

# *Logistic Regression*

Though this is called Regression, it falls under another category of **Supervised Learning** which is **Classification**.

## Classification

**Classification** is a process of categorizing a given set of data into **classes**. It can be performed on both structured or unstructured data. The process starts with predicting the class of given data points. The classes are often referred to as **targets, labels** or **categories**.

The classification predictive modeling is the task of approximating the mapping function from input variables to discrete output variables. The main goal is to identify which **class/category** the new data will fall into.

For example:

- Given the size of the cancer tumour, we can predict whether this is a benign or malignant tumour.
- Given the annual income of an individual, banks can predict whether that individual is capable of paying the debt.

As we can see here, the label that we want to predict take on a small number of discrete values. For **Binary Classification**, the label  $y$  only take on two values: 0 (malignant, cannot pay debt) and 1 (benign, can pay debt) .

We can attempt to use linear regression and map all predictions greater than 0.5 as 1 and all less than 0.5 as 0 . But this doesn't work well because classification is not actually a linear function.

## Multiclass Classification

**Multiclass Classification** occurs when the data set has more than two categories. So instead of  $y \in \{0, 1\}$  we will expand it so that  $y \in \{0, 1, 2, \dots, n\}$ .

Since we have  $n + 1$  discrete categories, we can divide our problems into  $n + 1$  **Binary Classification** problems; in each one, we predict the corresponding  $y$ . This approach is called **One-vs-All**.

$$y \in \{0, 1, 2, \dots, n\}$$

We perform **Logistic regression** for each category  $y$  to get:

$$f_{\theta}^{(0)} = P(y = 0|x; \theta)$$

$$f_{\theta}^{(1)} = P(y = 1|x; \theta)$$

$$f_{\theta}^{(2)} = P(y = 2|x; \theta)$$

...

$$f_{\theta}^{(n)} = P(y = n|x; \theta)$$

And then the prediction will be the best probability in all categories  $\max_i(f_{\theta}^{(i)}(x))$

Simply speaking, we are choosing one class and lumping all the others into a single second class. By repeating this process and apply binary logistic regression to each cases, and then use the hypothesis function that returned the highest value as our prediction (since with the given value  $x$ , the higher the odd, the more likely it is correct).

## Definition

### General formula


Since we want our hypothesis function to give certain discrete values (  $0 \leq f_{\theta}(x) \leq 1$  ), which cannot be achieved with normal linear function  $f_{\theta}(x) = \theta^T x$  . That is why we are using something called the **Sigmoid function**:

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

$$\Rightarrow f_{\theta}(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

This is the reason why **Logistic function** can be used interchangeably with **Sigmoid function**.

Here is how the Sigmoid function look like:

 Sigmoid function illustration

## Usage

This can be used to predict the probability that the label  $y = 1$  on input  $x$  . For example, if:

$$x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ \text{Tumor Size} \end{bmatrix}$$

$$\Rightarrow f_{\theta}(x) = 0.7$$

Then this tells that there is a 70% chance that the patient's tumor is malignant.

Statistically, this could be written as:

$f_{\theta}(x) = P(y = 1|x; \theta)$  Meaning: Probability that  $y = 1$ , given  $x$ , parameterized by  $\theta$

Since there are only two possible outcomes then

$$P(y = 0|x; \theta) + P(y = 1|x; \theta) = 1$$

# Decision Boundary

## Definition

To get our discrete 0 and 1 , we can translate the output of the hypothesis function as follows:

$$y = \begin{cases} 1, & \text{if } f_{\theta}(x) \geq 0.5 \\ 0, & \text{if } f_{\theta}(x) < 0.5 \end{cases}$$

We already know that if  $z \geq 0$  then our logistic function  $g(z) \geq 0.5$ .

So by assigning  $z = \theta^T X$ , we can conclude that  $f_{\theta}(x) = g(\theta^T X) \geq 0.5$  with  $\theta^T x \geq 0$ .

To sum up:

$$y = \begin{cases} 1, & \text{if } \theta^T X \geq 0 \\ 0, & \text{if } \theta^T X < 0 \end{cases}$$

The **decision boundary** is the line that separates the area where  $y = 0$  and where  $y = 1$ . It is created by our hypothesis function  $f_{\theta}(x)$ .

