INF283 | Weekly Exercise 02 | Regression

Deadline

Sep 07, 2018 | 23:59

What to deliver

You can try out your Python code within this notebook. But you should make a PDF file of answers for each of the tasks, and then submit this PDF file on Mitt UiB.

Where to deliver

On Mitt UiB (https://mitt.uib.no/courses/12791/assignments) in the assignments section.

Note to students

This is a Python notebook and all the examples are in Python. You are free to use any programming language you want to do the exercise questions in; we will grade your submissions. But, if you got stuck somewhere, we can provide help for Python only. Moreover, the solution provided after the deadline will also in be in Python.

In case you want to learn Python, you can quickly learn it using these set of short videos (https://www.youtube.com/watch?v=oVp1vrfL_w4&list=PLQVvvaa0QuDe8XSftW-RAxdo6OmaeL85M).

Table of Contents:

1. Univariate Linear Regression

In univariate regression, the response variable is modeled in terms of just one predictor variable. In this section, we will experiment with various ways in which we can perform linear regression.

Linear Regression with Gradient Descent

In the lecture, you saw the closed-form solution

$$\hat{w} = (X^T X)^{-1} X^T y$$

to find the best values of linear regression parameters. This closed-form solution exists only when the matrix X is invertible. Even if the matrix X is invertible, but is very large in its dimensions, it might not be possible to invert it with the compute resources that you might have at your disposal. In these scenarios, we can use gradient descent to find the best value of parameter estimates.

Gradient descent attempts to find the *best* values for these parameters so that the value of some error function is minimized. We will be using Mean Squared Error (MSE) as our error function, which is the mean of the sum of squared error (SSE). Error functions are also called the loss functions.

This code demonstrates how a gradient descent search may be used to solve the linear regression problem of fitting a line through a set of points. The goal is to model a set of points with a straight line. A straight line is defined by two parameters: the line's slope w_0 , and its y-intercept w_1 (some texts may denote w_0 and w_1 as b and m, respectively, so don't get confused by the notation).

w_0 and w_1 can take on any value, but only certain specific values of w_0 and w_1 will yield a line that minimizes the sum of squares error between the original points and the line that tries to model these points (refer to this interactive demo (http://www.dangoldstein.com/regression.html) to get a better understanding of this concept).

In [32]:

```
# install numpy first. If you have it installed already then pip won't install it
# and you will get requirement already satisfied message. That's normal.
!pip install numpy
```

```
Requirement already satisfied: numpy in c:\users\phili\appdata\roaming\pytho
n\python37\site-packages (1.15.1)
```

In [33]:

```
# To use numpy, we must first import it
import numpy as np
```

Finding the mean squared error (MSE)

Now we define a function called `compute_error_for_line_given_points` to find the mean squared error between a line (modeled by parameters `w0` and `w1`) and a set of points to which it tries to model. Mathematically, it is given as:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} [y_i - (w_0 x_i + w_1)]^2$$

where `n` is the number of data points.

In [34]:

```
# We are now going to define a function which takes the parameters (b and m) of
# a line and then finds the mean-squared error between the user-specified points
# and the line.
def compute_error_for_line_given_points(w0, w1, points):
    totalError = 0
    for i in range(0, len(points)):
        x = points[i, 0]
        y = points[i, 1]
        # accumulate 'sum of square' errors in totalError variable
        totalError += (y - (w1 * x + w0)) ** 2
    # find mean of sum of squared errors
    mse = totalError/len(points)
    return mse

# N.B.: Students who wish to do this excercise in R should implement this function in R the
```

Gradient Descent

In gradient descent, we start with some initial values of w_0 and w_1 and then refine these crude estimate until we can no longer see an appreciable decrease in the mean squared error. To refine the w_0 and w_1 estimates, we need to update them (make their values higher or lower) so that the mean squared error gets reduced. Taking partial derivative of MSE with respect to w_0 and w_1, can tells us whether to increase or decrease the values of these parameter to get an improved fit.

If we take the partial derivative of the MSE function with respect of w_0, we get:

$$\frac{\partial MSE}{\partial w_0} = \frac{\partial}{\partial w_0}\left(\frac{1}{n}\sum_{i=1}^{n}[y_i - (w_1 x_i + w_0)]^2\right) = -\frac{2}{n}\left(\sum_{i=1}^{n} y_i - (w_1 x_i + w_0)\right)$$

If we take the partial derivative of the MSE function with respect of w_1, we get:

$$\frac{\partial MSE}{\partial w_1} = \frac{\partial}{\partial w_1}\left(\frac{1}{n}\sum_{i=1}^{n}[y_i - (w_1 x_i + w_0)]^2\right) = -\frac{2}{n}\left(\sum_{i=1}^{n} x_i[y_i - (w_1 x_i + w_0)]\right)$$

The gradient ∇ of MSE is just as a vector of these partial derivatives of MSE with respect to w_0 and w_0:

$$\nabla MSE(w_0, w_1) = \left(\frac{\partial MSE}{\partial w_0}, \frac{\partial MSE}{\partial w_1}\right)$$

We can now define the update rule for w_0 and w_1. Update rule modify the current value of the parameter such that the updated parameter values cause a decrease in the MSE.

