Deep learning

Yanbo Liu

June 26, 2021

Contents

1 Linear regression

Let's say we have a lot of data points as shwon in figure 1, there are just some scatter points for use:

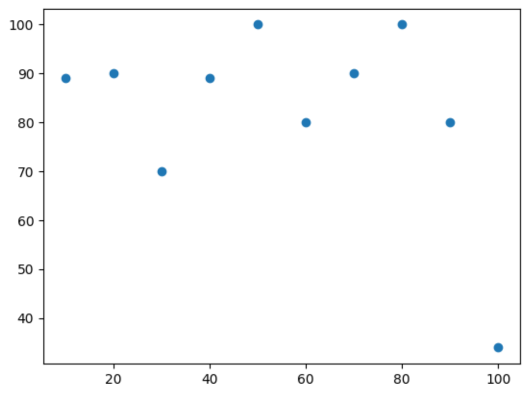


Figure 1: Scatter points

Then if we want to fit a line into this picture, let's say y=x as shown in figure 2:

How can we know that it is the best line that we want to fit into the scatter point? We can have a horizontal line at the scatter points as shown in figure 3, and caculate the distance between each point and the horizontal line and sum them up:

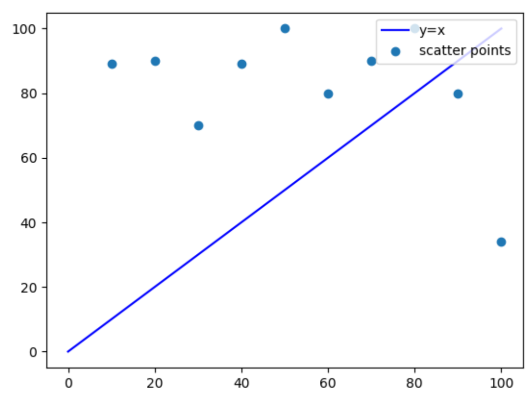


Figure 2: A line y=x fit in scatter points

In this figure the sum will be:

$$S = (b - y_1) + (b - y_2) + (b - y_3) + (b - y_4) + \cdots + (b - y_n) \tag{1}$$

where b is 84, as shown in figure 3, there are some points whose y value is bigger than b, which makes the value of $b - y$ is negative, which is not what we want since it will make the overall fit than it really is.

So we can square each term to ensure each term is positive:

$$S = (b - y_1)^2 + (b - y_2)^2 + (b - y_3)^2 + \cdots + (b - y_n)^2 \tag{2}$$

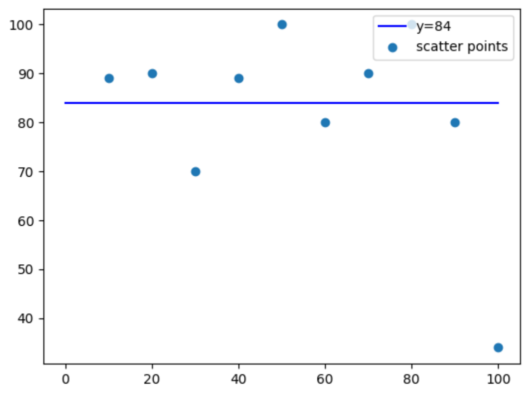


Figure 3: A line y=x fit in scatter points

This is our measure of how well this line fits the data.This is called **sum of squared residuals**, because the residuals are the differences between the real data and the line.In this case the final sum of squared residuals is 3362.

Let's get a new data set as shown in figure 4,in this case, the data point trend are much more smooth.So the fit is much better: 16

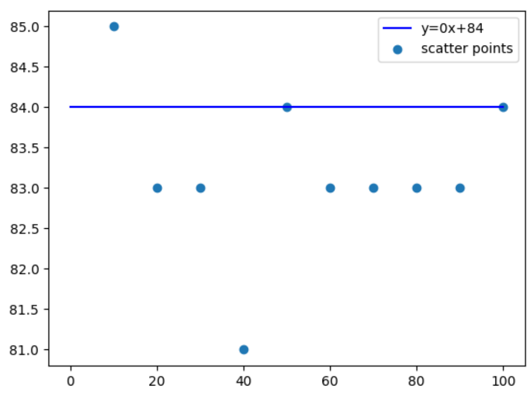


Figure 4: new data set

We can get ten different slop when the intercept doesn't change as shown in figure 5, the best point for the fit is when slope is 0 and intercept is 84 which is 16.