For example:

$$\theta = \begin{bmatrix} 5 \\ -1 \\ 0 \end{bmatrix} \quad X = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$$

$$\text{So } \theta^T x = 5 + (-1)x_1 + 0x_2$$

$$\text{And because } y = 1 \text{ if } \theta^T X \geq 0$$

$$\text{Then } 5 - x_1 \geq 0$$

$$\Rightarrow x_1 \leq 5$$

In this case, our decision boundary is a straight vertical line where  $x_1 = 5$ , and everything to the left of that denotes  $y = 1$ , while everything to the right denotes  $y = 0$ .

**One important note:** The input to the sigmoid function  $g(z)$  **does not** need to be linear. You can use polynomial features to fit the model better.

For example, it could be a function describe a circle  $z = \theta_0 + \theta_1 x_1^2 + \theta_2 x_2^2$

# Cost Function

## Definition

We cannot use the same cost function that we use for **Linear Regression** because the **Logistic Function** will cause the graph of the **Cost Function** to be wavy, resulting in many local optimas. In other words, it will not be a convex function. This is why we have to come up with another way to define the **Cost Function**.

The **Cost Function** for **Logistic Regression** looks like this:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{Cost}(f_{\theta}(x^{(i)}), y^{(i)})$$

With:

$$\text{Cost}(f_{\theta}(x^{(i)}), y^{(i)}) = \begin{cases} -\log(f_{\theta}(x)) & \text{if } y = 1 \\ -\log(1 - f_{\theta}(x)) & \text{if } y = 0 \end{cases}$$

Here I omitted the superscript  $^{(i)}$  to simplify it since it is the same for all training example.

From the given definition, we can conclude that:

$$\text{Cost}(f_{\theta}(x, y)) = 0 \text{ if } f_{\theta}(x) = y$$

$$\text{Cost}(f_{\theta}(x, y)) \rightarrow \infty \text{ if } y = 0 \text{ and } f_{\theta}(x) \rightarrow 1$$

$$\text{Cost}(f_{\theta}(x, y)) \rightarrow \infty \text{ if } y = 1 \text{ and } f_{\theta}(x) \rightarrow 0$$

The last two equations captures intuition that when the predicted label  $f_{\theta}(x)$  is wrong (opposite to  $y$ ), we will penalize the learning algorithm by a very large cost.

**For example**, if a patient with a malignant tumor was predicted as a benign one, the consequence would have been very large.

**By writing the Cost Function this way guarantees that  $J(\theta)$  is convex for Logistic Regression.**

## Simplifying the Cost Function

In reality, this **Cost Function** is too complicated since there are 2 cases. However, since  $y$  has strictly 2 discrete values: 0 and 1 so it can be simplified to this:

$$\text{Cost}(f_{\theta}(x, y)) = -y \log(f_{\theta}(x)) - (1 - y) \log(1 - f_{\theta}(x))$$

All right, it might look intimidating for you. But if you examine it closely, you will see that:

- If  $y = 0$  then  $\text{Cost}(f_{\theta}(x, y)) = -\log(1 - f_{\theta}(x))$
- If  $y = 1$  then  $\text{Cost}(f_{\theta}(x, y)) = -\log(f_{\theta}(x))$

It's that simple!

So now since we have the simplified  $\text{Cost}(f_{\theta}(x, y))$ . We can now plug it back into  $J(\theta)$  :

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \text{Cost}(f_{\theta}(x^{(i)}), y^{(i)}) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(f_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - f_{\theta}(x^{(i)}))]$$

This function has another name: **Binary Cross-Entropy Loss**.

The reason behind this **Cost Function** is that it can be derived from statistics using the **the Principle of Maximum Likelihood Estimation**, which is an idea in statistics for how to efficiently find parameters' data for different model. And it also has a nice property that it is **Convex**.

To fit the parameters  $\theta$ , we have to find the minimized **Cost Function**  $\min_{\theta} J(\theta)$ . After

minimized this, we will get the optimized parameters  $\theta$ , which could be put into the hypothesis function  $f_{\theta}(x)$  to make a prediction given a new  $x$ .

To achieve the optimal  $\theta$ , we will use some optimization algorithm. But for simplicity, we will use **Gradient Descent**.

