Machine Learning Programming Exercise 1: Linear Regression

adapted to Python language from Coursera/Andrew Ng

September 26, 2023

1 Introduction

In this exercise, you will implement linear regression and get to see it work on data. Before starting on this programming exercise, we strongly recommend watching the video lectures and completing the review questions for the associated topics. To get started with the exercise, you will need to download the starter code and unzip its contents to the directory where you wish to complete the exercise.

2 Files included in this exercise

- **ex1.py** - Python script that will help step you through the exercise
- **ex1_multi.py** - Python script for the later parts of the exercise
- **ex1data1.txt** - Dataset for linear regression with one variable
- **ex1data2.txt** - Dataset for linear regression with multiple features

⋆ **warmUpExercise.py** - Simple example function in Python

⋆ **plotData.py** - Function to display the dataset

⋆ **computeCost.py** - Function to compute the cost of linear regression

⋆ **gradientDescent.py** - Function to run gradient descent

⋆ **computeCostMulti.py** - Cost function for multiple features

⋆ **gradientDescentMulti.py** - Gradient descent for multiple features

⋆ **featureNormalize.py** - Function to normalize features

⋆ **normalEqn.py** - Function to compute the normal equations

[⋆] indicates files you will need to complete.

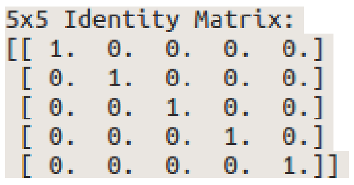
Throughout the exercise, you will be using the scripts **ex1.py** and **ex1_multi.py**. These scripts set up the dataset for the problems and make calls to functions that you will write. You do not need to modify either of them. You are only required to modify functions in other files, by following the instructions in this assignment.

3 Simple python function

The first part of **ex1.py** gives you practice with Python syntax. In the file **warmUpExercise.py**, you will find the outline of an Python function. Modify it to return a 5x5 identity matrix by filling in the following code:

```
A = eye(5)
```

When you are finished, run **ex1.py** and you should see output similar to the following:



```
5x5 Identity Matrix:
[[ 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.]]
```

4 Linear regression with one variable

In this part of this exercise, 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 **ex1data1.txt** contains the dataset for our linear regression problem. The first column is the population of a city and the second column is the profit of a food truck in that city. A negative value for profit indicates a loss. The **ex1.py** script has already been set up to load this data for you.

4.1 Plotting the Data

Before starting on any task, it is always 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 can't be plotted on a 2-d plot.)

In **ex1.py**, the dataset is loaded from the data file into the features X and y:

```
path = 'ex1data1.txt'
data = pd.read_csv(path,
header=None, names=['Population', 'Profit'])

nbCol = data.shape[1]
X = data.iloc[:,0:nbCol-1]
y = data.iloc[:,nbCol-1:nbCol]

X = np.array(X.values)
y = np.array(y.values)

m = length(y); % number of training examples
```

Next, the script calls the **plotData** function to create a scatter plot of the data. Your job is to complete **plotData.py** to draw the plot; modify the file and fill in the following code:

```
fig = plt.figure()  # open a new figure window
plt.plot(X, y, 'ro', markersize=10)
plt.grid(True) #Always plot.grid true!
plt.ylabel('Profit in $10,000s')
plt.xlabel('Population of City in 10,000s')
plt.show()
```

Now, when you continue to run **ex1.py**, our end result should look like Figure 1, with the same red 'x' markers and axis labels. To learn more about the plot command, you can search on the matplotlib python website.

