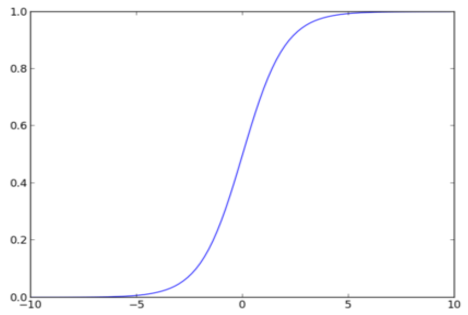
LESSON 14 LOGISTIC REGRESSION

Updated May 10, 2021

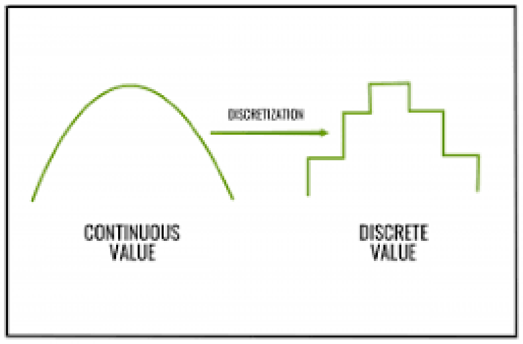
Conventions used in these lessons:

> **bold** - headings, keywords, code
> *italics* - code syntax
> underline - important words

Logistic Regression is a classification algorithm. A classification is a technique for determining which class the dependent variable y belongs to based on one or more independent variables x. A classification problem occurs when the output variable is a “yes” or “no” category like “disease” and “no disease”.
Logistic regression transforms its output using the logistic **sigmoid** function to return a probability value which can then be mapped to two or more discrete classes.

Sigmoid Function $sig = \frac{1}{1 + e^{-y}}$	

Logistic regression is different from linear regression. Linear regression is **continuous** where as logistic regression is **discrete values**.



Example **Logistic Regression** could help us predict the student's test score on a scale of 0 to 100. The scales could be fail (0-50), pass(50-70), good (80-90) and excellent (90-100). The **linear regression** predictions are continuous numbers in the range 0 to 100. Can be any value between 0 an 100.

Logistic Regression could help use predict whether the student passed or failed. Logistic regression predictions are only discrete specific values or categories like passed or fail, yes or so, sick or not sick.

Logistic regression is common and is a useful regression method for solving the binary classification problem. Logistic Regression can be used for various classification problems such as spam detection, diabetes prediction, or if a given customer will purchase a particular product or will they turn to another competitor. Logistic regression describes and estimates the relationship between one dependent binary variable and independent variables.
It is a special case of linear regression where the target variable is categorical in nature. It uses a log of odds as the dependent variable. Logistic Regression predicts the probability of occurrence of a binary event utilizing a logic function.

Linear Regression Equation:
Y = B0 + B1X1 + B2X2 + B3X3 + ...+ Bn Xn
Where, y is dependent variable and x1, x2 ... and Xn are explanatory variables.

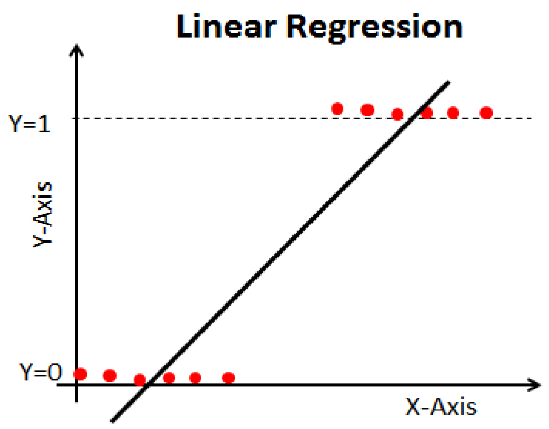
The Logistic Regression equation applies the sigmoid function on linear regression:

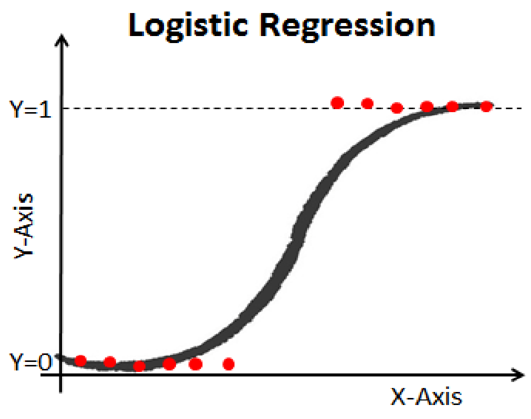
Sigmoid Function:

$$\text{sig} = \frac{1}{1 + e^{-y}}$$

$$\text{sig} = \frac{1}{1 + e^{-(1 + B0 + BiX1 + B2X2 + B3X3 + ...+ Bn\ Xn)}}$$

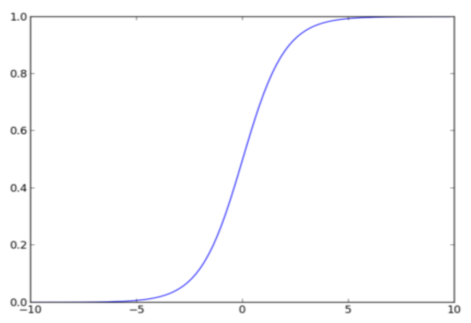
Linear regression is estimated using Ordinary Least Squares (OLS) while logistic regression is estimated using Maximum Likelihood Estimation (MLE) approach.





Ordinary Least squares (OLS) estimates are computed by fitting a regression line on given data points that has the minimum sum of the squared deviations (least square error). Both are used to estimate the parameters of a linear regression model. Maximum Likelihood Estimation (MLE) assumes a joint probability mass function, while OLS doesn't require any stochastic assumptions for minimizing distance.

The sigmoid function, also called logistic function gives an 'S' shaped curve that can take any real-valued number and map it into a value between 0 and 1. If the curve goes to positive infinity, y predicted will become 1, and if the curve goes to negative infinity, y predicted will become 0.

Sigmoid Function $$\text{sig} = \frac{1}{1 + e^{-y}}$$	

Logistic Regression model:

Output = 0 or 1
Hypothesis => Z = WX + B
ho(x) = sigmoid (Z)
if z = +inf then ho(x) = 1
if x = -inf then ho(x) = 0

Analysis of the hypothesis

The output from the hypothesis is the estimated probability. This is used to infer how confident the predicted value to the actual value when given an input X.

Consider the below example,

X = [x0 x1] = [1 IP-Address]

Based on the x1 value, let's say we obtained the estimated probability to be 0.8. This tells that there is 80% chance that an email will be spam.

Mathematically this can be written as,

$H_O(X) = P(Y=1|X;theta)$

Probability that y=1 given X which is parameterized by 'theta'

$P(Y=1|X;theta) + P(Y=0 \mid X;theta) = 1$
$P(Y=0|X;theta) - P(Y=0 \mid X;theta) = 1$

This justifies the name 'logistic regression'. Data is fit into linear regression model, which then be acted upon by a logistic function predicting the target categorical dependent variable.

Types of Logistic Regression:

1. Binary Logistic Regression

The categorical response has only two 2 possible outcomes. Example: Spam or NotSpam , Pass/Fail)

2. Multinomial Logistic Regression

Three or more categories without ordering. Example: Predicting which food is preferred more (Veg, Non-Veg, Vegan) or (Cats, Dogs, Sheep)

3. Ordinal Logistic Regression

Three or more categories with ordering. Example: Movie rating from 1 to 5

(Low, Medium, High)

Decision Boundary

To predict which class a data belongs, a threshold can be set. Based upon this threshold, the obtained estimated probability is classified into classes.

If the predicted value ≥ 0.5, then classify email as spam else as not spam.

Decision boundary can be linear or non-linear. Polynomial order can be increased to get complex decision boundary.

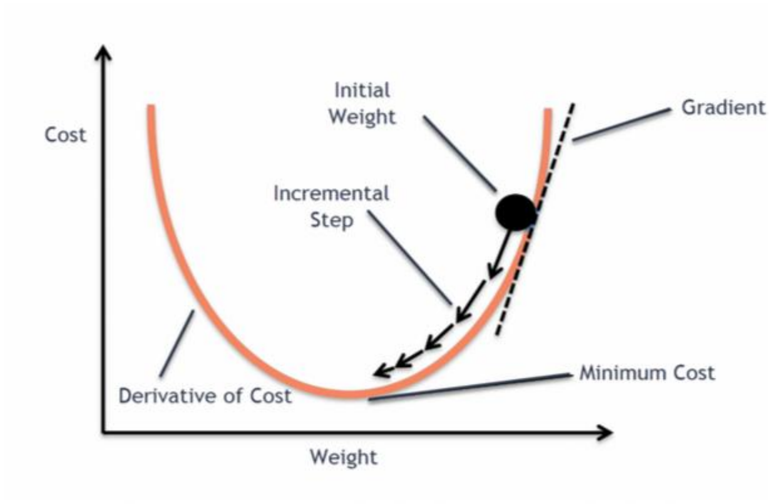
Cost Function

The cost function is used to measure the error. Error is calculated as predicted value – actual value. The goal is to have minimize error.

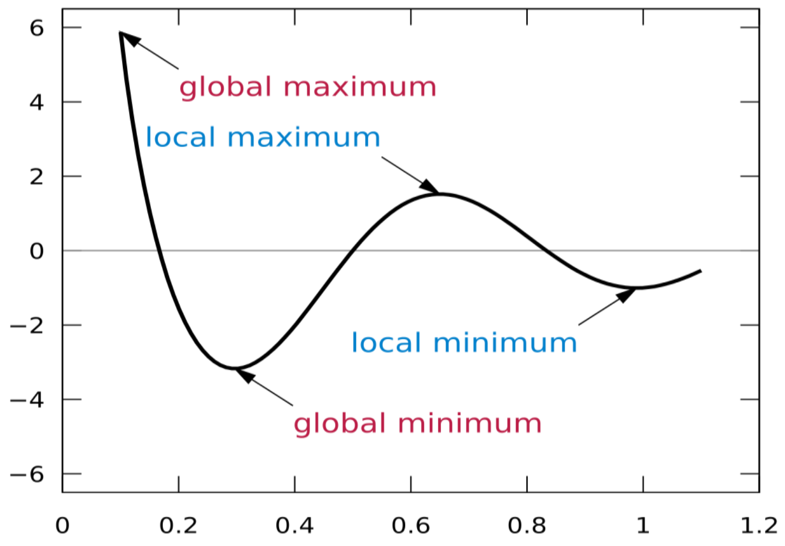
error = predicted value – actual value

Gradient Descent

A Gradient is the direction of a slope going up or down. Gradient Descent is used to minimize the error. Gradient Descent is an iterative process that finds the minima of a function using the derivative of the cost. A derivative is the rate of change for some small time unit, T Gradient Descent is an optimization algorithm using incremental steps that finds the parameters or coefficients of a function where the function has a minimum cost value.

