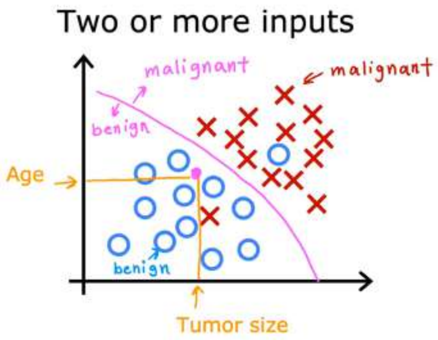
Week 1

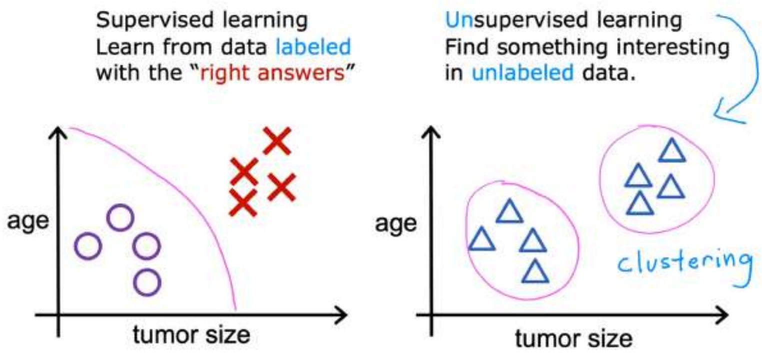
REFER TO THE GITHUB FOR THE LAB CODES - https://github.com/Bhardwaj-Saurabh/Machine_Learning_Specialization_AndrewNG_Coursera

Supervised learning part 2

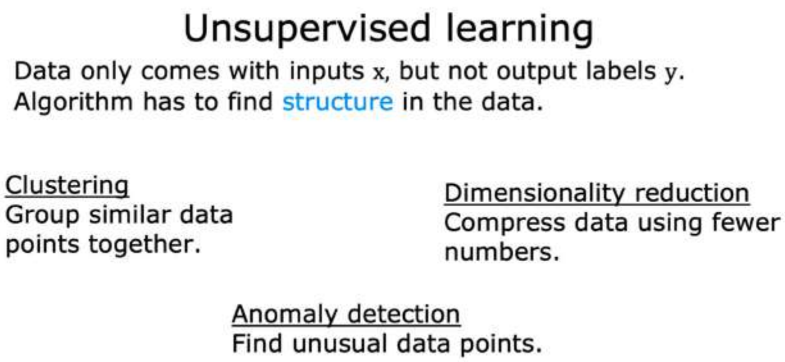


2 or more inputs can be given to a classification supervised model and here the machine would try to fit a line between the 2 classes/categories (here). the learning algorithm will learn to fit a boundary line to this data.

