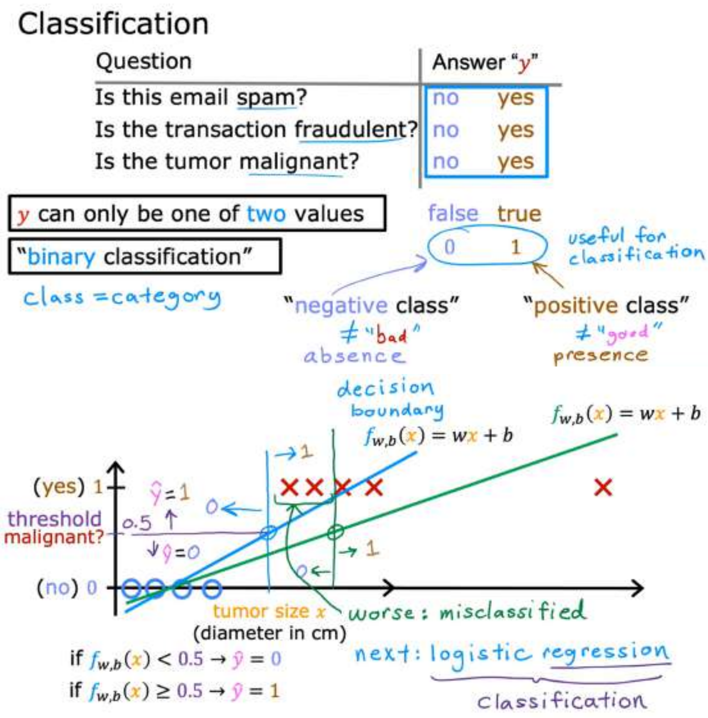
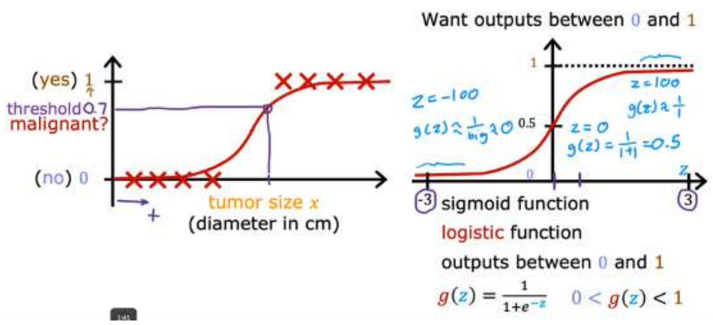
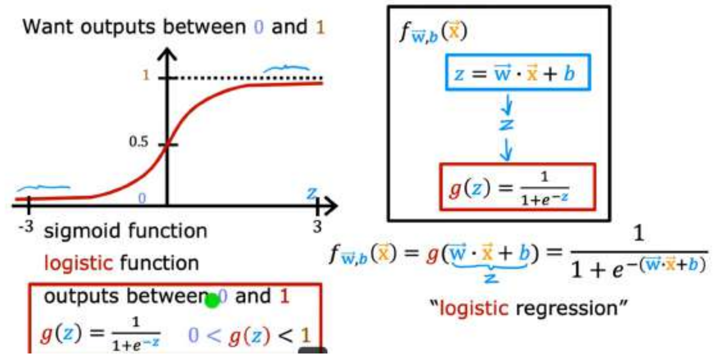
Week 3

