Classification

Output variable y can take only one out of a handful of possible values.

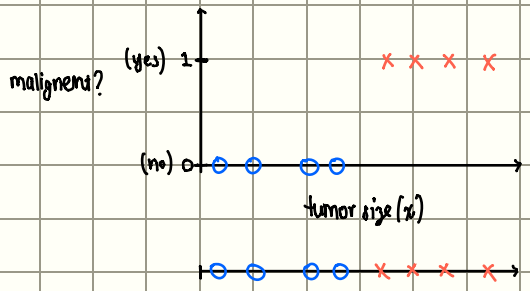
In regression y could take ∞ no of values.

Examples : is email spam?
is transac fraudulent?
is tumor malignent?

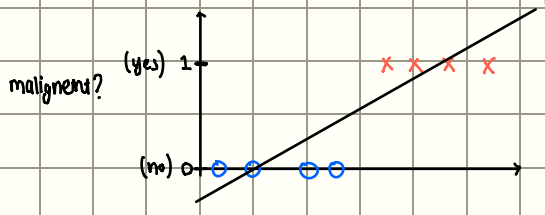
} binary classification → 2 classes / categories → no/yes , false/true , 0/1

negative class = bad = absence

positive class = good = presence



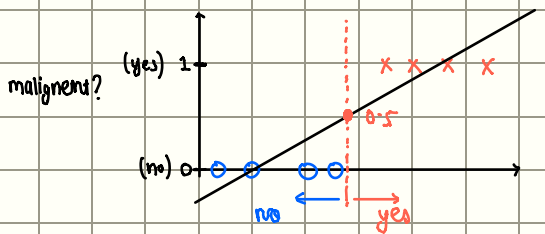
Linear regression not good , Logistical regression better for classification



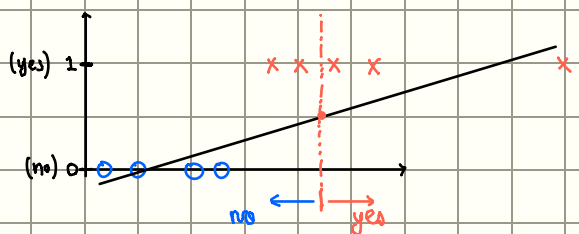
linear regression

- this predicts values other than 0 or 1 as well
- a threshhold like 0.5 could help . If pred <0.5, then not malignent

This could work & makes sense

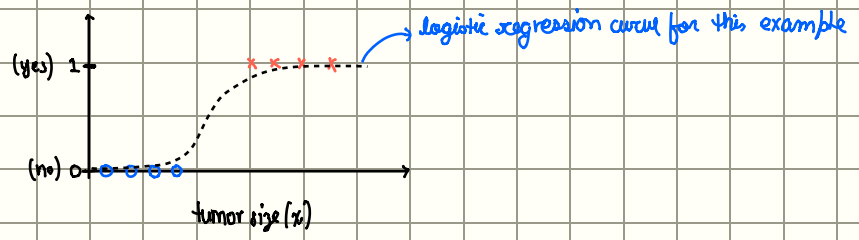


BUT what if one more data point

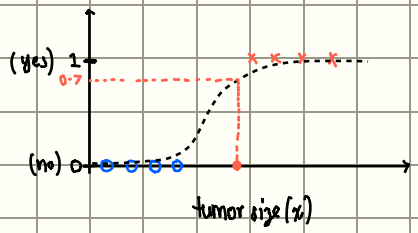


Just adding one example makes our function bad

Logistic Regression

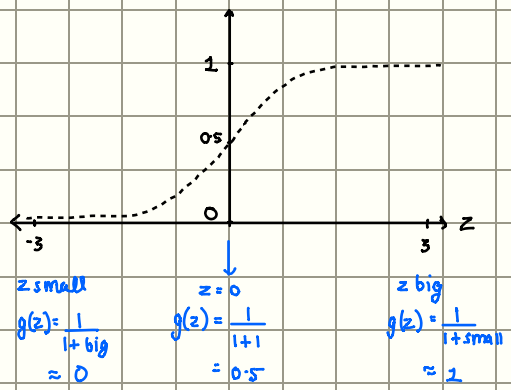


Here, If x = smth gives y = 0·7, then more likely malignent



The Sigmoid / Logistic function

Sigmoid func $g(z) = \frac{1}{1+e^{-z}}$ $\quad 0 < g(z) < 1$



We want to fit this sigmoid function to our data

The logistic regression algorithm

1] Run linear reg $\vec{w} \cdot \vec{x} + b$
Store the function in a variable z
$\Rightarrow z = \vec{w} \cdot \vec{x} + b$

2] Pass value of z to sigmoid function $g(z) = \frac{1}{1+e^{-z}}$
This will be our logistic reg. model

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

i/p: $\vec{x}$
o/p: no. b/w 0 & 1

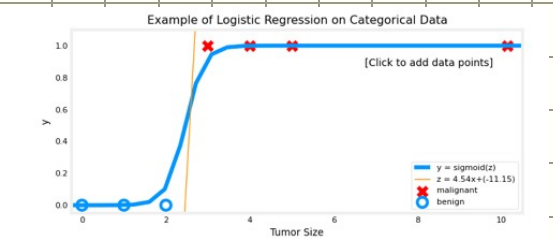
Logistic Regression Model

3] Interpret the output of logistic reg as the probability that the class / category with label $y = 1$ given certain i/p $\vec{x}$

i.e. If x = tumor size
y = 1 means malignant
& $f(x)$ gives 0.7
→ Model predicts 70% chance that y is actually 1

C1 W3 Lab02

```
def sigmoid(z):
    g = 1/(1+np.exp(-z))        # np.exp(x) makes each element e^x
    return g
```



orange line doesn't match linear regression line, why?!

because we classify first via linear reg & convert it to logistic reg curve.

Rather, as we keep plotting linear reg, we analyze cost of sigmoid func rather than $z = wx + b$

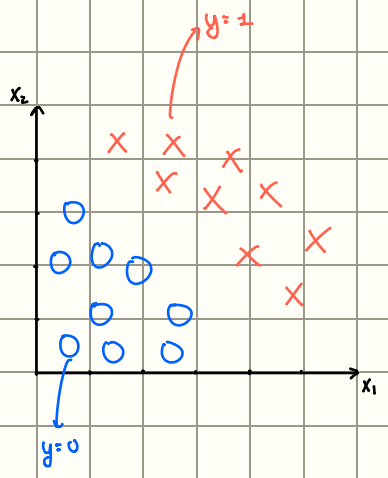
The decision boundary

The function gives us values b/w 0 & 1
How do we get either 0 or 1?

One solution is to set a threshold on value of $f(x)$ or $\hat{y}$
→ If $\hat{y}$ > threshold → $\hat{y} = 1$
< threshold → $\hat{y} = 0$

Eg: let threshold = 0.5
When is $f_{\vec{w},b}(\vec{x}) \geq 0.5$?
→ when $g(z) \geq 0.5$
→ when $z \geq 0$ (from sigmoid func graph)
→ when $wx+b \geq 0$

→ model predicts $\hat{y} = 1$ when $wx+b \geq 0$
$\hat{y} = 0$ when $wx+b \leq 0$



This example has 2 features x_1 & x_2

$$f_{\vec{w},b}(\vec{x}) = g(z) = g(w_1x_1 + w_2x_2 + b)$$

assume: $w_1 = 1$, $w_2 = 1$, $b = -3$

now we want to see when $w \cdot x + b \geq 0$ & $w \cdot x + b < 0$ in order to understand what model predicts

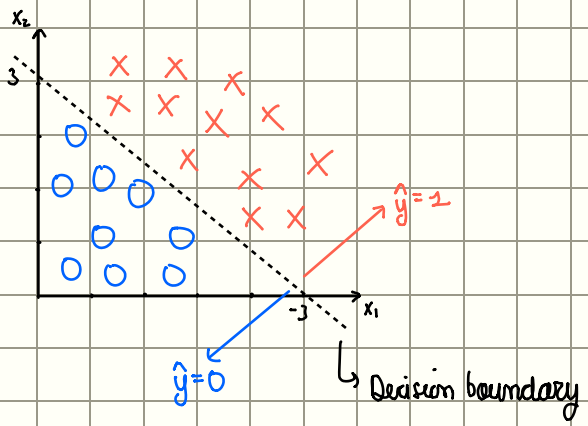
This is seperated by the line $w \cdot x + b = 0$

This line happens to be the decision boundary

⇒ Decision boundary ⇒ $z = \vec{w} \cdot \vec{x} + b = 0$

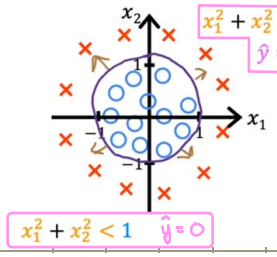
In this example decision boundary : $w_1x_1 + w_2x_2 + b = 0$

$x_1 + x_2 - 3 = 0$



Another example

Non-linear decision boundaries



$x_1^2 + x_2^2 \geq 1$ $\hat{y} = 1$

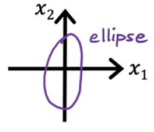
$x_1^2 + x_2^2 < 1$ $\hat{y} = 0$

$$f_{\vec{w},b}(\vec{x}) = g(z) = g(\overbrace{w_1x_1^2 + w_2x_2^2 + b}^{z})$$

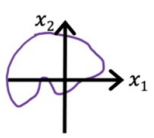
$w_1 = 1$, $w_2 = 1$, $b = -1$

decision boundary $z = x_1^2 + x_2^2 - 1 = 0$

$x_1^2 + x_2^2 = 1$



$$f_{\vec{w},b}(\vec{x}) = g(z) = g(w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_1x_2 + w_5x_2^2 + w_6x_1^3 + \cdots + b)$$



Cost function for logistic regression

Our training set

	tumor size x_1	.....	patient's age x_n	y (→ either 0 or 1)
$i=1$	10		52	1
⋮	2		73	0
⋮	5		55	0
$i=m$	12		49	1

Our model

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

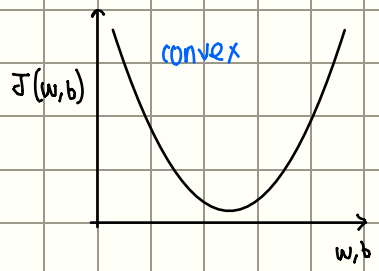
How do we choose parameters w & b ?

We need to come up w a cost func to tell if chosen w & b good or not

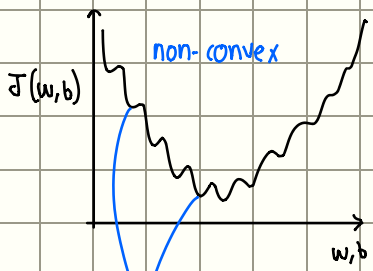
Considering cost func chosen for linear reg: $J(\vec{w},b) = \frac{1}{2m}\sum_{i=1}^{m}\left(f(x^i)-y^i\right)^2$

Applying this cost func in

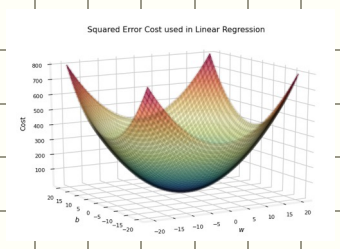
1] linear reg : $f(x)=wx+b$

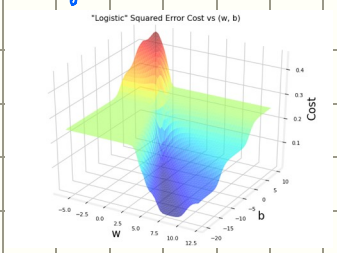


2] logistic reg : $f(x)=\frac{1}{1+e^{-(wx+b)}}$



lots of local minimas to get stuck in





∴ for linear reg, squared error cost function doesn't work well

We need to get a convex cost function to be able to GD on it

Building a new cost function

let loss on a single training example be $L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right)$

↑ Loss is a function of prediction at a point & actual value
Loss over all points will give total loss

For logistic reg :

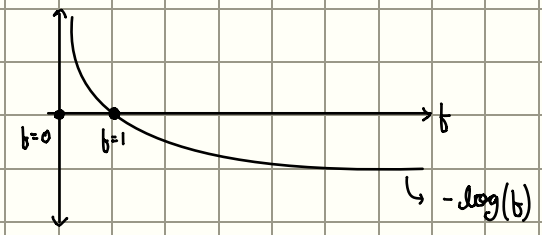
$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right) = \begin{cases} -\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 1 \\ -\log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 0 \end{cases}$$

This peculiar cost function has been derived using Maximum Likelihood Estimate (MLE)

Why does it make sense?

https://youtu.be/XepXtl9YKwc?si=WnubVeCYbBcYTRLN

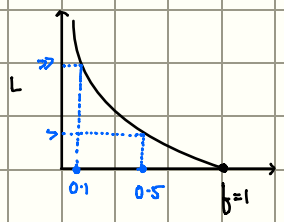
Consider $y = 1$



∵ f = output of logistic reg

∴ $0 \le f \le 1$

⇒ only some part relevent

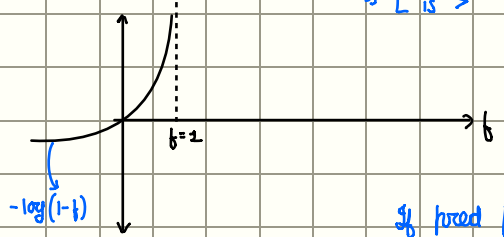


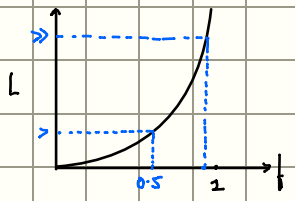
If pred $f = 1$
& true $y = 1$
⇒ $L = 0$

