Lab 2: Balls Cock for MSE Calculations (11/09/2024)

Matthew Loh

Table of contents

1 Define Training Data **1**

1.1 2. Check whether MSE = 0.21606 . . . 2

1.2 3. There are TWO ways to automatically calculate MSE: . . . 3

1.2.1 a) Using scikit learn . . . 3

1.2.2 b) MSE using Numpy module . . . 3

1.3 Previous Lab Code . . . 3

1.3.1 Trials . . . 5

1.4 Gradient Descent Understanding – based on lecture material . . . 6


```python
import numpy as np
import matplotlib.pyplot as plt
```

1 Define Training Data

Lab 3 Consider the given data points: (1,1), (2,1), (3,2), (4,2), (5,4). Regression line equation: Y = 0.7X – 0.1 ## 1. Check whether the following is correct:

X	Y	y hat
1	1	0.6
2	1	1.29
3	2	1.99
4	2	2.69
5	4	3.4

```python
# Calculating from the equation
def calculate_y_hat(x):
    return 0.7 * x - 0.1


x = [1, 2, 3, 4, 5]
y = [1, 1, 2, 2, 4]
y_hat = [calculate_y_hat(x[i]) for i in range(len(x))]

print(y_hat)

# [0.6, 1.2999999999999998, 1.9999999999999996, 2.6999999999999997, 3.4]
print("The values are correct")
```

```
[0.6, 1.2999999999999998, 1.9999999999999996, 2.6999999999999997, 3.4]
The values are correct
```

1.1 2. Check whether MSE = 0.21606

```python
# Calculating MSE
def calculate_mse(y, y_hat):
    return np.mean((np.array(y) - np.array(y_hat)) ** 2)


def calculate_mse_array_inputs(y, y_hat):
    return np.mean((y - y_hat) ** 2)


print(calculate_mse(y, y_hat))
# Lab values
# Given values
Y_true = [1, 1, 2, 2, 4]  # Y_true = Y (original values)
# calculated values
Y_pred = [0.6, 1.29, 1.99, 2.69, 3.4]  # Y_pred = Y'
# Calculation of Mean Squared Error (MSE)
print(calculate_mse(Y_true, Y_pred))
```

```
0.21999999999999992
0.21606
```

1.2 3. There are TWO ways to automatically calculate MSE:

1.2.1 a) Using scikit learn

```
from sklearn.metrics import mean_squared_error

# Given values
Y_true = [1, 1, 2, 2, 4]  # Y_true = Y (original values)
# calculated values
Y_pred = [0.6, 1.29, 1.99, 2.69, 3.4]  # Y_pred = Y'
# Calculation of Mean Squared Error (MSE)
mean_squared_error(Y_true, Y_pred)
```

```
0.21606
```

1.2.2 b) MSE using Numpy module

```
import numpy as np
# Given values
Y_true = [1, 1, 2, 2, 4]  # Y_true = Y (original values)
# calculated values
Y_pred = [0.6, 1.29, 1.99, 2.69, 3.4]  # Y_pred = Y'
# Mean squared Error
MSE = np.square(np.subtract(Y_true, Y_pred)).mean()
MSE
```

```
0.21606
```

4. Use both functions to calculate MSE for the different W__s and b__s in your previous lab code from last week

1.3 Previous Lab Code

```
# From previous lab code
# x_train is the input variable (size in 1000 square feet)
# y_train is the target (price in 1000s of dollars)
x_train = np.array([1.0, 2.0])
```

```python
y_train = np.array([300.0, 500.0])
print(f"x_train: {x_train}")
print(f"y_train: {y_train}")


def compute_model_output(x: np.ndarray, w: float, b: float) -> np.ndarray:
    """
    Computes the prediction of a linear model
    Args:
      x (np.ndarray (m,)): Data, m examples
      w,b (scalar) : Model parameters
    Returns:
      y (ndarray (m,)) : target values
    """
    m = x.shape[0]
    f_wb = np.zeros(m)
    for i in range(m):
        f_wb[i] = w * x[i] + b
    return f_wb


from sklearn.metrics import mean_squared_error
import numpy as np


def scikit_learn_mse(y_true, y_pred):
    return mean_squared_error(y_true, y_pred)


def numpy_mse(y_true, y_pred):
    return np.square(np.subtract(y_true, y_pred)).mean()


def output(x_train, y_train, w, b):
    y_pred = compute_model_output(x_train, w, b)
    print(f"y_pred: {y_pred}")
    print(f"scikit learn mse: {scikit_learn_mse(y_train, y_pred)}")
    print(f"numpy mse: {numpy_mse(y_train, y_pred)}")


print(f"y_true: {y_train}")
```

```
x_train: [1. 2.]
y_train: [300. 500.]
y_true: [300. 500.]
```

