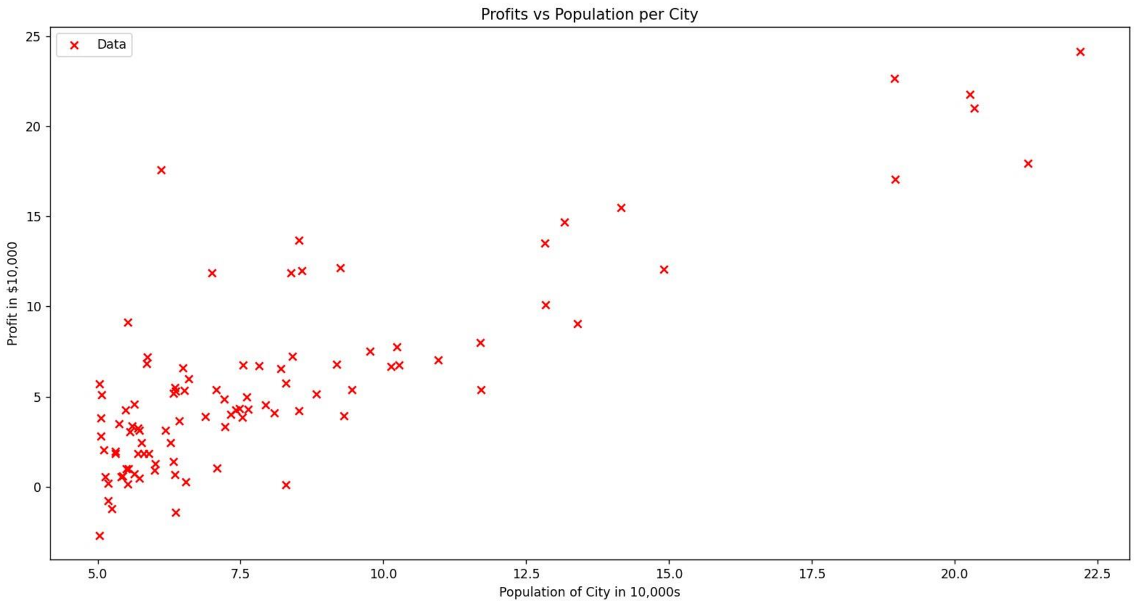
Machine Learning & Big Data Assignment – 2

Arda Harman

Let me first demonstrate you my data graph



This graph shows about real-life data…

x-axis is for the population of the city and y-axis is for the earned profit

A model gives us estimations about situations

For example, if population of a city is 5.0, model can say we will earn -$5 money (losing Money) but if the population is 10.0 it can say we will earn $7.5

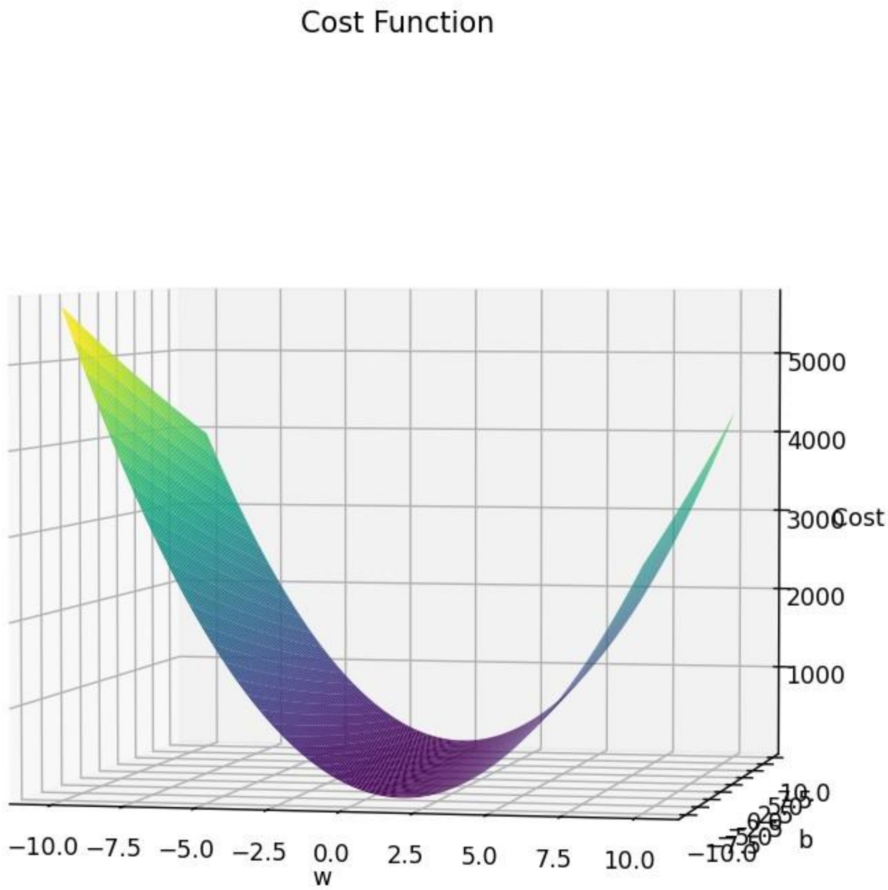
If a model is good, it's predictions are closer to the reality

