Polynomial Linear Regression Implementation

Summary

Here I have written code that performs polynomial linear regression using gradient descent and then visualizes the results.

Implementation Overview

Imports

Begin with importing numpy and the matplotlib.pyplot libraries. Numpy will be used for numerical operations, and pyplot will be used to plot and visualize the resulting output. We will also be importing matplotlib to visualize these results.

[Scikit Learn's Diabetes Dataset]([https://scikit-learn.org/stable/datasets/toy_dataset.html#diabetes-dataset](Scikit Learn's)) is 11 columns of data. The first 10 are numeric predictive values. Each of these 10 feature variables have been mean centered and scaled by the standard deviation times the square root of `n_samples` (i.e. the sum of squares of each column totals 1). Ultimately, this means I do not need to scale / standardize any of the data.

```
# Math
import math
import numpy as np
import matplotlib.pyplot as plt

# Dataset
from sklearn.datasets import load_diabetes
```

Load / Split Data

Here, we simply load the data into variables and then concatenate into `data`.

```
# Load Data
def load_data():
  dataset = load_diabetes()
  X = dataset.data
  y = dataset.target.reshape(-1, 1)

  # Concatenate
  data = np.hstack((X, y))

  return data
```

Shuffle and split is simply shuffling the rows prior to assigning a percentage of the set to variables training_set, dev_set, and test_set. We are using 60/20/20% splits, respectively.

```
# Shuffle and Create Split Training/Dev/Test 70/15/15%
def shuffle_and_split(data):
  np.random.shuffle(data)

  total_sample = len(data)
  training_set = data[:int(total_sample*0.6)]
  dev_set = data[int(total_sample*0.6):int(total_sample*0.8)]
  test_set = data[int(total_sample*0.8):]
  return total_sample, training_set, dev_set, test_set
```

You can verify loading and shuffling works correctly by running this cell:

```
# Run Cell to Visualize Data and Shuffle (random set)
data_2 = load_data()

# Data Types
def visualize_data():
  print(f"X is of type {type(data_2[0])}")
  print(data_2[0][:10])
  print()
  print(f"y is of type {type(data_2[1])}")
  print(data_2[1][:10])
  print()
  print("Shape:")
  print(data_2.shape)
  print()

# Shuffle
```

```
def visualize_shuffle():
  temp = data_2[10:15]
  print("Initial Data:")
  print(temp)
  print()
  print("Shuffled:")
  np.random.shuffle(temp)
  print(temp)

visualize_data()
visualize_shuffle()
```

We can then have code to check for any missing values:

```
# Check for Missing Values
def print_missing_values(data):
  missing_values = np.isnan(data).sum()
  if missing_values:
    print(f"Missing Values: {missing_values}")
  else:
    print("No Missing Values. Continuing...")
```

and also for data leaks:

```
# Check for Data Leaks
def data_leaking_check(data1, data2):
  data_leaking = False
  for d1 in data1:
    for d2 in data2:
      if (np.array_equal(d1, d2)):
        data_leaking = True
        print("Find same sample: ")
        print(d1)
  if (not data_leaking):
    print("No Data Leaking.")
```

While getting features we also reshape the labels so we don't get a shape mismatch later:

```
# Get Features and Labels
def get_features_and_labels(data):
  features = data[:, :-1]
  labels = data[:, -1]
  # Reshape labels
  labels = labels.reshape(-1, 1)

  # Check shapes
  # print("Shape of features:", features.shape)
  # print("Shape of labels:", labels.shape)

  return features, labels
```

Polynomial Gradient Descent

Here we calculate our *polynomial linear regression*:

```
# Regression Model
def perform_regression(theta, input):
  prediction = np.dot(input, theta)
  return prediction
```

For the *cost function* we are using mean squared error:

```
# Cost Function
def compute_cost(Y_pred, Y_true, length):
  m = length
  diff = Y_pred - Y_true
  squared_diff = np.square(diff)
  J = 1 / (2 * m) * np.sum(squared_diff)

  return J
```

And for *Gradient Descent* we are updating our theta values as follows:

```
# Gradient Descent
def update_theta(theta, X, Y_true, Y_pred, learning_rate, length):
  m = length
  number_features = X.shape[1]

  # Iterate over each coefficient and update based on error
  for j in range(number_features):
    error = Y_pred - Y_true
    gradient = np.sum(error * X[:, j]) / m
    theta[j] = theta[j] - learning_rate * gradient
  return theta
```