1.3.1 Trials

```
w = 100
b = 100
output(x_train, y_train, w, b)
```

```
y_pred: [200. 300.]
scikit learn mse: 25000.0
numpy mse: 25000.0
```

```
w = 150
b = 100
output(x_train, y_train, w, b)
```

```
y_pred: [250. 400.]
scikit learn mse: 6250.0
numpy mse: 6250.0
```

```
w = 50
b = 100
output(x_train, y_train, w, b)
```

```
y_pred: [150. 200.]
scikit learn mse: 56250.0
numpy mse: 56250.0
```

```
w = 200
b = 100
output(x_train, y_train, w, b)
```

```
y_pred: [300. 500.]
scikit learn mse: 0.0
numpy mse: 0.0
```

```python
w = 198
b = 102.5
output(x_train, y_train, w, b)
```

```
y_pred: [300.5 498.5]
scikit learn mse: 1.25
numpy mse: 1.25
```

1.4 Gradient Descent Understanding – based on lecture material

```python
def compute_cost(x: np.ndarray, y: np.ndarray, w: float, b: float) -> float:
    """
    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
    """
    # number of training examples
    m = x.shape[0]

    # You need to return this variable correctly
    total_cost = 0

    ### START CODE HERE ###
    cost = 0
    for i in range(m):
        f_wb = w * x[i] + b
        cost += (f_wb - y[i]) ** 2

    total_cost = cost / (2 * m)

    ### END CODE HERE ###

    return total_cost
```

```
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
    """# Number of training examples
    m = x.shape[0]

    # You need to return the following variables correctly
    dj_dw = 0
    dj_db = 0

    ### START CODE HERE ###
    for i in range(m):
        f_wb = w*x[i]+b
        dj_db += f_wb - y[i]
        dj_dw += (f_wb - y[i])*x[i]
    dj_dw /= m
    dj_db /= m

    ### END CODE HERE ###

    return dj_dw, dj_db
```

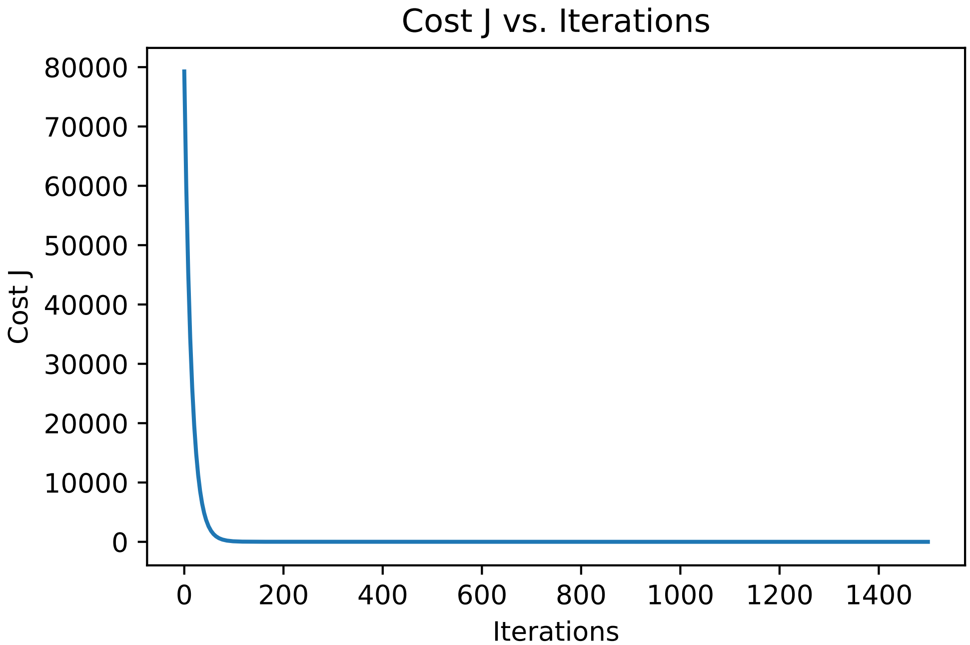
```
import math
import copy


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,)
```