If pred $= 0.5$
& true $y = 1$
⇒ L is >

If pred $= 0.1$
& true $y = 1$
⇒ L is >>

Consider $y = 0$





If pred $f = 0$
& true $y = 0$
⇒ $L = 0$

If pred $= 0.9$
& true $y = 0$
⇒ L is ≫

If pred $= 0.5$
& true $y = 0$
⇒ L is >

This new loss func makes cost func convex

The loss function can be written like

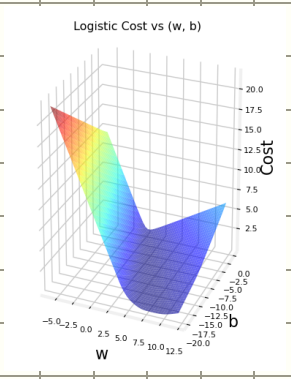
$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right) = \left[-y^{(i)} \log\left(f(\vec{x}^{(i)})\right)\right] + \left[-(1-y^{(i)}) \log\left(1 - f(\vec{x}^{(i)})\right)\right]$$

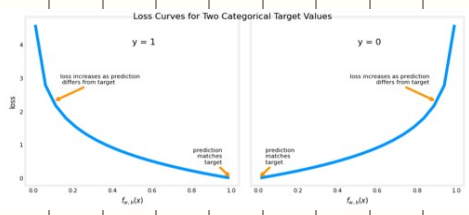
This form is easier to implement

The cost function is average / normalisation of all losses)

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

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





Finally can be written as

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

This cost function is convex

Now we can use the cost function to determine at which w & b it is min
& ∴ find good w & b for our logistic reg. model

Gradient descent on logistic cost func

Find $\vec{w}, b$ so that given new $\vec{x}$, model can make a good prediction of the probability that the label y is 1

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

↓

Gradient descent repeat {

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

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

}

#simultaneous updates

↓

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

Similar to linear regression, we can
- test for convergence
- vectorize for optimised code
- Do feature scaling

C1 W3 Lab 06 : GD for logistic reg

Logistic Gradient Descent

Recall the gradient descent algorithm utilizes the gradient calculation:

$$\begin{aligned} &\text{repeat until convergence: } \{ \\ &\quad w_j = w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \qquad \text{for j := 0..n-1} \qquad (1) \\ &\quad b = b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \\ &\} \end{aligned}$$

Where each iteration performs simultaneous updates on w_j for all j, where

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)} \qquad (2)$$

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) \qquad (3)$$

- m is the number of training examples in the data set
- $f_{\mathbf{w},b}(x^{(i)})$ is the model's prediction, while $y^{(i)}$ is the target
- For a logistic regression model
  $z = \mathbf{w} \cdot \mathbf{x} + b$
  $f_{\mathbf{w},b}(x) = g(z)$
  where $g(z)$ is the sigmoid function:
  $g(z) = \frac{1}{1+e^{-z}}$

Gradient Descent Implementation

The gradient descent algorithm implementation has two components:

- The loop implementing equation (1) above. This is `gradient_descent` below and is generally provided to you in optional and practice labs.
- The calculation of the current gradient, equations (2,3) above. This is `compute_gradient_logistic` below. You will be asked to implement this week's practice lab.

Calculating the Gradient, Code Description

Implements equation (2),(3) above for all w_j and b. There are many ways to implement this. Outlined below is this:

- initialize variables to accumulate `dj_dw` and `dj_db`
- for each example
  - calculate the error for that example $g(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) - \mathbf{y}^{(i)}$
  - for each input value $x_j^{(i)}$ in this example,
    - multiply the error by the input $x_j^{(i)}$, and add to the corresponding element of `dj_dw`. (equation 2 above)
  - add the error to `dj_db` (equation 3 above)
- divide `dj_db` and `dj_dw` by total number of examples (m)
- note that $\mathbf{x}^{(i)}$ in numpy `X[i,:]` or `X[i]` and $x_j^{(i)}$ is `X[i,j]`

Optional Lab: Gradient Descent for Logistic Regression

Goals

In this lab, you will:

- update gradient descent for logistic regression.
- explore gradient descent on a familiar data set

In [ ]:
```
import copy, math
import numpy as np
%matplotlib widget
import matplotlib.pyplot as plt
from lab_utils_common import  dlc, plot_data, plt_tumor_data, sigm
oid, compute_cost_logistic
from plt_quad_logistic import plt_quad_logistic, plt_prob
plt.style.use('./deeplearning.mplstyle')
```

Data set

Let's start with the same two feature data set used in the decision boundary lab.

In [ ]:
```
X_train = np.array([[0.5, 1.5], [1,1], [1.5, 0.5], [3, 0.5], [2,
2], [1, 2.5]])
y_train = np.array([0, 0, 0, 1, 1, 1])
```

As before, we'll use a helper function to plot this data. The data points with label $y = 1$ are shown as red crosses, while the data points with label $y = 0$ are shown as blue circles.

In [ ]:
```
fig,ax = plt.subplots(1,1,figsize=(4,4))
plot_data(X_train, y_train, ax)

ax.axis([0, 4, 0, 3.5])
ax.set_ylabel('$x_1$', fontsize=12)
ax.set_xlabel('$x_0$', fontsize=12)
plt.show()
```

In [ ]:
```
def compute_gradient_logistic(X, y, w, b):
    """
    Computes the gradient for logistic regression

    Args:
      X (ndarray (m,n): Data, m examples with n features
      y (ndarray (m,)): target values
      w (ndarray (n,)): model parameters
      b (scalar)      : model parameter
    Returns
      dj_dw (ndarray (n,)): The gradient of the cost w.r.t. the par
ameters w.
      dj_db (scalar)      : The gradient of the cost w.r.t. the par
ameter 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,)=sc
alar
        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
```

Check the implementation of the gradient function using the cell below.

In [ ]:
```
X_tmp = np.array([[0.5, 1.5], [1,1], [1.5, 0.5], [3, 0.5], [2, 2],
[1, 2.5]])
y_tmp = np.array([0, 0, 0, 1, 1, 1])
w_tmp = np.array([2.,3.])
b_tmp = 1.
dj_db_tmp, dj_dw_tmp = compute_gradient_logistic(X_tmp, y_tmp, w_t
mp, b_tmp)
print(f"dj_db: {dj_db_tmp}" )
print(f"dj_dw: {dj_dw_tmp.tolist()}" )
```

Expected output

```
dj_db: 0.49861806546328574
dj_dw: [0.498333393278696, 0.4988394298396693]
```

Gradient Descent Code

The code implementing equation (1) above is implemented below. Take a moment to locate and compare the functions in the routine to the equations above.

In [ ]:
```
def gradient_descent(X, y, w_in, b_in, alpha, num_iters):
    """
    Performs batch gradient descent

    Args:
      X (ndarray (m,n)   : Data, m examples with n features
      y (ndarray (m,))   : target values
      w_in (ndarray (n,)): Initial values of model parameters
      b_in (scalar)      : Initial values of model parameter
      alpha (float)      : Learning rate
      num_iters (scalar) : number of iterations to run gradient des
cent

    Returns:
      w (ndarray (n,))   : Updated values of parameters
      b (scalar)         : Updated value of parameter
    """
    # An array to store cost J and w's at each iteration primarily
for graphing later
    J_history = []
    w = copy.deepcopy(w_in)  #avoid modifying global w within funct
ion
    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

        # Save cost J at each iteration
        if i<100000:      # prevent resource exhaustion
            J_history.append( compute_cost_logistic(X, y, w, b) )

        # Print cost every at intervals 10 times or as many iterati
ons if < 10
        if i% math.ceil(num_iters / 10) == 0:
            print(f"Iteration {i:4d}: Cost {J_history[-1]}   ")

    return w, b, J_history         #return final w,b and J history
for graphing
```

Let's run gradient descent on our data set.

In [ ]:
```
w_tmp  = np.zeros_like(X_train[0])
b_tmp  = 0.
alph = 0.1
iters = 10000

w_out, b_out, _ = gradient_descent(X_train, y_train, w_tmp, b_tmp,
alph, iters)
print(f"\nupdated parameters: w:{w_out}, b:{b_out}")
```

Let's plot the results of gradient descent:

In [ ]:
```
fig,ax = plt.subplots(1,1,figsize=(5,4))
# plot the probability
plt_prob(ax, w_out, b_out)

# Plot the original data
ax.set_ylabel(r'$x_1$')
ax.set_xlabel(r'$x_0$')
ax.axis([0, 4, 0, 3.5])
plot_data(X_train,y_train,ax)

# Plot the decision boundary
x0 = -b_out/w_out[0]
x1 = -b_out/w_out[1]
ax.plot([0,x0],[x1,0], c=dlc["dlblue"], lw=1)
plt.show()
```

In the plot above:

- the shading reflects the probability y=1 (result prior to decision boundary)
- the decision boundary is the line at which the probability = 0.5

Another Data set

Let's return to a one-variable data set. With just two parameters, w, b, it is possible to plot the cost function using a contour plot to get a better idea of what gradient descent is up to.

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

As before, we'll use a helper function to plot this data. The data points with label $y = 1$ are shown as red crosses, while the data points with label $y = 0$ are shown as blue circles.

In [ ]:
```
fig,ax = plt.subplots(1,1,figsize=(4,3))
plt_tumor_data(x_train, y_train, ax)
plt.show()
```

In the plot below, try:

- changing w and b by clicking within the contour plot on the upper right.
  - changes may take a second or two
  - note the changing value of cost on the upper left plot.
  - note the cost is accumulated by a loss on each example (vertical dotted lines)
- run gradient descent by clicking the orange button.
  - note the steadily decreasing cost (contour and cost plot are in log(cost)
  - clicking in the contour plot will reset the model for a new run
- to reset the plot, rerun the cell

In [ ]:
```
w_range = np.array([-1, 7])
b_range = np.array([1, -14])
quad = plt_quad_logistic( x_train, y_train, w_range, b_range )
```

Congratulations!

You have:

- examined the formulas and implementation of calculating the gradient for logistic regression
- utilized those routines in
  - exploring a single variable data set
  - exploring a two-variable data set

In [ ]:
```

```

CI W3 Lab 07 : Scikit learn Logistic Regression

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

X = np.array([...])
y = np.array([...])

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

y_predic = lr_model.predict(X)
```

number or array

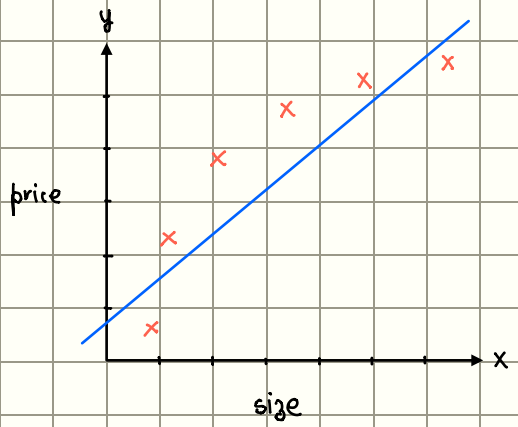
↳ output 0 or 1

