

PHÁT HIỆN VẾT NỨT TRÊN TƯỜNG BÊ TÔNG

CS231.022

Thực hiện: 22520240 Triệu Tấn Đạt

Mục lục

1. Lí do chọn đề tài

4. Dữ liệu

2. Phát biểu bài toán

5. Độ đo

3. Phương pháp

6. Kết quả

7. Minh họa



1. Lí do chọn đề tài

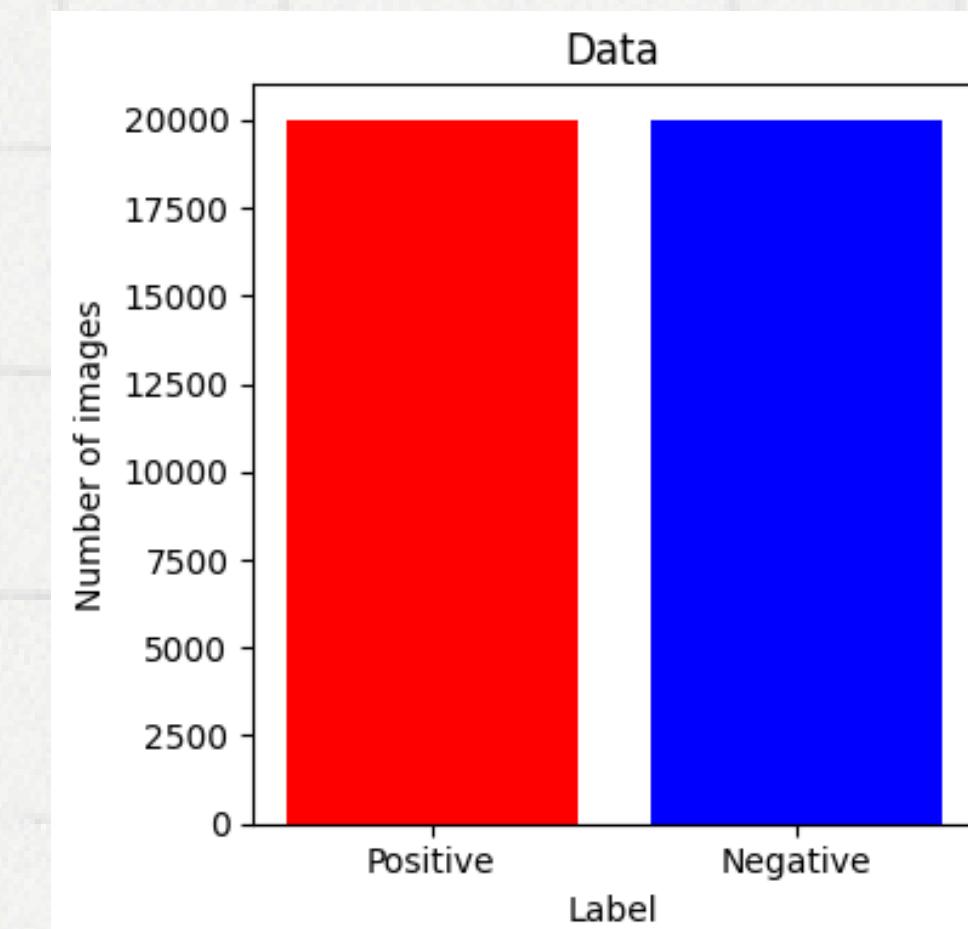
- Phát hiện vết nứt trên bề mặt tường bê tông là một vấn đề rất **quan trọng** trong cuộc sống hàng ngày. Bởi vì:
 - + **An toàn:** Trong lĩnh vực xây dựng nói chung, các vết nứt có thể gây mất an toàn cho con người. Phát hiện các vết nứt này sớm có thể ngăn chặn tai nạn và đảm bảo an toàn cho cộng đồng.
 - + **Tiết kiệm chi phí:** Phát hiện vết nứt sớm có thể ngăn chặn các sửa chữa hoặc thay thế tốn kém sau này.
 - + **Thẩm mỹ:** Vết nứt làm mất đi vẻ đẹp của bức tường, khiến công trình trông cũ kỹ và xuống cấp.

