MACHINE LEARNING AND BIG DATA – ASSIGNMENT 2 – MULTIVARIABLE REGRESSION

Multivariable regression is actually pretty similiar to linear regeression. The only difference from that is we have “mutliple varibales (parameters)”.

Because of that, we will now have a model with multiple parameters

## Model

Previously: $f_{w,b}(x) = wx + b$

$$f_{w,b}(x) = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b$$

example:

$$f_{w,b}(x) = 0.1x_1 + 4x_2 + 10x_3 + -2x_4 + 80$$

size

#bedrooms

years

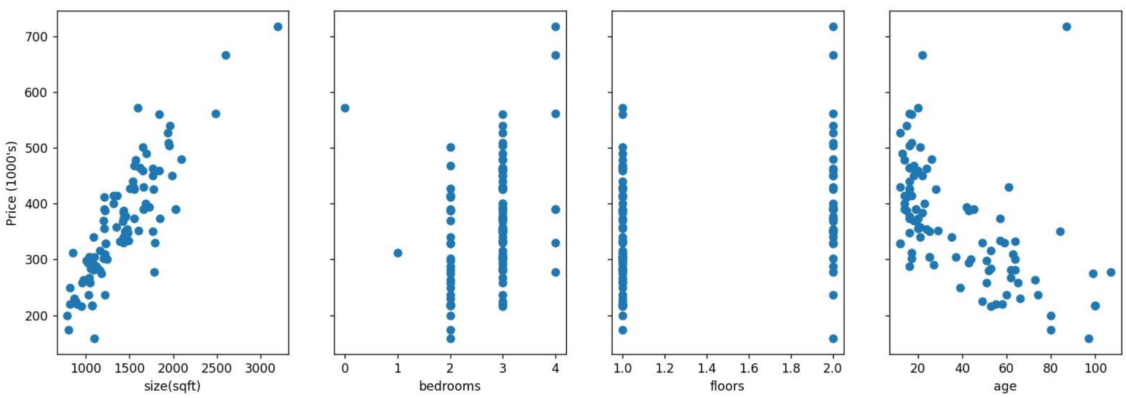
#floors

$$f_{w,b}(x) = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

$$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b = w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b$$

multiple linear regression
(not multivariate regression)

Here is how each of our variables are respect to the target




I will explain my functions one by one:

- Z-score
- Compute cost
- Compute descent
- Compute gradient descent

1) Z-score

Here, we will use this z-score function to “normalise” our X inputs

When constructing our model, we will normalise X values (which we got from houses.txt file) and use normalised X values

```
def zscore_normalize_features(X):
    """
    computes  X, zcore normalized by column

    Args:
      X (ndarray (m,n))     : input data, m examples, n features

    Returns:
      X_norm (ndarray (m,n)): input normalized by column
      mu (ndarray (n,))     : mean of each feature
      sigma (ndarray (n,))  : standard deviation of each feature
    """

    mu = np.mean(X, axis=0)  # Calculate the mean of each feature a
    sigma = np.std(X, axis=0)  # Calculate the standard deviation o
    X_norm = (X - mu) / sigma  # Normalize the input data by subtra


    return (X_norm, mu, sigma)
```

A SMALL BUT STILL AN IMPORTANT THING TO POINT OUT is that, we will also use the “sigma and mu” values in smaller sample examples…

2) Compute Cost

Compute cost, calculates the error… when we say “error” we mean “the ‘general’ rate how far our ‘estimations’ are from the ‘target’ ”

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

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

Here is my compute cost function…

```
def compute_cost(X, y, w, b):
    """
    compute 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 = len(y)  # Number of examples

    # Compute predictions
    predictions = np.dot(X, w) + b  # X * w + b

    # Compute squared error
    squared_error = np.square(predictions - y)

    # Compute mean squared error
    cost = 1 / (2 * m) * np.sum(squared_error)

    return cost
```

3) Compute gradient

Here, we are taking the derivative of the cost function (cost function is coming from compute_cost)

However, it is a little bit different to “finding derivative in multivarible” than linear (single variable function)
Since we have multiple variables, “respect to which one” will we be taking the derivative?