```
print(lr_model.score(X,y))
```

prints accuracy of this model on the training set

Overfitting & Underfitting

Lets understand by housing example



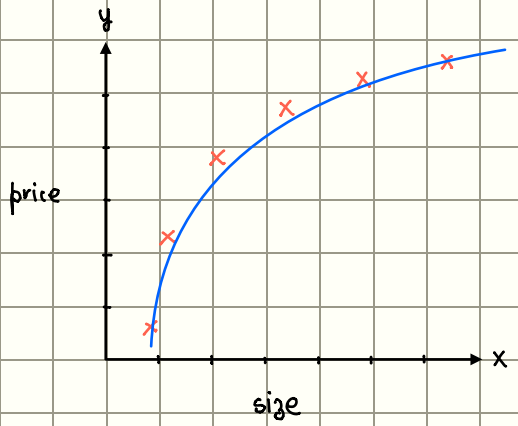
Model : $w_1 x + b$

When we perform linear reg & fit a straight line.
This algo doesn't fit the training data well

Underfitting

∵ Our model doesn't fit the training data well
We say the model is underfitting the training data
Or the algorithm has high bias → this doesn't mean model is biased towards some opinion, currently it means the model has a bias / preconception that the prices grow linearly despite data suggesting otherwise.

∴ wrong preconception / bias = underfitting



Model : $w_1 x + w_2 x^2 + b$

When we try fitting a quadratic function w 2 features x & x^2, then it fits better

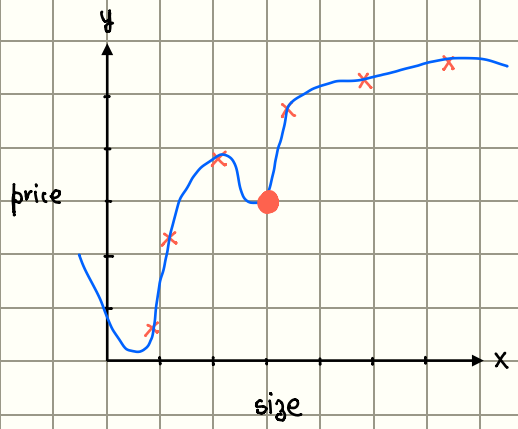
Generalized Model

Just right

This model would do well on predicting prices for even input x not in training data

Generalization : learning algorithms should do well even on values not in training data

This model fits data not perfectly but well enough that it would generalize nicely to new data.



Model : $w_1x + w_2x^2 + w_3x^3 + w_4x^4 + b$

This model fits the data too well.
It passes thru all training examples, might give cost even 0

Overfitting

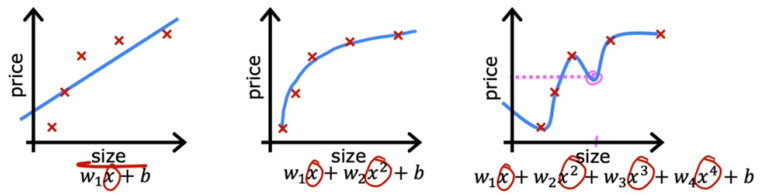
Algo is trying too hard to fit every example

But the model curve gets all over the place, it can ∴ fit training data very well but will not generalize to new data even within training data range.
Eg: the red dot, size ↑ but price ↓, doesn't make sense!

The model has overfit the data.
The algorithm has high variance

We say high variance ∵ even if training set was just a bit different, then the function/curve obtained could be totally different.
∴ highly variable predictions could be produced

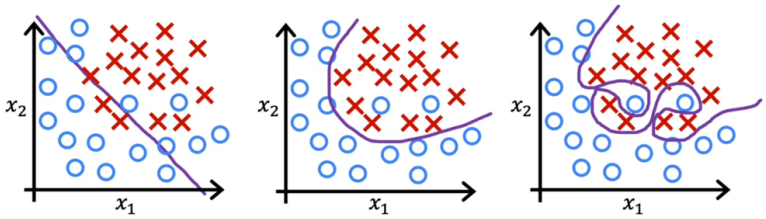
Regression example



underfit		overfit
• Does not fit the training set well	• Fits training set pretty well	• Fits the training set extremely well
high bias	generalization	high variance

∴ Find just the right number of features to train the model

Classification example



$z = w_1x_1 + w_2x_2 + b$
$f_{\vec{w},b}(\vec{x}) = g(z)$
g is the sigmoid function

underfit high bias

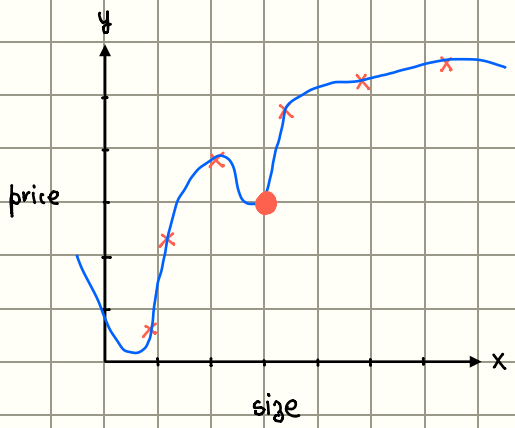
$z = w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_2^2 + w_5x_1x_2 + b$

just right

$z = w_1x_1 + w_2x_2 + w_3x_1^2x_2 + w_4x_1^2x_2^2 + w_5x_1^2x_2^3 + w_6x_1x_2 + \cdots + b$

Overfit

Addressing overfitting

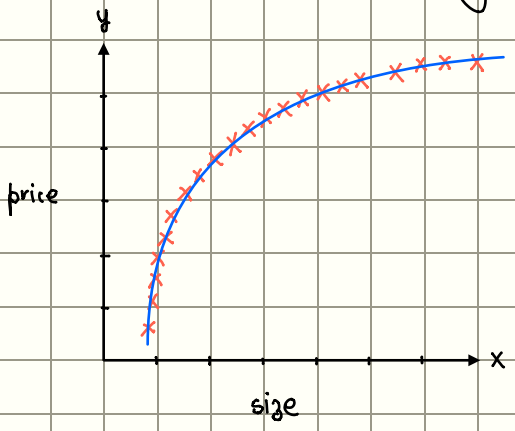


This is our overfit model

Model : $w_1x + w_2x^2 + w_3x^3 + w_4x^4 + b$

Fix

Option 1 : give more training examples



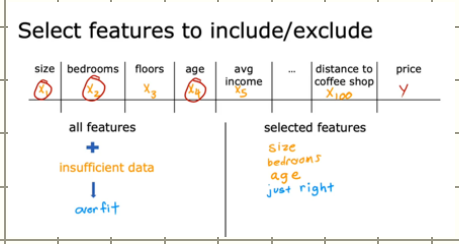
More examples will help make line less wiggly

Option 2 : try to use fewer features

Reduce no. of polynomial features.

Another example is if we have a lot of actual features, but not enough training data, then too, overfitting can occur.

→ In this case, try to choose the most revelent features instead of all.

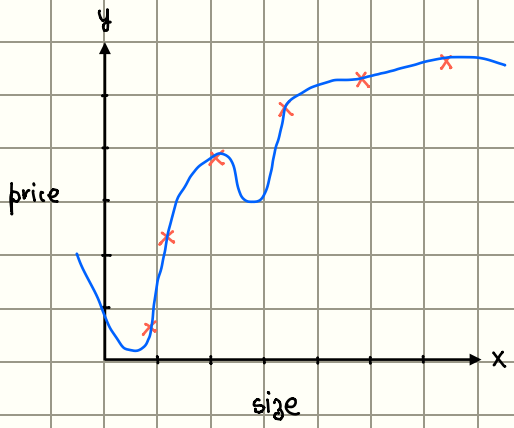


Feature selection : use only a subset of features

It does throw away some information

: i.e. eliminate some features by setting its param $w_j = 0$

Option 3 : Regularization



Say our model is

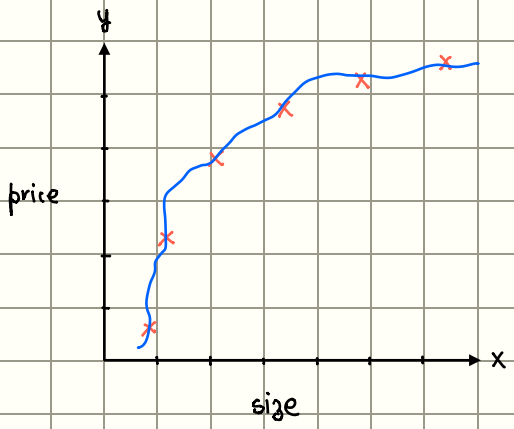
$f(x) = 28x - 385x^2 + 39x^3 - 174x^4 + 100$

Instead of eliminating features by setting $w_j = 0$, Regularization is a way to more gently reduce the impact of some of the features w/o harshly eliminating

→ Regularization encourages learning algo to shrink values of some parameters instead of setting them to 0

Regularization lets us keep all the features but prevents features from having overly large effects which may cause overfitting.

Only w_j are regularized, b is let be, it doesn't matter much in practice



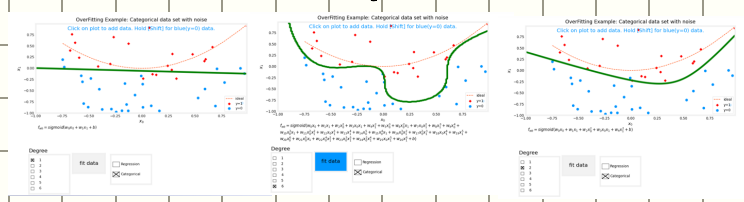
Regularized model

$f(x) = \underbrace{13x - 0.23x^2 + 0.00014x^3 - 0.00001x^4}_{\text{regularized parameters}} + 10$

Examples: Regression

Underfit Overfit Just right

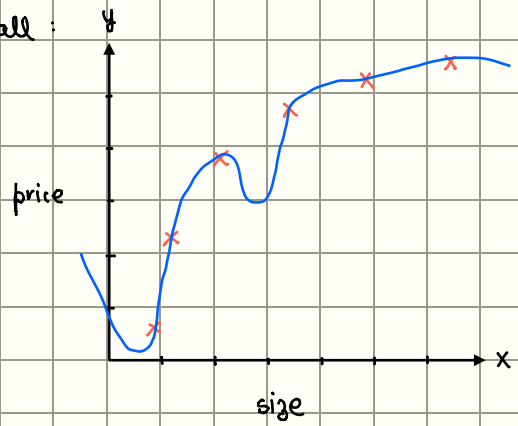
Classification



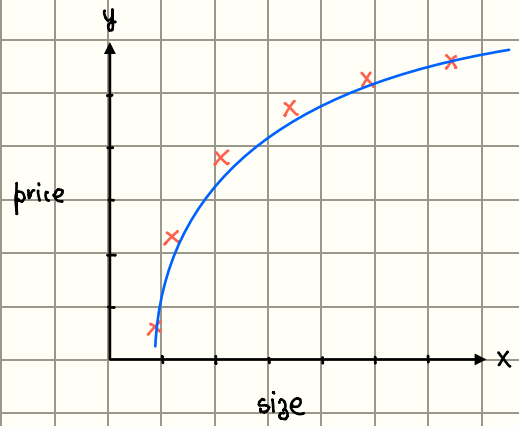
Underfit Overfit Just right

How to regularize?

Recall :



Model : $w_1x + w_2x^2 + w_3x^3 + w_4x^4 + b$



Model : $w_1x + w_2x^2 + b$

To regularize, we need to make w_3, w_4 very small (≈ 0)

How would we do that?

↳ Imagine a new cost function to minimize

$$\Rightarrow \min \frac{1}{2m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(x^{(i)}) - y^{(i)}\right)^2 + 1000\, w_3^2 + 1000\, w_4^2$$

old normal cost func — added these

Here ∵ we want to minimize the entire expression
We have taken big coefficients of w_3 & w_4
∴ when minimizing, value of w_3 & w_4 will have to be very small.

⇒ By adding these, we are penalizing the model if w_3 & w_4 are large.

⇒ we get almost quadratic curve with little contributions from x^3 & x^4 terms

More generally, we do not know out of many features, which are the most important & which ones to penalize.

∴ we penalize all the w_j parameters

⇒ This does give us a simpler & less wiggly, less prone to overfitting function

The regularized cost function

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

λ (lambda) = regularization parameter

$\lambda > 0$

Also divide λ by $2m$ so that both terms are scaled by $1/2m$.

: When both terms are scaled the same way, it becomes easier to choose a good value of λ.

: This helps even when training set size grows. ⇒ If $m \uparrow$, we don't have to update our λ value

∴ we minimize our new $J(\vec{w}, b)$ during GD

⇒ First term minimization encourages algo to fit data well by minimizing errors

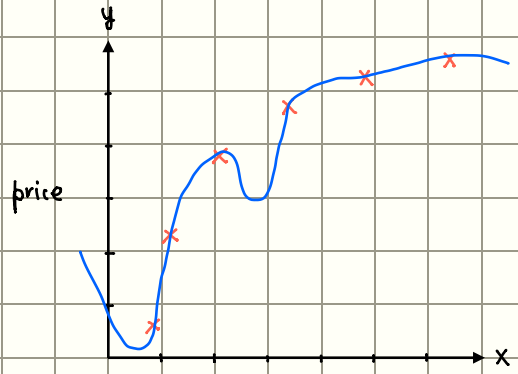
⇒ second term minimization encourages smaller values of w_j to prevent overfitting

Effect of λ

If $\lambda = 0$ no regularization

⇓

overfit

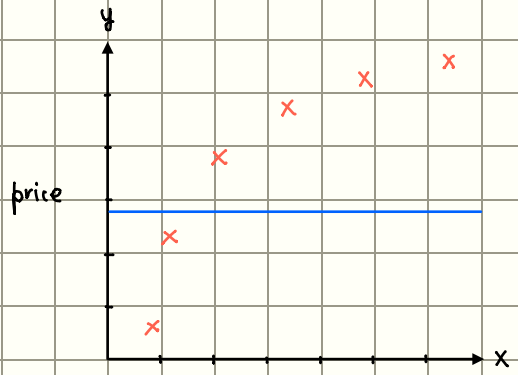


Model : $w_1x + w_2x^2 + w_3x^3 + w_4x^4 + b$

If $\lambda = 10^{10}$ heavy regularization

⇓

underfit



Model : $w_1x + w_2x^2 + w_3x^3 + w_4x^4 + b$

$\approx 0 \quad 0 \quad 0 \quad 0$

⇒ Model $f(x) = b$

We need to find just the right λ

1] Regularized linear regression

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

penalizing large w_j by increasing cost.

Run GD : repeat {

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

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

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

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

} update simul

Mathematical intuition for regularization

We have in GD : $w_j = w_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right) x_j^{(i)} + \frac{\lambda}{m} w_j \right]$, rearranging these terms

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

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

This term is the usual GD update for unregularized linear regression

This is the new change after regularization

Here : $\alpha \frac{\lambda}{m}$: α = v small , λ = small (usually 1-10)

$\Rightarrow \alpha \frac{\lambda}{m} \Rightarrow 0.01 \times \frac{1}{50} = 0.0002$

$\Rightarrow$ value of $\alpha \frac{\lambda}{m} \Rightarrow$ small

We have : $w_j \left(1 - \alpha \frac{\lambda}{m} \right) = 0.9998 \, w_j$

$\hookrightarrow 0.9998$

$\therefore$ in total we have $0.9998 \, w_j$ - usual update instead of $w_j = w_j$ - usual update

This new term gives the shrinking effect on all w_j
& more λ will mean more shrinking

2] Regularized logistic regression

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

penalizing large w_j by increasing cost.

Run GD : repeat {

$$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b) \quad \text{for } j = 1 \dots n$$

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

} update simul

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

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

same as for linear reg.

except for $f(x^{(i)}) = \frac{1}{1+e^{-(\vec{w}\vec{x}+b)}}$

C1 W3 lab 09 : regularized cost & descent

Optional Lab - Regularized Cost and Gradient

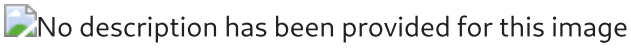
Goals

In this lab, you will:

- extend the previous linear and logistic cost functions with a regularization term.
- rerun the previous example of over-fitting with a regularization term added.

In [1]:
```
import numpy as np
%matplotlib widget
import matplotlib.pyplot as plt
from plt_overfit import overfit_example, output
from lab_utils_common import sigmoid
np.set_printoptions(precision=8)
```

Adding regularization



The slides above show the cost and gradient functions for both linear and logistic regression. Note:

- Cost
  - The cost functions differ significantly between linear and logistic regression, but adding regularization to the equations is the same.
- Gradient
  - The gradient functions for linear and logistic regression are very similar. They differ only in the implementation of f_{wb}.

Cost functions with regularization

Cost function for regularized linear regression

The equation for the cost function regularized linear regression is:

$$J(\mathbf{w},b) = \frac{1}{2m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=0}^{n-1} w_j^2 \tag{1}$$

where:

$$f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = \mathbf{w} \cdot \mathbf{x}^{(i)} + b \tag{2}$$

Compare this to the cost function without regularization (which you implemented in a previous lab), which is of the form:

$$J(\mathbf{w},b) = \frac{1}{2m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})^2$$

The difference is the regularization term, $\frac{\lambda}{2m}\sum_{j=0}^{n-1} w_j^2$

Including this term encourages gradient descent to minimize the size of the parameters. Note, in this example, the parameter b is not regularized. This is standard practice.

Below is an implementation of equations (1) and (2). Note that this uses a *standard pattern for this course*, a `for loop` over all `m` examples.

In [2]:
```
def compute_cost_linear_reg(X, y, w, b, lambda_ = 1):
    """
    Computes the cost over all examples
    Args:
      X (ndarray (m,n): Data, m examples with n features
      y (ndarray (m,)): target values
      w (ndarray (n,)): model parameters
      b (scalar)      : model parameter
      lambda_ (scalar): Controls amount of regularization
    Returns:
      total_cost (scalar):  cost
    """