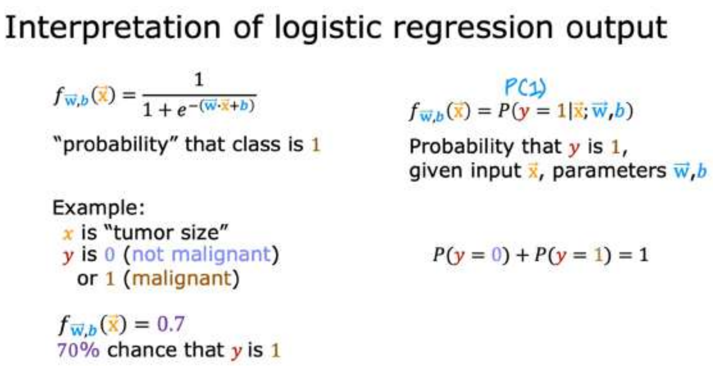
Motivations



Logistic regression







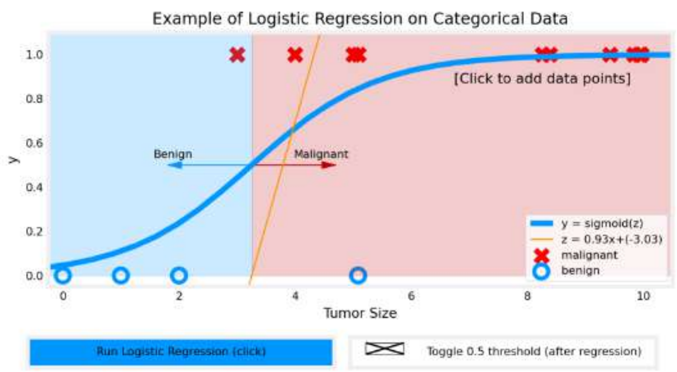
C1_W3_Lab02_Sigmoid_function_Soln

- we would like the predictions of our classification model to be between 0 and 1 since our output variable y is either 0 or 1.
- This can be accomplished by using a "sigmoid function" which maps all input values to values between 0 and 1.

```
 # Input is an array.
input_array = np.array([1,2,3])
exp_array = np.exp(input_array)
print("Input to exp:", input_array)
print("Output of exp:", exp_array)
# Input is a single number
input_val = 1
exp_val = np.exp(input_val)
print("Input to exp:", input_val)
print("Output of exp:", exp_val)
#Input to exp: [1 2 3]
#Output of exp: [ 2.72  7.39 20.09]
#Input to exp: 1
#Output of exp: 2.718281828459045


def sigmoid(z):
    g = 1/(1+np.exp(-z))

    return g




# Generate an array of evenly spaced values between -10 and 10
z_tmp = np.arange(-10,11)
# Use the function implemented above to get the sigmoid values
y = sigmoid(z_tmp)
#Input (z), Output (sigmoid(z))
#[[-1.000e+01  4.540e-05]
 #[-9.000e+00  1.234e-04]
 #[-8.000e+00  3.354e-04]
 #[-7.000e+00  9.111e-04]......
```

```
x_train = np.array([0., 1, 2, 3, 4, 5])
y_train = np.array([0,  0, 0, 1, 1, 1])

w_in = np.zeros((1))
b_in = 0
```

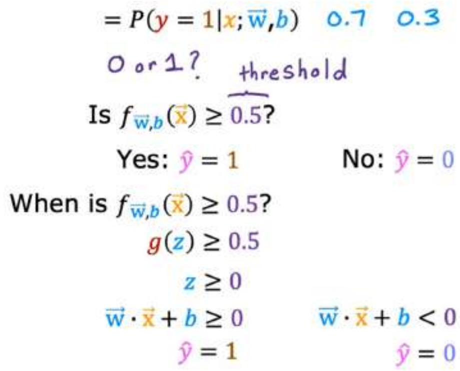
here we are giving a bunch of x having values 0 or 1 and we make a linear regression model and fit it in a sigmoid function to get a logistical regression model which is then plotted to give this curve where y is btw 0 and 1 and x is your normal x (input data x)



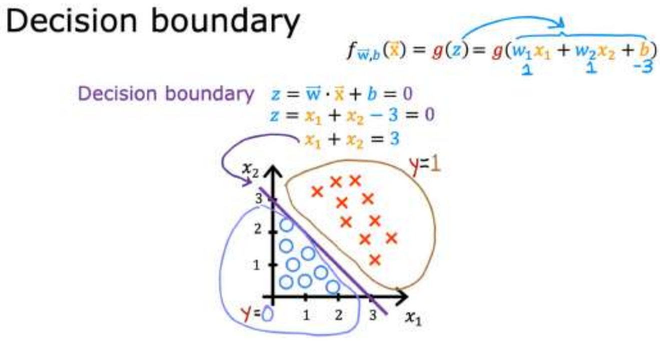
Decision boundary

- sigmoid fn is also called logistic fn

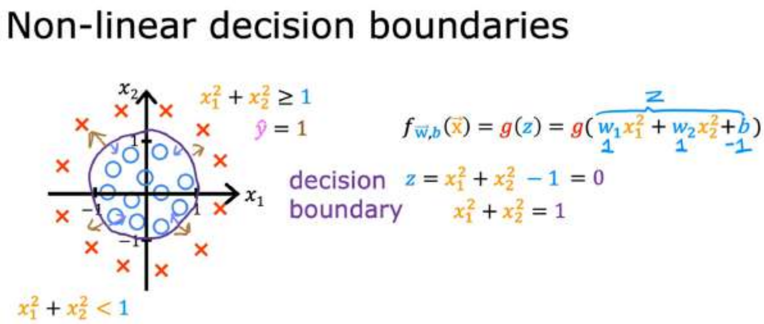
$$f_{\vec{w},b}(\vec{x}) = g(\underbrace{\vec{w}\cdot\vec{x}+b}_{z}) = \frac{1}{1+e^{-(\vec{w}\cdot\vec{x}+b)}}$$

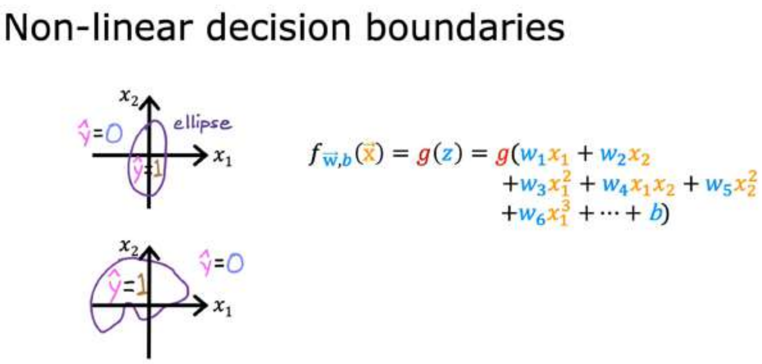


the decision boundary may not be a linear boundary always but also non linear/different shapes for other complex functions.