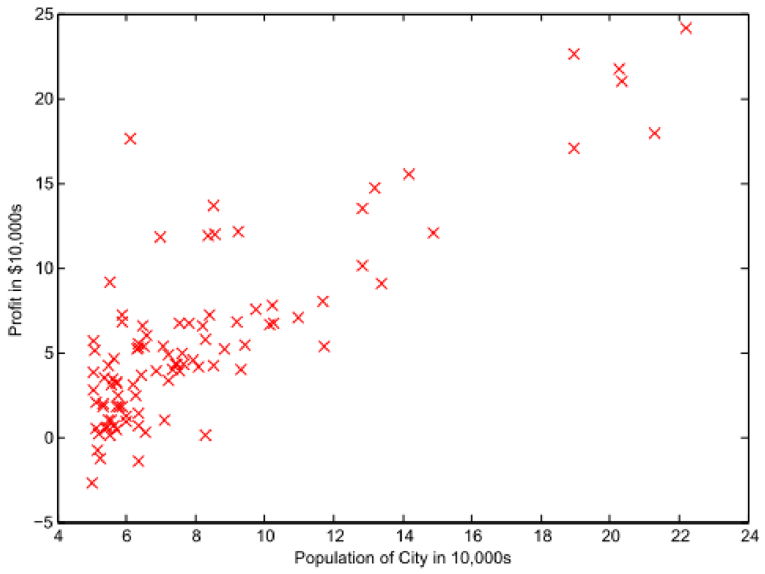


Figure 1 – Scatter plot of training data

4.2 Gradient Descent

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

4.2.1 Update Equations

The objective of linear regression is to minimize the cost function

$$J(\theta) = \frac{1}{2m} \sum_{i=0}^{m-1} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

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

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

where $\theta = (\theta_0, \theta_1)^T \in \mathbb{R}^{2x1}$ and $x = (1, x_1)^T \in \mathbb{R}^{2x1}$.

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

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=0}^{m-1} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} \text{(simultaneously update } \theta_j \text{ for all } j)$$

4.2.2 Implementation

In **ex1.py**, we have already set up the data for linear regression. In the following lines, we add another dimension to our data to accommodate the θ_0 intercept term. We also initialize the initial parameters to 0 and the learning rate alpha to 0.01.

```
# Add intercept term to X
X = np.column_stack((np.ones((m, 1)), X)) works fine too

# Initialize theta
theta = np.array([[0.,0.]]).T

# compute Descent gradient
# initialize features for learning rate and iterations
alpha = 0.01
iters = 1500
```

Hints Here are some things to keep in mind before you start implementing :

- Python array indices start from zero. If you're storing θ_0 and θ_1 in a vector called theta, the values will be `theta[0]` and `theta[1]` .
- If you are seeing many errors at runtime, inspect your matrix operations to make sure that you're adding and/or multiplying matrices of compatible dimensions. Printing the dimensions of variable n with the `n.shape` command will help you debug.
- For example, `A*B` does an element-wise array multiplication, while `A@B` or `np.dot(A,B)` or `A.dot(B)` does a matrix multiplication.

4.2.3 Computing the cost $J(\theta)$

As you perform gradient descent to learn minimizing the cost function $J(\theta)$, it is helpful to monitor the convergence by computing the cost. In this section, you will implement a function to calculate $J(\theta)$ so you can check the convergence of your gradient descent implementation. Your next task is to complete the code in the file **computeCost.py**, which is a function that computes $J(\theta)$. As you are doing this, remember that the features X and y are not scalar values, but matrices whose rows represent the examples from the training set. Once you have completed the function, the next step in **ex1.py** will run **computeCost** once using θ initialized to zeros, and you will see the cost printed to the screen. You should expect to see a cost of 32.07.

4.2.4 Gradient descent

Next, you will implement gradient descent in the file **gradientDescent.py**. The loop structure has been written for you, and you only need to supply the updates to θ within each iteration.

As you program, make sure you understand what you are trying to optimize and what is being updated. Keep in mind that the cost $J(\theta)$ is parameterized by the vector θ, not X and y. That is, we minimize the value of $J(\theta)$ by changing the values of the vector θ, not by changing X or y. Refer to the equations in this handout and to the video lectures if you are uncertain.