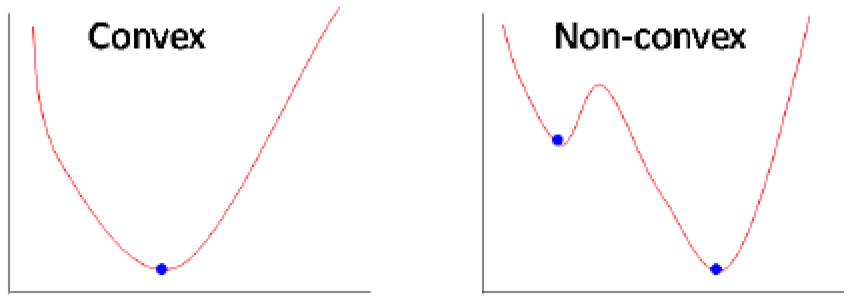


Although this function does not always guarantee to find a global minimum and can get stuck at a local minimum.



The global minimum is the least value of a function while a local minimum is the least value of a function in a certain neighborhood.

Gradient descent will converge into global minimum only if the function is convex.



A non-convex function can result in false minimum where as a convex function will find the true minimum.

Calculating Gradient Descent Example

We calculate gradient descent from the price of houses and the square area of the house indicating the size of the house.

Square Area (1000)	Price (100,000)
.5	1.4
2.3	1.9
2.9	3.2

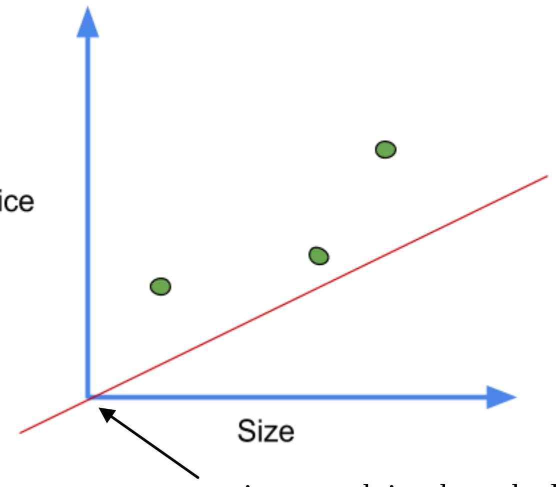
Every machine learning algorithm has an optimization algorithm at its core that wants to minimize its cost function.

Logistic regression is used to optimize linear regression. When we fit a line with a Linear Regression, we optimize the intercept and the slope. When we use Logistic Regression for classification, we optimize a curve or squiggle.

To fit the best fit line we have to optimize the slope of the line and the y-intercept of the line. The equation of a line is y = m x + b. Where m is the slope and b is the y intercept. The y intercept is the value of y when x = 0, where as the x intercept is the value of x when y is zero.

The slope m of a line is rise/run

$$m = \frac{\text{rise}}{\text{run}} = \frac{y2 - y1}{x2 - x1}$$



y = mx + b

y-intercept b is where the line crosses the y axis at x = 0

We will first use gradient descent optimize the y-intercept b. After this we will optimize the y-intercept and slope simultaneously. We will use a approximate slope m of .64.

Calculating Minimum of Residual Errors

First, we calculate the **residual errors** for each for slope and intercept. A residual error is the error between the predicted value and the actual value.

```
Residual error = predicted value – actual value
```

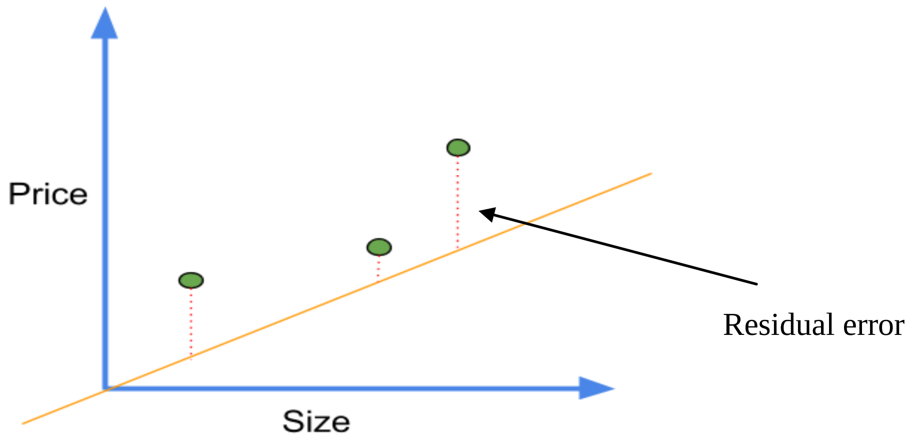
Where the predicted value is:

```
Predicted_value = yintercept + slope * x
```

The gradient descent is usually provided with a random guess for the value of the yintercept. We take a random guess of zero, so the equation becomes

```
Predicted_value = slope * x   (when yintercept = 0)
```

In our graph below the line is the predicted value, the dots are the actual value and Residual errors are red dotted lines connecting the predicted value to the actual value. The goal of gradient descent is to optimize residual errors to a minimum.



Square Area (1000)	Price (100,000)
.5	1.4
2.3	1.9
2.9	3.2

Next, we calculate the squared residual error for each point.

```
    Predicted_value = yintercept + slope * x

    Squared Residual error = (actual_value – predicted_value)^2
```

The squaring is necessary to remove any negative signs. It also gives more weight to larger differences. We are predicting house prices from area.

```
Calculating the first point:
squared residual error = (area-(0 + slope * .price))^2
                       = (1.4-(0 + .64 * .5))^2
                       = (1.4-0.32)^2 = (1.16)^2
```

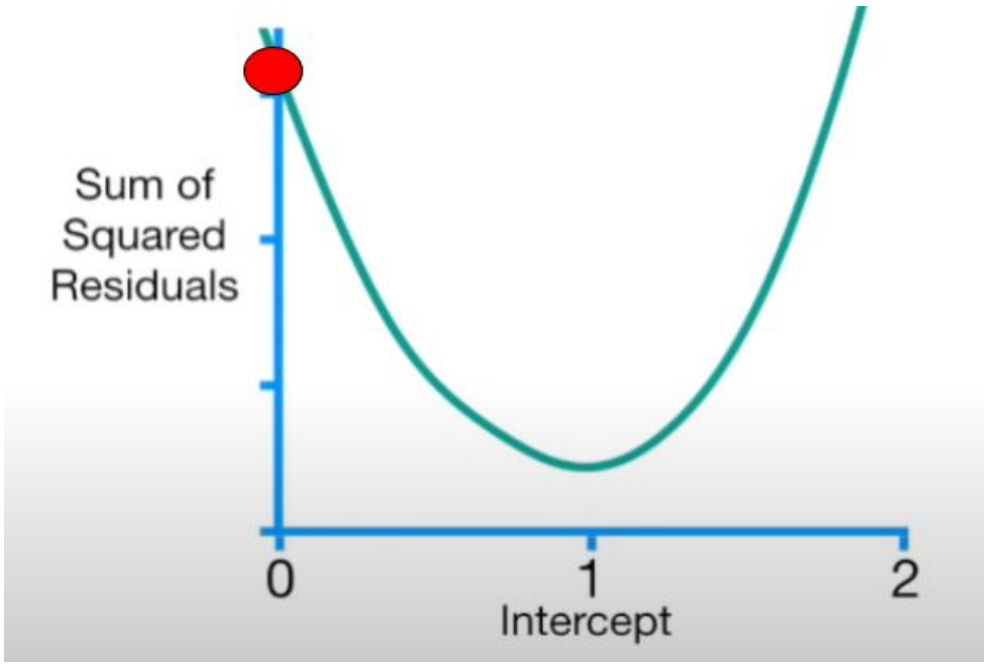
x (price)	**y (area)**	**Intercept (b)**	**Slope (m)**	**Predicted** intercept + slope * x	**(actual-predicted)^2**	**squared residual error**
.5	1.4	0	.64	0 + .64 * .5 = .32	(1.4 - .32)^2	(1.08) ^ 2 = 1.16
2.3	1.9	0	.64	0 + .64 * 2.3 = 1.472	(1.9 - 1.472)^2	(.428)^2 = .183
2.9	3.2	0	.64	0 + .64 * 2.9 = 1.856	(3.2-1.856)^2	(1.34)^2 = 1.79

```
The sum of squared error = 1.16 + .183 + 1.79 = 3.13
```

Cost Function

We continue and calculate all the squared residual error for all the intercept points 0 to 2

We then plot all the sum of squared error points for all intercepts 0 to 2 . The plot represents the cost function.



Our goal is to find the value of the y-intercept b, where the cost curve is at the minimum point.

Logistic Regression homework Question 1

Using the house data of area and prices from the above example, calculate all the **squared error points** for all y-intercepts 0 to 2 using a DataFrame. The Data Frame should have columns area (x) and price (y) . You can also have columns **predicted** and **squared_residual_error** to store calculations. Use the following equations to calculate predicted_value and squared Residual error:

```
Predicted_value = yintercept + slope * x

Squared Residual error = (actual error - predicted)^2
```

You should have a list to store the squared error sums and the y-intercept values 0 to 2. Use a step size of 0.1. Plot a scatter plot the regression line with m = .64 and b = 0. In another plot , plot the squared error points for each intercept 0 to 2.

Step 1:

Make a data frame using a dictionary of area and prices:

Area	Price
.5	1.4
2.3	1.9
2.9	3.2

Step 2:

Make a list of sums and intercepts

```
sums = []
intercepts = []
m = .64
```

in a loop b from 0 to 2 step by .1 calculate: (use np.arrange for the loop)

```
    predicted = m * area + b
    squared_residual_error = (price - predicted) ** 2
    sum_errors =  sum(squared_residual_error)

    sums += sum_errors                    note: += means append
    intercepts += b
```

Step 3:

plot a scatter plot of area vs price where x = area and y = price.