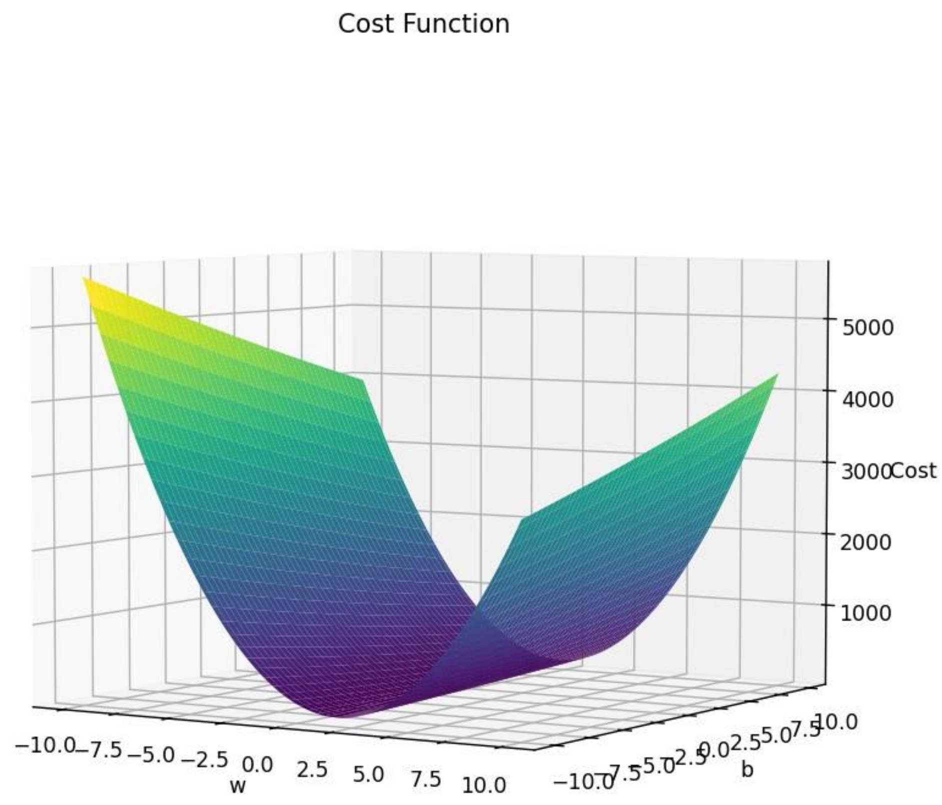
In other words, a model gets better as it's error rate decreases

Our objective is to finding the most optimal model, which has the least error rate…

- Finding "most optimal" model means, "finding the best w and b values"

- Which also means, “finding the best w and b values” which yields the least error





- There are this many w and b values… we need one with creates the smallest error (smallest value in the z-index)

In order to finding the best model, we should be able to do these 3 things:

- 1st step: Calculating the error rate
    - In order to understand "how good our model" right now, we must be able to calculate it's error rate
    - It is the J(w, b) function

```
################################################################################
# Cost function
#
def compute_cost(x, y, w, b):
    """
    Computes the cost function for linear regression.

    Args:
        x (ndarray): Shape (m,) Input to the model (Population of cities)
        y (ndarray): Shape (m,) Label (Actual profits for the cities)
        w, b (scalar): Parameters of the model

    Returns
        total_cost (float): The cost of using w,b as the parameters for linear regression
               to fit the data points in x and y
    """



    total_cost = np.sum( ( (w*x + b) - y) ** 2) / (2*(x.size))

    return total_cost


################################################################################
```

- 2nd step: Calculating the gradient
    - In order to improve our model, (after measuring it's success with first step) we must be able to take the derivative of J(w, b)
        - This makes us change our w and b inputs
        - We use the derivative of J(w, b) to change our w and b values

```
################################################################################
# Gradient function
#
def compute_gradient(x, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      x (ndarray): Shape (m,) Input to the model (Population of cities)
      y (ndarray): Shape (m,) Label (Actual profits for the cities)
      w, b (scalar): Parameters of the model
    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
     """

    dj_dw = np.sum( ( (w*x + b) - y) * x ) / x.size
    dj_db = np.sum( ( (w*x + b) - y) ) / x.size

    return dj_dw, dj_db


################################################################################
```

- 3^{rd} step: Improving until finding the best
    - We use gradient descent algorithm to find the best w and b values

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

    Args:
      x :    (ndarray): Shape (m,)
      y :    (ndarray): Shape (m,)
      w_in, b_in : (scalar) Initial values of parameters of the model
      cost_function: function to compute cost
      gradient_function: function to compute the gradient
      alpha : (float) Learning rate
      num_iters : (int) number of iterations to run gradient descent
    Returns
      w : (ndarray): Shape (1,) Updated values of parameters of the model after
          running gradient descent
      b : (scalar) Updated value of parameter of the model after
          running gradient descent
      J_history : (ndarray): Shape (num_iters,) J at each iteration,
          primarily for graphing later
    """

    w = w_in
    b = b_in
    J_history = np.zeros(num_iters)

    for iter in range(num_iters):
        # Compute the gradient
        dw, db = gradient_function(x, y, w, b)

