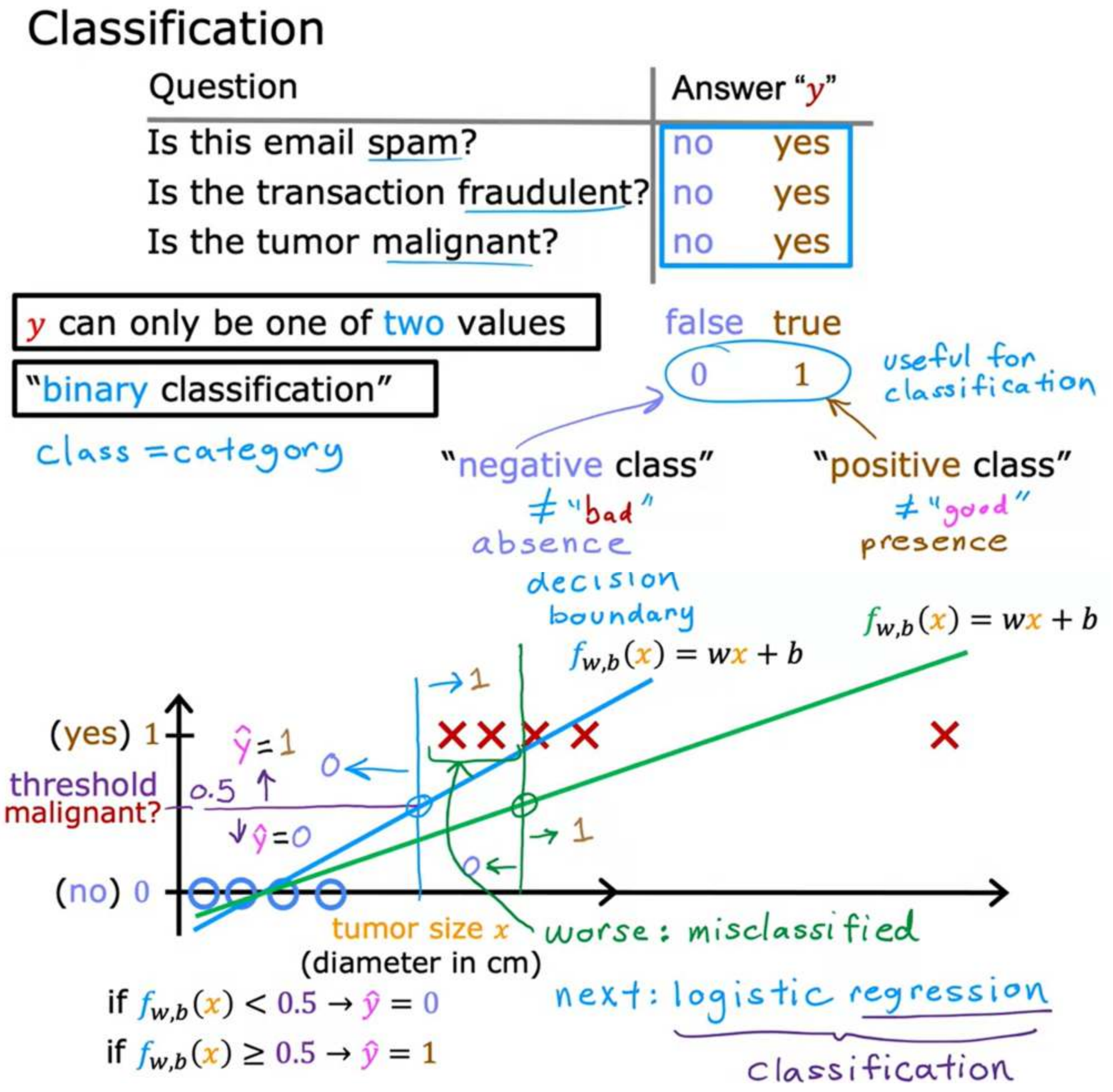
Classification with logistic regression

Motivation

classification is a supervised learning task that involves training a model to categorize or label input data into predefined classes or categories. The goal of classification is to teach the algorithm to recognize patterns and relationships in the input data, enabling it to accurately assign the correct class label to new, unseen instances.



Classification

Question	Answer "y"	
Is this email spam?	no	yes
Is the transaction fraudulent?	no	yes
Is the tumor malignant?	no	yes

y can only be one of two values

"binary classification"

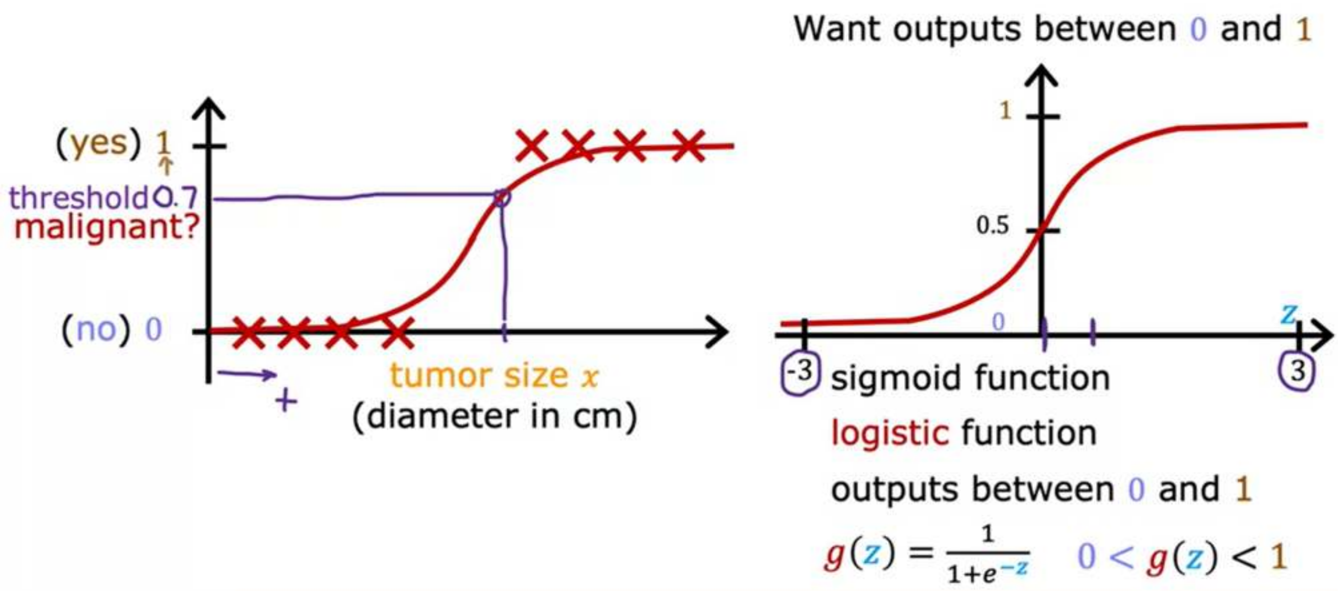
if $f_{w,b}(x) < 0.5 \rightarrow \hat{y} = 0$

if $f_{w,b}(x) \geq 0.5 \rightarrow \hat{y} = 1$

Logistic regression

Logistic regression is the most widely used classification algorithm in the world.

sigmoid function or logistic function is a mathematical function that maps any real-valued number to a value between 0 and 1. It is characterized by its S-shaped curve

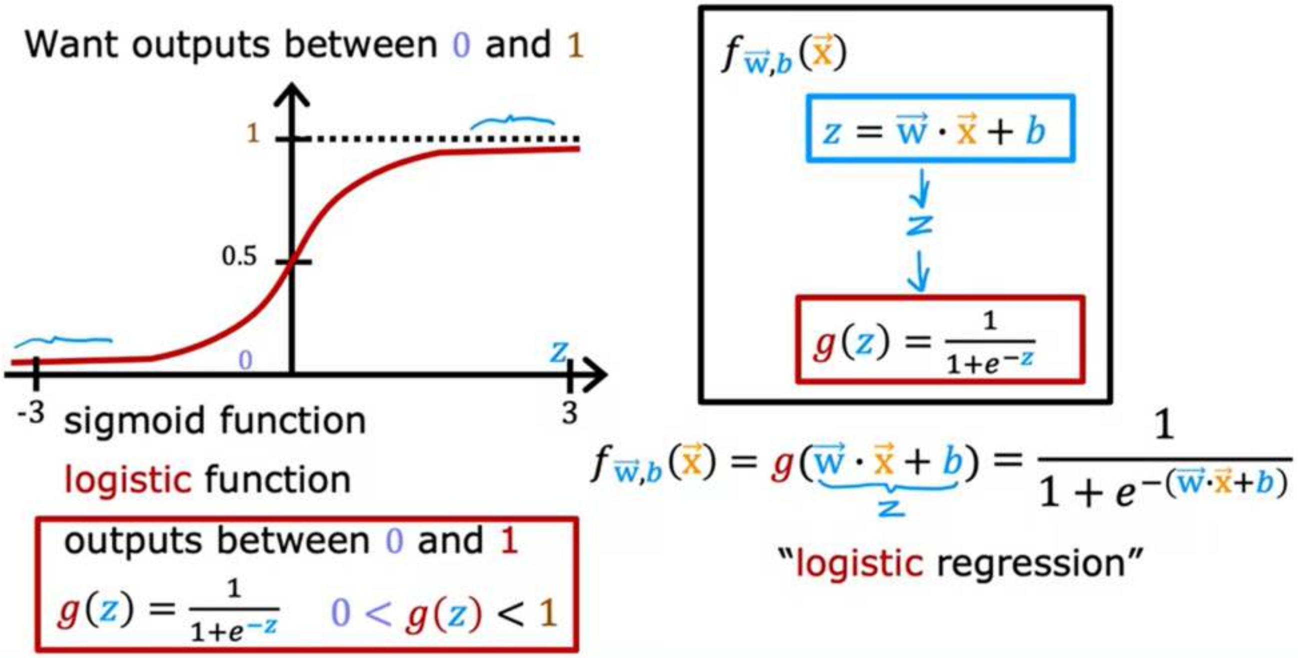


The most commonly used sigmoid function is the logistic function, defined as:

f(x) = 1 / (1 + e^(-x))

where:

- f(x) is the output value between 0 and 1.
- x is the input value.

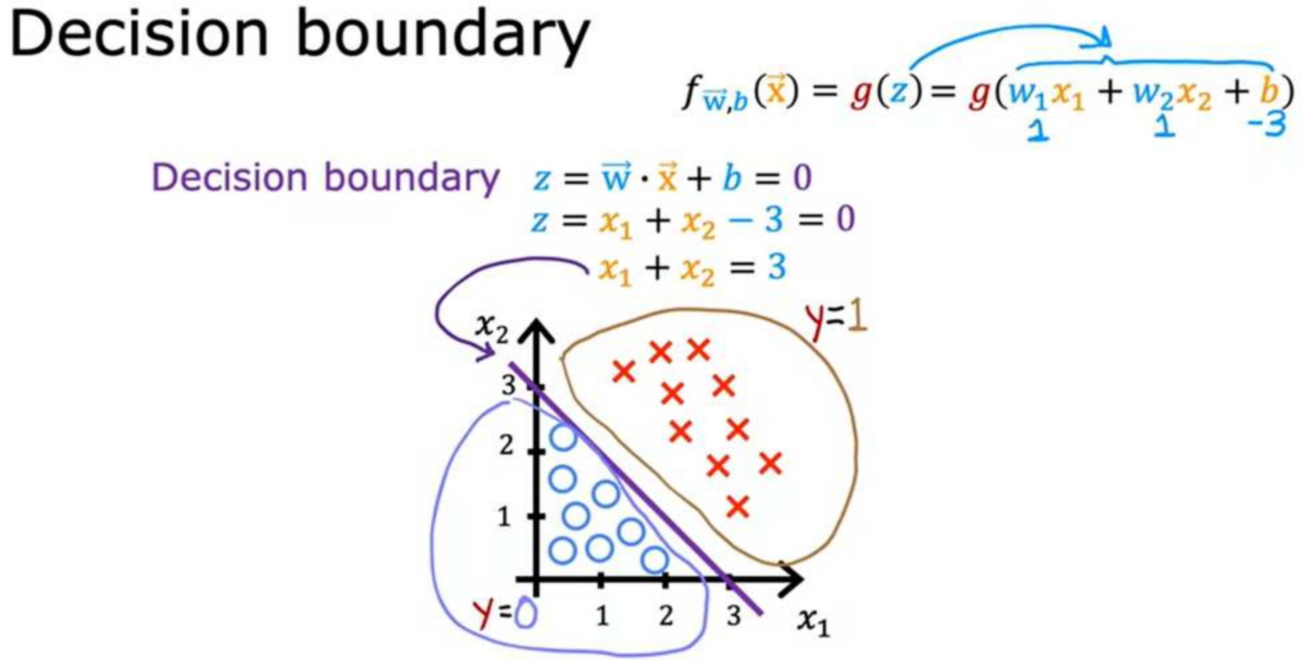

Want outputs between 0 and 1
1
0.5
0
z
-3
3
sigmoid function
logistic function
outputs between 0 and 1
$g(z) = \frac{1}{1+e^{-z}}$ $0 < g(z) < 1$
$f_{\vec{w},b}(\vec{x})$
$z = \vec{w} \cdot \vec{x} + b$
z
$g(z) = \frac{1}{1+e^{-z}}$
$f_{\vec{w},b}(\vec{x}) = g(\underbrace{\vec{w} \cdot \vec{x} + b}_{z}) = \frac{1}{1+e^{-(\vec{w} \cdot \vec{x} + b)}}$
"logistic regression"

If z is a larg negative number then g(z) is near zero

If z is a larg positive number then g(z) is near one

Decision boundary

ry is a critical concept used to classify data points into different classes.



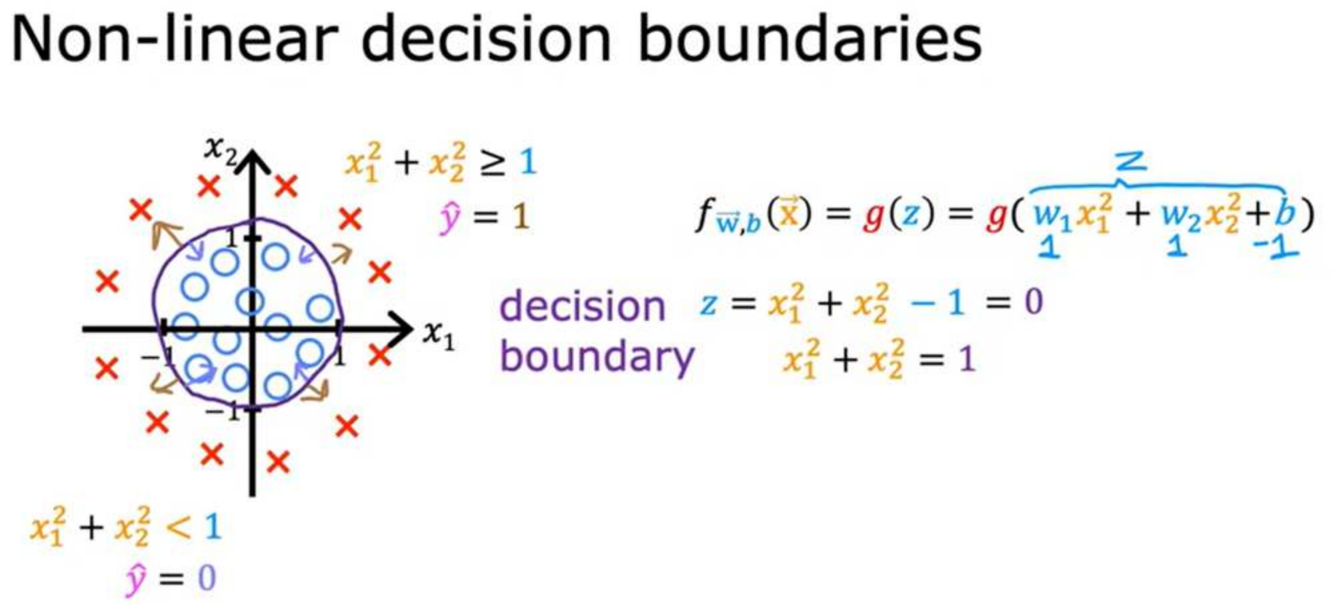
Decision boundary

$$f_{\vec{w},b}(\vec{x}) = g(z) = g(\overbrace{\underset{1}{w_1} x_1 + \underset{1}{w_2} x_2 + \underset{-3}{b}})$$

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

$z = x_1 + x_2 - 3 = 0$

$x_1 + x_2 = 3$



Non-linear decision boundaries

$x_1^2 + x_2^2 \geq 1$

$\hat{y} = 1$

$$f_{\vec{w},b}(\vec{x}) = g(z) = g(\overbrace{\underset{1}{w_1} x_1^2 + \underset{1}{w_2} x_2^2 + \underset{-1}{b}}^{z})$$