A good way to verify that gradient descent is working correctly is to look at the value of $J(\theta)$ and check that it is decreasing with each step. The starter code for **gradientDescent.py** calls **computeCost** on every iteration and prints the cost.

Assuming you have implemented **gradientDescent** and **computeCost** correctly, your value of $J(\theta)$ should never increase, and should converge to a steady value by the end of the algorithm.

After you are finished, **ex1.py** will use your final parameters to plot the linear fit. The result should look something like Figure 2:

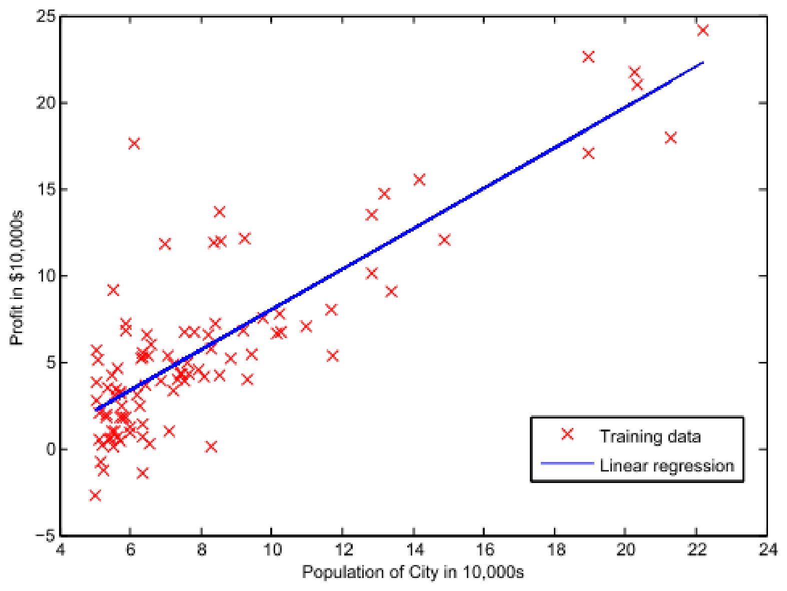


Figure 2 – Training data with linear regression fit.

Your final values for θ will also be used to make predictions on profits in areas of 35,000 and 70,000 people. Note the way that the following lines in **ex1.py** uses matrix multiplication, rather than explicit summation or looping, to calculate the predictions. This is an example of code vectorization in Python.

```
predict1 = np.array([[1, 3.5]]).dot(theta) # np.array([[1, 3.5]])@theta
predict2 = np.array([[1, 7]]).dot(theta) # np.array([[1, 7]])@theta
```

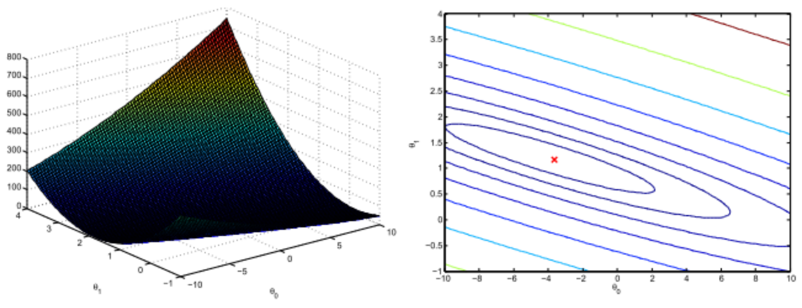
4.3 Visualizing $J(\theta)$

To understand the cost function $J(\theta)$ better, you will now plot the cost over a 2-dimensional grid of θ_0 and θ_1 values. You will not need to code anything new for this part, but you should understand how the code you have written already is creating these images. In the next step of **ex1.py**, there is code set up to calculate $J(\theta)$ over a grid of values using the computeCost function that you wrote.

```
# Create grid coordinates for plotting
theta0 = np.linspace(-10, 10, 100)
theta1 = np.linspace(-1, 4, 100)
theta0, theta1 = np.meshgrid(theta0, theta1, indexing='xy')


# Calculate Z-values (Cost) based on grid of coefficients
Z = np.zeros((theta0.shape[0],theta1.shape[0]))
for (i,j),v in np.ndenumerate(Z):
        t = np.array([[theta0[i,j], theta1[i,j]]]).T
        Z[i,j] = computeCost(X,y, t)
```