        # Update parameters
        w -= alpha * dw
        b -= alpha * db

        # Compute cost
        J_history[iter] = cost_function(x, y, w, b)

    return w, b, J_history
```

Proof that all my functions passes the tests

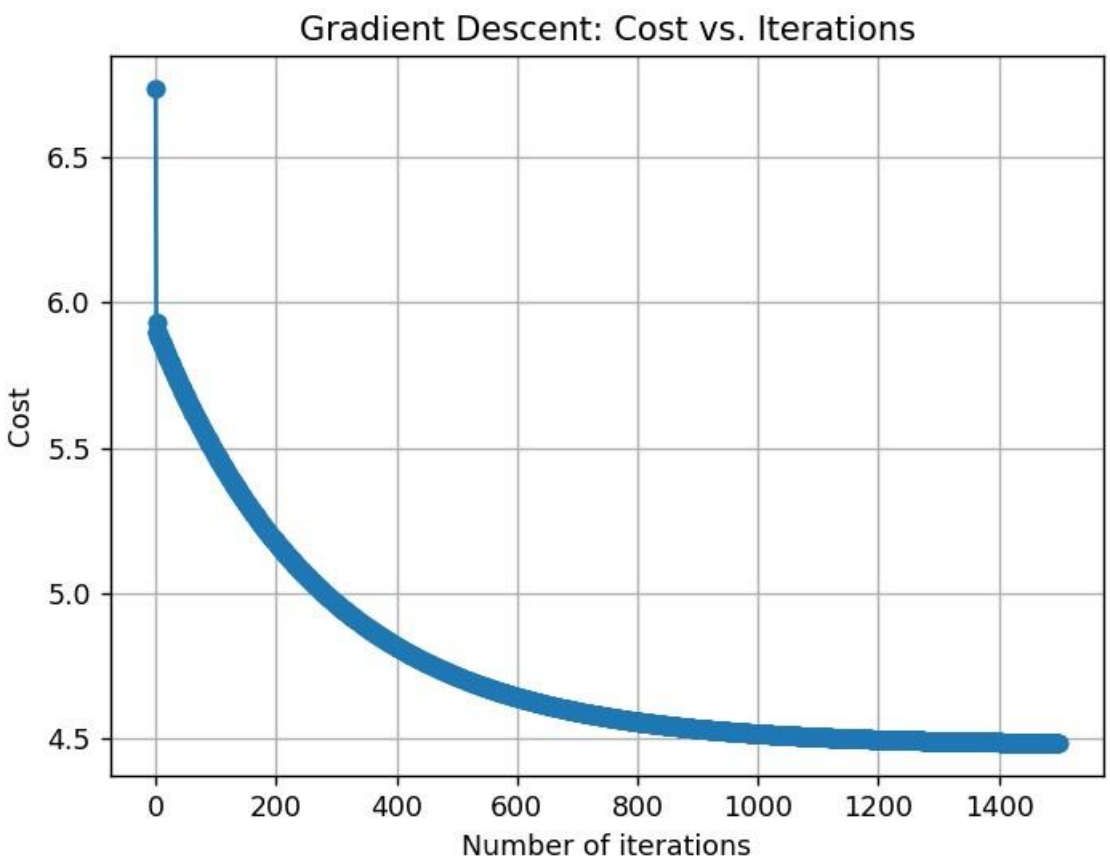
```
p1 > public_tests.py > ...
    # Case 2
    x = np.array([2, 4, 6, 8]).T
    y = np.array([4, 7, 10, 13]).T + 2
    initial_w = 1.5
    initial_b = 1
    dj_dw, dj_db = target(x, y, initial_w, initial_b)
    #assert dj_dw.shape == initial_w.shape, f"Wrong shape for dj_dw. {dj_dw} != {initial_w.shape}"
    assert dj_db == -2, f"Case 1: dj_db is wrong: {dj_db} != -2"
    assert np.allclose(dj_dw, -10.0), f"Case 1: dj_dw is wrong: {dj_dw} != -10.0"

    print("\033[92mAll tests passed!")
compute_cost_test(compute_cost)
compute_gradient_test(compute_gradient)
arr = load_data()
a = gradient_descent(arr[0], arr[1], 0, 0, cost_function=compute_cost, gradient_function=compute_gradient, alpha=0.01, num_iters=1500)
print("w: " + str(a[0]) + " b: " + str(a[1]))
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\ardah\OneDrive\Masaüstü\p1> python -u "c:\Users\ardah\OneDrive\Masaüstü\p1\public_tests.py"
All tests passed!
Using X with shape (4, 1)
All tests passed!
w: 1.166362350335582 b: -3.6302914394043597
PS C:\Users\ardah\OneDrive\Masaüstü\p1>
```

While finding the optimal values, this is how does the cost decreases continously



Finally, this is our model with the “most optimal values”