Logistic Regression as a Neural network

Binary classification

Introduction

classification problem were you want to map a set of training examples with predefined set of features into class labels of only two possible values.

for example: assume you have a set of images (X) that represents animals and plants and you want to classify each image as either animal or plant, here you have just two class labels denoted as (Y) which are

- Animal label (0)
- Plant label (1)

Image representation in computers

Each image is composed of three colors Red, Green and Blue, these colors are used to represent images in a form of a matrix for each color where each matrix is constituted number of cells that represents the pixel intensity of certain color at certain cell which ranges from 0-255. further the dimensions of the color matrix corresponds to the original image dimensions in which if the image dimensions are 64*64, then, each matrix color will have the same dimension. and thus, an image is represented in the computer by these three colors matrices.

for machine learning and deep learning purpose, an is converted into training training example by unrolling the three colors matrices into a single column vector where each row represents a color cell and the ultimate dimension of the feature vector is image **(nx = image width * image length * 3)**:

for example : assume you have a certain image of size 3*3 pixels as follow:

```
import numpy as np
image = np.array([
                  #Red color matrix
                  [[1,2,3],
                   [4,5,6],
                   [7,8,9]],
                  #Green Color matrix
                  [[10,11,12],
                   [13,14,15],
                   [16,17,18]],
                  #Blue Color matrix
                  [[19,20,21],
                   [22,23,24],
                   [25,26,27]]
                   ])
image.shape = (3,3,3)
# the first digit represents the number of color matricis
#the second and third digits represents the image dimensions
unrolled_image = np.array([[
                            [ 1],
                            [ 2],
                            [ 3],
                            [ 4],
                            [ 5],
                            [ 6],
                            [ 7],
                            [ 8],
                            [ 9],
                            [10],
                                          [11],
                                          [12],
                                          [13],
                                          [14],
                                          [15],
                                          [16],
                                          [17],
                                          [18],
                                          [19],
                                          [20],
                                          [21],
                                          [22],
                                          [23],
                                          [24],
                                          [25],
                                          [26],
                                          [27]
                                     ]])
unrolled_image.shape =(1,27,1)
```

Important Notations

note: a symbol followed by (_) indicates subscript and (^) indicates superscript

a single training example is represented by (x,y) were x is an x-dimensional feature vector of length n_x and the label y belongs to {0,1}

m: the number of the examples in the datasets (training +testing)

$$M : \{(x^1, y^1), (x^2, y^2), (x^i, y^i)\ldots\ldots\ldots(x^m, y^m)\} \tag{1}$$

m-training: indicates the number of training examples

m-test: indicates the number of testing examples

X : the matrix of all training examples where each column represents a single training example and each row represents a feature. and thus the total number of columns equal M and the total number of rows equal to n_x and the The dimensions of X matrix is (n_x * M)

$$X = [x^1, x^2, x^i\ldots\ldots x^m] \tag{2}$$

Y: the matrix of all class labels values where each column represents the class label of the corresponding training example The dimensions of Y matrix is (1 * M)

$$Y = [y^1, y^2, y^i\ldots\ldots y^m] \tag{3}$$

Logistic Regression

Logistic function

given input x and you want an algorithm to predict y-hat = P(y=1|x), for example, if we have a plant picture, what is the probability that this picture is a plant picture where x is n_x dimensional vector an while the parameter (w) is n_x dimensional vector and b is real number, the th output y-hat is given by the equation the linear function :

$$y - hat = z = w^T x + b \tag{4}$$

but as we want the output of this equation to be between {0-1}, the previous equation in the current for could not serve the purpose of the binary classification as it may produce values greater than {0-1}. and thus, to overcome such drawback, a sigmoid function is applied as follow:

$$y - hat = sigmoid(w^T x + b) \tag{5}$$

where the sigmoid function is

$$sigmoid(z) = \frac{1}{1 + e^{-z}} \tag{6}$$

and hence, if z is a large value then the sigmoid output will approach to 1 and vice versa.

in logistic logistic regression you want to learn parameters {w,b} that give a better estimate of y-hat being either 1 or 0 according the available training examples.