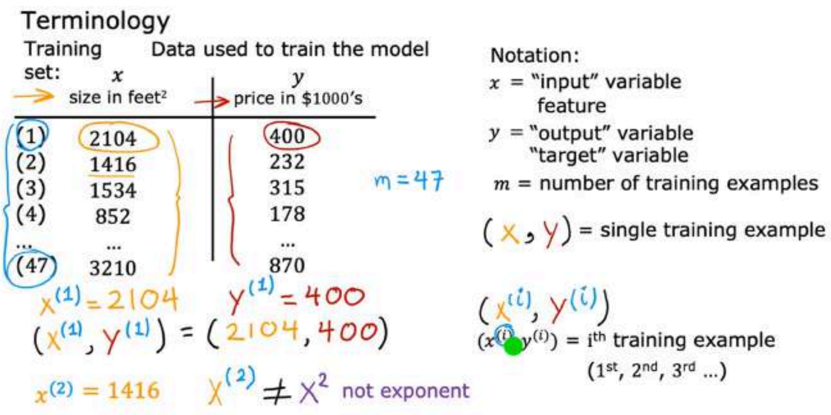
Unsupervised learning part 1



Unsupervised learning part 2



Linear regression model part 1



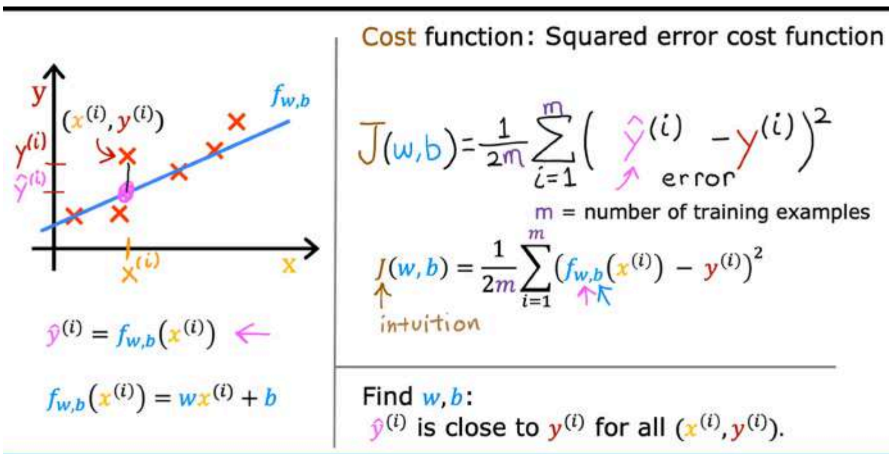
C1_W1_Lab03_Model_Representation_Soln.ipynb

```
w = 100
b = 100
print(f"w: {w}")
print(f"b: {b}")


def compute_model_output(x, w, b):

    m= x.shape[0]   #----> x.shape returns (n,0) where n is the length of the array
    f_wb= np.zeros(m)
    for i in range(m):
            f_wb[i]= w* x[i]+ b

    return f_wb
    #f_wb is nothing but y values!!! which we plot
```

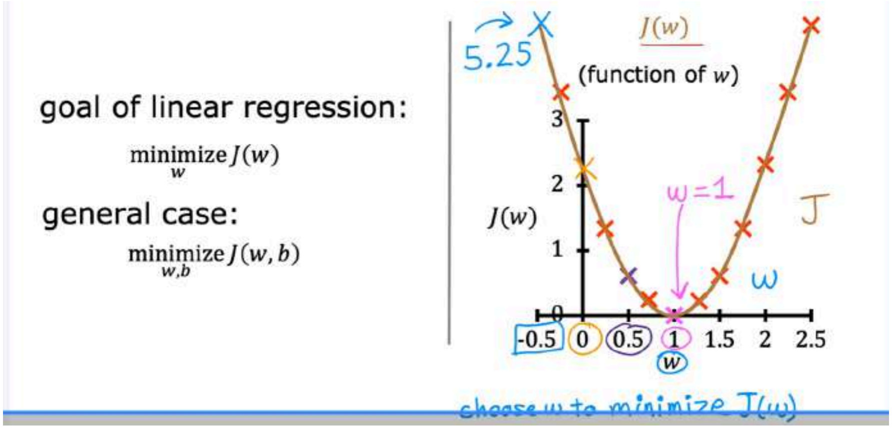
here we are finding the value of `f_wb` for every value of X given a particular value we defined (w,b). we then took the size of x, made an array of that size and then changed the values to get a list of `f_wb` values. which we plot.

```
tmp_f_wb= compute_model_output(x_train, w, b,)

# Plot our model prediction
plt.plot(x_train, tmp_f_wb, c='b',label='Our Prediction')

# Plot the data points
plt.scatter(x_train, y_train, marker='x', c='r',label='Actual Values')

# Set the title
plt.title("Housing Prices")
# Set the y-axis label
plt.ylabel('Price (in 1000s of dollars)')
# Set the x-axis label
plt.xlabel('Size (1000 sqft)')
plt.legend()
plt.show()
```