2. Phát biểu bài toán

- **Input:**

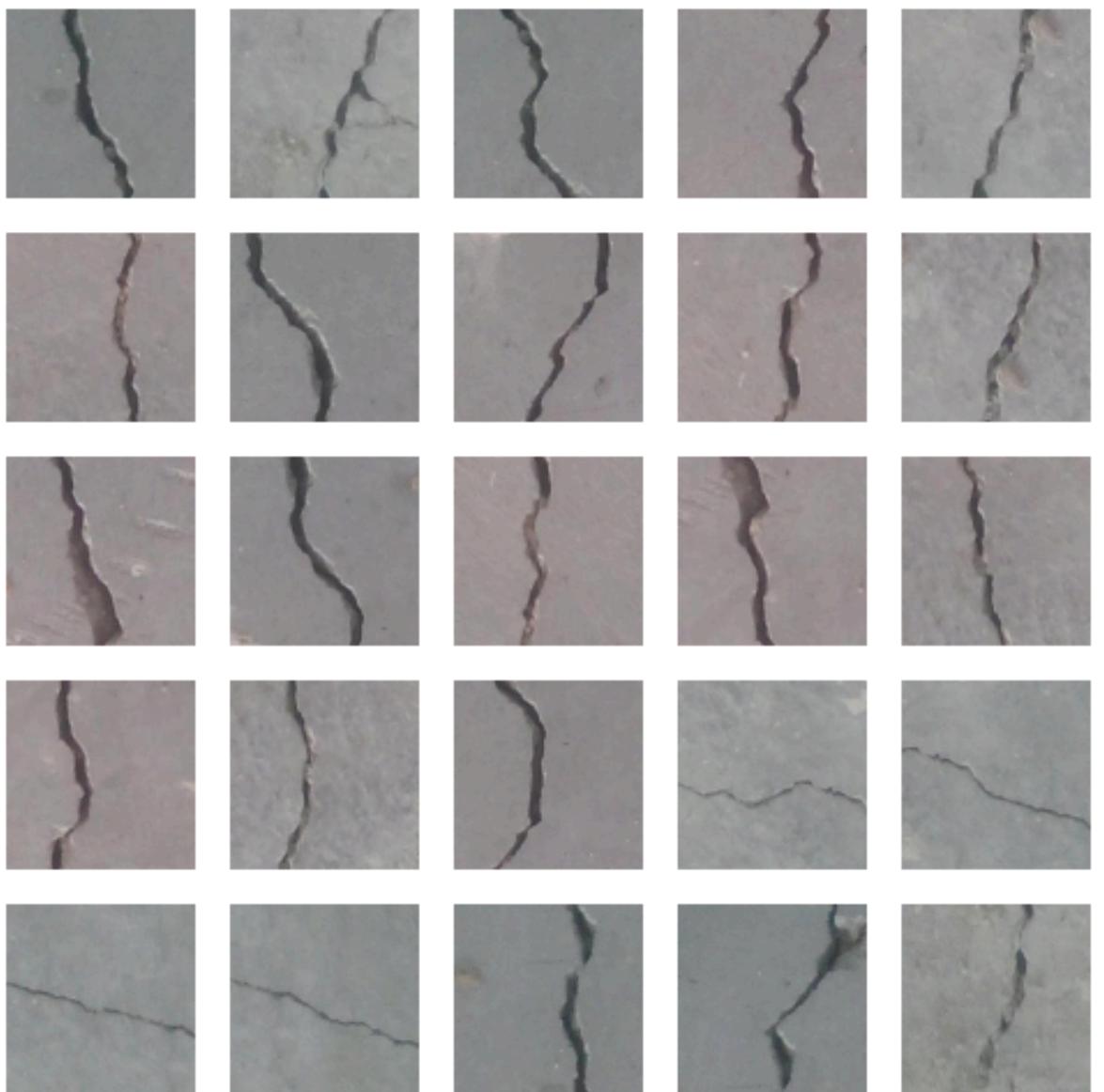
- + Data set : gồm 20000 ảnh có label Positive và 20000 ảnh có label Negative chia thành Train set, Validation set và Test set với tỉ lệ lần lượt là (6 : 2 : 2)
- + Ảnh số : ảnh chụp tường bê tông

- **Output :** Label (1 : Positive, 0 : Negative)