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

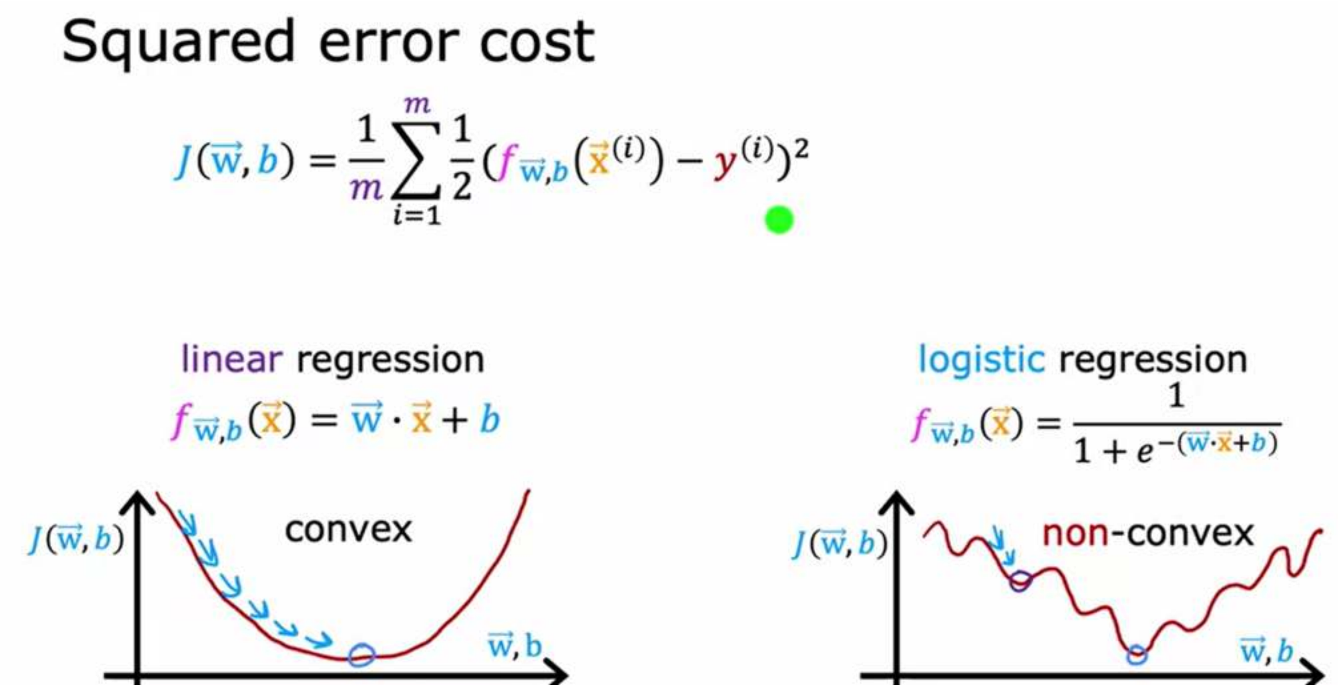
$x_1^2 + x_2^2 = 1$

$x_1^2 + x_2^2 < 1$

$\hat{y} = 0$

Cost function for logistic regression

Squared error cost function is not a good choice bcuz of the graph (non-convex)



Squared error cost

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

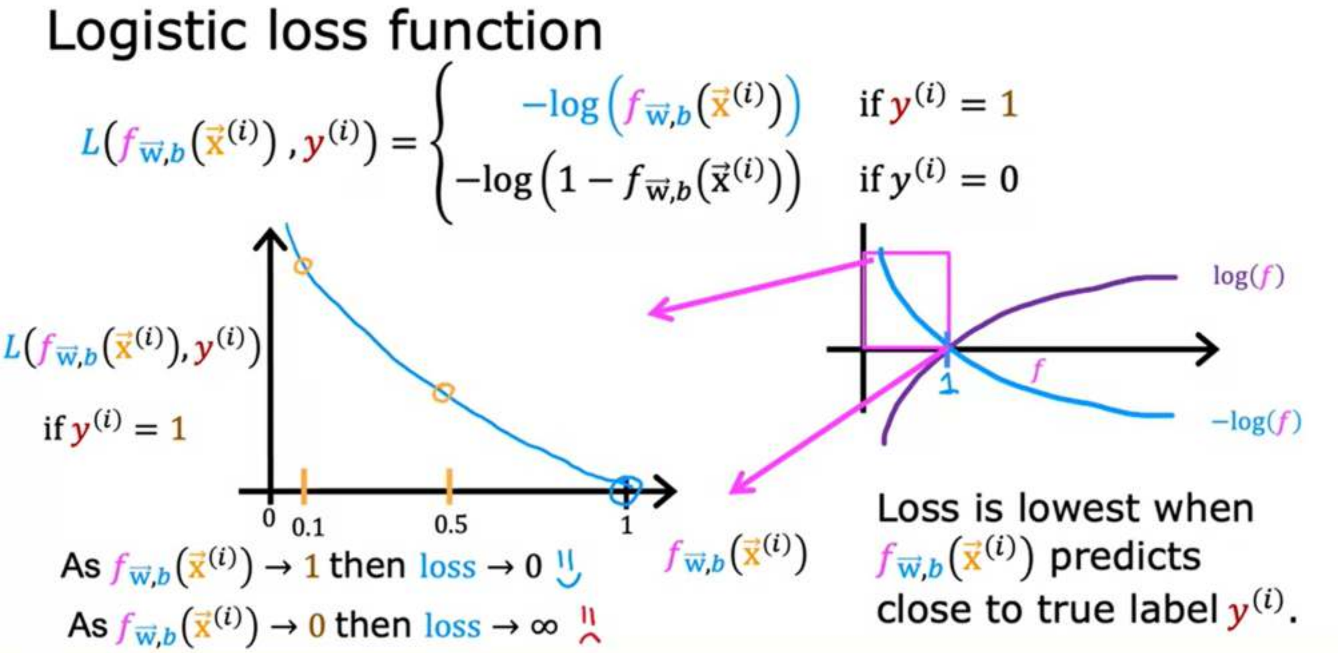
linear regression

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

logistic regression

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

INSTEAD, we use loss function



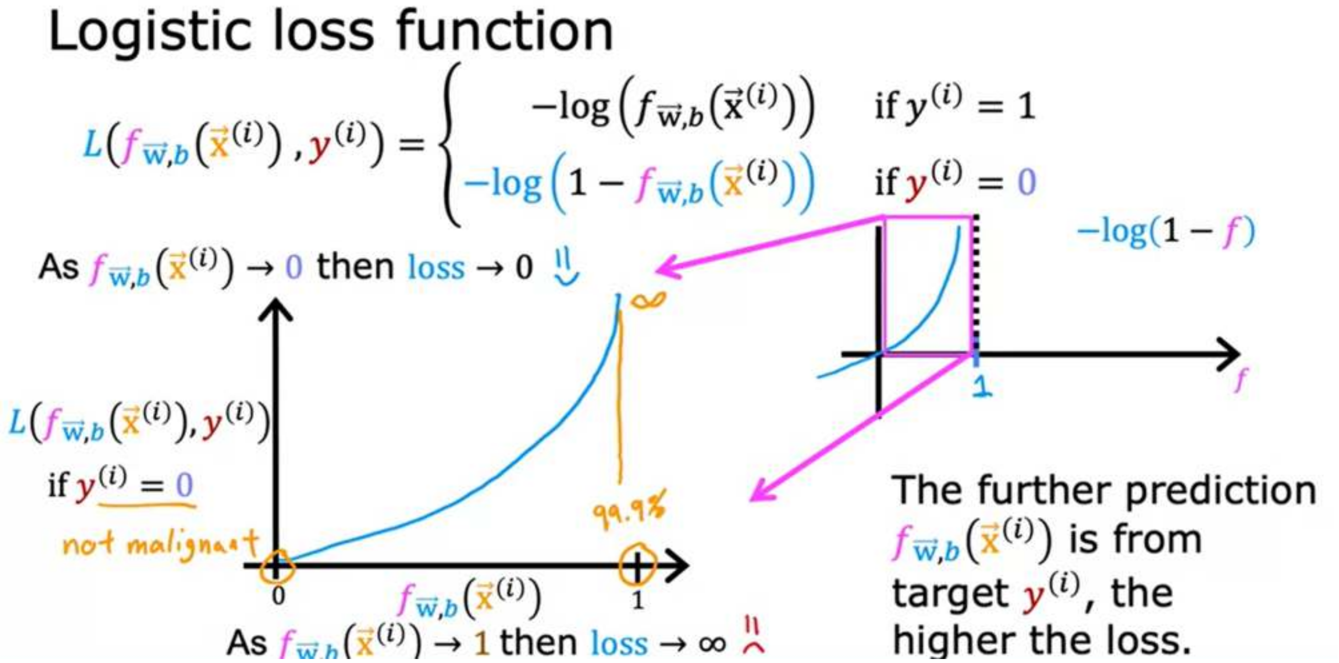
Logistic loss function

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

As $f_{\vec{w},b}(\vec{x}^{(i)}) \to 1$ then loss $\to 0$

As $f_{\vec{w},b}(\vec{x}^{(i)}) \to 0$ then loss $\to \infty$

Loss is lowest when $f_{\vec{w},b}(\vec{x}^{(i)})$ predicts close to true label $y^{(i)}$.



Logistic loss function

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

As $f_{\vec{w},b}(\vec{x}^{(i)}) \to 0$ then loss $\to 0$

As $f_{\vec{w},b}(\vec{x}^{(i)}) \to 1$ then loss $\to \infty$

The further prediction $f_{\vec{w},b}(\vec{x}^{(i)})$ is from target $y^{(i)}$, the higher the loss.

Lab: Logistic Regression, Logistic Loss

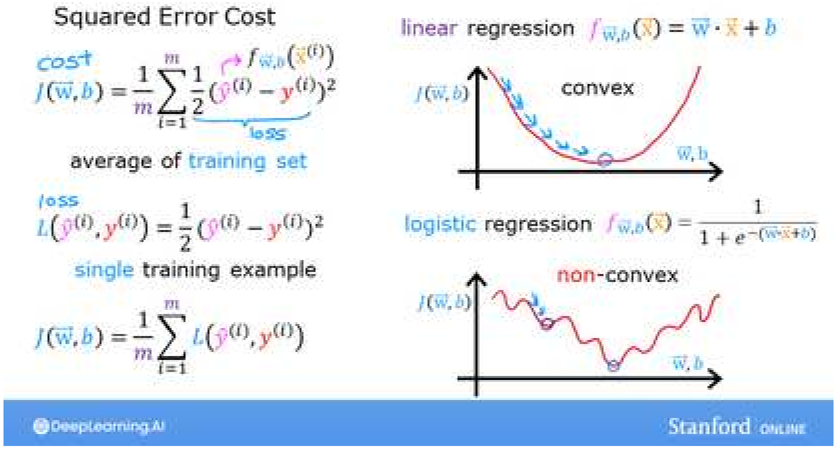
In this ungraded lab, you will:

- explore the reason the squared error loss is not appropriate for logistic regression
- explore the logistic loss function

In [1]:

```
import numpy as np
%matplotlib widget
import matplotlib.pyplot as plt
from plt_logistic_loss import  plt_logistic_cost, plt_two_logistic_loss_curves, plt_simp
from plt_logistic_loss import soup_bowl, plt_logistic_squared_error
plt.style.use('./deeplearning.mplstyle')
```

Loading [MathJax]/extensions/Safe.js



Squared Error Cost

cost

$$J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}\underbrace{\frac{1}{2}(\overset{f_{\vec{w},b}(\vec{x}^{(i)})}{\hat{y}^{(i)}} - y^{(i)})^2}_{loss}$$

average of training set

loss

$$L(\hat{y}^{(i)},y^{(i)}) = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2$$

single training example

$$J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)},y^{(i)})$$

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

convex

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

non-convex

Squared error for logistic regression?

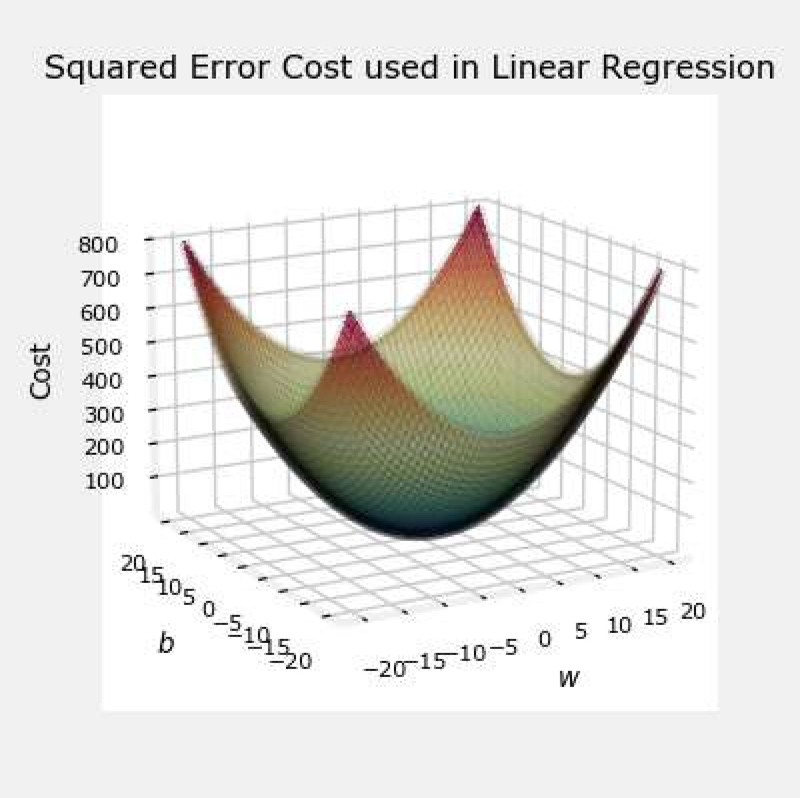
Recall for **Linear** Regression we have used the **squared error cost function**: The equation for the squared error cost with one variable is: $$J(w,b) = \frac{1}{2m} \sum\limits_{i = 0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)})^2 \tag{1}$$

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

Recall, the squared error cost had the nice property that following the derivative of the cost leads to the minimum.

```
In [2]: soup_bowl()
```

Figure



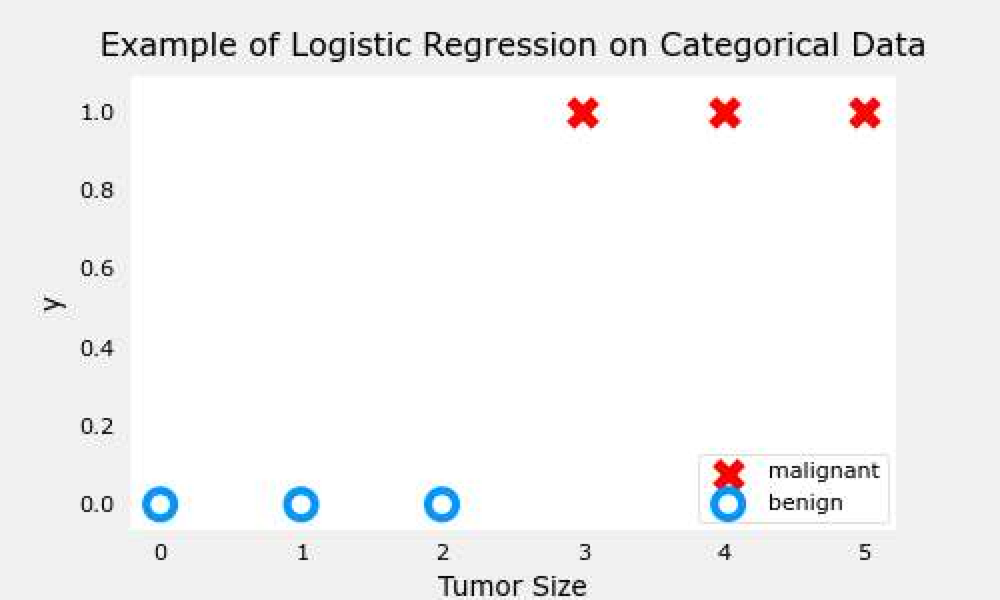
This cost function worked well for linear regression, it is natural to consider it for logistic regression as well. However, as the slide above points out, $f_{wb}(x)$ now has a non-linear component, the sigmoid function:

$f_{w,b}(x^{(i)}) = sigmoid(wx^{(i)} + b)$. Let's try a squared error cost on the example from an earlier lab, now including the sigmoid.

Here is our training data:

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

Figure

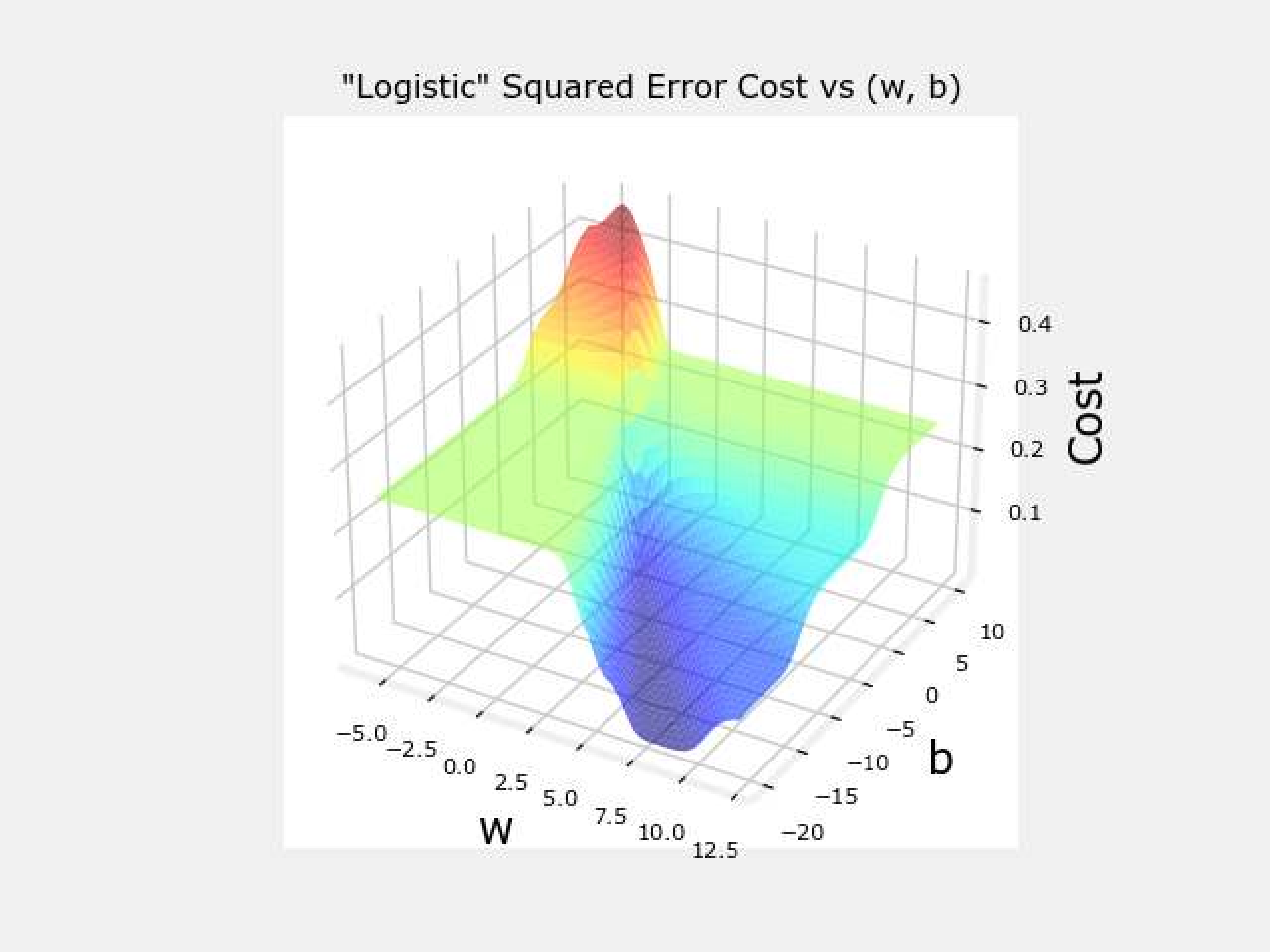


Now, let's get a surface plot of the cost using a *squared error cost*: $$J(w,b) = \frac{1}{2m} \sum\limits_{i = 0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)})^2 $$

where $$f_{w,b}(x^{(i)}) = sigmoid(wx^{(i)} + b)$$

In [4]:
```python
plt.close('all')
plt_logistic_squared_error(x_train,y_train)
plt.show()
```

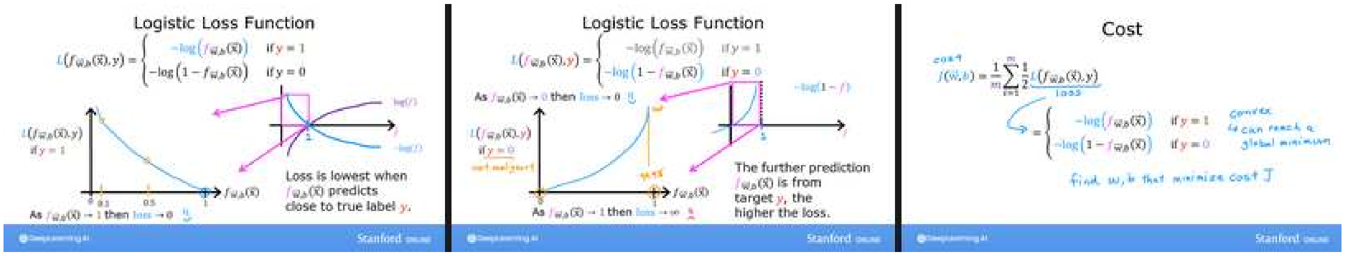
Figure



While this produces a pretty interesting plot, the surface above not nearly as smooth as the 'soup bowl' from linear regression!