Use m = .64 and the value of b where the sums of errors are minimum plot a regression line of x and y where x = area and y = price.

Hint: use numpy **argmin** function to find **b** where the **sums** are minimum.
All plots should have x and y labeled axis and a title

step 4:

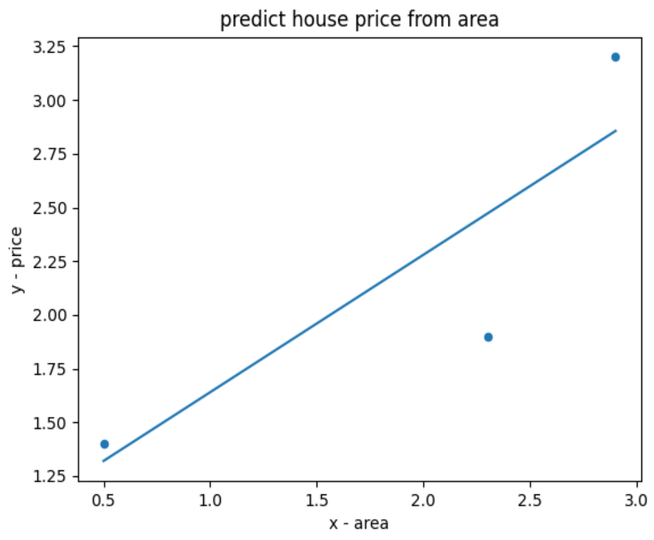
plot intercepts,sums as a line plot as the cost function curve

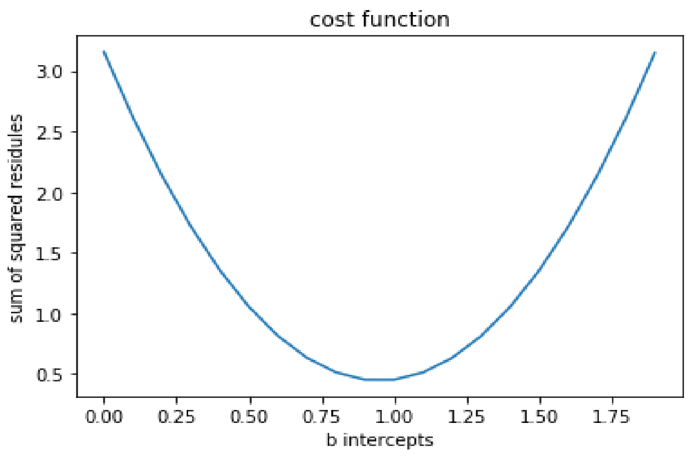
The lowest point on the curve is the optimized y-intercept b having the lowest (minimum) sum of squared residual errors

all plots should have x and y labeled axis and a title

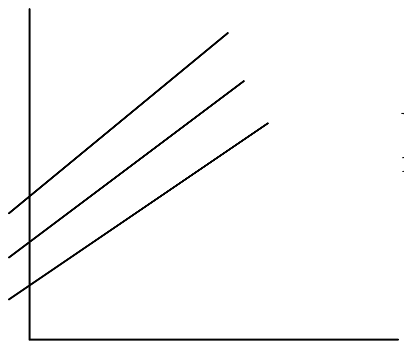
Put everything in a file called logic_regression_homework_123.py.

You should get something like this.





Using Gradient Descent to Optimize Intercept


Vary y-intercept till minimum cost
found but do not change slope

The primary task of Gradient Descent is to find the minimum of this cost function. To find the minimum point, we find its derivatives with respect to the y-intercept. So the equation of this cost function is given by

```
f(intercept) = (price1 - predicted_price1)^2 +
               (price2 - predicted_price2)^2 +
               (price3 – predicted_price3)^2

f(intercept) = (price1 – (intercept + slope * area1))^2 +
               (price2 - (intercept + slope * area2))^)^2 +
               (price3 – (intercept + slope * area3))^)^2

f(intercept) = (price1-(intercept + 0.64 * area1))^2 +
               (price2-(intercept + 0.64 * area2))^2 +
               (price3-(intercept + 0.64 * area3))^2

f(intercept) = (1.4-(intercept+ 0.64 * 0.5))^2 +
               (1.9-(intercept+0.64 * 2.3))^2 +
               (3.2-(intercept+0.64 * 2.9))^2
```

The derivative of this function with respect to intercept is given by

```
derivative = d/d(intercept)(1.4-(intercept+ 0.64 * 0.5))^2
           + d/d(intercept) (1.9-(intercept+0.64 * 2.3))^2
           + d/d(intercept)(3.2-(intercept+0.64 * 2.9))^2
```

We find derivative of each term individually and add them up. Note that the slope is taken constant so its derivative is zero.

derivative of (1.4-(intercept+0.64 * 0.5))^2 = -2 * (1.4-(intercept+0.64 * 0.5))

In a similar way we find derivatives of next two terms and the value we get is:

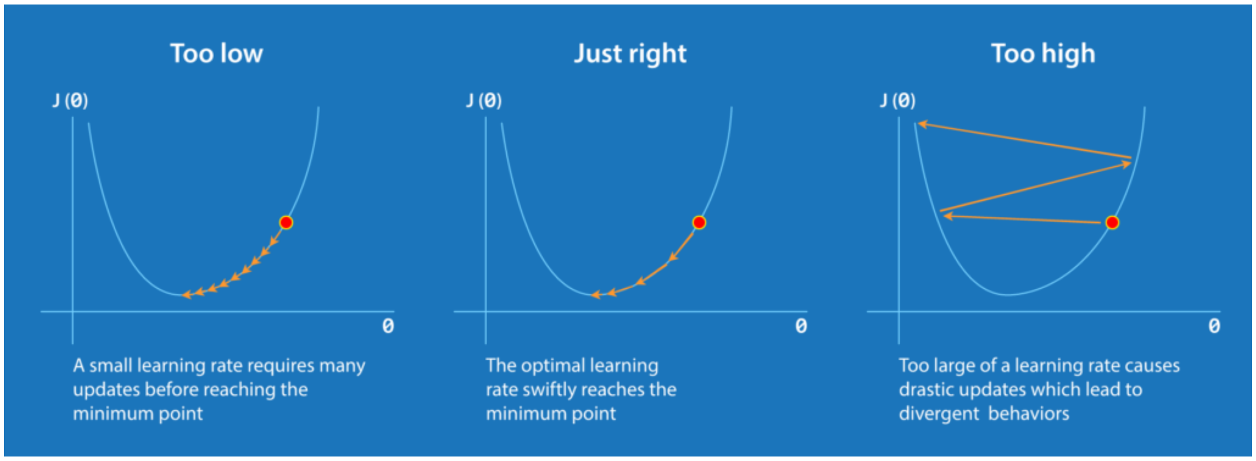
```
derivative = -2 * (1.4-(intercept+0.64 * 0.5))+
              -2 * (1.9-(intercept+0.64 * 2.3))+
             -2 * (3.2-(intercept+0.64 * 2.9))
```

If you know calculus: d/dx(x^2)   = 2x

Let us put the value of intercept=0 to find the value of the next intercept

```
derivative =  -2 * (1.4-(0+0.64 * 0.5))+
               -2 * (1.9-(0+0.64 * 2.3))+
              -2 * (3.2-(0+0.64 * 2.9))
           =  -5.7
```

Gradient descent subtracts the step size from the current value of intercept to get the new value of intercept. This step size is calculated by multiplying the derivative which is -5.7 to a small number called the learning rate. Usually, we take the value of the learning rate to be 0.1, 0.01 or 0.001. The value of the step should not be too big as it can skip the minimum point and thus the optimization can fail. It is a **hyper-parameter** and you need to experiment with its values. A **hyperparameter** is a parameter whose value is used to control the learning process.


Too low
J (θ)
θ
A small learning rate requires many updates before reaching the minimum point
Just right
J (θ)
θ
The optimal learning rate swiftly reaches the minimum point
Too high
J (θ)
θ
Too large of a learning rate causes drastic updates which lead to divergent behaviors

We are using a learning rate of 0.1, then the step size is equal to

Step_size is: intercept * learning rate;
Step_size = -5.7 * .1 = -.57

Our old intercept was 0

Our new intercept is = old_intercept – step_size
New_intercept = 0 – (-.57) = .57

We now plug the new intercept into our equation

d (sum of squared error)/d(intercept) = -2 * (1.4-(0.57+0.64 * 0.5))+
-2 * (1.9-(0.57+0.64 * 2.3))+
-2 * (3.2-(0.57+0.64 * 2.9))
= -2.3

Now we calculate the next step size:

Step size =-2.3*0.1 = -0.23

New intercept = old intercept-step size
= 0.57 - (-0.23) = 0.8

Again let us now put the new intercept in the derivative function

d (sum of squared error) /d(intercept) = -2 *(1.4-(0.8+0.64 * 0.5)) +
-2 *(1.9-(0.8+0.64 * 2.3)) +
-2 *(3.2-(0.8+0.64 * 2.9))
= -0.9

Step size = -0.9*0.1 = -0.09

New intercept = old intercept-step size
= 0.8 - (-0.09) = 0.89

You might have noticed that the value of the step is high when the optimal solution is far away and this value is less as we approached an optimal solution. Thus we can say that gradient descent takes a bigger step when away from the solution and takes small steps when nearer to an optimal solution. This is the reason why gradient descent is efficient and fast.