Logistic Regression Cost Function

cost function

given M training example and you want to predict y-hat^(i) is approximately equal to y^(i) where :

$$y^i - hat = sigmoid(w^T x^i + b) \tag{7}$$

$$sigmoid(z^i) = sigmoid(z) = \frac{1}{1 + e^{-z^i}} \tag{8}$$

note that the superscript (i) represents the i-th training example.

the role of the loss function is to measure the performance of our model with respect to the original values which is defined as follow:

$$Loss = L(y - hat, Y) = 0.5(y - hat - y)^2 \tag{9}$$

other loss functions than equation (9) can be defined depending on the problem you solve

another form of loss function which is suitable for logistic regression is:

$$Loss = L(\hat{y}, y) = -((ylog(y - hat) + (1 - y)log(1 - y - hat)) \tag{10}$$

why equation (10) is suitable for logistic regression than (9)?

for (9) and (10) you want it to be as small as possible to approach the actual value

if y = 1 ====>>>> L =-log(y-hat) to be as small as possible <<<<==== log(y-hat) is large and y-hat is large

if y = 0 ====>>>> L =-log(1-y-hat) to be as small as possible <<<<==== log(1-y-hat) is large and y-hat is small

equation 10 gives a convex shape as (U) when plotted while while equation 9 gives non-convex shape as (w) when plotted and hence it is easier to find global minima in equation 10 while equation 9 may cause your algorithm to stuck to local minima rather than global minima.

up to this moment the loss is defined for a single training example. and thus to obtain the the losses for all training examples which is called the cost function and represents the average loss of all training examples is given as follow:

$$j(w, b) = -\frac{1}{m} * \sum_{i=1}^{n} L(y^i - hat, y^i) = -\frac{1}{m} \sum_{i=1}^{n} ((y^i log(y^i - hat) + (1 - y^i)log(1 - y^i - hat)) \tag{11}$$

Gradient Descent

Gradient descent algorithm

to obtain a good results out of your binary classification algorithm (logistic regression), you want to minimize the defined cost function j(w,b) which is a surface function of your algorithm (w,b) and th lowest point in this surface represents the minima of the cost function.

to minimize the the cost function, an initialization of the parameters (w,b) is performed first by any initialization method even zeros , consequently, the gradient descent takes a step in the steepest downhill direction of the cost function until you reach the global minima of the cost function.

Gradient algorithm steps

for each parameter in your algorithm find the derivative of the cost function with respect to that parameter and then subtract the derivative value times the learning rates from the current parameter value to obtain the new parameter value as follow:

$$w := w - \alpha \frac{dj(w)}{dw} \tag{12}$$

$$b := b - \alpha \frac{dj(b)}{db} \tag{13}$$

the learning rate is a small constant that specifies the step length to be moved by the gradient descent algorithm in downhill direction

even if your parameter is positive or negative, the gradient descent algorithm will end up with giving the optimum value of the parameter that minimize the cost function

the derivative of the at certain point represents the slope of the function at that point

Derivatives

Intuition about derivatives with linear function

assume you have straight line equation f(x) = 3x, when x = 2 f(x) = 6 and if x increased by a small value to be 2.001 and f(x) = 6.003, we can observe that increasing x by 0.001 increased f(x) by 0.003, and thus the slope (derivative) of the given function at the point 2 equal to:

```
slope = (6.003-6)/(2.001-2) = 3
```

```
the slope of the function (derivative) equal to ((f(x_2)-f(x_1))/(x_2-x_1)).

another example : x = 5 ===> f(x) = 15

•                                  x=5.001===>> f(x)=15.003

the the slope = ((15.003-15)/(5.003-5)) =3

df(x)/dx = 3
```

to conclude, the exact meaning of the derivative is the amount of change that happening to the function when you increase the value of the independent variable with a very very small amount.

