Problem Set 1

Muhammad Hamza Ikram

April 25, 2021

Contents

In this exercise, you will implement linear regression and get to see it work on data.

All the information you need for solving this assignment is in directory.

Section	Part	Submitted Function	Points
1	Warm up exercise	warmUpExercise	10
2	Compute cost for one variable	computeCost	40
3	Gradient descent for one variable	gradientDescent	50
4	Feature normalization	featureNormalize	10
5	Compute Cost for multiple variables	computeCostMulti	20
6	Gradient descent for multiple variables	gradientDescentMulti	20
		Total Points	130

1 Simple python and numpy function

The first part of this assignment gives you practice with python and numpy syntax and the homework submission process. In the next code block, you will find the outline of a python function. Modify it to return a 5 x 5 identity matrix by filling in the following code:

A = np.eye(5)

```
def warmUpExercise(
        matrix
):
    """
    Example function in Python which computes the identity matrix.

    Returns
    -------
    matrix : array_like
        The 5x5 identity matrix.

    Instructions
    ------------
    Return the 5x5 identity matrix.
    """
    # ======== YOUR CODE HERE ======
    identity_matrix =   # modify this line
```

```
    # ==============================
    return identity
```

The previous code block only defines the function warmUpExercise. Run it by executing it in your own script. You should see output similar to the following for a 5x5 input matrix:

```
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

2 Linear regression with one variable

Now you will implement linear regression with one variable to predict profits for a food truck. Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet. The chain already has trucks in various cities and you have data for profits and populations from the cities. You would like to use this data to help you select which city to expand to next.

The file Data/ex1data1.txt contains the dataset for our linear regression problem. The first column is the population of a city (in 10,000s) and the second column is the profit of a food truck in that city (in $10,000s). A negative value for profit indicates a loss.

You are provided with the code needed to load this data. The dataset is loaded from the data file into the variables x and y:

```
import numpy as np
import os
# Read comma separated data
data = np.loadtxt(os.path.join('Data', 'ex1data1.txt'), delimiter=',')
X, y = data[:, 0], data[:, 1]

m = y.size  # number of training examples
```

2.1 Plotting the Data

Before starting on any task, it is often useful to understand the data by visualizing it. For this dataset, you can use a scatter plot to visualize the data, since it has only two properties to plot (profit and population). Many other problems that you will encounter in real life are multi-dimensional and cannot be plotted on a 2-d plot. There are many plotting libraries in python (see this blog post for a good summary of the most popular ones).

In this course, we will be mostly be using matplotlib to do all our plotting. matplotlib is one of the most popular scientific plotting libraries in python and has extensive tools and functions to make beautiful plots. pyplot is a module within matplotlib which provides a simplified interface to matplotlib's most common plotting tasks, mimicking MATLAB's plotting interface.

```
from matplotlib.pyplot as plt
def plotData(x, y):
    """
    Plots the data points x and y into a new figure. Plots the data
    points and gives the figure axes labels of population and profit.

    Parameters
    ----------
    x : array_like
        Data point values for x-axis.

    y : array_like
        Data point values for y-axis. Note x and y should have the same size.

    Instructions
    ------------
    Plot the training data into a figure using the "figure" and "plot"
    functions. Set the axes labels using the "xlabel" and "ylabel" functions.
    Assume the population and revenue data have been passed in as the x
    and y arguments of this function.

    Hint
    ----
    You can use the 'ro' option with plot to have the markers
    appear as red circles. Furthermore, you can make the markers larger by
```

```
using plot(..., 'ro', ms=10), where 'ms' refers to marker size. You
can also set the marker edge color using the 'mec' property.
"""
fig = plt.figure()  # open a new figure