Cost function formula



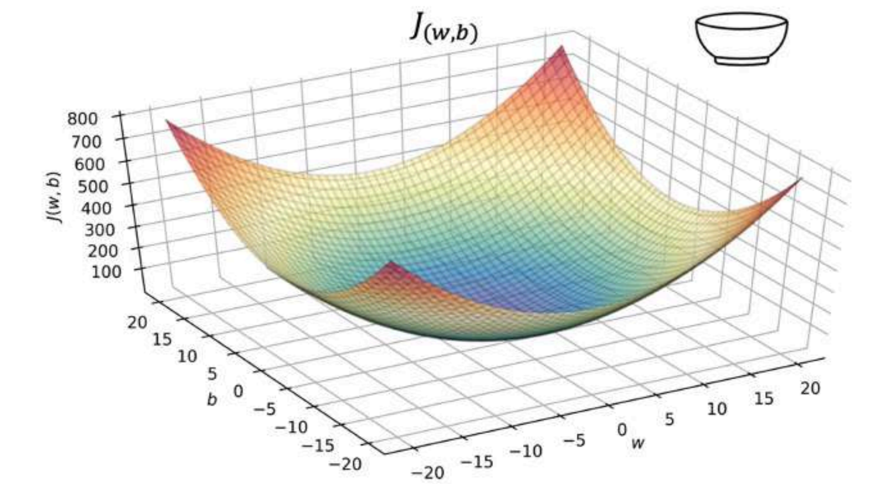
Cost function intuition



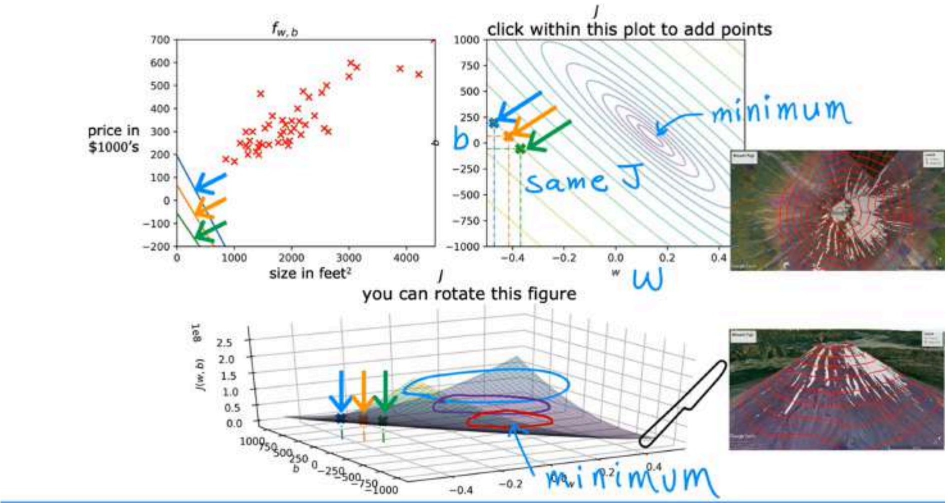
model function is the function of X for a fixed value of w,b