LOGISTIC REGRESSION HOMEWORK QUESTION 2

Continue calculating the sum of square derivative and new intercepts. Use the data frame from previous homework. You will need some new calculation columns. Use a learning rate of .1. Loop till the absolute sum of differences is less than .001. Plot sum of difference vs intercepts and costs function vs intercepts, provide a legend of labels. Finally make a scatter plot and regression line from the final calculated value of b. Print out m and b and number of iterations. Try out different accuracy values.

step 1:

Make a data frame using a dictionary of area and prices:

Area	Price
.5	1.4
2.3	1.9
2.9	3.2

step 2:

Make lists to store derivatives, intercepts and costs

```
derivatives = []
intercepts = []
costs = []
b = 0
m = .64
learning_rate = .1
sum_derivatives = 100
iterations = 0
```

step 3:

loop while abs(sum_derivatives) > .001

calculate:

```
# calculate intercept
intercept = (price  - (b + m * area))**2
                + (price - (b + m * area))**2
                + (price - (b + m * area))**2
sum_cost =  sum(intercept)
 costs += sum_cost                                              note: += means append

# calculate derivative
 derivative = -2 *(price - (b + m * area))
 sum_derivatives = sum(derivative)
 derivatives += sum_derivatives                                 note: += means append
 intercepts +=b
 b = b - sum_derivatives * learning_rate;
```

Repeat step 3 till accuracy found

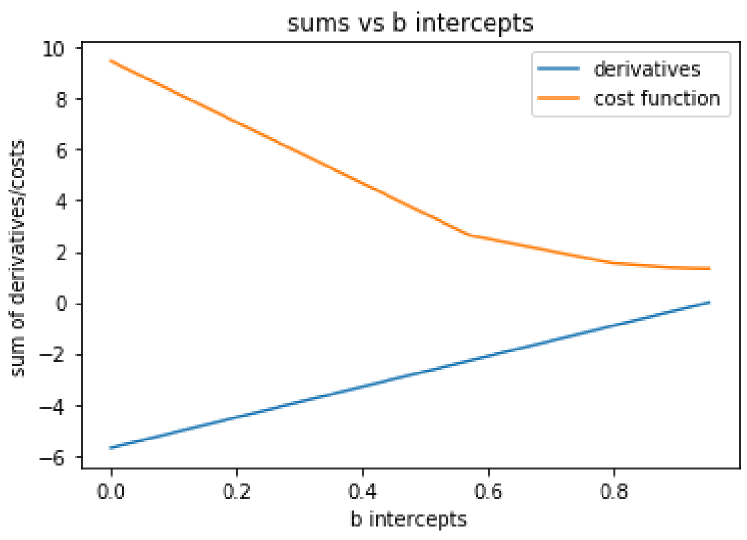
step 4:

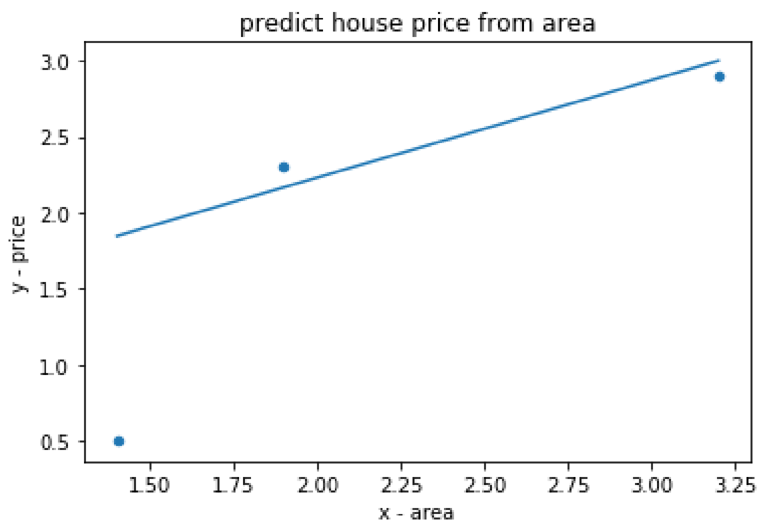
plot: intercepts,derivatives as derivatives
plot: intercepts,costs   as cost function
plot: scatter plot of area and price
plot: regression line using new values of b where m = .64

step 5:

Print out m and b and number of iterations.

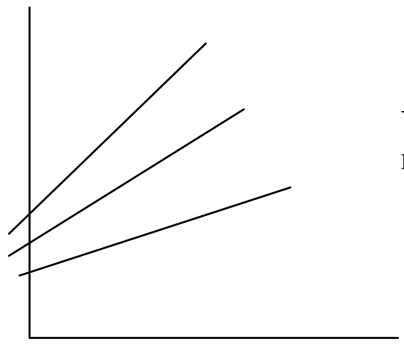
You should get something like this:





```
m =  0.64 b =  0.9506267928166401 iterations:  11
```

Using Gradient Descent to Optimize Slope and Intercept



Vary slope and y-intercept still
minimum cost found

As before we take the derivatives but this time of this equation:

```
f(intercept) = (price1-(intercept + slope * area1))^2 +
               (price2-(intercept + slope * area2))^2 +
               (price3-(intercept + slope * area3))^2

f(intercept) =  (1.4-(intercept + slope * 0.5))^2+
                (1.9-(intercept + slope * 2.3))^2+
                (3.2-(intercept + slope * 2.9))^2
```

We find the derivative with respect to intercept keeping slope as constant:

```
Derivative w.r.t intercept = -2*(1.4-(intercept+slope * 0.5))+
                             -2*(1.9-(intercept+slope * 2.3))+
                            -2*(3.2-(intercept+slope * 2.9))
```

Now we find derivative with respect to slope and consider intercept as constant

```
Derivative w.r.t slope = -2(0.5) (1.4-(intercept+slope * 0.5))+
                         -2(2.3) (1.9-(intercept+slope * 2.3))+
                        -2(2.9)(3.2-(intercept+slope * 2.9)
```

When we have two or more derivatives of the same function, they are called **gradients**. We use these gradients to descend down the cost function. Thus the algorithm is called **gradient descent**.

The cost function we have been using so far is the sum of the square residuals.

As before we initialize intercept and slope randomly as 0 and 1. Now putting these values in the above gradients.

```
Derivative w.r.t intercept = -2*(1.4-(0+1 * 0.5)) +
                               -2*(1.9-(0+1 * 2.3)) +
                              -2*(3.2-(0+1 * 2.9))
                           = -1.6
```

We take a different learning rate here of .01:

```
Step size = -1.6 * 0.01 = -0.016
New intercept = 0-(-0.016) = 0.016

d/d(slope)= -2(0.5) * (1.4-(0+1 * 0.5)) +
             -2(2.3) * (1.9-(0+1 * 2.3)) +
            -2(2.9) * (3.2-(0+1 * 2.9))
             =-0.8
```

```
Step size= -0.8*0.01=-0.008
New slope= 1-(-0.008)=1.008
```

Repeating this process until we get step size near zero for both slope and intercept gives us an optimal solution and best fit line.

LOGISTIC REGRESSION HOMEWORK QUESTION 3

Continue calculating the sum of square derivative and new intercepts and new slopes. Use the data frame from previous homework. You will need some new calculation columns. Use a learning rate of .01. Loop till the absolute sum of differences and slopes is less is less than .001. Plot sum of difference vs intercepts sum of slopes and costs function vs intercepts, provide a legend of labels. Finally make a scatter plot and regression line from the final calculated value of m and b. Try out different accuracy values.

Print out m and b and number of iterations.

Step 1:

Make a data frame using a dictionary of area and prices:

Area	Price
.5	1.4
2.3	1.9
2.9	3.2

Step 2:

Make the following lists:

derivative_intercepts = []
derivative_slopes = []
intercepts = []
slopes = []
costs=[]
b = 0
m = 1

```
learning_rate = .01
sum_derivative_intercepts = 100
sum_derivative_slopes = 100
iterations=0
```

step 3:

```
loop while (abs(sum_derivative_intercepts) > .001 and
abs(sum_derivative_slopes) > .001):
```

step 4:

```
  # calculate y intercept
  intercept = price-(b + m * area))**2
              + (price-(b + m * area))**2
               + (price-(b + m * area))**2
  sum_cost =  sum(intercept)
  costs+=sum_cost                                           note: += means append

   # calculate derivative intercepts
   derivative_intercepts = -2 *(price - (b + m * area))
   sum_derivative_intercepts = sum(derivative_intercepts)
   derivative_intercepts += sum_derivative_intercepts       note: += means append

   # calculate derivative slopes
   derivative_slopes = -2*area *(price - (b + m * area))
   sum_derivative_slopes = sum(derivative_slopes)
   derivative_slopes += sum_derivative_slopes              note: += means append
   # store and calculate new y intercept
   intercepts+=b
   b = b - sum_derivative_intercepts * learning_rate;

   # store and calculate new slope
   slopes+=m
   m = m - sum_derivative_slopes * learning_rate;
```

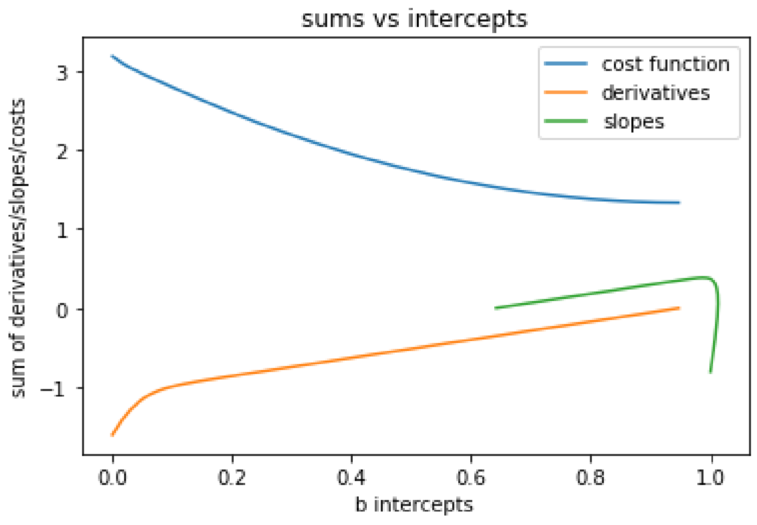
step 5:

plot: intercepts,derivatives as derivatives
plot: intercepts,costs as cost function
plot: slopes,derivative_slopes as slopes
plot: scatter plot of area and price
plot: regression line using new values of b and m

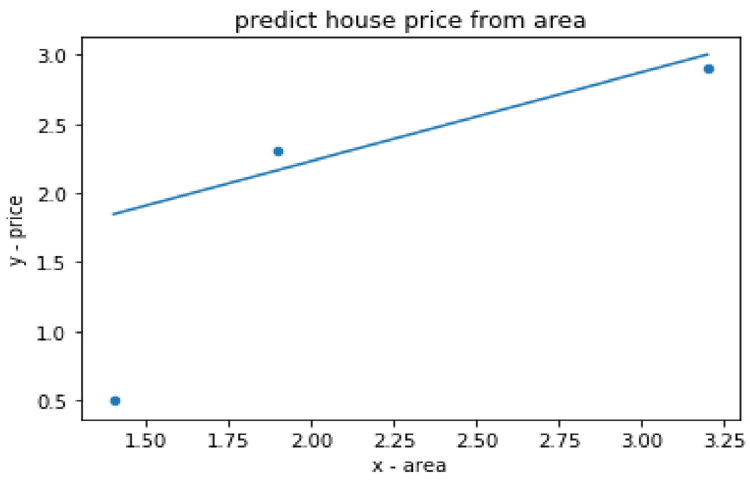
step 6:

Print out m and b and number of iterations.

