình chữ nhật có diện tích lớn nhất trong các hình chữ nhật ở đó để giữ lại được lượng thông tin lớn nhất để dự đoán. Thật ra trong hầu hết các trường hợp thì hình chữ nhật lớn nhất sẽ chính là hình chữ nhật bao tất cả các nét vẽ nên sẽ không gây mất mát bất cứ thông tin gì về dữ liệu. Chính bước xử lý này đã giúp cho chương trình tránh được lỗi đã nêu ra. Nếu 2 hình chữ nhật thật sự cách rời nhau đủ xa, thì đây là trường hợp đơn giản, tức là hai hình chứa hai con số khác nhau, ta sẽ đơn giản thêm hai con số đó vào danh sách các số cần gắn nhãn.

Sau các bước xử lý này, ta sẽ đẩy dữ liệu các khung ảnh chứa các con số vào hàm **AdjustImage()** cũng được triển khai trong *file Function.py* để thực hiện các bước xử lý dữ liệu cuối cùng, và thông tin trả về sẽ được lưu vào hai danh sách **list_info** để lưu tọa độ cũng như chiều rộng, chiều cao của khung chữ nhật nhằm vẽ ra khung lúc dự đoán (đã được demo ở trên) cho dễ nhìn cũng như xác định tọa độ trong hình để gắn nhãn cho chuẩn xác; và **list_digits** là danh sách thông tin của các con số cần dự đoán, đã khớp với đầu vào của mô hình.

Nói một chút về các bước xử lý trong hàm **AdjustImage()** ở *file Function.py*. Hàm này sẽ làm những bước xử lý như sau:

- Nới rộng khung chữ nhật bao quanh các chữ số, tạo điều kiện cho việc làm đậm nét.
- Tiến hành cắt khung chứa số.
- Cân chỉnh lại khung ảnh và đổi màu phù hợp với dữ liệu đưa vào mô hình
- Làm đậm nét chữ số giúp dữ liệu giống với tập *MNIST* hơn.
- Nén lại kích thước $(28, 28)$ rồi trải ra thành *array* $(784,)$ phù hợp với dữ liệu đưa vào mô hình.

Và cuối cùng ta được một ứng dụng hoàn chỉnh đáp ứng được những nhu cầu đề ra!

8 Tài liệu tham khảo.

- *Web Machine Learning cơ bản*
- *scikit-learn: Machine Learning in Python*
- *Multicollinearity | Detecting Multicollinearity with VIF*
- *Xử Lý Ảnh với OpenCV: Phóng To, Thu Nhỏ và Xoay Ảnh*