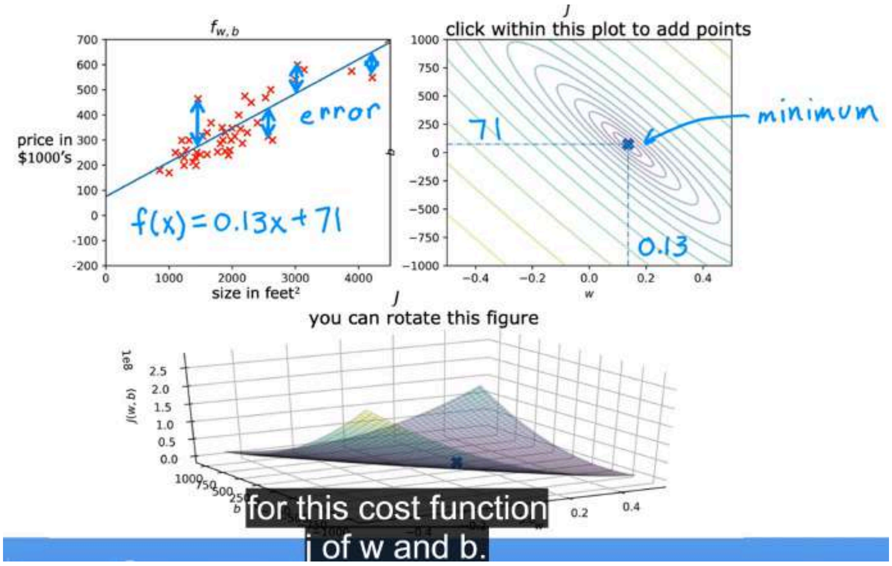
cost function is a function of w,b! not X!

Cost function visualization

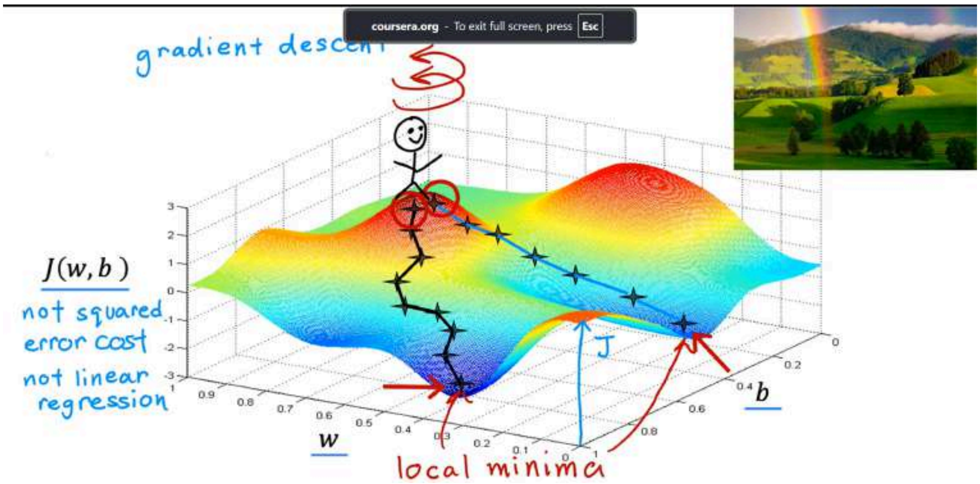


when we plot j(w,b) we get a 3d plot. when we plot j(w) keeping b=0, we get a 2d parabolic graph.





Gradient descent

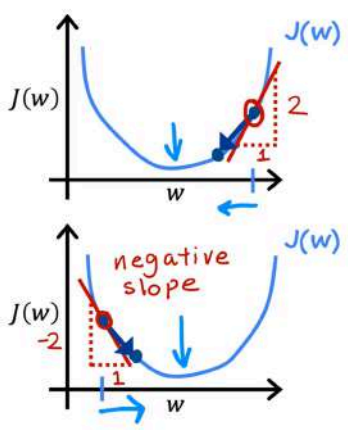


here we have an example of a cost fn not “squared error” one but a diff one and also not a LR model (which would give you a parabolic curve to the cost fn. so here too we use GD to get to the local minima of the graph.

the baby steps here is the alpha is the learning rate, how fast you move down the hill.

correct and incorrect way to implement gradient descent simultaneous updation of parameters

Correct: Simultaneous update	Incorrect
$tmp\_w = w - \alpha \frac{\partial}{\partial w} J(w,b)$	$tmp\_w = w - \alpha \frac{\partial}{\partial w} J(w,b)$
$tmp\_b = b - \alpha \frac{\partial}{\partial b} J(w,b)$	$w = tmp\_w$
$w = tmp\_w$	$tmp\_b = b - \alpha \frac{\partial}{\partial b} J(w,b)$
$b = tmp\_b$	$b = tmp\_b$