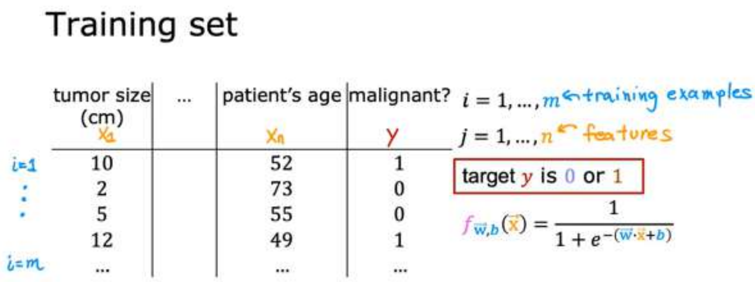
> **For a decision boundary line, z=0!!! the line for boundary is found at z=0**





Cost function for logistic regression

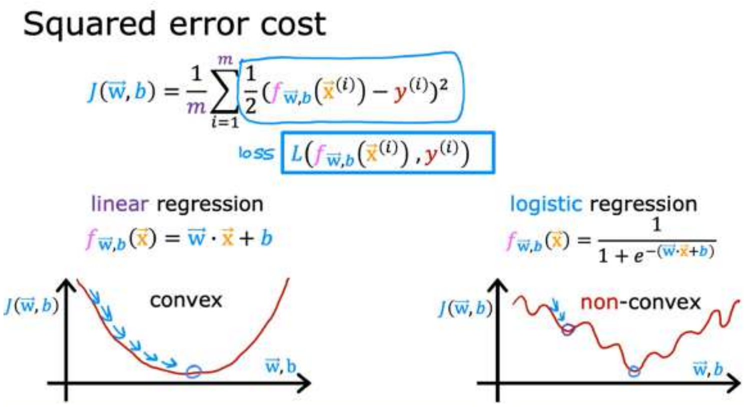
- **the loss function measures how well you're doing on one training example and is by summing up the losses on all of the training examples that you then get, the cost function, which measures how well you're doing on the entire training set**



Training set

	tumor size (cm) x_1	...	patient's age x_n	malignant? y
i=1	10		52	1
:	2		73	0
:	5		55	0
	12		49	1
i=m	...		...	...

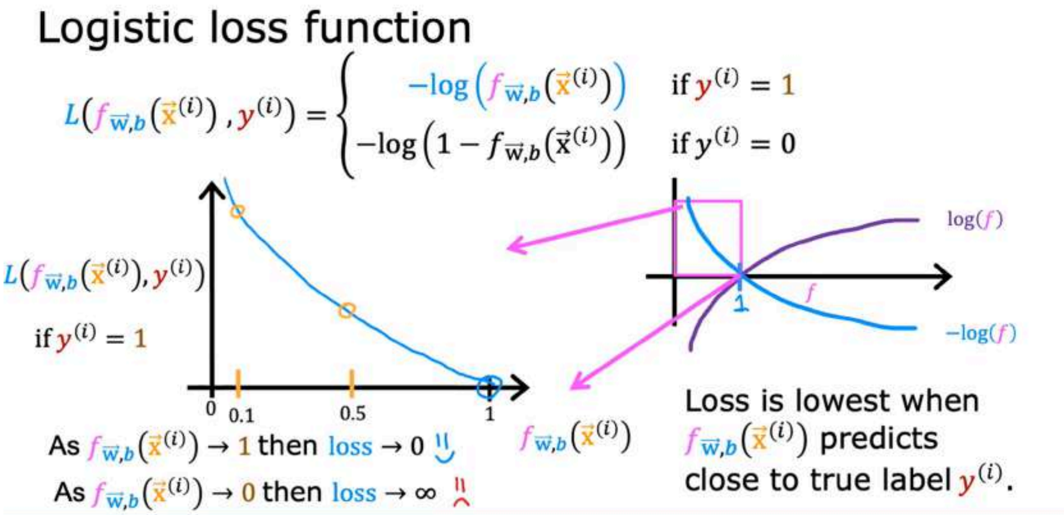
here we are given n features with m training examples, the objective is to find the right set of [w1,w2,w3 ..] and b using the cost function for logistic regression. (not squared error cost fn!!!)