Plots

As we iterate through we will keep track of the cost_history and number of iterations. This will eventually be plotted to show how the cost changes across iterations.

```
# Plot Cost
def plot_cost(k, iterations, cost_history):
  plt.plot(iterations, cost_history)
  plt.title(f'Cost Over {k} Iterations')
  plt.xlabel('Iteration')
  plt.ylabel('Cost')
```

Get Regression

Here we are tying together most of the prior code into one function and initializing a theta of all zeroes if a theta value was not provided.

```
# Get Regression Analysis

def get_regression_analysis(data, learning_rate, run_type, number_epochs, degree=1,
theta=[]):
  # Check for missing values
  print_missing_values(data)

  # Shuffle and Split 60/20/20
  total_sample, training_set, dev_set, test_set = shuffle_and_split(data)

  # Check Data
  data_leaking_check(training_set, dev_set)

  # Get Features/Labels
  train_x, train_y = get_features_and_labels(training_set)
  dev_x, dev_y = get_features_and_labels(dev_set)
  test_x, test_y = get_features_and_labels(test_set)

  # Generate Polynomial Features
  train_x_poly = generate_polynomial_features(train_x, degree)
  dev_x_poly = generate_polynomial_features(dev_x, degree)
  test_x_poly = generate_polynomial_features(test_x, degree)
```

```
    # Initialize theta with appropriate dimensions if no theta given
    if not theta:
      theta = np.zeros(train_x_poly.shape[1])
```

Next, I have defined a switch that will allow me to easily swap between train, dev, and test datasets (without having to manually change all of the variables).

```
    def switch(run_type):
      return {
          'train': (train_x_poly, train_y),
          'dev': (dev_x_poly, dev_y),
          'test': (test_x_poly, test_y)
      }.get(run_type, 'dev')
```

Then I am initializing all the variables used as well as the figure. `cost history` and `iteration_count` will be plotted to show how the cost changes across epochs. I am keeping track of the theta and learning rates associated with the lowest cost and will print them so they can be used if needed.

```
    plt.figure(figsize=(30, 30))
    k = 0
    length = len(switch(run_type)[1])
    cost_history = []
    iteration_count = []
    lowest_cost = 0
    best_theta_temp = theta
    best_learning_rate = learning_rate
    previous_cost = 0
    converged = False
    counter = 0
```

Lastly, the code that runs the regression and prints the results begins by getting predictions and costs. If the cost goes up from epoch to epoch, the learning rate is adjusted based on whether the epoch is divisible by two. If the cost goes down, we track the variables used to cause it to go down and then perform a check. If the change in cost is less than 0.001 we declare convergence and print the results. Otherwise, we continue on (assuming k<number epochs). Note there is code suppressing Python's string representation of floats.

```
  # Start Regression
    print("Beginning...")
    while not converged and k < number_epochs:
      prediction = perform_regression(theta, switch(run_type)[0])
      cost = compute_cost(prediction, switch(run_type)[1], length)
      if k == 0:
        previous_cost = cost
        lowest_cost = cost
      cost_change = cost - previous_cost

      if cost_change > 0:
        print("Cost went up.")
        if k % 2 == 0:
          updated_learning_rate = learning_rate * 10
          print(f"Adjusting Learning Rate * 10...\n{learning_rate} is now
  {updated_learning_rate}.")
          learning_rate = updated_learning_rate
        else:
          updated_learning_rate = learning_rate * 3
          print("Adjusting something Learning Rate * 3")
          learning_rate = updated_learning_rate
      else:
        # Cost went down
        best_theta_temp = theta
        best_learning_rate = learning_rate
        # If the absolute value of the change is <.001 would work here
        if cost_change > -0.001:
          # Deal with first pass
          if cost_change == 0:
            print("First pass...")
          else:
            print("Convergence! Getting results...")
            converged = True




      theta = update_theta(theta, switch(run_type)[0], switch(run_type)[1], prediction,
  learning_rate, length)
      cost_history.append(cost)

      # keep track of the lowest cost
      lowest_cost = cost if cost < lowest_cost else lowest_cost

      k += 1
      iteration_count.append(k)
```

```
  print(f"Lowest Cost of {lowest_cost} was acheived after {k} epochs.")
  print("Theta Values:")
  best_theta = []

  for value in best_theta_temp:
    # Suppress scientific notation for copy/paste
    value = f'{value:.10f}'
    best_theta.append(float(value))

  print(best_theta)
  print(f"Learning Rate: {best_learning_rate}")
  plot_cost(k, iteration_count, cost_history)
```