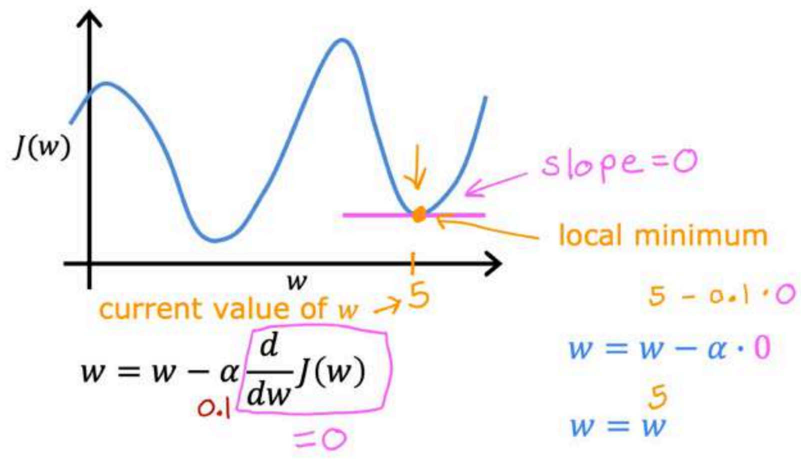
$$w = w - \alpha \underbrace{\boxed{\frac{d}{dw} J(w)}}_{>0}$$

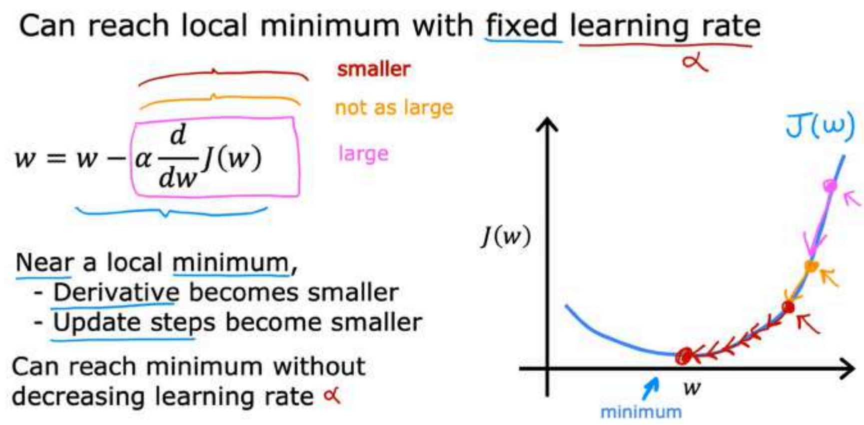
$$w = w - \alpha \cdot (positive\ number)$$

$$\underbrace{\frac{d}{dw} J(w)}_{<0}$$

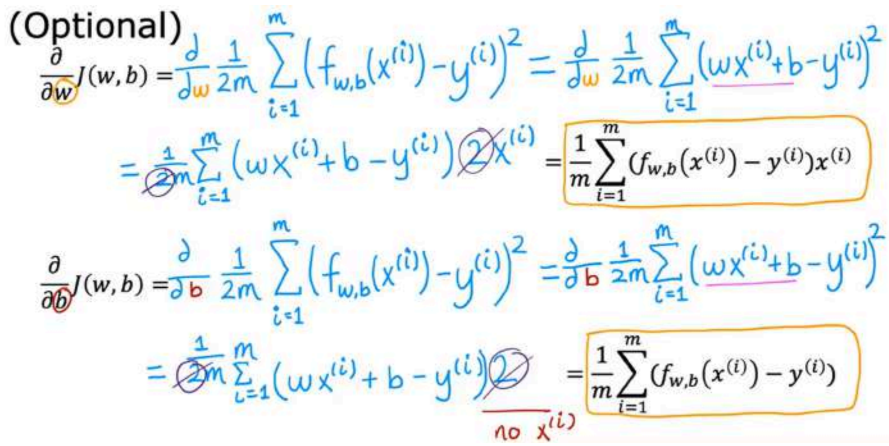
$$w = w - \alpha \cdot (negative\ number)$$