The best way to find the turning point is to find the point where the derivative of the function is 0.

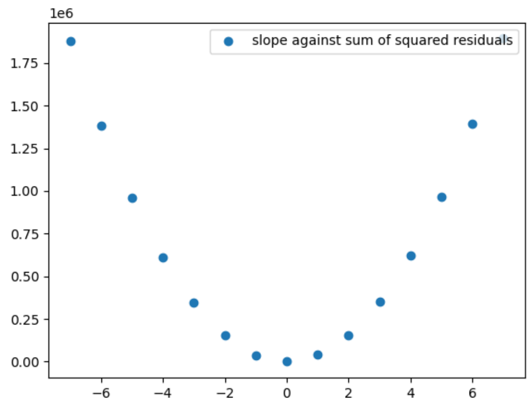


Figure 5: slope against sum of squared residuals

Corresponding code can be shown below:

```
import matplotlib.pyplot as plt
import numpy as np
girls_grades = [85, 83, 83, 81, 84, 83, 83, 83, 83, 84]
boys_grades = [30, 29, 49, 48, 100, 48, 38, 45, 20, 30]
grades_range = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

def function(data,data_range,a,b):
    #set the range of x
    x = np.linspace(0, 100, 100)
    y=a*x+b
    # fig=plt.figure()
    # if a==0:
    #     plt.plot(x,y,"-b",label="y={}".format(b))
    # else:
    #     plt.plot(x, y, "-b", label="y={}".format(b))
    # plt.scatter(data_range,data,label="scatter points")
    # plt.legend(loc="upper right")
    # plt.plot()
    # plt.show()
    sum=0
    for i in range(len(data_range)):
        x=data_range[i]
        y=a*x+b
        sum=sum+np.power(y-data[i],2)
    return sum



def linear_regression(slope_range,data,data_range,b):
    sum_lst=[]
    for i in range(len(slope_range)):
        print(i)
        sum_lst.append(function(data,data_range,slope_range[i],b))
    fig=plt.figure()
    plt.scatter(slope_range,sum_lst,label="slope against sum of squared residuals")
    plt.legend(loc="upper right")
    plt.plot()
    plt.show()
    print(sum_lst)
linear_regression([-6,-7,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7],girls_grades,grades_range,84)
```

But the situation above set the intercept to fixed value, the best fit will be found if we can use two independent variable they are slope and intercept of the function respectively as shown in figure 6

$$Q = \sum_{1}^{n} (y_i - (ax_i + b))^2 \tag{3}$$

To get the derivative:

$$\frac{\partial Q}{\partial b} = -2 \sum_{1} n(y_i - b - ax_i) = 0 \tag{4}$$

$$\frac{\partial Q}{\partial a} = -2a \sum_{1} n(y_i - b - ax_i)x_i = 0 \tag{5}$$