You should get something like this:

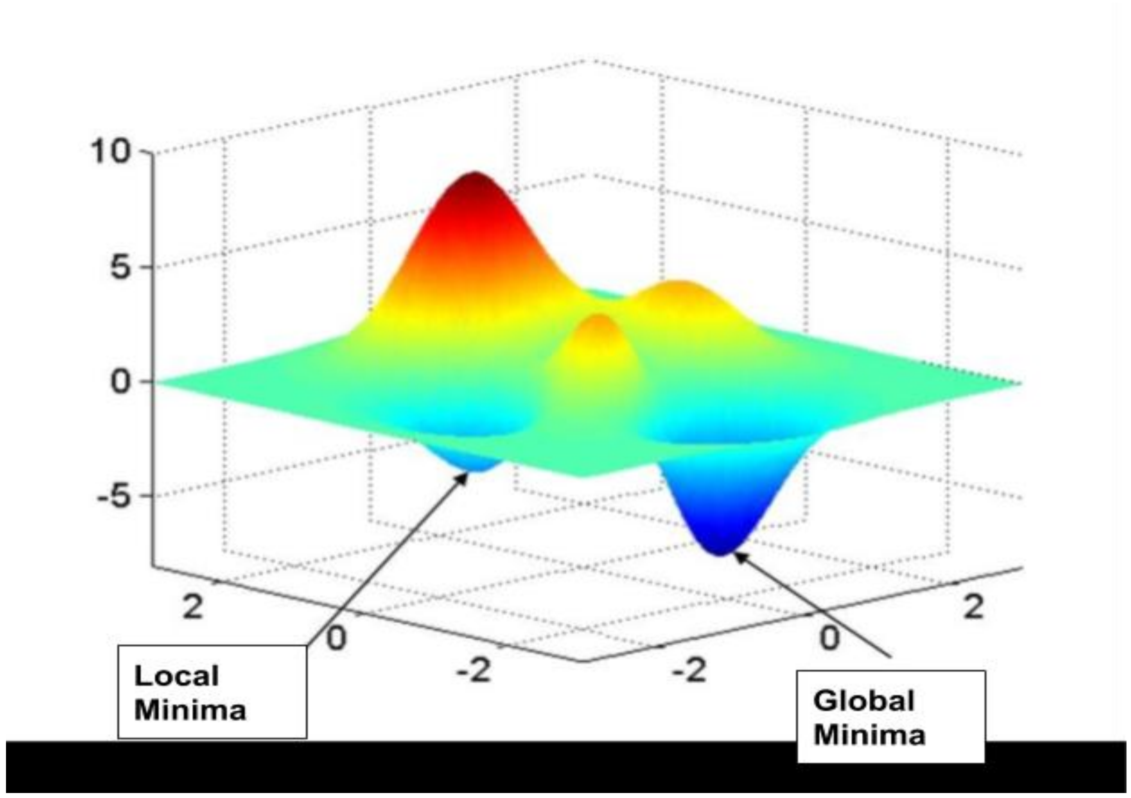


```
m =  0.6418829376708766 b =  0.94670578027403 iterations:  534
```



More Than 1 x Parameter

If we have more than one parameter, such as the number of rooms, the process remains the same but the number of derivatives increases. Also here we used the sum of squared residuals as loss function, but we can use any other loss function as well such as least squares.



To briefly summaries the process, here are some points

1. Take the gradient of the loss function which means take the derivative of the loss function for each parameter in it.
2. Randomly select the initialization values.
3. Substitute these parameter values in the gradient
4. Calculate step size by using appropriate learning rate.
5. Calculate new parameters
6. Repeat from step 3 until an optimal solution is obtained.

Gradient Descent Example

Our cost function is the sum of the residual errors squared:

$$\mathbf{Cost} = \frac{\mathbf{1}}{\mathbf{n}} \sum_{\mathbf{i=1}}^{\mathbf{n}} (\mathbf{Y_i} - \boldsymbol{\theta}_\mathbf{i}\ \mathbf{X}_{\ \mathbf{i}})^{\mathbf{2}}$$

Taking the log of both sides

$$log\ loss = -\frac{1}{N}\sum_{i=1}^{N} yi * log\big(p(yi)\big) + (1 - yi) * log\ (1 - p(yi))$$

Our cost function is now:

$$\mathbf{Cost(h_o(x), Y(actual)) = -y\ log(h_o(x)) - (1\text{-}y)\ log(1\text{-}h_o(x))}$$

If y = 1, (1-y) term will become zero, therefore – log (h_o*x) alone will be present
If y = 0, (y) term will become zero, therefore – log (1-h_o*x) alone will be present

MLE Approach:

Logistic regression uses a method called maximum likelihood estimation (MLE) to find the best fit line. MLE works as follows:

- Pick a probability curve at random using the probability function we created above.
- Calculate the likelihood (probability) of the outcome of a single data point.
- Calculate likelihoods for all data points and multiply likelihoods of all data points.
- Shift the probability curve and repeat the process.
- Select the curve with the highest ("Maximum") likelihoods.

We now Calculate the formula for gradient descent:

$Z = w1x + w2x + b \rightarrow \bar{y} = a = r(z) \rightarrow L(\bar{y},y) \quad a = \bar{y}$

$$\frac{dL}{dw1} = \frac{dL}{da} \cdot \frac{da}{dz} \cdot \frac{dz}{dw1}$$

using the chain rule

Chain rule:

If a variable *z* depends on the variable *y*, which itself depends on the variable *x* (i.e., *y* and *z* are dependent variables), then *z*, via the intermediate variable of *y*, depends on *x* as well. In which case, the chain rule states that:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}.$$

$$\frac{dL}{da} = \frac{d}{da} (-y \log a - (1-y) \log (1-a))$$

$$\frac{dL}{da} = \frac{-y}{a} + \frac{(1-y)}{(1-a)}$$

$$\frac{da}{dz} = a\,(1-a)$$

$$\frac{dz}{dw1} = x1$$

substituting into:

$$\frac{dL}{dw1} = \frac{dL}{da} * \frac{da}{dz} * \frac{dz}{dw1}$$

$$\frac{dL}{dw1} = \frac{-y}{a} + \frac{(1-y)}{1-a} * (a)\,(1-a) * x1 = (a-y) * x1$$

$$\frac{dL}{dw1} = \frac{-y\,(1-a) + a\,(1-y)}{a\ \ (1-a)} * (a)\,(1-a) * x1$$

$$\frac{dL}{Dw1} = -y + ya + a - ay * x1 = (a-y) * x1$$

Derivative of a log function

$$\frac{d}{dz} \log(x) = \frac{1}{x}$$

Sample Test Program

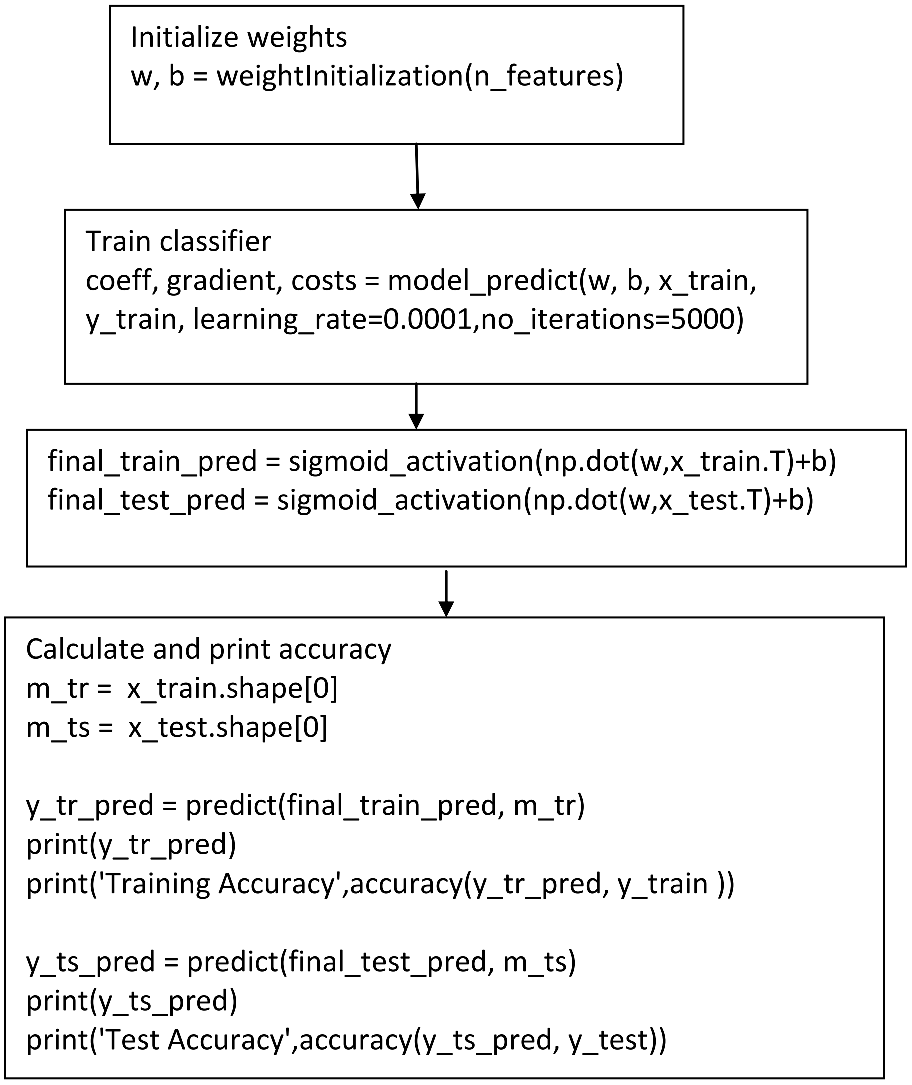
We want a program that will classify number as even or odd. We represent numbers as binary as only digits 0 and 1.

number	Binary	Result
0	000	Even
1	001	Odd
2	010	Even
3	011	Odd
4	100	Even
5	101	Odd
6	110	Even
7	111	Odd