Logistic regression requires a cost function more suitable to its non-linear nature. This starts with a Loss function. This is described below.

Logistic Loss Function


Logistic Loss Function

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

As $f_{\vec{w},b}(\vec{x}) \to 1$ then loss $\to 0$

Loss is lowest when $f_{\vec{w},b}(\vec{x})$ predicts close to true label y.

Logistic Loss Function

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

As $f_{\vec{w},b}(\vec{x}) \to 0$ then loss $\to 0$

As $f_{\vec{w},b}(\vec{x}) \to 1$ then loss $\to \infty$

The further prediction $f_{\vec{w},b}(\vec{x})$ is from target y, the higher the loss.

Cost

$$\overset{\text{cost}}{J(\vec{w},b)} = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\frac{1}{2}L(f_{\vec{w},b}(\vec{x}),y)}_{\text{loss}}$$

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

convex
can reach a global minimum

find w, b that minimize cost J

Logistic Regression uses a loss function more suited to the task of categorization where the target is 0 or 1 rather than any number.

> **Definition Note:** In this course, these definitions are used:
>
> **Loss** is a measure of the difference of a single example to its target value while the
>
> **Cost** is a measure of the losses over the training set

This is defined:

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

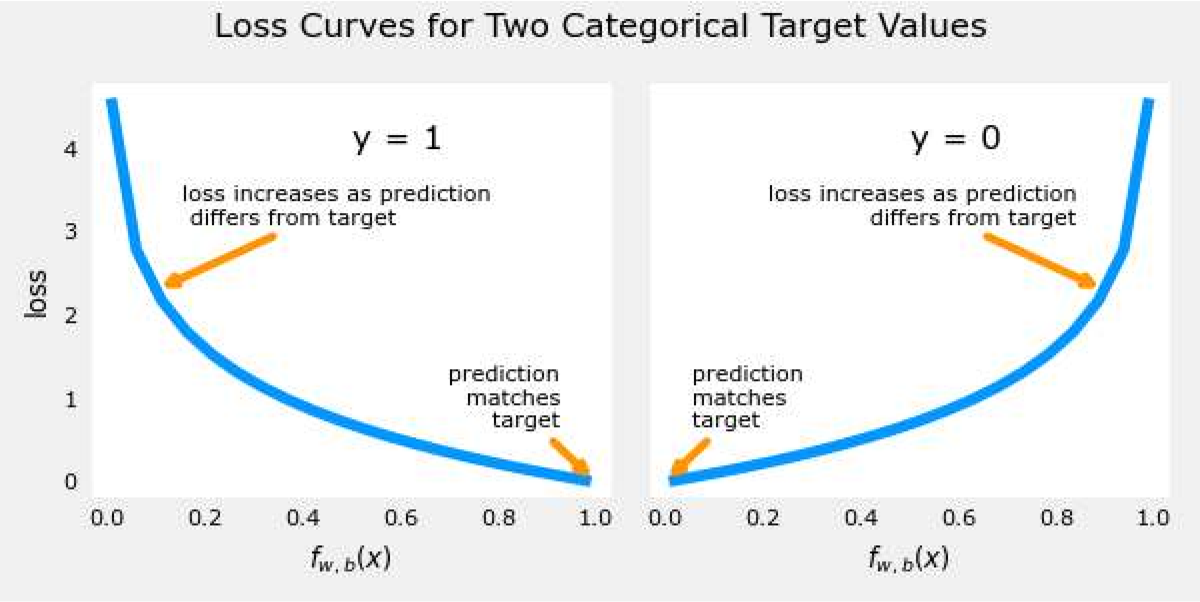
\begin{equation} loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)}) = \begin{cases} - \log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)} \right) \right) & \text{if $y^{(i)}=1$}\\ - \log \left(1 - f_{\mathbf{w},b}\left(\mathbf{x}^{(i)} \right) \right) & \text{if $y^{(i)}=0$} \end{cases} \end{equation}

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

The defining feature of this loss function is the fact that it uses two separate curves. One for the case when the target is zero or ($y=0$) and another for when the target is one ($y=1$). Combined, these curves provide the behavior useful for a loss function, namely, being zero when the prediction matches the target and rapidly increasing in value as the prediction differs from the target. Consider the curves below:

```
In [5]:  plt_two_logistic_loss_curves()
```

Figure



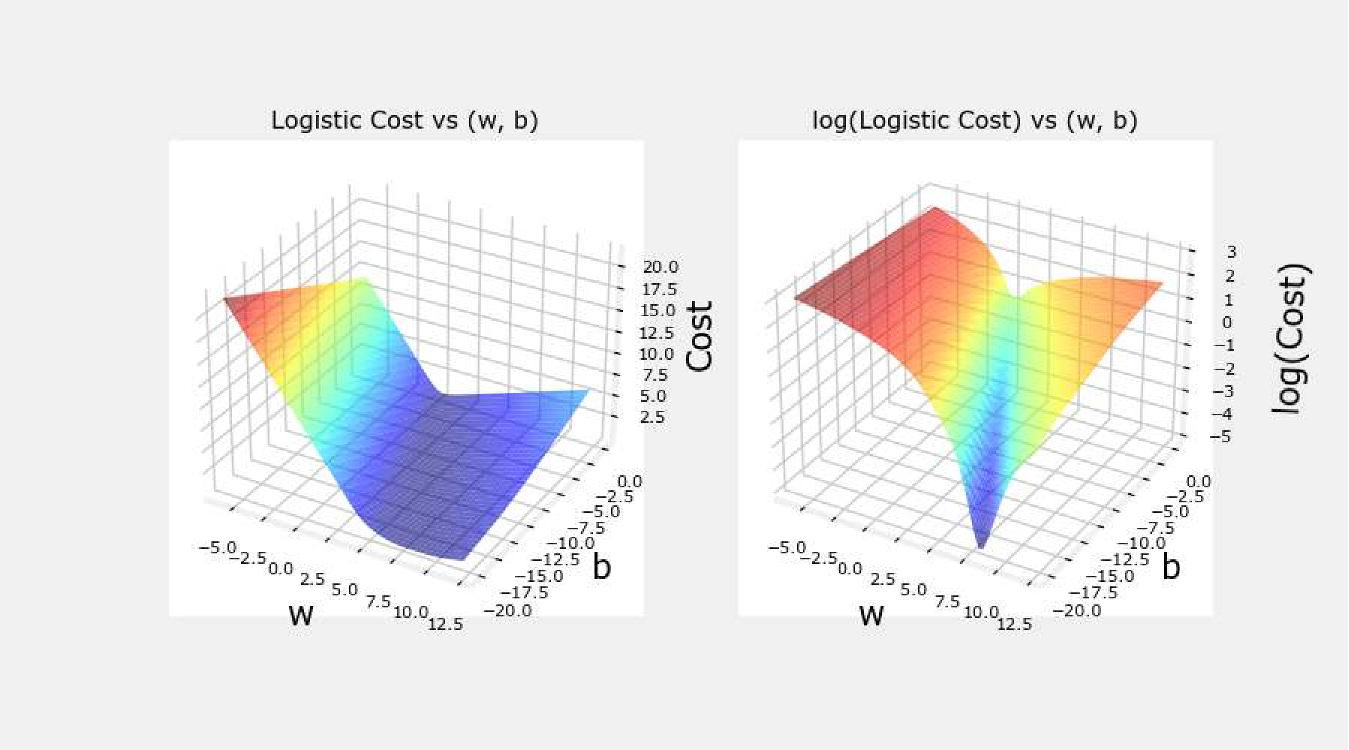
The loss function above can be rewritten to be easier to implement. $$loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)}) = (-y^{(i)} \log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)} \right) \right) - \left(1 - y^{(i)}\right) \log \left(1 - f_{\mathbf{w},b}\left(\mathbf{x}^{(i)} \right) \right)$$

This is a rather formidable-looking equation. It is less daunting when you consider $y^{(i)}$ can have only two values, 0 and 1. One can then consider the equation in two pieces:
when $ y^{(i)} = 0$, the left-hand term is eliminated: $$ \begin{align} loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), 0) &= (-(0) \log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)} \right) \right) - \left(1 - 0\right) \log \left(1 - f_{\mathbf{w},b}\left(\mathbf{x}^{(i)} \right) \right) \\ &= -\log \left(1 - f_{\mathbf{w},b}\left(\mathbf{x}^{(i)} \right) \right) \end{align} $$ and when $ y^{(i)} = 1$, the right-hand term is eliminated: $$ \begin{align} loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), 1) &= (-(1) \log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)} \right) \right) - \left(1 - 1\right) \log \left(1 - f_{\mathbf{w},b}\left(\mathbf{x}^{(i)} \right) \right)\\ &= -\log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)} \right) \right) \end{align} $$

OK, with this new logistic loss function, a cost function can be produced that incorporates the loss from all the examples. This will be the topic of the next lab. For now, let's take a look at the cost vs parameters curve for the simple example we considered above:

In [6]:
```
plt.close('all')
cst = plt_logistic_cost(x_train,y_train)
```

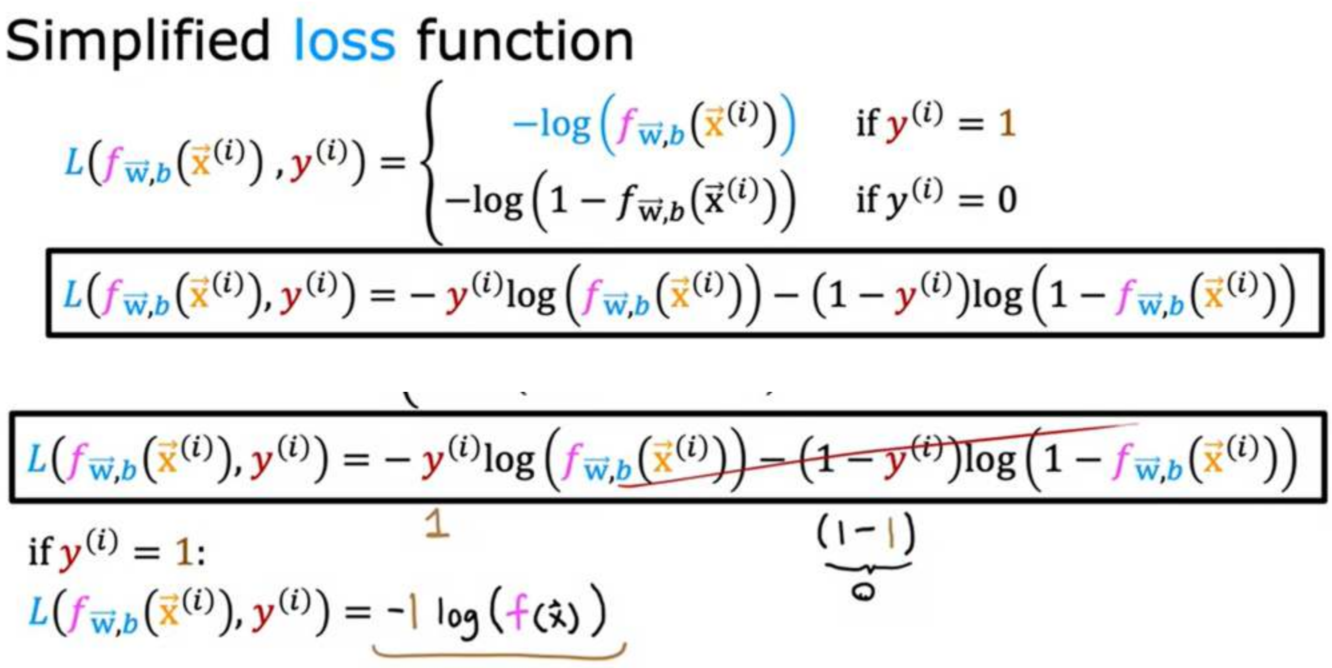
Figure



This curve is well suited to gradient descent!

It does not have plateaus, local minima, or discontinuities. Note, it is not a bowl as in the case of squared error. Both the cost and the log of the cost are plotted to illuminate the fact that the curve, when the cost is small, has a slope and continues to decline. Reminder: you can rotate the above plots using your mouse.

Simplified Cost Function for Logistic Regression



Simplified loss function

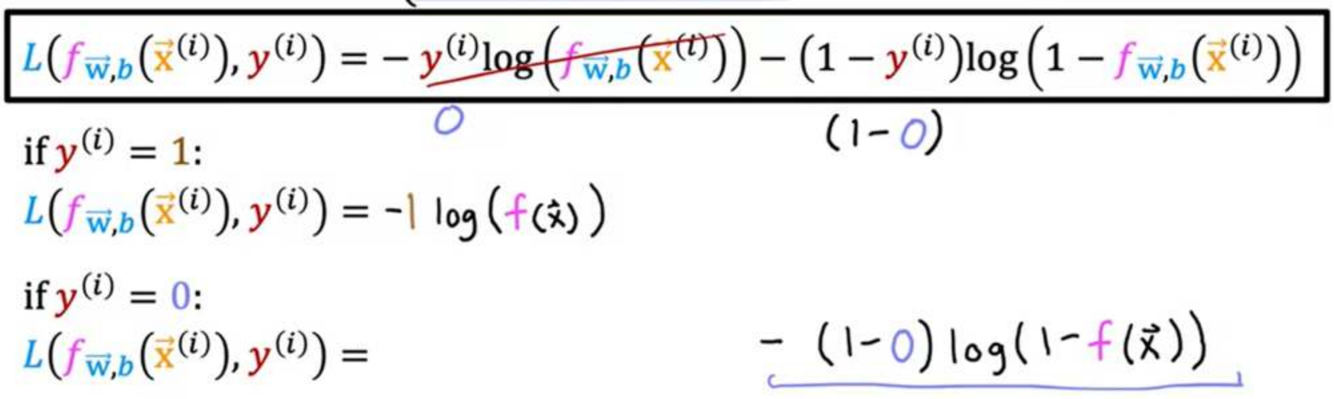
$$L\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right), y^{(i)}\right) = \begin{cases} -\log\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right)\right) & \text{if } y^{(i)} = 1 \\ -\log\left(1 - f_{\vec{w},b}\left(\vec{x}^{(i)}\right)\right) & \text{if } y^{(i)} = 0 \end{cases}$$

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

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

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

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



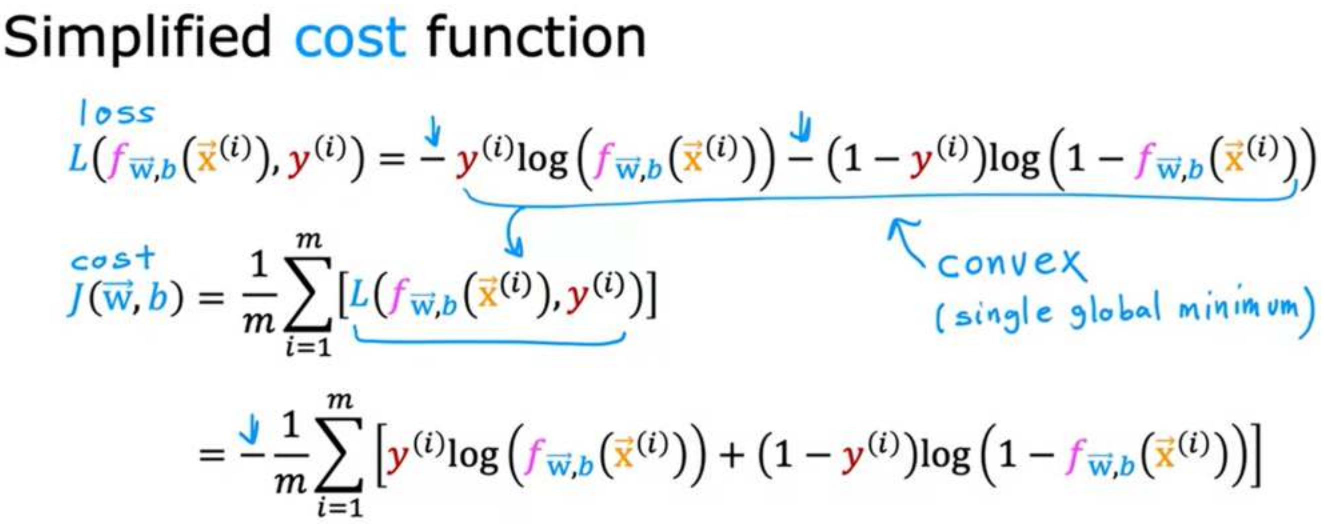
$$\boxed{L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}) = -\underset{0}{\cancel{y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right)}} - \underset{(1-0)}{(1-y^{(i)})}\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right)}$$

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

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

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

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



Simplified cost function

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

convex (single global minimum)

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

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

Lab: Cost Function for Logistic Regression

Goals

In this lab, you will:

- examine the implementation and utilize the cost function for logistic regression.