More on Derivatives

Derivative with higher dimensional functions

assume you have straight line equation f(x) = x^2, when x = 2 f(x) = 4 and if x increased by a small value to be 2.001 and f(x) = 4.004, we can observe that increasing x by 0.001 increased f(x) by 0.004, and thus the slope (derivative) of the given function at the point 2 equal to:

```
slope = (4.004-4)/(2.001-2) = 4
```

```
another example : x = 5 ===> f(x) = 25

•                                  x=5.001===>> f(x)=25.01

the the slope = ((25.01-25)/(5.001-5)) =10

df(x)/dx = 10
```

you can observe that the derivative value changed by changing the value of x and thus for non-linear function the slope value differs from point to point this is because the shape of the function differs at different point as an example for f(x) =x^2 the shape of the function at x= 0 is nearly ensembles a horizontal line while at higher x values say 10, 100 or 1000 the function shape tend to have vertical shape and thus the derivative values differs at different points.

to obtain the derivative of any function multiply the variable by its current power value and subtract one from the power for each variable in the equation as follow:

$$\frac{df(x)}{dx} = power * x^{power-1} \tag{14}$$

if you increase x value by any value you will obtain the f(x) changed with the (slope value) multiplied by the amount of changes, for example:

for f(x) = x^2 , increasing x by .001 will results in increase in f(x) of (2*x)*0.001

Computational Graphs

Computational graphs basics

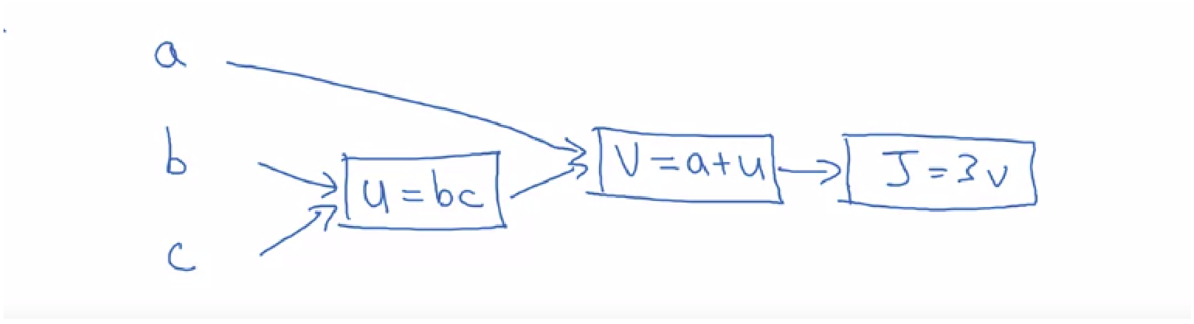
Computational graph is a graphical representation of the calculations that pertains a certain equation to obtain the ultimate value of that calculation in conquer and divide fashion , such method is used heavily in Neural Networks (nn) as the nn has two phases of calculation which are forward propagation and backward propagation, and thus, performing such computations using computational graphs would make it easier to handle complex computations in a step by step way, to understand how computational graphs works look at the following equation

```
j(a,b,c) = 3(a +bc)
```

to evaluate this equation at certain value we need first to divide this equation into smaller chunks as follow:

let :

```
u =bc
v = a +u
j =3v
```



figure(1): Equation representation by computational graph.

you can observe from figure(1) that the original variables (a,b,c) represents the input to the computational graph, while the defined variables (u,v,j) represents the mathematical computation on the input variables, for example, assume :

```
a = 5, b = 3 c = 2
u = 3*2 = 6
v = 5+6 = 11
j = 3*11 = 33
```

these computations represents the forward pass to obtain the equation value, a backward computation can be performed on the same graph which is useful in computing the back propagation computations in nn.

one more thing to mention is the arrows represents the direction of the computations while the rectangles represents the mathematical operations.

Derivative with computational graphs

Computing derivative

depending on figure (1), assume you want to calculate the derivative dj/dv and you have :

```
j = 3v
v = 11 ==> j = 33
v = 11.001 ==> j = 33.003
dj/dv = 33.003-33/11.001-11 = 3
```

up to this moment we have performed one step back propagation by deriving (j) wit respect to v.

to gain wider view assume we want to compute dj/da, in other words, what will happen to the value of (v) when we slightly increase (a):

```
a= 5 ==> v= 5+6 =11 ==> j = 3*11 =33
a= 5.001 ==> v= 5.001+6 =11.001 ==> j = 3*11.001 =33.003
dj/da = 33.003-33/11.001-11 = 3
```

