

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN



MÔN: HỌC MÁY (CAO HỌC)

LÝ THUYẾT VỀ MẠNG NEURAL
NETWORK ĐA TẦNG ẨN

Người hướng dẫn: PGS TS Lê Anh Cường

Người thực hiện: PHẠM ANH DUY - 51702088

TRẦN NGỌC BẢO DUY - 51702091

Khóa: : 21

TP.Hồ Chí Minh, 2020

LỜI NÓI ĐẦU

Trong thời gian làm đồ án này, chúng em đã nhận được nhiều sự giúp đỡ, đóng góp ý kiến và chỉ bảo nhiệt tình của thầy và bạn bè.

Nhóm em xin gửi lời cảm ơn chân thành đến thầy Lê Anh Cường, người đã tận tình hướng dẫn, chỉ bảo nhóm em trong suốt quá trình làm đồ án.

Nhóm em cũng xin chân thành cảm ơn các thầy cô giáo trong trường DH Tôn Đức Thắng nói chung, các thầy cô trong khoa CNTT nói riêng đã dạy dỗ cho em kiến thức về các môn đại cương cũng như các môn chuyên ngành, giúp chúng em có được cơ sở lý thuyết vững vàng và tạo điều kiện giúp đỡ nhóm em trong suốt quá trình học tập.

Cuối cùng, nhóm em xin chân thành cảm ơn gia đình và bạn bè, đã luôn tạo điều kiện, quan tâm, giúp đỡ, động viên em trong suốt quá trình học tập và hoàn thành bài đồ án này.

ĐỒ ÁN NÀY ĐƯỢC HOÀN THÀNH TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

Tôi xin cam đoan đây là sản phẩm đồ án của riêng chúng tôi và được sự hướng dẫn của thầy Lê Anh Cường. Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo.

Ngoài ra, trong bài đồ án còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc.

Nếu phát hiện có bất kỳ sự gian lận nào tôi xin hoàn toàn chịu trách nhiệm về nội dung đồ án của mình. Trường đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

TP.Hồ Chí Minh Ngày 21 tháng 3 năm 2020

(Ký tên và ghi rõ họ tên)

PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN

Phần xác nhận của GV hướng dẫn

TP.Hồ Chí Minh, 2020

(Ký tên và ghi rõ họ tên)

Phần đánh giá của GV chấm bài

TP.Hồ Chí Minh, 2020

(Ký tên và ghi rõ họ tên)

Mục lục

Danh sách hình vẽ

Danh sách bảng

Chương 1

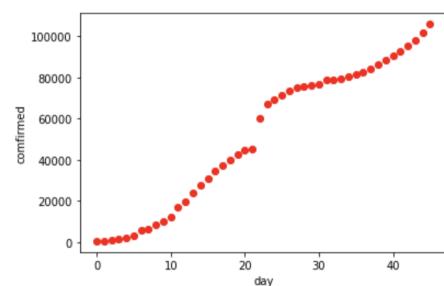
LINEAR REGRESSION

1.1 ĐỊNH NGHĨA

Thuật toán linear regression giải quyết các bài toán có đầu ra là giá trị thực, ví dụ: dự đoán giá cổ phiếu, dự đoán tuổi, dự đoán thời tiết, dự đoán giá nhà,....

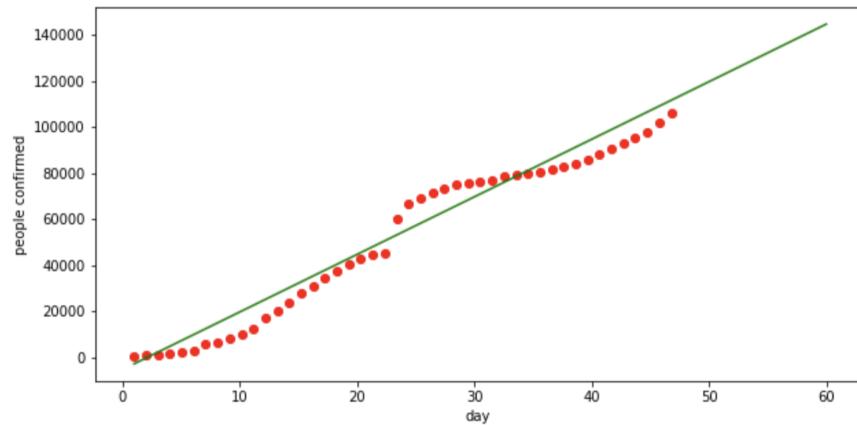
Như chúng ta đã biết, gần đây khi dịch bệnh n-COVID 19 (SARS-CoV-2) bùng lên, số lượng người tăng lên tuyến tính và chưa có dấu hiệu thuyên giảm qua từng ngày, chúng ta cũng có thể xây dựng một bài toán tuyến tính, cụ thể là bài toán dự đoán số người bị nhiễm tăng lên trong tương lai.

	date	people confirmed
0	01/22/2020	555.0
1	01/23/2020	653.0
2	01/24/2020	941.0
3	01/25/2020	1438.0
4	01/26/2020	2118.0
5	01/27/2020	2927.0
6	01/28/2020	5578.0
7	01/29/2020	6165.0
8	01/30/2020	8235.0
9	01/31/2020	9925.0



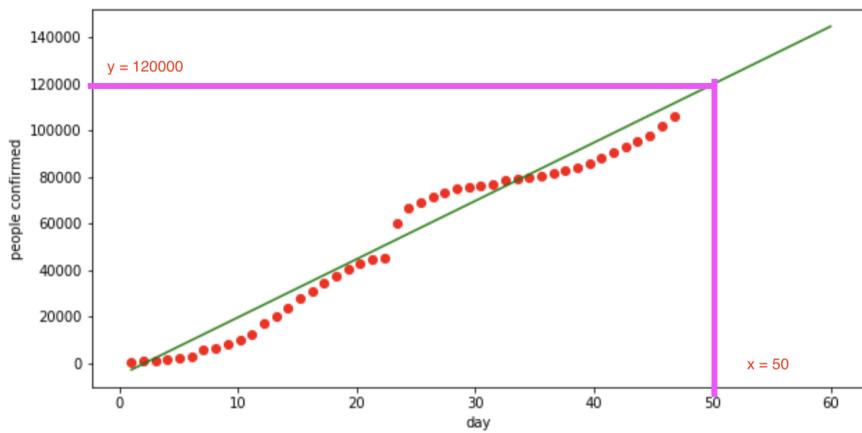
Hình 1.1: Bảng dữ liệu bệnh nhân theo ngày

Mô hình cần tìm đó chính là một đường thẳng tuyến tính thể hiện được trực quan số lượng người bị nhiễm theo mỗi ngày.



Hình 1.2: Mô hình cần tìm

Khi đã có đường tuyến tính như thế rồi chúng ta chỉ cần dựa vào đó đổi chiều xuống x và y để tìm ra số người dự đoán bị nhiễm trong tương lai. Ví dụ như vào ngày 50 chúng ta sẽ dự đoán được khoảng 120000 người bị nhiễm virus



Hình 1.3: Dự đoán dữ liệu

Vậy câu hỏi đặt ra là làm sao chúng ta có thể vẽ được đường thẳng tuyến tính như trên, 2 điều chúng ta cần chú ý để có thể tìm được đường thẳng tuyến tính đó là

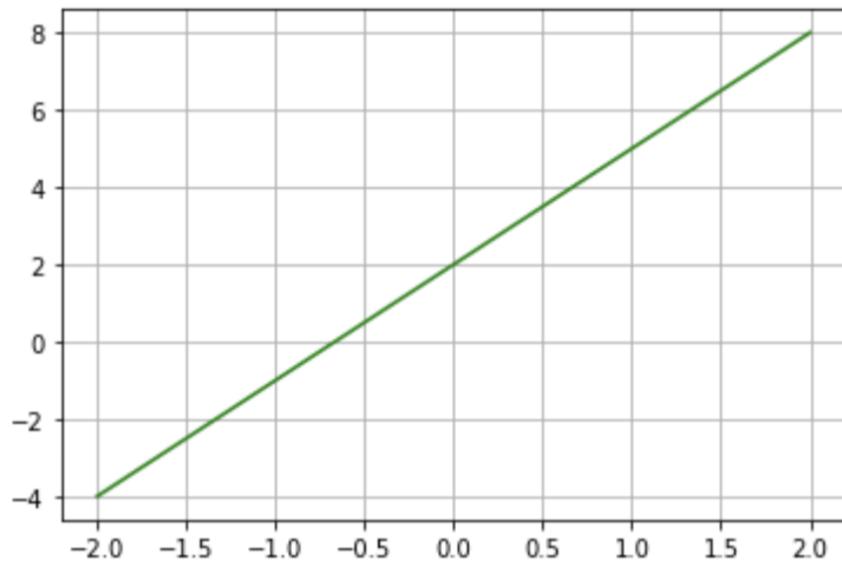
1. Training: tìm đường thẳng hay thường gọi là model đường thẳng mà các giá trị dữ liệu gần với đường thẳng này nhất. Khi chúng ta nhìn các điểm dữ liệu, ngay lập tức chúng ta có thể vẽ ngay đường thẳng, nhưng đối với máy tính chúng ta cần phải cho nó học từ các điểm dữ liệu, sau đó dùng phương pháp Gradient Descent để cập nhật lại đường thẳng.
2. Predict: dự đoán dữ liệu dựa trên đường thẳng đã training, như bài toán trên là dự đoán xem vào ngày 60 sẽ có bao nhiêu người bị nhiễm virus

1.2 XÂY DỰNG MÔ HÌNH

Để xây dựng một bài toán học máy sẽ bao gồm các bước sau:

1. Dựng mô hình
2. Thiết lập hàm mất mát - Loss function
3. Training mô hình, tìm trọng số thể hiện đường thẳng
4. Dự đoán dữ liệu mới vừa tìm được

1.2.1 Dựng mô hình



Hình 1.4: Mô hình đường thẳng tuyến tính $y = 3x + 2$

Vì mô hình chúng ta mong đợi đó là một đường thẳng tuyến tính như hình trên nên sẽ sử dụng phương trình đường thẳng có dạng $y = ax + b$

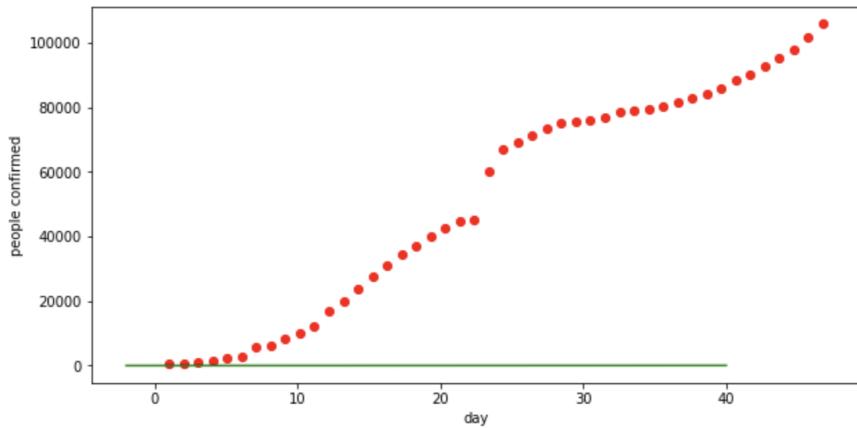
Như vậy điều chúng ta cần làm là tìm ra hệ số a và b dựa trên dữ liệu đầu vào là x và y . Thuận tiện cho việc kí hiệu nên chúng ta chuyển a và b thành w_0 và w_1 . Vậy phương trình được biểu diễn với mỗi điểm dữ liệu x_i là số ngày cụ thể, tương ứng với số người bị nhiễm trong ngày đó là y_i , chúng ta sẽ có phương trình ở dạng

$$y_i = w_1 * x_i + w_0$$

1.2.2 Thiết lập hàm mất mát - Loss function

Hàm mất mát hay còn gọi là hàm loss là một hàm được dùng để đánh giá mô hình của chúng ta đi có đúng hướng hay không. Ta sẽ khởi tạo giá trị $w_0 = 1$ và

$w_1 = 1$ sau đó sẽ cập nhật dừng qua bước training và dùng hàm loss để đánh giá.



Hình 1.5: Đường thẳng ban đầu $y = 1x + 0$

Chúng ta sẽ xây dựng công thức đánh giá dựa trên công thức Mean Square Error - Sai số toàn phương trung bình (MSE) với N là số dòng dữ liệu và $\hat{y}_i = w_1 * x_i + w_0$ với tham số w_0, w_1 hiện tại

$$Loss = \frac{1}{2N} * \sum_{i=1}^N (\hat{y}_i - y_i)^2$$

Qua phương trình trên ta thấy:

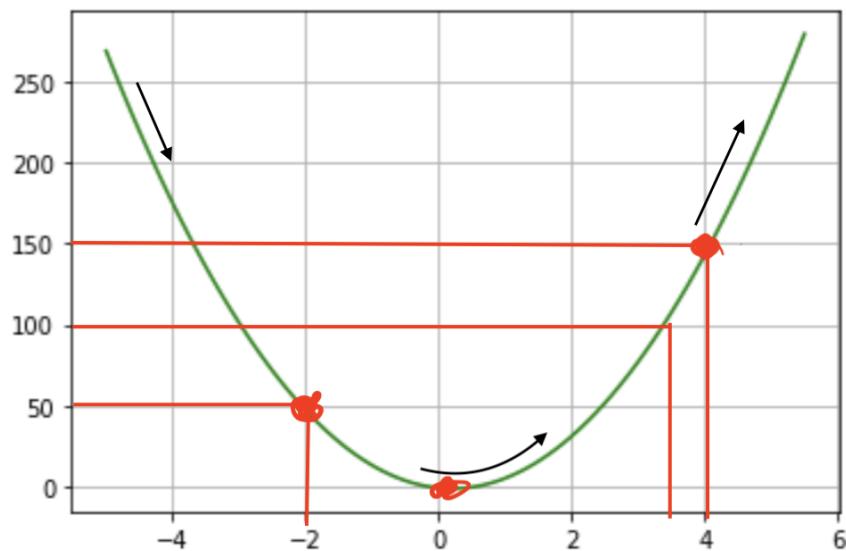
- Loss sẽ không âm
- Hàm loss càng nhỏ mô hình sẽ càng chính xác, vì \hat{y} gần bằng y giá trị thực tế sẽ gần bằng giá trị dự đoán
- Nếu Loss bằng 0 đường thẳng sẽ đi qua tất cả các điểm dữ liệu

Sau khi tìm được cách để đánh giá mô hình hiện tại có tốt hay không, chúng ta cần đi tìm thuật toán để cập nhật giá trị dữ liệu w_0 và w_1 . Phương pháp đó gọi là Gradient Descent

1.2.3 Training mô hình - Gradient descent

* Đạo hàm

Như chúng ta đã biết đạo hàm của một hàm số chính là một đường thẳng biểu thị độ dốc của phương trình hàm số đó.



Hình 1.6: Đồ thị $y = 10x^2 - 4x$

Đạo hàm của đồ thị trên là $y' = 20x - 4$.

- Khi $x = 0, y = 0$ thì $y' = 0$
- Khi $x = -2, y = 50$ thì $y' = -44$
- Khi $x = 4, y = 100$ thì $y' = 76$

Trị tuyệt đối của đạo hàm tại một điểm của hàm số càng lớn thì nó sẽ thể hiện được ngay tại điểm đó độ dốc càng cao.