    m  = X.shape[0]
    n  = len(w)
    cost = 0.
    for i in range(m):
        f_wb_i = np.dot(X[i], w) + b                                    #(
        cost = cost + (f_wb_i - y[i])**2                                #s
    cost = cost / (2 * m)                                               #s

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

    total_cost = cost + reg_cost                                        #s
    return total_cost                                                   #s
```

Run the cell below to see it in action.

In [3]:
```
np.random.seed(1)
X_tmp = np.random.rand(5,6)
y_tmp = np.array([0,1,0,1,0])
w_tmp = np.random.rand(X_tmp.shape[1]).reshape(-1,)-0.5
b_tmp = 0.5
lambda_tmp = 0.7
cost_tmp = compute_cost_linear_reg(X_tmp, y_tmp, w_tmp, b_tmp, lambda_tmp

print("Regularized cost:", cost_tmp)
```

```
Regularized cost: 0.07917239320214277
```

Expected Output:

Regularized cost: 0.07917239320214275

Cost function for regularized logistic regression

For regularized **logistic** regression, the cost function is of the form

$$J(\mathbf{w},b) = \frac{1}{m}\sum_{i=0}^{m-1}\left[-y^{(i)}\log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right)\right] + \frac{\lambda}{2m}$$

where:

$$f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = sigmoid(\mathbf{w}\cdot\mathbf{x}^{(i)} + b) \tag{4}$$

Compare this to the cost function without regularization (which you implemented in a previous lab):

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

As was the case in linear regression above, the difference is the regularization term, which is $\frac{\lambda}{2m}\sum_{j=0}^{n-1} w_j^2$

Including this term encourages gradient descent to minimize the size of the parameters. Note, in this example, the parameter b is not regularized. This is standard practice.

In [4]:
```
def compute_cost_logistic_reg(X, y, w, b, lambda_ = 1):
    """
    Computes the cost over all examples
    Args:
    Args:
      X (ndarray (m,n): Data, m examples with n features
      y (ndarray (m,)): target values
      w (ndarray (n,)): model parameters
      b (scalar)      : model parameter
      lambda_ (scalar): Controls amount of regularization
    Returns:
      total_cost (scalar):  cost
    """

    m,n  = X.shape
    cost = 0.
    for i in range(m):
        z_i = np.dot(X[i], w) + b                                       #(
        f_wb_i = sigmoid(z_i)                                           #s
        cost +=  -y[i]*np.log(f_wb_i) - (1-y[i])*np.log(1-f_wb_i)       #s

    cost = cost/m                                                       #s

    reg_cost = 0
    for j in range(n):
```

```
        reg_cost += (w[j]**2)                                                #s
    reg_cost = (lambda_/(2*m)) * reg_cost                                    #s

    total_cost = cost + reg_cost                                             #s
    return total_cost                                                        #s
```

Run the cell below to see it in action.

In [5]:
```
np.random.seed(1)
X_tmp = np.random.rand(5,6)
y_tmp = np.array([0,1,0,1,0])
w_tmp = np.random.rand(X_tmp.shape[1]).reshape(-1,)-0.5
b_tmp = 0.5
lambda_tmp = 0.7
cost_tmp = compute_cost_logistic_reg(X_tmp, y_tmp, w_tmp, b_tmp, lambda_t

print("Regularized cost:", cost_tmp)
```

```
Regularized cost: 0.6850849138741673
```

Expected Output:

Regularized cost: 0.6850849138741673

Gradient descent with regularization

The basic algorithm for running gradient descent does not change with regularization, it is:

$$\begin{aligned}
&\text{repeat until convergence: } \{ \\
&\quad w_j = w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \qquad \text{for j := 0..n-1} \\
&\quad b = b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \\
&\}
\end{aligned} \tag{1}$$

Where each iteration performs simultaneous updates on w_j for all j.

What changes with regularization is computing the gradients.

Computing the Gradient with regularization (both linear/logistic)

The gradient calculation for both linear and logistic regression are nearly identical, differing only in computation of $f_{\mathbf{w}b}$.

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)} + \frac{\lambda}{m} w_j \tag{2}$$

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) \tag{3}$$

- m is the number of training examples in the data set

- $f_{\mathbf{w},b}(x^{(i)})$ is the model's prediction, while $y^{(i)}$ is the target
- For a **linear** regression model
  $f_{\mathbf{w},b}(x) = \mathbf{w} \cdot \mathbf{x} + b$
- For a **logistic** regression model
  $z = \mathbf{w} \cdot \mathbf{x} + b$
  $f_{\mathbf{w},b}(x) = g(z)$
  where $g(z)$ is the sigmoid function:
  $g(z) = \frac{1}{1+e^{-z}}$

The term which adds regularization is the $\frac{\lambda}{m} w_j$.

Gradient function for regularized linear regression

In [6]:
```
def compute_gradient_linear_reg(X, y, w, b, lambda_):
    """
    Computes the gradient for linear regression
    Args:
      X (ndarray (m,n): Data, m examples with n features
      y (ndarray (m,)): target values
      w (ndarray (n,)): model parameters
      b (scalar)      : model parameter
      lambda_ (scalar): Controls amount of regularization

    Returns:
      dj_dw (ndarray (n,)): The gradient of the cost w.r.t. the parameter
      dj_db (scalar):       The gradient of the cost w.r.t. the parameter
    """
    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
```

Run the cell below to see it in action.

In [7]:
```
np.random.seed(1)
X_tmp = np.random.rand(5,3)
y_tmp = np.array([0,1,0,1,0])
w_tmp = np.random.rand(X_tmp.shape[1])
b_tmp = 0.5
lambda_tmp = 0.7
dj_db_tmp, dj_dw_tmp =  compute_gradient_linear_reg(X_tmp, y_tmp, w_tmp,
```

```
print(f"dj_db: {dj_db_tmp}", )
print(f"Regularized dj_dw:\n {dj_dw_tmp.tolist()}", )
```

```
dj_db: 0.6648774569425726
Regularized dj_dw:
 [0.29653214748822276, 0.4911679625918033, 0.21645877535865857]
```

Expected Output

```
dj_db: 0.6648774569425726
Regularized dj_dw:
 [0.29653214748822276, 0.4911679625918033,
0.21645877535865857]
```

Gradient function for regularized logistic regression

In [8]:
```
def compute_gradient_logistic_reg(X, y, w, b, lambda_):
    """
    Computes the gradient for linear regression

    Args:
      X (ndarray (m,n): Data, m examples with n features
      y (ndarray (m,)): target values
      w (ndarray (n,)): model parameters
      b (scalar)      : model parameter
      lambda_ (scalar): Controls amount of regularization
    Returns
      dj_dw (ndarray Shape (n,)): The gradient of the cost w.r.t. the par
      dj_db (scalar)            : The gradient of the cost w.r.t. the par
    """
    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
```

Run the cell below to see it in action.

In [9]:
```
np.random.seed(1)
X_tmp = np.random.rand(5,3)
y_tmp = np.array([0,1,0,1,0])
w_tmp = np.random.rand(X_tmp.shape[1])
b_tmp = 0.5
lambda_tmp = 0.7
dj_db_tmp, dj_dw_tmp =  compute_gradient_logistic_reg(X_tmp, y_tmp, w_tmp
```

```
print(f"dj_db: {dj_db_tmp}", )
print(f"Regularized dj_dw:\n {dj_dw_tmp.tolist()}", )
```

```
dj_db: 0.341798994972791
Regularized dj_dw:
 [0.17380012933994293, 0.32007507881566943, 0.10776313396851499]
```

Expected Output

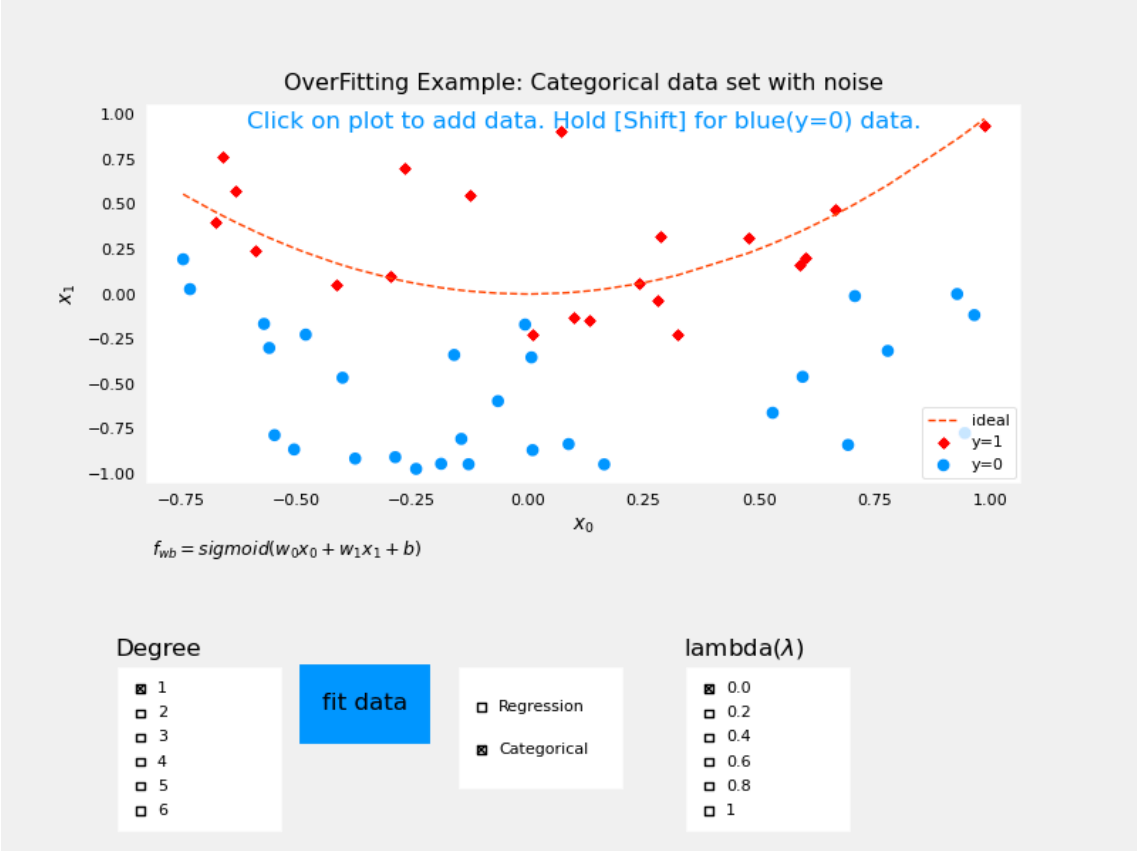
```
dj_db: 0.341798994972791
Regularized dj_dw:
 [0.17380012933994293, 0.32007507881566943,
0.10776313396851499]
```

Rerun over-fitting example

In [10]:
```
plt.close("all")
display(output)
ofit = overfit_example(True)
```

```
Output()
```

Figure



In the plot above, try out regularization on the previous example. In particular:

- Categorical (logistic regression)
  - set degree to 6, lambda to 0 (no regularization), fit the data
  - now set lambda to 1 (increase regularization), fit the data, notice the difference.
- Regression (linear regression)
  - try the same procedure.

Congratulations!

You have:

- examples of cost and gradient routines with regularization added for both linear and logistic regression
- developed some intuition on how regularization can reduce over-fitting

Final Practice lab course 1

Logistic Regression

In this exercise, you will implement logistic regression and apply it to two different datasets.

Outline

- 1 - Packages
- 2 - Logistic Regression
  - 2.1 Problem Statement
  - 2.2 Loading and visualizing the data
  - 2.3 Sigmoid function
  - 2.4 Cost function for logistic regression
  - 2.5 Gradient for logistic regression
  - 2.6 Learning parameters using gradient descent
  - 2.7 Plotting the decision boundary
  - 2.8 Evaluating logistic regression
- 3 - Regularized Logistic Regression
  - 3.1 Problem Statement
  - 3.2 Loading and visualizing the data
  - 3.3 Feature mapping
  - 3.4 Cost function for regularized logistic regression
  - 3.5 Gradient for regularized logistic regression
  - 3.6 Learning parameters using gradient descent
  - 3.7 Plotting the decision boundary
  - 3.8 Evaluating regularized logistic regression model

NOTE: *To prevent errors from the autograder, you are not allowed to edit or delete non-graded cells in this lab. Please also refrain from adding any new cells.* ***Once you have passed this assignment*** *and want to experiment with any of the non-graded code, you may follow the instructions at the bottom of this notebook.*

1 - Packages

First, let's run the cell below to import all the packages that you will need during this assignment.

- numpy is the fundamental package for scientific computing with Python.
- matplotlib is a famous library to plot graphs in Python.
- `utils.py` contains helper functions for this assignment. You do not need to modify code in this file.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from utils import *
import copy
import math

%matplotlib inline
```

2 - Logistic Regression

In this part of the exercise, you will build a logistic regression model to predict whether a student gets admitted into a university.