In [7]:

```
import numpy as np
%matplotlib widget
import matplotlib.pyplot as plt
from lab_utils_common import  plot_data, sigmoid, dlc
plt.style.use('deeplearning.mplstyle')
```

Dataset

Let's start with the same dataset as was used in the decision boundary lab.

In [8]:

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

We will 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.

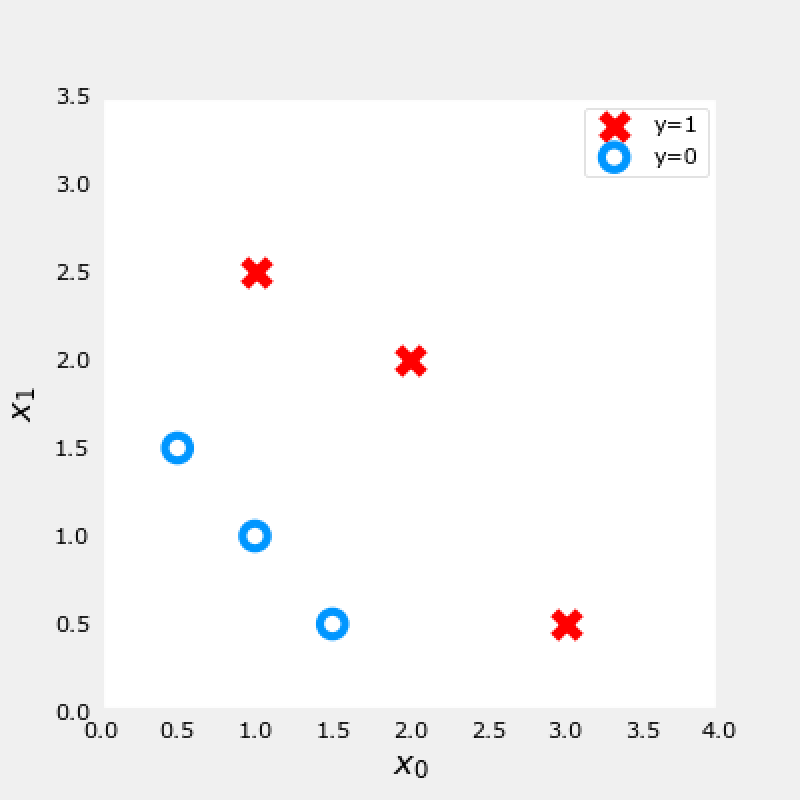
In [9]:

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

Loading [MathJax]/extensions/Safe.js

```
# Set both axes to be from 0-4
ax.axis([0, 4, 0, 3.5])
ax.set_ylabel('$x_1$', fontsize=12)
ax.set_xlabel('$x_0$', fontsize=12)
plt.show()
```

Figure



Cost function

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

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

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

where

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

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

- where m is the number of training examples in the data set and: $$ \begin{align} f_{\mathbf{w},b}(\mathbf{x^{(i)}}) &= g(z^{(i)})\tag{3} \\ z^{(i)} &= \mathbf{w} \cdot \mathbf{x}^{(i)}+ b\tag{4} \\ g(z^{(i)}) &= \frac{1}{1+e^{-z^{(i)}}}\tag{5} \end{align} $$

Code Description

Loading [MathJax]/extensions/Safe.js

The algorithm for `compute_cost_logistic` loops over all the examples calculating the loss for each example and accumulating the total.

Note that the variables X and y are not scalar values but matrices of shape ($m, n$) and ($m$,) respectively, where $n$ is the number of features and $m$ is the number of training examples.

In [12]:

```
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

    """

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

    return g
```

In [13]:

```
def compute_cost_logistic(X, y, w, b):
    """
    Computes cost

    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:
      cost (scalar): cost
    """

    m = X.shape[0]
    cost = 0.0
    for i in range(m):
        z_i = np.dot(X[i],w) + b
        f_wb_i = sigmoid(z_i)
        cost +=  -y[i]*np.log(f_wb_i) - (1-y[i])*np.log(1-f_wb_i)

    cost = cost / m
    return cost
```

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

In [14]:

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

```
0.36686678640551745
```

Expected output: 0.3668667864055175

Example

Now, let's see what the cost function output is for a different value of $w$.

Loading [MathJax]/extensions/Safe.js

- In a previous lab, you plotted the decision boundary for $b = -3, w_0 = 1, w_1 = 1$. That is, you had `b = -3, w = np.array([1,1])`.
- Let's say you want to see if $b = -4, w_0 = 1, w_1 = 1$, or `b = -4, w = np.array([1,1])` provides a better model.

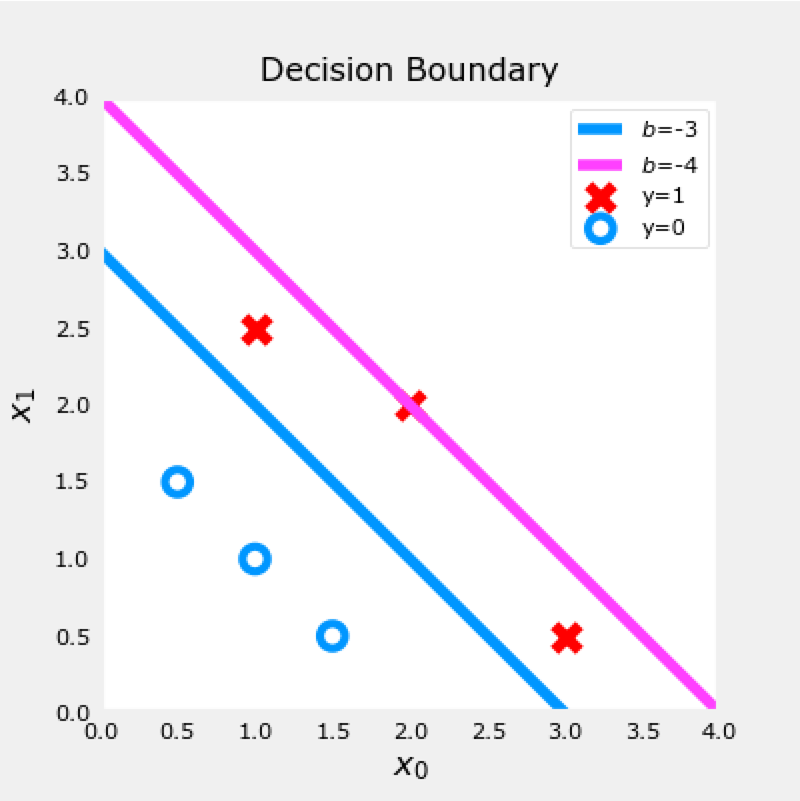
Let's first plot the decision boundary for these two different $b$ values to see which one fits the data better.

- For $b = -3, w_0 = 1, w_1 = 1$, we'll plot $-3 + x_0+x_1 = 0$ (shown in blue)
- For $b = -4, w_0 = 1, w_1 = 1$, we'll plot $-4 + x_0+x_1 = 0$ (shown in magenta)

In [15]:

```
import matplotlib.pyplot as plt

# Choose values between 0 and 6
x0 = np.arange(0,6)

# Plot the two decision boundaries
x1 = 3 - x0
x1_other = 4 - x0

fig,ax = plt.subplots(1, 1, figsize=(4,4))
# Plot the decision boundary
ax.plot(x0,x1, c=dlc["dlblue"], label="$b$=-3")
ax.plot(x0,x1_other, c=dlc["dlmagenta"], label="$b$=-4")
ax.axis([0, 4, 0, 4])

# Plot the original data
plot_data(X_train,y_train,ax)
ax.axis([0, 4, 0, 4])
ax.set_ylabel('$x_1$', fontsize=12)
ax.set_xlabel('$x_0$', fontsize=12)
plt.legend(loc="upper right")
plt.title("Decision Boundary")
plt.show()
```

Figure



You can see from this plot that `b = -4, w = np.array([1,1])` is a worse model for the training data. Let's see if the cost function implementation reflects this.

In [16]:

```
w_array1 = np.array([1,1])
b_1 = -3
w_array2 = np.array([1,1])
b_2 = -4

print("Cost for b = -3 : ", compute_cost_logistic(X_train, y_train, w_array1, b_1))
print("Cost for b = -4 : ", compute_cost_logistic(X_train, y_train, w_array2, b_2))
```

```
Cost for b = -3 :  0.36686678640551745
Cost for b = -4 :  0.5036808636748461
```

Expected output

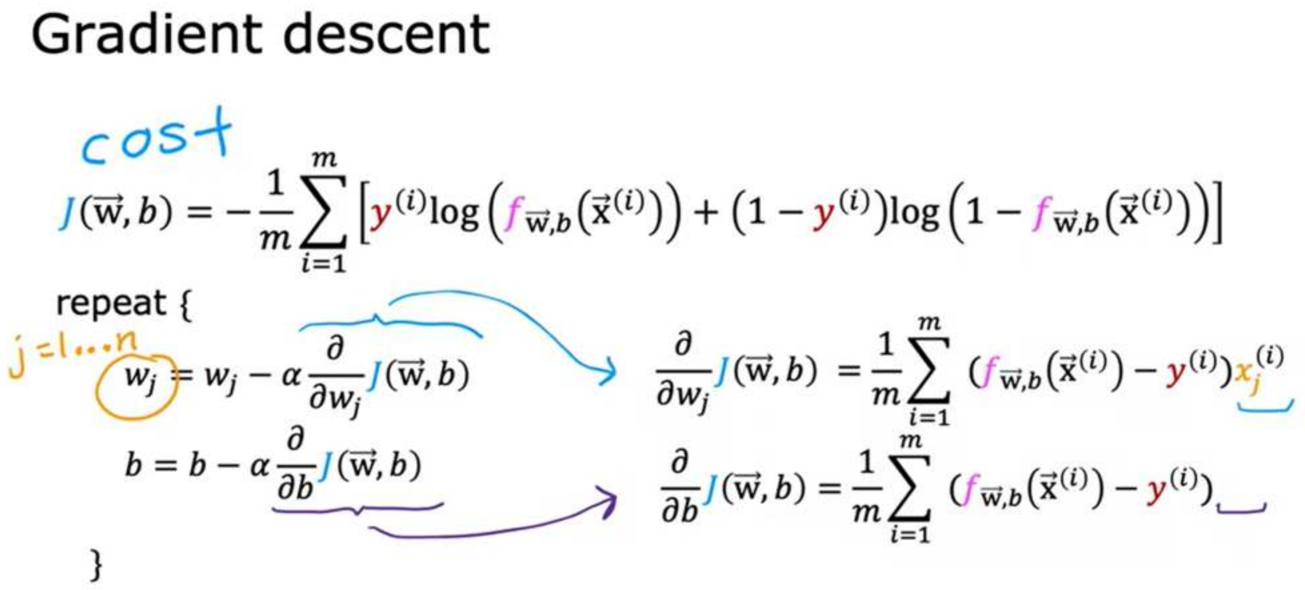
Cost for b = -3 : 0.3668667864055175

Cost for b = -4 : 0.5036808636748461

You can see the cost function behaves as expected and the cost for `b = -4, w = np.array([1,1])` is indeed higher than the cost for `b = -3, w = np.array([1,1])`

Gradient Descent Implementation

Loading [MathJax]/extensions/Safe.js



Gradient descent

cost

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

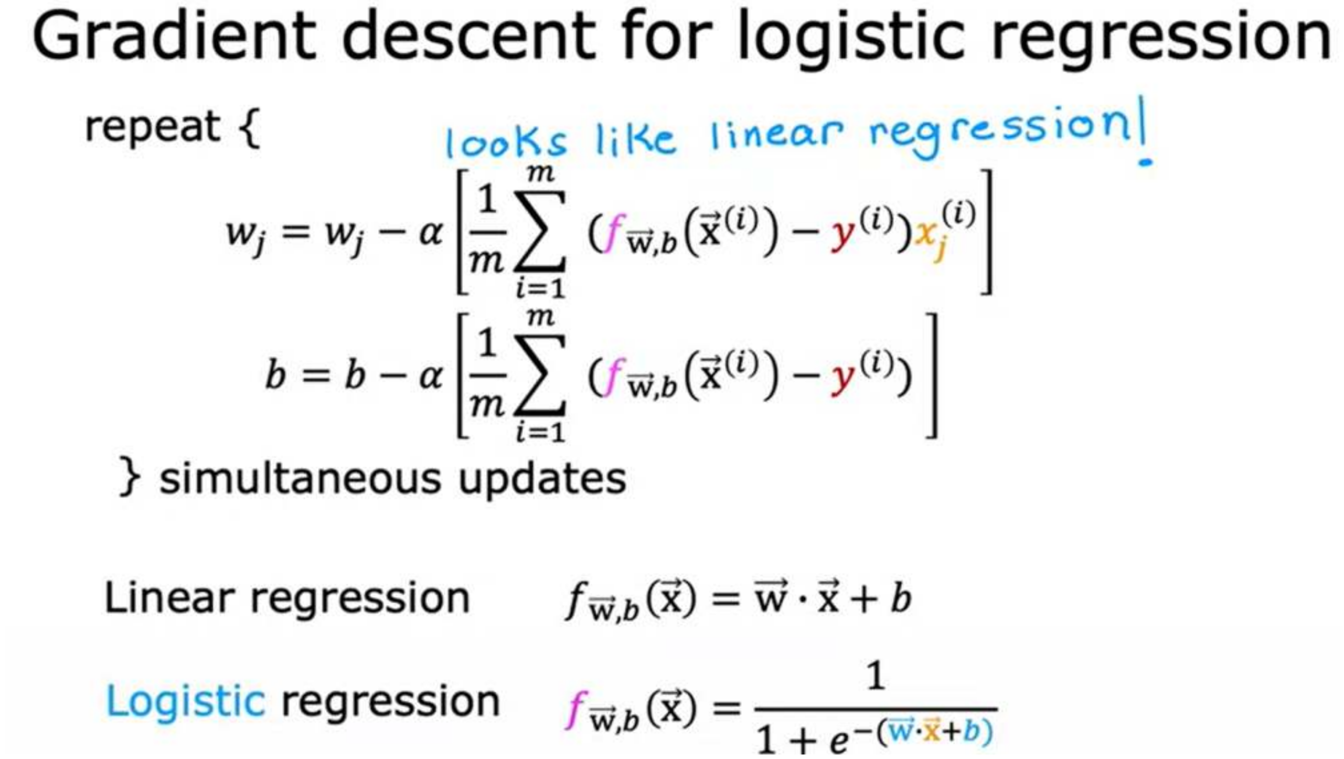
repeat {

j=1...n

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

$$b = b - \alpha \underbrace{\frac{\partial}{\partial b}J(\vec{w},b)}_{} \longrightarrow \frac{\partial}{\partial b}J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})\underbrace{\quad}_{}$$

}



Gradient descent for logistic regression

repeat { looks like linear regression!

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

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

} simultaneous updates

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

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

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 [17]:

```
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, sigmoid, compute_cost_logi
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 [18]:

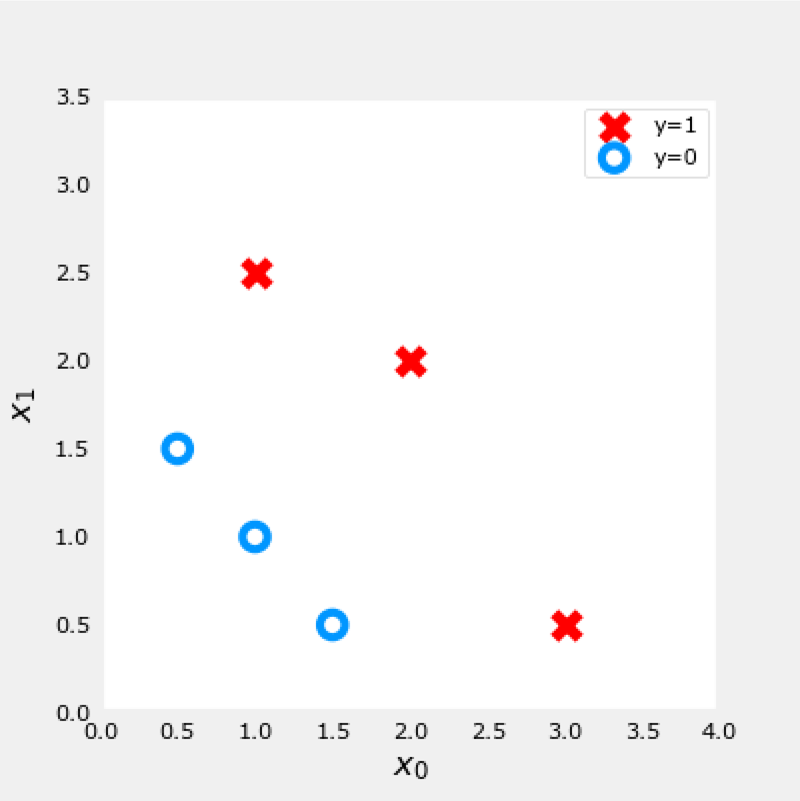
```
X_train = np.array([[0.5, 1.5], [1,1], [1.5, 0.5], [3, 0.5], [2, 2], [1, 2.5]])
y_train = np.array([0, 0, 0, 1, 1, 1])
```

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.

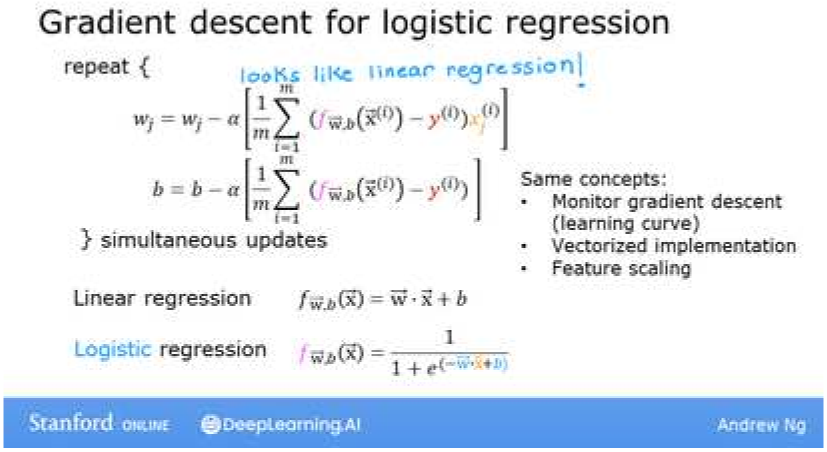
In [19]:

```
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()
```

Figure



Logistic Gradient Descent



Gradient descent for logistic regression

repeat {

looks like linear regression!

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

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

} simultaneous updates

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

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

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

Recall the gradient descent algorithm utilizes the gradient calculation: $$\begin{align*} &\text{repeat until convergence:} \; \lbrace \\ & \; \; \;w_j = w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \tag{1} \; & \text{for j := 0..n-1} \\ & \; \; \; \; \;b = b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \\ &\rbrace \end{align*}$$

Where each iteration performs simultaneous updates on $w_j$ for all $j$, where $$\begin{align*} \frac{\partial J(\mathbf{w},b)}{\partial w_j} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})x_{j}^{(i)} \tag{2} \\ \frac{\partial J(\mathbf{w},b)}{\partial b} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) \tag{3} \end{align*}$$

- 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)
  - e error to `dj_db` (equation 3 above)

Loading [MathJax]/extensions/Safe.js

- 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]`

In [20]:

```
def compute_gradient_logistic(X, y, w, b):
    """
    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
    Returns
      dj_dw (ndarray (n,)): The gradient of the cost w.r.t. the parameters w.
      dj_db (scalar)      : The gradient of the cost w.r.t. the parameter b.
    """
    m,n = X.shape
    dj_dw = np.zeros((n,))                           #(n,)
    dj_db = 0.