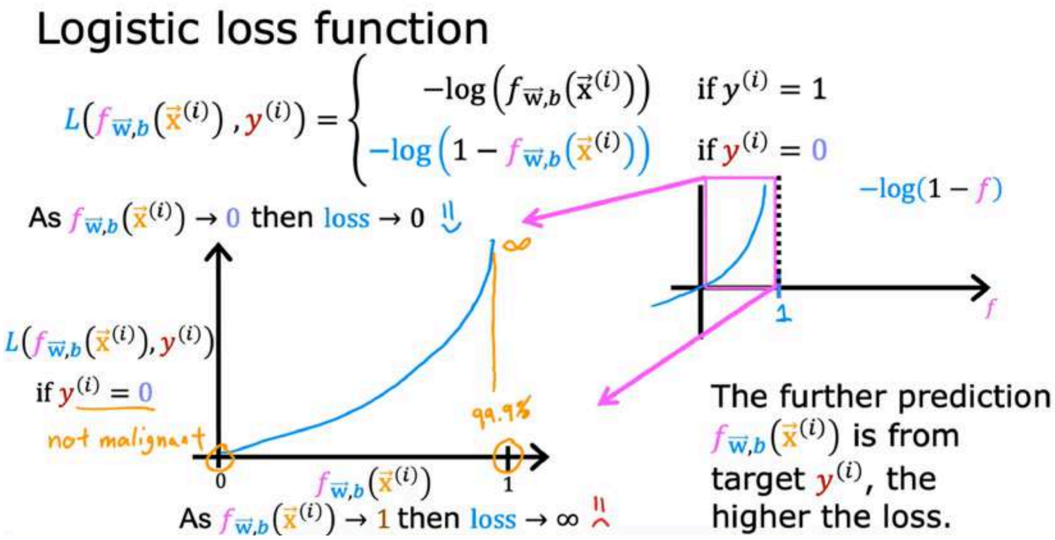
here we see that we have modified the cost fn b taking 1/2 inside the summation and defined the loss **L** to be equal to the term inside ∑

loss $L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}) = \frac{1}{2}(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})^2$

the loss fn takes to parameters, f(x) and output y (0 or 1)
also in the image, in linear regression the curve is convex when plotting cost fn against weights but in logistic regression, it is a non-convex with multiple local minimums hence its hard to do GD, thus we need to make it a good curve.



- here we see we have defined Loss in such a way that the curve is smooth. also the fn is different for y=0 and y=1.
- *the x axis of the graph on the left is fwb(xi) (the prediction), this graph tells us* ***that given real output is 1 and the prediction should be 1*** *we are plotting loss against predicted output (between 0 and 1 obv)*
- here for y=1, we see that as the model predicts output (y) closer to 1, the cost tends to 0. this means that lesser the cost, more the model pushes to adapt to the answer having loss tending to 0. when output tends to 0, loss tends to ∞.
- also in the graph on the right, since we are dealing with 0<y<1, we only consider the part/curve of the graph between 0 and 1.
we penalize the model if the loss is very high! (discourage)



- *the x axis of the graph on the left is fwb(xi) (the prediction), this graph tells us* ***that given real output is 0 and the prediction should be 0*** *we are plotting loss against predicted output (between 0 and 1 obv)*
- here for y=0, we see that as the model predicts output (y) closer to 0, the cost tends to 0. this means that lesser the cost, more the model pushes to adapt to the answer having loss tending to 0. when output tends to 1, loss tends to ∞.
here above the loss is defined like this

$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}),y^{(i)}\right)=\begin{cases}-\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)}=1\\ -\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)}=0\end{cases}$$

the loss is defined for every training example *(i)* and the cost is defined as such

Cost

cost

$$J(\vec{w},b)=\frac{1}{m}\sum_{i=1}^{m}\underbrace{L\left(f_{\vec{w},b}(\vec{x}^{(i)}),y^{(i)}\right)}_{\text{loss}}$$

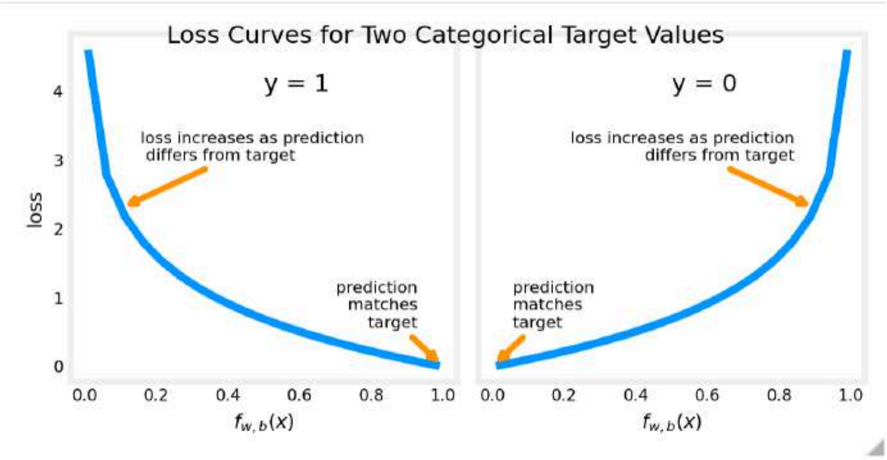
$$=\begin{cases}-\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)}=1\\ -\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)}=0\end{cases}$$

convex
↳ can reach a global minimum

find w, b that minimize cost J

so we know the real output (0 OR 1), we know the fw,b(x(i)) (the prediction), for every iteration we will choose the correct formula of loss depending on whether the real answer is 0 or 1 and then find the loss by putting fw,b(x(i)) in the equation and finally sum all the losses and divide by m (no. of training examples) to get total cost. objective is now to reduce cost to finally get some set of optimized w,b that minimizes cost! *remember the model is z = w.x + b* and logistic regression is 1/(1+e^-(w.x + b)) and this is the actual predicted output!
z = w.x + b
fwb(z) = 1/(1+e^-(w.x + b)) = 1/(1+e^-(z))

Loss is a measure of the difference of a single example to its target value while the **Cost** is a measure of the losses over the training set