2.1 Problem Statement

Suppose that you are the administrator of a university department and you want to determine each applicant's chance of admission based on their results on two exams.

- You have historical data from previous applicants that you can use as a training set for logistic regression.
- For each training example, you have the applicant's scores on two exams and the admissions decision.
- Your task is to build a classification model that estimates an applicant's probability of admission based on the scores from those two exams.

2.2 Loading and visualizing the data

You will start by loading the dataset for this task.

- The `load_dataset()` function shown below loads the data into variables `X_train` and `y_train`
  - `X_train` contains exam scores on two exams for a student
  - `y_train` is the admission decision
    - `y_train = 1` if the student was admitted
    - `y_train = 0` if the student was not admitted
  - Both `X_train` and `y_train` are numpy arrays.

In [2]:
```
# load dataset
X_train, y_train = load_data("data/ex2data1.txt")
```

View the variables

Let's get more familiar with your dataset.

- A good place to start is to just print out each variable and see what it contains.

The code below prints the first five values of `X_train` and the type of the variable.

In [3]:
```
print("First five elements in X_train are:\n", X_train[:5])
print("Type of X_train:",type(X_train))
```

```
First five elements in X_train are:
 [[34.62365962 78.02469282]
 [30.28671077 43.89499752]
 [35.84740877 72.90219803]
 [60.18259939 86.3085521 ]
 [79.03273605 75.34437644]]
Type of X_train: <class 'numpy.ndarray'>
```

Now print the first five values of `y_train`

In [4]:
```
print("First five elements in y_train are:\n", y_train[:5])
print("Type of y_train:",type(y_train))
```

```
First five elements in y_train are:
 [0. 0. 0. 1. 1.]
Type of y_train: <class 'numpy.ndarray'>
```

Check the dimensions of your variables

Another useful way to get familiar with your data is to view its dimensions. Let's print the shape of `X_train` and `y_train` and see how many training examples we have in our dataset.

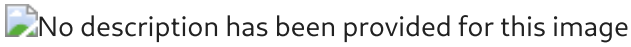
In [5]:
```
print ('The shape of X_train is: ' + str(X_train.shape))
print ('The shape of y_train is: ' + str(y_train.shape))
print ('We have m = %d training examples' % (len(y_train)))
```

```
The shape of X_train is: (100, 2)
The shape of y_train is: (100,)
We have m = 100 training examples
```

Visualize your data

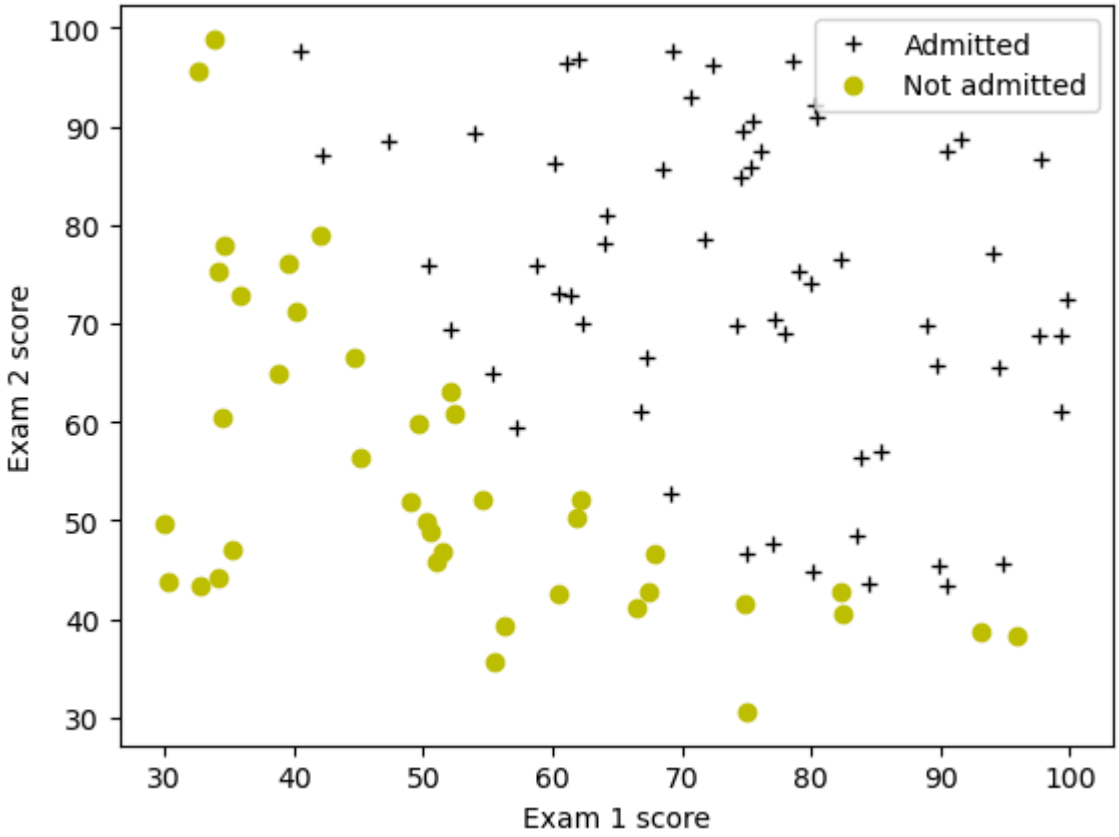
Before starting to implement any learning algorithm, it is always good to visualize the data if possible.

- The code below displays the data on a 2D plot (as shown below), where the axes are the two exam scores, and the positive and negative examples are shown with different markers.
- We use a helper function in the `utils.py` file to generate this plot.



In [6]:
```
# Plot examples
plot_data(X_train, y_train[:], pos_label="Admitted", neg_label="Not admit

# Set the y-axis label
plt.ylabel('Exam 2 score')
# Set the x-axis label
plt.xlabel('Exam 1 score')
plt.legend(loc="upper right")
plt.show()
```



Your goal is to build a logistic regression model to fit this data.

- With this model, you can then predict if a new student will be admitted based on their scores on the two exams.

2.3 Sigmoid function

Recall that for logistic regression, the model is represented as

$$f_{\mathbf{w},b}(x) = g(\mathbf{w} \cdot \mathbf{x} + b)$$

where function g is the sigmoid function. The sigmoid function is defined as:

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

Let's implement the sigmoid function first, so it can be used by the rest of this assignment.

Exercise 1

Please complete the `sigmoid` function to calculate

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

Note that

- `z` is not always a single number, but can also be an array of numbers.

- If the input is an array of numbers, we'd like to apply the sigmoid function to each value in the input array.

If you get stuck, you can check out the hints presented after the cell below to help you with the implementation.

In [7]:
```
# UNQ_C1
# GRADED FUNCTION: sigmoid

def sigmoid(z):
    """
    Compute the sigmoid of z

    Args:
        z (ndarray): A scalar, numpy array of any size.

    Returns:
        g (ndarray): sigmoid(z), with the same shape as z

    """

    ### START CODE HERE ###

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

    ### END SOLUTION ###

    return g
```

▼ Click for hints

* `numpy` has a function called `np.exp()`, which offers a convinient way to calculate the exponential (e^z) of all elements in the input array (`z`).

▼ Click for more hints

- You can translate e^{-z} into code as `np.exp(-z)`

- You can translate $1/e^{-z}$ into code as `1/np.exp(-z)`

  If you're still stuck, you can check the hints presented below to figure out how to calculate `g`

  #### ▼ Hint to calculate g

  `g = 1 / (1 + np.exp(-z))`

When you are finished, try testing a few values by calling `sigmoid(x)` in the cell below.

- For large positive values of x, the sigmoid should be close to 1, while for large negative values, the sigmoid should be close to 0.
- Evaluating `sigmoid(0)` should give you exactly 0.5.

In [8]:
```
# Note: You can edit this value
value = 0

print (f"sigmoid({value}) = {sigmoid(value)}")
```

```
sigmoid(0) = 0.5
```

Expected Output:

sigmoid(0)	0.5

- As mentioned before, your code should also work with vectors and matrices. For a matrix, your function should perform the sigmoid function on every element.

In [9]:
```
print ("sigmoid([ -1, 0, 1, 2]) = " + str(sigmoid(np.array([-1, 0, 1, 2]))

# UNIT TESTS
from public_tests import *
sigmoid_test(sigmoid)
```

```
sigmoid([ -1, 0, 1, 2]) = [0.26894142 0.5        0.73105858 0.88079708]
All tests passed!
```

Expected Output:

sigmoid([-1, 0, 1, 2])	[0.26894142 0.5 0.73105858 0.88079708]

2.4 Cost function for logistic regression

In this section, you will implement the cost function for logistic regression.

Exercise 2

Please complete the `compute_cost` function using the equations below.

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

- m is the number of training examples in the dataset
- $loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)})$ is the cost for a single data point, which is -

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

- $f_{\mathbf{w},b}(\mathbf{x}^{(i)})$ is the model's prediction, while $y^{(i)}$, which is the actual label
- $f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = g(\mathbf{w} \cdot \mathbf{x^{(i)}} + b)$ where function g is the sigmoid function.

- It might be helpful to first calculate an intermediate variable $z_{\mathbf{w},b}(\mathbf{x}^{(i)}) = \mathbf{w} \cdot \mathbf{x}^{(\mathbf{i})} + b = w_0 x_0^{(i)} + \ldots + w_{n-1} x_{n-1}^{(i)} + b$ where n is the number of features, before calculating $f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = g(z_{\mathbf{w},b}(\mathbf{x}^{(i)}))$

Note:

- As you are doing this, remember that the variables `X_train` and `y_train` are not scalar values but matrices of shape (m, n) and $(m,1)$ respectively, where n is the number of features and m is the number of training examples.
- You can use the sigmoid function that you implemented above for this part.

If you get stuck, you can check out the hints presented after the cell below to help you with the implementation.

In [10]:
```
# UNQ_C2
# GRADED FUNCTION: compute_cost
def compute_cost(X, y, w, b, *argv):
    """
    Computes the cost over all examples
    Args:
      X : (ndarray Shape (m,n)) data, m examples by n features
      y : (ndarray Shape (m,))  target value
      w : (ndarray Shape (n,))  values of parameters of the model
      b : (scalar)              value of bias parameter of the model
      *argv : unused, for compatibility with regularized version below
    Returns:
      total_cost : (scalar) cost
    """

    m, n = X.shape

    ### START CODE HERE ###
    total_cost = 0
    for i in range(m):
        f_wb  = sigmoid(np.dot(w,X[i]) + b)
        total_cost += -y[i] * np.log(f_wb) - (1 - y[i]) * np.log(1 - f_wb
    total_cost /= m


    ### END CODE HERE ###

    return total_cost
```

▼ Click for hints

- You can represent a summation operator eg: $h = \sum_{i=0}^{m-1} 2i$ in code as follows:

```
h = 0
    for i in range(m):
        h = h + 2*i
```

- In this case, you can iterate over all the examples in `X` using a for loop and add the `loss` from each iteration to a variable (`loss_sum`) initialized outside the loop.
- Then, you can return the `total_cost` as `loss_sum` divided by `m` .
- If you are new to Python, please check that your code is properly indented with consistent spaces or tabs. Otherwise, it might produce a different output or raise an `IndentationError: unexpected indent` error. You can refer to this topic in our community for details.

▼ Click for more hints

- Here's how you can structure the overall implementation for this function

```
def compute_cost(X, y, w, b, *argv):
    m, n = X.shape

    ### START CODE HERE ###
    loss_sum = 0

    # Loop over each training example
    for i in range(m):

        # First calculate z_wb = w[0]*X[i][0]+...+w[n-1]*X[i][n-
1]+b
        z_wb = 0
        # Loop over each feature
        for j in range(n):
            # Add the corresponding term to z_wb
            z_wb_ij = # Your code here to calculate w[j] * X[i]
[j]
            z_wb += z_wb_ij # equivalent to z_wb = z_wb + z_wb_ij
        # Add the bias term to z_wb
        z_wb += b # equivalent to z_wb = z_wb + b

        f_wb = # Your code here to calculate prediction f_wb for
a training example
        loss =  # Your code here to calculate loss for a training
example

        loss_sum += loss # equivalent to loss_sum = loss_sum +
loss

    total_cost = (1 / m) * loss_sum
    ### END CODE HERE ###

    return total_cost
```

If you're still stuck, you can check the hints presented below to figure out how to calculate `z_wb_ij` , `f_wb` and `cost` .