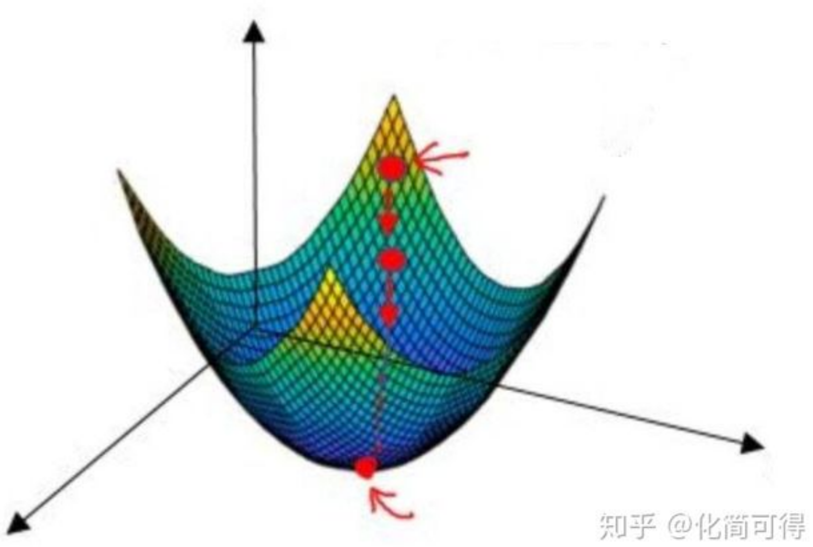


Figure 6: Two independent variables

2 Gradient Descent

When we fit a line with **Linear Regression**, we optimize **Intercept** and **Slope**.When we use **Logistic Regression**, we optimize a **squiggle**, when we use T-SNE, we optimize **clusters**

First of all,set the slope of the line to a fixed value, then change the intercept as shown in figure 7.In this example we will still need to evaluate how well these line fit the data by using sum of squared residuals.

$$Q = \sum_{1}^{n} (y_i - (ax_i + b))^2 \tag{6}$$

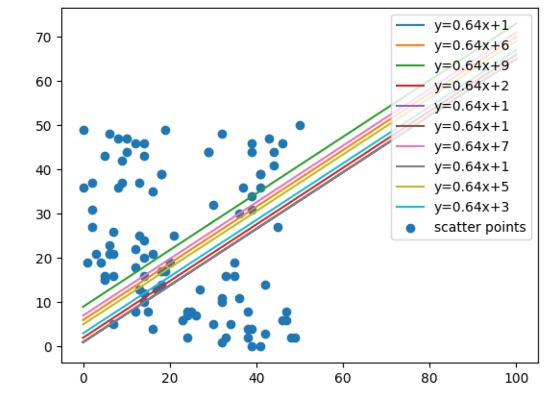


Figure 7: Changeable intercepts

In this case, relationship between sum of squared residuals and intercept is shown in figure 8.

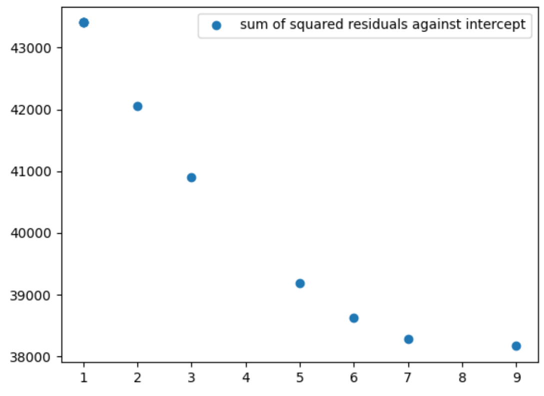


Figure 8: Sum of squared residuals against intercept

But this is quite inconvenient if we have to take a lot of steps. So, as we did before, we can find the derivative of the quardratic function with respect to intercept.So here we can use gradient descent to get the sum.

$$\frac{\partial Q}{\partial b} = -2 \sum_{1} n(y_i - b - ax_i) = 0 \tag{7}$$

$$\frac{\partial Q}{\partial a} = -2 \sum_{1} n(y_i - b - ax_i)x_i = 0 \tag{8}$$

Now that we have the derivative,gradient descent will use it to find where the sum of squared residuals is lowest.If we use **Linear Regression**,We just simply find the lowest slope of the curve. In contrast,**Gradient Descent** finds the minimum value by taking steps from initial guess until it reaches the best value,which make **Gradient Descent** very useful when it not very easy to find the point where the derivative is 0.

Also, the closer we get to the optimal value, the closer we get to the best value.That means if the slope is getting closer to 0 then we should use baby steps otherwise we should use big steps.

The **Step size** can be determined by multiplying the slope by **Learning Rate** as shown in figure 9:

And in this case, the Learning rate we use is 0.00001,otherwise it is diverged.