# ====================== YOUR CODE HERE ======================


# ============================================================
```

2.2 Gradient Descent

In this part, you will fit the linear regression parameters θ to our dataset using gradient descent.

2.2.1 Update Equations The objective of linear regression is to minimize the cost function

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

where the hypothesis $h_\theta(x)$ is given by the linear model

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

Recall that the parameters of your model are the θ_j values. These are the values you will adjust to minimize cost $J(\theta)$. One way to do this is to use the batch gradient descent algorithm. In batch gradient descent, each iteration performs the update

$$\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} \qquad \text{simultaneously update } \theta_j \text{ for all } j$$

With each step of gradient descent, your parameters θ_j come closer to the optimal values that will achieve the lowest cost J(θ).

> **Vectors and matrices in numpy - Important implementation notes**
>
> A vector in numpy is a one dimensional array, for example np.array([1, 2, 3]) is a vector. A matrix in numpy is a two dimensional array, for example np.array([[1, 2, 3], [4, 5, 6]]). However, the following

is still considered a matrix np.array(1, 2, 3) since it has two dimensions, even if it has a shape of 1x3 (which looks like a vector).

Given the above, the function np.dot which we will use for all matrix/vector multiplication has the following properties:

It always performs inner products on vectors. If x=np.array([1, 2, 3]), then np.dot(x, x) is a scalar. For matrix-vector multiplication, so if X is a $m \times n$ matrix and y is a vector of length m, then the operation np.dot(y, X) considers y as a $1 \times m$ vector. On the other hand, if y is a vector of length n, then the operation np.dot(X, y) considers y as a $n \times 1$ vector. A vector can be promoted to a matrix using y[None] or [y[np.newaxis]. That is, if y = np.array([1, 2, 3]) is a vector of size 3, then y[None, :] is a matrix of shape 1×3. We can use y[:, None] to obtain a shape of 3×1.

```
def gradientDescent(X, y, theta, alpha, num_iters):
    """
    Performs gradient descent to learn `theta`. Updates theta by taking `num_iters`
    gradient steps with learning rate `alpha`.

    Parameters
    ----------
    X : array_like
        The input dataset of shape (m x n+1).

    y : array_like
        Value at given features. A vector of shape (m, ).

    theta : array_like
        Initial values for the linear regression parameters.
        A vector of shape (n+1, ).

    alpha : float
        The learning rate.

    num_iters : int
        The number of iterations for gradient descent.

    Returns
```

```
    -------
    theta : array_like
        The learned linear regression parameters. A vector of shape (n+1, ).

    J_history : list
        A python list for the values of the cost function after each iteration.

    Instructions
    ------------
    Peform a single gradient step on the parameter vector theta.

    While debugging, it can be useful to print out the values of
    the cost function (computeCost) and gradient here.
    """
    # Initialize some useful values
    m = y.shape[0]  # number of training examples

    # make a copy of theta, to avoid changing the original array, since numpy arrays
    # are passed by reference to functions
    theta = theta.copy()

    J_history = [] # Use a python list to save cost in every iteration

    for i in range(num_iters):
        # ==================== YOUR CODE HERE ================================



        # ====================================================================

        # save the cost J in every iteration
        J_history.append(computeCost(X, y, theta))

    return theta, J_history
```

After you are finished call the implemented gradientDescent function and print the computed θ. We initialize the θ parameters to 0 and the learning rate α to 0.01. Execute the following cell to check your code.

```
# initialize fitting parameters
theta = np.zeros(2)

# some gradient descent settings
iterations = 1500
alpha = 0.01

theta, J_history = gradientDescent(X ,y, theta, alpha, iterations)
print('Theta found by gradient descent: {:.4f}, {:.4f}'.format(*theta))
print('Expected theta values (approximately): [-3.6303, 1.1664]')
```

We will use your final parameters to plot the linear fit.

```
# plot the linear fit
plotData(X[:, 1], y)
pyplot.plot(X[:, 1], np.dot(X, theta), '-')
pyplot.legend(['Training data', 'Linear regression']);
```

2.3 Linear regression with multiple variables

In this part, you will implement linear regression with multiple variables to predict the prices of houses. Suppose you are selling your house and you want to know what a good market price would be. One way to do this is to first collect information on recent houses sold and make a model of housing prices.

The file Data/ex1data2.txt contains a training set of housing prices in Portland, Oregon. The first column is the size of the house (in square feet), the second column is the number of bedrooms, and the third column is the price of the house.

2.3.1 Feature Normalization

We start by loading and displaying some values from this dataset. By looking at the values, note that house sizes are about 1000 times the number of bedrooms. When features differ by orders of magnitude, first performing feature scaling can make gradient descent converge much more quickly.