## Vectorized Implementation

$$h = g(\theta X)$$

$$J = \frac{1}{m} \cdot (-y^T \log(h) - (1 - y)^T \log(1 - h))$$

With:

- $\theta$  is a vector with a dimension of  $n \times 1$  contains  $\theta_0, \theta_1, \dots, \theta_n$ .
- $X$  is a matrix with a dimension of  $m \times n$  contains every training examples on each row.
- $y$  is a vector with a dimension of  $n \times 1$  contains every training label on each row.

## Gradient Descent

### Definition

The algorithm is still the same:

repeat until convergence:

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \quad (\text{simultaneously update all } \theta_j)$$

Since

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(f_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - f_{\theta}(x^{(i)}))]$$

In which  $f_{\theta}(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n)$

The derivative now will be:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^m [(f_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}]$$

I won't prove since it is very long and confusing, but if you understand Calculus then you could prove it yourself.

So the **Gradient Descent** algorithm will become:

repeat until convergence:

$$\theta_j = \theta_j - \frac{\alpha}{m} \sum_{i=1}^m [(f_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}] \quad (\text{simultaneously update all } \theta_j)$$

If you take a closer look, you will see that the algorithm looks identical to **Linear Regression!!!** But it is not the same since the hypothesis in **Logistic Regression** is different from **Linear Regression**.

So even though the  $\theta$  – updating rule looks identical, it is not the same as the Gradient Descent for **Linear Regression** because the definition of the hypothesis function has changed.

## Vectorized Implementation

$$\theta = \theta - \frac{\alpha}{m} X^T (g(X\theta) - y)$$

## Regularized Gradient Descent for Logistic Regression

The principle is the same as before: add a regularizing term to the original **Cost Function** to penalize  $\theta$  parameters, and of course, we have to exclude the bias term  $\theta_0$ .

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left[ y^{(i)} \log(f_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - f_{\theta}(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

By taking partial derivative for each  $\theta_j$  and update  $\theta_0$  separately we have the regularized algorithm:

Repeat until convergence:

$$\begin{aligned} \theta_0 &= \theta_0 - \alpha \cdot \frac{1}{m} \sum_{i=1}^n \left[ (f_{\theta}(x^{(i)}) - y^{(i)}) x_0^{(i)} \right] \\ \theta_j &= \theta_j - \alpha \cdot \left\{ \frac{1}{m} \sum_{i=1}^n \left[ (f_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \right] + \frac{\lambda}{m} \theta_j \right\} \end{aligned}$$

## Other Optimization Algorithms

Aside from **Gradient Descent**, we have other more advanced algorithms such as: **Conjugate Gradient**, **BFGS**, **L-BFGS**. One of their advantages is that it does not need to manually choose the learning rate  $\alpha$ . But these are very complex algorithms to implement. So we should use predefined function from libraries.

## Demo Model

### Data Preprocessing

Read input, build DataFrame and plot data points.

```
In [ ]: import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```
plt.rcParams["figure.figsize"] = (10, 8)

header_names = ['Years of Experience', 'Salary', 'Loan']

df = pd.read_csv('../Training_set/loan_salary_data.csv', names=header_names)

m = len(df)
n = len(df.columns)

df.head()
```

```
Out[ ]:
```

	Years of Experience	Salary	Loan
0	6.3	10702	0
1	4.2	11180	0
2	6.1	11566	0
3	3.3	12669	0
4	5.1	13055	0

Now we implement  $\theta$  as a  $3 \times 1$  dimensions matrix,  $X$  as a  $m \times 3$  dimensions matrix by adding a bias term with all 1 and  $Y$  as a  $m \times 1$  dimensions matrix.

```
In [ ]:
```

```
data = np.array(df)

X = np.array(df.iloc[:, 0:2])
X = np.hstack((np.ones((m, 1)), X))
y = np.array(df.iloc[:, [2]])

theta = np.zeros((3, 1))

print(theta)
print(X[:5])
print(y[:5])
```

```
[[0.]
 [0.]
 [0.]]
[[1.0000e+00 6.3000e+00 1.0702e+04]
 [1.0000e+00 4.2000e+00 1.1180e+04]
 [1.0000e+00 6.1000e+00 1.1566e+04]
 [1.0000e+00 3.3000e+00 1.2669e+04]
 [1.0000e+00 5.1000e+00 1.3055e+04]]
[[0]
 [0]
 [0]
 [0]
 [0]]
```

We can see that the range of  $x_2$  is too large. That's why we need to normalize the data.