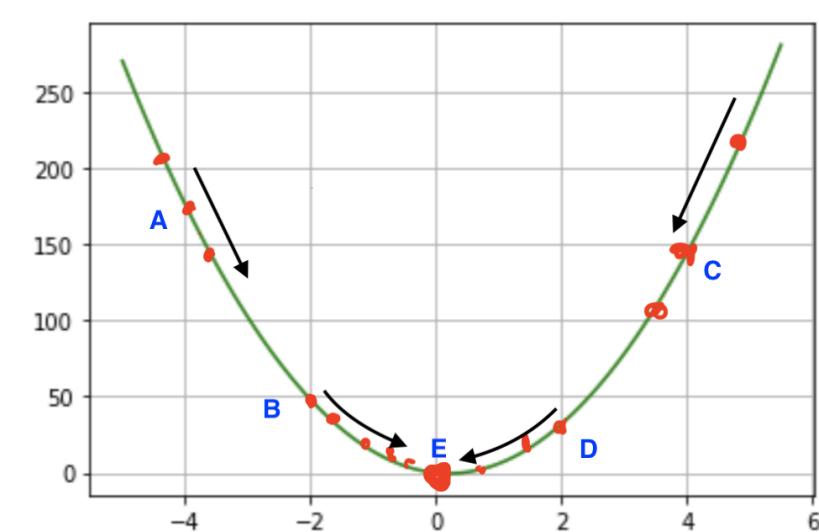
* Gradient Descent

Gradient Descent là một thuật toán dùng để tìm giá trị nhỏ nhất của hàm số dựa trên đạo hàm của hàm số $f(x)$. Các bước thực hiện như sau:

Bước 1: Khởi tạo giá trị ban đầu cho w

Bước 2: Cập nhật w theo công thức $w = w - lr * f'(x)$, với lr là learning rate hay gọi là tốc độ học tập thường là 0.1 hoặc 0.01

Bước 3: Tính $f(x)$ nếu $f(x)$ chưa đạt yêu cầu quay lại bước 2, nên nhớ $f(x)$ được tính dựa trên x và w



Hình 1.7: Gradient Descent của đồ thị $y = 10x^2 - 4x$

Theo các bước thực hiện gradient descent:

Bước 1: Khởi tạo điểm A $x = -4$

Bước 2: Cập nhật x theo công thức $x = x - lr * f'(x)$, vì đạo hàm tại A là âm nên x tăng

Bước 3: Tiếp tục thực hiện các điểm trên đồ thị, như điểm C, thì x sẽ tiến dần về D.

Chúng ta sẽ tiếp tục cập nhật theo công thức trên cho đến khi hàm số đạt giá trị nhỏ nhất

Đạo hàm của đồ thị là $y' = 20x - 4$. Trị tuyệt đối của đạo hàm tại điểm A sẽ lớn hơn điểm B, cho nên khi x tiến dần về 0 thì lúc đó công thức cập nhật hệ số $x = x - lr * f'(x)$ sẽ thay đổi không đáng kể, lúc đó ta sẽ đạt được giá trị nhỏ nhất của hàm số, giải thích hình bên trên

Điểm A: Khi $x = -4$ thì $y' = -84$ là một số âm cho nên áp dụng công thức $x = x - lr * f'(x)$ thì x tăng nên $f(x)$ tăng, A tiến dần B

Điểm C Khi $x = 4$ thì $y' = 84$ là một số dương cho nên áp dụng công thức $x = x - lr * f'(x)$ thì x giảm nên $f(x)$ tăng, C tiến dần D

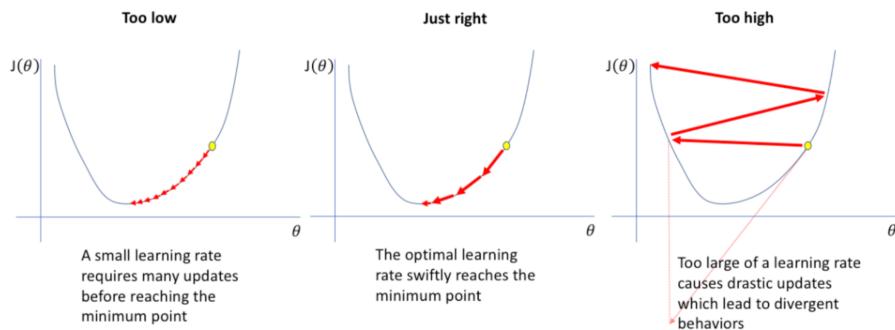
Nhận xét

- Thuật toán hoạt động tốt trong trường hợp không thể tìm giá trị nhỏ nhất bằng đại số tuyến tính
- Việc quan trọng là phải tính được đạo hàm trên từng biến, sau đó lặp lại đến khi nào tìm ra giá trị nhỏ nhất.

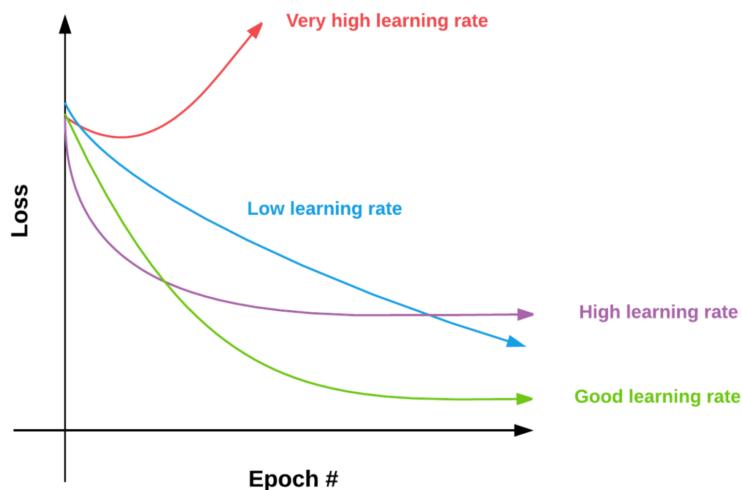
Lưu ý là việc chọn learning rate rất quan trọng, có 3 trường hợp khi ta chọn learning rate:

- Lr nhỏ: Chúng ta sẽ phải thực hiện rất nhiều vòng lặp để có thể cập nhật giá trị của hàm số cho tới khi nó đạt giá trị nhỏ nhất

- Lr hợp lí: Sau một vài lần lặp, chúng ta sẽ nhanh chóng tìm ra được giá trị nhỏ nhất
- Lr lớn: Gây ra hiện tượng overshoot và không bao giờ đạt được giá trị mong muốn



Hình 1.8: Ba trường hợp của learning rate



Hình 1.9: Các trường hợp của learning rate tương ứng với loss

* Áp dụng vào bài toán dự đoán linear regression

Vậy chúng ta cần tìm giá trị nhỏ nhất của hàm

$$Loss(w_0, w_1) = \frac{1}{2N} * \sum_{i=1}^N ((w_0 + w_1 * x_i) - y_i)^2$$

Dựa vào công thức gradient descent trước tiên ta sẽ đi tính đạo hàm của Loss theo w_0, w_1

Tại 1 điểm (x_i, y_i) ta gọi hàm

$$f(w_0, w_1) = \frac{1}{2N} * ((w_0 + w_1 * x_i) - y_i)^2$$

Ta có:

$$\begin{aligned}\frac{df}{dw_0} &= w_0 + w_1 * x_i - y_i \\ \frac{df}{dw_1} &= x_i * (w_0 + w_1 * x_i - y_i)\end{aligned}$$

Công thức trên tính tại 1 điểm, cho nên khi tính trên toàn bộ dữ liệu ta sẽ có:

$$\begin{aligned}\frac{dL}{dw_0} &= \sum_{i=1}^N w_0 + w_1 * x_i - y_i \\ \frac{dL}{dw_1} &= \sum_{i=1}^N x_i * (w_0 + w_1 * x_i - y_i)\end{aligned}$$

* Biểu diễn dưới dạng ma trận

Để thuận tiện cho việc tính toán ta sẽ sử dụng ma trận để tính toán trên toàn bộ dữ liệu thay vì phải tính từng dòng dữ liệu với công thức $\hat{y}_i = w_0 + w_1 * x_i$ hay công thức đạo hàm khác

$$X = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ ... & ... \\ 1 & x_n \end{bmatrix}, Y = \begin{bmatrix} y_1 \\ y_2 \\ ... \\ y_n \end{bmatrix}, W = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

$$\hat{Y} = X * W = \begin{bmatrix} w_0 + w_1 * x_1 \\ w_0 + w_1 * x_2 \\ ... \\ w_0 + w_1 * x_n \end{bmatrix}$$

Hình 1.10: Xử lí dữ liệu đầu vào với ma trận

$$X[:, 1] = \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{bmatrix}, \text{sum}(X[:, 1]) = x_1 + x_2 + \dots + x_n$$

$$\frac{dJ}{dw_0} = \text{sum}(\hat{Y} - Y), \frac{dJ}{dw_1} = \text{sum}(X[:, 1] \otimes (\hat{Y} - Y))$$

Hình 1.11: Sử dụng ma trận để tính loss cũng như đạo hàm

1.2.4 Hiện thực hoá bằng code

```

class LinearModel():
    def __init__(self, X=None, y=None, normalize=True):
        self.X_init = np.array(X)
        self.y_init = np.array(y)
        self.X = X
        self.y = y
        self.size = X.shape[-1]
        self.weight = np.zeros(X.shape[-1]).reshape(-1,1)
        self.mean = np.mean(self.X[:, 1:], axis=0)
        self.std = np.std(self.X[:, 1:], axis=0)
        if normalize:
            self.feature_normalization()

    # chuẩn hoá về khoảng [-1,1]
    def feature_normalization(self):
        self.X[:, 1:] = (self.X[:, 1:] - np.mean(self.X[:, 1:], axis=0)) / np.std(self.X[:, 1:], axis=0)

    def forward(self):
        return np.dot(self.X, self.weight)

    def loss(self):
        return np.sum(((self.forward() - self.y)**2) / (self.size)*2)

    def derivative_w0(self):
        return np.sum(self.forward() - self.y)

    def derivative_w1(self):
        return np.sum(np.multiply((self.forward() - self.y), self.X[:,1].reshape(-1,1)))

```

Hình 1.12: Xây dựng model linear regression

```

def gradient_descend(self):
    old_loss = 0
    count = 0
    while True:
        count+=1
        self.weight[0] = self.weight[0] - 0.01 * self.derivative_w0()
        self.weight[1] = self.weight[1] - 0.01 * self.derivative_w1()
        new_loss = self.loss()
        if abs(new_loss - old_loss) < 10e+5:
            break
        old_loss = new_loss
        print(new_loss)

    print("Final loss is:",old_loss)
    for index in range(len(X[:,1])):
        if index == 0:
            break
        plt.scatter(X[:,index],y.T[0])
    plt.show()

def predict(self):
    y_pred = np.dot(X, self.weight)
    return y_pred

def predict_by_day(self, day):
    day = np.array(day,dtype=float)
    day[:,1:] = (day[:,1:]-self.mean) / self.std
    y_pred = np.dot(day, self.weight)
    return y_pred

```

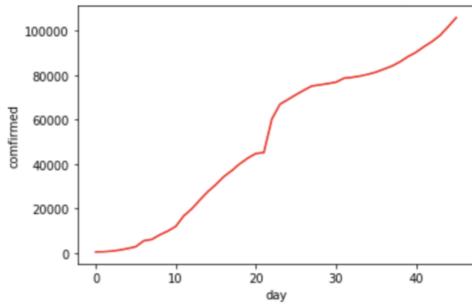
Hình 1.13: Cập nhật w qua thuật toán gradient descent

```

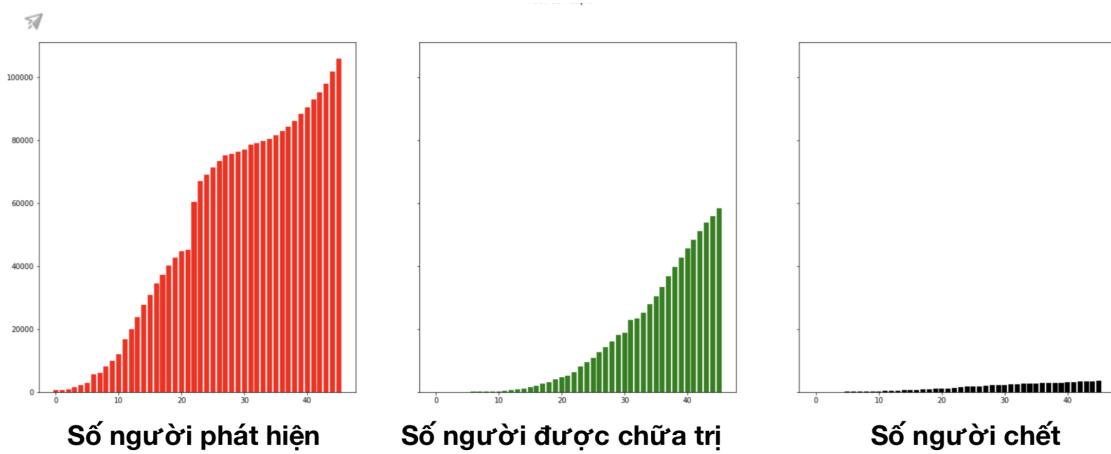
x_true, y_true = handle_data(data)
X = np.copy(X_true)
y = np.copy(y_true)
model = LinearModel(X,y)

Tổng số ngày trong khảo sát là 46 ngày
['01/22/2020', '01/23/2020', '01/24/2020', '01/25/2020', '01/26/2020',
'01/27/2020', '01/28/2020', '01/29/2020', '01/30/2020', '01/31/2020',
'02/01/2020', '02/02/2020', '02/03/2020', '02/04/2020', '02/05/2020',
'02/06/2020', '02/07/2020', '02/08/2020', '02/09/2020', '02/10/2020',
'02/11/2020', '02/12/2020', '02/13/2020', '02/14/2020', '02/15/2020',
'02/16/2020', '02/17/2020', '02/18/2020', '02/19/2020', '02/20/2020',
'02/21/2020', '02/22/2020', '02/23/2020', '02/24/2020', '02/25/2020',
'02/26/2020', '02/27/2020', '02/28/2020', '02/29/2020', '03/01/2020',
'03/02/2020', '03/03/2020', '03/04/2020', '03/05/2020', '03/06/2020',
'03/07/2020']

```



Hình 1.14: Dữ liệu đầu vào



Hình 1.15: Minh họa dữ liệu đầu vào

```

# show
plt.scatter(X,true[:,1],y_true,c='g')
plt.plot(X,true[:,1],model.predict(),c='red')
plt.xlabel('num day')
plt.ylabel('confirmed')
plt.show()

DAY = 60
predict = []
for item in range(1,DAY):
    day = item
    day = [[1,item]]
    pre = model.predict(day)
    predict.append(pre[0][1])
    if item < 45:
        real = y_true[item]
        print('Ngày thứ',item,'Dự đoán sẽ có:',int(model.predict_by_day(item)),',real:',int(real))
    else:
        print('Ngày thứ',item,'Dự đoán sẽ có:',int(model.predict_by_day(day)),',real:',int(real/item))
print('tổng số ca nhiễm là:',int(sum(predict)))