Note: I ended up removing the tracking of best_theta and best_learning_rate as they were not actually needed. I also removed the attempt at suppressing scientific notation as it was not working. Unfortunately, that meant I was not able to easily copy in the values I needed. Given more time, this is something I would want to fix.

Running the Program

Here we can define our hyperparameters as well as which dataset we want to run. We can also define a list of theta values or leave it empty and it will be initialized with zeroes.

```
# Run the Program:
# Hyperparameters

# 'dev', 'train', 'test' for the respective datasets
run_type = 'dev'

learning_rate = 0.030
polynomial_degree = 10
number_epochs = 5000

data = load_data()

# Here you can specify a theta
theta = []

get_regression_analysis(data, learning_rate, run_type, number_epochs, polynomial_degree,
theta)
```

Results

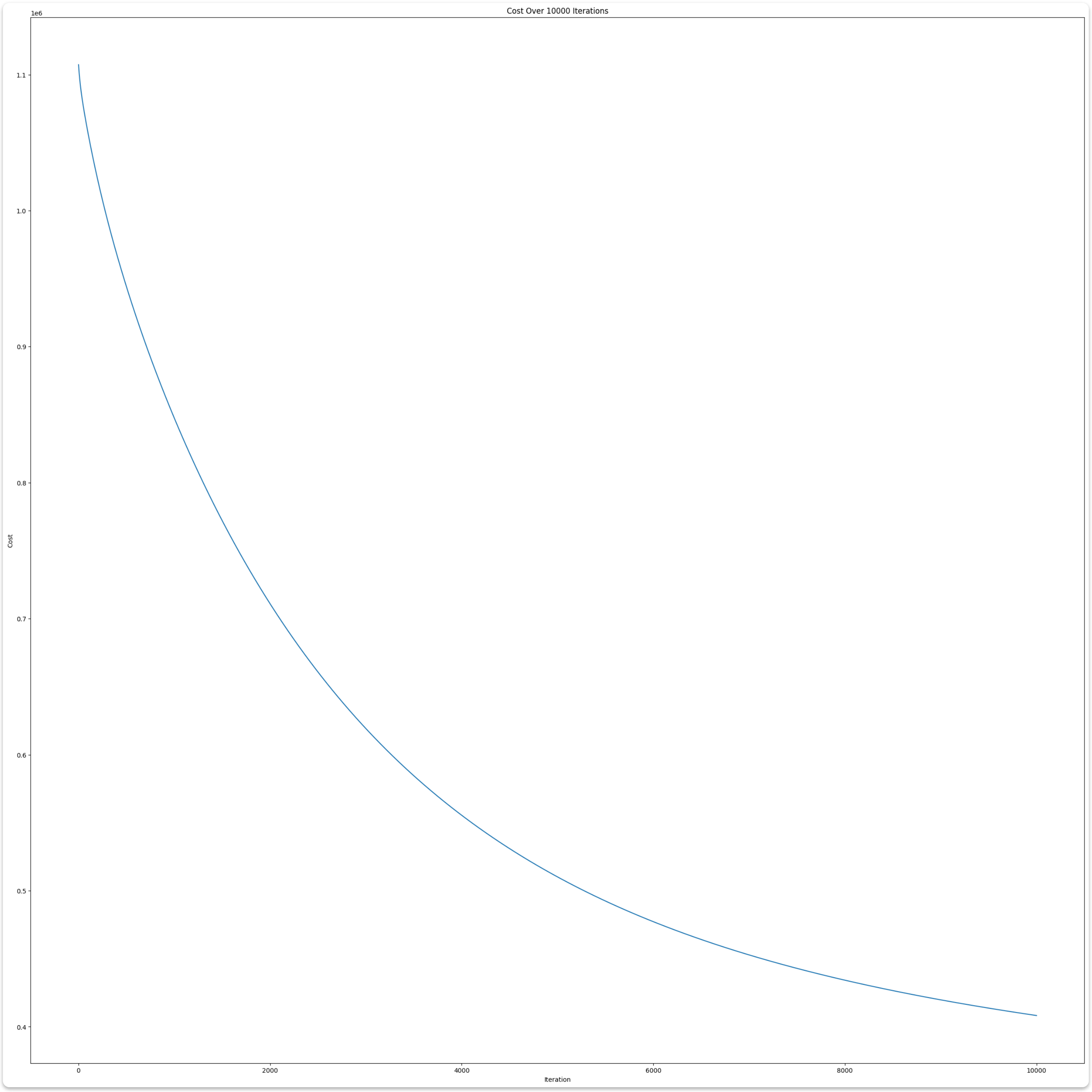
After running the dev set with the following:

```
learning_rate = 0.030
polynomial_degree = 10
number_epochs = 10000
```