## Data Normalization

This time we will use Max - Min normalization. Recall that the formula for Max - Min normalization is:

$$x'_i = \frac{x_i - x_{min}}{\sigma}$$

```
In [ ]: def normalize(X) :  
        sigma = np.std(X, 0)  
        minX = np.min(X, 0)  
  
        X = (X - minX) / sigma  
  
        return X, sigma, minX
```

```
In [ ]: X[:, 1:], sigma, minX = normalize(X[:, 1:])  
  
print(X[:5])
```

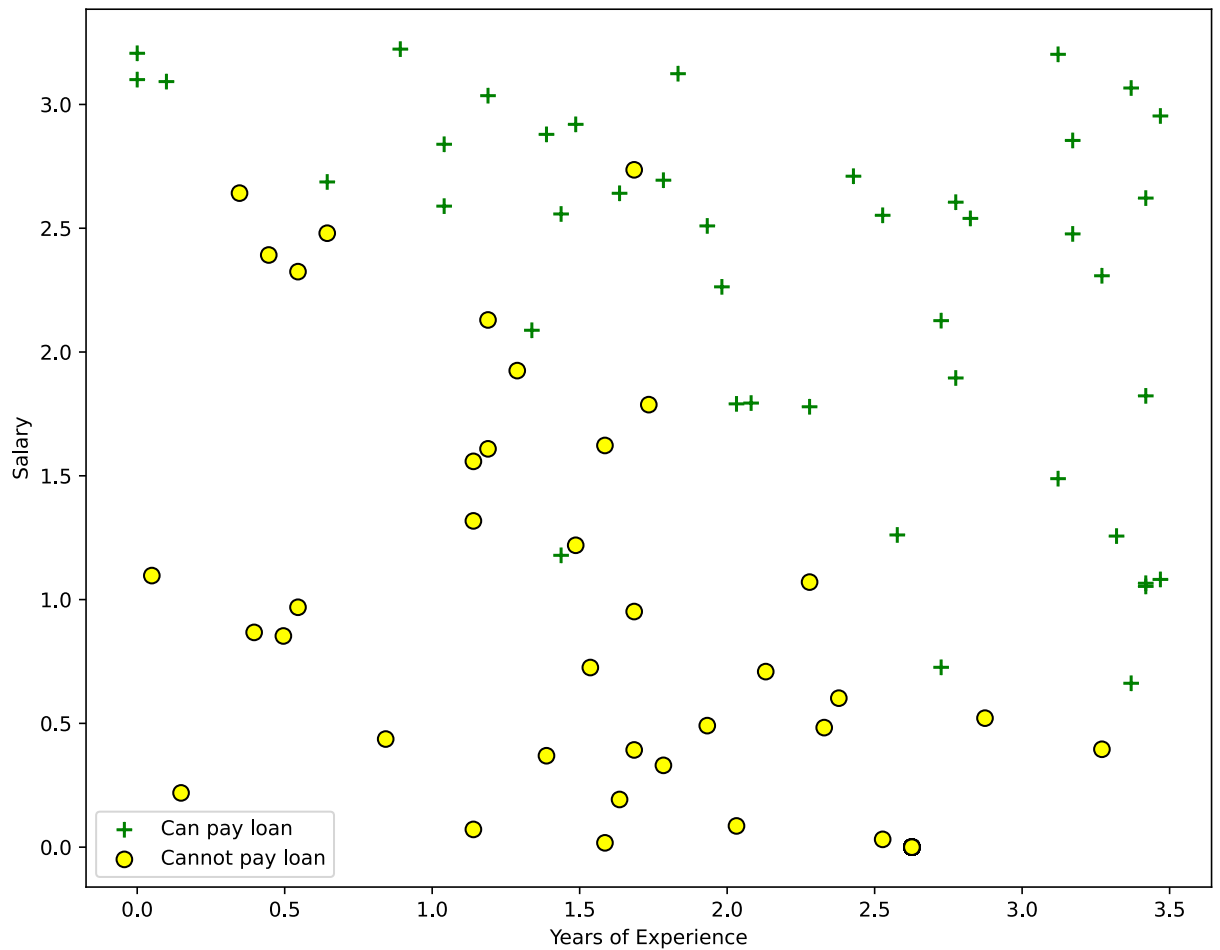
```
[[1.          2.62635637 0.          ]  
 [1.          1.5857246  0.01740621]  
 [1.          2.52724858 0.03146226]  
 [1.          1.13973956 0.07162763]  
 [1.          2.03170964 0.08568368]]
```