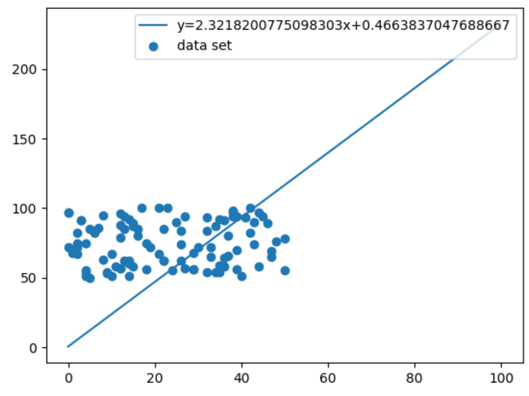


Figure 9: Gradient Descent

Summary 2.1 *So,we can use following orders to implement gradient descent:*

Step 1: Take the derivative of the ***Loss Function*** *for each parameter in it.In fancy Machine Learning Lingo,take the Gradient of the Loss function.*

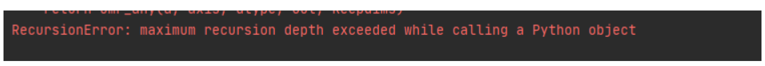
Step 2: Pick random values for the parameters

*Step 3: Plug the parameter values into the derivatives(**Gradient**)*

Step 4: Caculate the step sizes:Step size=Slope $*$ *LearningRate*

*Step 5: Caculate the new parameters:**New parameters=Old Parameter-Step Size***

Learning Rate *is quite important in gradient descent, if it was set to very high value,it will cause divergence as shown in figure 10:*



```
    return umr_any(a, axis, dtype, out, keepdims)
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Figure 10: Divergence of Gradient Descent

Corresponding code:

```
import matplotlib.pyplot as plt
import numpy as np
import random
grades=[]
grades_range=[]
for i in range(1000):
    grades.append(random.randint(50,100))
    grades_range.append(random.randint(0,50))
Learning_rate=0.001

# derivative of sum of squared residuals with respect to intercept
def d_intercept(data_y,b,a,data_x):
    return -2*(data_y-(b+a*data_x))

# derivative of sum of squared residuals with respect to slope
def d_slope(data_y,b,a,data_x):
    return -2*data_x*(data_y-(b+a*data_x))
a=10
b=0
# gradient descent using recursion
def gradient_descent_recursion(data_range,data,a,b,learning_rate,counter=0):
    count=0
    x=np.linspace(0,100,100)
    y=a*x+b
    # figure=plt.figure()
    # if a==0:
    #     plt.plot(x,y,label="y={}".format(b))
    # elif b==0:
    #     plt.plot(x,y,label="y={}x".format(a))
    # else:
    #     plt.plot(x, y, label="y={}x+{}".format(a,b))
    # plt.scatter(data_range,data,label="data set")
    # plt.legend(loc="upper right")
    # plt.plot()
    # plt.show()
    slope_slope=0
    slope_intercept=0
    for i in range(len(data)):
        slope_intercept=slope_intercept+d_intercept(data[i],b,a,data_range[i])
        slope_slope=slope_slope+d_slope(data[i],b,a,data_range[i])
    step_size_intercept=learning_rate*slope_intercept
    step_size_slope=learning_rate*slope_slope
    new_intercept=b-step_size_intercept
    new_slope=a-step_size_slope
    print("step_size_intercept:",step_size_intercept)
    print("step_size_slope:",step_size_slope)
    print("\n\n")
    print("counter:",counter)
    if abs(step_size_slope)<0.01 or abs(step_size_intercept)<0.001 or counter>1000:
        print("The final slope is:",a)
        print("The final intercept:",b)
        print("The count of recursion is :",counter)
        figure=plt.figure()
        if a==0:
            plt.plot(x,y,label="y={}".format(b))
        elif b==0:
            plt.plot(x,y,label="y={}x".format(a))
        elif b<0:
            plt.plot(x,y,label="y={}x-{}".format(a,abs(b)))
        else:
            plt.plot(x, y, label="y={}x+{}".format(a,b))
        plt.scatter(data_range,data,label="data set")
        plt.legend(loc="upper right")
        plt.plot()
        plt.show()
        return
    else:
        gradient_descent_recursion(data_range,data,new_slope,new_intercept,learning_rate,counter+1)