Simplified Cost Function for Logistic Regression

Simplified loss function

$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}),y^{(i)}\right)=\begin{cases}-\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)}=1\\ \boxed{-\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right)} & \text{if } y^{(i)}=0\end{cases}$$

$$\boxed{L\left(f_{\vec{w},b}(\vec{x}^{(i)}),y^{(i)}\right)=-\underset{0}{\cancel{y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right)}}-\underset{(1-0)}{(1-y^{(i)})}\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right)}$$

if $y^{(i)}=1$:

$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}),y^{(i)}\right)=-1\log(f(\vec{x}))$$

if $y^{(i)}=0$:

$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}),y^{(i)}\right)=-(1-0)\log(1-f(\vec{x}))$$

the loss function can be written in 1 single line like shown above and plugging in 0 or 1 will give your the appropriate eqn

Simplified cost function

loss

$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right) = -y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) - (1-y^{(i)})\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right)$$

cost

$$J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}\left[L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right)\right]$$

convex
(single global minimum)

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

maximum likelihood
(don't worry about it!)

the reason we chose this particular cost fn is because of the statistical method of "maximum likelihood" and also that this gave us a convex curve

$$\boxed{L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right) = -y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) - (1-y^{(i)})\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right)}$$

loss

$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right) = \frac{1}{2}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2$$

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

cost

C1_W3_Lab05_Cost_Function_Soln

Cost function

In a previous lab, you developed the *logistic loss* function. Recall, loss is defined to apply to one example. Here you combine the losses to form the **cost**, which includes all the examples.

Recall that for logistic regression, the cost function is of the form

$$J(\mathbf{w},b) = \frac{1}{m}\sum_{i=0}^{m-1}\left[loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)})\right] \tag{1}$$

where

- $loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)})$ is the cost for a single data point, which is:

$$loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)}) = -y^{(i)}\log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) \tag{2}$$

- where m is the number of training examples in the data set and:

$$f_{\mathbf{w},b}(\mathbf{x^{(i)}}) = g(z^{(i)}) \tag{3}$$
$$z^{(i)} = \mathbf{w}\cdot\mathbf{x}^{(i)} + b \tag{4}$$
$$g(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}} \tag{5}$$

```python
import numpy as np
from lab_utils_common import  plot_data, sigmoid, dlc

X_train = np.array([[0.5, 1.5], [1,1], [1.5, 0.5], [3, 0.5], [2, 2], [1, 2.5]])  #(m,n)
y_train = np.array([0, 0, 0, 1, 1, 1])                                           #(m,)

def compute_cost_logistic(X, y, w, b):
# here the linear model z = x.w + b
    m = X.shape[0]
    cost = 0.0
    for i in range(m):
        z_i = np.dot(X[i],w) + b
        f_wb_i = sigmoid(z_i) #retuns the sigmoid of z, this is not a built in but imported, see top
        cost +=  -y[i]*np.log(f_wb_i) - (1-y[i])*np.log(1-f_wb_i)

    cost = cost / m
    return cost

w_tmp = np.array([1,1])
b_tmp = -3
print(compute_cost_logistic(X_train, y_train, w_tmp, b_tmp))
```

Gradient Descent Implementation

in GD the algo remains same, simultaneous update of w and b and that same formula we used, shown below
aim: to find w (vector) and b to be able to predict output for a fresh new dataset.

Gradient descent

cost

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

repeat {

$j=1\ldots n$

$$w_j = w_j - \alpha\frac{\partial}{\partial w_j}J(\vec{w},b) \qquad \frac{\partial}{\partial w_j}J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}$$

$$b = b - \alpha\frac{\partial}{\partial b}J(\vec{w},b) \qquad \frac{\partial}{\partial b}J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)$$

} simultaneous updates

remember we have j no. of w's, thus while finding out derivative of cost wrt j, we will use x (i,j)

Gradient descent for logistic regression

repeat { looks like linear regression!

$$w_j = w_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}\right]$$

$$b = b - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)\right]$$

} simultaneous updates

Same concepts:
- Monitor gradient descent (learning curve)
- Vectorized implementation
- Feature scaling

Linear regression $f_{\vec{w},b}(\vec{x}) = \vec{w}\cdot\vec{x} + b$

Logistic regression $f_{\vec{w},b}(\vec{x}) = \frac{1}{1+e^{-(\vec{w}\cdot\vec{x}+b)}}$