```

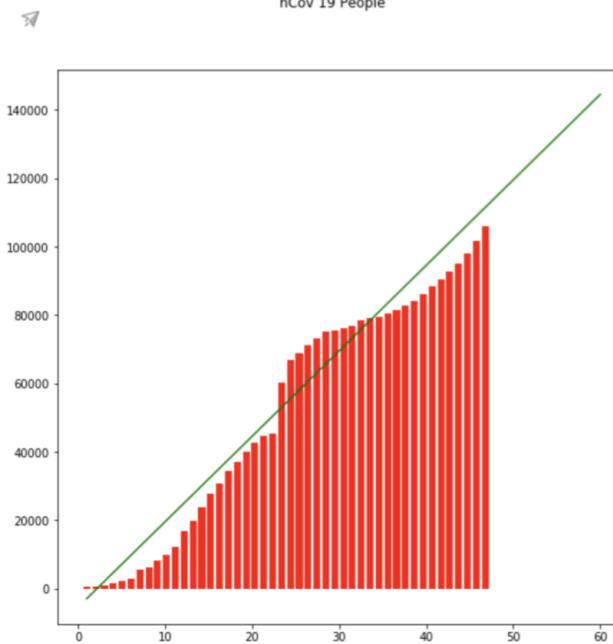
Dự đoán sẽ có: 121642 ca nhiễm. Trung bình 1 ngày sẽ có 2432
 Ngày thứ 51
 Ngày thứ 52
 Ngày thứ 53
 Ngày thứ 54
 Ngày thứ 55
 Ngày thứ 56
 Ngày thứ 57
 Ngày thứ 58
 Ngày thứ 59
 Ngày thứ 60

Hình 1.16: Training dữ liệu với model linear

```

: num_day = np.asarray(np.linspace(start=1,stop=DAY,num=DAY-1))
: predict = np.array(predict)
: fig, axs = plt.subplots(1, figsize=(9,9), sharey=True)
: axs.plot(num_day, predict,color='green')
: axs.bar(num_day[:46], y_true.T[0],color='r')
: fig.suptitle('nCov 19 People')
: Text(0.5, 0.98, 'nCov 19 People')

```



Hình 1.17: Vẽ hình dự đoán những ngày trong tương lai

Chương 2

PERCEPTRON LEARNING ALGORITHM

2.1 GIỚI THIỆU

2.1.1 Khái niệm

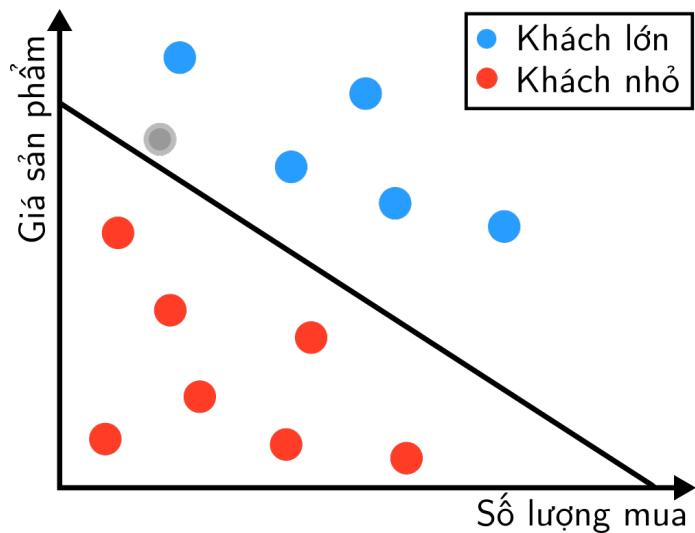
Thuật toán đầu tiên trong Classification có tên là Perceptron Learning Algorithm (PLA) hoặc đôi khi được viết gọn là Perceptron.

Perceptron là một thuật toán giúp chúng ta thực hiện công việc phân loại với hai lớp như trên, ta sẽ gọi hai lớp này là +1, -1. Thuật toán ban đầu được Frank Rosenblatt đề xuất dựa trên ý tưởng của Neural thần kinh, nó nhanh chóng tạo nên tiếng vang lớn trong lĩnh vực AI. Không giống như Naive Bayes - một thuật toán phân loại bằng cách tính xác suất trên toàn bộ tập dữ liệu (batch learning), Perceptron sẽ đọc từng dữ liệu và điều chỉnh biên giới sao cho tất cả các điểm nằm cùng một phía của biên giới có nhãn giống nhau (online learning). Bên cạnh đó, nó

là nền tảng cho một mảng lớn quan trọng của Machine Learning là Neural Networks và sau này là Deep Learning.

2.1.2 Bài toán phân loại (Classification)

Giả sử chúng ta cần chia khách hàng ra làm hai loại/lớp (category/class) dựa vào nguồn lợi họ đem lại cho công ty: khách hàng nhỏ và khách hàng lớn. Về cơ bản, nguồn lợi được tính theo giá mặt hàng và số lượng khách mua. Như vậy, ta sẽ biểu diễn khách hàng theo hai yếu tố trên trên mặt phẳng:



Hình 2.1: Perceptron - Mô hình phân loại cơ bản

Bài toán của chúng ta là từ những điểm xanh và đỏ cho trước (tức marketer đã xác định), hãy xây dựng một quy tắc phân loại để dự đoán class của điểm màu xám. Nói cách khác, chúng ta cần xác định một biên giới để chia lãnh thổ của hai class này, rồi với điểm cần phân loại màu xám ta chỉ cần xem nó nằm ở phía bên nào của đường biên giới là xong. Biên giới đơn giản nhất (theo đúng nghĩa toán học) trong mặt phẳng là một đường thẳng (đường màu đen trong hình), trong không

gian ba chiều là một mặt phẳng, trong không gian nhiều chiều là một siêu phẳng (hyperplane, một đường thẳng nằm trong nhiều chiều).

Lưu ý rằng các khái niệm lớp, nhãn, danh mục ở bài toán phân loại là tương tự nhau.

2.1.3 Bài toán Perceptron

Bài toán Perceptron được phát biểu như sau: Cho hai class được gán nhãn, hãy tìm một đường phẳng sao cho toàn bộ các điểm thuộc class 1 nằm về 1 phía, toàn bộ các điểm thuộc class 2 nằm về phía còn lại của đường phẳng đó. Với giả định rằng tồn tại một đường phẳng như thế.

Nếu tồn tại một đường phẳng phân chia hai class thì ta gọi hai class đó là linearly separable. Các thuật toán classification tạo ra các boundary là các đường phẳng được gọi chung là Linear Classifier.

2.1.4 Linear Separability

Một nhược điểm của Perceptron là nó chỉ hoạt động thực sự tốt khi dữ liệu phân tách tuyến tính (linearly separable), tức có thể dùng đường thẳng, mặt phẳng, siêu phẳng để làm biên giới, nếu dữ liệu không thỏa điều kiện này, không những Perceptron mà bất kỳ thuật toán phân loại tuyến tính nào cũng sẽ fail "nhè nhẹ". Điều này dẫn đến một khái niệm quan trọng:

Linear Separability: cho vector \mathbf{x}_i biểu diễn một dữ liệu, y_i là nhãn của \mathbf{x}_i và $f(\mathbf{x}_i, y_i) = y_i \mathbf{x}_i$ là feature function của dữ liệu, tập dữ liệu này được gọi là *linearly separable* nếu tồn tại một vector trọng số \mathbf{w} và một margin $p > 0$ sao cho:

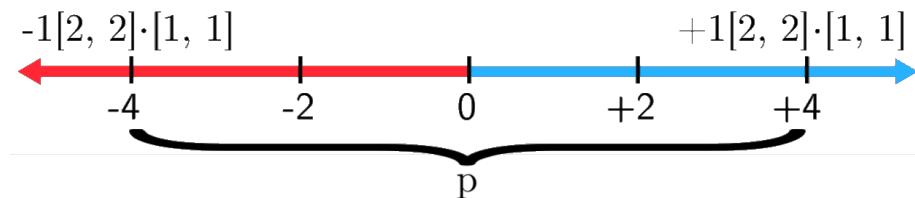
$$\forall (\mathbf{x}_i, y_i) \in \text{Dataset}, \mathbf{w} \cdot f(\mathbf{x}_i, y_i) \geq p + \mathbf{w} \cdot f(\mathbf{x}_i, y_i)$$

với y' khác y_i , là nhãn của lớp còn lại. Điều này khá dễ hiểu vì khi $\mathbf{x}_i \cdot \mathbf{w}^T$ cùng dấu với y_i , tức phân lớp đúng, thì tích của chúng không âm.

Ví dụ, ta có vector trọng số $\mathbf{w} = [1, 1]$ và vector $\mathbf{x} = [2, 2]$ có nhãn $y = 1$, như vậy thay vào công thức trên ta được:

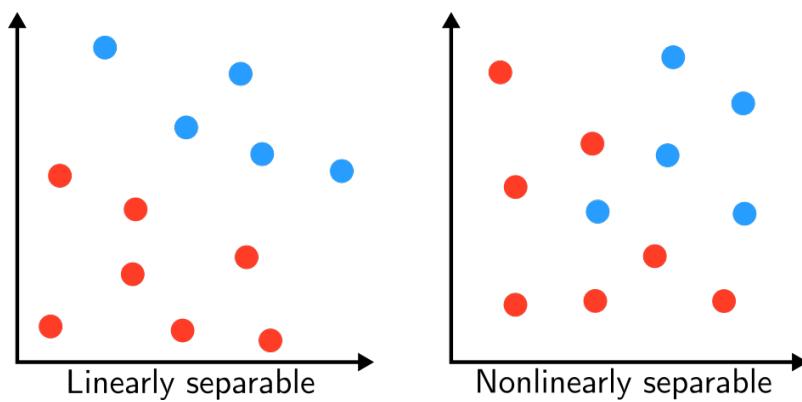
$$1[2, 2] \cdot [1, 1] \geq p + (-1)[2, 2] \cdot [1, 1]$$

$$\Leftrightarrow 4 \geq p + (-4) \Leftrightarrow p \leq 8$$



Hình 2.2: Linear Separability - Biểu diễn p

Nói cách khác, dữ liệu là linearly separable khi hai tập bao lồi các điểm của hai lớp không giao nhau.

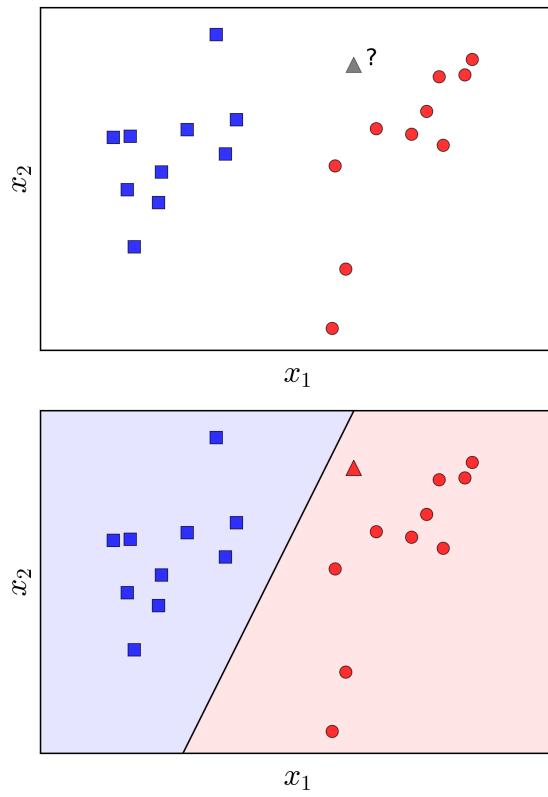


Hình 2.3: Linear Separability - Biểu diễn Linearly & Nonlinearly Separable

Dữ liệu trên thực tế thường hiếm khi linearly separable là một hạn chế cho Perceptron, tuy nhiên Perceptron vẫn là nền tảng cho các thuật toán Neural Network hay Deep Learning sau này.

2.2 PERCEPTRON

2.2.1 Ý tưởng



Hình 2.4: Bài toán Perceptron

Chúng ta có đường nét liền là biên giới phân lớp. Thuật toán sẽ dùng đường biên giới để thử phân loại dữ liệu, với mỗi lần phân loại thử bị sai, thuật toán sẽ

điều chỉnh lại biên giới. Ở hình bên trên, có một điểm tam giác nằm cùng lanh thổ với các điểm màu đỏ. Trong trường hợp này, điểm dữ liệu mới hình tam giác được phân vào class đỏ.

2.2.2 Một số ký hiệu và công thức toán học

Giả sử $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N] \in \mathbb{R}^{d \times N}$ là ma trận chứa các điểm dữ liệu mà mỗi cột $\mathbf{x}_i \in \mathbb{R}^{d \times 1}$ là một điểm dữ liệu trong không gian d chiều.

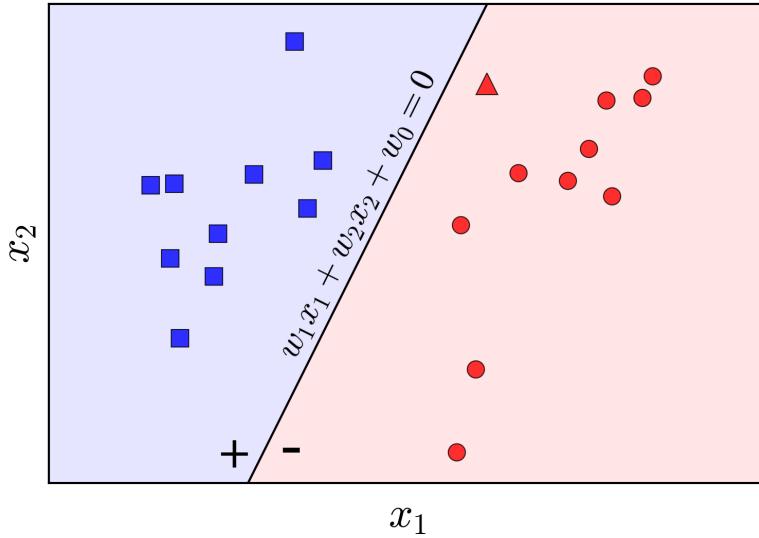
Giả sử thêm các nhãn tương ứng với từng điểm dữ liệu được lưu trong một vector hàng $\mathbf{y} = [y_1, y_2, \dots, y_N] \in \mathbb{R}^{d \times N}$, với $y_i = 1$ nếu \mathbf{x}_i thuộc class 1 (xanh) và $y_i = -1$ nếu \mathbf{x}_i thuộc class 2 (đỏ).

Tại một thời điểm, giả sử ta tìm được boundary là đường phẳng có phương trình:

$$f_w(\mathbf{x}) = w_1x_1 + \dots + w_dx_d + w_0 = \mathbf{w}^T\bar{\mathbf{x}} = 0$$

Với $\bar{\mathbf{x}}$ là điểm dữ liệu mở rộng bằng cách thêm phần tử $x_0 = 1$ lên trước vector \mathbf{x} tương tự như trong Linear Regression.