The chain rule

if we have variable (a) that changing it affects the variable (v) which in turn affect the variable (j) then the amounts of change in (j) when changing (a) is the product of the change in (v) when (a) changed times the change in (j) when changing (v) which can be mathematically represented as

$$\frac{dj}{da} = \frac{dj}{dv} * \frac{dv}{da} \tag{15}$$

when doing back propagation you really care about the final variable you want to optimize which is (j) in our case, and thus in programming back propagation we will refer to the derivative with respect to a certain variable, say (a) for example, in our code as (da) which represents the derivative dj/da.

let's complete our example by computing (du)

```
u= 6 ==> v= 5+6 =11 ==> j = 3*11 =33
u= 6.001 ==> v= 5.001+6 =11.001 ==> j = 3*11.001 =33.003
dj/du = dj/dv * dv/du = 3 *1 =3
```

for (db) and (dc) bu chain rule

```
db = dj/dv * dv/du * du/db = (3 * 1 * c) = 3 * 1 * 2 = 6
dc = dj/dv * dv/du * du/dc = (3 * 1 * b) = 3 * 1 * 3 = 9
```

Logistic Regression Gradient Descent 1

Gradient descent for single training example

let's have a look about the logistic regression equations we have defined earlier

the input equation is :

```
z = W^T*x +b
```

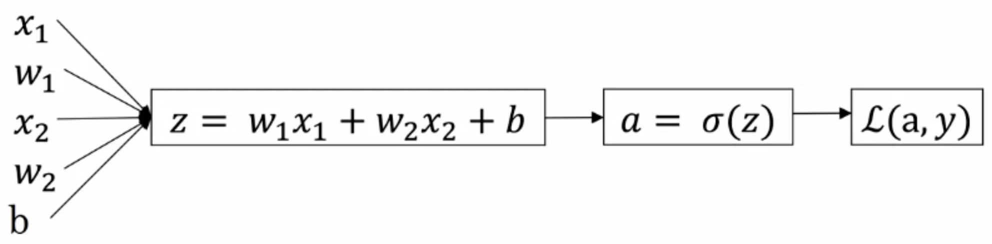
the output equation is :

```
y-hat = a = sigmoid(z)
```

the loss function is:

```
L(a,y) = -(ylog(a)+(1-y)log(1-a))
```

assume we have a training example with just two input feature x_1 and x_2 . and thus to compute (z) we need a parameter (w) for each input feature as follow w_1 and w_2 for x_1 and x_2 respectively as well as a bias parameter (b). to perform the gradient descent algorithm, we will represents the equations as a computation graph to as shown in figure(2)



Figure(2): Logistic regression computational graph

our purpose is to modify w_1, w_2 and b to minimize the loss function L(a,y), so that we will derive the loss function with respect to w_1, w_2 and b by going back through the computational graph until reaching the parameters w_1, w_2 and b as follow.

```
the derivataives:
>> dl/da derivative:
>> dl/da = d/da (-(ylog(a)+(1-y)log(1-a)))
>> knowing from calculas that log_e(x) derivative is 1/x and (1-a) is -1:
>> dl/da = ((-y/a)+(1-y)/(1-a))
>> by simplification:
>> dl/da = ((a-y)/a(1-a))


>> dl/dz derivative:
>> by chain rule:
>> dl/dz = dl/da * da/dz     (1)
>> dl/dz = ((a-y)/a(1-a)) * da/dz
>> da/dz = d/dz sigmoid(z)
>> da/dz = sigmoid(z)*(1-sigmoid(z))
>> as a = sigmoid(z)
>> da/dz = a(1-a)            (2)
>> by substituting (2) into (1) we get :
>> dl/dz = ((a-y)/a(1-a)) * a(1-a)
>> dl/dz = a-y


>> dl/dw_1 derivative:
>> by chain rule:
>> dl/dw_1 = dl/da * da/dz * dz/dw_1
>> dl/dw_1 = ((a-y)/a(1-a)) * a(1-a) * dz/dw_1
>> dl/dw_1 = a-y * dz/dw_1      (3)
>> knowing from calculas that deriving equation that has multiple variables
with respec one of them, the derivation will be perofrmed only with respect to
that varible and the other variable will be treated in derivation as constant
which mean their derivative is zero
>> dz/dw_1 = d/dw_1 (w_1*x_1 + w_2*x_2 + b)
>> dz/dw_1 = x_1                (4)
>> by substituting (4) into (3) we get :
>> dl/dw_1 = (a-y) * x_1


>> dl/dw_2 derivative:
>> dl/dw_2 = (a-y) * x_2


>> dl/db derivative:
>> dl/db = (a-y)


the updates:
w_1 := w_1 - alpha*dw_1
w_2 := w_2 - alpha*dw_1
b = b - alpha*db
```