The formula of dj_dw and dj_db is the same as LR

C1_W3_Lab06_Gradient_Descent_Soln

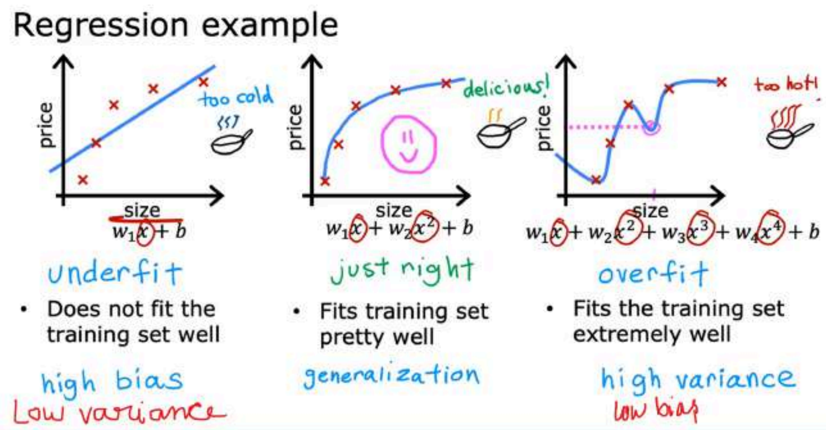
```
def compute_gradient_logistic(X, y, w, b):
    m,n = X.shape
    dj_dw = np.zeros((n,))                           #(n,)
    dj_db = 0.

    for i in range(m):
        f_wb_i = sigmoid(np.dot(X[i],w) + b)          #(n,)(n,)=scalar
        err_i  = f_wb_i  - y[i]                       #scalar
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err_i * X[i,j]      #scalar
        dj_db = dj_db + err_i
    dj_dw = dj_dw/m                                   #(n,)
    dj_db = dj_db/m                                   #scalar

    return dj_db, dj_dw



def gradient_descent(X, y, w_in, b_in, alpha, num_iters, compute_gradient_logistic):
    # An array to store cost J and w's at each iteration primarily for graphing later
    w = copy.deepcopy(w_in)  #avoid modifying global w within function
    b = b_in

    for i in range(num_iters):
        # Calculate the gradient and update the parameters
        dj_db, dj_dw = compute_gradient_logistic(X, y, w, b)

        # Update Parameters using w, b, alpha and gradient
        w = w - alpha * dj_dw
        b = b - alpha * dj_db

    return w, b      #return final w,b
```

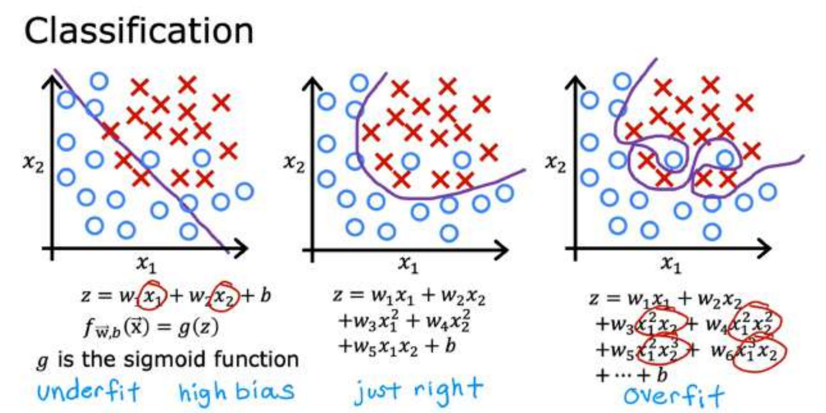
C1_W3_Lab07_Scikit_Learn_Soln

implement GD for logistic regression using sci-kit learn

```
import numpy as np
from sklearn.linear_model import LogisticRegression

X = np.array([[0.5, 1.5], [1,1], [1.5, 0.5], [3, 0.5], [2, 2], [1, 2.5]])
y = np.array([0, 0, 0, 1, 1, 1])

lr_model = LogisticRegression()
lr_model.fit(X, y)
y_pred = lr_model.predict(X)

print("Prediction on training set:", y_pred)
print("Accuracy on training set:", lr_model.score(X, y))
```