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

    return dj_db, dj_dw
```

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

In [21]:

```
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_tmp, b_tmp)
print(f"dj_db: {dj_db_tmp}" )
print(f"dj_dw: {dj_dw_tmp.tolist()}" )
```

```
dj_db: 0.49861806546328574
dj_dw: [0.498333393278696, 0.49883942983996693]
```

Expected output

```
dj_db: 0.49861806546328574
dj_dw: [0.498333393278696, 0.49883942983996693]
```

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 [22]:

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

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

        # Update Parameters using w, b, alpha and gradient
        w = w - alpha * dj_dw               
        b = b - alpha * dj_db               
      
        # 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 iterations 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 [23]:
```
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}")
```

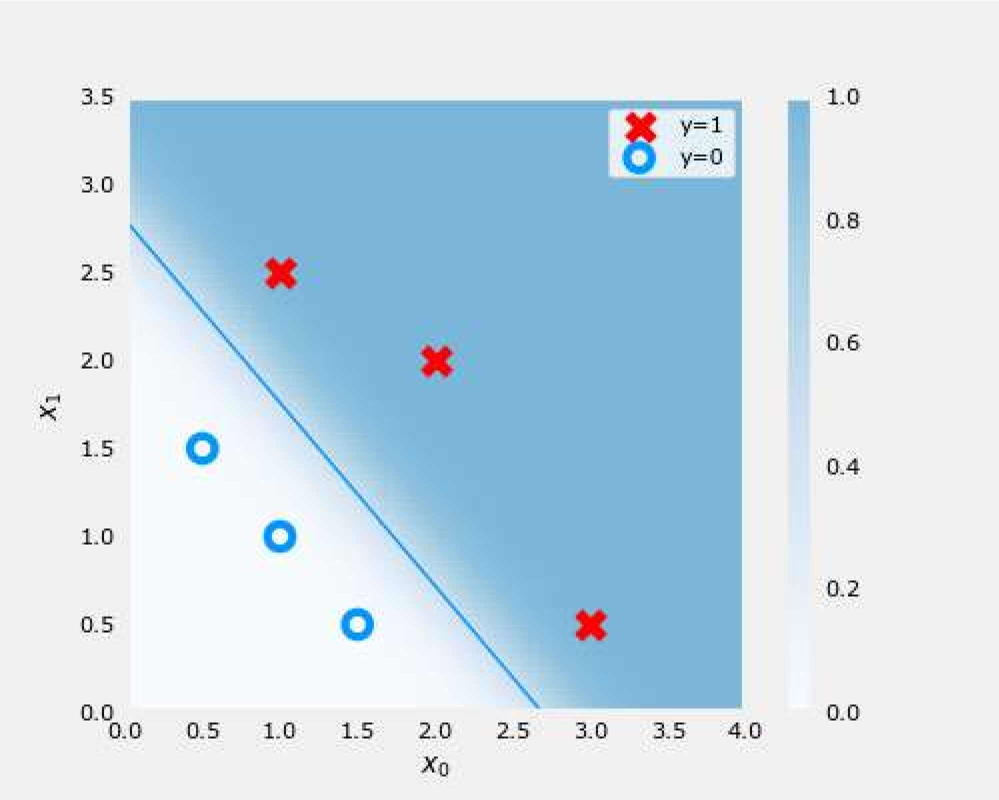
```
Iteration    0: Cost 0.684610468560574   
Iteration 1000: Cost 0.1590977666870457   
Iteration 2000: Cost 0.08460064176930078   
Iteration 3000: Cost 0.05705327279402531   
Iteration 4000: Cost 0.04290759421682   
Iteration 5000: Cost 0.03433847729884557   
Iteration 6000: Cost 0.02860379802212006   
Iteration 7000: Cost 0.02450156960879306   
Iteration 8000: Cost 0.02142370332569295   
Iteration 9000: Cost 0.019030137124109114   

updated parameters: w:[5.28 5.08], b:-14.222409982019837
```

Let's plot the results of gradient descent:

In [24]:
```
fig,ax = plt.subplots(1,1,figsize=(5,4))
# plot the probability 
plt_prob(ax, w_out, b_out)

# Plot the original data
Loading [MathJax]/extensions/Safe.js .(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()
```

Figure



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 [25]:

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

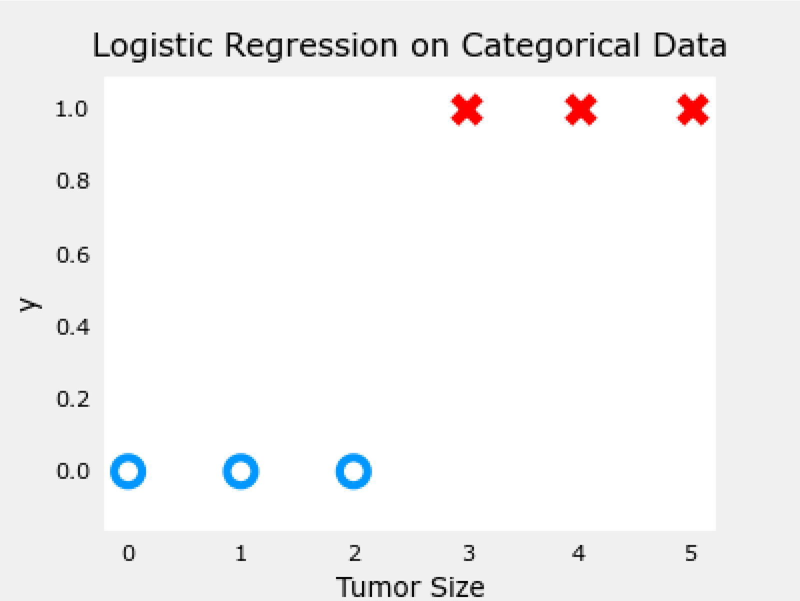
As before, we'll use a helper function to plot this data. The data points with label $y=1$ are shown as red crosses, while the data points with label $y=0$ are shown as blue circles.

In [26]:

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

Loading [MathJax]/extensions/Safe.js

Figure



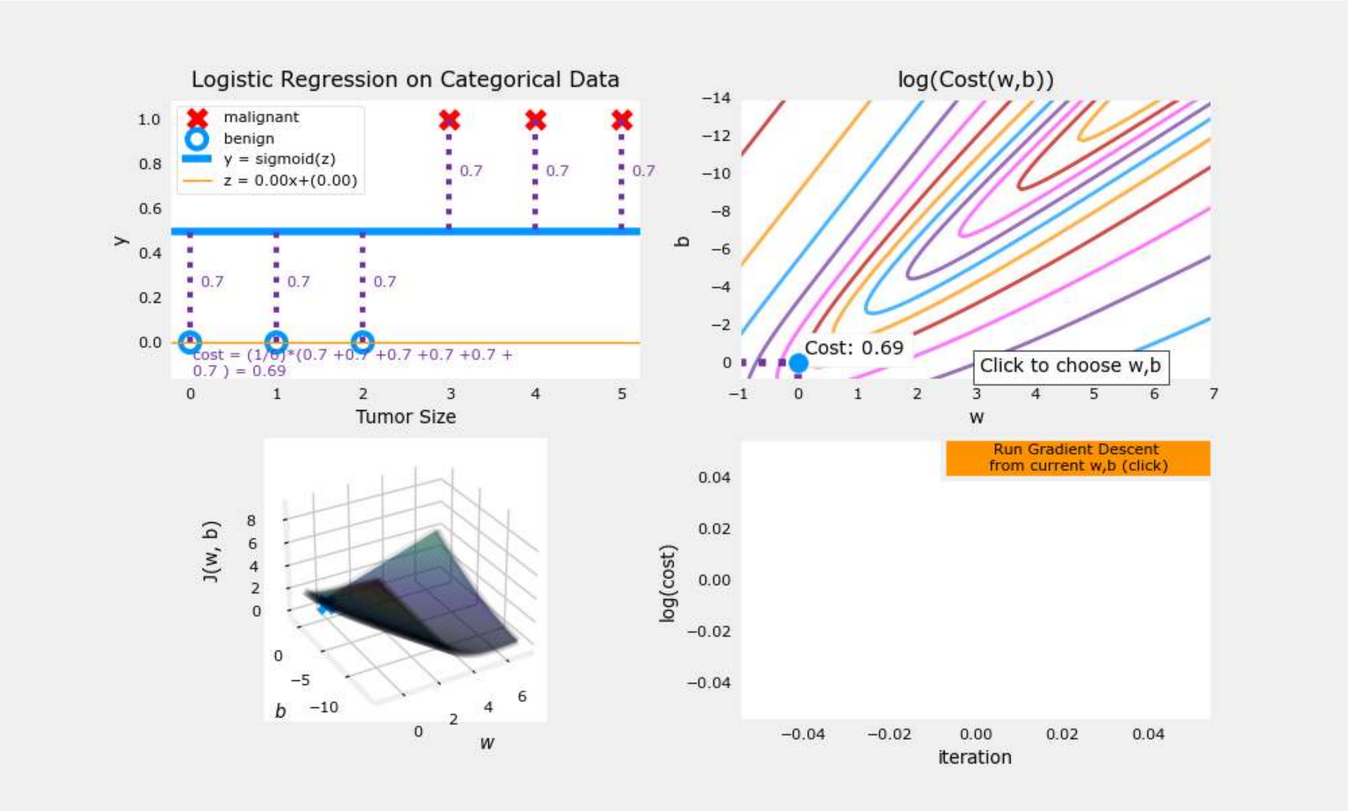
In the plot below, try:

- 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 [27]:

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

Figure



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

Lab: Logistic Regression using Scikit-Learn

Goals

In this lab you will:

- Train a logistic regression model using scikit-learn.

Dataset

Let's start with the same dataset as before.

In [32]:
```
import numpy as np

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

Fit the model

Loading [MathJax]/extensions/Safe.js

The code below imports the logistic regression model from scikit-learn. You can fit this model on the training data by calling `fit` function.

In [33]:

```python
from sklearn.linear_model import LogisticRegression

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

Out[33]:

```
LogisticRegression()
```

Make Predictions

You can see the predictions made by this model by calling the `predict` function.

In [34]:

```python
y_pred = lr_model.predict(X)

print("Prediction on training set:", y_pred)
```

```
Prediction on training set: [0 0 0 1 1 1]
```

Calculate accuracy

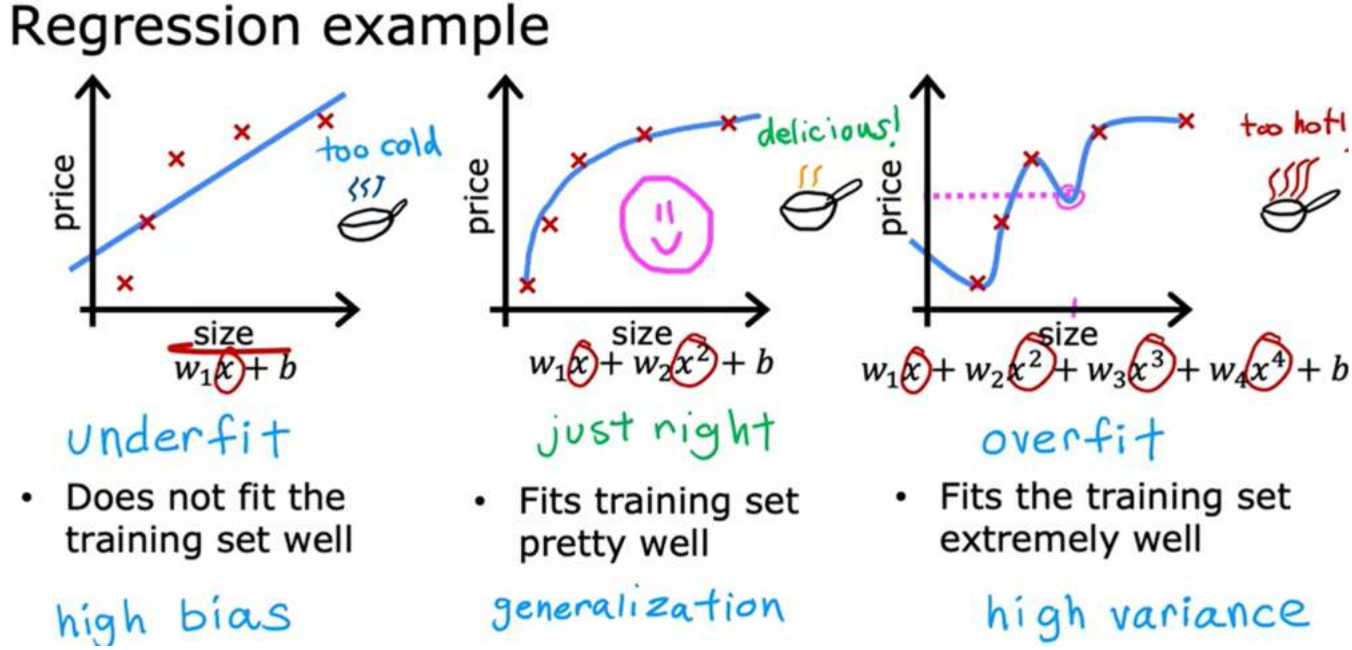
You can calculate this accuracy of this model by calling the `score` function.

In [35]:

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