We divide our number into a train set and a test set. The train set is used to train the classifier. The test set is used to test the classifier for accuracy after training has finished. The number of features are the number of columns in the X data.

```
#Split test data into train set and test set
x_train = np.array([[0,0,0],
          [0,0,1],
          [0,1,0],
          [0,1,1],
          [1,0,1],
          [1,1,0]])

y_train = np.array([0,1,0,1,1,0])

x_test =  np.array([[0,0,0],
          [0,0,1],
          [0,1,0],
          [1,0,1],
          [1,1,0],
          [1,1,1]])

y_test = np.array([0,1,0,1,0,1])

n_features = x_train.shape[1]
```

Logistic Regression Program Flow:



```
Initialize weights
w, b = weightInitialization(n_features)
```

```
Train classifier
coeff, gradient, costs = model_predict(w, b, x_train,
y_train, learning_rate=0.0001,no_iterations=5000)
```

```
final_train_pred = sigmoid_activation(np.dot(w,x_train.T)+b)
final_test_pred = sigmoid_activation(np.dot(w,x_test.T)+b)
```

```
Calculate and print accuracy
m_tr =  x_train.shape[0]
m_ts =  x_test.shape[0]

y_tr_pred = predict(final_train_pred, m_tr)
print(y_tr_pred)
print('Training Accuracy',accuracy(y_tr_pred, y_train ))

y_ts_pred = predict(final_test_pred, m_ts)
print(y_ts_pred)
print('Test Accuracy',accuracy(y_ts_pred, y_test))
```

Here is the complete program:

```
"""
logisticregression.py
logistic regression
"""
import numpy as np

def weightInitialization(n_features):
   w = np.zeros((1,n_features))
   b = 0
   return w,b
```

```
def sigmoid_activation(result):
    final_result = 1/(1+np.exp(-result))
    return final_result

def model_optimize(w, b, X, Y):
    m = X.shape[0]

    #Prediction
    final_result = sigmoid_activation(np.dot(w,X.T)+b)
    Y_T = Y.T
    cost = (-1/m)*(np.sum((Y_T*np.log(final_result)) + ((1-Y_T)*(np.log(1-final_result)))))

    #Gradient calculation
    dw = (1/m)*(np.dot(X.T, (final_result-Y.T).T))
    db = (1/m)*(np.sum(final_result-Y.T))

    grads = {"dw": dw, "db": db}
    return grads, cost

def model_predict(w, b, X, Y, learning_rate, no_iterations):
    costs = []
    for i in range(no_iterations):
        # train model
        grads, cost = model_optimize(w,b,X,Y)
        #get gradients
        dw = grads["dw"]
        db = grads["db"]
        #weight update
        w = w - (learning_rate * (dw.T))
        b = b - (learning_rate * db)

        if (i % 100 == 0):
            costs.append(cost)
            #print("Cost after %i iteration is %f" %(i, cost))

    #final parameters
    coeff = {"w": w, "b": b}
    gradient = {"dw": dw, "db": db}

    return coeff, gradient, costs
```

```
# return predictions
def predict(final_pred, m):
   y_pred = np.zeros((1,m))
   for i in range(final_pred.shape[1]):
      if final_pred[0][i] > 0.5:
         y_pred[0][i] = 1
   return y_pred

# return accuracy score
def accuracy_score(predicted_labels, actual_labels):
   diff = predicted_labels - actual_labels
   return 1.0 - (float(np.count_nonzero(diff)) / len(diff))

# train data
x_train = np.array([[0,0,0],
          [0,0,1],
          [0,1,0],
          [0,1,1],
          [1,0,1],
          [1,1,0]])

print("x train")
print(x_train)

# even/odd numbers
y_train = np.array([0,1,0,1,1,0])
print("y train")
print(y_train)

# test data
x_test =  np.array([[0,0,0],
          [0,0,1],
          [0,1,0],
          [1,0,1],
          [1,1,0],
          [1,1,1]])

print("x test")
print(x_test)
```

```
# even/odd numbers
y_test = np.array([0,1,0,1,0,1])

print("y test")
print(y_test)

#Get number of features
n_features = x_train.shape[1]
print('Number of Features', n_features)
w, b = weightInitialization(n_features)

#Gradient Descent
coeff, gradient, costs = model_predict(w, b, x_train, y_train,
learning_rate=0.0001,no_iterations=4500)

#Final prediction
w = coeff["w"]
b = coeff["b"]

print('Optimized weights', w)
print('Optimized intercept',b)

# proabilities
final_train_pred = sigmoid_activation(np.dot(w,x_train.T)+b)
final_test_pred = sigmoid_activation(np.dot(w,x_test.T)+b)

# model train and test
m_tr =  x_train.shape[0]
m_ts =  x_test.shape[0]

# y train predictions
y_tr_pred = predict(final_train_pred, m_tr)
print(y_tr_pred)
print('Training Accuracy',accuracy_score(y_tr_pred, y_train ))

# y test predictions
y_ts_pred = predict(final_test_pred, m_ts)
print(y_ts_pred)
print('Test Accuracy',accuracy_score(y_ts_pred, y_test))
```

Test Program Output:

```
logistic regression
x train
[[0 0 0]
 [0 0 1]
 [0 1 0]
 [0 1 1]
 [1 0 1]
 [1 1 0]]
y train
[0 1 0 1 1 0]
x test
[[0 0 0]
 [0 0 1]
 [0 1 0]
 [1 0 1]
 [1 1 0]
 [1 1 1]]
y test
[0 1 0 1 0 1]
Number of Features 3
Optimized weights [[-0.00079864 -0.04158738  0.12165701]]
Optimized intercept -0.0024190394042546545
[[0. 1. 0. 1. 1. 0.]]
Training Accuracy 1.0
[[0. 1. 0. 1. 0. 1.]]
Test Accuracy 1.0
```

Todo:

Copy and paste the above program into a file called logisticregression.py and run it. Try different training and test values.

Logistic Regression using SkLearn

Sklearn has many pre-built classifiers that make classification quite easy to do.

We first import all the required libraries:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
import pandas as pd
```

We use the previous data:

x is binary numbers 0 to 7

```
x =  np.array([[0,0,0],
             [0,0,1],
            [0,1,0],
            [0,1,1],
            [1,0,0],
            [1,0,1],
            [1,1,0],
            [1,1,1]])
```

y is even/odd result

```
y = np.array([0,1,0,1,0,1,0,1])
```

we use train_test_split function to split the data into a training set and a test set

```
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.4)
```

We make the LogisticRegression model

```
logreg = LogisticRegression()
```

We fit the model with data

```
logreg.fit(x_train,y_train)
```

We get the prediction results:

```
y_pred=logreg.predict(x_test)
print("prediction")
print(y_pred)
```

We make confusion matrix:

```
cnf_matrix = metrics.confusion_matrix(y_test, y_pred)
```

A confusion matrix is a table is used to describe the performance of a classification model (or “classifier”) on a set of test data for which the true values are known. It allows the visualization of the performance of an algorithm.
It allows easy identification of confusion between classes e.g. one class is commonly mislabeled as the other

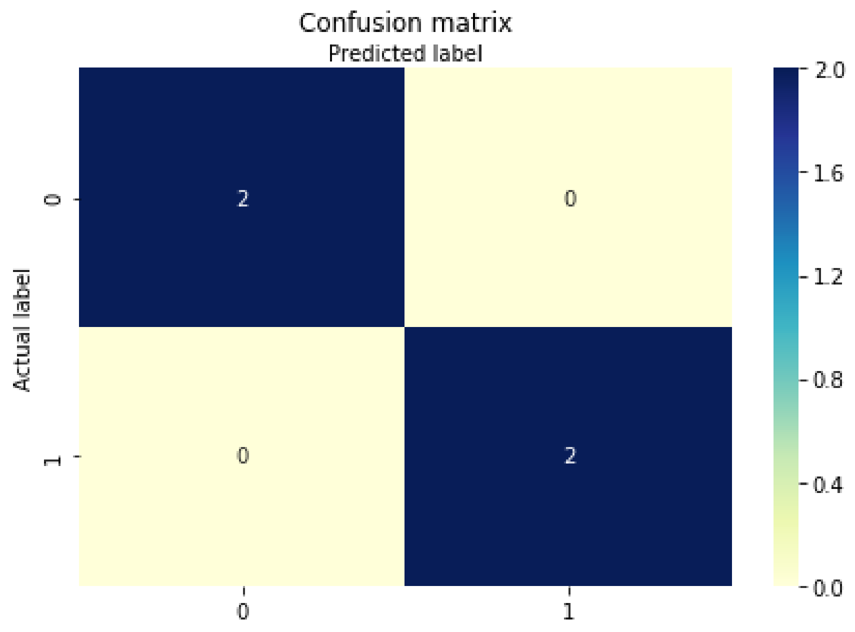
The number of correct and incorrect predictions are summarized with count values and broken down by each class.

	Class 1 Predicted	**Class 2 Predicted**
Class 1 Actual	True Positive	False Negative
Class 2 Actual	False Positive	True Negative

We plot confusion matrix and a heat map together

A heatmap is a way of representing the data in a 2-dimensional form. The data values are represented as colors in the graph. The goal of the heatmap is to provide a colored visual summary of information. We use the Seaborn **heatmap** function to plot our confusion matrix.

```
class_names=[0,1]
fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)

sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
ax.xaxis.set_label_position("top")
plt.tight_layout()
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
plt.show()
```



we print the statistics

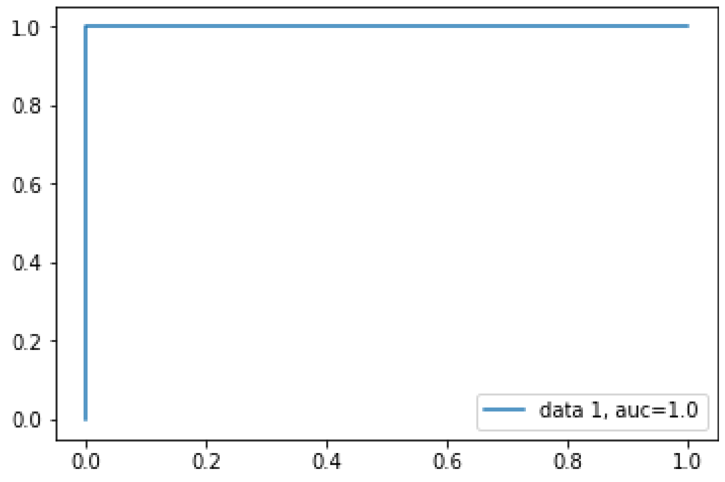
```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
print("Precision:",metrics.precision_score(y_test, y_pred))
print("Recall:",metrics.recall_score(y_test, y_pred))
```

```
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
```

We plot ROC curve to show accuracy:

A Receiver Operator Characteristic (ROC) curve is a graphical plot used to show the diagnostic ability of binary classifiers. The ROC curve shows the trade-off between **sensitivity** True Positive Rate (or TPR) and **specificity** 1-False Positive RAATE (1 – FPR). The true positive rate is the proportion of observations that were correctly predicted to be positive out of all positive observations (TP/(TP + FN)). Similarly, the false positive rate is the proportion of observations that are incorrectly predicted to be positive out of all negative observations (FP/(TN + FP)). For example, in medical testing, the true positive rate is the rate in which people are correctly identified to test positive for the disease in question.
For binary classifiers ROC give a probability or score that reflects the degree to which an instance belongs to one class rather than another. To compare different classifiers, it can be useful to summarize the performance of each classifier into a single measure. The area under the ROC curve, which is abbreviated to AUC. is equivalent to the probability that a randomly chosen positive instance is ranked higher than a randomly chosen negative instance.