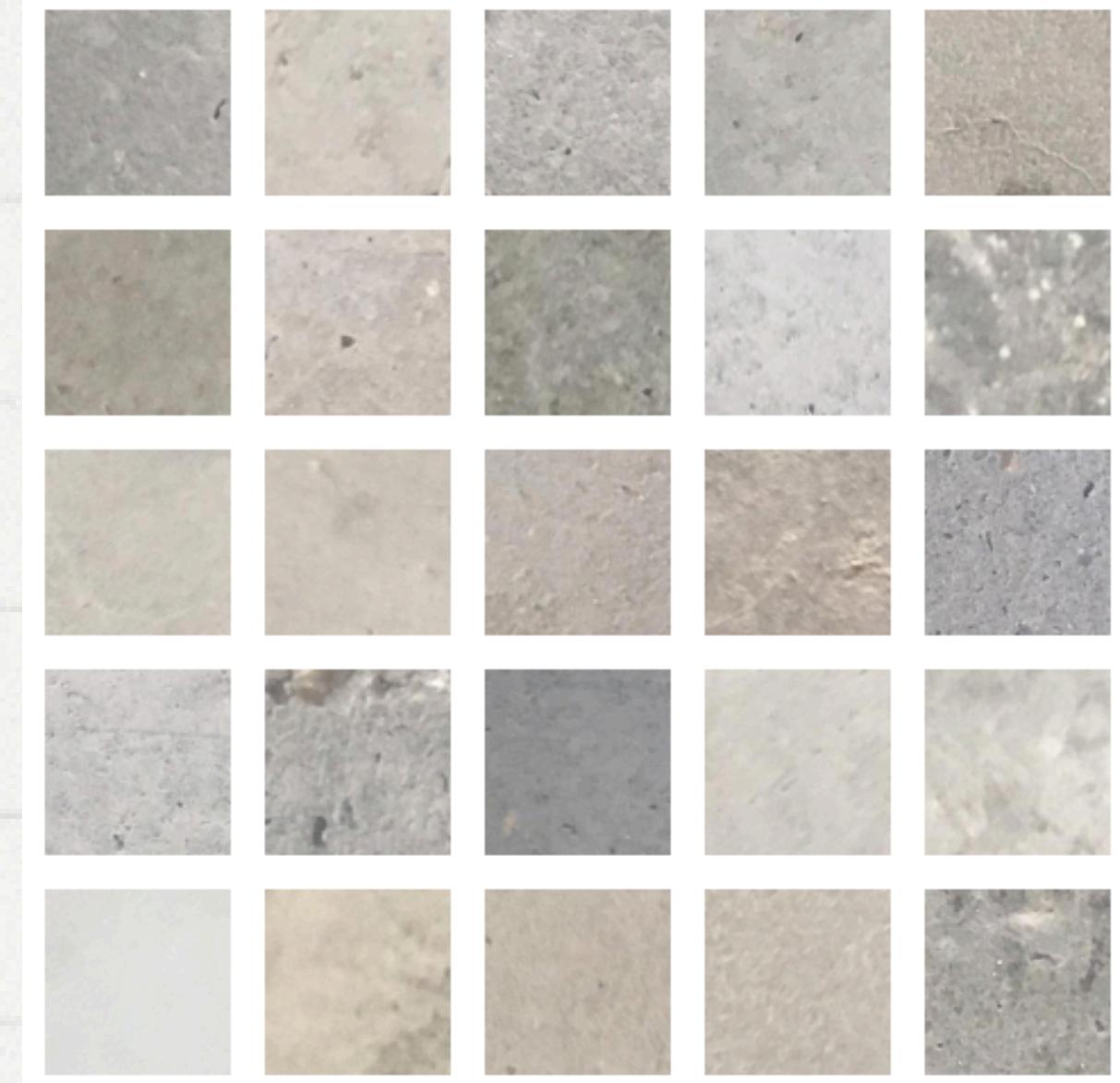


2. Phát biểu bài toán

- Data set:



Positive



Negative

3. Phương pháp

3.1 Giới thiệu

3.2 Ý tưởng chính

3.3 Cài đặt



3. Phương pháp

3.1 Giới thiệu

- **Logistic Regression:** là thuật toán phân loại học máy được sử dụng để dự đoán xác suất một biến thuộc vào một trong hai nhóm. Sự phát triển của thuật toán này là kết quả của nhiều đóng góp từ các nhà khoa học khác nhau trong suốt nhiều thế kỷ.

Link tham khảo: <https://machinelearningcoban.com/2017/01/27/logisticregression/>

3. Phương pháp

3.2 Ý tưởng chính

- Linear Regression:

+ Mô hình: $f(x) = w^T x + b$

+ Đầu ra: giá trị là số thực

+ Hàm mất mát: MSE
(Mean Squared Error)

- Logistic Regression:

+ Mô hình: $f(x) = \frac{1}{1 + e^{-(w^T x + b)}}$

+ Đầu ra: giá trị thuộc đoạn [0, 1] (nhờ vào hàm sigmoid)

+ Hàm mất mát : NLL
(Negative Log Likelihood)

3. Phương pháp

3.3 Cài đặt

- Các tham số:

```
def Logistic_Regression(X_train, y_train, learning_rate, epoch):
```

- + X_train: ảnh sau khi được xử lí, biểu diễn dưới dạng vector
- + y_train: label của từng ảnh, có giá trị 0 hoặc 1 và cùng số hàng với X_train

3. Phương pháp

3.3 Cài đặt

- Các tham số:

```
def Logistic_Regression(x_train, y_train, learning_rate, epoch):
```

- + learning_rate: bước nhảy mà mô hình thực hiện để điều chỉnh các tham số của mình trong mỗi lần cập nhật
- + epoch: số lần lặp để tìm ra các tham số

3. Phương pháp

3.3 Cài đặt

- Hàm sigmoid: $f(z) = \frac{1}{1+e^{-z}}$

```
def sigmoid(z): # z = wx + b
    return 1/(1 + np.exp(-z))
```

- Negative log likelihood:

$$L(w; \mathbf{x}, y) = -(y \log a + (1 - y) \log(1 - a))$$

(xét trên 1 điểm dữ liệu)

```
def loss(y, a): # a = sigmoid(z)
    return -1 * (y * np.log(a) + (1-y) * np.log(1-a))
```

3. Phương pháp

3.3 Cài đặt

```
def Logistic_Regression(X_train, y_train, learning_rate, epoch)
    # Khởi tạo w, b
    w = np.zeros(X_train.shape[1])
    b = 0.0
    for epoch in range(epoch):
        dw = np.zeros(w.shape)
        db = 0.0
        total_loss = 0.0
        for i in range(X_train.shape[0]):
            # Lấy ra dữ liệu thứ i
            x_i = X_train[i,:]
            y_i = y_train[i]
            # Trước khi tính Loss
            z_i = w.dot(x_i) + b
            a_i = sigmoid(z_i)
            # Loss của 1 điểm dữ liệu
            loss_i = loss(y_i, a_i)
            # Đạo hàm
            dw_i = x_i * (a_i - y_i) # dL / dw_i
            db_i = a_i - y_i # dL / db
            # Tổng đạo hàm, Loss
            dw += dw_i
            db += db_i
            total_loss += loss_i
        # Tính trung bình dw, db, total_loss
        dw = (1.0/X_train.shape[0]) * dw
        db = (1.0/X_train.shape[0]) * db
        # Gradient Descent
        w = w - learning_rate * dw
        b = b - learning_rate * db
    return w, b
```

