Lab 5: Multiple Variable Prediction Using Linear Regression and Gradient Descent Understanding (25/09/2024)

Matthew Loh

Table of contents

Model Prediction with Multiple Variables **2**
Testing and Predicted Output 2
Single Prediction, Vector 3
Testing and Predicted Output 3
Testing and Predicted Output 4
5.1 Compute Gradient With Multiple Variables 4
Testing and Predicted Output 5
5.2 Gradient Descent With Multiple Variables 5
Testing and Predicted Output 6

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

```
X_train = np.array([[2104, 5, 1, 45], [1416, 3, 2, 40], [852, 2, 1, 35]])
y_train = np.array([460, 232, 178])
```

```
# data is stored in numpy array / matrix
print(f"X Shape: {X_train.shape}, X Type: {type(X_train)}")
print(X_train)
print(f"y Shape: {y_train.shape}, y Type: {type(y_train)}")
print(y_train)
```

```
X Shape: (3, 4), X Type: <class 'numpy.ndarray'>
[[2104    5    1   45]
```

```
 [1416    3    2   40]
 [ 852    2    1   35]]
y Shape: (3,), y Type: <class 'numpy.ndarray'>
[460 232 178]
```

```
b_init = 785.1811367994083
w_init = np.array([0.39133535, 18.75376741, -53.36032453, -26.42131618])
print(f"w_init shape: {w_init.shape}, b_init type: {type(b_init)}")
```

```
w_init shape: (4,), b_init type: <class 'float'>
```

Model Prediction with Multiple Variables

```
def predict_single_loop(x, w, b):
    """
    single predict using linear regression
    Args:
      x (ndarray): Shape (n,) vector of input values
      w (ndarray): Shape (n,) vector of coefficients
      b (scalar):  intercept
    Returns:
      p (scalar): prediction
    """
    n = x.shape[0]
    p = 0
    for i in range(n):
        p_i = x[i] * w[i]
        p = p + p_i
    p = p + b
    return p
```

Testing and Predicted Output

```
# get a row from our training data
x_vec = X_train[0]
print(f"x_vec shape {x_vec.shape}, x_vec value: {x_vec}")
```

```
# make a prediction
f_wb = predict_single_loop(x_vec, w_init, b_init)
print(f"f_wb shape {f_wb.shape}, prediction: {f_wb}")
```

```
x_vec shape (4,), x_vec value: [2104    5    1   45]
f_wb shape (), prediction: 459.9999976194083
```

Single Prediction, Vector

```
def predict(x, w, b):
    """
    single predict using linear regression
    Args:
      x (ndarray): Shape (n,) example with multiple features
      w (ndarray): Shape (n,) model parameters
      b (scalar):  model parameter
    Returns:
      p (scalar): prediction
    """
    p = np.dot(x, w) + b
    return p
```

Testing and Predicted Output

```
# get a row from our training data
x_vec = X_train[0, :]
print(f"x_vec shape {x_vec.shape}, x_vec value: {x_vec}")

# make a prediction
f_wb = predict(x_vec, w_init, b_init)
print(f"f_wb shape {f_wb.shape}, prediction: {f_wb}")
```

```
x_vec shape (4,), x_vec value: [2104    5    1   45]
f_wb shape (), prediction: 459.99999761940825
```

```
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 = X.shape[0]
    cost = 0.0
    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
    return cost
```

Testing and Predicted Output

```
# Compute and display cost using our pre-chosen optimal parameters.
cost = compute_cost(X_train, y_train, w_init, b_init)
print(f"Cost at optimal w: {cost}")
```

```
Cost at optimal w: 1.5578904880036537e-12
```

5.1 Compute Gradient With Multiple Variables

```
def compute_gradient(X, y, w, b):
    """
    Compute 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  # (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

    return dj_db, dj_dw
```

Testing and Predicted Output

```
tmp_dj_db, tmp_dj_dw = compute_gradient(X_train, y_train, w_init, b_init)
print(f"dj_db at initial w,b: {tmp_dj_db}")
print(f"dj_dw at initial w,b: {tmp_dj_dw}")
```

```
dj_db at initial w,b: -1.673925169143331e-06
dj_dw at initial w,b: [-2.72623581e-03 -6.27197272e-06 -2.21745580e-06 -6.92403399e-05]
```

5.2 Gradient Descent With Multiple Variables

```
import copy
import math


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 (m,n))  : Data, m examples with n features
      y (ndarray (m,))   : target values
      w_in (ndarray (n,)) : initial model parameters
      b_in (scalar)      : initial model 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 descent

    Returns:
      w (ndarray (n,)) : updated model parameters
      b (scalar)       : updated model 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 = 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
            J_history.append(cost_function(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]:8.2f}   ")

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

Testing and Predicted Output

```
# initialize parameters
initial_w = np.zeros_like(w_init)
```

```python
initial_b = 0.
# some gradient descent settings
iterations = 1000
alpha = 5.0e-7
# run gradient descent
w_final, b_final, J_hist = gradient_descent(
    X_train, y_train, initial_w, initial_b, compute_cost, compute_gradient, alpha, iterations
print(f"b,w found by gradient descent: {b_final:0.2f}, {w_final}")
m, _ = X_train.shape
for i in range(m):
    print(f"prediction: {
          np.dot(X_train[i], w_final) + b_final:0.2f}, target value: {y_train[i]}")
```

```
Iteration    0: Cost  2529.46
Iteration  100: Cost   695.99
Iteration  200: Cost   694.92
Iteration  300: Cost   693.86
Iteration  400: Cost   692.81
Iteration  500: Cost   691.77
Iteration  600: Cost   690.73
Iteration  700: Cost   689.71
Iteration  800: Cost   688.70
Iteration  900: Cost   687.69
b,w found by gradient descent: -0.00, [ 0.20396569  0.00374919 -0.0112487  -0.0658614 ]
prediction: 426.19, target value: 460
prediction: 286.17, target value: 232
prediction: 171.47, target value: 178
```