▼ Hint to calculate z_wb_ij

`z_wb_ij = w[j]*X[i][j]`

▼ **Hint to calculate f_wb**

$f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = g(z_{\mathbf{w},b}(\mathbf{x}^{(i)}))$ where g is the sigmoid function. You can simply call the `sigmoid` function implemented above.

▼ **More hints to calculate f**

You can compute f_wb as `f_wb = sigmoid(z_wb)`

▼ **Hint to calculate loss**

You can use the np.log function to calculate the log

▼ **More hints to calculate loss**

You can compute loss as `loss = -y[i] * np.log(f_wb) - (1 - y[i]) * np.log(1 - f_wb)`

Run the cells below to check your implementation of the `compute_cost` function with two different initializations of the parameters w and b

In [11]:
```
m, n = X_train.shape

# Compute and display cost with w and b initialized to zeros
initial_w = np.zeros(n)
initial_b = 0.
cost = compute_cost(X_train, y_train, initial_w, initial_b)
print('Cost at initial w and b (zeros): {:.3f}'.format(cost))
```

```
Cost at initial w and b (zeros): 0.693
```

Expected Output:

Cost at initial w and b (zeros)	0.693

In [12]:
```
# Compute and display cost with non-zero w and b
test_w = np.array([0.2, 0.2])
test_b = -24.
cost = compute_cost(X_train, y_train, test_w, test_b)

print('Cost at test w and b (non-zeros): {:.3f}'.format(cost))


# UNIT TESTS
compute_cost_test(compute_cost)
```

```
Cost at test w and b (non-zeros): 0.218
All tests passed!
```

Expected Output:

Cost at test w and b (non-zeros):	0.218

2.5 Gradient for logistic regression

In this section, you will implement the gradient for logistic regression.

Recall that the gradient descent algorithm is:

$$\begin{aligned}
&\text{repeat until convergence: } \{ \\
&b := b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \\
&w_j := w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \qquad \text{for j := 0..n-1} \qquad (1) \\
&\}
\end{aligned}$$

where, parameters b, w_j are all updated simultaniously

Exercise 3

Please complete the `compute_gradient` function to compute $\frac{\partial J(\mathbf{w},b)}{\partial w}$, $\frac{\partial J(\mathbf{w},b)}{\partial b}$ from equations (2) and (3) below.

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)}) \qquad (2)$$

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)})x_j^{(i)} \qquad (3)$$

- m is the number of training examples in the dataset
- $f_{\mathbf{w},b}(x^{(i)})$ is the model's prediction, while $y^{(i)}$ is the actual label
- **Note**: While this gradient looks identical to the linear regression gradient, the formula is actually different because linear and logistic regression have different definitions of $f_{\mathbf{w},b}(x)$.

As before, you can use the sigmoid function that you implemented above and if you get stuck, you can check out the hints presented after the cell below to help you with the implementation.

In [13]:
```
# UNQ_C3
# GRADED FUNCTION: compute_gradient
def compute_gradient(X, y, w, b, *argv):
    """
    Computes the gradient for logistic regression

    Args:
      X : (ndarray Shape (m,n)) data, m examples by n features
      y : (ndarray Shape (m,))  target value
      w : (ndarray Shape (n,))  values of parameters of the model
      b : (scalar)              value of bias parameter of the model
      *argv : unused, for compatibility with regularized version below
    Returns
      dj_dw : (ndarray Shape (n,)) The gradient of the cost w.r.t. the pa
      dj_db : (scalar)             The gradient of the cost w.r.t. the pa
    """
    m, n = X.shape
    dj_dw = np.zeros(w.shape)
    dj_db = 0.
```

```
    ### START CODE HERE ###


#     for i in range(m):
#         z_wb = None
#         for j in range(n):
#             z_wb += None
#         z_wb += None
#         f_wb = None

#         dj_db_i = None
#         dj_db += None

#         for j in range(n):
#             dj_dw[j] = None

#     dj_dw = None
#     dj_db = None


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

    dj_dw /= m
    dj_db /= m

    ### END CODE HERE ###


    return dj_db, dj_dw
```

▼ **Click for hints**

- Here's how you can structure the overall implementation for this function

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

        ### START CODE HERE ###
        for i in range(m):
            # Calculate f_wb (exactly as you did in the
compute_cost function above)
            f_wb =

            # Calculate the  gradient for b from this example
            dj_db_i = # Your code here to calculate the error

            # add that to dj_db
            dj_db += dj_db_i

            # get dj_dw for each attribute
            for j in range(n):
                # You code here to calculate the gradient from
```

```
                the i-th example for j-th attribute
                dj_dw_ij =
                dj_dw[j] += dj_dw_ij

        # divide dj_db and dj_dw by total number of examples
        dj_dw = dj_dw / m
        dj_db = dj_db / m
        ### END CODE HERE ###

        return dj_db, dj_dw
```

- If you are new to Python, please check that your code is properly indented with consistent spaces or tabs. Otherwise, it might produce a different output or raise an `IndentationError: unexpected indent` error. You can refer to this topic in our community for details.
- If you're still stuck, you can check the hints presented below to figure out how to calculate `f_wb` , `dj_db_i` and `dj_dw_ij`

▼ **Hint to calculate f_wb**

Recall that you calculated f_wb in `compute_cost` above — for detailed hints on how to calculate each intermediate term, check out the hints section below that exercise

▼ **More hints to calculate f_wb**

You can calculate f_wb as

```
            for i in range(m):
                # Calculate f_wb (exactly how you did it in
the compute_cost function above)
                z_wb = 0
                # Loop over each feature
                for j in range(n):
                    # Add the corresponding term to z_wb
                    z_wb_ij = X[i, j] * w[j]
                    z_wb += z_wb_ij

                # Add bias term
                z_wb += b

                # Calculate the prediction from the model
                f_wb = sigmoid(z_wb)
```

▼ **Hint to calculate dj_db_i**

You can calculate dj_db_i as `dj_db_i = f_wb - y[i]`

▼ **Hint to calculate dj_dw_ij**

You can calculate dj_dw_ij as `dj_dw_ij = (f_wb - y[i])* X[i][j]`

Run the cells below to check your implementation of the `compute_gradient` function with two different initializations of the parameters w and b

In [14]:
```
# Compute and display gradient with w and b initialized to zeros
initial_w = np.zeros(n)
initial_b = 0.

dj_db, dj_dw = compute_gradient(X_train, y_train, initial_w, initial_b)
```

```
print(f'dj_db at initial w and b (zeros):{dj_db}' )
print(f'dj_dw at initial w and b (zeros):{dj_dw.tolist()}' )
```

```
dj_db at initial w and b (zeros):-0.1
dj_dw at initial w and b (zeros):[-12.00921658929115, -11.262842205513591]
```

Expected Output:

dj_db at initial w and b (zeros)	-0.1
dj_dw at initial w and b (zeros):	[-12.00921658929115, -11.262842205513591]

In [15]:
```
# Compute and display cost and gradient with non-zero w and b
test_w = np.array([ 0.2, -0.5])
test_b = -24
dj_db, dj_dw  = compute_gradient(X_train, y_train, test_w, test_b)

print('dj_db at test w and b:', dj_db)
print('dj_dw at test w and b:', dj_dw.tolist())

# UNIT TESTS
compute_gradient_test(compute_gradient)
```

```
dj_db at test w and b: -0.5999999999991071
dj_dw at test w and b: [-44.831353617873795, -44.37384124953978]
All tests passed!
```

Expected Output:

dj_db at test w and b (non-zeros)	-0.5999999999991071
dj_dw at test w and b (non-zeros):	[-44.8313536178737957, -44.37384124953978]

2.6 Learning parameters using gradient descent

Similar to the previous assignment, you will now find the optimal parameters of a logistic regression model by using gradient descent.

- You don't need to implement anything for this part. Simply run the cells below.
- A good way to verify that gradient descent is working correctly is to look at the value of $J(\mathbf{w}, b)$ and check that it is decreasing with each step.
- Assuming you have implemented the gradient and computed the cost correctly, your value of $J(\mathbf{w}, b)$ should never increase, and should converge to a steady value by the end of the algorithm.

In [16]:
```
def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function,
    """
    Performs batch gradient descent to learn theta. Updates theta by taki
    num_iters gradient steps with learning rate alpha

    Args:
      X :    (ndarray Shape (m, n) data, m examples by n features
      y :    (ndarray Shape (m,))  target value
      w_in : (ndarray Shape (n,))  Initial values of parameters of the mo
```

```
      b_in : (scalar)              Initial value of parameter of the mode
      cost_function :              function to compute cost
      gradient_function :          function to compute gradient
      alpha : (float)              Learning rate
      num_iters : (int)            number of iterations to run gradient d
      lambda_ : (scalar, float)    regularization constant

    Returns:
      w : (ndarray Shape (n,)) Updated values of parameters of the model
          running gradient descent
      b : (scalar)                Updated value of parameter of the model
          running gradient descent
    """

    # number of training examples
    m = len(X)

    # An array to store cost J and w's at each iteration primarily for gr
    J_history = []
    w_history = []

    for i in range(num_iters):

        # Calculate the gradient and update the parameters
        dj_db, dj_dw = gradient_function(X, y, w_in, b_in, lambda_)

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

        # Save cost J at each iteration
        if i<100000:      # prevent resource exhaustion
            cost =  cost_function(X, y, w_in, b_in, lambda_)
            J_history.append(cost)

        # Print cost every at intervals 10 times or as many iterations if
        if i% math.ceil(num_iters/10) == 0 or i == (num_iters-1):
            w_history.append(w_in)
            print(f"Iteration {i:4}: Cost {float(J_history[-1]):8.2f}   "

    return w_in, b_in, J_history, w_history #return w and J,w history for
```

Now let's run the gradient descent algorithm above to learn the parameters for our dataset.

Note The code block below takes a couple of minutes to run, especially with a non-vectorized version. You can reduce the `iterations` to test your implementation and iterate faster. If you have time later, try running 100,000 iterations for better results.

In [17]:
```
np.random.seed(1)
initial_w = 0.01 * (np.random.rand(2) - 0.5)
initial_b = -8

# Some gradient descent settings
iterations = 10000
alpha = 0.001
```

```
w,b, J_history,_ = gradient_descent(X_train ,y_train, initial_w, initial_
                                    compute_cost, compute_gradient, alpha,
```

```
Iteration    0: Cost     0.96
Iteration 1000: Cost     0.31
Iteration 2000: Cost     0.30
Iteration 3000: Cost     0.30
Iteration 4000: Cost     0.30
Iteration 5000: Cost     0.30
Iteration 6000: Cost     0.30
Iteration 7000: Cost     0.30
Iteration 8000: Cost     0.30
Iteration 9000: Cost     0.30
Iteration 9999: Cost     0.30
```

▶ **Expected Output: Cost 0.30, (Click to see details):**

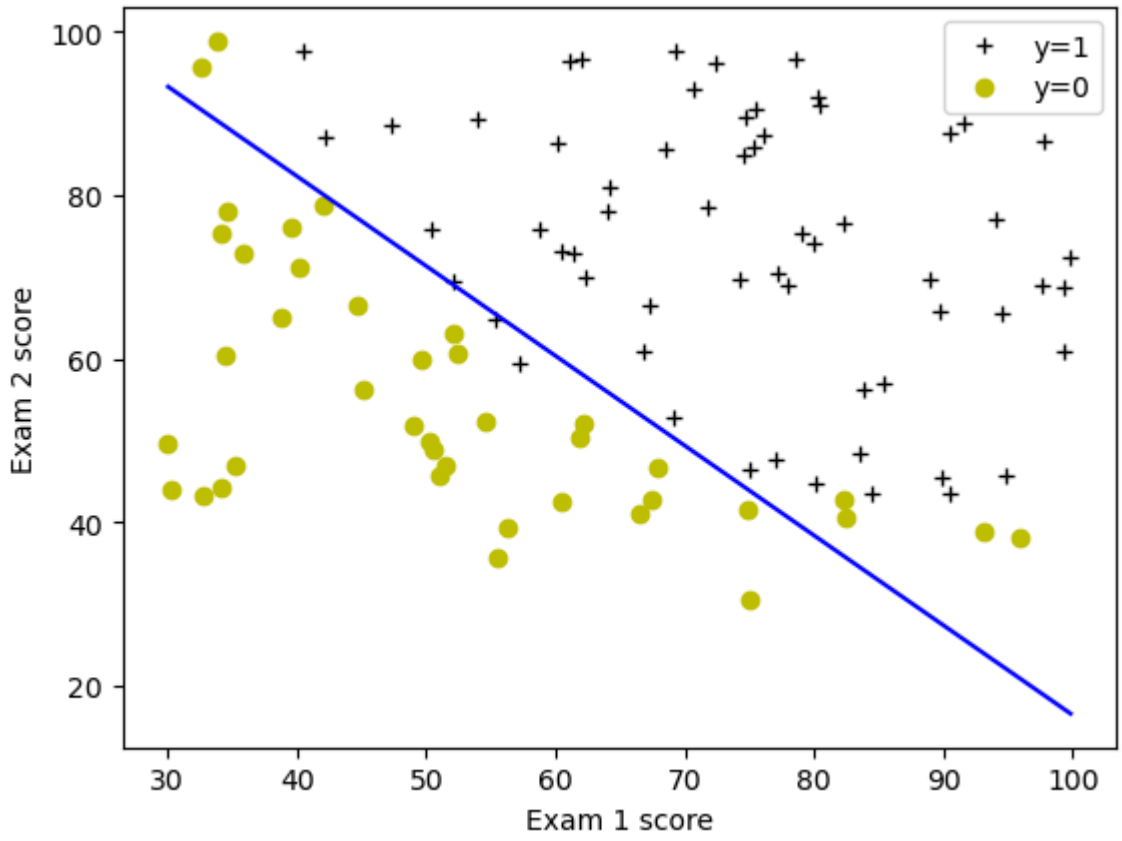
2.7 Plotting the decision boundary

We will now use the final parameters from gradient descent to plot the linear fit. If you implemented the previous parts correctly, you should see a plot similar to the following plot:

 No description has been provided for this image

We will use a helper function in the `utils.py` file to create this plot.

In [18]:
```
plot_decision_boundary(w, b, X_train, y_train)
# Set the y-axis label
plt.ylabel('Exam 2 score')
# Set the x-axis label
plt.xlabel('Exam 1 score')
plt.legend(loc="upper right")
plt.show()
```



2.8 Evaluating logistic regression

We can evaluate the quality of the parameters we have found by seeing how well the learned model predicts on our training set.

You will implement the `predict` function below to do this.

Exercise 4

Please complete the `predict` function to produce `1` or `0` predictions given a dataset and a learned parameter vector w and b.

- First you need to compute the prediction from the model $f(x^{(i)}) = g(w \cdot x^{(i)} + b)$ for every example
    - You've implemented this before in the parts above
- We interpret the output of the model ($f(x^{(i)})$) as the probability that $y^{(i)} = 1$ given $x^{(i)}$ and parameterized by w.
- Therefore, to get a final prediction ($y^{(i)} = 0$ or $y^{(i)} = 1$) from the logistic regression model, you can use the following heuristic -

  if $f(x^{(i)}) >= 0.5$, predict $y^{(i)} = 1$

  if $f(x^{(i)}) < 0.5$, predict $y^{(i)} = 0$

If you get stuck, you can check out the hints presented after the cell below to help you with the implementation.

In [19]:
```
# UNQ_C4
# GRADED FUNCTION: predict