Extra reading:

Log_e derivative : Logarrithmic function derivation rules

Sigmoid derivative: step-by-step sigmoid derivation

Chain rule : The Chain Rule

Logistic Regression Gradient Descent 1

Gradient descent on multiple training example

up to this point we have calculate gradient descent for a single training example, what we are going to do in this section is to compute the gradient descent for all training examples. recall from equation (11) that the cost function of all examples is just the average of the losses from all training examples as follow:

```
j(w,b) = (1/m)*sum[i=1:m](L(a^i,y^i))
a^i = y^i-hat = sigmoid(z^i) =  sigmoid(w^T*x^i + b)
from the previous section we know how to compute each of:
dw_1^i, dw_2^i and db^i for a single training example (x^i,y^i)

as we said previously that the overall cost function is the average of the
individual losses so it turns out that the derivative of the cost function
j(w,b) with respect to w_1 is going to be the average of the derivatives with
respect to w_1 of the individual losses (x^i,y^i) as follow:
dj(w,b)/dw_1  = (1/m)*sum[i=1:m] d(L(a^i,y^i))/dw_1

let's start with the comlete algorithm
#first we have to initialize our variables of interest
J = 0
dw_1 = 0
dw_2 = 0
db = 0

#next we will start a for loop to compute the derivative for all training
examples
for i = 1 to m:
    z^i = w^T*x^i + b
    a^i = sigmoid(z^i)
    j += -(y^i*log(a^i) + (1-y^i)log(1-a^i))

        for j = 1 to n:
            dz^i = a^i - y^i
            dw_1 += x_1^i*dz^i
            dw_2 += x_2^i*dz^i
            ....
            ....
            ....
            dw_n += x_n^i*dz^i
            db += dz^i
#note that we are using dw_1, dw_2 and db as accumelators as they represent the
derivative of the over all cost function with respect to each of these
acculmelators, so that they do not have superscript i while dz is computed for
single training example and thus it has a superscript i

# and ultimately computing the avreges
J /= m
dw_1 /= m
```

```
dw_2 /= m
db /= m

# the last step is to update the wieghts as follow
w_1 := w_1 - alpha*dw_1
w_2 := w_2 - alpha*dw_1
b = b - alpha*db
```

Python and Vectorization

Vectorization

What is Vectorization?!

Neural networks computations are performed using matrices, and thus to code your neural network you have to use for loop to pass over each weight variable or feature, as an example , consider the logistic regression where you have to compute the following term:

```
z = W^T * x + b
where:
w: a column vector representing th weights
x : column vector representing the features

to perform this operation in non-ectorized form:
z  0
for  in range(n_x):
    z+= w[i] * x[i]
z +=b

the vectorized implementation:
z = np.dot(w,x) + b
```

And hence, the main purpose of using Vectorization is to git rid of using explicit for loops in your code which may take a very long duration especially if your data is very large but rather using built in functions like those available in numpy library which are optimized functions

lets explain it in a demo python code to see the difference between vectorized and non-vectorized computations:

```
import numpy as np
import time

a = np.random.rand(1000000)
b = np.random.rand(1000000)
tic = time.time()
c = np.dot(a,b)
toc = time.time()
print('Vectorized veriosn: 'str(1000*(toc-tic))+'ms')
Vectorized veriosn: 3.5 ms
```

```
c =0
tic = time.time()
for i in range(1000000):
    c += a[i] * b[i]
toc = time.time()
print('Non-vectorized veriosn: 'str(1000*(toc-tic))+'ms')
Non-vectorized veriosn: 475 ms
```