We will plot the data points onto a plot to have a better visualization.

```
In [ ]: def plotData(X, y):  
        positive = np.where(y == 1)  
        negative = np.where(y == 0)  
  
        plt.scatter(X[positive, -2], X[positive, -1], s=60, c='green', marker='.')  
        plt.scatter(X[negative, -2], X[negative, -1], s=60, c='yellow', marker='.')  
        plt.xlabel('Years of Experience')  
        plt.ylabel('Salary')  
        plt.legend(['Can pay loan', 'Cannot pay loan'])
```

```
In [ ]: plotData(X, y)
```





## Hypthesis, Sigmoid Function and Cost Function

### Sigmoid Function

Recall that the hypothesis is defined as:

$$f_{\theta}(x) = g(\theta^T X)$$

where function  $g$  is the Sigmoid Function. The Sigmoid Function is defined as:

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

So first we implement the Sigmoid Function:

```
In [ ]: def sigmoid(z) :
        return 1 / (1 + np.exp(-z))
```

### Cost Function and Gradient

Now we implement the Cost Function and Gradient for the Logistic Regression. Recall that the cost function in logistic function is:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left[ y^{(i)} \log(f_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - f_{\theta}(x^{(i)})) \right],$$

and the Gradient of the cost is a vector of the same length as  $\theta$  where  $j^{th}$  (for  $j \in [0, 1, \dots, n]$ ) is defined as:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^m \left( f_{\theta}(x^{(i)}) - y^{(i)} \right) x^{(i)}$$

So we will define a function that return the result of the Cost Function and the matrix contains the Gradient of each  $\theta$ .

```
In [ ]: def costFunction(theta, X, y) :
        m = len(y)
        z = np.dot(X, theta)

        f = sigmoid(z)

        J = (np.dot(y.T, np.log(f)) + np.dot((1 - y).T, np.log(1 - f))) / -m

        grad = np.dot(X.T, f - y) / m

        return J, grad
```

```
In [ ]: cost, grad = costFunction(theta, X, y)
        print('Cost at the initial theta (zeros):', cost[0][0])
```

Cost at the initial theta (zeros): 0.6931471805599452

```
In [ ]: print('Gradient of the initial theta:', grad)
```

Gradient of the initial theta:  $\begin{bmatrix} -0.0375 & \\ -0.26077737 & \\ -0.37936242 & \end{bmatrix}$

## Gradient Descent for Logistic Regression

Since our

```
def costFunction(theta, X, y)
```

returns the cost and the gradient for all  $\theta$ , we can use it to implement the gradient descent function. This function returns the optimized  $\theta$  and the vector **J\_arr** contains all descent steps that the algorithm performs each iteration.

```
In [ ]: def gradientDescent(theta, X, y, max_iterations=500, learning_rate=1.1, threshold=1e-10):
        J_history, grad = costFunction(theta, X, y)

        diff = 1e10
        i = 0

        while i < max_iterations:
            theta = theta - learning_rate * grad

            J, grad = costFunction(theta, X, y)

            J_history = np.vstack((J_history, J))

            i += 1

        return theta, J_history
```