Để cho đơn giản, chúng ta hãy cùng làm việc với trường hợp mỗi điểm dữ liệu có số chiều $d = 2$. Giả sử đường thẳng $w_1x_1 + w_2x_2 + w_0 = 0$ chính là nghiệm cần tìm như hình dưới đây:



Hình 2.5: Phương trình đường thẳng boundary

Nhận xét rằng các điểm nằm về cùng một phía so với đường thẳng này sẽ làm cho hàm số $f_w(\mathbf{x})$ mang cùng dấu. Chỉ cần đổi dấu của w nếu cần thiết, ta có thể giả sử các điểm nằm trong nửa mặt phẳng nền xanh mang dấu dương (+), các điểm nằm trong nửa mặt phẳng nền đỏ mang dấu âm (-). Các dấu này cũng tương đương với nhãn y của mỗi class. Vậy nếu \mathbf{w} là một nghiệm của bài toán Perceptron, với một điểm dữ liệu mới \mathbf{x} chưa được gán nhãn, ta có thể xác định class của nó bằng phép toán đơn giản như sau:

$$label(\mathbf{x}) = 1 \text{ if } \mathbf{w}^T \mathbf{x} \geq 0, \text{ otherwise } -1$$

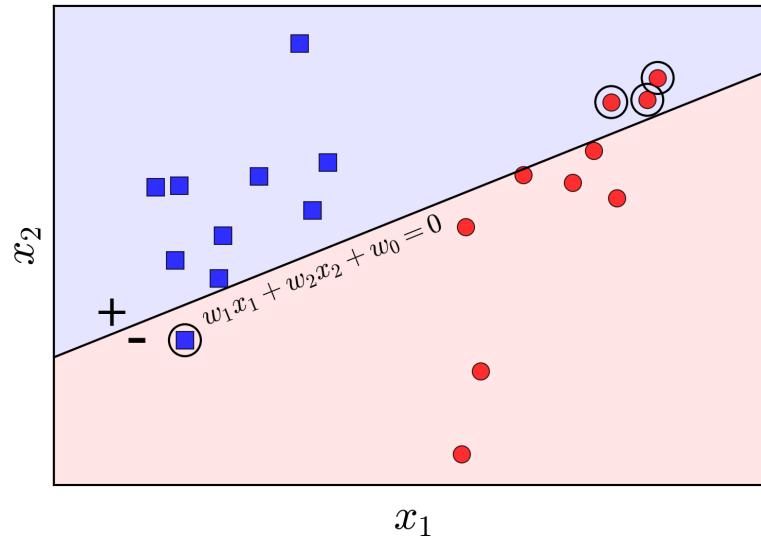
Ngắn gọn hơn:

$$label(\mathbf{x}) = sgn(\mathbf{w}^T \mathbf{x})$$

trong đó, sgn là hàm xác định dấu, với giả sử rằng $sgn(0) = 1$.

2.2.3 Xây dựng hàm mất mát (loss function)

Cần phải xây dựng hàm mất mát với các tham số là \mathbf{w} bất kỳ. Vẫn trong không gian hai chiều, giả sử đường thẳng $w_1x_1 + w_2x_2 + w_0 = 0$ được cho như hình dưới đây:



Hình 2.6: Đường thẳng bất kỳ và các điểm bị misclassified được khoanh tròn

Trong trường hợp này, các điểm được khoanh tròn là các điểm bị misclassified (phân lớp lỗi). Điều chúng ta mong muốn là không có điểm nào bị misclassified. Hàm mất mát đơn giản nhất chúng ta nghĩ đến là hàm đếm số lượng các điểm bị misclassified và tìm cách tối thiểu hàm số này:

$$J_1(\mathbf{w}) = \sum_{\mathbf{x}_i \in \mathcal{M}} (-y_i \operatorname{sgn}(\mathbf{w}^T \mathbf{x}_i))$$

trong đó \mathcal{M} là tập hợp các điểm bị misclassified (tập hợp này thay đổi theo \mathbf{w} . Với mỗi điểm $\mathbf{x}_i \in \mathcal{M}$, vì điểm này bị misclassified nên y_i và $\operatorname{sgn}(\mathbf{w}^T \mathbf{x})$ khác

nhau, và vì thế $-y_i \text{sgn}(\mathbf{w}^T \mathbf{x}_i) = 1$. Vậy $J_1(\mathbf{w})$ chính là hàm đếm số lượng các điểm bị misclassified. Khi hàm số này đạt giá trị nhỏ nhất bằng 0 thì ta không còn điểm nào bị misclassified.

Một điểm quan trọng, hàm số này là rắc rối, không tính được đạo hàm theo \mathbf{w} nên rất khó tối ưu. Chúng ta cần tìm một hàm mất mát khác để việc tối ưu khả thi hơn.

Xét hàm mất mát sau đây:

$$J(\mathbf{w}) = \sum_{\mathbf{x}_i \in \mathcal{M}} (-y_i \mathbf{w}^T \mathbf{x}_i)$$

Hàm $J()$ khác một chút với hàm $J_1()$ ở việc bỏ đi hàm sgn. Nhận xét rằng khi một điểm misclassified \mathbf{x}_i nằm càng xa boundary thì giá trị $-y_i \mathbf{w}^T \mathbf{x}_i$ sẽ càng lớn, nghĩa là sự sai lệch càng lớn. Giá trị nhỏ nhất của hàm mất mát này cũng bằng 0 nếu không có điểm nào bị misclassified. Hàm mất mát này cũng được cho là tốt hơn hàm $J_1()$ vì nó trừng phạt rất nặng những điểm lấn sâu sang lãnh thổ của class kia. Trong khi đó, $J_1()$ trừng phạt các điểm misclassified như nhau (đều = 1), bất kể chúng xa hay gần với đường biên giới.

Tại một thời điểm, nếu chúng ta chỉ quan tâm tới các điểm bị misclassified thì hàm số $J(\mathbf{w})$ khả vi (tính được đạo hàm), vậy chúng ta có thể sử dụng Gradient Descent hoặc Stochastic Gradient Descent (SGD) để tối ưu hàm mất mát này. Với ưu điểm của SGD cho các bài toán large-scale, chúng ta sẽ làm theo thuật toán này.

Với một điểm dữ liệu \mathbf{x}_i bị misclassified, hàm mất mát trở thành:

$$J(\mathbf{w}; \mathbf{x}_i; y_i) = -y_i \mathbf{w}^T \mathbf{x}_i$$

Đạo hàm tương ứng:

$$\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{x}_i; y_i) = -y_i \mathbf{x}_i$$

Vậy quy tắc cập nhật là:

$$\mathbf{w} = \mathbf{w} + \eta y_i \mathbf{x}_i$$

với η là learning rate bất kỳ nhưng ở đây ta sẽ chọn bằng 1. Ta có một quy tắc cập nhật rất gọn là: $\mathbf{w}_{t+1} = \mathbf{w}_t + y_i \mathbf{x}_i$. Nói cách khác, với mỗi điểm \mathbf{x}_i bị misclassified, ta chỉ cần nhân điểm đó với nhãn y_i của nó, lấy kết quả cộng vào \mathbf{w} ta sẽ có được \mathbf{w} mới.

$$\begin{aligned}\mathbf{w}_{t+1}^T \mathbf{x}_i &= (\mathbf{w}_t + y_i \mathbf{x}_i)^T \mathbf{x}_i \\ &= \mathbf{w}_t^T \mathbf{x}_i + y_i \|\mathbf{x}_i\|_2^2\end{aligned}$$

Nếu $y_i = 1$, vì \mathbf{x}_i bị misclassified nên $\mathbf{w}_t^T \mathbf{x}_i < 0$. Cũng vì $y_i = 1$ nên $y_i \|\mathbf{x}_i\|_2^2 = \|\mathbf{x}_i\|_2^2 \geq 1$ (chú ý $x_0 = 1$), nghĩa là $\mathbf{w}_{t+1}^T \mathbf{x}_i > \mathbf{w}_t^T \mathbf{x}_i$. Lý giải bằng lời, \mathbf{w}_{t+1} tiến về phía làm cho \mathbf{x}_i được phân lớp đúng. Điều tương tự xảy ra nếu $y_i = -1$.

2.2.4 Tóm tắt Perceptron Learning Algorithm

1. Chọn ngẫu nhiên một vector hệ số \mathbf{w} với các phần tử gần 0.
2. Duyệt ngẫu nhiên qua từng điểm dữ liệu \mathbf{x}_i :
 - Nếu \mathbf{x}_i được phân lớp đúng, tức $\text{sgn}(\mathbf{w}^T \mathbf{x}_i) = y_i$, thì không cần làm gì.
 - Nếu \mathbf{x}_i bị misclassified, cập nhật \mathbf{w} theo công thức

$$\mathbf{w} = \mathbf{w} + y_i \mathbf{x}_i$$

- Kiểm tra xem có bao nhiêu điểm bị misclassified. Nếu không còn điểm nào, dừng thuật toán. Nếu còn, quay lại bước 2.

2.2.5 Pseudocode

Algorithm: Perceptron Learning Algorithm

```
P ← inputs with label 1;  
N ← inputs with label 0;  
Initialize w randomly;  
while !convergence do  
    Pick random x ∈ P ∪ N ;  
    if x ∈ P and w.x < 0 then  
        | w = w + x ;  
    end  
    if x ∈ N and w.x ≥ 0 then  
        | w = w - x ;  
    end  
end  
//the algorithm converges when all the  
inputs are classified correctly
```

Hình 2.7: Mã giả Perceptron

2.2.6 Hiện thực hóa bằng code

Step 1: sinh data

```
# step 1: sinh data
means = [[1, 2], [5, 4]]
cov = [[.4, .2], [.2, .4]]
N = 80
X0 = np.random.multivariate_normal(means[0], cov, N).T
X1 = np.random.multivariate_normal(means[1], cov, N).T
X = np.concatenate((X0, X1), axis = 1)
y = np.concatenate((np.ones((1, N)), -1*np.ones((1, N))), axis = 1)
# vẽ các điểm trên tọa độ
import matplotlib.pyplot as plt
plt.plot(X[0][:N], X[1][:N], 'ro')
plt.plot(X[0][N:], X[1][N:], 'b^')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
# thêm X0=1 vào vector input X
X = np.concatenate((np.ones((1, 2*N)), X), axis = 0)
X = X.T
y = y.T
X0.shape, X1.shape
```

Hình 2.8: Tạo data ngẫu nhiên

step 2: xây dựng các hàm f, loss, derivative

```
# # step 2: xây dựng các hàm f, loss, derivative
# # hàm tính tất cả các điểm
# # X là ma trận chứa tất cả các điểm

# giá trị của y mà âm => -1
# giá trị của y mà dương => 1
def f(X, w):
    result=np.dot(X,w) # w0 + w1*x1 + w2*x2 = y_value
    y_pred = []
    # check nếu f(x) >= 0 thì label là 1 và ngược lại là -1
    for y_value in result:
        if y_value >= 0:
            y_pred.append(1)
        else:
            y_pred.append(-1)
    return np.array(y_pred).reshape(-1,1)
```

Hình 2.9: Hàm forward

```

# hàm loss
# loss dựa trên y_predict và y_true
def loss(X, w):
    y1 = f(X,w) # giá trị value thật
    y_pre = []
    y_new = []
    for index in range(len(y1)):
        # chỉ tính dựa trên các điểm bị classify sai
        if np.sign(y1[index]) != np.sign(y[index]):
            y_pre.append(y1[index]) # y predict
            y_new.append(y[index]) # y đúng
    t = np.sum(-np.multiply(y_pre,y_new))
    return t

```

```

# hàm derivative
def derivative(X):
    y_sub = -y
    d1 = np.dot(y_sub.T,X)
    return d1[0]

```

```

print(derivative(X))

print(y.T.shape,X.shape)

```

```

[ 0.          317.33763554 161.99148622]
(1, 160) (160, 3)

```

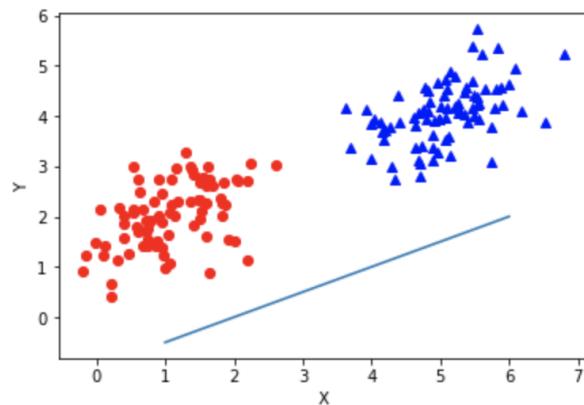
Hình 2.10: Hàm Loss & Derivative

Vẽ 2d

```
# # vẽ đường w1x1 + w2x2 + w0x0 = 0 ban đầu
w = [2,-1,2]
X1 = X.T[1:]
plt.plot(X1[0][:N],X1[1][:N],'ro') # X1[0][:N] are red dots
plt.plot(X1[0][N:],X1[1][N:],'b^') # X1[0][N:] are blue triangles
plt.xlabel('X')
plt.ylabel('Y')

x1 = np.linspace(start=1, stop=6, num=50)
x2 = (-w[0]-w[1]*x1)/w[2]
plt.plot(x1,x2)

plt.show()
```



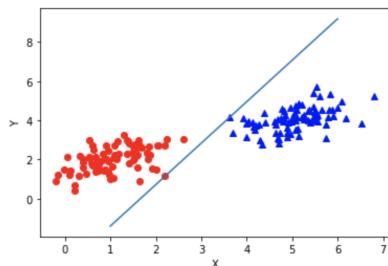
Hình 2.11: Đường phân loại trước khi training

```

w = np.ones(X.shape[1]).reshape(-1,1)
learning_rate = 0.1
epoch = 100
old_loss = -10
d = derivative(X).reshape(-1,1)
for i in range(epoch):
    for index in range(len(X)):
        Xi = np.copy(X[index]) # w0 + w1x1 + w2x2 + ...
        # print(Xi,Xi.shape)
        x = np.array(Xi.reshape(1,3))
        # print(x)
        y1 = np.dot(x,w)[0][0]
        # print(y1,y1.shape)
        # chả tính những point bị classify sai
        # => vì mình dùng 1 để thế hiện label cho nên nếu đã hết điểm bị classify sai thì loss sẽ = 0
        if old_loss != loss(X,w):
            old_loss = loss(X,w)
            draw(w)
            print("new_loss",old_loss,"time",i)
        if np.sign(y1) != np.sign(y[index]):
            w = w + learning_rate*(-y1*X.reshape(3,1))

```

new_loss 2.0 time 99



Hình 2.12: Đường phân loại sau khi training

Chương 3

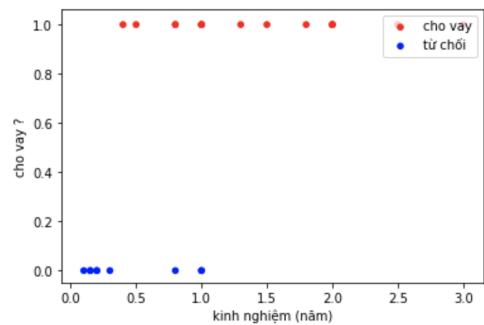
LOGISTIC REGRESSION

3.1 ĐỊNH NGHĨA

Ở chương trước chúng ta tìm hiểu về Linear Regression, đầu ra là một số thực để dự đoán một vấn đề gì đó như giá nhà, giá cổ phiếu,... Thì ở chương này chúng ta sẽ tìm hiểu thêm một loại hình regression nữa đó chính là Logistic Regression nhưng đầu ra có vẻ hơi khác biệt chút, vì kết quả là một kết quả nhị phân (giá trị 0 hoặc 1) thay vì một số thực như Linear Regression.

