Univariate Linear Regression Implementation - Diabetes

Summary

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

I made it where the hyperparameters can be adjusted for quick iterations to perform the overall performance.

Ultimately, the model performs better than expected given the simplicity and is able to get a somewhat well-fitting line with the data.

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.

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



# Dataset
from sklearn.datasets import load_diabetes
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
```

Load / Split Data

We begin with loading the dataset. Afterwards, we will assign X and y to variables, and then send X to the feature_reduction function. This function utilizes sklearn's PCA library to perform principal component analysis, reducing our 10 features into 1. Essentially, we are using all of the other variables to guess the blood glucose level (which is an indicator of diabetic status).

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

  # Reduce features
  X = feature_reduction(X)

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

  return data
```

Reducing the features is simple.

```
def feature_reduction(X):
  pca = PCA(n_components = 1)
  return pca.fit_transform(X)
```

Note: It is standard practice to standardize values prior to PCA. However, in this implementation the StandardScaler was causing a bunch of headaches but was not vital enough to spend time on, given the assignment's deadline.

Shuffle and split is my own implementation from a prior assignment. We simply shuffle prior to assigning a percentage of the set to variables train, dev, and test.

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

  total_sample = len(data)
  train = data[:int(total_sample*0.7)]
```

```
  dev = data[int(total_sample*0.7):int(total_sample*0.85)]
  test = data[int(total_sample*0.85):]
  return total_sample, train, dev, test
```

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()
```

Univariate Gradient Descent

Here we are using a simple algorithm to calculate our *linear regression*:

```
# Linear Regression Model
def univariate_linear_regression(theta, input):
```

```
  prediction = theta[0] + theta[1] * input
  return prediction
```

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

```
# Cost Function
def compute_cost(Y_pred, Y_true, length):
  m = length
  J = 1/(2*m) * np.sum((Y_pred - Y_true)**2) # Mean Squared Error
  # J = (1/2*m) * (np.abs(Y_pred - Y_true)) # Mean Absolute Error

  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
  theta[0] = theta[0] - (learning_rate * (1/m) * np.sum(Y_pred - Y_true))
  theta[1] = theta[1] - (learning_rate * (1/m) * np.sum((Y_pred - Y_true) * X))
  return theta
```

Plots

We plot our data and then make predications on the feature values and graph it in. Plots where i/5 = 0 are illustrated, and they show each iteration of gradient descent the algorithm has gone through.

```
# Plot Results
def plot_results(k, length, theta, X, Y, cost):
  plt.subplot(5, 5, k)
  plt.scatter(X, Y, color='b')


  # Make predictions on X values
  temp_input = np.array([x_val for x_val in X])
  temp_prediction = univariate_linear_regression(theta, temp_input)
  plt.plot(temp_input, temp_prediction, 'g')

  s = 'theta:[%.4f, %.4f]' %(theta[0], theta[1])
```

```
  c = 'cost:%.4f' %cost
  plt.title(s+'\n'+c)
```

Running the Program

Here we are synthesizing everything into a single function call that will perform everything we have defined thus far. We start by shuffling and splitting the data as described above. Using the dev set, I found that a theta = [35.0, 255.0] gave some of the lowest costs of any of the numbers I tried. It would obviously be better to have a program that iterates over many, many more than I did, but for the purposes of learning this is a good starting step.

We run 100 iterations, and for every 5 we plot a graph of the values (blue dots) and the prediction (green line). As you can see from the included image, the model gets better at predicting the values. Clearly, it would be impossible to fit the line very well given the linear nature. Adding more complex values into the mix, to give the line a 'squiggle' would increase our accuracy, decreasing the cost.

Our lowest cost (in terms of mean squared error) was 179609, as seen in the linked image above. While this is quite large, it should be noted the data has a fairly wide distribution and the line is univariate, relying on PCA that was performed on data that had not been standardized.

```
def get_regression_analysis(theta, data, learning_rate):
  # Shuffle and Split
  total_sample, train, dev, test = shuffle_and_split(data)

  train_x, train_y = get_features_and_labels(train)
  dev_x, dev_y = get_features_and_labels(dev)
  test_x, test_y = get_features_and_labels(test)

  plt.figure(figsize=(30, 30))
  k = 0
  length = len(test_y)

  for i in range(100):
    prediction = univariate_linear_regression(theta, test_x)
    cost = compute_cost(prediction, test_y, length)
    theta = update_theta(theta, test_x, test_y, prediction, learning_rate, length)

    if i%5 == 0:
      k += 1
      plot_results(k, length, theta, test_x, test_y, cost)
```

```
# Hyperparameters
theta = [35.0, 255.0]
learning_rate = 0.0015

# Run the Program
data = load_data()
get_regression_analysis(theta, data, learning_rate)
```