I got the following output:

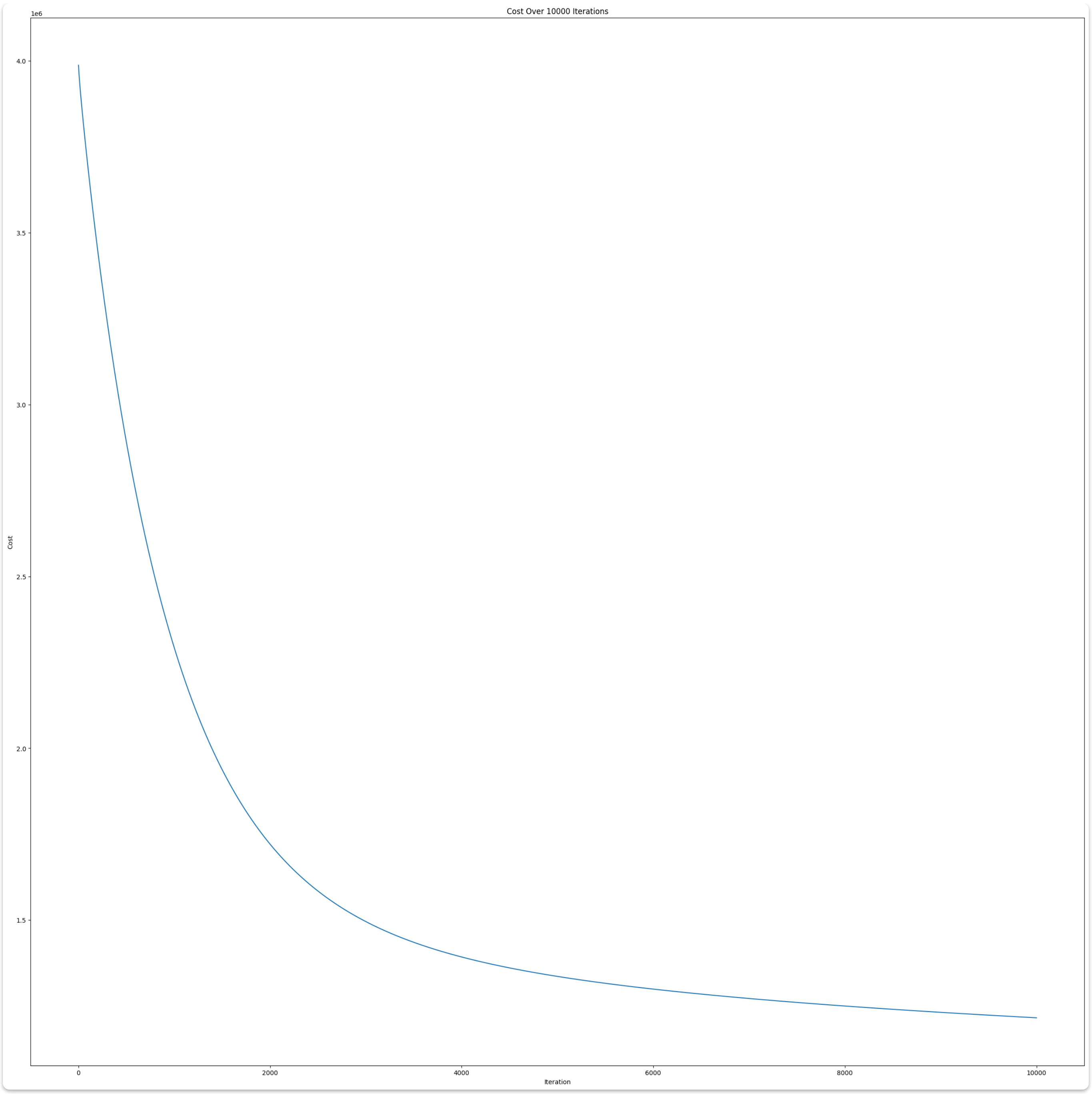
```
No Missing Values. Continuing...
No Data Leaking.
Beginning...
First pass...
Lowest Cost of 408215.62606002047 was acheived after 10000 epochs.
Theta Values: [374.3104210582, 4074.9953009513, -26.469461613, 21.4805819392,
-0.1812668604, 0.161046712, -0.0005681594, 0.0014327552, 9.1288e-06, 1.403e-05,
-50.6396630882, 4574.2279739027, 27.5068249758, 10.5150315416, 0.1257274668,
0.0245488672, 0.0004326892, 5.81532e-05, 1.3301e-06, 1.396e-07, 125.784405068,
3810.2120313161, 60.3341462852, 23.1156029651, 1.5057005783, 0.3043393424, 0.0381691766,
0.0066289701, 0.0010390355, 0.0001765211, 94.11522243, 4061.6359718569, 7.1659563271,
20.848639181, -0.1052738051, 0.1546503961, -0.0025219077, 0.0014027391, -3.92155e-05,
1.42451e-05, -842.2562332172, 3843.933475955, 120.833009068, 25.1497437249, 2.0327909568,
0.2856586395, 0.031503465, 0.0040525376, 0.0004955458, 6.28812e-05, 836.8473268289,
3407.6393893838, 105.3196859107, 19.0982470542, 1.1966634665, 0.0869876209, -0.007958798,
-0.0045815254, -0.0012804991, -0.0003081229, -502.1490766908, 4053.8994090059,
135.2586102768, 30.8886561147, 3.1811050822, 0.5672072187, 0.0862255082, 0.0149025202,
0.0024944882, 0.0004304518, -1379.0934751551, 3693.0581550684, 106.3508353239,
22.0014323008, 1.5592139654, 0.2103997585, 0.0177452358, 0.0015568718, -8.6469e-06,
-3.7861e-05, 432.237172923, 3759.4769941533, 70.2984298696, 21.0376438902, 1.0272166588,
0.2004728436, 0.0157407418, 0.0025762113, 0.0002631013, 3.92e-05, -46.9280038272,
2957.2308376032, 15.6228500723, 11.8706619667, 0.1473314854, 0.0644609812, 0.0010430511,
0.0004062616, 5.9751e-06, 2.8039e-06]
Learning Rate: 0.03
```