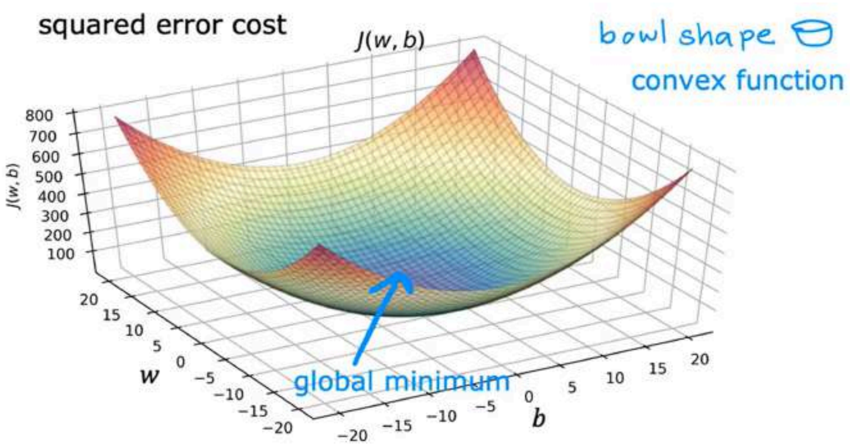
understanding the derivative used in GD





Gradient descent for linear regression



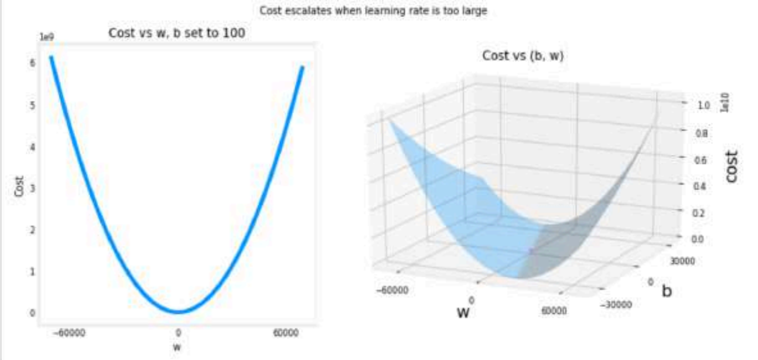


C1_W1_Lab04_Gradient_Descent_Soln

code to implement GD for LR

```python
import math, copy
import numpy as np
import matplotlib.pyplot as plt

# Load our data set
x_train = np.array([1.0, 2.0])   #features
y_train = np.array([300.0, 500.0])   #target value

#Function to calculate the cost
def compute_cost(x, y, w, b):

    m = x.shape[0]
    cost = 0

    for i in range(m):
        f_wb = w * x[i] + b
        cost = cost + (f_wb - y[i])**2
    total_cost = 1 / (2 * m) * cost

    return total_cost


"""
You will implement gradient descent algorithm for one feature. You will need three functions.

- `compute_gradient` implementing equation (4) and (5) above
- `compute_cost` implementing equation (2) above (code from previous lab)
- `gradient_descent`, utilizing compute_gradient and compute_cost

Conventions:

- The naming of python variables containing partial derivatives follows this pattern,∂J(w,b)∂b∂J(w,b)∂b will be `dj_db`.
- w.r.t is With Respect To, as in partial derivative of J(wb)J(wb) With Respect To bb.
"""

def compute_gradient(x, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      x (ndarray (m,)): Data, m examples
      y (ndarray (m,)): target values
      w,b (scalar)    : model parameters
    Returns
      dj_dw (scalar): 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
     """

    # Number of training examples
    m = x.shape[0]
    dj_dw = 0
    dj_db = 0

    for i in range(m):
        f_wb = w * x[i] + b
        dj_dw_i = (f_wb - y[i]) * x[i]
        dj_db_i = f_wb - y[i]
        dj_db += dj_db_i
        dj_dw += dj_dw_i
    dj_dw = dj_dw / m
    dj_db = dj_db / m

    return dj_dw, dj_db

def gradient_descent(x, y, w_in, b_in, alpha, num_iters, cost_function, gradient_function):
    # An array to store cost J and w's at each iteration primarily for graphing later
    #J_history = []
    #p_history = []
    b = b_in
    w = w_in

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

        # Update Parameters using equation (3) above
        b = b - alpha * dj_db
        w = w - alpha * dj_dw

        # Save cost J at each iteration
```

```
        if i<100000:      # prevent resource exhaustion
            J_history.append( cost_function(x, y, w , b))