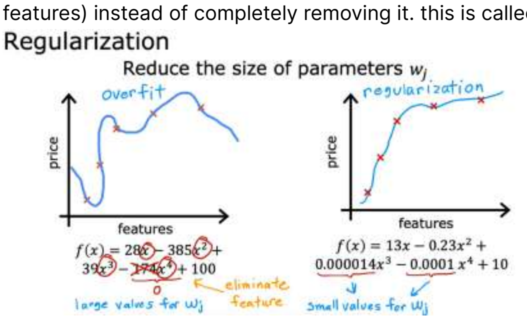
The problem of overfitting



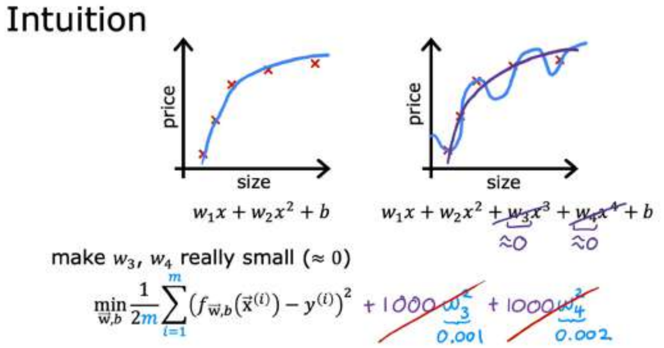


Addressing overfitting

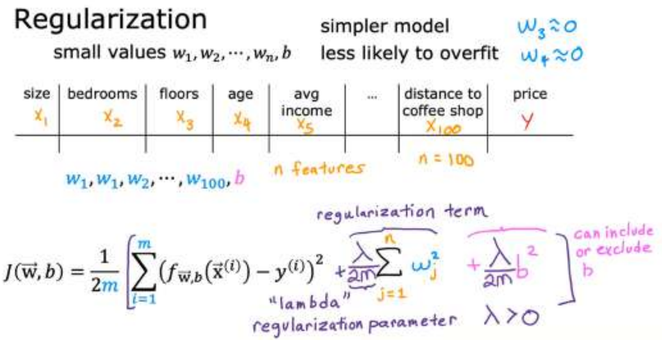
method 1: collect more data, collecting more data would help the line fit data better and not be wiggly around data points ultimately not fitting new data properly
method 2: it may be possible that using all the features in the data set + having less data may be the reason causing overfitting. one solution to this maybe is to use a select few no. of those features and not all of them. the disadvantage to this is that maybe some imp. features be lost. this process is called feature selection.
method 3: Regularization, sometimes we may not want to completely eliminate some features, thus we just reduce it (reduce the impact of that features) instead of completely removing it. this is called Regularization.



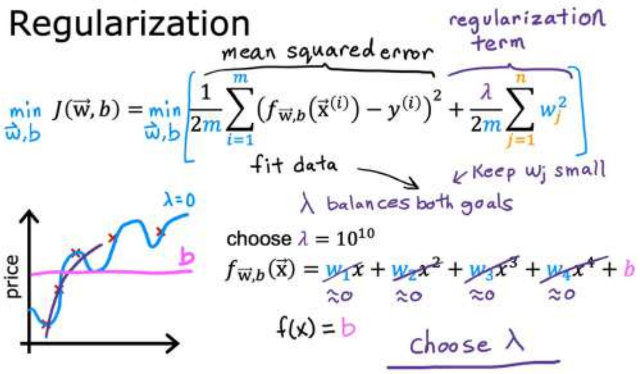
in regularization we tend to regularize w's only and not b. mostly not reqd. to regularize or make b small. in practice it would make very less diff. to make b small.



here if we have to penalize the parameters, we can just add them to the cost fn and assign a large value of coeff to the, just their values would have to be very small to make cost fn small

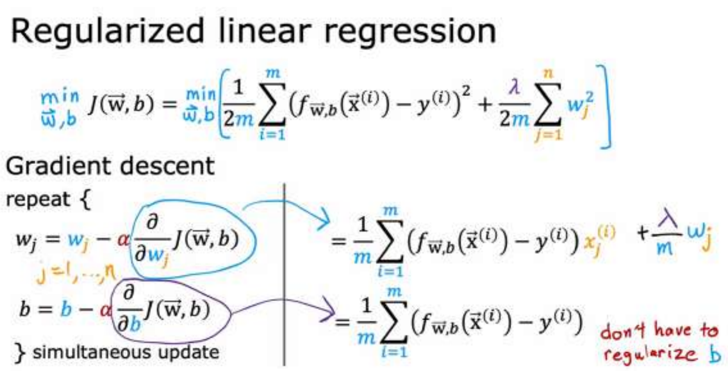


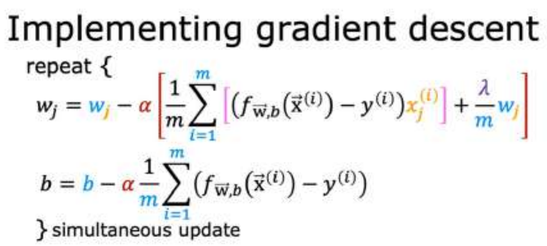
here **λ** is the **regularization parameter**, 2m is scaling the **regularization term**, wj is every parameter, **λ** has to >0 and (λ/2m)*b² can be added or excluded*.



we have to choose regularization parameter in such a way that its not enormous making the weight→0 or make it extremely small giving all the weights a chance to influence the model (outcome). thus an appropriate λ would make the model fit properly

Regularized linear regression





some extra things

Implementing gradient descent

repeat {

$$w_j = w_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\left[\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}\right] + \frac{\lambda}{m}w_j\right]$$

$$b = b - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)$$

} simultaneous update $j = 1 \ldots n$

$$w_j = \underbrace{\underbrace{1w_j - \alpha\frac{\lambda}{m}w_j}_{w_j\left(1-\alpha\frac{\lambda}{m}\right)}}_{\text{shrink } w_j} \underbrace{- \alpha\frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}}_{\text{usual update}}$$

$$\alpha\frac{\lambda}{m}$$

$$0.01\,\frac{1}{50} = 0.0002$$

$$w_j\,(\underbrace{1-0.0002}_{0.9998})$$

for every time wj is updated, the wj is reduced by a very small number which is 1-α λ/*m, thus the wj is decreasing by a very small amount every iteration. hence we see the wj is being updated and also the normal -α* mse is there like in the LR GD. so this is basically the working/ how GD works in regularization

how the derivative term came for the regularization term

How we get the derivative term (optional)

$$\frac{\partial}{\partial w_j}J(\vec{w},b) = \frac{d}{dw_j}\left[\frac{1}{2m}\sum_{i=1}^{m}\left(\underbrace{f(\vec{x}^{(i)})}_{\vec{w}\cdot\vec{x}^{(i)}+b} - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n}w_j^2\right]$$