```
Accuracy on training set: 1.0
```

The problem of overfitting



Regression example

$w_1x + b$ — underfit

- Does not fit the training set well

high bias

$w_1x + w_2x^2 + b$ — just right

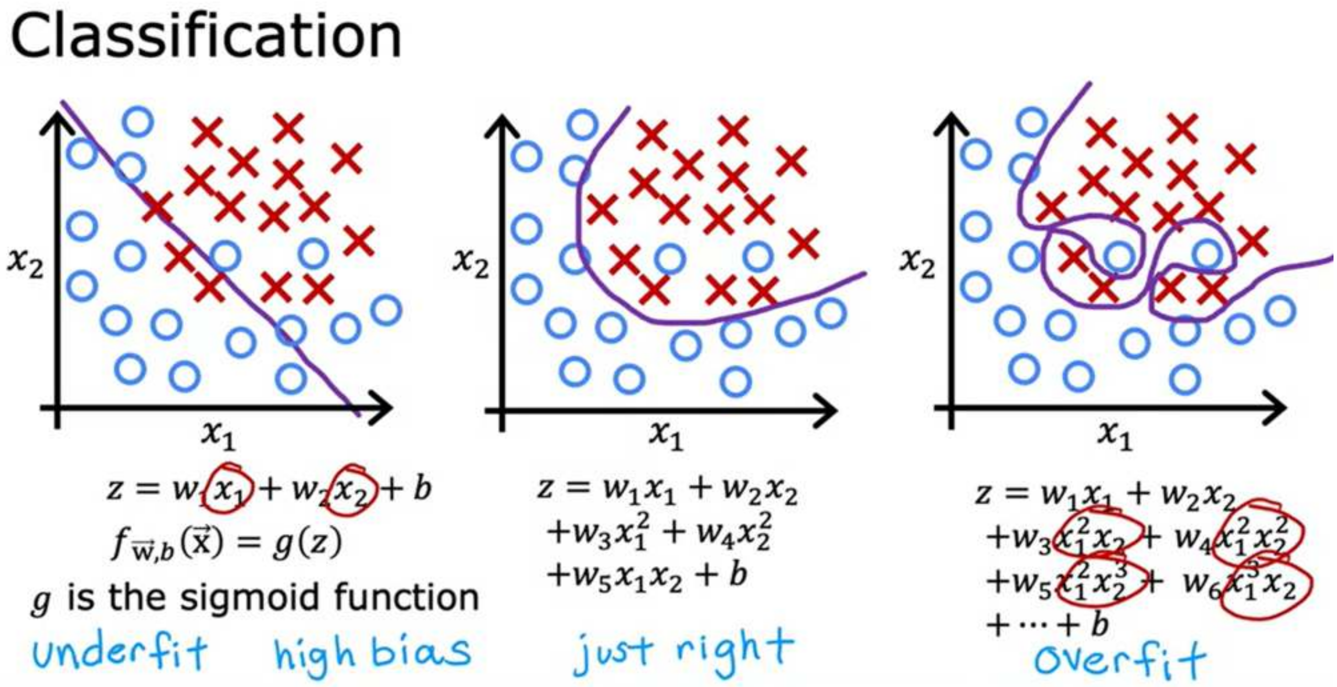
- Fits training set pretty well

generalization

$w_1x + w_2x^2 + w_3x^3 + w_4x^4 + b$ — overfit

- Fits the training set extremely well

high variance



Classification

$z = w_1x_1 + w_2x_2 + b$

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

g is the sigmoid function

underfit high bias

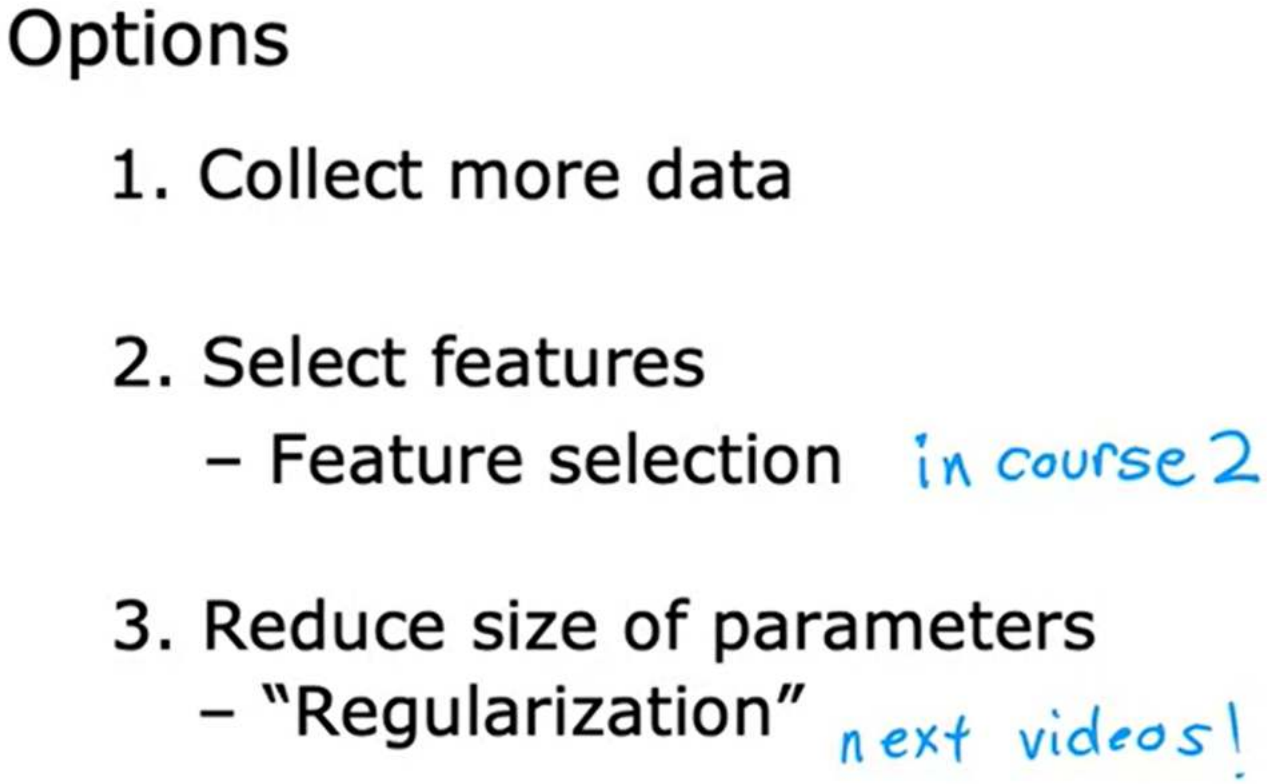
$z = w_1x_1 + w_2x_2$
$+w_3x_1^2 + w_4x_2^2$
$+w_5x_1x_2 + b$

just right

$z = w_1x_1 + w_2x_2$
$+w_3x_1^2x_2 + w_4x_1^2x_2^2$
$+w_5x_1^2x_2^3 + w_6x_1^3x_2$
$+ \cdots + b$

overfit

Addressing overfitting



Options

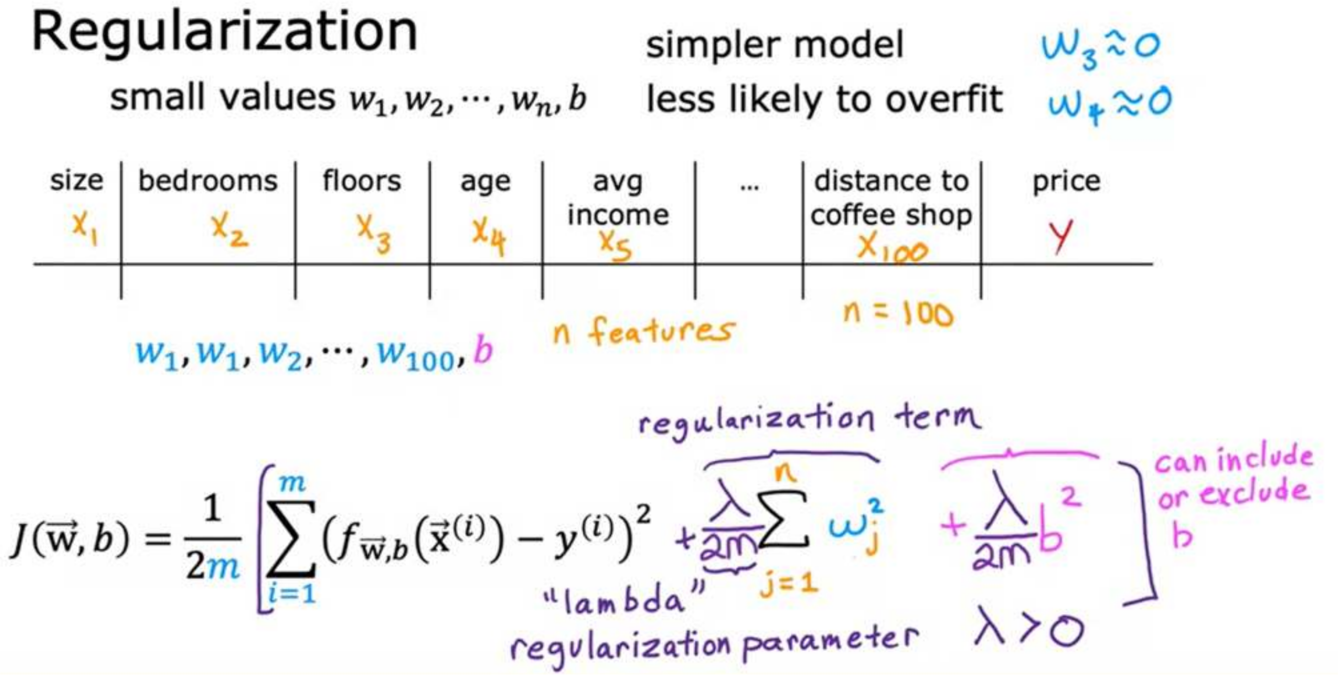
1. Collect more data

2. Select features
   - Feature selection in course 2

3. Reduce size of parameters
   - “Regularization” next videos!

Cost function with regularization

Regularization is a technique used to prevent overfitting in machine learning models by adding a penalty term to the cost function. This penalty term discourages the model from fitting the noise in the training data and encourages it to have simpler and more generalizable solutions.



Regularization

small values $w_1, w_2, \cdots, w_n, b$

simpler model $w_3 \approx 0$

less likely to overfit $w_4 \approx 0$

size	bedrooms	floors	age	avg income	...	distance to coffee shop	price
x_1	x_2	x_3	x_4	x_5		x_{100}	y

$w_1, w_1, w_2, \cdots, w_{100}, b$

n features

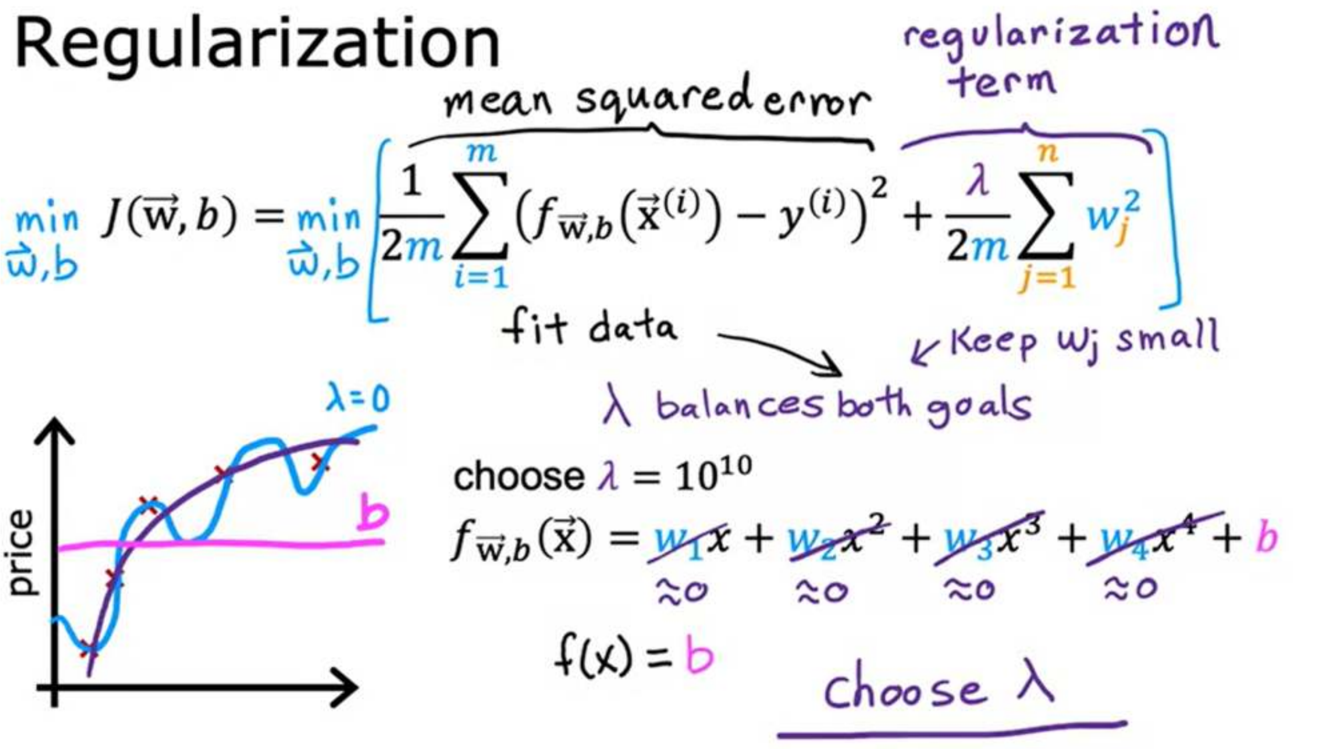
$n = 100$

$$J(\vec{\mathrm{w}}, b) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(f_{\vec{\mathrm{w}},b}\left(\vec{\mathrm{x}}^{(i)}\right) - y^{(i)}\right)^2 + \overbrace{\underbrace{\frac{\lambda}{2m}}_{\text{"lambda"}}\sum_{j=1}^{n} w_j^2}^{\text{regularization term}} + \frac{\lambda}{2m}b^2\right]$$

"lambda"
regularization parameter

$\lambda > 0$

can include or exclude b



Regularization

$$\min_{\vec{\mathrm{w}},b} J(\vec{\mathrm{w}}, b) = \min_{\vec{\mathrm{w}},b}\left[\overbrace{\frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{\mathrm{w}},b}\left(\vec{\mathrm{x}}^{(i)}\right) - y^{(i)}\right)^2}^{\text{mean squared error}} + \overbrace{\frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2}^{\text{regularization term}}\right]$$

fit data → ↙ Keep w_j small

λ balances both goals

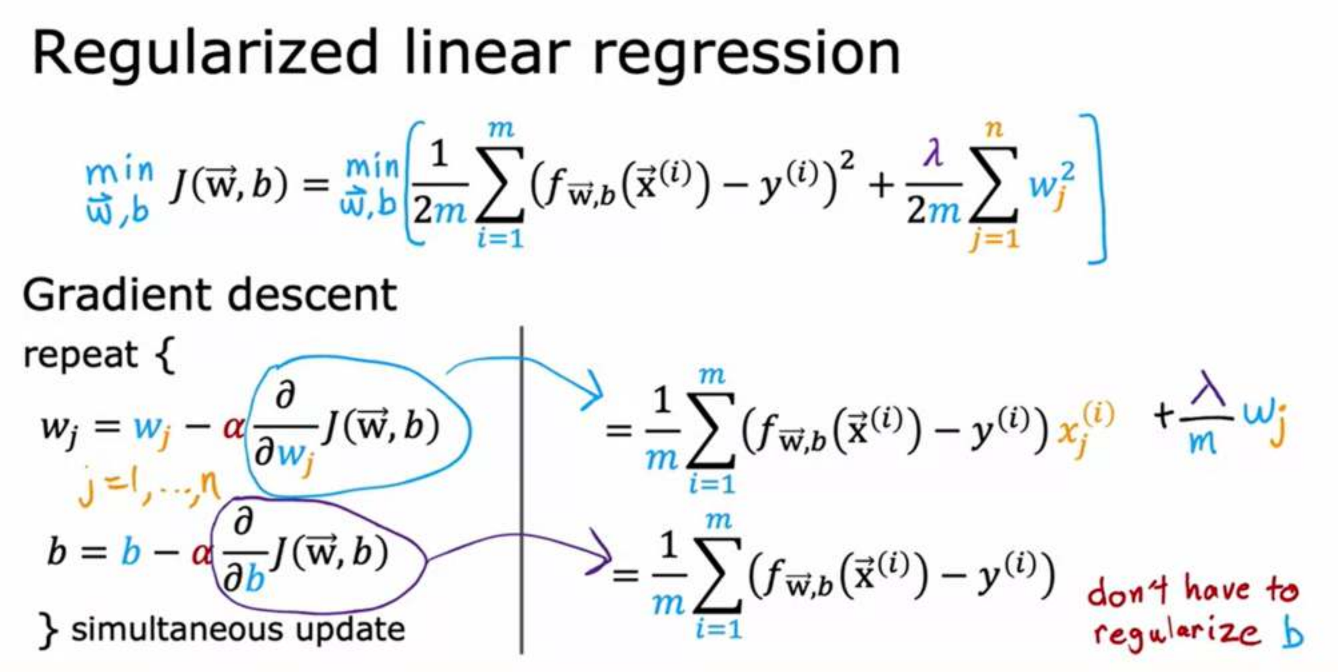
choose $\lambda = 10^{10}$

$$f_{\vec{\mathrm{w}},b}(\vec{\mathrm{x}}) = \underset{\approx 0}{\cancel{w_1 x}} + \underset{\approx 0}{\cancel{w_2 x^2}} + \underset{\approx 0}{\cancel{w_3 x^3}} + \underset{\approx 0}{\cancel{w_4 x^4}} + b$$

$f(x) = b$

Choose λ

Regularized linear regression



Regularized linear regression

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

Gradient descent

repeat {

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

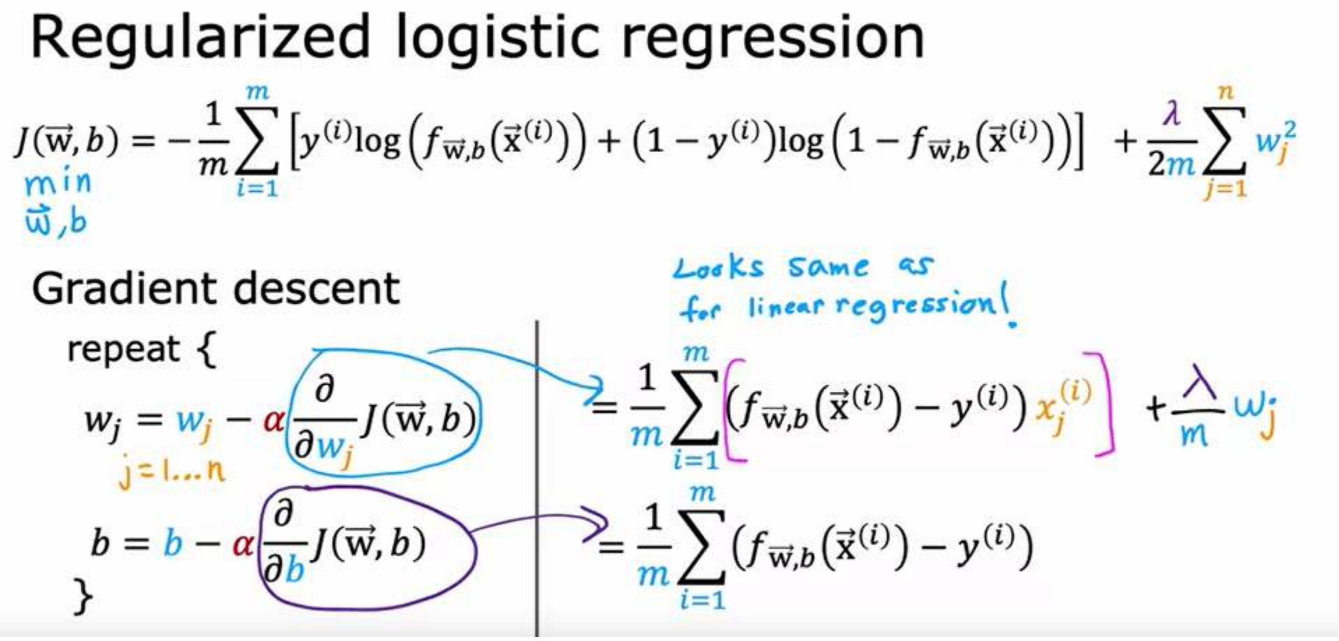
$j=1,\ldots,n$

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

} simultaneous update

don't have to regularize b

Regularized logistic regression



Regularized logistic regression

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

Gradient descent

Looks same as for linear regression!

repeat {

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

$j=1\ldots n$

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

}

Lab - Regularized Cost and Gradient

Goals

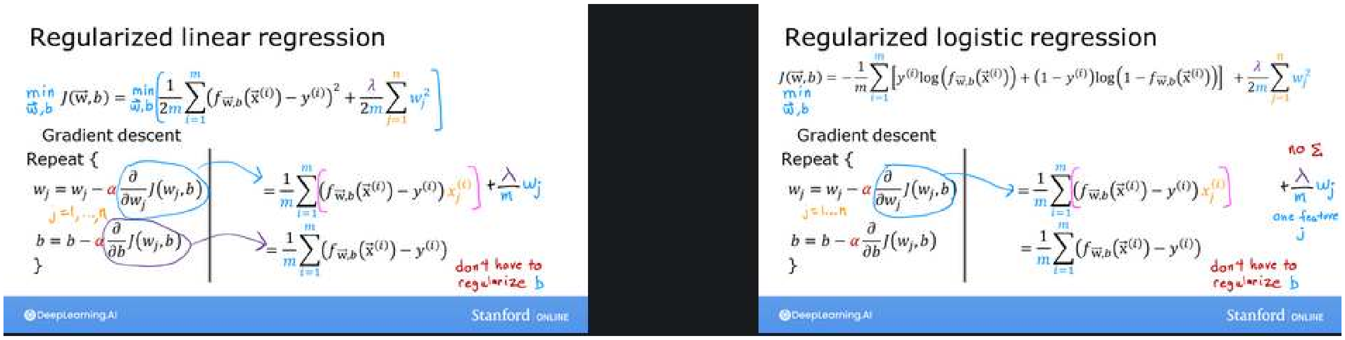
In this lab, you will:

- 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 [36]:

```
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



Regularized linear regression

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

Gradient descent

Repeat {

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

$j = 1, \ldots, n$

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

}

don't have to regularize b

DeepLearning.AI Stanford ONLINE

Regularized logistic regression

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

Gradient descent

Repeat {

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

$j = 1 \ldots n$

no Σ

one feature j

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

}

don't have to regularize b

DeepLearning.AI Stanford ONLINE

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

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

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 [37]:

```
def compute_cost_linear_reg(X, y, w, b, lambda_ = 1):
    """
    Computes the cost over all examples
```

```
      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                    #(n,)(n,)=scalar, see np.dot
        cost = cost + (f_wb_i - y[i])**2                #scalar
    cost = cost / (2 * m)                               #scalar

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

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

Run the cell below to see it in action.

In [38]:

```
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: 0.07917239320214275
```

Expected Output:

Regularized cost: 0.07917239320214275

Cost function for regularized logistic regression

For regularized **logistic** regression, the cost function is of the form $$J(\mathbf{w},b) = \frac{1}{m} \sum_{i=0}^{m-1} \left[-y^{(i)} \log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)} \right) \right) - \left(1 - y^{(i)}\right) \log \left(1 - f_{\mathbf{w},b}\left(\mathbf{x}^{(i)} \right) \right) \right] + \frac{\lambda}{2m} \sum_{j=0}^{n-1} w_j^2 \tag{3}$$ 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 [39]:

```
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                                       #(n,)(n,)=scalar,
        f_wb_i = sigmoid(z_i)                                           #scalar
        cost +=  -y[i]*np.log(f_wb_i) - (1-y[i])*np.log(1-f_wb_i)       #scalar

    cost = cost/m                                                       #scalar

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

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

Run the cell below to see it in action.