# gradient descent using iteration
def gradient_descent_iteration(data_range,data,a,b,learning_rate):
    temp_intercept=b
    temp_slope=a
    for i in range(1000):
        slope_slope = 0
        slope_intercept = 0
        for j in range(len(data)):
            slope_intercept = slope_intercept + d_intercept(data[j], temp_intercept, temp_slope, data_range[j])
            slope_slope = slope_slope + d_slope(data[j], temp_intercept, temp_slope, data_range[j])
        step_size_intercept = learning_rate * slope_intercept
        step_size_slope = learning_rate * slope_slope
        new_intercept = temp_intercept - step_size_intercept
        new_slope = temp_slope - step_size_slope
        temp_slope=new_slope
        temp_intercept=new_intercept
        if abs(step_size_slope) < 0.001 or abs(step_size_intercept) < 0.001:
            x = np.linspace(0, 100, 100)
            y = new_slope * x + new_intercept
            figure = plt.figure()
            if new_slope == 0:
                plt.plot(x, y, label="y={}".format(new_intercept))
            elif new_intercept == 0:
                plt.plot(x, y, label="y={}x".format(new_slope))
            elif new_intercept< 0:
                plt.plot(x, y, label="y={}x-{}".format(new_slope, abs(new_intercept)))
            else:
                plt.plot(x, y, label="y={}x+{}".format(new_slope, new_intercept))
            plt.scatter(data_range, data, label="data set")
            plt.legend(loc="upper right")
            plt.plot()
            plt.show()
            return
    print("The model is diverged!!!")