After these lines are executed, you will have a 2-D array of $J(\theta)$ values. The script **ex1.py** will then use these values to produce surface and contour plots of $J(\theta)$ using the **plot_surface** and **contour** commands. The plots should look something like Figure 3:



(a) Surface (b) Contour, showing minimum

Figure 3 – Cost function $J(\theta)$.

The purpose of these graphs is to show you that how $J(\theta)$ varies with changes in θ_0 and θ_1. The cost function $J(\theta)$ is bowl-shaped and has a global mininum. (This is easier to see in the contour plot than in the 3D surface plot). This minimum is the optimal point for θ_0 and θ_1 , and each step of gradient descent moves closer to this point.

5 Linear regression with multiple features

In this part, you will implement linear regression with multiple features 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 **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.

The **ex1_multi.py** script has been set up to help you step through this exercise.

5.1 Feature Normalization

The **ex1_multi.py** script will 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.

Your task here is to complete the code in **featureNormalize.py** to

- subtract the mean value of each feature from the dataset.
- After subtracting the mean, additionally scale (multiply) the feature values by the inverse of their respective standard deviations (or divide by stds).

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 Python, you can use the `np.std` function to compute the standard deviation. For example, inside **featureNormalize.py**, 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 **featureNormalize.py** is called, the extra column of 1's corresponding to $x_0 = 1$ has not yet been added to X (see **ex1_multi**.py for details).

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.

Important implementation Note : When normalizing the features, it is important to store the values used for normalization - the mean value and the standard deviation - used for the computations. After learning the parameters from the model, we often want to predict the prices of houses we have not seen before. Given a new x value (living room area and number of bedrooms), we must first normalize x using the mean and standard deviation that we had previously computed from the training set.

5.2 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 complete the code in **computeCostMulti.py** and **gradientDescentMulti.py** to implement the cost function and gradient descent for linear regression with multiple features. If your code in the previous part (single variable) already supports multiple features, you can use it here too.

Make sure your code supports any number of features and is well-vectorized. You can use `n = X.shape[1]` 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:

$$\begin{aligned} J(\theta) &= \frac{1}{2m} (X\theta - y)^T (X\theta - y) \\ &= \frac{1}{2m} \sum_{i=0}^{m-1} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 \\ &= \frac{1}{2m} \sum_{i=0}^{m-1} \left(\left(x^{(i)} \right)^T \theta - y^{(i)} \right)^2 \end{aligned}$$

$$X = \begin{pmatrix} \left(x^{(1)}\right)^T \\ \left(x^{(2)}\right)^T \\ \vdots \\ \left(x^{(m)}\right)^T \end{pmatrix} \in \mathbb{R}^{mxn},\ y = \begin{pmatrix} \left(y^{(1)}\right) \\ \left(y^{(2)}\right) \\ \vdots \\ \left(y^{(m)}\right) \end{pmatrix} \in \mathbb{R}^{mx1},\ \theta = (\theta_0, \theta_1, \ldots, \theta_n)^T \in \mathbb{R}^{nx1}$$

where $x^{(i)}_{i=1,\ldots,m}$ is a n-dimensional vector.

The vectorized version is efficient when you're working with numerical computing tools like Python. If you are an expert with matrix operations, you can prove to yourself that the two forms are equivalent.

5.2.1 Selecting learning rates

In this part of the exercise, you will get to try out different learning rates for the dataset and find a learning rate that converges quickly. You can change the learning rate by modifying **ex1_multi.py** and changing the part of the code that sets the learning rate.