4. Dữ liệu

4.1 Tìm dữ liệu

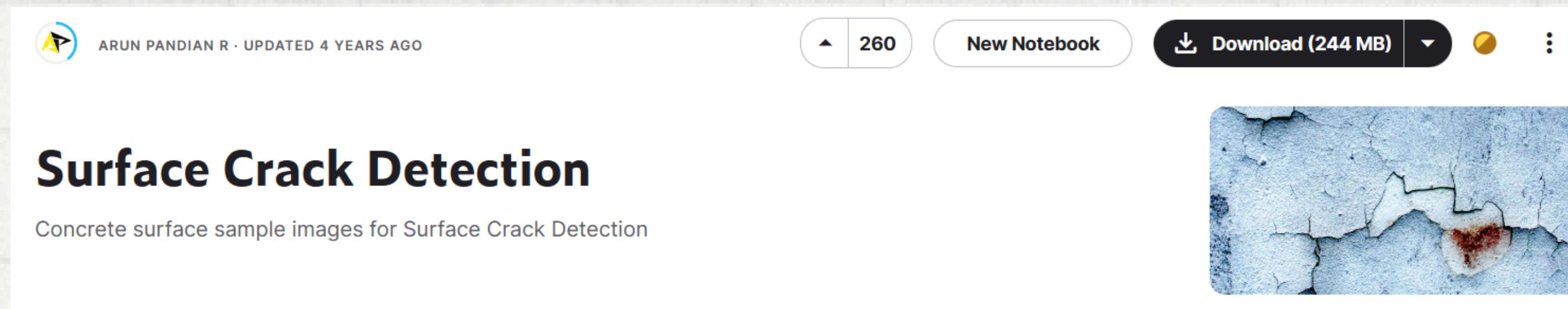
4.2 Tiền xử lý dữ liệu



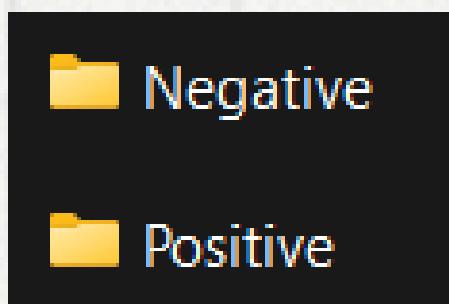
4.1 Tìm dữ liệu

- Tải xuống từ Kaggle:

<https://www.kaggle.com/datasets/arunrk7/surface-crack-detection>



- Gồm 2 folder Positive và Negative: mỗi folder gồm 20000 ảnh, kích thước 227x227



4.2 Tiền xử lí dữ liệu

- **Bước 1:** Đọc ảnh dưới dạng ảnh xám
- **Bước 2:** Chuyển ảnh về kích thước 120×120
- **Bước 3:** Làm mờ ảnh
- **Bước 4:** Áp dụng bộ lọc Sobel
- **Bước 5:** Áp dụng threshold
- **Bước 6:** Đưa về dạng vector
- **Bước 7:** Min-max Scaler

4.2 Tiền xử lý dữ liệu

- **Bước 1: Đọc ảnh dưới dạng ảnh xám**
 - + Giảm kích thước dữ liệu: Mỗi điểm ảnh trong một ảnh màu cần ba giá trị (R, G, B) để biểu diễn, trong khi mỗi điểm ảnh trong một ảnh xám chỉ cần một giá trị.
 - + Giảm độ phức tạp của xử lý

Code: `img = cv2.imread(os.path.join(path, img_name), cv2.IMREAD_GRAYSCALE)`

4.2 Tiền xử lý dữ liệu

- **Bước 2:** Chuyển ảnh về kích thước 120x120
 - + Đồng nhất kích thước đầu vào
 - + Giảm bớt tài nguyên tính toán: Ảnh lớn đòi hỏi nhiều tài nguyên để xử lý

Code: `resized_img = cv2.resize(img, (120, 120))`

4.2 Tiền xử lí dữ liệu

- **Bước 3: Làm mờ ảnh**
 - + Giúp giảm nhiễu
 - + Làm nổi bật các đặc trưng quan trọng và giúp mô hình tập trung vào thông tin chính yếu hơn.

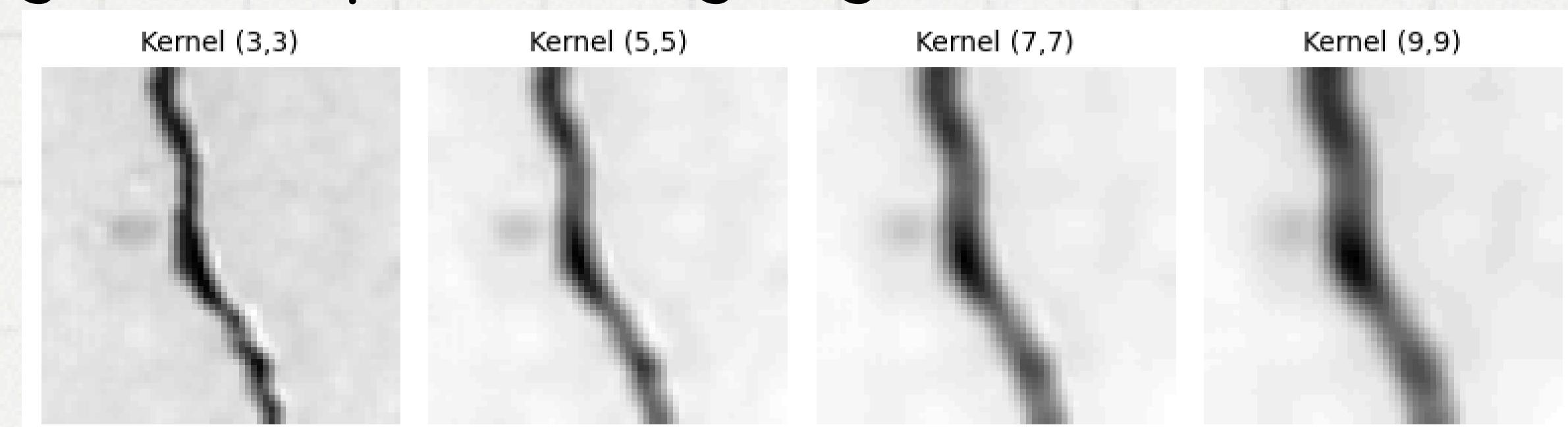
Code: `blured_img = cv2.GaussianBlur(resized_img, ksize=(9,9), sigmaX=0, sigmaY=0)`