and as you can see from the following image, the cost was apparently heading towards convergence:



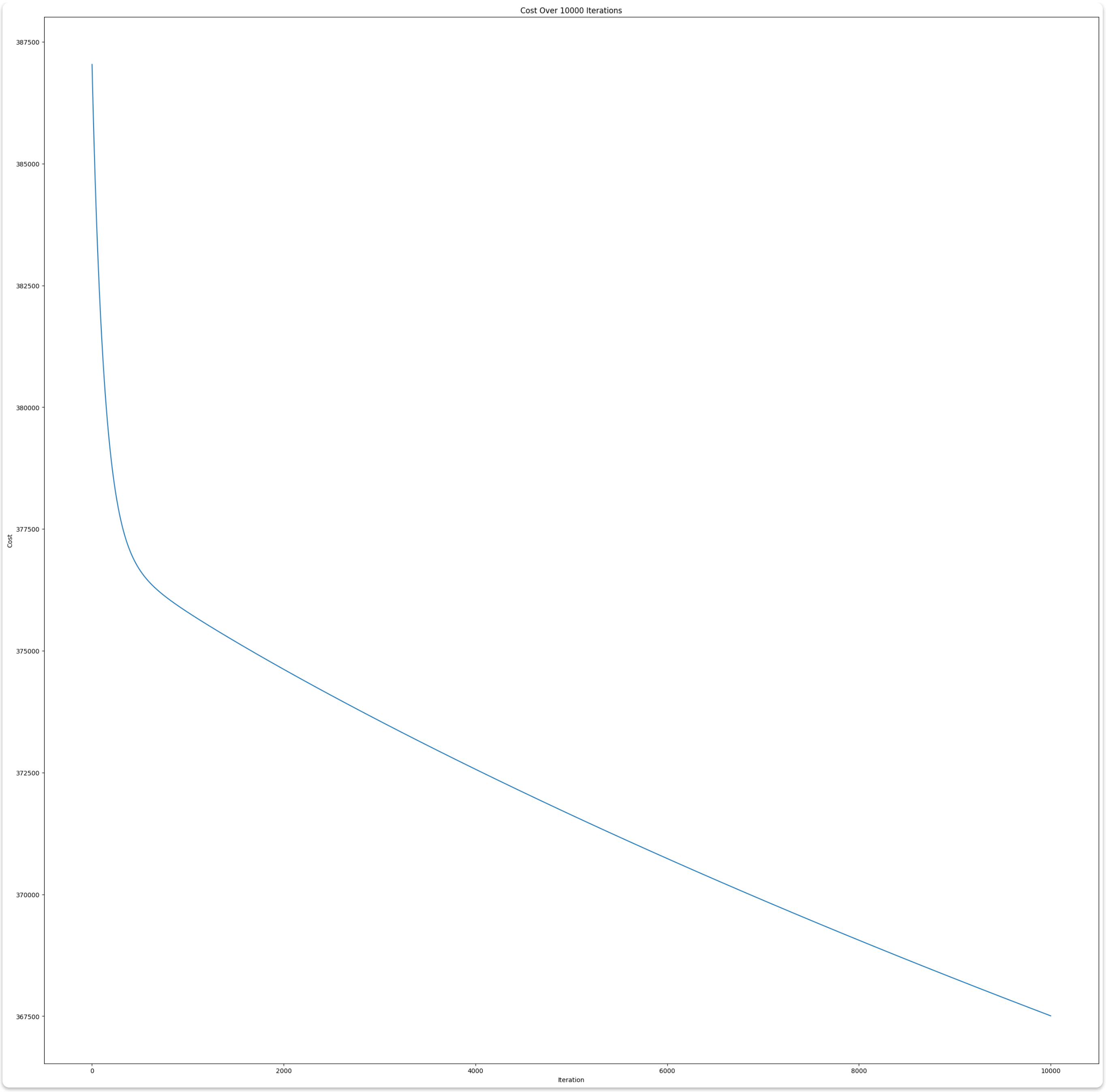
Since the model seemed to be working I loaded the training dataset and sent 10,000 epochs to find the ideal parameters. The following output and graph were obtained:

```
No Missing Values.
Continuing...
No Data Leaking.
Beginning...
First pass...
Lowest Cost of 1215820.3799123834 was acheived after 10000 epochs.
Theta Values: [ 2.73029299e+02 6.49328774e+03 -9.24955025e+01 2.51059887e+01
-8.02429349e-01 1.34747006e-01 -6.34511273e-03 8.44916250e-04 -4.91803019e-05
5.79749315e-06 -3.83249973e+01 9.27460391e+03 5.59178228e+01 2.13209255e+01 2.55256941e-
01 4.97787526e-02 8.78092505e-04 1.17923958e-04 2.69871635e-06 2.83092436e-07
-1.74436562e+02 6.02252244e+03 1.29662828e+02 2.39521214e+01 1.18171488e+00 1.54845933e-
01 1.24983035e-02 1.64975504e-03 1.95704512e-04 2.91550199e-05 -3.24176642e+01
5.82610866e+03 7.78256901e+01 2.19173616e+01 7.62470045e-01 1.30076416e-01 8.04486586e-03
