LinearRegression_WineQuality

September 19, 2019

1 Linear Regression Practice

This noteook will implement gradient descent using multivariate linear regression to predict wine quality. It will provide a full walkthrough of how I implemented the linear regression model, followed by an analysis of the performance of the model with proposed changes to make the model better.

1.1 Part 1: Implementing the model

[1]:
```
# Scientific and vector computation for python
import numpy as np

# Plotting library
from tabulate import tabulate
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D  # needed to plot 3-D surfaces

# tells matplotlib to embed plots within the notebook
%matplotlib inline
```

1.1.1 Loading the Data and perform feature scaling

First, we load the data and use feature normalization to ensure gradient descent converges much more quickly

[2]:
```
# Load data
data = np.loadtxt(open("data/wineQuality.csv", "rb"), delimiter=",", skiprows=1)

# Create feature matrix and output variables
# Here, X denotes the feature matrix and y is the output
X = data[:, :-1]
y = data[:, -1]
m = y.size

# print out some data points
sample = data[:10, :]
table = [column for column in sample]
```

```
print(tabulate(table, headers=["X[0]", "X[1]", "X[2]", "X[3]", "X[4]", "X[5]",␣
↪"X[6]", "X[7]", "X[8]", "X[9]", "X[10]", "quality (1-10)" ]))
```

```
  X[0]    X[1]    X[2]    X[3]    X[4]    X[5]    X[6]    X[7]    X[8]    X[9]
X[10]    quality (1-10)
------  ------  ------  ------  ------  ------  ------  ------  ------  ------
-------  ----------------
   7.4    0.7     0        1.9   0.076      11      34  0.9978    3.51    0.56
9.4                5
   7.8    0.88    0        2.6   0.098      25      67  0.9968    3.2     0.68
9.8                5
   7.8    0.76    0.04     2.3   0.092      15      54  0.997     3.26    0.65
9.8                5
  11.2    0.28    0.56     1.9   0.075      17      60  0.998     3.16    0.58
9.8                6
   7.4    0.7     0        1.9   0.076      11      34  0.9978    3.51    0.56
9.4                5
   7.4    0.66    0        1.8   0.075      13      40  0.9978    3.51    0.56
9.4                5
   7.9    0.6     0.06     1.6   0.069      15      59  0.9964    3.3     0.46
9.4                5
   7.3    0.65    0        1.2   0.065      15      21  0.9946    3.39    0.47
10                 7
   7.8    0.58    0.02     2     0.073       9      18  0.9968    3.36    0.57
9.5                7
   7.5    0.5     0.36     6.1   0.071      17     102  0.9978    3.35    0.8
10.5               5
```

[3]:
```
# This function returns a normalized version of the feature matrix
# Parameters: X = feature matrix
def featureNormalization(X):
    # create mu and sigma vector
    # mu[x] denotes the mean value of column x
    # sigma[x] denotes the standard deviation of column x
    X_normalized = X.copy()
    mu = np.zeros(X.shape[1])
    sigma = np.zeros(X.shape[1])

    # set the values of mu and sigma
    mu = np.mean(X, axis = 0)
    sigma = np.std(X, axis = 0)
    X_normalized = (X - mu) / sigma

    # return normalized feature matrix, mu and sigma vector
    return X_normalized, mu, sigma
```

[4]:
```
# call featureNormalization on the data
X_normalized, mu, sigma = featureNormalization(X)

print("Computed mean vector: ", mu)
print("\nComputed sigma vector: ", sigma)
```

```
Computed mean vector:  [ 8.31963727  0.52782051  0.27097561  2.5388055
0.08746654 15.87492183
 46.46779237  0.99674668  3.3111132   0.65814884 10.42298311]

Computed sigma vector:  [1.74055180e+00 1.79003704e-01 1.94740214e-01
1.40948711e+00
 4.70505826e-02 1.04568856e+01 3.28850367e+01 1.88674370e-03
 1.54338181e-01 1.69453967e-01 1.06533430e+00]
```

Finally, before we use the feature matrix to compute the cost function, we must add the intercept term

[5]:
```
# Add intercept term to X
X = np.concatenate([np.ones((m, 1)), X_normalized], axis=1)
```

1.1.2 Cost Function

Next, we must implement the cost function for our multivariate linear regression model. This function computes the average of all the results of our linear hypothesis with inputs from our feature matrix compared to the actual output of our dataset.

[6]:
```
# Cost function for multivariate linear regression
# Parameters: X = feature matrix, y = output, theta = parameter vector
# Returns: cost = the computed cost of fitting data points using theta, the␣
 ↪parameter vector
def costFunction(X, y, theta):

    # number of training examples
    n = y.shape[0]
    cost = 0
    # hypothesis
    h = X.dot(theta)

    # vectorized equation for cost function
    cost = (1/(2 * n)) * np.dot((h - y).T, (h - y))

    return cost
```

1.1.3 Gradient Descent Algorithm

Using the cost function defined above, we now implement the gradient descent algorithm to train the model to find the optimal values for our parameter vector.