4.2 Tiền xử lí dữ liệu

- Bước 3: Làm mờ ảnh

Code: `blured_img = cv2.GaussianBlur(resized_img, ksize=(9,9), sigmaX=0, sigmaY=0)`

+ ksize: Đây là kích thước của kernel (hoặc ma trận lọc). Kích thước này phải là một số lẻ và dương. Nó chỉ ra chiều rộng và chiều cao của kernel. Một kernel lớn hơn sẽ tạo ra hiệu ứng làm mờ mạnh hơn nhưng cũng có thể làm mất chi tiết



4.2 Tiền xử lí dữ liệu

- **Bước 4:** Áp dụng bộ lọc Sobel

+ Phát hiện cạnh

Code:

```
def sobel_filters(img):
    kernels = np.array([[-1,0,1],
                      [-2,0,2],
                      [-1,0,1]])
    return cv2.filter2D(img, -1, kernels)
```

```
sobelized_img = sobel_filters(blurred_img)
```

4.2 Tiền xử lí dữ liệu

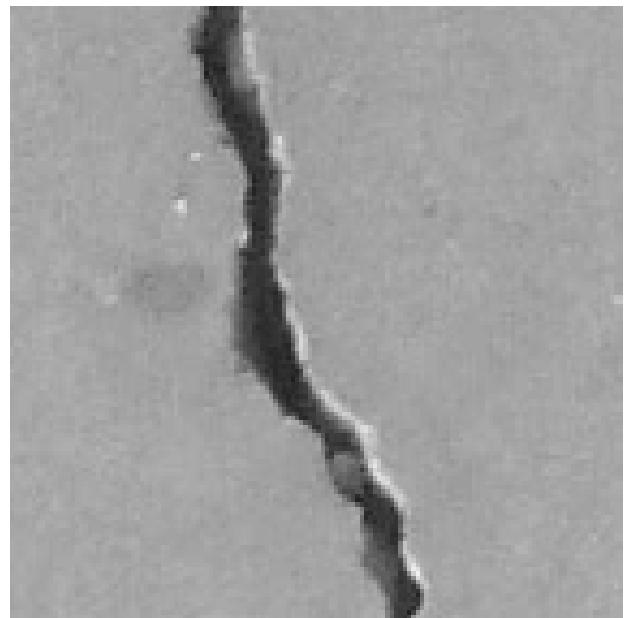
- **Bước 5: Áp dụng threshold**
 - + Làm nổi bật các đặc điểm quan trọng của hình ảnh

Code:

```
(_, thresholded_img) = cv2.threshold(sobelized_img, 30, 255, cv2.THRESH_BINARY)
```