def predict(X, w, b):
    """
    Predict whether the label is 0 or 1 using learned logistic
    regression parameters w

    Args:
      X : (ndarray Shape (m,n)) data, m examples by n features
      w : (ndarray Shape (n,))  values of parameters of the model
      b : (scalar)              value of bias parameter of the model

    Returns:
      p : (ndarray (m,)) The predictions for X using a threshold at 0.5
    """
    # number of training examples
    m, n = X.shape
    p = np.zeros(m)

    ### START CODE HERE ###
    # Loop over each example
    for i in range(m):
        z_wb = np.dot(w, X[i]) + b
#         # Loop over each feature
#         for j in range(n):
#             # Add the corresponding term to z_wb
#             z_wb += None

#         # Add bias term
#         z_wb += None

        # Calculate the prediction for this example
        f_wb = sigmoid(z_wb)

        # Apply the threshold
        p[i] = f_wb >= 0.5

    ### END CODE HERE ###
    return p
```

▼ Click for hints

- Here's how you can structure the overall implementation for this function

```
def predict(X, w, b):
        # number of training examples
        m, n = X.shape
        p = np.zeros(m)

        ### START CODE HERE ###
        # Loop over each example
        for i in range(m):

            # Calculate f_wb (exactly how you did it in the
```

```
        compute_cost function above)
                    # using a couple of lines of code
                    f_wb =

                    # Calculate the prediction for that training
        example
                    p[i] = # Your code here to calculate the
        prediction based on f_wb

                ### END CODE HERE ###
                return p
```

If you're still stuck, you can check the hints presented below to figure out how to calculate `f_wb` and `p[i]`

▼ **Hint to calculate f_wb**

Recall that you calculated f_wb in `compute_cost` above — for detailed hints on how to calculate each intermediate term, check out the hints section below that exercise

▼ **More hints to calculate f_wb**

You can calculate f_wb as

```
        for i in range(m):
            # Calculate f_wb (exactly how you did it in
the compute_cost function above)
            z_wb = 0
            # Loop over each feature
            for j in range(n):
                # Add the corresponding term to z_wb
                z_wb_ij = X[i, j] * w[j]
                z_wb += z_wb_ij

            # Add bias term
            z_wb += b

            # Calculate the prediction from the model
            f_wb = sigmoid(z_wb)
```

▼ **Hint to calculate p[i]**

As an example, if you'd like to say x = 1 if y is less than 3 and 0 otherwise, you can express it in code as `x = y < 3` . Now do the same for p[i] = 1 if f_wb >= 0.5 and 0 otherwise.

▼ **More hints to calculate p[i]**

You can compute p[i] as `p[i] = f_wb >= 0.5`

Once you have completed the function `predict` , let's run the code below to report the training accuracy of your classifier by computing the percentage of examples it got correct.

In [20]:
```
# Test your predict code
np.random.seed(1)
tmp_w = np.random.randn(2)
tmp_b = 0.3
```

```
tmp_X = np.random.randn(4, 2) - 0.5

tmp_p = predict(tmp_X, tmp_w, tmp_b)
print(f'Output of predict: shape {tmp_p.shape}, value {tmp_p}')

# UNIT TESTS
predict_test(predict)
```

```
Output of predict: shape (4,), value [0. 1. 1. 1.]
All tests passed!
```

Expected output

Output of predict: shape (4,),value [0. 1. 1. 1.]

Now let's use this to compute the accuracy on the training set

In [21]:
```
#Compute accuracy on our training set
p = predict(X_train, w,b)
print('Train Accuracy: %f'%(np.mean(p == y_train) * 100))
```

```
Train Accuracy: 92.000000
```

Train Accuracy (approx): 92.00

3 - Regularized Logistic Regression

In this part of the exercise, you will implement regularized logistic regression to predict whether microchips from a fabrication plant passes quality assurance (QA). During QA, each microchip goes through various tests to ensure it is functioning correctly.

3.1 Problem Statement

Suppose you are the product manager of the factory and you have the test results for some microchips on two different tests.

- From these two tests, you would like to determine whether the microchips should be accepted or rejected.
- To help you make the decision, you have a dataset of test results on past microchips, from which you can build a logistic regression model.

3.2 Loading and visualizing the data

Similar to previous parts of this exercise, let's start by loading the dataset for this task and visualizing it.

- The `load_dataset()` function shown below loads the data into variables `X_train` and `y_train`
  - `X_train` contains the test results for the microchips from two tests
  - `y_train` contains the results of the QA
    - `y_train = 1` if the microchip was accepted

    - `y_train = 0` if the microchip was rejected
- Both `X_train` and `y_train` are numpy arrays.

In [22]:
```
# load dataset
X_train, y_train = load_data("data/ex2data2.txt")
```

View the variables

The code below prints the first five values of `X_train` and `y_train` and the type of the variables.

In [23]:
```
# print X_train
print("X_train:", X_train[:5])
print("Type of X_train:",type(X_train))

# print y_train
print("y_train:", y_train[:5])
print("Type of y_train:",type(y_train))
```

```
X_train: [[ 0.051267  0.69956 ]
 [-0.092742  0.68494 ]
 [-0.21371   0.69225 ]
 [-0.375     0.50219 ]
 [-0.51325   0.46564 ]]
Type of X_train: <class 'numpy.ndarray'>
y_train: [1. 1. 1. 1. 1.]
Type of y_train: <class 'numpy.ndarray'>
```

Check the dimensions of your variables

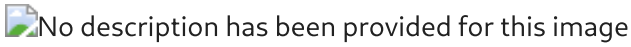
Another useful way to get familiar with your data is to view its dimensions. Let's print the shape of `X_train` and `y_train` and see how many training examples we have in our dataset.

In [24]:
```
print ('The shape of X_train is: ' + str(X_train.shape))
print ('The shape of y_train is: ' + str(y_train.shape))
print ('We have m = %d training examples' % (len(y_train)))
```

```
The shape of X_train is: (118, 2)
The shape of y_train is: (118,)
We have m = 118 training examples
```

Visualize your data

The helper function `plot_data` (from `utils.py`) is used to generate a figure like Figure 3, where the axes are the two test scores, and the positive (y = 1, accepted) and negative (y = 0, rejected) examples are shown with different markers.



In [25]:
```
# Plot examples
plot_data(X_train, y_train[:], pos_label="Accepted", neg_label="Rejected"

# Set the y-axis label
plt.ylabel('Microchip Test 2')
# Set the x-axis label
plt.xlabel('Microchip Test 1')
plt.legend(loc="upper right")
plt.show()
```

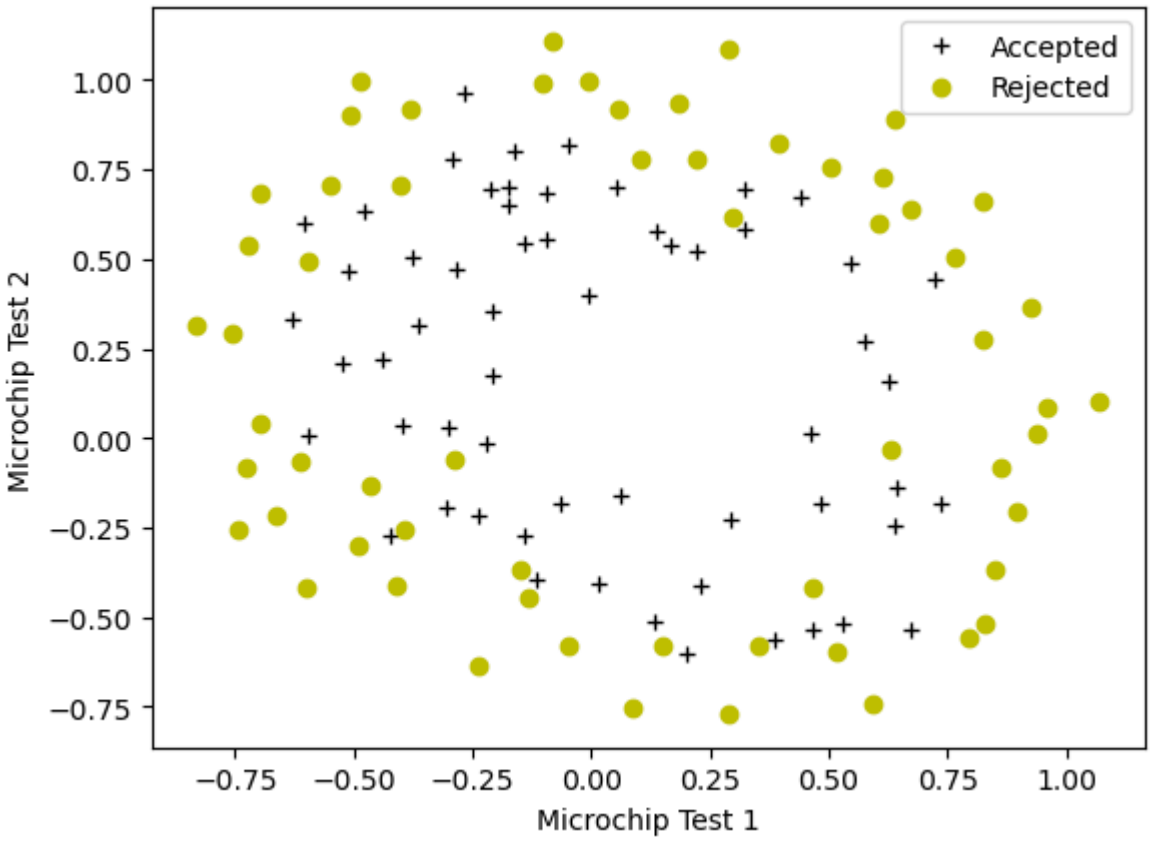


Figure 3 shows that our dataset cannot be separated into positive and negative examples by a straight-line through the plot. Therefore, a straight forward application of logistic regression will not perform well on this dataset since logistic regression will only be able to find a linear decision boundary.

3.3 Feature mapping

One way to fit the data better is to create more features from each data point. In the provided function `map_feature`, we will map the features into all polynomial terms of x_1 and x_2 up to the sixth power.

$$\text{map\_feature}(x) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \\ x_1^3 \\ \vdots \\ x_1 x_2^5 \\ x_2^6 \end{bmatrix}$$

As a result of this mapping, our vector of two features (the scores on two QA tests) has been transformed into a 27-dimensional vector.

- A logistic regression classifier trained on this higher-dimension feature vector will have a more complex decision boundary and will be nonlinear when drawn in our 2-

dimensional plot.

- We have provided the `map_feature` function for you in utils.py.

In [26]:
```
print("Original shape of data:", X_train.shape)

mapped_X =  map_feature(X_train[:, 0], X_train[:, 1])
print("Shape after feature mapping:", mapped_X.shape)
```

```
Original shape of data: (118, 2)
Shape after feature mapping: (118, 27)
```

Let's also print the first elements of `X_train` and `mapped_X` to see the tranformation.

In [27]:
```
print("X_train[0]:", X_train[0])
print("mapped X_train[0]:", mapped_X[0])
```