In [40]:

```
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_tmp)

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: $$\begin{align*} &\text{repeat until convergence:} \; \lbrace \\ & \; \; \;w_j = w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \tag{1} \; & \text{for j := 0..n-1} \\ & \; \; \; \;b = b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \\ &\rbrace \end{align*}$$ Where each iteration performs simultaneous updates on $w_j$ for all $j$.

Loading [MathJax]/extensions/Safe.js

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 $f_{\mathbf{w}b}$. $$\begin{align*} \frac{\partial J(\mathbf{w},b)}{\partial w_j} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})x_{j}^{(i)} + \frac{\lambda}{m} w_j \tag{2} \\ \frac{\partial J(\mathbf{w},b)}{\partial b} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) \tag{3} \end{align*}$$

- 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 [41]:

```
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 parameters w.
      dj_db (scalar):       The gradient of the cost w.r.t. the parameter b.
    """
    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 [42]:

```
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, b_tmp, lambda_t

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

```
dj_db: 0.6648774569425727
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 [43]:

```
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 parameters w.
      dj_db (scalar)            : The gradient of the cost w.r.t. the parameter b.
    """
    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 [44]:

```
np.random.seed(1)
ndom.rand(5,3)
```

Loading [MathJax]/extensions/Safe.js

```
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, b_tmp, lambda

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

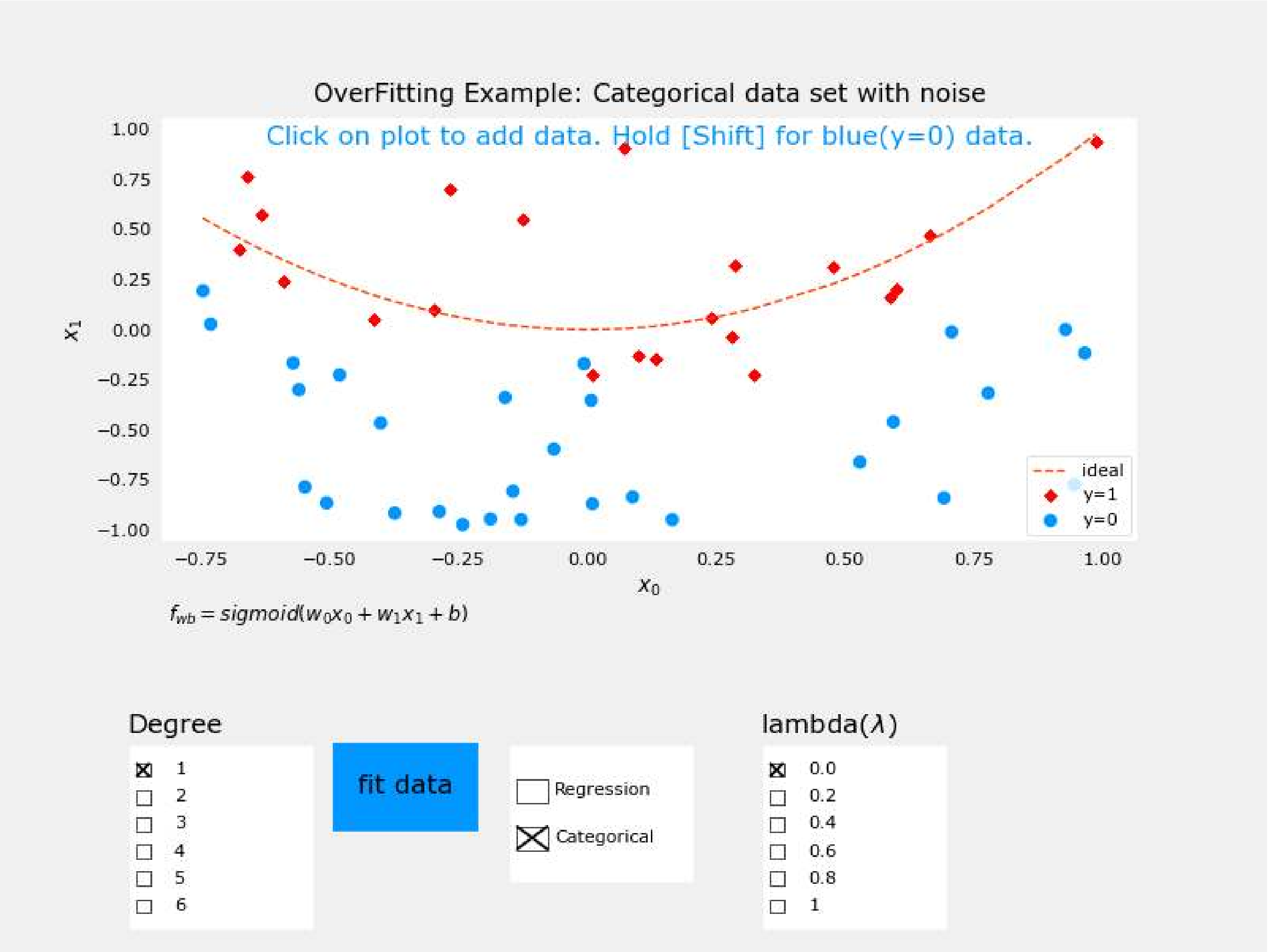
```
dj_db: 0.341798994972791
Regularized dj_dw:
 [0.17380012933994293, 0.32007507881566943, 0.10776313396851499]
```

Rerun over-fitting example

```
In [45]: plt.close("all")
         display(output)
         ofit = overfit_example(True)
```

```
Output()
```

Figure



OverFitting Example: Categorical data set with noise
Click on plot to add data. Hold [Shift] for blue(y=0) data.
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
x_1
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x_0
ideal
y=1
y=0
$f_{wb} = sigmoid(w_0x_0 + w_1x_1 + b)$
fit data

Degree

☒ 1
□ 2
□ 3
□ 4
□ 5
□ 6

□ Regression
☒ Categorical

lambda(λ)

☒ 0.0
□ 0.2
□ 0.4
□ 0.6
□ 0.8
□ 1

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 Lab

Logistic Regression

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

Outline

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 [140…

```
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 [120…

```
# Initialize empty lists to store the data
X_train = []
y_train = []

# Open the file for reading
with open("data/ex2data1.txt", "r") as file:
    # Read each line of the file
    for line in file:
        # Split the line into features and label (assuming they are separated by commas)
        data = line.strip().split(',')
```

Loading [MathJax]/extensions/Safe.js

```
        # Extract the features (first two elements) and the label (last element)
        features = [float(data[0]), float(data[1])]
        label = int(data[2])

        # Append the features and label to X_train and y_train, respectively
        X_train.append(features)
        y_train.append(label)

# Convert X_train and y_train to numpy arrays
X_train = np.array(X_train)
y_train = np.array(y_train)

# Now X_train and y_train should contain the data from the file
```

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 [121…

```
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 [122…

```
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 [123…

```
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.

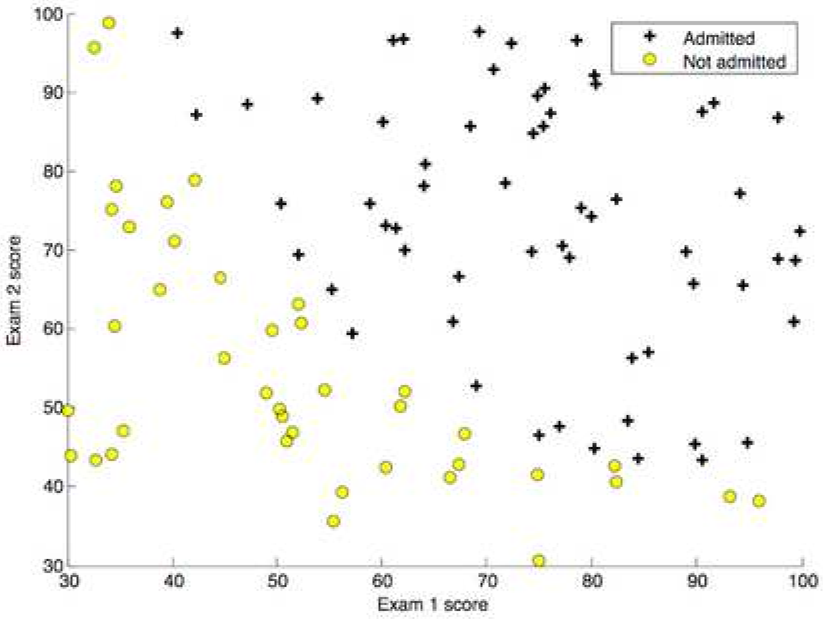
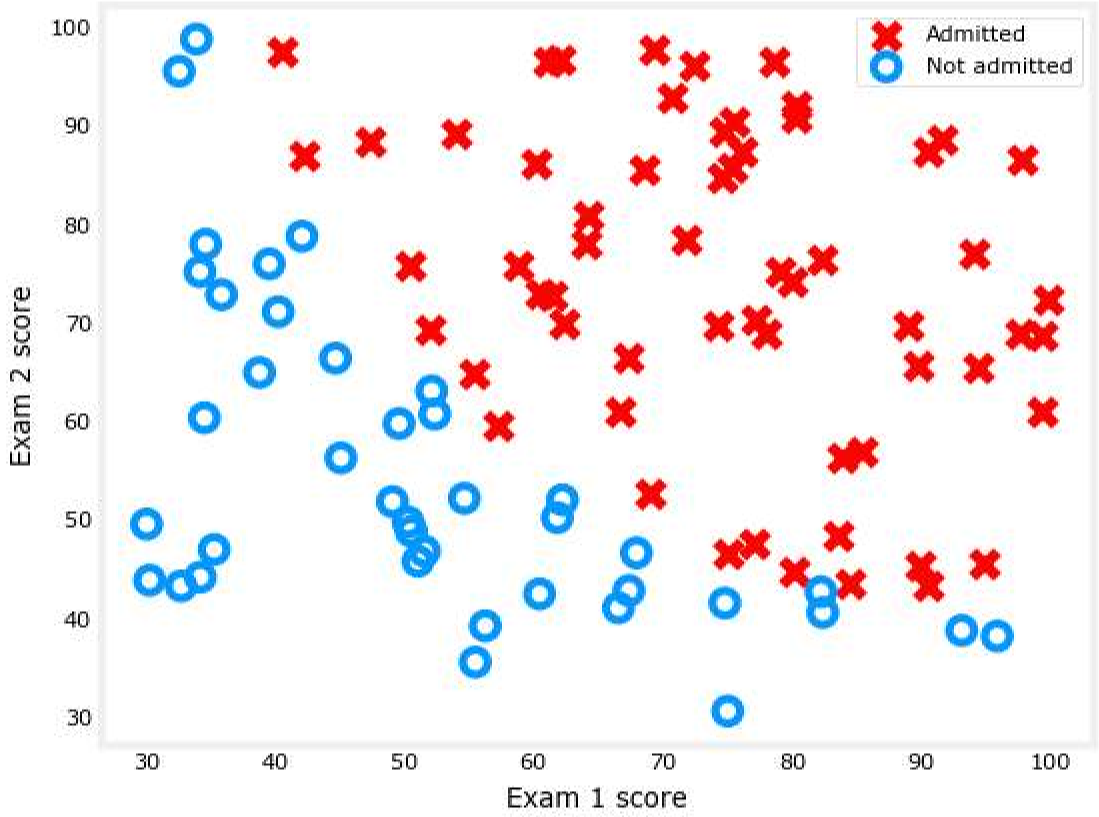


Figure 1: Scatter plot of training data

In [126…

```
# Call the plot_data() function with the 'ax' argument
fig, ax = plt.subplots()  # Create a new figure and axes
plot_data(X_train, y_train[:], pos_label="Admitted", neg_label="Not admitted", ax=ax)

# 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 [127…

```
# 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
```

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 [128…

```
# 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 [129…

```
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 -

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

- $f_{\mathbf{w},b}(\mathbf{x}^{(i)})$ 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 $z_{\mathbf{w},b}(\mathbf{x}^{(i)}) = \mathbf{w} \cdot \mathbf{x^{(i)}} + b = w_0x^{(i)}_0 + ... + w_{n-1}x^{(i)}_{n-1} + b$ 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 [130…

```
# 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
    cost = 0.
    ### SATRT CODE HERE ###
```

Loading [MathJax]/extensions/Safe.js

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

    total_cost = cost/m

    ### END CODE HERE ###

    return total_cost
```

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

In [131…

```
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 [132…

```
# 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:

$$\begin{align*}& \text{repeat until convergence:} \; \lbrace \newline \; & b := b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \newline \; & w_j := w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \tag{1} \; & \text{for j := 0..n-1}\newline & \rbrace\end{align*}$$
where, parameters $b$, $w_j$ are all updated simultaniously

Loading [MathJax]/extensions/Safe.js

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.

$$ \frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)}) \tag{2} $$$$ \frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)})x_{j}^{(i)} \tag{3} $$

- 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 [133…

```python
# 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 parameters w.
      dj_db : (scalar)             The gradient of the cost w.r.t. the parameter b.
    """
    m, n = X.shape
    dj_dw = np.zeros(w.shape)
    dj_db = 0.

    ### START CODE HERE ###
    for i in range(m):
        f_wb = sigmoid(np.dot(X[i], w) + b)
        err = f_wb - y[i]
        for j in range(n):
            dj_dw[j] += err * X[i, j]
        dj_db += err

    dj_dw = dj_dw/m
    dj_db = dj_db/m

    ### END CODE HERE ###


    return dj_db, dj_dw
```

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

Loading [MathJax]/extensions/Safe.js

In [134…

```
# 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 [136…

```
# 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 [137…

```
def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function, alpha, num_iter
    """
    Performs batch gradient descent to learn theta. Updates theta by taking
    num_iters gradient steps with learning rate alpha

    Args:
Loading [MathJax]/extensions/Safe.js (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 model
      b_in : (scalar)              Initial value of parameter of the model
      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 descent
      lambda_ : (scalar, float)    regularization constant

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

    # number of training examples
    m = len(X)

    # An array to store cost J and w's at each iteration primarily for graphing later
    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 < 10
        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 graphing
```

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 [138…

```
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_b,
                                   compute_cost, compute_gradient, alpha, iterations, 0)
```