Ảnh lần lượt từ bước 1 đến bước 5

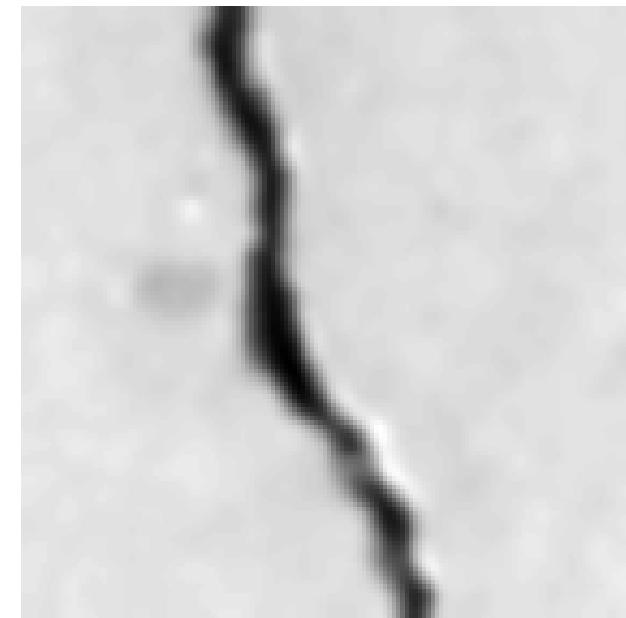
Gray img



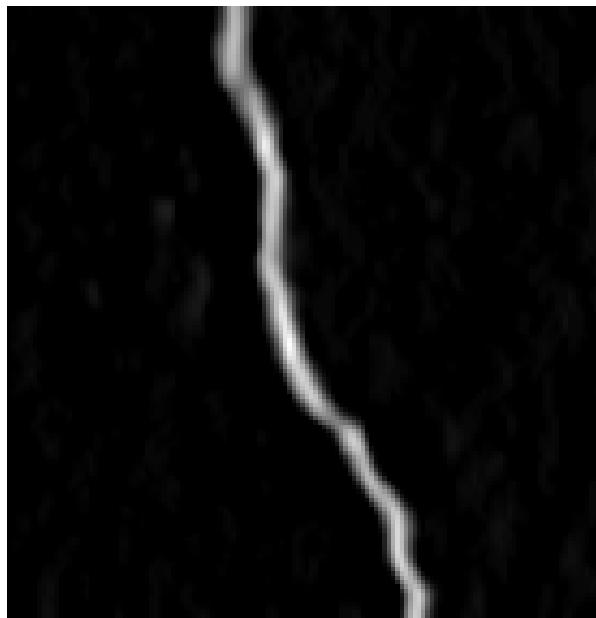
Resized img



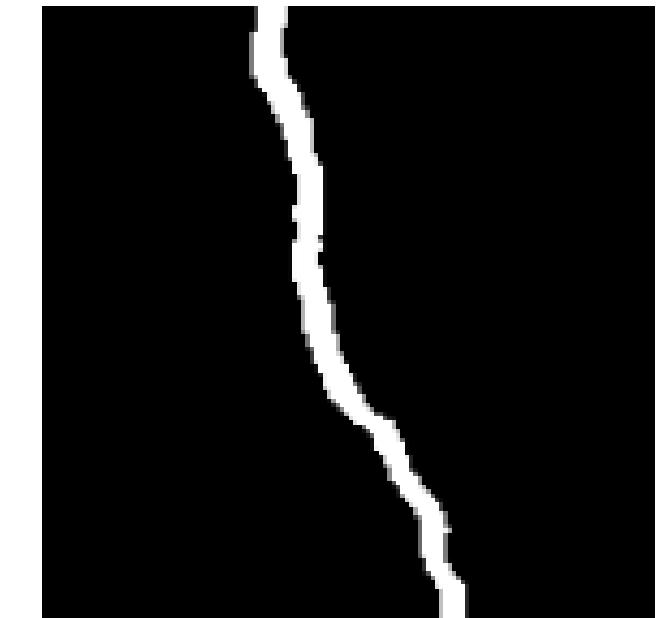
Blured img



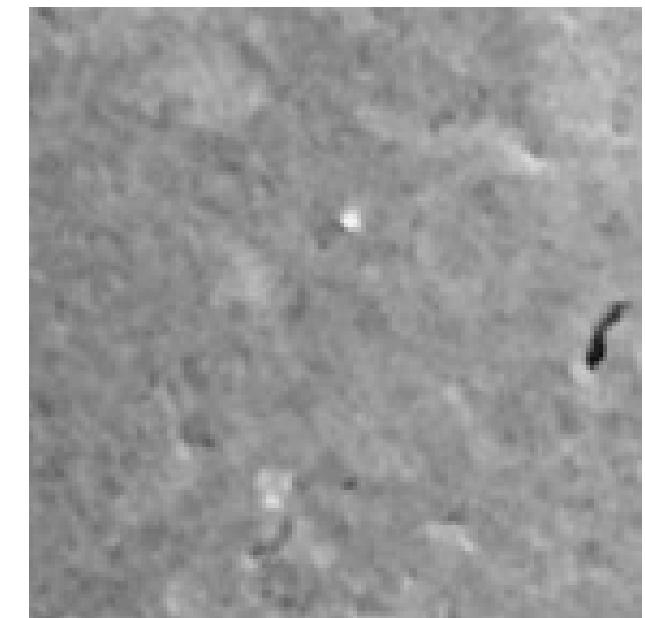
Sobelized img



Thresholded img



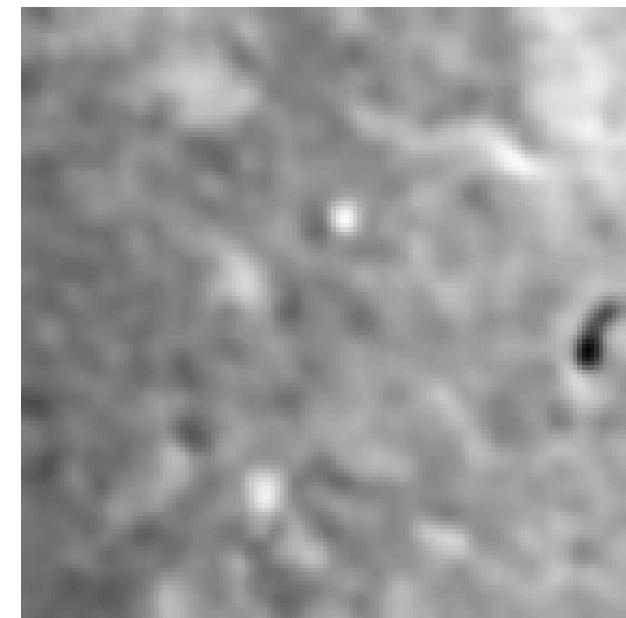
Gray img



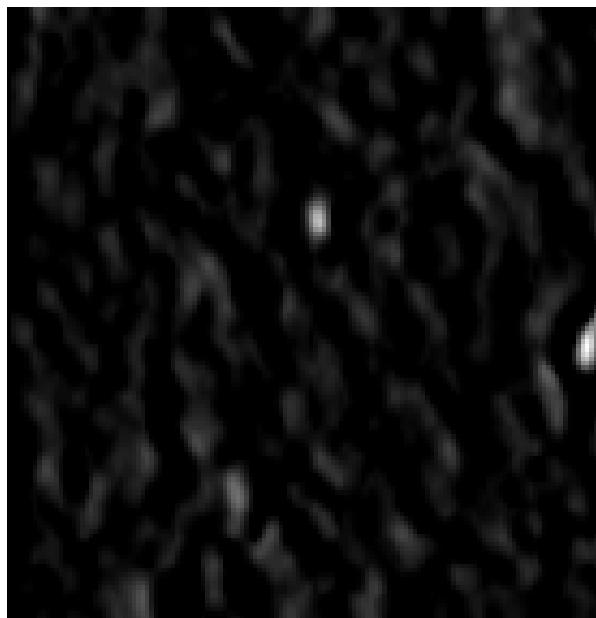
Resized img



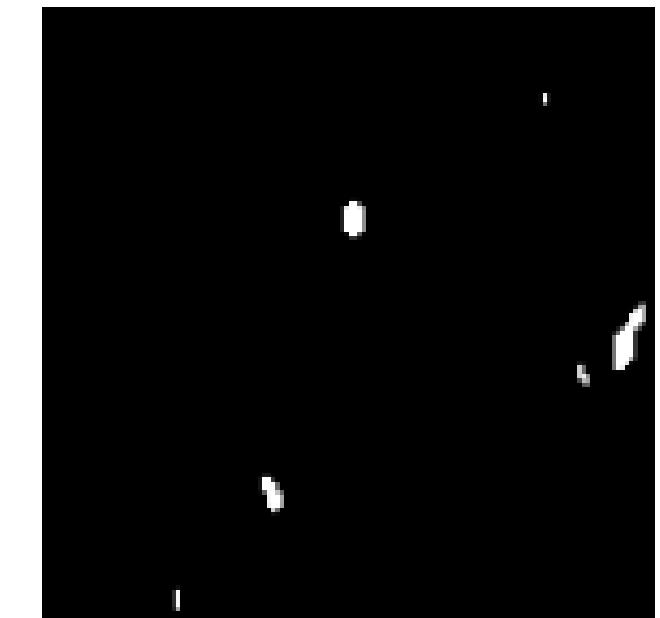
Blured img



Sobelized img



Thresholded img



4.2 Tiền xử lí dữ liệu

- **Bước 6:** Đưa về dạng vector và lưu vào biến img_data

Code:

```
img_data.append(np.reshape(sobelized_img, -1))
```

4.2 Tiền xử lí dữ liệu

- Bước 7: Min - Max Scaler

- + Chuyển đổi các giá trị của biến thành một phạm vi cụ thể, thường là từ 0 đến 1.
- + Giúp cân bằng dữ liệu và đảm bảo rằng các biến có cùng phạm vi, từ đó cải thiện hiệu suất của các mô hình học máy

Code:

```
min_max_scaler = preprocessing.MinMaxScaler()  
img_data = min_max_scaler.fit_transform(img_data)
```

5. Độ đo

- Accuracy: đo lường tỷ lệ các dự đoán chính xác trên tổng số lượng các dự đoán được thực hiện.

Công thức: $Accuracy = \frac{Số\ lượng\ dự\ đoán\ chính\ xác}{Tổng\ số\ lượng\ dự\ đoán}$