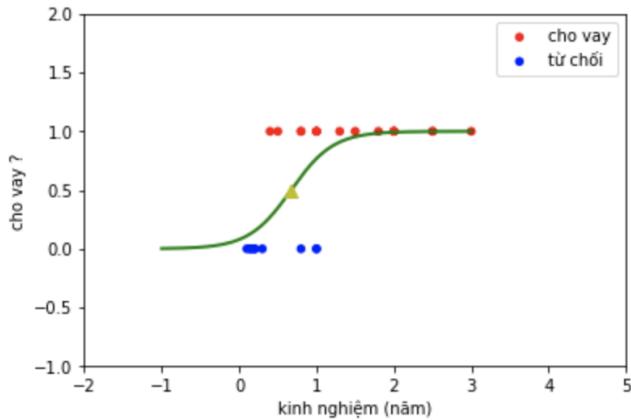
Thông thường nó được áp dụng để phân loại nhiều hơn là dự đoán nhưng tên gọi là là regression, các bài toán mà nó có thể giải quyết như là: phân biệt email rác hay không, dự đoán xem kết quả thi là trượt hay đậu, dự đoán khối u lành tính hay ác tính, dự đoán có nhiễm virus hay không

	kinh nghiệm (năm)	cho vay
0	1.00	1.0
1	2.00	1.0
2	1.80	1.0
3	1.00	1.0
4	2.00	1.0
5	0.50	1.0
6	3.00	1.0
7	2.50	1.0
8	1.00	1.0
9	2.50	1.0
10	0.10	0.0
11	0.15	0.0
12	1.00	0.0
13	0.80	0.0
14	2.00	1.0
15	1.00	0.0
16	1.50	1.0
17	1.30	1.0
18	0.20	0.0
19	0.15	0.0



Hình 3.1: Bảng dữ liệu cho vay tiền dựa vào số năm kinh nghiệm làm việc

Khi người làm việc muốn vay tiền với số năm làm việc của mình, thì ngân hàng sẽ kiểm tra bạn có đủ yêu cầu cho vay hay không? Điều chúng ta cần là tìm một đường vẽ có thể phân chia được hai vùng dữ liệu xanh - người không cho vay, đỏ - người cho vay.



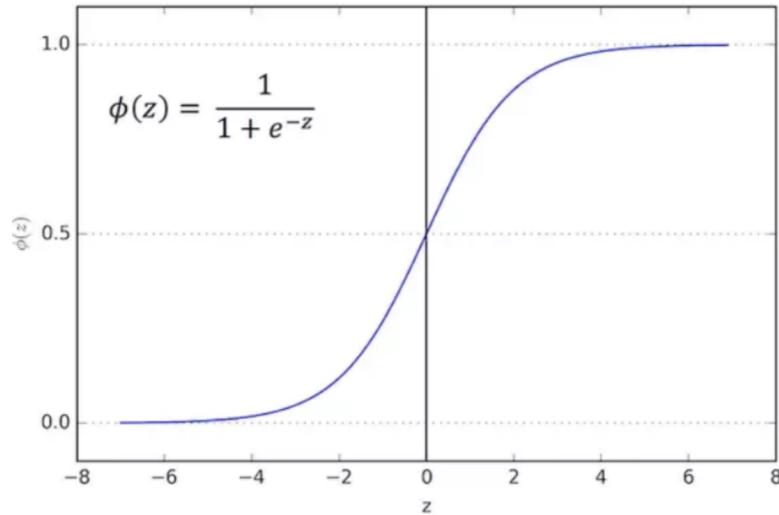
Hình 3.2: Đường vẽ cần tìm trong mô hình

3.2 HÀM SIGMOID

Với bài toán này đầu ra chúng ta cần tìm ra được xác suất dự đoán xem có nên cho người đó vay với số năm kinh nghiệm đó không, tức đầu ra sẽ nằm trong khoảng $[0, 1]$ vì nó là xác suất.

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

Hàm sigmoid là một hàm với đầu ra của nó luôn luôn nằm trong khoảng $[0, 1]$ nên sẽ rất phù hợp với bài toán trên.



Hình 3.3: Đồ thị giá trị của hàm sigmoid

Ta chọn hàm sigmoid vì nó có 2 tính chất:

- Hàm số liên tục trong khoảng giá trị $[0, 1]$
- Vì nó liên tục, nên có đạo hàm tại mọi điểm \Rightarrow dễ dàng áp dụng Gradient Descent

3.3 XÂY DỰNG MÔ HÌNH

1. Dựng mô hình
2. Thiết lập hàm mất mát - Loss function
3. Training mô hình, tìm trọng số thể hiện đường thẳng
4. Dự đoán dữ liệu mới vừa tìm được

3.3.1 Dựng mô hình

Dữ liệu đầu vào của chúng ta là số năm kinh nghiệm x^i tương ứng với người thứ i. Ta sẽ có:

- $p((x^i) == 1) = \hat{y}_i$ là xác suất CHO người thứ i vay
- $p((x^i) == 0) = 1 - \hat{y}_i$ là xác suất KHÔNG cho người thứ i vay

Ta thấy rằng $p((x^i) == 1) + p((x^{(i)}) == 0) = 1$, có nghĩa là với mỗi người thứ i sẽ có 2 xác suất được cho vay và không cho vay. Áp dụng hàm sigmoid

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

và linear regression

$$\hat{y}^{(i)} = w_0 + x^{(i)} * w_1$$

ta sẽ có công thức:

$$\hat{y}^{(i)} = \phi(z^{(i)}) = \phi(w_0 + x^{(i)} * w_1) = \frac{1}{1 + e^{-(w_0 + x^{(i)} * w_1)}}$$

3.3.2 Thiết lập hàm mất mát - Loss function

Sau khi đã có mô hình, chúng ta cần một hàm đánh giá để đánh giá kết quả tốt hay không, hàm đánh giá này sẽ đáp ứng được các điều kiện:

- Nếu người thứ i được cho vay, tức là $y^{(i)} = 1$ thì ta mong muốn xác suất dự đoán $\hat{y}^{(i)}$ của người đó càng cao càng tốt
- Nếu người thứ i không được cho vay, tức là $y^{(i)} = 0$ thì ta mong muốn xác suất dự đoán $\hat{y}^{(i)}$ của người đó càng thấp càng tốt

Hàm loss của Logistic Regression sẽ tương đối khác so với Linear Regression vì nó không phải là đo các giá trị thông thường, bởi vì đầu ra của chúng ta là xác suất nên chúng ta phải cần một hàm có thể đánh giá giữa hai phân phối xác suất mà cụ thể là toàn bộ xác suất giữa giá trị thực và giá trị dự đoán.

Vì vậy chúng ta sẽ sử dụng một phương pháp là Entropy để đo độ chính xác

$$\mathbf{H}(p, q) = - \sum_{i=1}^C p_i \log(q_i)$$

Vì đầu ra của Logistic Regression đầu ra gồm 2 loại: có hoặc không, cho nên nó sẽ rơi vào trường hợp đặc biệt của Entropy với N là số dòng dữ liệu:

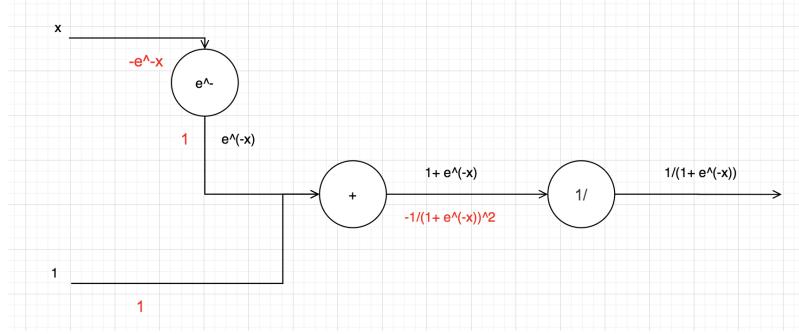
$$J(w) = - \sum_{i=1}^N (y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i))$$

Để tiện cho việc kí hiệu và tính toán ta quy định công thức

$$z = w_0 + w_1 * x$$

Và sigmoid của nó là

$$\hat{y} = \phi(z) = \phi(w_0 + w_1 * x)$$



Hình 3.4: Đạo hàm *sigmoid*

3.3.3 Training mô hình

Sau khi đã tìm được hàm Loss chúng ta sẽ tính đến Gradient Descent, ta thấy Loss tại từng dòng dữ liệu là:

$$L = -(y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i))$$

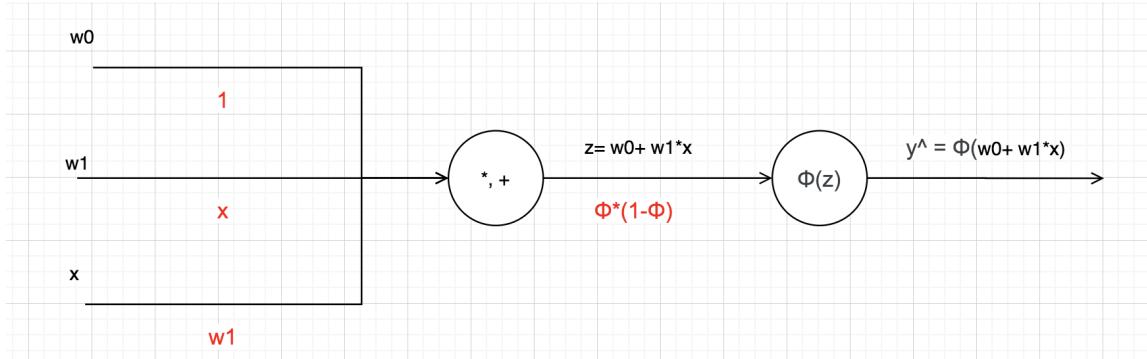
Ta cần tính đạo hàm L này trên từng w . Áp dụng công thức chain rule:

$$\frac{dL}{d\hat{y}_i} = \frac{dL}{d\hat{y}_i} * \frac{d\hat{y}_i}{dw}$$

Đạo hàm L trên \hat{y} :

$$\frac{dL}{d\hat{y}_i} = -\frac{(y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i))}{d\hat{y}} = -\left(\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i}\right) \quad (3.1)$$

Dựa vào quy luật chain rule ta tính đạo hàm của từng phương trình:



Hình 3.5: Đạo hàm của z

Dựa vào hình trên ta có thể thấy

$$\frac{d\hat{y}_i}{dw_0} = \frac{d\hat{y}_i}{dz_i} \frac{dz_i}{dw_0} = \phi(z_i) * (1 - \phi(z_i)) = \hat{y}_i * (1 - \hat{y}_i) \quad (3.2)$$

$$\frac{d\hat{y}_i}{dw_1} = \frac{d\hat{y}_i}{dz_i} \frac{dz_i}{dw_1} = x^{(i)}(\phi(z_i) * (1 - \phi(z_i))) = x^{(i)}(\hat{y}_i * (1 - \hat{y}_i)) \quad (3.3)$$

Từ (3.1), (3.2) và quy luật chain rule ta có thể suy ra công thức:

$$\frac{dL}{dw_0} = \frac{dL}{d\hat{y}_i} * \frac{d\hat{y}_i}{dw_0} = -\left(\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i}\right) * \hat{y}_i * (1 - \hat{y}_i) = \hat{y}_i - y_i$$

Từ (3.1), (3.3) và quy luật chain rule ta có thể suy ra công thức:

$$\frac{dL}{dw_1} = \frac{dL}{d\hat{y}_i} * \frac{d\hat{y}_i}{dw_1} = -\left(\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i}\right) * x^{(i)}\hat{y}_i * (1 - \hat{y}_i) = x^{(i)}(\hat{y}_i - y_i)$$

Trên đây mới là công thức dành cho một dòng dữ liệu, vậy toàn bộ dữ liệu sẽ như thế này

$$\frac{dL}{dw_0} = \sum_{i=1}^N \hat{y}_i - y_i$$

$$\frac{dL}{dw_1} = \sum_{i=1}^N x^{(i)}(\hat{y}_i - y_i)$$

* Biểu diễn dữ liệu dưới dạng ma trận

Cũng như Linear Regression, để thuận tiện cho việc tính toán ta sẽ sử dụng ma trận để tính toán trên toàn bộ dữ liệu thay vì phải tính từng dòng dữ liệu:

$$X = \begin{bmatrix} 1 & x^{(1)} \\ 1 & x^{(2)} \\ 1 & \dots \\ 1 & x^{(n)} \end{bmatrix}, y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix}, w = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

$$\hat{y} = \sigma(Xw)$$

$$J = -\text{sum}(y \otimes \log(\hat{y}) + (1 - y) \otimes \log(1 - \hat{y}))$$

$$\frac{dJ}{dw} = X^T * (\hat{y} - y), X^T = \begin{bmatrix} 1 & 1 & \dots & 1 \\ x^{(1)} & x^{(2)} & \dots & x^{(n)} \end{bmatrix}$$

Hình 3.6: Xử lí dữ liệu đầu vào với ma trận

3.3.4 Hiện thực hoá bằng code

```

if __name__ == '__main__':
    # Load data từ file csv: tiền lương và mức kinh nghiệm quyết định có được vay hay không
    data = pd.read_csv('logistic.csv').values
    N, d = data.shape
    # dữ liệu tiền lương và kinh nghiệm
    x = data[:, 1:2].reshape(-1, 1)
    # dữ liệu cho vay hay không 0 là không cho, 1 là cho vay
    y = data[:, 2].reshape(-1, 1)

    x_cho_vay = x[y[:, 0]==1]
    x_tu_choi = x[y[:, 0]==0]
    y_cho_vay = y[y[:, 0]==1]
    y_tu_choi = y[y[:, 0]==0]
    # Vẽ data bằng scatter
    plt.scatter(x_cho_vay[:, 0], y_cho_vay, c='red', edgecolors='none', s=30, label='cho vay')
    plt.scatter(x_tu_choi[:, 0], y_tu_choi, c='blue', edgecolors='none', s=30, label='từ chối')

    plt.legend(loc=1)
    plt.ylabel('cho vay ?')
    plt.xlabel('kinh nghiệm (năm)')

    # thêm 1 cột giá trị 1 vào đầu trước để nhân với w0 tiện tính toán thay vì phải tính bằng tay
    # ta tính bằng vector
    x = np.hstack((np.ones((N, 1)), x))

    # thực hiện model với lr là 0.01 và số vòng 10000
    model = LogisticModel(X=x, y=y, lr=0.01, num_iters = 10000)
    model.gradient_descent()

```

Hình 3.7: Xử lí dữ liệu đầu vào với ma trận

```

class LogisticModel:
    def __init__(self, X, y, y_pred=None, num_iters=100, lr=0.001):
        self.X = X
        self.y = y
        self.weight = np.zeros(X.shape[-1]).reshape(-1, 1)
        self.lr = lr
        self.num_iters = num_iters
        self.loss = np.zeros((num_iters, 1))
        self.y_predict = np.zeros((X.shape[0], 1))

    # hàm sigmoid
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def forward(self):
        return self.sigmoid(np.dot(self.X, self.weight))

    # đạo hàm theo w0
    def derivative_0(self):
        return np.sum(self.forward() - self.y)

    # đạo hàm theo wl
    def derivative_1(self):
        return np.sum(np.dot(self.X[:, 1].reshape(-1), self.forward() - self.y))