- ➔ Answer is, respect to all…
  - Taking derivative of each variable is taking “partial derivative”s of cost function
  - If we take the derivative of all variables (if we find all partial derivatives) (after summing them) we will have the “derivative of cost function”

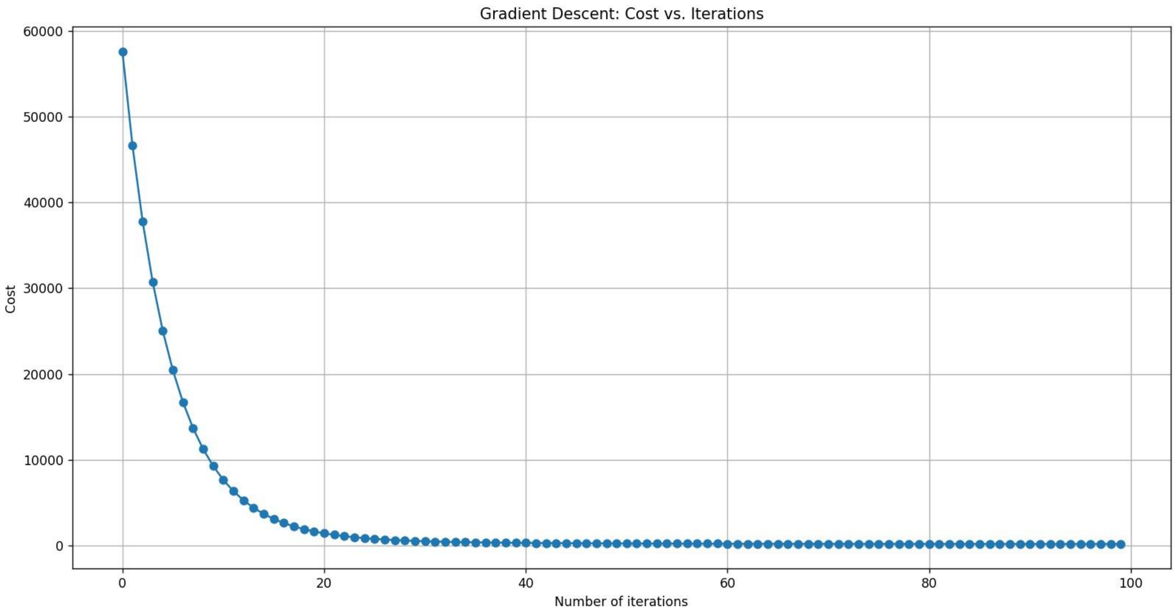
$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})x_j^{(i)}$$
$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})$$