gradient_descent_iteration(grades_range,grades,a,b,Learning_rate)
gradient_descent_recursion(grades_range,grades,a,b,Learning_rate)
```

3 Stochastic Gradient Descent

Stochastic Gradient Descent is very similar to Gradient Descent,but instead of caculating sum of squared residuals it uses one sample or mini batch of samples per each step.It is very useful when we have a lot of datas and a lot of parameters.In these situations, regular gradient may not be computationally feasible.In supervised learning, eduach example is a pair consisting of an input object (typically a vector) and a **desired output** value (also called the supervisory signal). A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples.

3.1 Supervised Learning

Theorem 3.1 *Supervised learning (SL) is the machine learning task of learning a function that maps an input to an output based on example input-output pairs.[1] It infers a function from labeled training data consisting of a set of training examples.*

In Supervised learning, the correct output has already been given.It is basically a process of modifying the weight for each step. For Supervised learning, it should obeyed following order:

Step 1: Initialize the weights

Step 2: Caculate the error from the difference between output and correct output

Step 3: Caculate the weights updates.

Step 4: Adjust the weight updates

Step 5: Repeat step 2 to 4 for all training data

Step 6: Repeat step 2 to 5 until the error reaches an acceptable level.

Here is a simple figure of neural network as shown figure 10. And we just want to build a single layer neural network, so just ignore the Hidden layer.

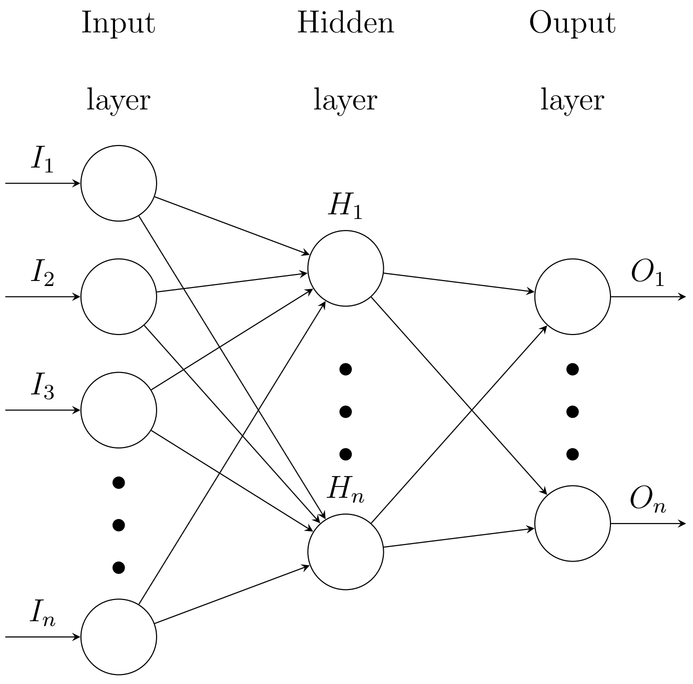


Figure 10:A simple Neural Network

For Supervised Learning, it is a process of modifying the weights for each step and the desired output has already been fixed which can be expressed as following equations:

$$e_i = d_i - y_i \tag{9}$$

$$\Delta w_{ij} = \alpha e_i x_j \tag{10}$$

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \tag{11}$$

These equations can be also called **Delta rule**where e is the error between the correct output and real output. And d is the correct output, y is the real output. α is the learning rate which is usually set between 0 and 1.

But the above equations are only correct when the activation function is linear function as shown in below in figure 11.

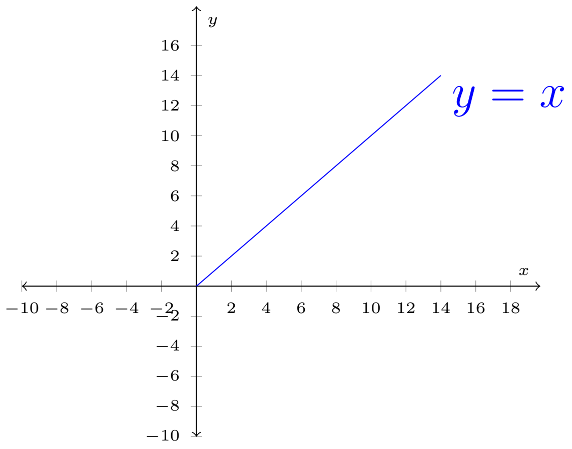


Figure 11:Linear activation equation

Then we should consider about when activation function is not a linear function , for example if it is a sigmoid function as shown in figure 12.

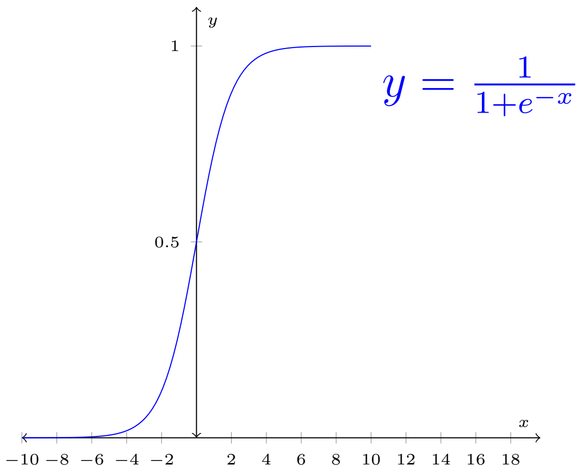


Figure 12: Sigmoid function

So in this case, the **Delta rule** is not useful anymore it should be transformed to another form called **generalized Delta rule** which can be expressed as equations:

$$w_{ij} \leftarrow w_{ij} + \alpha \delta e_i x_j \tag{12}$$