```

Hình 3.8: Xây dựng mô hình logistic

```

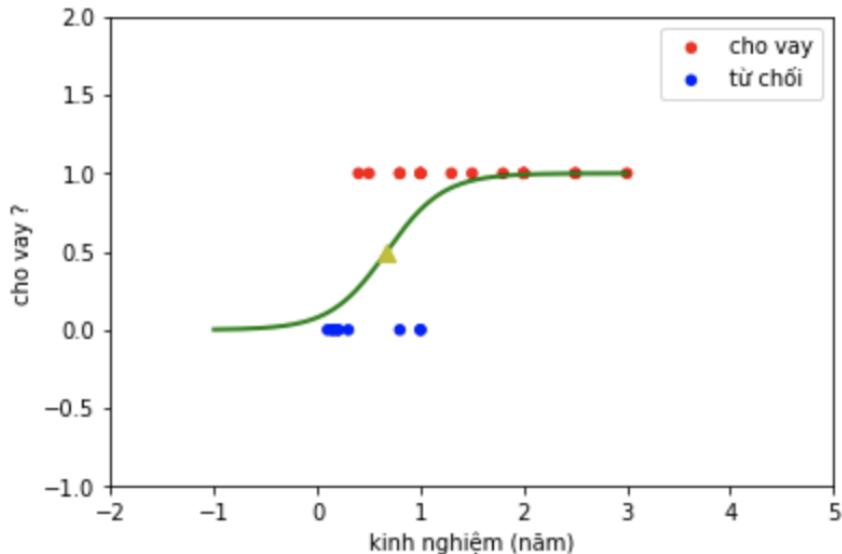
# hàm gradient_descend cập nhật giá trị dữ liệu cho w
def gradient_descend(self):
    for i in range(1, self.num_iters):
        # cập nhật loss
        self.loss[i] = -np.sum(np.multiply(self.y, np.log(self.forward())) +
                              np.multiply(1-y, np.log(1-self.forward())))
        # cập nhật weight
        self.weight[0] -= self.lr * self.derivative_0()
        self.weight[1] -= self.lr * self.derivative_1()

    # vẽ đường phân cách.
    print(self.weight)
    xx = np.linspace(0, 6, 300)
    threshold = -self.weight[0]/self.weight[1]
    yy = self.sigmoid(self.weight[0] + self.weight[1]*xx)
    plt.axis([-2, 5, -1, 2])
    plt.plot(xx, yy, 'g-', linewidth = 2)
    plt.plot(threshold, .5, 'y^', markersize = 8)
    plt.show()

```

Hình 3.9: Áp dụng gradient descent cập nhật dữ liệu

3.3.5 Dự đoán dữ liệu



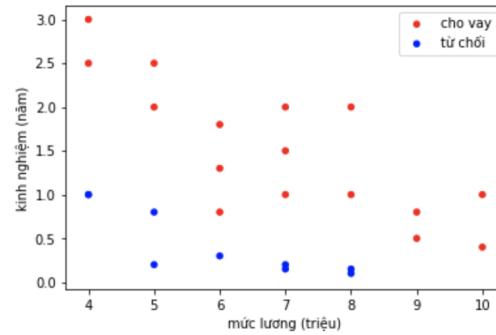
Hình 3.10: Vẽ đường thẳng dự đoán

3.4 ÁP DỤNG VÀO BÀI TOÁN NHIỀU BIẾN

Bên trên ta đã thấy được tác dụng của hàm sigmoid khi dự đoán người thứ i được vay hoặc không được vay dựa trên số năm kinh nghiệm, nhưng trên thực tế thì chúng ta không thể dựa vào một thuộc tính mà có thể đánh giá tổng thể cho nên chúng ta sẽ xử lý trên nhiều biến với nhiều thuộc tính. Ở đây em chỉ demo 2 biến là số năm kinh nghiệm và mức lương của người đó.

3.4.1 Dữ liệu đầu vào

	mức lương (triệu)	kinh nghiệm (năm)	cho vay
0	1.00	10.0	1.0
1	2.00	5.0	1.0
2	1.80	6.0	1.0
3	1.00	7.0	1.0
4	2.00	8.0	1.0
5	0.50	9.0	1.0
6	3.00	4.0	1.0
7	2.50	5.0	1.0
8	1.00	8.0	1.0
9	2.50	4.0	1.0
10	0.10	8.0	0.0
11	0.15	7.0	0.0
12	1.00	4.0	0.0
13	0.80	5.0	0.0
14	2.00	7.0	1.0
15	1.00	4.0	0.0
16	1.50	7.0	1.0
17	1.30	6.0	1.0
18	0.20	7.0	0.0
19	0.15	8.0	0.0



Hình 3.11: Bảng dữ liệu cho vay tiền dựa vào số năm kinh nghiệm làm việc và mức lương

Vì là nhiều biến nên dữ liệu khi visualize lên sẽ ở dạng nhiều chiều khác nhau, ở chiều không gian 2D này chúng ta nhìn thấy đường cần tìm là một đường thẳng thay vì đường cong trong khoảng $[0,1]$

3.4.2 Thiết lập công thức và áp dụng gradient descent

* Công thức

Việc bây giờ chúng ta sẽ thiết lập công thức cho hàm 2 biến. Để tiện cho việc kí hiệu và tính toán ta quy định công thức

$$z = w_0 + w_1 * x_1 + w_2 * x_2$$

Và sigmoid của nó là

$$\hat{y} = \phi(z) = \phi(w_0 + w_1 * x_1 + w_2 * x_2)$$

Loss tổng quát của hàm sigmoid là:

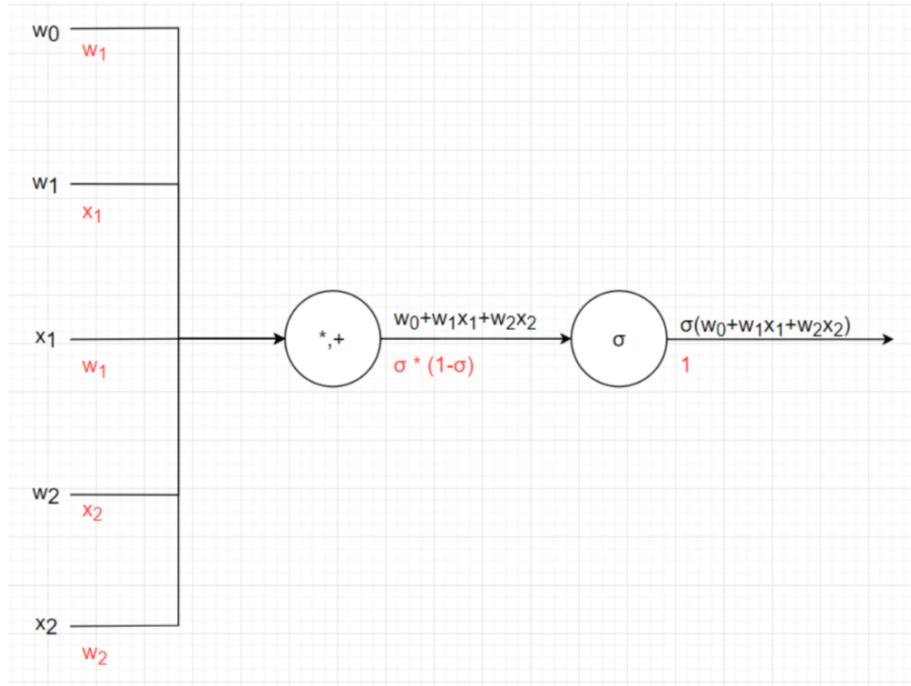
$$J(w) = - \sum_{i=1}^N (y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)) \quad (3.4)$$

* Gradient Descent

Ta cần tính đạo hàm L này trên từng w . Áp dụng công thức chain rule:

$$\frac{dL}{d\hat{y}_i} = \frac{dL}{d\hat{y}_i} * \frac{d\hat{y}_i}{dw}$$

Vì ở đây chúng ta có nhiều w hơn nên đạo hàm \hat{y} sẽ khác một chút:



Hình 3.12: Đạo hàm của z

Dựa vào hình trên ta có thể thấy

$$\frac{d\hat{y}_i}{dw_0} = \frac{d\hat{y}_i}{dz_i} \frac{dz_i}{dw_0} = \phi(z_i) * (1 - \phi(z_i)) = \hat{y}_i * (1 - \hat{y}_i) \quad (3.5)$$

$$\frac{d\hat{y}_i}{dw_1} = \frac{d\hat{y}_i}{dz_i} \frac{dz_i}{dw_1} = x_1^{(i)} (\phi(z_i) * (1 - \phi(z_i))) = x^{(i)} (\hat{y}_i * (1 - \hat{y}_i)) \quad (3.6)$$

$$\frac{d\hat{y}_i}{dw_2} = \frac{d\hat{y}_i}{dz_i} \frac{dz_i}{dw_2} = x_2^{(i)} (\phi(z_i) * (1 - \phi(z_i))) = x^{(i)} (\hat{y}_i * (1 - \hat{y}_i)) \quad (3.7)$$

Từ (3.4), (3.5) và quy luật chain rule ta có thể suy ra công thức:

$$\frac{dL}{dw_0} = \frac{dL}{d\hat{y}_i} * \frac{d\hat{y}_i}{dw_0} = -\left(\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i}\right) * \hat{y}_i * (1 - \hat{y}_i) = \hat{y}_i - y_i$$

Từ (3.4), (3.6) và quy luật chain rule ta có thể suy ra công thức:

$$\frac{dL}{dw_1} = \frac{dL}{d\hat{y}_i} * \frac{d\hat{y}_i}{dw_1} = -\left(\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i}\right) * x^{(i)} \hat{y}_i * (1 - \hat{y}_i) = x_1^{(i)} (\hat{y}_i - y_i)$$

Từ (3.4), (3.7) và quy luật chain rule ta có thể suy ra công thức:

$$\frac{dL}{dw_2} = \frac{dL}{d\hat{y}_i} * \frac{d\hat{y}_i}{dw_2} = -\left(\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i}\right) * x^{(i)} \hat{y}_i * (1 - \hat{y}_i) = x_2^{(i)} (\hat{y}_i - y_i)$$

Áp dụng trên toàn bộ dữ liệu sẽ như thế này

$$\frac{dL}{dw_0} = \sum_{i=1}^N \hat{y}_i - y_i$$

$$\frac{dL}{dw_1} = \sum_{i=1}^N x_1^{(i)} (\hat{y}_i - y_i)$$

$$\frac{dL}{dw_2} = \sum_{i=1}^N x_2^{(i)} (\hat{y}_i - y_i)$$

3.4.3 Biểu diễn bằng ma trận

Biểu diễn ở hai hay đa biến nó cũng không khác gì so với một biến

$$X = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} \\ 1 & \dots & \dots \\ 1 & x_1^{(n)} & x_2^{(n)} \end{bmatrix}, y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix}, w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$$

$$\hat{y} = \sigma(Xw)$$

$$J = -\text{sum}(y \otimes \log(\hat{y}) + (1 - y) \otimes \log(1 - \hat{y}))$$

$$\frac{dJ}{dw} = X^T * (\hat{y} - y), X^T = \begin{bmatrix} 1 & 1 & \dots & 1 \\ x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(n)} \\ x_2^{(1)} & x_2^{(2)} & \dots & x_2^{(n)} \end{bmatrix}$$

Hình 3.13: Xử lý dữ liệu đầu vào nhiều biến

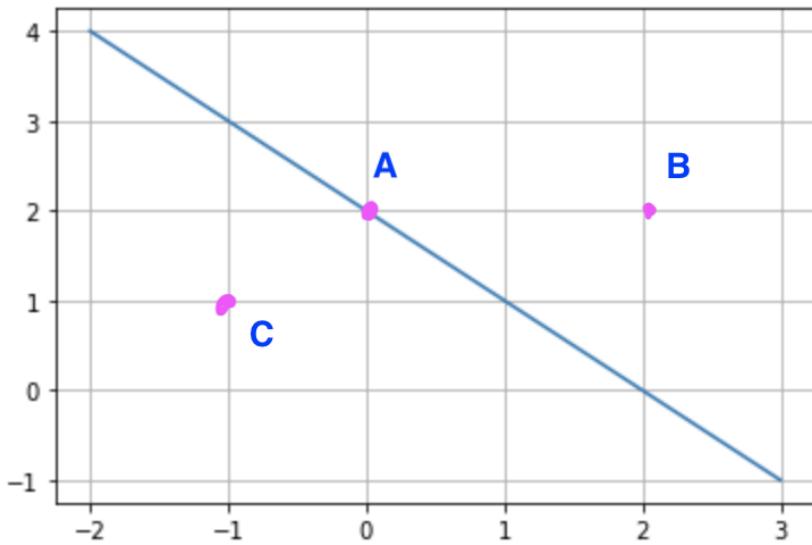
3.4.4 Quan hệ giữa đường thẳng và phần trăm

Vì bài toán đang xử lý là không gian 2 chiều nên chúng ta cần phải biết về quan hệ này để vẽ đường thẳng cho đầu ra. Xét 1 đường thẳng $y = ax + b$, gọi

$$\text{value} = y - (ax + b)$$

Ta thấy đường thẳng value này sẽ chia ra 2 miền dữ liệu

- Một miền dữ liệu là dương nếu giá trị value dương
- Miền còn lại là âm nếu value âm
- Những điểm nằm trên đường value này sẽ bằng 0



Hình 3.14: Đường thẳng $y = -x + 2$

Với đường thẳng $f = y - (-x + 2)$. Ta xét 3 điểm:

- Điểm $A(0, 2)$: $f = 0 \Rightarrow$ Điểm A thuộc đường thẳng
- Điểm $B(2, 2)$: $f = 2 \Rightarrow$ Điểm B miền trên có giá trị DƯƠNG
- Điểm $C(-1, 1)$: $f = -2 \Rightarrow$ Điểm C miền dưới có giá trị ÂM

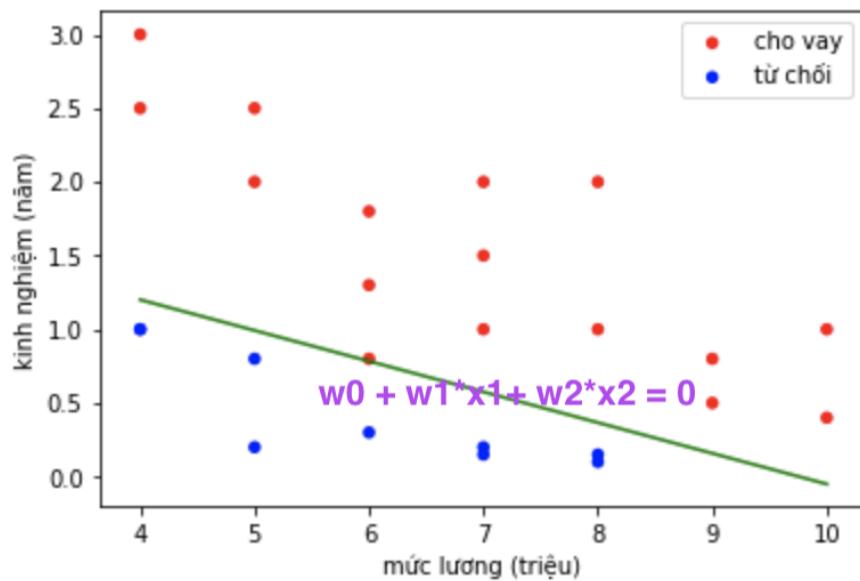
Áp dụng vào bài toán dự đoán trên, ta thấy nếu đầu ra của hàm sigmoid tức giá trị dự đoán $\hat{y} \geq 0.5$ ta sẽ cho vay nếu ngược lại không cho vay. Thê nên:

$$\begin{aligned}
& \hat{y} \geq 0.5 \\
\Leftrightarrow & \phi(w_0 + w_1 * x_1 + w_2 * x_2) \geq 0.5 \\
\Leftrightarrow & \frac{1}{1 + e^{-(w_0 + w_1 * x_1 + w_2 * x_2)}} \geq 0.5 \\
\Leftrightarrow & 1 + e^{-(w_0 + w_1 * x_1 + w_2 * x_2)} \leq 2 \\
\Leftrightarrow & e^{-(w_0 + w_1 * x_1 + w_2 * x_2)} \leq 1 \\
\Leftrightarrow & e^{-(w_0 + w_1 * x_1 + w_2 * x_2)} \leq e^0 \\
\Leftrightarrow & -(w_0 + w_1 * x_1 + w_2 * x_2) \leq 0 \\
\Leftrightarrow & w_0 + w_1 * x_1 + w_2 * x_2 \geq 0
\end{aligned}$$

Tương tự với $\hat{y} < 0.5$ ta cũng có:

$$\begin{aligned}
& \hat{y} < 0.5 \\
\Leftrightarrow & \phi(w_0 + w_1 * x_1 + w_2 * x_2) < 0.5 \\
\Leftrightarrow & \frac{1}{1 + e^{-(w_0 + w_1 * x_1 + w_2 * x_2)}} < 0.5 \\
\Leftrightarrow & 1 + e^{-(w_0 + w_1 * x_1 + w_2 * x_2)} > 2 \\
\Leftrightarrow & e^{-(w_0 + w_1 * x_1 + w_2 * x_2)} > 1 \\
\Leftrightarrow & e^{-(w_0 + w_1 * x_1 + w_2 * x_2)} > e^0 \\
\Leftrightarrow & -(w_0 + w_1 * x_1 + w_2 * x_2) < 0 \\
\Leftrightarrow & w_0 + w_1 * x_1 + w_2 * x_2 < 0
\end{aligned}$$

Từ đó ta có thể suy ra đường thẳng $w_0 + w_1 * x_1 + w_2 * x_2 = 0$ chính là đường phân chia các điểm thể hiện hồ sơ cho vay hoặc không cho vay.