Loading [MathJax]/extensions/Safe.js

```
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:

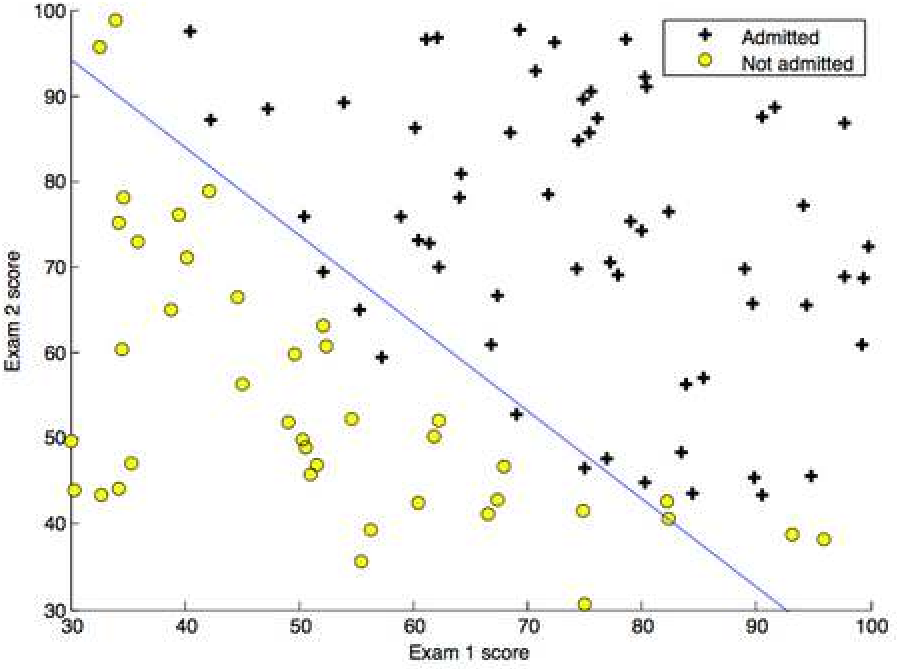


Figure 2: Training data with decision boundary

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

In [165…

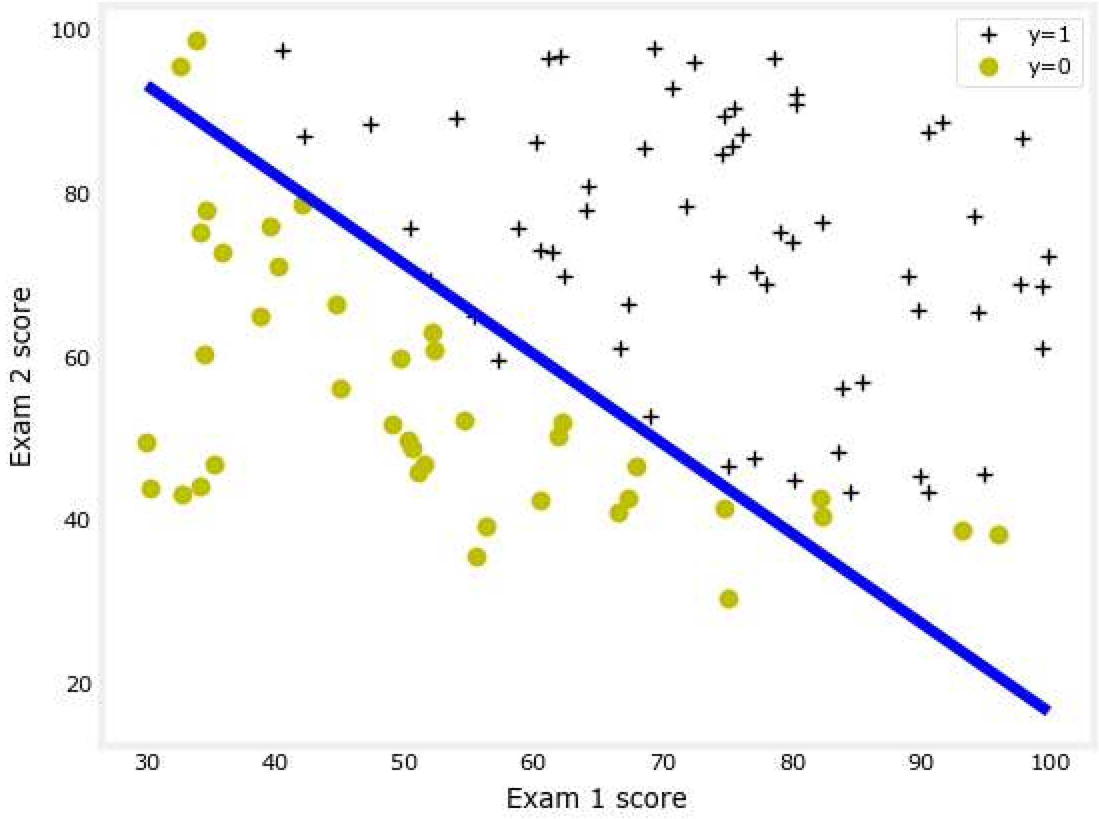
```python
def plot_data(X, y, pos_label="y=1", neg_label="y=0", ax=None):
    positive = y == 1
    negative = y == 0

    if ax is None:
        ax = plt.gca()  # Get the current axes if none is provided

    # Plot examples
    ax.plot(X[positive, 0], X[positive, 1], 'k+', label=pos_label)
    ax.plot(X[negative, 0], X[negative, 1], 'yo', label=neg_label)

def plot_decision_boundary(w, b, X, y):
    # Credit to dibgerge on Github for this plotting code

    fig, ax = plt.subplots()  # Create a new figure and axes
```

Loading [MathJax]/extensions/Safe.js

```
    plot_data(X[:, 0:2], y, ax=ax)  # Pass the 'ax' argument here

    if X.shape[1] <= 2:
        plot_x = np.array([min(X[:, 0]), max(X[:, 0])])
        plot_y = (-1. / w[1]) * (w[0] * plot_x + b)

        ax.plot(plot_x, plot_y, c="b")  # Use the 'ax' object to plot

    else:
        u = np.linspace(-1, 1.5, 50)
        v = np.linspace(-1, 1.5, 50)

        z = np.zeros((len(u), len(v)))

        # Evaluate z = theta*x over the grid
        for i in range(len(u)):
            for j in range(len(v)):
                z[i,j] = sigmoid(np.dot(map_feature(u[i], v[j]), w) + b)

        # important to transpose z before calling contour
        z = z.T

        # Plot z = 0.5
        ax.contour(u,v,z, levels=[0.5], colors="g")
```

In [146…

```
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 [147…

```
# 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):

        # Calculate f_wb (exactly how you did it in the compute_cost function above)
        # using a couple of lines of code
        f_wb = sigmoid(np.dot(X[i], w) + b)

        # Calculate the prediction for that training example
        p[i] = f_wb >= 0.5# Your code here to calculate the prediction based on f_wb
```

Loading [MathJax]/extensions/Safe.js

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

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 [148…

```
# 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 [149…

```
#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 [151…

```
# Initialize empty lists to store the data
X_train = []
y_train = []

# Open the file for reading
with open("data/ex2data2.txt", "r") as file:
    # Read each line of the file
    for line in file:
        # Split the line into features and label (assuming they are separated by commas)
        data = line.strip().split(',')

        # Extract the features (first two elements) and the label (last element)
        features = [float(data[0]), float(data[1])]
        label = int(data[2])

        # Append the features and label to X_train and y_train, respectively
        X_train.append(features)
        y_train.append(label)

# Convert X_train and y_train to numpy arrays
X_train = np.array(X_train)
y_train = np.array(y_train)

# Now X_train and y_train should contain the data from the file
```

View the variables

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

In [152…

```
# 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 [153…

```
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.

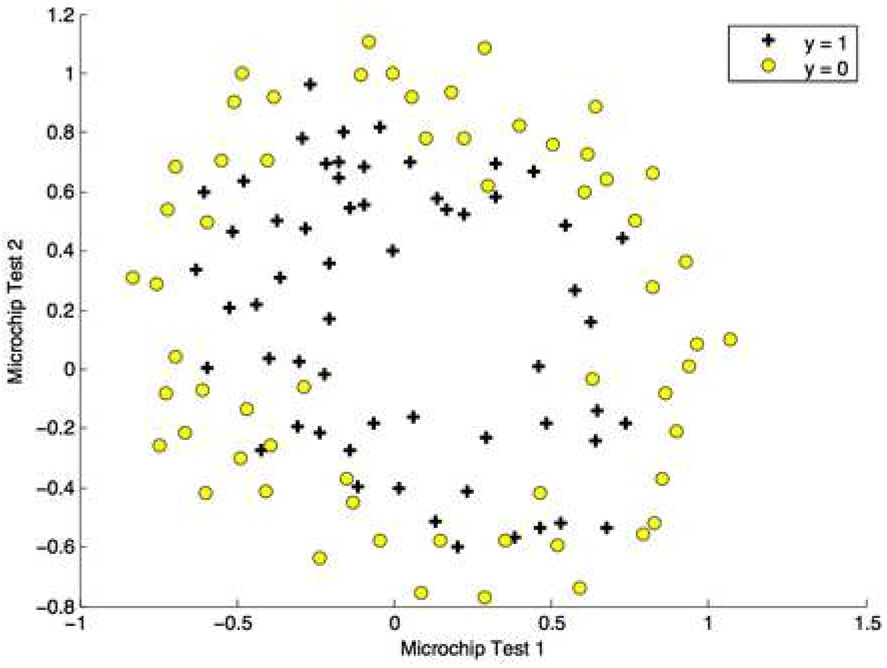


Figure 3: Plot of training data

In [154…

```
# 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()
```

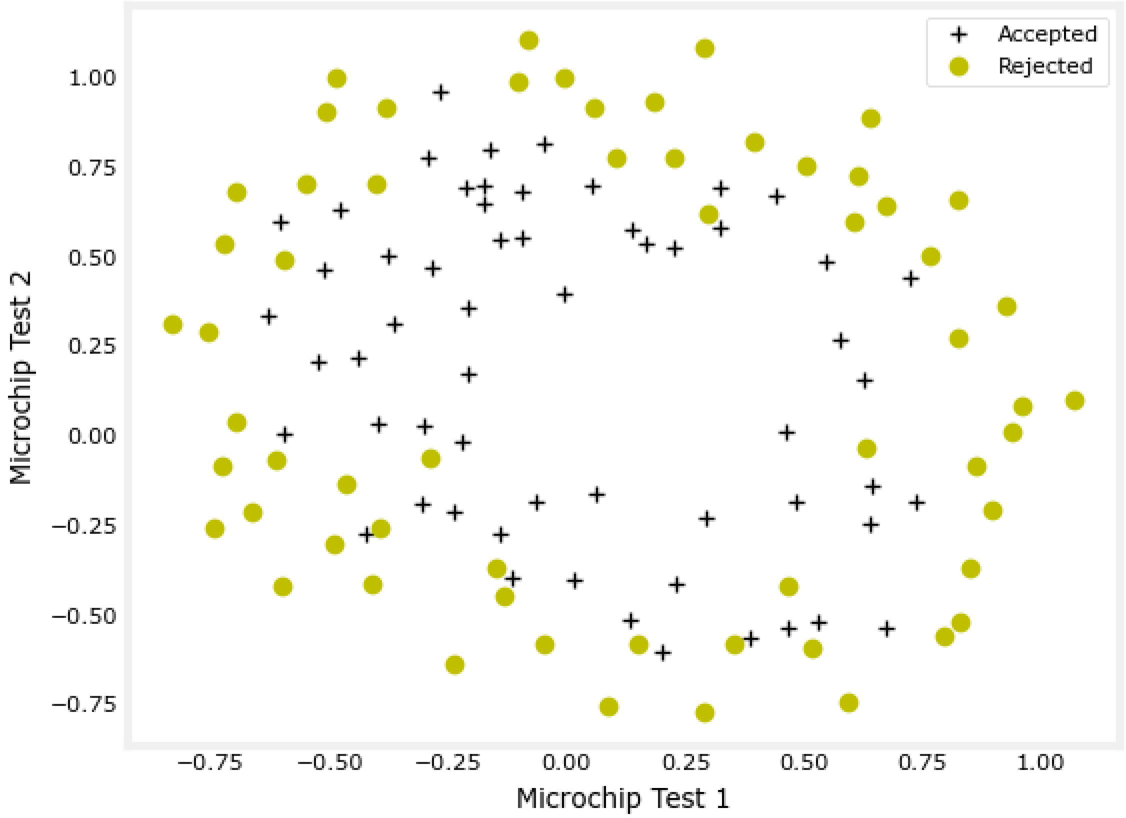


Figure 3 shows that our dataset cannot be separated into positive and negative examples by a straight-line through the plot. Therefore, a straight forward application of logistic regression will not perform well on this dataset since logistic regression will only be able to find a linear decision boundary.

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.

$$\mathrm{map\_feature}(x) = \left[\begin{array}{c} x_1\\ x_2\\ x_1^2\\ x_1 x_2\\ x_2^2\\ x_1^3\\ \vdots\\ x_1 x_2^5\\ x_2^6\end{array}\right]$$

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 [156…

```
def map_feature(X1, X2):
    """
    Feature mapping function to polynomial features
    """
    X1 = np.atleast_1d(X1)
    X2 = np.atleast_1d(X2)
    degree = 6
    out = []
    for i in range(1, degree+1):
        for j in range(i + 1):
            out.append((X1**(i-j) * (X2**j)))
    .stack(out, axis=1)
```

Loading [MathJax]/extensions/Safe.js

In [157…

```
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 [158…

```
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 3.58643425e-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 $$J(\mathbf{w},b) = \frac{1}{m} \sum_{i=0}^{m-1} \left[-y^{(i)} \log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)} \right) \right) - \left(1 - y^{(i)}\right) \log \left(1 - f_{\mathbf{w},b}\left(\mathbf{x}^{(i)} \right) \right) \right] + \frac{\lambda}{2m} \sum_{j=0}^{n-1} w_j^2$$

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 [159…

```
# 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_j = w[j]**2
        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
```

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

In [160…

```
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.

Loading [MathJax]/extensions/Safe.js

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 $j^\mathrm{th}$ element is defined as follows:

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) $$$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \left(\frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} w_j \quad\, \mbox{for $j=0...(n-1)$}$$ Compare this to the gradient of the cost function without regularization (which you implemented above), which is of the form $$ \frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)}) \tag{2} $$ $$ \frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)})x_{j}^{(i)} \tag{3} $$

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

Exercise 6

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

$$\frac{\lambda}{m} w_j \quad\, \mbox{for $j=0...(n-1)$}$$
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 [161…

```
# 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 parameter b.
      dj_dw : (ndarray Shape (n,)) The gradient of the cost w.r.t. the parameters w.

    """
    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 = (lambda_ / m) * w[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
```

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

In [162…

```
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_b, lambda_)

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, 0.003140278267313463
7]
All tests passed!
```

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 [163…

```
# 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_b,
                                    compute_cost_reg, compute_gradient_reg,
                                    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.

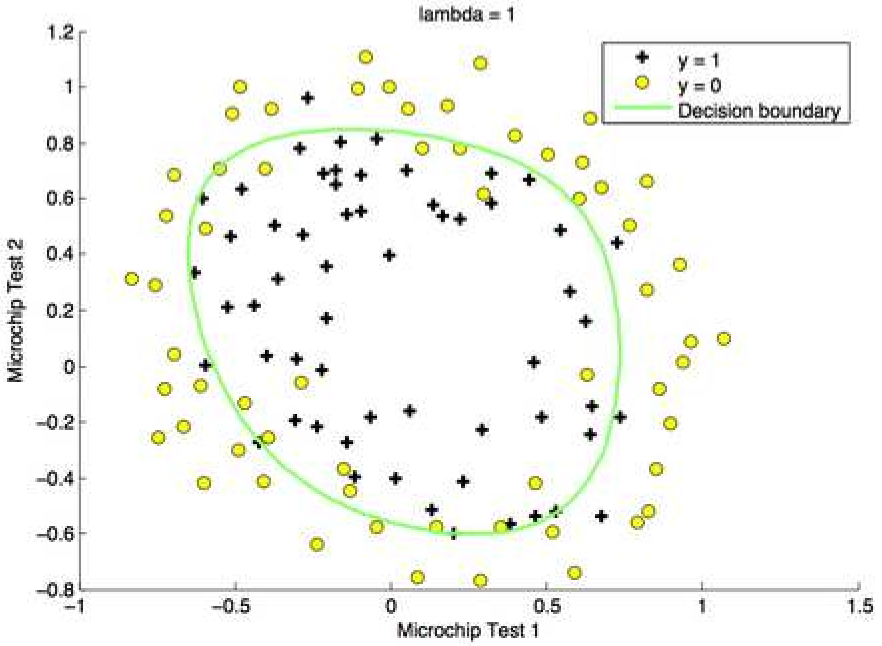
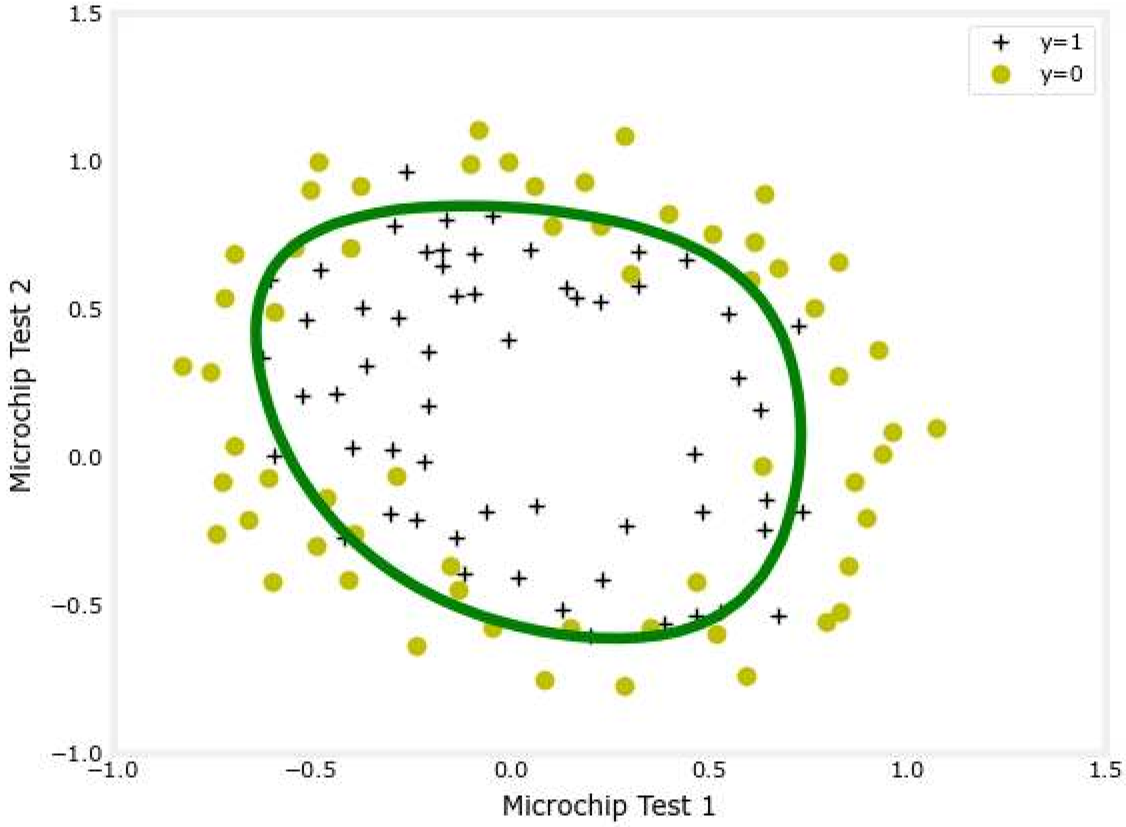


Figure 4: Training data with decision boundary ($\lambda = 1$)

In [166…

```
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()
```

Loading [MathJax]/extensions/Safe.js



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 [167…

```
#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!

In [ ]:

```
```