```
y_pred_proba = logreg.predict_proba(x_test)[::,1]
fpr, tpr, _ = metrics.roc_curve(y_test,  y_pred_proba)
auc = metrics.roc_auc_score(y_test, y_pred_proba)
plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
plt.legend(loc=4)
plt.title("ROC")
plt.show()
```



We are very accurate.

Here is the complete program:

```
#  Logistic Regression using SkLearn

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
import pandas as pd

# x is binary numbers 0 to 7
x =  np.array([[0,0,0],
              [0,0,1],
             [0,1,0],
             [0,1,1],
             [1,0,0],
             [1,0,1],
             [1,1,0],
             [1,1,1]])

# y is  even/odd  result
y = np.array([0,1,0,1,0,1,0,1])

# split the data into a training set and a test set
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=0.4)

# make logic regression model
logreg = LogisticRegression()

# fit the model with data
logreg.fit(x_train,y_train)

# predict results:
y_pred=logreg.predict(x_test)
print("prediction")
print(y_pred)

# plot confusion matrix:
cnf_matrix = metrics.confusion_matrix(y_test, y_pred)
class_names=[0,1]
fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
```

```
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)

# plot heat map
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
ax.xaxis.set_label_position("top")
plt.tight_layout()
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
plt.show()

# print statistics
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
print("Precision:",metrics.precision_score(y_test, y_pred, zero_division=1))
print("Recall:",metrics.recall_score(y_test, y_pred, zero_division=1))

# plot ROC curve
y_pred_proba = logreg.predict_proba(x_test)[::,1]
fpr, tpr, _ = metrics.roc_curve(y_test,  y_pred_proba)
auc = metrics.roc_auc_score(y_test, y_pred_proba)
plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
plt.legend(loc=4)
plt.show()
```

todo:

Type in or copy/paste in the above program and run it.
Try different values of test size for the train_test_split.

LOGISTIC REGRESSION HOMEWORK   Question 4

For the Logistic Regression sample program above calculate the confusion matrix using sklearn, plot the heatmap using a Data Frame and Seaborn. Calculate Accuracy, Precision and Recall using the sklearn. Finally calculate ROC and AUC using sklearn and plot the ROC curve using matplotlib. Label all plots. Save your homework 4 into a file called logistic_regression_homework4.py
You can use the following to extract the y test probabilities:

```
        y_proab = final_test_pred[0]
```

Here are the steps to follow:

Step 1:

Copy the logistic regression program into file logistic_regression_homework4.py

Step 2: make confusion matrix

```
from sklearn import metrics

#  make confusion matrix
cnf_matrix = metrics.confusion_matrix(y_test, y_ts_pred[0])
```

Step 3: plot confusion matrix using seaborn and a data frame

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# plot confusion using seaborn and a data frame
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
plt.show()
```

Step 4: Calculate Accuracy, Precision and Recall using the sklearn.

```
# Calculate Accuracy, Precision and Recall
print("Accuracy:",metrics.accuracy_score(y_test, y_ts_pred[0]))
print("Precision:",metrics.precision_score(y_test, y_ts_pred[0]))
print("Recall:",metrics.recall_score(y_test, y_ts_pred[0]))
```

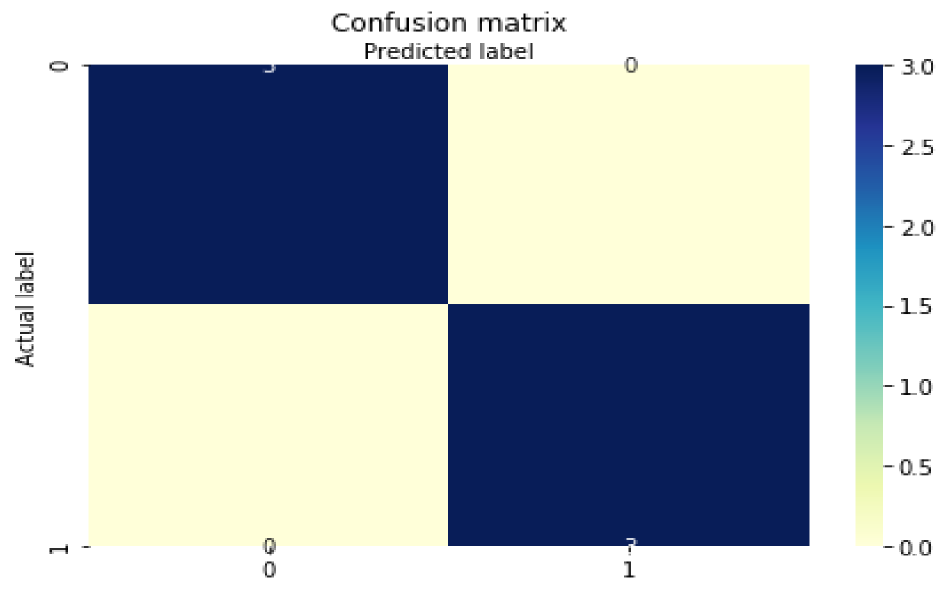
Step 5: calculate ROC and AUC using sklearn

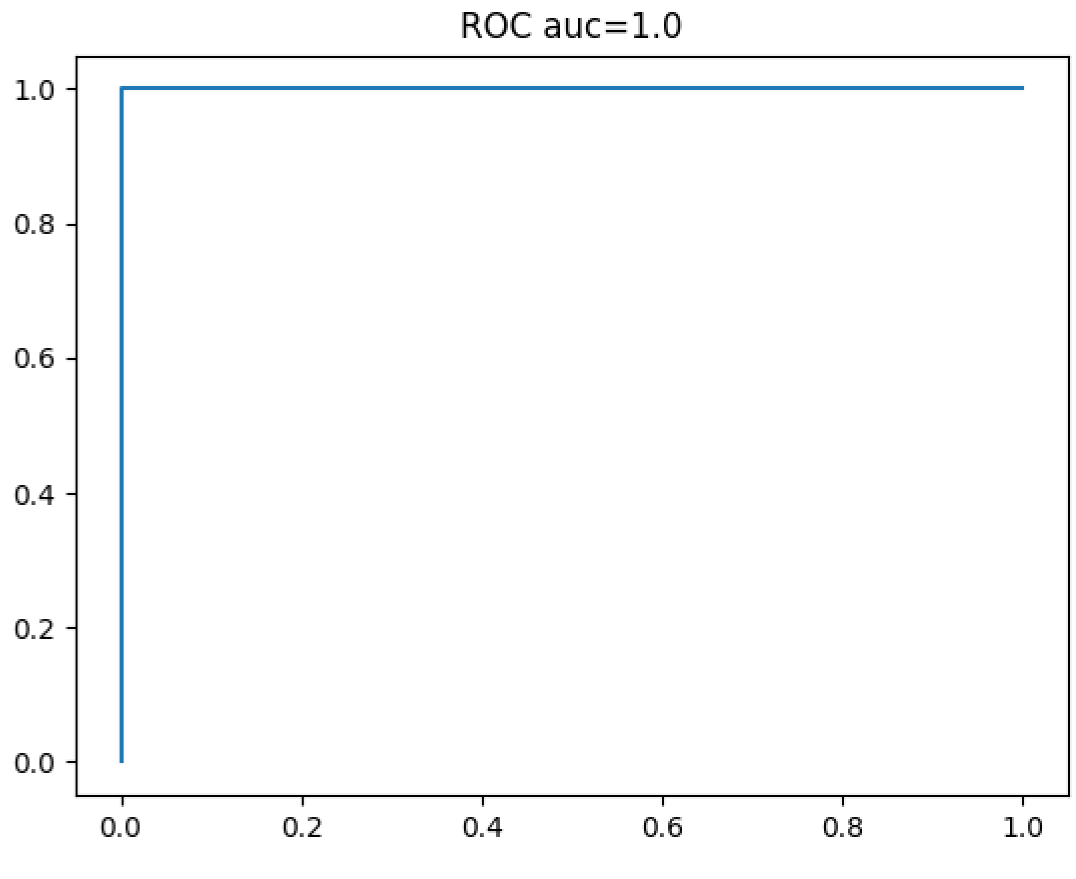
```
# calculate  ROC and AUC using sklearn
y_proab = final_test_pred[0]
fpr, tpr, _ = metrics.roc_curve(y_test,  y_proab)
auc = metrics.roc_auc_score(y_test, y_proab)
```

Step 6: plot the ROC curve using matplotlib

```
plt.plot(fpr,tpr)
plt.title("ROC" + " auc="+str(auc))
plt.show()
```

You should get something like this:





Multinomial Logistic Regression

In Multinomial Logistic Regression, the output variable can have **more than two possible discrete outputs**. Our binary digits now have outputs 0 to 7. Multinomial Logistic Regression uses the softmax function rather than the sigmoid function.

The Softmax function calculates the probabilities distribution of the event over 'n' different events. In general way of saying, this function will calculate the probabilities of each target class over all possible target classes. Later the calculated probabilities will be helpful for determining the target class for the given inputs.

The main advantage of using Softmax is the output probabilities range. The range will **0 to 1**, and the sum of all the probabilities will be **equal to one**. If the softmax function used for multi-classification model it returns the probabilities of each class and the target class will have the high probability.

The formula computes the **exponential (e-power)** of the given input value and the **sum of exponential values** of all the values in the inputs. Then the ratio of the exponential of the input value and the sum of exponential values is the output of the softmax function.