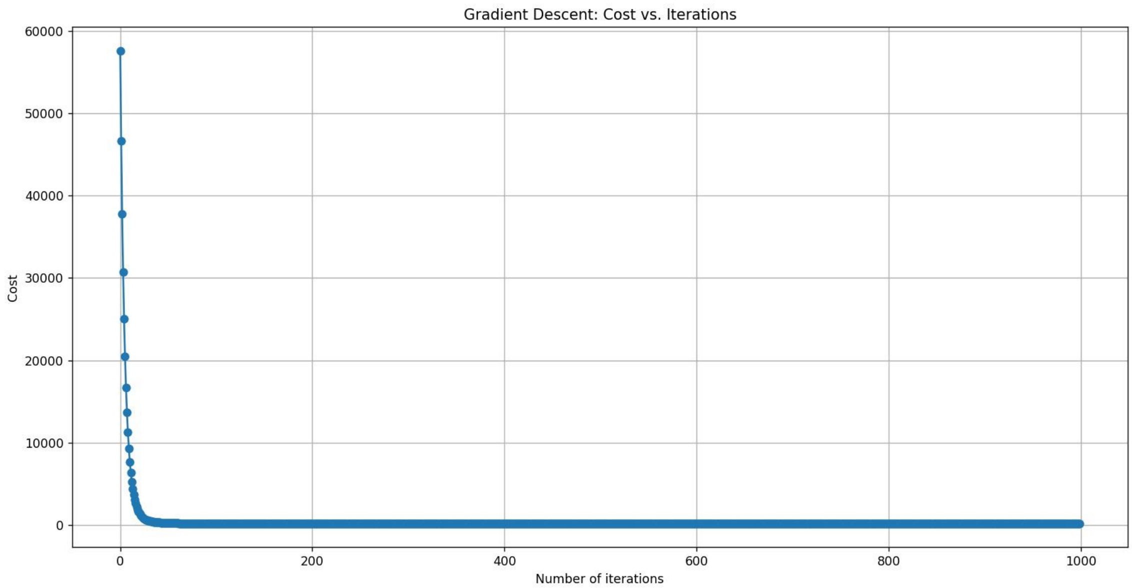
```
def compute_gradient(X, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      X : (ndarray Shape (m,n)) matrix of examples
      y : (ndarray Shape (m,))  target value of each
      w : (ndarray Shape (n,))  parameters of the mo
      b : (scalar)              parameter of the mod
    Returns
      dj_dw : (ndarray Shape (n,)) The gradient of t
      dj_db : (scalar)             The gradient of t
    """

    m = len(y)

    predictions = np.zeros(1)
    '''
    print(X)
    print(y)
    print(w)
    print(b)
    '''

    predictions = np.dot(X, w) + b  # X * w + b

    error = predictions- y

    # Compute mean squared error
    dj_dw = 1 / (m) * np.dot(X.T, error)

    dj_db = 1 / (m) * np.sum(error)

    #print("dj_db: ", dj_db)

    #print("sss")

    return dj_db, dj_dw
```

4) Compute gradient descent

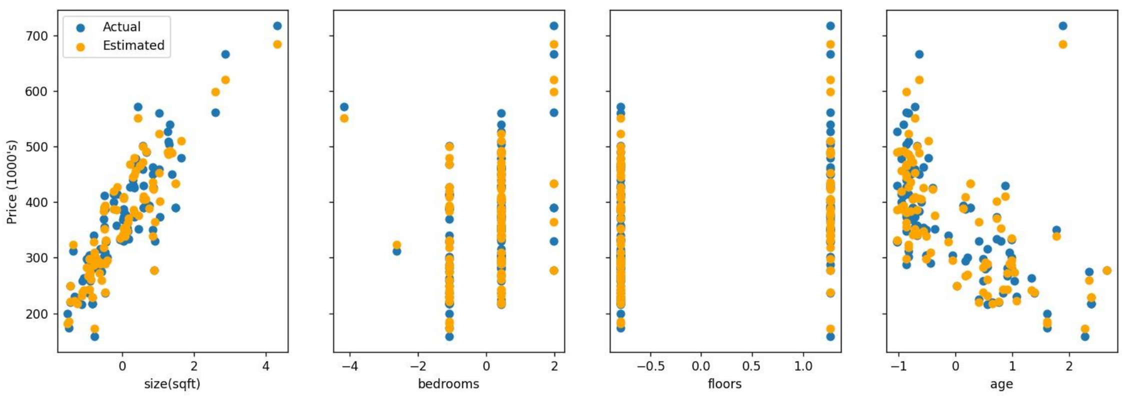
Gradient descent is same as linear regression… We just need too apply compute_gradient multiple times until we reach the best w and b values (lowest cost rate)

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

    Args:
      X : (array_like Shape (m,n)    matrix of examples
      y : (array_like Shape (m,))    target value of each e
      w_in : (array_like Shape (n,)) Initial values of para
      b_in : (scalar)                Initial value of param
      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 gradien
    Returns
      w : (array_like Shape (n,)) Updated values of paramet
          after running gradient descent
      b : (scalar)                Updated value of paramete
          after running gradient descent
      J_history : (ndarray): Shape (num_iters,) J at each i
          primarily for graphing later
    """

    # Normalize the features
    X_norm, mu, sigma = zscore_normalize_features(X)

    # Initialize parameters
    w = np.array(w_in)
    b = b_in
    J_history = np.zeros(num_iters)

    for i in range(num_iters):
        # Compute gradients on normalized data
        db, dw = gradient_function(X_norm, y, w, b)