We will now use gradient descent to optimize our  $\theta$ .

```
In [ ]: theta, J_arr = gradientDescent(theta, X, y)

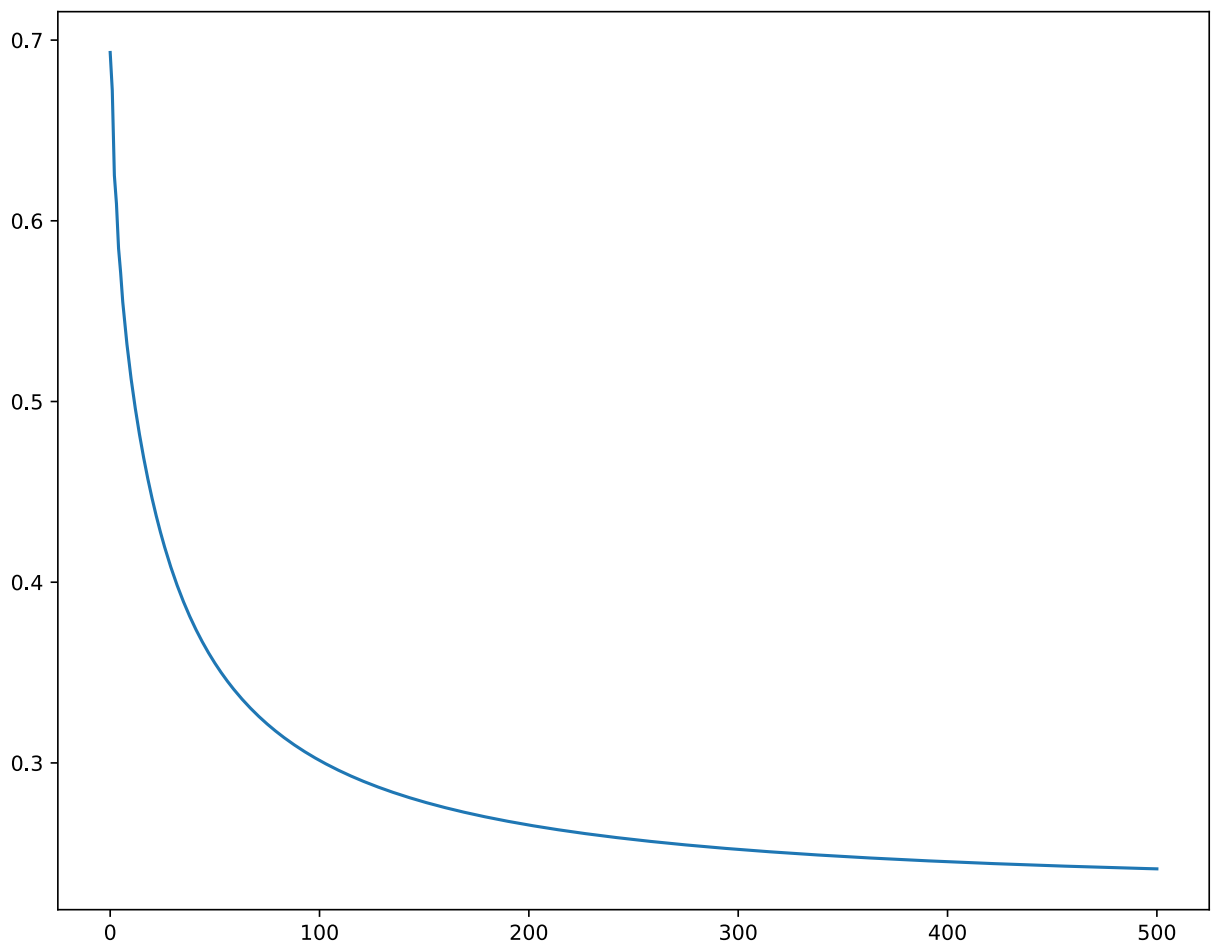
print('The optimized theta is:', theta)
print('The optimal cost is:', J_arr[-1])
```

```
The optimized theta is: [[-9.32828103]
 [ 2.37132467]
 [ 3.11076249]]
The optimal cost is: [0.24137698]
```

We will test if our learning rate  $\alpha$  is sufficient or not by graphing a graph of the cost with respect to each iteration.

```
In [ ]: plt.plot(range(len(J_arr)), J_arr)
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x7fb16f4511d0>]
```



We can now plot the decision boundary to make predictions.