1.04266487e-03 9.11759993e-05 1.02870603e-05 -1.92923038e+03 2.65623341e+03
-1.39287004e+02 -3.19123734e+01 -4.97132522e+00 -9.64136497e-01 -1.30109868e-01
-2.25182238e-02 -3.13470176e-03 -5.11923035e-04 2.46510633e+03 2.04474765e+03
-3.53144989e+02 -7.38001275e+01 -1.53875594e+01 -3.06254313e+00 -5.89553798e-01
-1.15763021e-01 -2.25195280e-02 -4.42682450e-03 -3.77199801e+02 6.56631828e+03
-4.03425643e+01 -2.74600728e+00 -5.26852944e+00 -1.02095330e+00 -2.10377721e-01
-3.89208717e-02 -7.17605403e-03 -1.29995702e-03 -1.58350758e+03 4.08040391e+03
-1.44827300e+02 -3.29449968e+01 -8.54003133e+00 -1.67688053e+00 -3.21354970e-01
-6.02053291e-02 -1.12053967e-02 -2.07851876e-03 8.12778702e+02 7.00673246e+03
1.20876839e+02 4.49470384e+01 2.29051307e+00 5.21424194e-01 4.02854182e-02 7.56324689e-03
6.99643005e-04 1.19637042e-04 -2.25473708e+01 5.23239245e+03 3.90240805e+01
1.66954743e+01 6.90537789e-01 1.26928413e-03 8.60543615e-03 -2.00046306e-03 9.40571712e-
05 -5.46426779e-05]
Learning Rate: 0.03
```