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

        # Compute cost on normalized data
        J_history[i] = cost_function(X_norm, y, w, b)

    return w, b, J_history
```




Here is how my model performs after everything



Proof that it passes tests:

```
compute_cost_test(compute_cost)
compute_gradient_test(compute_gradient)
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

```
 390.8   354.    350.    460.    237.    288.304 282.    249.    304.
 332.    351.8   310.    216.96  666.336 330.    480.    330.3   348.
 304.    384.    316.    430.4   450.    284.    275.    414.    258.
 378.    350.    412.    373.    225.    390.    267.4   464.    174.
 340.    430.    440.    216.    329.    388.    390.    356.    257.8  ]
W:  [110.56039756 -21.26715096 -32.70718139 -37.97015909]
b:  363.15608080808056
Value test:  318.7090923199992
All tests passed!
All tests passed!
PS C:\Users\ardah\UCM> 
```

Here is my whole code…

Multi_linear_reg.py

```python
import numpy as np
import copy
import math

def zscore_normalize_features(X):
    """
    computes  X, zcore normalized by column

    Args:
      X (ndarray (m,n))     : input data, m examples, n features

    Returns:
      X_norm (ndarray (m,n)): input normalized by column
      mu (ndarray (n,))     : mean of each feature
      sigma (ndarray (n,))  : standard deviation of each feature
    """

    mu = np.mean(X, axis=0)  # Calculate the mean of each feature along the columns (axis=
    sigma = np.std(X, axis=0)  # Calculate the standard deviation of each feature along th
    X_norm = (X - mu) / sigma  # Normalize the input data by subtracting the mean and divi


    return (X_norm, mu, sigma)


def compute_cost(X, y, w, b):
    """
    compute 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 = len(y)  # Number of examples

    # Compute predictions
    predictions = np.dot(X, w) + b  # X * w + b

    # Compute squared error
    squared_error = np.square(predictions - y)

    # Compute mean squared error
    cost = 1 / (2 * m) * np.sum(squared_error)

    return cost


def compute_gradient(X, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      X : (ndarray Shape (m,n)) matrix of examples
      y : (ndarray Shape (m,))  target value of each example
      w : (ndarray Shape (n,))  parameters of the model
      b : (scalar)              parameter of the model
    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 = len(y)

    predictions = np.zeros(1)
    '''
```

```python
def compute_gradient(X, y, w, b):
    predictions = np.zeros(1)
    '''
    print(X)
    print(y)
    print(w)
    print(b)
    '''

    predictions = np.dot(X, w) + b  # X * w + b

    error = predictions- y

    # Compute mean squared error
    dj_dw = 1 / (m) * np.dot(X.T, error)

    dj_db = 1 / (m) * np.sum(error)

    #print("dj_db: ", dj_db)

    #print("sss")

    return dj_db, dj_dw


def gradient_descent(X, y, w_in, b_in, cost_function,
                     gradient_function, alpha, num_iters):
    """
    Performs batch gradient descent to learn theta. Updates thet
    num_iters gradient steps with learning rate alpha

    Args:
      X : (array_like Shape (m,n)    matrix of examples
      y : (array_like Shape (m,))    target value of each exampl
      w_in : (array_like Shape (n,)) Initial values of parameter
      b_in : (scalar)                Initial value of parameter
      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 des
    Returns
      w : (array_like Shape (n,)) Updated values of parameters o
          after running gradient descent
      b : (scalar)                Updated value of parameter of
          after running gradient descent
      J_history : (ndarray): Shape (num_iters,) J at each iterat
          primarily for graphing later
    """
    # Normalize the features
    X_norm, mu, sigma = zscore_normalize_features(X)
    # Initialize parameters
    w = np.array(w_in)
    b = b_in
    J_history = np.zeros(num_iters)

    for i in range(num_iters):
        # Compute gradients on normalized data
        db, dw = gradient_function(X_norm, y, w, b)

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

        # Compute cost on normalized data
        J_history[i] = cost_function(X_norm, y, w, b)

    return w, b, J_history
```