More Vectorization Examples

Neural network programming Guideline

Rule : Whenever possible, avoid using explicit for-loop

example 1:

```
u = A*v
where A is a matrix and v is  a vector
u_i = sum_j(A_ij * v_j)

Non-vectorized:
u =np.zeros(n,1)
for i ...
    for j ...
        u[i] += A[i][j] * v[j]

Vectorized:
u = np.dot(A,v)
```

example 2:

```
you have:
V =[v_1, v_2,.........v_n]
you want to compute the e of each value

Non-vectorized:
u = np.zeros((n,1))
for i in range(n):
    u[i] = math.exp(v[i])

vectorized:
u = np.exp(v)
```

Vectorizing Logistic Regression

Forward propagation Vectorization

to make prediction via logistic regression on m training examples you have to make the following computations:

```
for each training example:
z^1 = W^T*x^1 +b        z^2 = W^T*x^2 +b        z^1 = W^T*x^2 +b        z^m =
W^T*x^m +b
a^1 = sigmoid(z^1)      a^2 = sigmoid(z^2)      a^3 = sigmoid(z^3)      a^m =
sigmoid(z^m)
```

to do so you need a for loop but luckily there is a way to do so without using for loops, let's see how:

```
remember that:
X is a matrix that contain all training examples stacked in columns
X =[x^1, x^2,x^3 ...x^m] wit shape (n_x,m)

to compute z^1,....z^m, first we have to define (1,m) row vector of z's
[z^1,z^2,z^3......z^m](1,m) = w^T * X +[b b b b](1,m)
w^T is a row vector of shape (1,nx)
by matrix multiplication
[z^1,z^2,z^3......z^m]= w^T * X +[b b b b] = [w^T * x^1 + b, w^T * x^2 + b
..w^T * x^m + b]
the matrix b in reality is a single real number but it is extended by python to
the same shape of the left operand to complete the summation
by python this operation is performed as
Z = np.dot(w.T,X) +b

for a's:
A = [a^1, a^2, a^3....a^m] wit shape of (1,m)
numpy exponential function np.exp() can take an array and returun the
exponential out of it, so that to compute the A just define the sigmoid
function and pass the Z array to it to get A array

for the cost function

cost = (-1/m)*(np.dot(Y,np.log(A).T)+ np.dot((1-Y),np.log(1-A).T))
```

Backward propagation Vectorization

to to perform back propagation via logistic regression on m training examples you have to make the following computations:

```
dz^1 = a^1 -y^1     dz^2 = a^2 -y^2     dz^3 = a^3 -y^3     dz^m = a^m -y^m
to do so we have to define array dZ that contains all individual dz's as
follow:
dZ = [dz^1, dz^2, dz^3,.....dz^m] with shape (1,m)

then A is difined from the previous step and Y is the data label and both
are(1,m) arays
dZ = A-Y = [a^1-y^1, a^2-y^2, a^3-y^3, ..... a^m-y^m]

now for db and dw
db = (1/m) * sum_i_m(dz^i)
and thus by vectorization:
```

```
db =  (1/m) * np.sum(dZ)
for dw:
dw = (1/m) * X *dz = (1/m) * X * dZ = 1/m [x^1, x^2, x^3....x^m] *
[dz^1,dz^2,dz^3,...dz^m]
remember that X shape is (nx,m) and dZ.T is a cpolumn vector of shape (m,1) and
thus by matrix multiplication:
dw = (1/m) * [x^1*dz^1, x^2*dz^2,x^3*dz^3,....x^i*dz^i] with shape of (nx,1)
```

put them all together

```
#Using one for loop to perform multible iterations
for i in range(1000):
    #Forward propagation
    Z = np.dot(w^T, X) + b
    A = sigmoid(Z)
    cost = (-1/m)*(np.dot(Y,np.log(A).T)+ np.dot((1-Y),np.log(1-A).T))

    #Backward propagation
    dZ = A - Y
    dw = (1/m) * np.dot(X dZ.T)
    db = (1/m) * np.sum(dZ)

    #Weights updatae
    w = w - alpha * dw
    b = b - alpha * db
```