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

we use the previous data:

```
# x is binary numbers 0 to 7
x =  np.array([[0,0,0],
             [0,0,1],
            [0,1,0],
            [0,1,1],
            [1,0,0],
            [1,0,1],
            [1,1,0],
            [1,1,1]])
```

Y is the number 0 to 7 to represent the binary numbers above

```
y = np.array([0,1,2,3,4,5,6,7])
```

We use the x and y values are used as both train and test values for greater accuracy. There is not enough bits for separate train and test data

```
x_train=x
x_test=x
y_train=y
y_test=y;
```

we make the LogisticRegression model

```
logreg = LogisticRegression()
```

we fit the model with data

```
logreg.fit(x_train,y_train)
```

we get prediction results

```
y_pred=logreg.predict(x_test)
print("prediction")
print(y_pred)
```

we make confusion matrix

```
cnf_matrix = metrics.confusion_matrix(y_test, y_pred)
```

	Class 1 Predicted	**Class 2 Predicted**
Class 1 Actual	True Positive	False Negative
Class 2 Actual	False Positive	True Negative

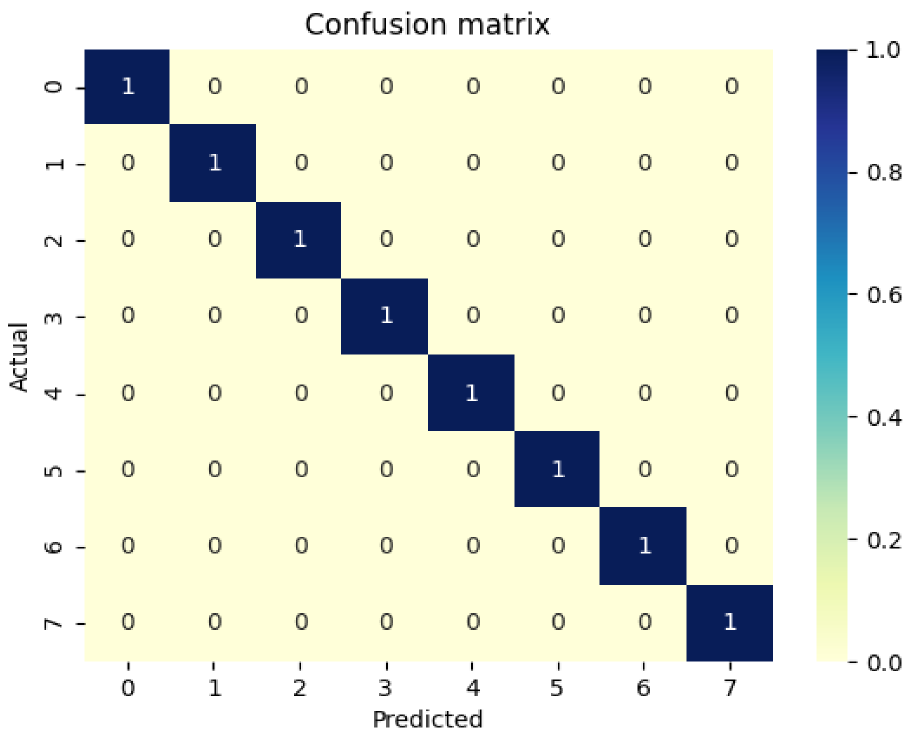
We plot confusion matrix and a heat map together

```
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
plt.title('Confusion matrix')
plt.ylabel('Actual ')
plt.xlabel('Predicted')
plt.show()
```

We print out the statics, we need to set average='weighted' because we no longer have binary classification

```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
print("Precision:",metrics.precision_score(y_test, y_pred,average='weighted'))
```

Here is the result



	0	1	2	3	4	5	6	7
0	1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0
3	0	0	0	1	0	0	0	0
4	0	0	0	0	1	0	0	0
5	0	0	0	0	0	1	0	0
6	0	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0	1

```
Accuracy: 1.0
Precision: 1.0
```

Here is the complete program:

```
#  Multinomial Logistic Regression

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
import pandas as pd

# x is binary numbers 0 to 7
x =  np.array([[0,0,0],
            [0,0,1],
           [0,1,0],
           [0,1,1],
           [1,0,0],
           [1,0,1],
           [1,1,0],
           [1,1,1]])

# Y is the number 0 to 7 to represent the binary numbers x
y = np.array([0,1,2,3,4,5,6,7])

# use the x and y values are used as both train and test values for greater accuracy.
# there is not enough bits for separate train and test data
x_train=x
x_test=x
y_train=y
y_test=y;

# make the LogisticRegression model
logreg = LogisticRegression()

# fit the model with data
logreg.fit(x_train,y_train)

#get  prediction results
y_pred=logreg.predict(x_test)
print("prediction")
print(y_pred)

# make  confusion matrix
cnf_matrix = metrics.confusion_matrix(y_test, y_pred)
```

```
# plot confusion matrix and a heat map together
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')

plt.title('Confusion matrix')
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
plt.show()

print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
print("Precision:",metrics.precision_score(y_test, y_pred,average='weighted'))
```

todo:

Try different binary inputs and output numbers.

LOGISTIC REGRESSION HOMEWORK   Question 5

Use Multinomial Logistic Regression to solve the house area and price problem from homework's question 1,2 and 3 Use x-axis for area and y –axis price.

Square Area (1000)	Price (100,000)
.5	1.4
2.3	1.9
2.9	3.2

Use whole numbers not fractions so multiply everything by 10. Plot a seaborn heat map and a dual plot of x_test vs y_pred and y_test vs y_pred. Save your homework 5 into a file called logistic_regression_homework5.py

Step 1: Multinomial Logistic Regression into file logistic_regression_homework5.py

Step 2: replace x with square area

```
# square area
x =  np.array([[5],
             [23],
            [29]
            ])
```

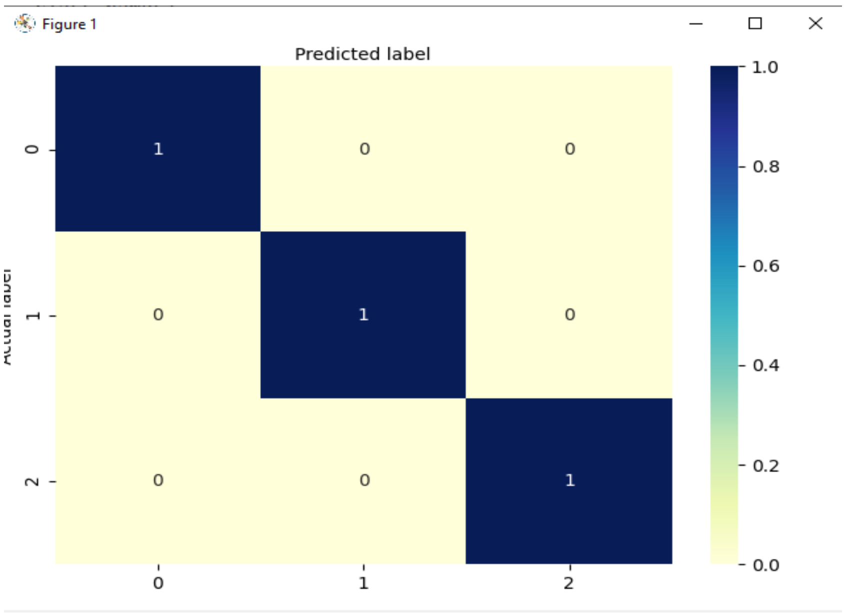
Step 3: replace y with prices

```
# prices
y = np.array([14,19,32])
```

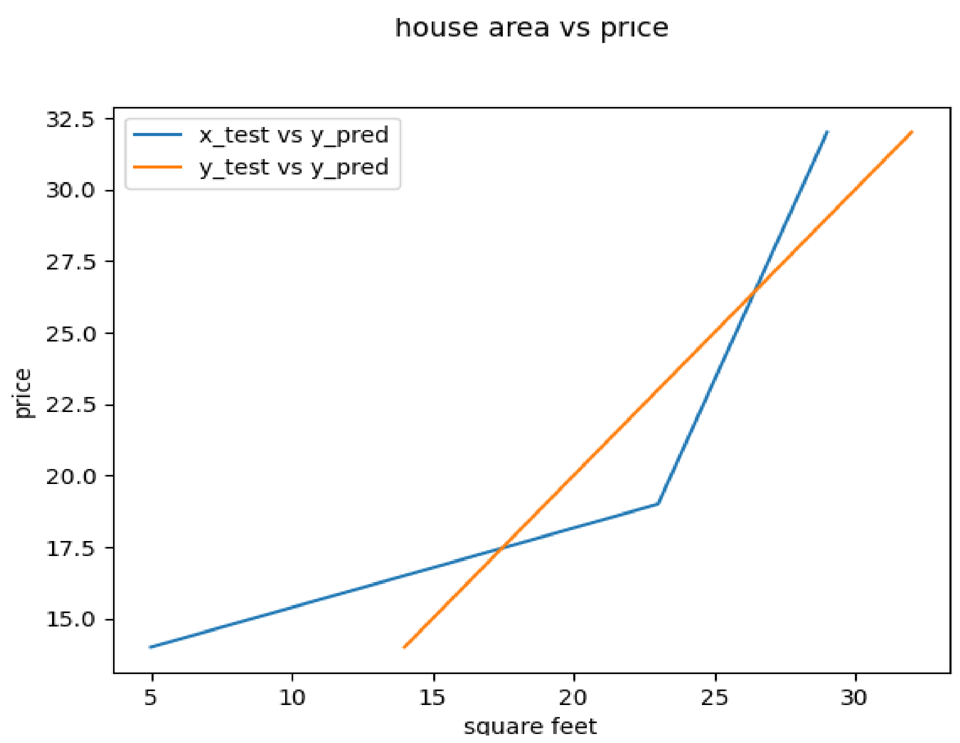
Step 4: plot of x_test vs y_pred and y_test vs y_pred

```
plt.plot(x_test, y_pred,label="x_test vs y_pred")
plt.plot(y_test, y_pred,label="y_test vs y_pred")
```

You should get something like this:



	0	1	2
0	1	0	0
1	0	1	0
2	0	0	1


house area vs price
x_test vs y_pred
y_test vs y_pred
price
square feet
32.5
30.0
27.5
25.0
22.5
20.0
17.5
15.0
5
10
15
20
25
30

End