I took those theta values and plugged them into the get_regression_analysis() function along with the test dataset. After 10,000 epochs I got the following output:

```
No Missing Values.
Continuing...
No Data Leaking.
Beginning...
First pass...
Lowest Cost of 367506.68258893664 was acheived after 10000 epochs.
Theta Values: [57.62370578762455, 6389.921845295166, -75.84236888542341,
20.76879861938794, -0.5311418421416911, 0.08774266112156588, -0.0031892626423641454,
0.000386481928069003, -1.5376238910610496e-05, 1.3698181857773293e-06,
-7.607252636122764, 10435.823618460496, 62.99932453782277, 23.990880932969773,
0.2874009165888314, 0.0560134765770006, 0.0009884647016332826, 0.00013269614890357325,
3.0376288144227114e-06, 3.185601992830766e-07, 88.03654000839127, 5800.302304288799,
117.6418113686387, 18.735955419009343, 0.9456846270696253, 0.10133589424056777,
0.009298942237386787, 0.0011086738362180797, 0.00015562697733235674, 2.340396100736385e-
05, 58.02531591913796, 6261.009757600728, 70.69613441006877, 22.434386490542703,
0.6882682820835058, 0.1290617250966524, 0.007562586750685833, 0.0010299697996540996,
8.883482595400651e-05, 1.0253632460303579e-05, -2513.4101548399312, 2298.2921727716625,
-153.76722067039802, -39.634637376266085, -5.403468316753028, -1.0701082834787865,
-0.13840299670694756, -0.023987071331446702, -0.003274598936433077,
-0.0005330457400176543, 2950.8971045639764, 2020.221664572912, -373.54578232830437,
-78.08410252287915, -15.770416449875626, -3.1236105984874607, -0.59511776772382,
-0.11657337766632103, -0.022596678638711593, -0.00443761109321292, -22.25927696248033,
7047.791638581727, -38.63767526867751, -4.6950173880055655, -5.465702566397845,
-1.081285784596719, -0.21570316128738296, -0.03998073930366277, -0.007281839878257876,
-0.001317271220689537, -1508.7726455604668, 4285.314848407246, -131.7219016853197,
-35.37776054663965, -8.570705041131893, -1.7064115712074488, -0.3221673919056724,
-0.06035385483896236, -0.011190256896286959, -0.0020745987301334025, 996.9037274110756,
7243.412839217817, 203.18998352583586, 48.386609909249145, 3.5924973326620306,
0.6100726586626443, 0.05978290595713121, 0.009310945673170125, 0.0009988511106682296,
0.00015089180682388443, -128.2483509406422, 5131.6756428353565, -24.530454333718318,
6.714653048322725, -0.7412202980397615, -0.19800131479834548, -0.018385069695867957,
-0.005626492355554045, -0.00039196000324643845, -0.00011929006036351775]
Learning Rate: 0.03
```



At this point, the best J(θ) was 367507 (Mean Squared Error) after 10,000 epochs. It would be beneficial to run longer tests but I was running into issues with time-outs at this point.

In the future, it would be interesting to explore how these variables are manipulated programmatically in industry. It would also be interesting and would add value to have some charts showing the line being fit to the data, although choosing which x to display might prove tricky.