Broadcasting in python

Broadcasting

Assume you have the following 2*2 X matrix   matrix:

```
X = [[1,2],
     [3,4]]
our goal is to add 100 to X
then by broadcasting, python would convert 100 into 2*2 matrix and add it up to
X as follow
X = [[1,2],     +   100     = [[1,2],   + [[100, 100]       =       [[101, 102]
     [3,4]]                    [3,4]]       [100,100]]      =        [103,
104]]

example 2: adding (m,n) matrix to (1,n) matrix
[[1, 2, 3]      + [1, 2, 3]     = [[1, 2, 3]        + [1, 2, 3]     =   [[2, 5,
6]
 [4, 5, 6]]                         [4, 5, 6]]        [1, 2, 3]]         [5, 7,
9]]
```

General broadcasting principle

if you have (m,n) matrix and you performed any of these orations(+, -, *, /) with (1, n) or (m, 1)matrix then the latter will be converted into (m, n) matrix to complete the operation element-wise

if you have (m, 1) or (1, n) matrix and you performed any of these orations(+, -, *, /) with real number then the real number will be converted into (m, 1) or (1, n) matrix to complete the operation element-wise

A note on python/numpy vector

Rank-one arrays

Rank-one array are vectors that have the shape (x,) in python and x can be any >0 integer, the problem with this type of data structure that it can not be considered as either column or row vector and thus performing operations such as transposition or inner product will produce misleading output that cause your code to have not easy to detect bugs:

```
import numpy as np
a = np.random.randn(3)
print(a)
[0.57, 0.95, 0.32]
print(a.shape)
(3, )
print(a.T)
[0.57, 0.95, 0.32]
print(np.dot(a,a.T))
2.1256
```

to avoid such issue, specify the dimensions of the defined array explicitly as follow:

```
a = np.ranom.randn(3,1)
print(a)
[[0.57],
 [0.95],
 [0.32]]
print(a.T)
[[0.57, 0.95, 0.32]]
```

an extra trick to avid rank-one arrays is to use assert statement as follow:

```
# for column vectors
assert(a.shape == (x,1))
# for row vector
assert(a.shape == (1,x))
```

the ultimate trick is to reshape rank-one array as follow:

```
# for column vectors
a = a.reshape((x, 1))
# for row vector
a = a.reshape(1, x)
```

Explanation of logistic regression cost function

Logistic regression loss function

let's have a better understanding of logistic regression loss function:

```
y-hat = sigmoid(w^T * x + b) wher sgmoid = 1/(1+e^-z)
and we want our algorithm to learn that:
y-hat = 1 as p(y = 1| x)

more clearly:
if: y = 1 then p(y | x) = y-hat         (1)
conversely by propabilty rules:
if: y = 0 then p(y | x) = 1 - y-hat     (2)
equations (1) and (2) represent the propability of a certain class giiven the
input data p(y | x), to summarize these two equations, the can be written in
the following form

p(y | x) = (y-hat)^y * (1 - y-hat)^(1-y)    (3)
to explain equation (3) summarize both (1) and (2)

if y = 1 then (y-hat)^1 * (1 - y-hat)^(1-1) = (y-hat) * (1 - y-hat)^(0) = y-hat
if y = 0 then (y-hat)^0 * (1 - y-hat)^(1-0) = (y-hat)^0 * (1 - y-hat)^(1) = 1 -
y-hat

Now, finally, because the log function is a strictly monotonically increasing
function, your maximizing log p(y|x) should give you a similar result as
optimizing p(y|x) then
log p(y | x) = log((y-hat)^y * (1 - y-hat)^(1-y))
which gives (by calculas) the following equation:
L(y-hat, y) = -(y * log(y-hat) * (1-y) * log(1 - y-hat))
and the negative sign comes from to indicate maximizing the propapility as we
minimizing the loss function
```

####

Logistic regression cost function

let's have a better understanding of logistic regression cost function:

```
p(all labels in the training set) = product_i=1_m( p(y^i | x^i))

And so if you want to carry out maximum likelihood estimation, right, then you
want to maximize the, find the parameters that maximizes the chance of your
observations and training set. But maximizing this is the same as maximizing
the log as follow:

log(p(all labels in the training set)) = log(product_i=1_m( p(y^i | x^i)))
= sum_i=0_m(log(p(y^i | x^i)) = -sum_i=0_m(L(y-hat^i y^i))
And then finally for convenience, to make sure that our quantities are better
scale, we just add a 1 over m extra scaling factor there
j(w,b) = (-1/m) * (-sum_i=0_m(L(y-hat^i y^i)))
```