Hình 3.15: Đường thẳng $w_0 + w_1 * x_1 + w_2 * x_2 = 0$

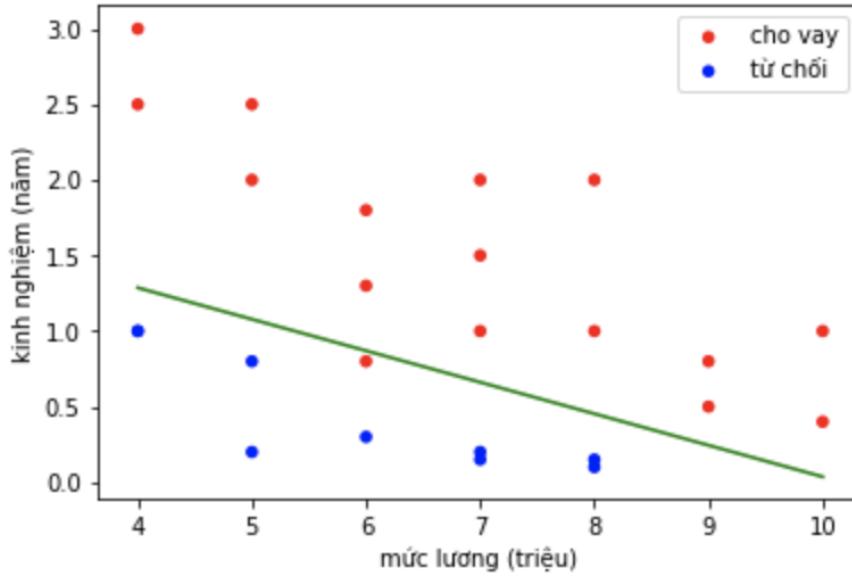
Khi nay ta lây u xác suất là 50% để thể hiên nếu giá trị dự đoán hơn 0.5 thì sẽ cho hồ sơ đó vay, cho nên trong trường hợp tổng quát với u xác suất ta có thể ghi là

$$\hat{y} > u \Leftrightarrow w_0 + w_1 * x_1 + w_2 * x_2 > -\ln\left(\frac{1}{u} - 1\right)$$

Từ đó ta có thể suy ra giá trị cần tính của đường dựa vào u xác suất.

Khi $u = 0.7$ ta có:

$$\begin{aligned} w_0 + w_1 * x_1 + w_2 * x_2 &> -\ln\left(\frac{1}{0.7} - 1\right) \\ \Leftrightarrow w_0 + w_1 * x_1 + w_2 * x_2 &> 0.8472978604 \end{aligned}$$



Hình 3.16: Đường thẳng phân chia với $u = 0.7$

3.4.5 Hiện thực bằng code

```

if __name__ == '__main__':
    # Load data từ file csv: tiền lương và mức kinh nghiệm quyết định có được vay hay không
    data = pd.read_csv('logistic.csv').values
    N, d = data.shape
    # dữ liệu tiền lương và kinh nghiệm
    x = data[:, 0:d-1].reshape(-1, d-1)
    # dữ liệu cho vay hay không 0 là không cho, 1 là cho vay
    y = data[:, 2].reshape(-1, 1)

    # Vẽ data bằng scatter
    # tách x dữ vay, và không được vay để vẽ hình
    x_cho_vay = x[y[:, 0]==1]
    x_tu_choi = x[y[:, 0]==0]

    plt.scatter(x_cho_vay[:, 0], x_cho_vay[:, 1], c='red', edgecolors='none', s=30, label='cho vay')
    plt.scatter(x_tu_choi[:, 0], x_tu_choi[:, 1], c='blue', edgecolors='none', s=30, label='từ chối')
    plt.legend(loc=1)
    plt.xlabel('mức lương (triệu)')
    plt.ylabel('kinh nghiệm (năm)')

    # thêm 1 cột giá trị 1 vào đầu trước để nhân với w0 tiện tính toán thay vì phải tính bằng tay
    # ta tính bằng vector
    x = np.hstack((np.ones((N, 1)), x))

    # thực hiện model với lr là 0.01 và số vòng 10000
    model = LogisticModel(x=x,y=y,lr=0.01,num_iters = 10000)
    model.gradient_descent()

```

Hình 3.17: Xử lí dữ liệu đầu vào với ma trận

```

class LogisticModel:
    def __init__(self, X, y, y_pred=None, num_iters=100, lr=0.001):
        self.X = X
        self.y = y
        self.weight = np.zeros(X.shape[-1]).reshape(-1, 1)
        self.lr = lr
        self.num_iters = num_iters
        self.loss = np.zeros((num_iters, 1))
        self.y_predict = np.zeros((X.shape[0], 1))

    # hàm sigmoid
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def forward(self):
        return self.sigmoid(np.dot(self.X, self.weight))

    # đạo hàm theo w0
    def derative_0(self):
        return np.sum(self.forward() - self.y)

    # đạo hàm theo w1
    def derative_1(self):
        return np.sum(np.dot(self.X[:, 1].reshape(-1), self.forward() - self.y))

    # đạo hàm theo w2
    def derative_2(self):
        return np.sum(np.dot(self.X[:, 2].reshape(-1), self.forward() - self.y))

```

Hình 3.18: Xây dựng mô hình logistic với 2 biến

```

# hàm gradient_descend cập nhật giá trị dữ liệu cho w
def gradient_descend(self):

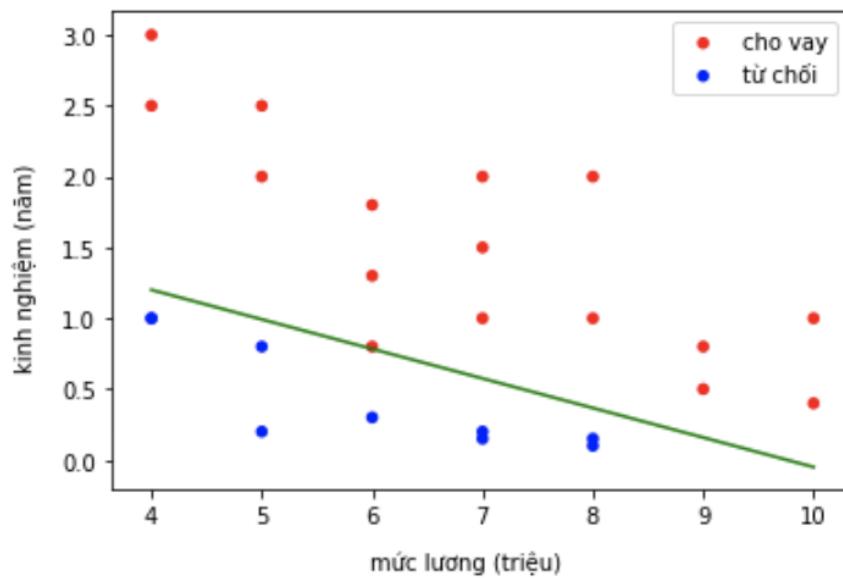
    for i in range(1, self.num_iters):
        # cập nhật loss
        self.loss[i] = -np.sum(np.multiply(self.y, np.log(self.forward())) +
                              np.multiply(1-y, np.log(1-self.forward())))
        # cập nhật weight
        self.weight[0] -= self.lr * self.derative_0()
        self.weight[1] -= self.lr * self.derative_1()
        self.weight[2] -= self.lr * self.derative_2()

    # vẽ đường phân cách.
    print(self.weight)
    t = 0.5
    plt.plot((4, 10),(-(self.weight[0]+4*self.weight[1]+ np.log(1/t-1))/self.weight[2], -(self.weight[0] + 10
    plt.show()
    print(self.loss)
    plt.plot(np.linspace(start=1, stop=10000, num=9999), self.loss[1:], 'r-')
    plt.show()

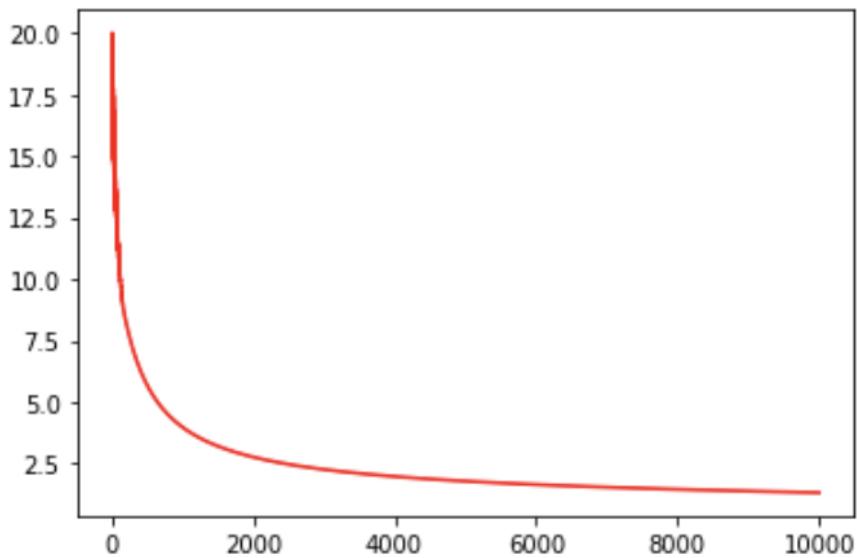
```

Hình 3.19: Áp dụng gradient descent cập nhật dữ liệu

3.4.6 Dự đoán dữ liệu



Hình 3.20: Vẽ đường thẳng dự đoán



Hình 3.21: Loss theo epoch

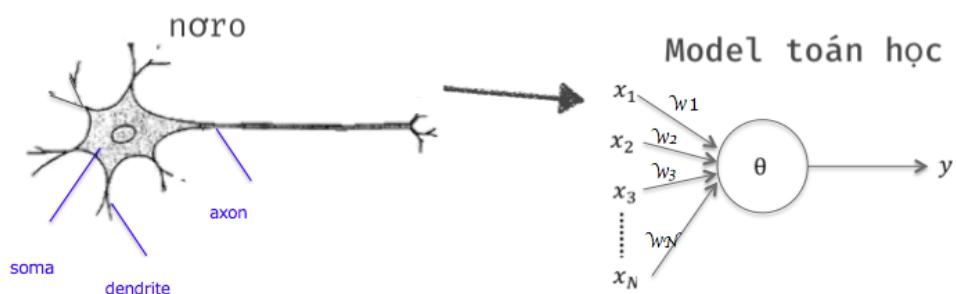
Chương 4

NEURAL NETWORK

4.1 Khái niệm

4.1.1 Noron trong học máy

Neural là mô hình toán học mô phỏng nơron trong hệ thống thần kinh con người. Model đó biểu hiện cho một số chức năng của nơron(neuron) thần kinh con người.



Hình 4.1: Mô hình noron cơ bản

Dầu tiên là tính chất truyền đi của thông tin trên neuron, khi neuron nhận tín hiệu đầu vào từ các dendrite, khi tín hiệu vượt qua một ngưỡng(threshold) thì tín hiệu sẽ được truyền đi sang neuron khác (Neurons Fire) theo sợi trực(axon). Neural của model toán học ở đây cũng được mô phỏng tương tự như vậy. Công thức tính output y sẽ như sau:

$$y = a(w_1x_1 + w_2x_2 + w_3x_3 - \theta)(1)$$

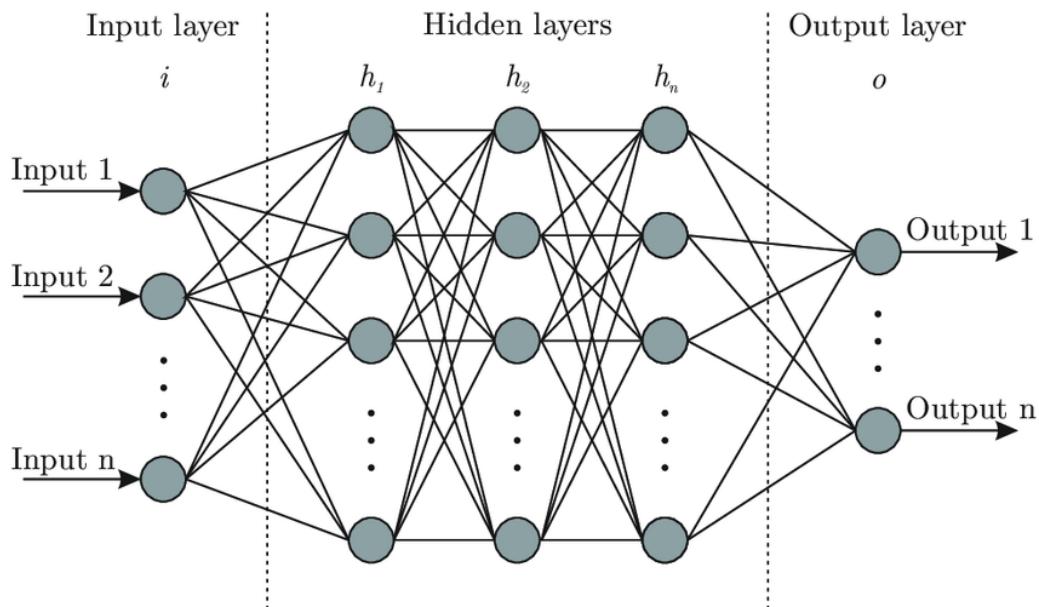
a là một function, còn được gọi là activation function, có nhiệm vụ là chuẩn hoá output. Nếu như công thức trên bỏ đi activation function thì output y sẽ là 1 giá trị không có giới hạn (-inf -> inf). Activation Function có các đại diện tiêu biểu như:

- Step function
- Linear function
- Sigmoid function
- Tanh function
- ReLu

Từ công thức (1), thực tế threshold trong phạm vi toán học có thể mang cả dấu (-) và (+) nên các bậc đầu to hơn bình thường 1 chút đã đưa vào thuật ngữ $bias = b = -\theta$. . Đơn giản ta sẽ có công thức sau:

$$y = a(w_1x_1 + w_2x_2 + w_3x_3 + b)(2)$$

4.1.2 Neural Network là gì ?



Hình 4.2: Mô hình Neural Network

Neural Network là sự kết hợp từ nhiều unit. Và tín hiệu sẽ được xử lý theo từng tầng(layer), tầng ở giữa được gọi là tầng ẩn(hidden layer), còn lại là tầng input và output.