6. Kết quả

- Với learning_rate = 0.15 và epoch = 800

```
Evaluate Model

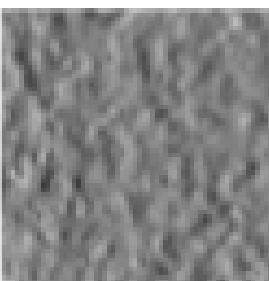
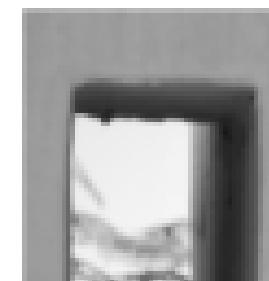
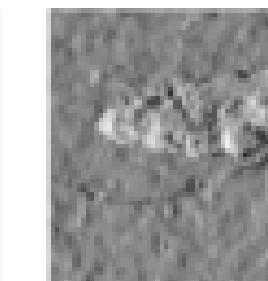
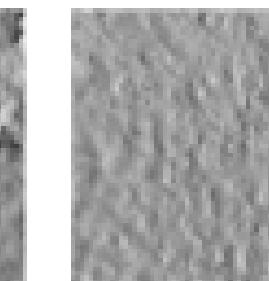
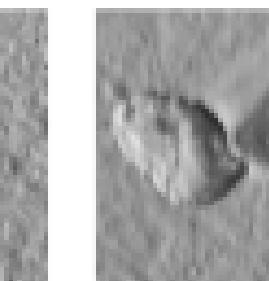
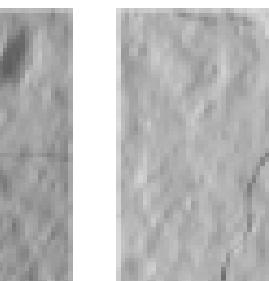
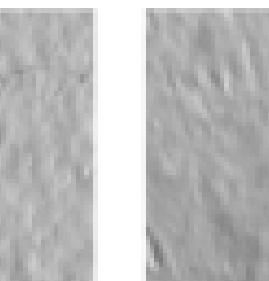
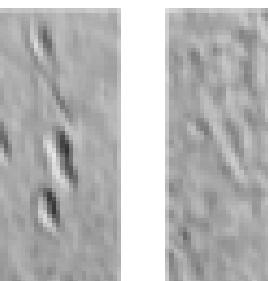
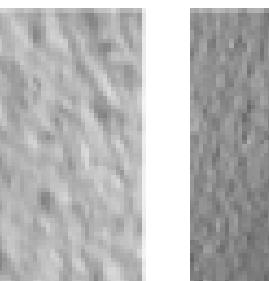
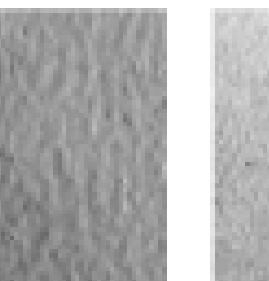
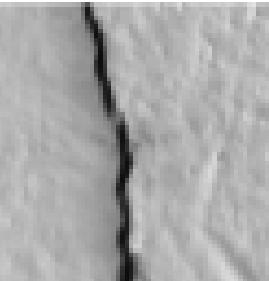
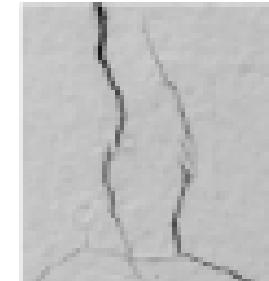
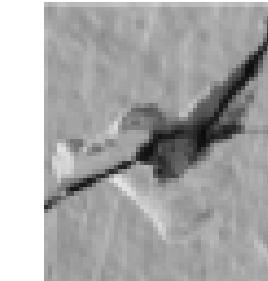
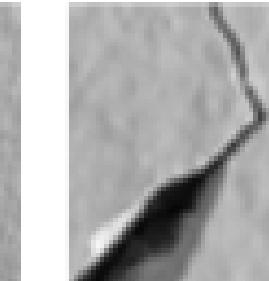
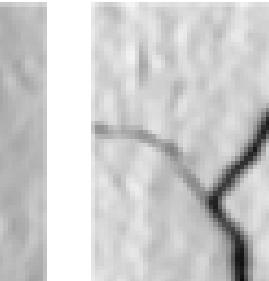
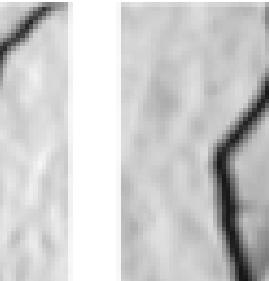
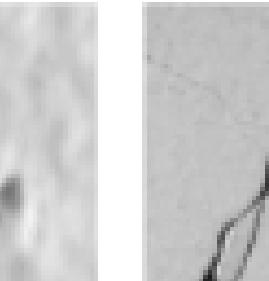
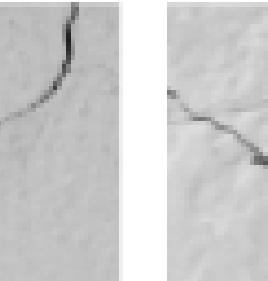
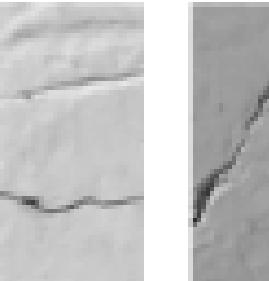
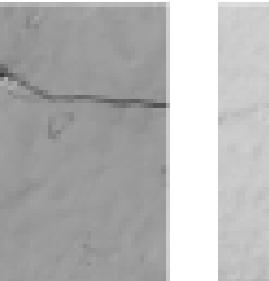
p_vali = predict(X_vali, w, b)
p_test = predict(X_test, w , b)
accur_vali = accuracy(p_vali, y_valid)
accur_test = accuracy(p_test, y_test)
print(f'Accuracy on Validation set = {accur_vali:.4f}')
print(f'Accuracy on Test set = {accur_test:.4f}')

[167] ✓ 1.4s

... Accuracy on Validation set = 0.9414
Accuracy on Test set = 0.9397
```

7. Minh họa

- Dữ liệu gồm: 10 ảnh có nhãn Positive và 10 ảnh có nhãn Negative

Predict: 0 Label: 0	Predict: 1 Label: 0	Predict: 1 Label: 0	Predict: 0 Label: 0	Predict: 1 Label: 0	Predict: 0 Label: 0				
									
Predict: 1 Label: 1	Predict: 1 Label: 1	Predict: 1 Label: 1	Predict: 1 Label: 1	Predict: 1 Label: 1	Predict: 1 Label: 1	Predict: 0 Label: 1	Predict: 0 Label: 1	Predict: 0 Label: 1	Predict: 1 Label: 1
									

Accuracy: 70%

Tài liệu tham khảo:

- <https://visionblog.github.io/image-processing>
- <https://www.imc.org.vn/thi-giac-may-tinh-voi-opencvpython-bai-4-tao-nguong-hinh-anh.html>
- <https://machinelearningcoban.com/2017/01/27/logisticregression/>
- https://phamdinhkhanh.github.io/deepai-book/ch_ml/FeatureEngineering.html
- <https://trituenhantao.io/machine-learning-co-ban/bai-6-logistic-regression-hoi-quy-logistic/>

**Thank you
very much!**