```
# Load data
data = np.loadtxt(os.path.join('Data', 'ex1data2.txt'), delimiter=',')
```

```
X = data[:, :2]
y = data[:, 2]
m = y.size

# print out some data points
print('{:>8s}{:>8s}{:>10s}'.format('X[:,0]', 'X[:, 1]', 'y'))
print('-'*26)
for i in range(10):
    print('{:8.0f}{:8.0f}{:10.0f}'.format(X[i, 0], X[i, 1], y[i]))
```

Your task here is to complete the code in featureNormalize function:

Subtract the mean value of each feature from the dataset. After subtracting the mean, additionally scale (divide) the feature values by their respective "standard deviations." The standard deviation is a way of measuring how much variation there is in the range of values of a particular feature (most data points will lie within ± 2 standard deviations of the mean); this is an alternative to taking the range of values (max-min). In numpy, you can use the std function to compute the standard deviation.

For example, the quantity X[:, 0] contains all the values of x_1 (house sizes) in the training set, so np.std(X[:, 0]) computes the standard deviation of the house sizes. At the time that the function featureNormalize is called, the extra column of 1's corresponding to $x_0 = 1$ has not yet been added to X.

You will do this for all the features and your code should work with datasets of all sizes (any number of features / examples). Note that each column of the matrix X corresponds to one feature.

```
def  featureNormalize(X):
    """
    Normalizes the features in X. returns a normalized version of X where
    the mean value of each feature is 0 and the standard deviation
    is 1. This is often a good preprocessing step to do when working with
    learning algorithms.

    Parameters
    ----------
    X : array_like
        The dataset of shape (m x n).

    Returns
```

```
    -------
    X_norm : array_like
        The normalized dataset of shape (m x n).

    Instructions
    ------------
    First, for each feature dimension, compute the mean of the feature
    and subtract it from the dataset, storing the mean value in mu.
    Next, compute the  standard deviation of each feature and divide
    each feature by it's standard deviation, storing the standard deviation
    in sigma.

    Note that X is a matrix where each column is a feature and each row is
    an example. You needto perform the normalization separately for each feature.

    Hint
    ----
    You might find the 'np.mean' and 'np.std' functions useful.
    """
    # You need to set these values correctly
    X_norm = X.copy()
    mu = np.zeros(X.shape[1])
    sigma = np.zeros(X.shape[1])

    # ============================ YOUR CODE HERE =====================


    # ================================================================
    return X_norm, mu, sigma
```

Execute the next cell to run the implemented featureNormalize function.

In [ ]:

```
# call featureNormalize on the loaded data
X_norm, mu, sigma = featureNormalize(X)

print('Computed mean:', mu)
print('Computed standard deviation:', sigma)
```

After the featureNormalize function is tested, we now add the intercept term to X_{norm}:

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

3.2 Gradient Descent

Previously, you implemented gradient descent on a univariate regression problem. The only difference now is that there is one more feature in the matrix X. The hypothesis function and the batch gradient descent update rule remain unchanged.

You should now creat functions computeCostMulti and gradientDescentMulti to implement the cost function and gradient descent for linear regression with multiple variables. If your code in the previous part (single variable) already supports multiple variables you can move on to testing it. Make sure your code supports any number of features and is well-vectorized. You can use the shape property of numpy arrays to find out how many features are present in the dataset.

> **Implementation Note:** In the multivariate case, the cost function can also be written in the following vectorized form:
>
> $$J(\theta) = \frac{1}{2m}(X\theta - \vec{y})^T(X\theta - \vec{y})$$
>
> where
>
> $$X = \begin{pmatrix} -(x^{(1)})^T- \\ -(x^{(2)})^T- \\ \vdots \\ -(x^{(m)})^T- \end{pmatrix} \qquad \mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$
>
> the vectorized version is efficient when you are working with numerical computing tools like 'numpy'. If you are an expert with matrix operations, you can prove to yourself that the two forms are equivalent.

```
"""
Instructions
------------
```

```
The following starter code runs
gradient descent with a particular learning rate (alpha).

Your task is to first make sure that your functions - ‘computeCost‘
and ‘gradientDescent‘ already work with  this starter code and
support multiple variables.

After that, try running gradient descent with different values of
alpha and see which one gives you the best result.


Hint
----
At prediction, make sure you do the same feature normalization.
"""
# Choose some alpha value - change this
alpha = 0.1
num_iters = 400

# init theta and run gradient descent
theta = np.zeros(3)
theta, J_history = gradientDescentMulti(X, y, theta, alpha, num_iters)

# Plot the convergence graph
plt.plot(np.arange(len(J_history)), J_history, lw=2)
plt.xlabel('Number of iterations')
plt.ylabel('Cost J')
```