[7]:
```python
# Gradient Descent algorithm
# Parameters: X = feature matrix, y = output, theta = feature vector, alpha =␣
↪learning rate, iterations = number of iterations
# Returns: theta: The learned parameter vector
#          costVector: a list containign the cost function after each iteration
def gradientDescent(X, y, theta, alpha, iterations):

    # copy the theta vector to be updated by gradient descent
    theta = theta.copy()
    m = y.shape[0]
    costVector = []

    for i in range(iterations):
        theta = theta - (alpha / m) * np.dot((np.dot(X, theta) - y), X)
        costVector.append(costFunction(X, y, theta))

    return theta, costVector
```

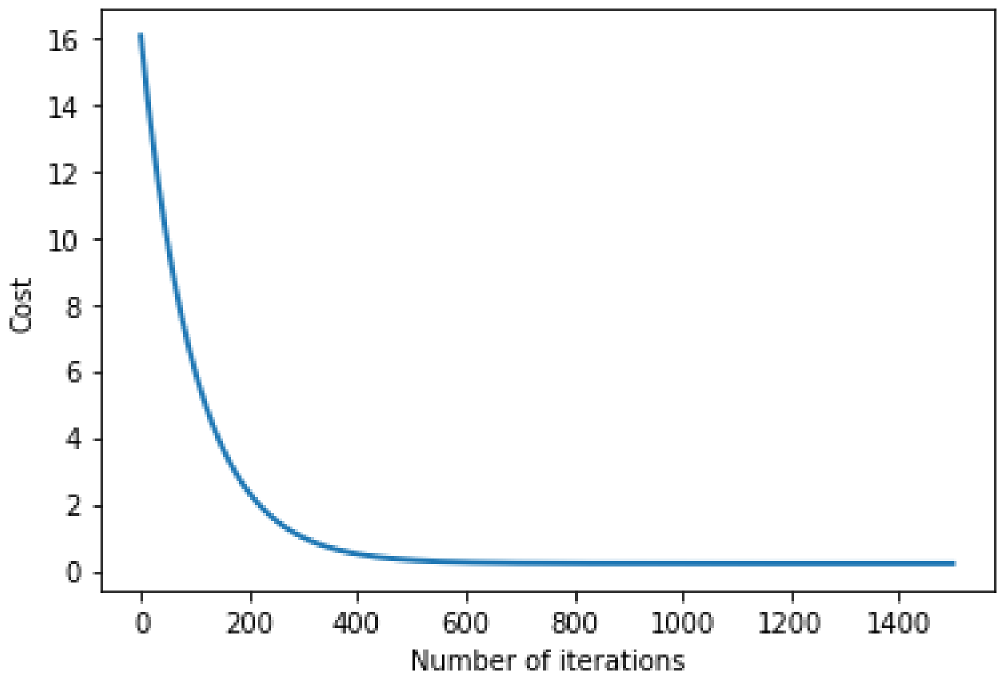
To make sure that the algorithm is implemented correctly, we can plot a graph of the learning rate over the number of iterations.

[41]:
```python
# Learning rate and number of iterations
alpha = 0.005
iterations = 1500

theta = np.zeros(X[0].shape)
theta, costVector = gradientDescent(X, y, theta, alpha, iterations)

# plot the gradient descent convergence
pyplot.plot(np.arange(len(costVector)), costVector, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost')
```

[41]:
```
Text(0, 0.5, 'Cost')
```



[46]:
```
#output the trained parameter vector
print('With alpha = {} and iterations = {}, the trained parameters are:\n\n {}' .
 ↪format(alpha, iterations, theta))

#Test with random feature values
X_test = np.array([5.7, 0.29, 0.42, 2.8, 0.140, 11, 42, 0.9942, 3.39, 0.71, 10.
 ↪3])
print("\nThe predicted wine value for the following features:\n\n"
      "fixed acidity = {}\n"
      "volatile acidity = {}\n"
      "citric acid content = {}\n"
      "residual sugar = {}\n"
      "chlorides = {}\n"
      "free sulfur dioxide = {}\n"
      "total sulfur dioxide = {}\n"
      "density = {}\n"
      "pH = {}\n"
      "sulphates = {}\n"
      "alcohol % = {}\n"
      .format(X_test[0], X_test[1], X_test[2], X_test[3], X_test[4],
              X_test[5], X_test[6], X_test[7], X_test[8], X_test[9], X_test[10]))

#add intercept term
X_test = np.concatenate(([1], X_test))
predict = np.dot(X_test1, theta)
```

```
print("wine quality =", predict)
```

```
With alpha = 0.005 and iterations = 1500, the trained parameters are:

 [ 5.63296341  0.06208009 -0.18391945 -0.01786227  0.0320261  -0.09068463
  0.04331262 -0.10730923 -0.06264955 -0.04634955  0.15953812  0.27393323]

The predicted wine value for the following features:

fixed acidity = 5.7
volatile acidity = 0.29
citric acid content = 0.42
residual sugar = 2.8
chlorides = 0.14
free sulfur dioxide = 11.0
total sulfur dioxide = 42.0
density = 0.9942
pH = 3.39
sulphates = 0.71
alcohol % = 10.3

wine quality = 5.757194394118167
```