In this experiment, a SGD method was used for supervised learning.And the number of training and mean errors were recorded to plot the graph as shown in figure 13:

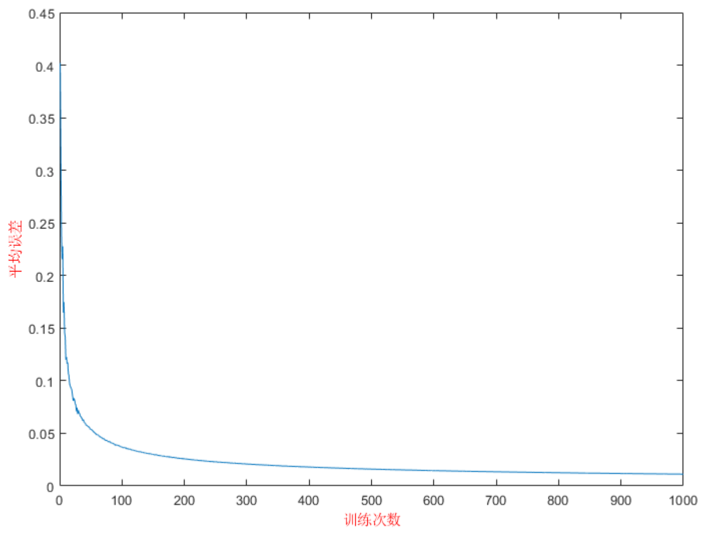


Figure 11: Mean Error against Training number

4 Logistic Regression

Logistic regression is basically like the linear regression, but a bit different. As shown in figure 14, we have a lot of scatter points but the corresponding y value is either 1 or 0 which represent some event that will happen or not in the real world.

And as we did in the linear regression, we try to add a fitting line as shown in figure 15:

It seems like that the fitting line is not relevant to all of these scatter points.But it actually looks like sigmoid function as shown below:

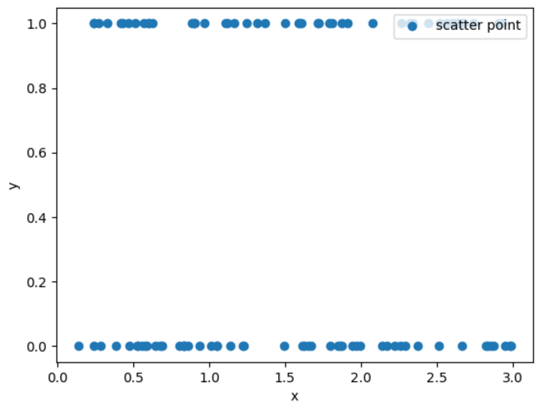


Figure 12: data set

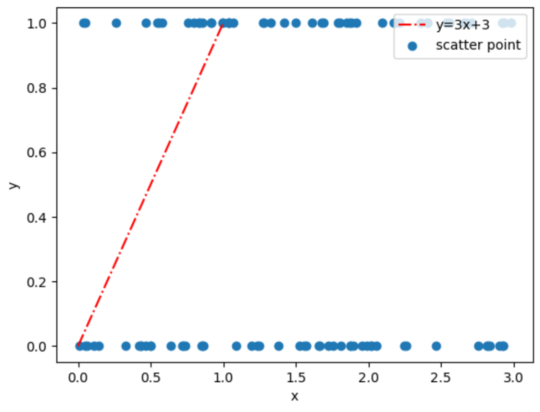
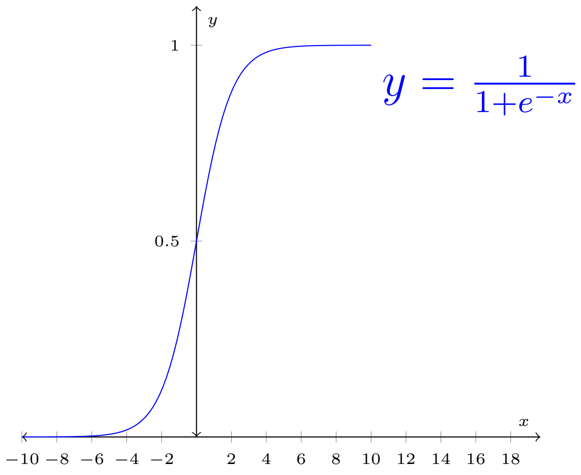


Figure 13: add a fitting line



Sigmoid function

The basic form of multiple linear regression is like this: $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \cdots + \beta_p x_p$ which can be simplified into $\mathbf{Y} = \mathbf{X}\beta$ where,

$$\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ y_n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & x_{13} \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & x_{23} \cdots & x_{2p} \\ 1 & x_{31} & x_{32} & x_{33} \cdots & x_{3p} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & x_{n3} \cdots & x_{np} \end{bmatrix} \tag{13}$$

Substitute Y into the sigmoid function which can be expressed as $y = \frac{1}{1+exp(-x\beta)}$,the probability of positive is $P(Y = 1)$ then the form of logistic regression comes up:

$$P(Y=1)=\frac{1}{1+exp(-X\beta)}. \tag{14}$$

Now,before we figure out the value of β, we need to know a concept: **Likelihood function**

4.1 Likelihood function

Definition 4.1 (Likelihood function) *In statistics, the likelihood function (often simply called the likelihood) measures the goodness of fit of a statistical model to a sample of data for given values of the unknown parameters. It is formed from the joint probability distribution of the sample, but viewed and used as a function of the parameters only, thus treating the random variables as fixed at the observed values.*

We usually use possibility to describe how possible an event that can happen, but we use likelihood function to describe with which parameter,the event is most likely to happen.which can be expressed in mathematical word:

- which a parameter β the possibility of an event which will happen is $P(x|\beta)$
- If we already an event will happen, the likelihood function of the pa-

rameter is $L(\beta|x)$;

- $L(\beta|x) = P(x|\beta)$

One parameter β is related to one value of likelihood function,when the value of β changes ,$L(\beta|x)$will also change, when the value of likelihood function reaches the peak this at this value of parameter it is most reasonable.