```
X_train[0]: [0.051267 0.69956 ]
mapped X_train[0]: [5.12670000e-02 6.99560000e-01 2.62830529e-03 3.5864342
5e-02
 4.89384194e-01 1.34745327e-04 1.83865725e-03 2.50892595e-02
 3.42353606e-01 6.90798869e-06 9.42624411e-05 1.28625106e-03
 1.75514423e-02 2.39496889e-01 3.54151856e-07 4.83255257e-06
 6.59422333e-05 8.99809795e-04 1.22782870e-02 1.67542444e-01
 1.81563032e-08 2.47750473e-07 3.38066048e-06 4.61305487e-05
 6.29470940e-04 8.58939846e-03 1.17205992e-01]
```

While the feature mapping allows us to build a more expressive classifier, it is also more susceptible to overfitting. In the next parts of the exercise, you will implement regularized logistic regression to fit the data and also see for yourself how regularization can help combat the overfitting problem.

3.4 Cost function for regularized logistic regression

In this part, you will implement the cost function for regularized logistic regression.

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

$$J(\mathbf{w},b) = \frac{1}{m}\sum_{i=0}^{m-1}\left[-y^{(i)}\log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=0}^{n-1}u$$

Compare this to the cost function without regularization (which you implemented above), which is of the form

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

The difference is the regularization term, which is

$$\frac{\lambda}{2m}\sum_{j=0}^{n-1}w_j^2$$

Note that the b parameter is not regularized.

Exercise 5

Please complete the `compute_cost_reg` function below to calculate the following term for each element in w

$$\frac{\lambda}{2m}\sum_{j=0}^{n-1} w_j^2$$

The starter code then adds this to the cost without regularization (which you computed above in `compute_cost`) to calculate the cost with regulatization.

If you get stuck, you can check out the hints presented after the cell below to help you with the implementation.

In [28]:
```
# UNQ_C5
def compute_cost_reg(X, y, w, b, lambda_ = 1):
    """
    Computes the cost over all examples
    Args:
      X : (ndarray Shape (m,n)) data, m examples by n features
      y : (ndarray Shape (m,))  target value
      w : (ndarray Shape (n,))  values of parameters of the model
      b : (scalar)              value of bias parameter of the model
      lambda_ : (scalar, float) Controls amount of regularization
    Returns:
      total_cost : (scalar)     cost
    """

    m, n = X.shape

    # Calls the compute_cost function that you implemented above
    cost_without_reg = compute_cost(X, y, w, b)

    # You need to calculate this value
    reg_cost = 0.

    ### START CODE HERE ###

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



    ### END CODE HERE ###

    # Add the regularization cost to get the total cost
    total_cost = cost_without_reg + reg_cost

    return total_cost
```

▼ Click for hints

- Here's how you can structure the overall implementation for this function

```
def compute_cost_reg(X, y, w, b, lambda_ = 1):

        m, n = X.shape

        # Calls the compute_cost function that you implemented
above
        cost_without_reg = compute_cost(X, y, w, b)

        # You need to calculate this value
        reg_cost = 0.

        ### START CODE HERE ###
        for j in range(n):
            reg_cost_j = # Your code here to calculate the
cost from w[j]
            reg_cost = reg_cost + reg_cost_j
        reg_cost = (lambda_/(2 * m)) * reg_cost
        ### END CODE HERE ###

        # Add the regularization cost to get the total cost
        total_cost = cost_without_reg + reg_cost

    return total_cost
```

If you're still stuck, you can check the hints presented below to figure out how to calculate `reg_cost_j`

▼ **Hint to calculate reg_cost_j**

You can use calculate reg_cost_j as `reg_cost_j = w[j]**2`

Run the cell below to check your implementation of the `compute_cost_reg` function.

In [29]:
```
X_mapped = map_feature(X_train[:, 0], X_train[:, 1])
np.random.seed(1)
initial_w = np.random.rand(X_mapped.shape[1]) - 0.5
initial_b = 0.5
lambda_ = 0.5
cost = compute_cost_reg(X_mapped, y_train, initial_w, initial_b, lambda_)

print("Regularized cost :", cost)

# UNIT TEST
compute_cost_reg_test(compute_cost_reg)
```

```
Regularized cost : 0.6618252552483948
All tests passed!
```

Expected Output:

Regularized cost : 0.6618252552483948

3.5 Gradient for regularized logistic regression

In this section, you will implement the gradient for regularized logistic regression.

The gradient of the regularized cost function has two components. The first, $\frac{\partial J(\mathbf{w},b)}{\partial b}$ is a scalar, the other is a vector with the same shape as the parameters $\mathbf{w}$, where the j^{th} element is defined as follows:

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})$$

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \left(\frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}w_j \quad \text{for } j = 0...(n-1)$$

Compare this to the gradient of the cost function without regularization (which you implemented above), which is of the form

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)}) \tag{2}$$

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)})x_j^{(i)} \tag{3}$$

As you can see, $\frac{\partial J(\mathbf{w},b)}{\partial b}$ is the same, the difference is the following term in $\frac{\partial J(\mathbf{w},b)}{\partial w}$, which is

$$\frac{\lambda}{m}w_j \quad \text{for } j = 0...(n-1)$$

Exercise 6

Please complete the `compute_gradient_reg` function below to modify the code below to calculate the following term

$$\frac{\lambda}{m}w_j \quad \text{for } j = 0...(n-1)$$

The starter code will add this term to the $\frac{\partial J(\mathbf{w},b)}{\partial w}$ returned from `compute_gradient` above to get the gradient for the regularized cost function.

If you get stuck, you can check out the hints presented after the cell below to help you with the implementation.

In [30]:
```
# UNQ_C6
def compute_gradient_reg(X, y, w, b, lambda_ = 1):
    """
    Computes the gradient for logistic regression with regularization

    Args:
      X : (ndarray Shape (m,n)) data, m examples by n features
      y : (ndarray Shape (m,))  target value
      w : (ndarray Shape (n,))  values of parameters of the model
      b : (scalar)              value of bias parameter of the model
      lambda_ : (scalar,float)  regularization constant
    Returns
```

```
      dj_db : (scalar)             The gradient of the cost w.r.t. the pa
      dj_dw : (ndarray Shape (n,)) The gradient of the cost w.r.t. the pa

    """
    m, n = X.shape

    dj_db, dj_dw = compute_gradient(X, y, w, b)

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

    ### END CODE HERE ###

    return dj_db, dj_dw
```

▼ Click for hints

- Here's how you can structure the overall implementation for this function

```
def compute_gradient_reg(X, y, w, b, lambda_ = 1):
    m, n = X.shape

    dj_db, dj_dw = compute_gradient(X, y, w, b)

    ### START CODE HERE ###
    # Loop over the elements of w
    for j in range(n):

        dj_dw_j_reg = # Your code here to calculate the
regularization term for dj_dw[j]

        # Add the regularization term  to the correspoding
element of dj_dw
        dj_dw[j] = dj_dw[j] + dj_dw_j_reg

    ### END CODE HERE ###

    return dj_db, dj_dw
```

If you're still stuck, you can check the hints presented below to figure out how to calculate `dj_dw_j_reg`

▼ Hint to calculate dj_dw_j_reg

You can use calculate dj_dw_j_reg as `dj_dw_j_reg = (lambda_ / m) * w[j]`

Run the cell below to check your implementation of the `compute_gradient_reg` function.

In [31]:
```
X_mapped = map_feature(X_train[:, 0], X_train[:, 1])
np.random.seed(1)
initial_w  = np.random.rand(X_mapped.shape[1]) - 0.5
initial_b = 0.5

lambda_ = 0.5
```

```
dj_db, dj_dw = compute_gradient_reg(X_mapped, y_train, initial_w, initial

print(f"dj_db: {dj_db}", )
print(f"First few elements of regularized dj_dw:\n {dj_dw[:4].tolist()}",

# UNIT TESTS
compute_gradient_reg_test(compute_gradient_reg)
```

```
dj_db: 0.07138288792343662
First few elements of regularized dj_dw:
 [-0.010386028450548701, 0.011409852883280122, 0.0536273463274574, 0.00314
0278267313462]
All tests passed!
```

Expected Output:

dj_db:0.07138288792343

First few elements of regularized dj_dw:

[[-0.010386028450548], [0.011409852883280], [0.0536273463274], [0.003140278267313]]

3.6 Learning parameters using gradient descent

Similar to the previous parts, you will use your gradient descent function implemented above to learn the optimal parameters w,b.

- If you have completed the cost and gradient for regularized logistic regression correctly, you should be able to step through the next cell to learn the parameters w.
- After training our parameters, we will use it to plot the decision boundary.

Note

The code block below takes quite a while to run, especially with a non-vectorized version. You can reduce the `iterations` to test your implementation and iterate faster. If you have time later, run for 100,000 iterations to see better results.

In [32]:
```
# Initialize fitting parameters
np.random.seed(1)
initial_w = np.random.rand(X_mapped.shape[1])-0.5
initial_b = 1.

# Set regularization parameter lambda_ (you can try varying this)
lambda_ = 0.01

# Some gradient descent settings
iterations = 10000
alpha = 0.01

w,b, J_history,_ = gradient_descent(X_mapped, y_train, initial_w, initial
                                    compute_cost_reg, compute_gradient_re
                                    alpha, iterations, lambda_)
```

```
Iteration    0: Cost     0.72
Iteration 1000: Cost     0.59
Iteration 2000: Cost     0.56
Iteration 3000: Cost     0.53
Iteration 4000: Cost     0.51
Iteration 5000: Cost     0.50
Iteration 6000: Cost     0.48
Iteration 7000: Cost     0.47
Iteration 8000: Cost     0.46
Iteration 9000: Cost     0.45
Iteration 9999: Cost     0.45
```

▶ **Expected Output: Cost < 0.5 (Click for details)**

3.7 Plotting the decision boundary

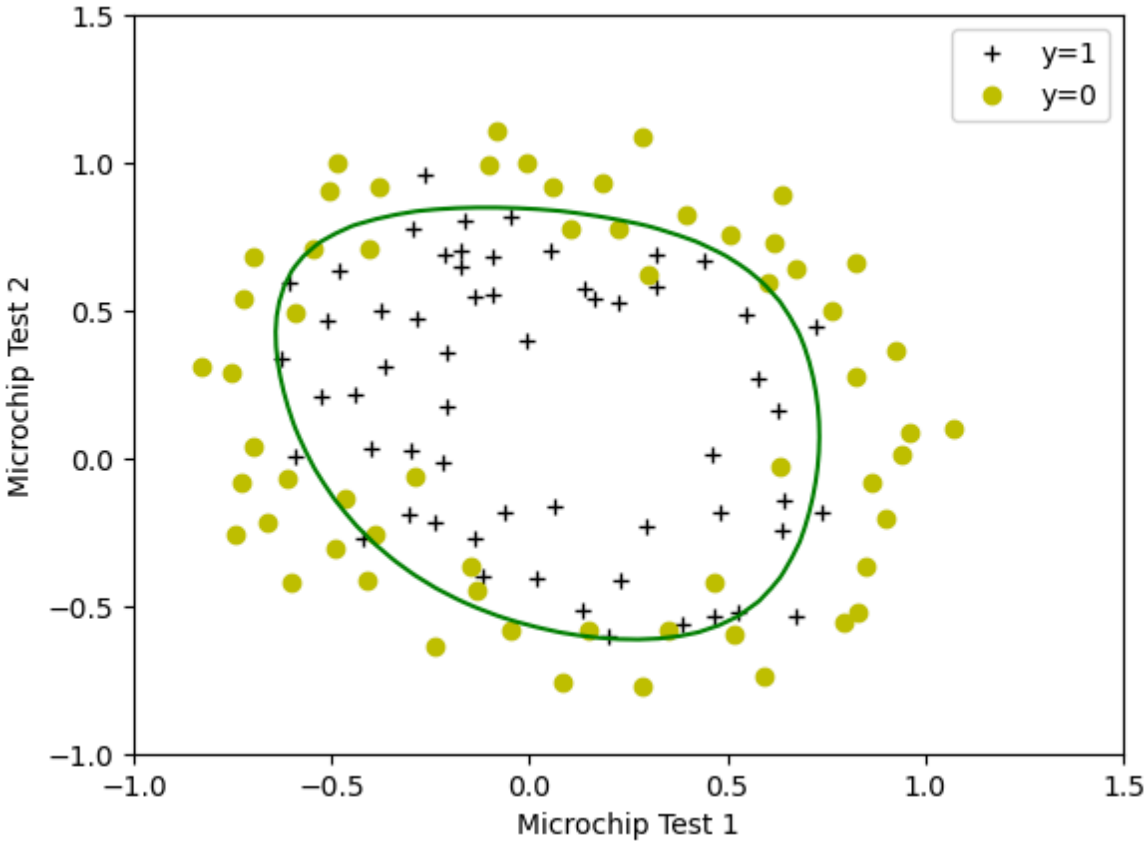
To help you visualize the model learned by this classifier, we will use our `plot_decision_boundary` function which plots the (non-linear) decision boundary that separates the positive and negative examples.

- In the function, we plotted the non-linear decision boundary by computing the classifier's predictions on an evenly spaced grid and then drew a contour plot of where the predictions change from y = 0 to y = 1.
- After learning the parameters w,b, the next step is to plot a decision boundary similar to Figure 4.

 No description has been provided for this image

In [33]:
```
plot_decision_boundary(w, b, X_mapped, y_train)
# Set the y-axis label
```

```
plt.ylabel('Microchip Test 2')
# Set the x-axis label
plt.xlabel('Microchip Test 1')
plt.legend(loc="upper right")
plt.show()
```



3.8 Evaluating regularized logistic regression model

You will use the `predict` function that you implemented above to calculate the accuracy of the regularized logistic regression model on the training set

In [34]:
```
#Compute accuracy on the training set
p = predict(X_mapped, w, b)

print('Train Accuracy: %f'%(np.mean(p == y_train) * 100))
```

```
Train Accuracy: 82.203390
```

Expected Output:

Train Accuracy:~ 80%

Congratulations on completing the final lab of this course! We hope to see you in Course 2 where you will use more advanced learning algorithms such as neural networks and decision trees. Keep learning!

▶ **Please click here if you want to experiment with any of the non-graded code.**