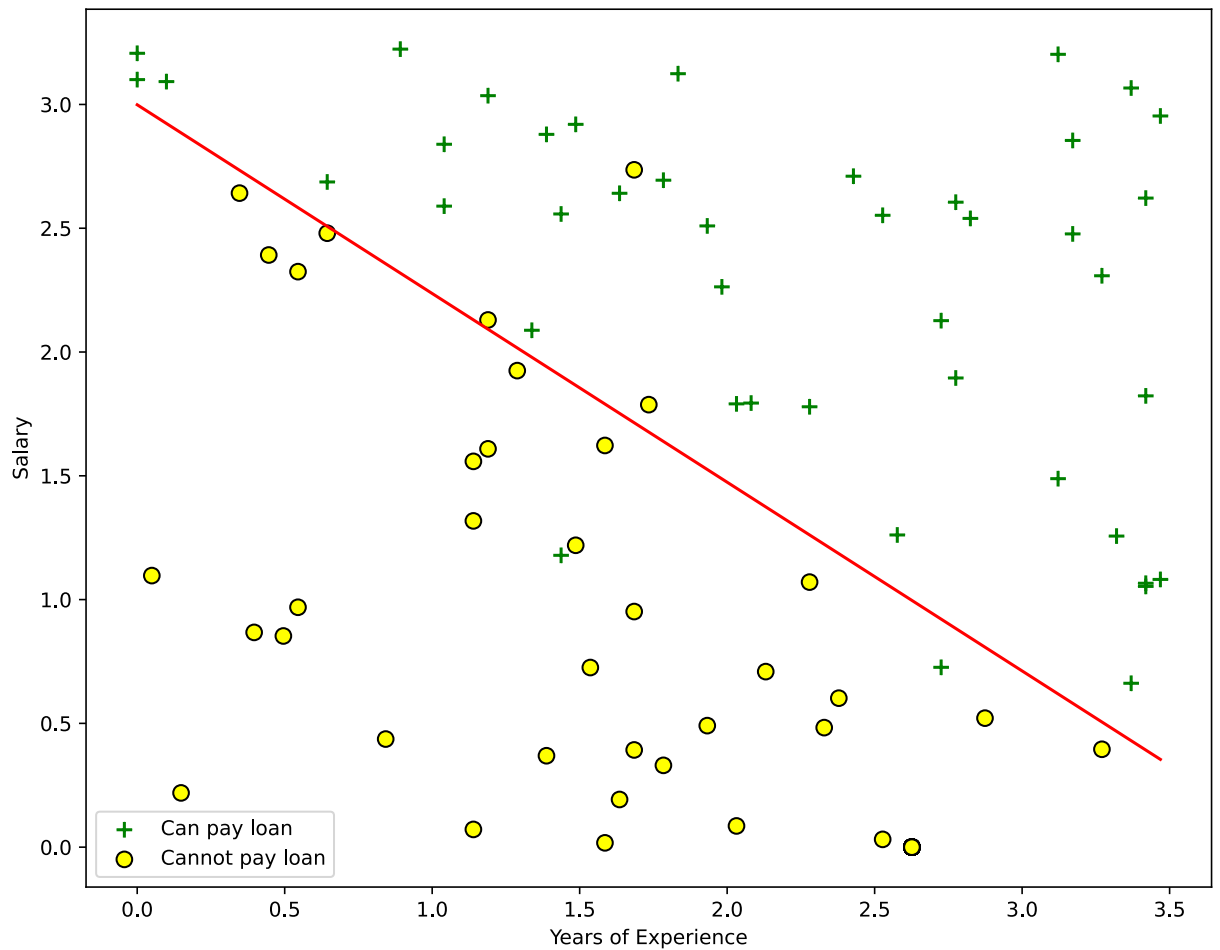
```
In [ ]: def plotDecisionBoundary(theta, X, y) :
        plotData(X, y)

        plotX = np.array([np.min(X[:, -2]), np.max(X[:, -2])])

        plotY = -(theta[0] + theta[1] * plotX) / theta[2]

        plt.plot(plotX, plotY, c='red', markersize=10)
```

```
In [ ]: plotDecisionBoundary(theta, X, y)
```



So we can see a line that separates two different labels. We can now start to make some predictions.

## Testing and Evaluating the Model

```
In [ ]: def predict(theta, X, sigma, minX) :
        X[:, 1:] = (X[:, 1:] - minX) / sigma
        return sigmoid(np.dot(X, theta))
```

```
In [ ]: np.random.seed(221)

x0_test = np.ones((5, 1))
x1_test = np.round(np.random.uniform(0, 8, 5).reshape((5, 1)), 1)
x2_test = np.random.randint(70000, 120000, 5).reshape((5, 1))

x = np.hstack((x0_test, x1_test, x2_test))
print("The inputs for testing:")
print("x0 =", x[:, 0])
print("x1 =", x[:, 1])
print("x2 =", x[:, 2])

predictions = predict(theta, x, sigma, minX)

print("The output is:", np.round(predictions[:, 0], 4))
```

```
The inputs for testing:
x0 = [1. 1. 1. 1. 1.]
x1 = [4.4 7.7 0.5 3.1 7.7]
x2 = [115374. 111823. 86240. 111722. 109074.]
The output is: [0.9985 1.      0.2044 0.9899 0.9999]
```