The next phase in **ex1_multi.py** will call your **gradientDescent.py** function and run gradient descent for about 50 iterations at the chosen learning rate. The function should also return the history of $J(\theta)$ values in a vector J. After the last iteration, the **ex1_multi.py** script plots the J values against the number of the iterations.

If you picked a learning rate within a good range, your plot look similar Figure 4. If your graph looks very different, especially if your value of $J(\theta)$ increases or even blows up, adjust your learning rate and try again. We recommend trying values of the learning rate α on a log-scale, at multiplicative steps of about 3 times the previous value (i.e., 0.3, 0.1, 0.03, 0.01 and so on). You may also want to adjust the number of iterations you are running if that will help you see the overall trend in the curve.

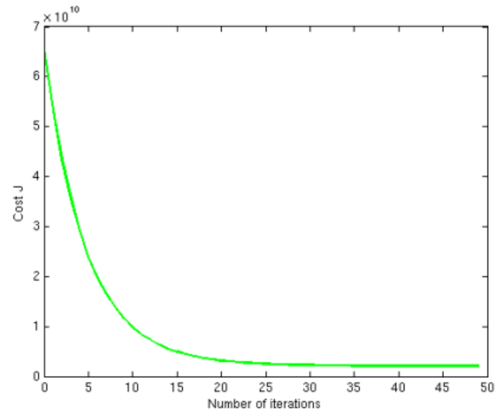


Figure 4 – Convergence of gradient descent with an appropriate learning rate.

Implementation Note: If your learning rate is too large, $J(\theta)$ can diverge and "blow up", resulting in values which are too large for computer calculations. In these situations, Python will tend to return NaNs. NaN stands for "not a number" and is often caused by undefined operations that involve $-\infty$ and $+\infty$.

Python Tip: To compare how different learning learning rates affect convergence, it's helpful to plot J for several learning rates on the same figure. Concretely, just try three different values of alpha (you should probably try more values than this), store the costs in $J1$, $J2$ and $J3$ and plot and analyse them.

Notice the changes in the convergence curves as the learning rate changes. With a small learning rate, you should find that gradient descent takes a very long time to converge to the optimal value. Conversely, with a large learning rate, gradient descent might not converge or might even diverge!

Using the best learning rate that you found, run the **ex1_multi.py** script to run gradient descent until convergence to find the final values of θ. Next, use this value of θ to predict the price of a house with 1650 square feet and 3 bedrooms. You will use value later to check your implementation of the normal equations. Don't forget to normalize your features when you make this prediction!

5.3 Optional exercise: Normal Equations

In the lecture videos, you learned that the closed-form solution to linear regression is

$$\theta = \left(X^T X\right)^{-1} X^T y$$

Using this formula does not require any feature scaling, and you will get an exact solution in one calculation: there is no "loop until convergence" like in gradient descent.

Complete the code in **normalEqn.py** to use the formula above to calculate θ. Remember that while you don't need to scale your features, we still need to add a columns of 1's to the X matrix to have an intercept term (θ_0).

Optional exercise: Now, once you have found θ using this method, use it to make a price prediction for a 1650-square-foot house with 3 bedrooms. You should find that gives the same predicted price as the value you obtained using the model fit with gradient descent (in Section 5.2.1).

5.4 Some questions, you have to answer...

Le codage n'est qu'un prétexte pour comprendre les différents concepts du machine learning. **Ici aucune équation...Exprimez avec vos mots les concepts pour les comprendre.**

- Définissez les termes:
  - Approches supervisées
  - Approches non-supervisées
  - Régression

  - Classification

- Représenter en un schéma général, les processus d'apprentissage et de prédiction ?
- Comment fonctionne l'apprentissage? Par quels moyens? A quoi sert la fonction de coût? Comment est-résolu le problème? Connaissez vous d'autres moyens de le résoudre?
- Pourquoi faut-il parfois normaliser les descripteurs (features)?