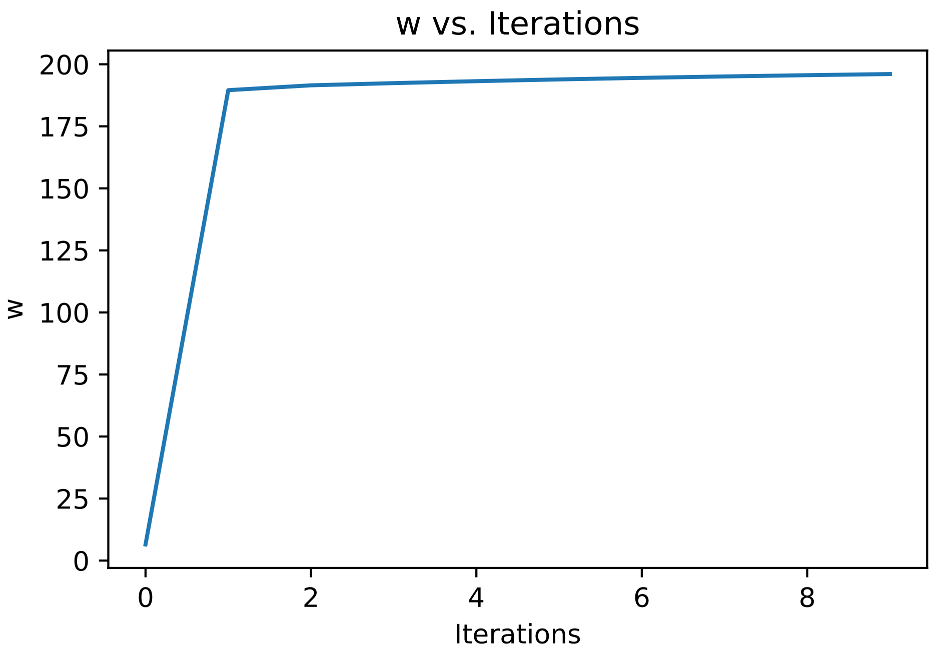
```
      y :    (ndarray): Shape (m,)
      w_in, b_in : (scalar) Initial values of parameters of the model
      cost_function: function to compute cost
      gradient_function: function to compute the gradient
      alpha : (float) Learning rate
      num_iters : (int) number of iterations to run 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
    """

    # 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 = []
    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_dw, dj_db = gradient_function(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
            cost = cost_function(x, y, w, b)
            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:
            w_history.append(w)
            print(f"Iteration {i:4}: Cost {float(J_history[-1]):8.2f}   ")

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

```python
# initialize fitting parameters. Recall that the shape of w is (n,)
initial_w = 0.0
initial_b = 0.0

# some gradient descent settings
iterations = 1500
alpha = 0.01

w, b, j_history, w_history = gradient_descent(
    x_train,
    y_train,
    initial_w,
    initial_b,
    compute_cost,
    compute_gradient,
    alpha,
    iterations,
)
print("w,b found by gradient descent:", w, b)


# Plotting J_history and w_history

plt.plot(j_history)
plt.xlabel("Iterations")
plt.ylabel("Cost J")
plt.title("Cost J vs. Iterations")
plt.show()

plt.plot(w_history)
plt.xlabel("Iterations")
plt.ylabel("w")
plt.title("w vs. Iterations")
plt.show()
```

```
Iteration    0: Cost 79274.81
Iteration  150: Cost    14.07
Iteration  300: Cost     9.48
Iteration  450: Cost     7.62
Iteration  600: Cost     6.12
Iteration  750: Cost     4.92
Iteration  900: Cost     3.95
```

```
Iteration 1050: Cost     3.17
Iteration 1200: Cost     2.55
Iteration 1350: Cost     2.05
w,b found by gradient descent: 196.46688050502766 105.71670742918006
```






```
def compute_multiple_features(x_1, x_2, x_3, x_4):
    w_1 = 0.1
    w_2 = 4
    w_3 = 10
    w_4 = -2
    b = 80
    return w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4 + b

compute_multiple_features(1416, 3, 2, 40)
```

```
173.6
```

```
import numpy as np


w = np.array([1.0, 2.5, -3.3])
b = 4
x = np.array([10, 20, 30])

f = w[0] * x[0] + w[1] * x[1] + w[2] * x[2] + b
print(f)
```

```python
f = 0

for j in range(0, len(w)):
    f += w[j] * x[j]
f += b
print(f)

f = np.dot(w, x) + b
print(f)
```

```
-35.0
-35.0
-34.99999999999999
```