Tầng sau sẽ nhận giá trị output của tầng trước để tiến hành xử lý. Còn xử lý ra sao là một chuyện khác, phụ thuộc vào từng bài toán mà công việc xử lý sẽ khác nhau. Và số lượng Hidden layer là không giới hạn, việc lựa chọn số tầng ẩn và cách xử lý ở mỗi tầng là chuyện không hề đơn giản. Nhưng đều có công thức cơ bản như (2) chỉ khác là thay đổi Activation function, Input Nói chung công thức ở trên là công thức tổng quát.

- **INPUT LAYER:** Đây là feature đầu tiên của mạng neural, dùng để input data hoặc feature.

- **HIDDEN LAYER:** Một feedforward network bao gồm một chuỗi các function. Bởi có nhiều hidden layers, chúng ta có thể mô tả hay tính toán những hàm số phức tạp bằng cách kết hợp các layer và nhiều hàm số đơn giản. Ví dụ chúng ta muốn mô phỏng hàm số mũ 7, ta có thể sử dụng những hàm đơn giản như square hoặc cube để tạo ra hàm số này. Trong hidden layer, active function được sử dụng phổ biến là hàm ReLU (Rectified Linear Unit).
- **OUTPUT LAYER:** Đây là layer đưa ra kết quả dự đoán. Với những bài toán khác nhau thì active function ở layer cũng sẽ khác nhau. Ví dụ với bài toán binary classification chúng ta có output là 0 hoặc 1, do đó active function là sigmoid function. Với bài toán dạng multiclass classification, active function được sử dụng là Softmax. Trong trường hợp là regression, output không phải phân chia vào category, ta có thể sử dụng linear unit.

4.2 Mô hình Neural Network

4.2.1 Mô hình Logistic Regression

Như phần Logistic ở phía trên chúng em đã trình bày nên ở đây thì chỉ nhắc sơ lại về công thức. Vì mô hình Logistic Regression là mô hình cơ bản nhất của mạng Neural Network, khi chỉ có Input, Output Layer chứ không có Hidden Layer nào. Và công thức dự đoán được tính như sau:

$$\hat{y}_i = \sigma(w_1 * x_1 + w_2 * x_2 + w_0)$$

Để có được công thức như trên chúng ta phải trải qua hai quá trình:

1. Tính tổng linear $z = w_1 * x_1 + w_2 * x_2 + w_0 * 1$

2. Áp dụng Activation Function cụ thể là Sigmoid: $\hat{y}_i = \sigma(z)$

Hệ số w_0 được gọi là *bias*. Trong các mô hình hệ số *bias* rất quan trọng, vì khi thiếu *bias* thì đường thẳng sẽ luôn luôn đi qua gốc tọa độ và nó không tổng quát hóa phương trình đường thẳng, khi đó chúng ta sẽ không tìm được phương trình mong muốn.

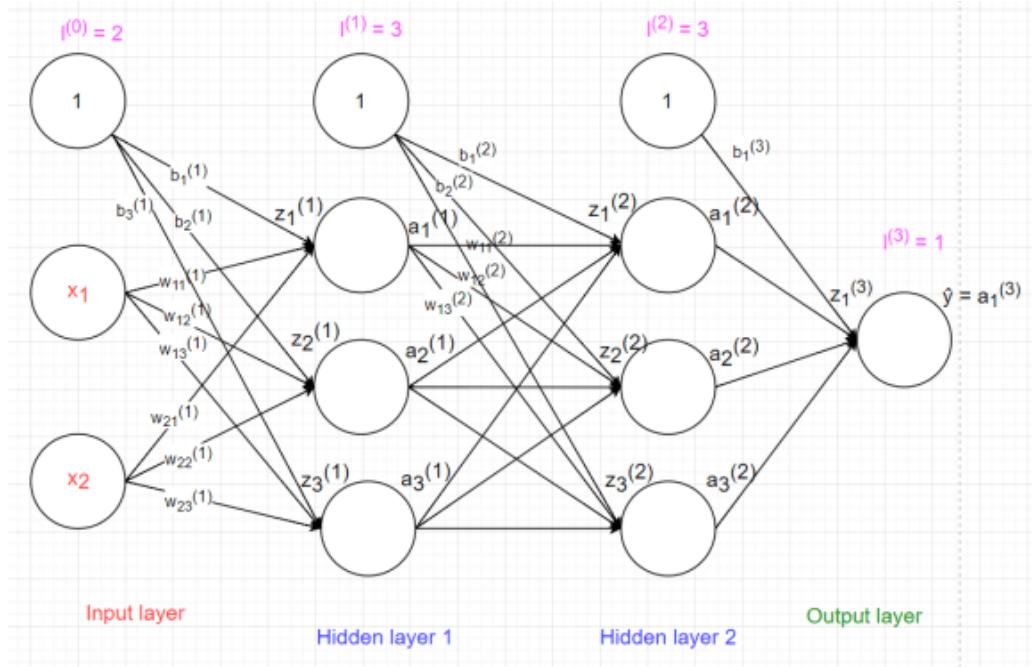
4.2.2 Công thức tổng quát

Như ta đã biết trong một mạng Neural Network thường có một hoặc nhiều hidden layer. Nhưng trong bài này chúng em chỉ tập trung vào đa hidden layer như đề bài Thầy yêu cầu.

Các tầng hidden layer liên kết với nhau thông qua output của node ở tầng layer ở phía trước làm input cho node và layer ở hiện tại. Trước khi đến các thuật toán để các node có thể liên kết với nhau một cách liền mạch thì chúng ta cần phải thống nhất và tìm hiểu rõ một số quy tắc và kí hiệu như sau:

- $l^{(i)}$: Số node trong hidden layer thứ i.
- $W^{(k)}$: ma trận trọng số weight có kích thước $l^{(k-1)} * l^{(k)}$.
- $w_{ij}^{(k)}$: trọng số weight kết nối từ node thứ i của layer k-1 đến node thứ j của layer k.
- $b^{(k)}$: là vector hệ số bias của các node trong layer k có kích thước $l^k * 1$.
- $b_i^{(k)}$: bias của node thứ i ở layer k.
- $z^{(k)}$: là vector chứa các giá trị của các node trong layer k sau bước tính tổng linear có kích thước $l^k * 1$.

- $a^{(k)}$: là vector chứa các giá trị của các node trong layer k sau khi áp dụng hàm activation function có kích thước $l^k * 1$.



Hình 4.3: Mô hình Neural Network chi tiết

Để dễ hiểu hơn, chúng em sẽ mô tả lại các bước thực hiện tính toán của các tầng hidden layer trong Neural Network trong hình ở phía trên.

Với node thứ i trong layer 1 có bias $b_i^{(l)}$ ta sẽ thực hiện 2 bước tính toán như sau:

1. Tính tổng linear: $z_i^{(l)} = \sum_{j=1}^{l^{(l-1)}} a_j^{(l-1)} * w_{ji}^{(l)} + b_i^{(l)}$, là tổng tất cả các output của các node ở layer trước nhân với hệ số w tương ứng, rồi cộng với bias b.
2. Áp dụng activation function: $a_i^{(l)} = \sigma(z_i^{(l)})$, cụ thể activation function trong bài này chúng em sử dụng sẽ là sigmoid.

Vậy khi áp dụng các bước trên tính toán tại node thứ 2 ở layer 1, ta có:

- $z_2^{(1)} = x_1 * w_{12}^{(1)} + x_2 * w_{22}^{(1)} + b_2^{(1)}$

- $a_2^{(1)} = \sigma(z_2^{(1)})$

Tương tự như vậy, ở node thứ 3 tại layer 2, ta có:

- $z_3^{(2)} = a_1^{(1)} * w_{12}^{(1)} + a_2^{(1)} * w_{23}^{(2)} + a_3^{(1)} * w_{33}^{(2)} + b_3^{(2)}$

- $a_3^{(2)} = \sigma(z_3^{(2)})$

4.2.3 Feedforward

Để dễ thông nhất, ta gọi input layer là $a^{(0)}$ (x) có kích thước $2*1$ vì chỉ có 2 input.

$$\begin{aligned} z^{(1)} &= \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \end{bmatrix} = \begin{bmatrix} a_1^{(0)} * w_{11}^{(1)} + a_2^{(0)} * w_{21}^{(1)} + a_3^{(0)} * w_{31}^{(1)} + b_1^{(1)} \\ a_1^{(0)} * w_{12}^{(1)} + a_2^{(0)} * w_{22}^{(1)} + a_3^{(0)} * w_{32}^{(1)} + b_2^{(1)} \\ a_1^{(0)} * w_{13}^{(1)} + a_2^{(0)} * w_{23}^{(1)} + a_3^{(0)} * w_{33}^{(1)} + b_3^{(1)} \end{bmatrix} \\ &= (W^{(1)})^T * a^{(0)} + b^{(1)} \\ a^{(1)} &= \sigma(z^{(1)}) \end{aligned}$$

Hình 4.4: Feedforward

Tương tự như vậy, ta có:

- $z^{(2)} = (W^{(2)})^T * a^{(1)} + b^{(2)}$

- $a^{(2)} = \sigma(z^{(2)})$

- $z^{(3)} = (W^{(3)})^T * a^{(2)} + b^{(3)}$

- $\hat{y} = a^{(3)} = \sigma(z^{(3)})$



Hình 4.5: Biểu diễn Feedforward

4.3 Hiện thực hóa bằng code

```

# định nghĩa hàm activation

# Hàm sigmoid
def sigmoid(x):
    return 1/(1+np.exp(-x))

# Đạo hàm hàm sigmoid
def sigmoid_derivative(x):
    return x*(1-x)
    
```

Hình 4.6: Hàm sigmoid & derivative sigmoid

```

class MyNeuralNetwork:
    def __init__(self, layers, x, y, y_predict=None, lr =0.1, num_iters=100):
        self.x = x
        self.y = y
        self.y_predict = y_predict
        self.lr = lr
        self.layers = layers
        self.num_iters = num_iters
        self.weight = []
        self.bias = []

        # Khởi tạo các tham số ở mỗi layer
        for i in range(0, len(layers)-1):
            _w = np.random.randn(layers[i], layers[i+1])
            _b = np.zeros((layers[i+1], 1))
            self.weight.append(_w/layers[i])
            self.bias.append(_b)

    def forward(self, x):
        y_pred = [x]
        next_layer = x
        # forward từ x cho đến layer cuối cùng
        for i in range(0, len(self.layers) - 1):
            next_layer = sigmoid(np.dot(next_layer, self.weight[i]) + (self.bias[i]).T)
            y_pred.append(next_layer)
        return y_pred

```

Hình 4.7: Khởi tạo class MyNeuralNetwork (1)

```

def backward(self, x, y_true, y_pred):
    y_true = y_true.reshape(-1, 1)
    dA = [-(y_true/y_pred[-1] - (1-y_true)/(1-y_pred[-1]))]
    dW = []
    db = []
    # backward từ layer cuối cùng về ngược layer đầu tiên
    # chúng ta đang dùng activate function là hàm sigmoid
    for i in reversed(range(0,len(self.layers)-1)):
        _w = np.dot((y_pred[i]).T, dA[-1] * sigmoid_derivative(y_pred[i+1]))
        _b = (np.sum(dA[-1] * sigmoid_derivative(y_pred[i+1]), 0)).reshape(-1,1)
        _A = np.dot(dA[-1] * sigmoid_derivative(y_pred[i+1]), self.weight[i].T)

        dW.append(_w)
        db.append(_b)
        dA.append(_A)

    # Đảo ngược dW, db
    dW = dW[::-1]
    db = db[::-1]

    return dW, db

def fit_each_epoch(self, x, y):
    y_pred = self.forward(x)
    dW, db = self.backward(x, y, y_pred)
    # print(dW, db)
    # Gradient descent
    for i in range(0, len(self.layers)-1):
        self.weight[i] = self.weight[i] - self.lr * dW[i]
        self.bias[i] = self.bias[i] - self.lr * db[i]

```

Hình 4.8: Khởi tạo class MyNeuralNetwork (2)

```

def train(self, epochs=20, verbose=10):
    for epoch in range(0, epochs):
        self.fit_each_epoch(self.X, self.y)
    if epoch % verbose == 0:
        loss = self.calculate_loss(self.X, self.y)
        print("Epoch {}, loss {}".format(epoch, loss))

# Dự đoán
def predict(self, X):
    for i in range(0, len(self.layers) - 1):
        X = sigmoid(np.dot(X, self.weight[i]) + (self.bias[i].T))
    return X

def draw(self):
    y_predict = self.forward(X)[-1]
    plt.plot(y_predict)
    t = 0.5
    plt.show()

# Tính loss function
def calculate_loss(self, X, y):
    y_predict = self.predict(X)
    return -(np.sum(y*np.log(y_predict) + (1-y)*np.log(1-y_predict)))

```

Hình 4.9: Khởi tạo class MyNeuralNetwork (3)

```

if __name__ == '__main__':
    # Dataset bài 2
    data = pd.read_csv('dataset.csv').values
    N, d = data.shape
    X = data[:, 0:d-1].reshape(-1, d-1)
    y = data[:, 2].reshape(-1, 1)

```

Hình 4.10: Import data

```

p = MyNeuralNetwork(layers = [X.shape[1], 3, 8, 10, 2, 1], X = X, y= y, lr = 0.01, num_iters=10000)
p.train(30001, 10000)

Epoch 0, loss 21.002696406583915
Epoch 10000, loss 18.75894229167623
Epoch 20000, loss 0.28468497317711156
Epoch 30000, loss 0.019691859722757304

```

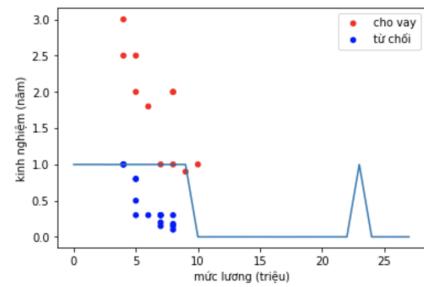
Hình 4.11: Training data

```

# Vẽ data bằng scatter
x_cho_vay = X[y[:,0]==1]
x_tu_choi = X[y[:,0]==0]

plt.scatter(x_cho_vay[:, 0], x_cho_vay[:, 1], c='red', edgecolors='none', s=30, label='cho vay')
plt.scatter(x_tu_choi[:, 0], x_tu_choi[:, 1], c='blue', edgecolors='none', s=30, label='tú chối')
plt.legend(loc=1)
plt.xlabel('mức lương (triệu)')
plt.ylabel('kinh nghiệm (năm)')
plt.draw()

```



Hình 4.12: Kết quả sau khi training data

Tài liệu tham khảo

[1] MachineLearningCoBan,

<https://machinelearningcoban.com/2017/01/21/perceptron/>

[2] Perceptron - Viblo,

<https://viblo.asia/p/perceptron-learning-algorithm-ByEZk0zgZQ0>

[3] CodeTuDau,

<https://codetudau.com/neural-network-va-deep-learning-la-gi/index.html>