            #p_history.append([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:4}: Cost {J_history[-1]:0.2e} ",
                  f"dj_dw: {dj_dw: 0.3e}, dj_db: {dj_db: 0.3e}  ",
                  f"w: {w: 0.3e}, b:{b: 0.5e}")"""

    return w, b, J_history, p_history #return w and J,w history for graphing




#--------client code

# initialize parameters
w_init = 0
b_init = 0
# some gradient descent settings
iterations = 10000
tmp_alpha = 1.0e-2
# run gradient descent
w_final, b_final, J_hist, p_hist = gradient_descent(x_train ,y_train, w_init, b_init, tmp_alpha,
                                                    iterations, compute_cost, compute_gradient)
print(f"(w,b) found by gradient descent: ({w_final:8.4f},{b_final:8.4f})")
```



Above, the left graph shows w's progression over the first few steps of gradient descent. w oscillates from positive to negative and cost grows rapidly. Gradient Descent is operating on both w and b simultaneously, so one needs the 3-D plot on the right for the complete picture.

final code for GD for LR in univariate

```
import numpy as np



x_train = np.array([1.0, 2.0])

y_train = np.array([300.0, 500.0])



w_init = 0

b_init = 0

alpha = 0.001

num_iters = 100000

def costfn(x_train, y_train, w,b):

    j_wb = 0

    m = x_train.shape[0]

    for i in range(m):

        f_wb = w*x_train[i]+b

        j_wb = j_wb + (f_wb-y_train[i])**2

    j_wb = j_wb/(2*m)
```

```
    return j_wb

def gradientfn(x_train, y_train, w,b):

    m = x_train.shape[0]

    d_jwb_w = 0

    d_jwb_b = 0



    for i in range(m):

        f_wb = w*x_train[i] + b

        d_jwb_w = d_jwb_w + (f_wb-y_train[i])*x_train[i]

        d_jwb_b = d_jwb_b + (f_wb-y_train[i])

    d_jwb_w = d_jwb_w/m
    d_jwb_b = d_jwb_b/m



    return d_jwb_w, d_jwb_b





def gradient_descent(x_train, y_train, alpha, w_init, b_init, num_iters, costfn, gradientfn):

    w,b = w_init, b_init

    dj_dw , dj_db = gradientfn(x_train, y_train, w,b)

    jwb = []



    for i in range(num_iters):

        dj_dw , dj_db = gradientfn(x_train, y_train, w,b)

        w = w - alpha*dj_dw

        b = b - alpha*dj_db

        if (i%1000)==0:

            jwb.append(costfn(x_train, y_train, w,b))

    return w,b,jwb



w_final, b_final,j_wb = gradient_descent(x_train, y_train, alpha, w_init,b_init, num_iters, costfn, gradientfn)

print(f'w_final:{w_final:.2f}, b_final:{b_final:.2f}')

print(j_wb)
```