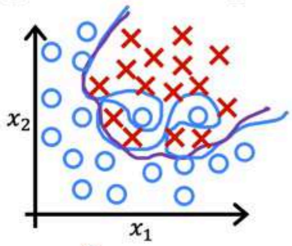
$$= \frac{1}{\cancel{2}m}\sum_{i=1}^{m}\left[\left(\vec{w}\cdot\vec{x}^{(i)} + b - y^{(i)}\right)\cancel{2}x_j^{(i)}\right] + \frac{\lambda}{\cancel{2}m}\cancel{2}w_j \qquad \text{No } \sum_{j=1}^{n}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left[\left(\underbrace{\vec{w}\cdot\vec{x}^{(i)} + b}_{f(\vec{x})} - y^{(i)}\right)x_j^{(i)}\right] + \frac{\lambda}{m}w_j$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left[\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}\right] + \frac{\lambda}{m}w_j$$

Regularized logistic regression

Regularized logistic regression



$$z = w_1x_1 + w_2x_2 + w_3x_1^2x_2 + w_4x_1^2x_2^2 + w_5x_1^2x_2^3 + \cdots + b$$

$$f_{\vec{w},b}(\vec{x}) = \frac{1}{1+e^{-z}}$$

Cost function

$$J(\vec{w},b) = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) + (1-y^{(i)})\log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}w_j^2}_{\min_{\vec{w},b} J(\vec{w},b) \;\rightarrow\; w_j \downarrow}$$

Regularized logistic regression

$$\min_{\vec{w},b} J(\vec{w},b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) + (1-y^{(i)})\log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}w_j^2$$

Gradient descent

Looks same as for linear regression!

repeat {

$$w_j = w_j - \alpha\frac{\partial}{\partial w_j}J(\vec{w},b) \qquad j = 1\ldots n$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left[\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}\right] + \frac{\lambda}{m}w_j$$

logistic regression

$$b = b - \alpha\frac{\partial}{\partial b}J(\vec{w},b)$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)$$

}

you will see that the GD terms are exactly the same for Logistic regression as Linear regression but the only difference here is the meaning of fwb(xi) which here is a sigmoid fn unlike a linear fn in LinearR.

C1_W3_Lab09_Regularization_Soln

```python
import numpy as np
import matplotlib.pyplot as plt

def compute_cost_linear_reg(X, y, w, b, lambda_ = 1):
# cost fn for REGULARISED LINEAR REGRESSION
    m  = X.shape[0]
    n  = len(w)
    cost = 0.
    for i in range(m):
        f_wb_i = np.dot(X[i], w) + b                                   #(n,)(n,)=scalar, see np.dot
        cost = cost + (f_wb_i - y[i])**2                               #scalar
    cost = cost / (2 * m)                                              #scalar

    reg_cost = 0
    for j in range(n):
        reg_cost += (w[j]**2)                                          #scalar
    reg_cost = (lambda_/(2*m)) * reg_cost                              #scalar

    total_cost = cost + reg_cost                                       #scalar
    return total_cost

def compute_cost_logistic_reg(X, y, w, b, lambda_ = 1):
# cost fn for REGULARISED LOGISTIC REGRESSION
    m,n  = X.shape
    cost = 0.
    for i in range(m):
        z_i = np.dot(X[i], w) + b                                      #(n,)(n,)=scalar, see np.dot
        f_wb_i = sigmoid(z_i)                                          #scalar
        cost +=  -y[i]*np.log(f_wb_i) - (1-y[i])*np.log(1-f_wb_i)      #scalar

    cost = cost/m                                                      #scalar

    reg_cost = 0
    for j in range(n):
        reg_cost += (w[j]**2)                                          #scalar
    reg_cost = (lambda_/(2*m)) * reg_cost                              #scalar

    total_cost = cost + reg_cost                                       #scalar
    return total_cost                                                  #scalar


def compute_gradient_linear_reg(X, y, w, b, lambda_):
# ### Gradient function for REGULARISED LINEAR REGRESSION
    m,n = X.shape           #(number of examples, number of features)
    dj_dw = np.zeros((n,))
    dj_db = 0.

    for i in range(m):
        err = (np.dot(X[i], w) + b) - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err * X[i, j]
        dj_db = dj_db + err
    dj_dw = dj_dw / m
    dj_db = dj_db / m

    for j in range(n):
        dj_dw[j] = dj_dw[j] + (lambda_/m) * w[j]

    return dj_db, dj_dw

def compute_gradient_logistic_reg(X, y, w, b, lambda_):
# Gradient function for REGULARISED LOGISTIC REGRESSION
    m,n = X.shape
    dj_dw = np.zeros((n,))                          #(n,)
    dj_db = 0.0                                     #scalar

    for i in range(m):
        f_wb_i = sigmoid(np.dot(X[i],w) + b)        #(n,)(n,)=scalar
        err_i  = f_wb_i  - y[i]                     #scalar
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err_i * X[i,j]    #scalar
        dj_db = dj_db + err_i
    dj_dw = dj_dw/m                                 #(n,)
    dj_db = dj_db/m                                 #scalar

    for j in range(n):
        dj_dw[j] = dj_dw[j] + (lambda_/m) * w[j]
```

```
    return dj_db, dj_dw
```

do read the final lab assignment of week 3, very imp and useful