4.2 Loss function of Logistic Regression

Before we talk about the logistic function we need to know the most likelihood estimation.In **binary classification**, when y is either 1 or 0, the combination can be used to show the possibility that y can happen $P(y) = P(y=1)^y P(y=0)^{1-y}$

Definition 4.2 (Binary Classification) *Binary classification is the task of classifying the elements of a set into two groups on the basis of a classification rule.*

Technically, the equation should be changed into $P(y|x,\beta) = P(y = 1|\beta, x)^y [1 - P(y = 1|\beta, x)^{1-y}]$, then substitute $\frac{1}{1+exp(-x\beta)}$, then $p(y|x,\beta) = (\frac{1}{1+exp(-x\beta)})^y (1 - \frac{1}{1+exp(-x\beta)})^{1-y}$. we want to maximize the possibility of our samples, so we need to find the maximum value of the following equations:

$$L(\beta|x) = \prod_{i=1}^{n} P(y_i|x_i, \beta) = \prod_{i=1}^{n} p(y|x, \beta) = (\frac{1}{1 + exp(-x_i\beta)})^{y_i}(1 - \frac{1}{1 + exp(-x_i\beta)})^{1-y_i} \tag{15}$$

It is hard to find the maximum vale of above equations, so we can use $log(L(\beta|x)) = \sum_{i=1}^{n}([y_i log(\frac{1}{1+exp(-x_i\beta)})] + [(1 - y_i)log(1 - \frac{1}{1+exp(-x_i\beta)})])$

In the **Linear Regression** section, we use $\sum_{1}^{n}(y_i - \hat{y_i})^2 = \sum_{1}^{n}(y_i - x_i\beta)^2$,if logistic function use the same loss function it will be

$$Q = \sum_{1}^{n}(y_i - \frac{1}{1 + exp(-x_i\beta)})^2 \tag{16}$$

unfortunately, it is not a convex function,it will have a lot of optimal local values,so the logistic function uses **log loss function**.

For **log loss**, it will be

$$J(\beta) = -logL(\beta) = -\sum_{i=1}^{n}[y_i logP(y_i) + (1 - y_i)log(1 - P(y_i))] \tag{17}$$

So for this equation, if i get x and corresponding y=0.6, but the real y is 1 so the loss will be log(0.6) and if the corresponding y=0.3, and the real y is 0 so the loss will be log(0.3).

We know that the derivative of sigmoid function is

$$f'(x) = f(x)(1 - f(x)) \tag{18}$$

substitute $f(x_i\beta) = \frac{1}{1+exp(-x_i\beta)}$ into $J(\beta)$

$$J(\beta) = -\sum_{i=1}^{n}[y_i log(f(x_i\beta))) + (1 - y_i)log(1 - f(x_i\beta))] \tag{19}$$

We know

$$\frac{\partial f(x_i\beta)}{\partial \beta_j} = f(x_i\beta) * (1 - f(x_i\beta)) * x_{ij} \tag{20}$$

Using chain rule,the derivative of loss function can be simplified into:

$$\frac{\partial J(\beta)}{\partial \beta_j} = \sum_{i=1}^{n}(\frac{1}{exp(-x_i\beta)} - y_i)x_{ij} \tag{21}$$

Now we can use gradient descent as we talked before.Here is the following order:

- Initialize β vector, which is known as θ_0, then substitute it into current gradient
- Use learning rate(step) α multiply the current gradient, and get the

current step size G.

- update θ_0, the equation will be $\theta 1 = \theta_0 - \alpha G$
- repeat the procedures above until the step size is very small(you can set the condition)