Update rule for the weight vector of w_0 and w_1:

$$(w_0, w_1)_{t+1} = (w_0, w_1)_t - \eta\left(\frac{\partial MSE}{\partial w_0}, \frac{\partial MSE}{\partial w_1}\right)$$

where η is the learning rate.

The `step_gradient` function below shows you how to implement the gradient descent.

In [35]:

```
def step_gradient(w0_current, w1_current, points, learningRate):
    #initialize the partial derivatives for the cummlative sum
    w0_par_der = 0
    w1_par_der = 0

    n = len(points)

    # computation for the summation
    for i in range(0, len(points)):
        x = points[i, 0]
        y = points[i, 1]
        # partial derivative (of MSE) with respect to w0
        w0_par_der += (y - ((w1_current * x) + w0_current))
        # partial derivative (of MSE) with respect to w1
        w1_par_der += x * (y - ((w1_current * x) + w0_current))

    # multiplcation of summation results with -2/n
    w0_par_der = -(2/n) * w0_par_der
        # partial derivative (of MSE) with respect to w1
    w1_par_der = -(2/n) * w1_par_der

    # make a gradient vector from the partial derivatives
    gradient_mse = np.array([w0_par_der, w1_par_der])

    # make a vector of weights
    weight_vector = np.array([w0_current, w1_current])

    # update rule for weights
    updated_weight_vector = weight_vector - (learningRate * gradient_mse)

    # return the updated weight vector as a list
    return np.ndarray.tolist(updated_weight_vector)

# N.B. Students who wish to do this excercise in R should implement this function in R them
```

Running Gradient Descent Iteratively

Gradient descent is an iterative method of improving the parameter estimates. Therefore, we run `step_gradient` function until we reach the absolute minimum of error, or until we have executed the `gradient_step` function a certain number of times, or if the error is small. Here we implement a function called `gradient_descent_runner` that runs the gradient_step function for `num_iterations` times. In the exercise later on, we will ask you to make some changes in this function.

In [36]:

```
def gradient_descent_runner(points, starting_w0, starting_w1, learning_rate, num_iterations
    w0 = starting_w0
    w1 = starting_w1
    for i in range(num_iterations):
        w0, w1 = step_gradient(w0, w1, points, learning_rate)
        mse = compute_error_for_line_given_points(w0, w1, points)
        print(f'Iteration {i+1}: w0={w0:0.5f}, w1={w1:0.5f}, mse={mse:0.5f}')
    return [w0, w1, mse]
# N.B.: Students who wish to do this exercise in R should implement this function in R them
```

Bringing it all together

All the functions we have define above won't do anything unless we call them. But we first need the data. We will generate the data points from a straight line and then add noise to it:

$$y = 4 + 3x + noise$$

The y-intercept (w_0) of the line of the line is 4, and the slope (w_1) of the line is 3.

We will then use gradient descent to estimate the parameters of the line that generated this noisy linear data. If everything works correctly, then the parameters estimates should be close to the actual values of w_0 and w_1 used to generate the original data.

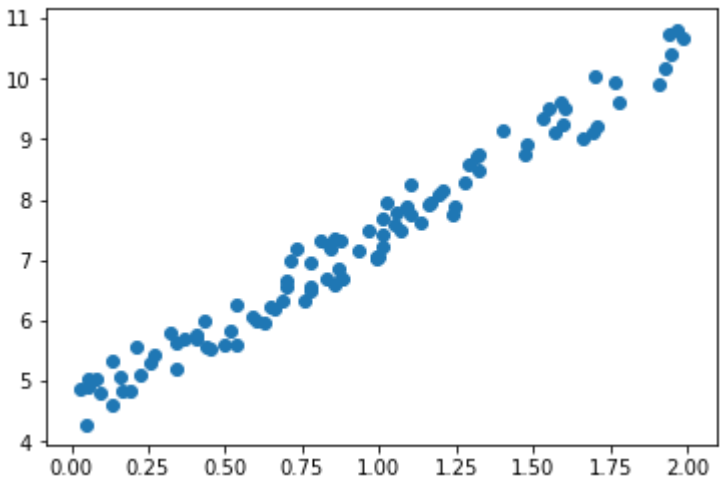
The first column is the x values, and the second column contains the y-values.

In [37]:

```
np.random.seed(2)

# generate 100 x values from 0 to 2 randomly, then sort them in ascending order
X = 2 * np.random.rand(100, 1)
X.sort(axis=0)

# generate y values and add noise to it
y = 4 + 3 * X + np.random.rand(100, 1)

# let us plot the data
import matplotlib.pyplot as plt
%matplotlib inline
plt.scatter(X, y)
```

Out[37]:

```
<matplotlib.collections.PathCollection at 0x1421cbf0>
```



In [38]:

```
# combine the x and y values into a single array called points
points = np.column_stack((X, y))

num_iterations = 100
learning_rate = 0.001
initial_w0 = 0 # initial y-intercept guess
initial_w1 = 0 # initial slope guess
[w0, w1, mse] = gradient_descent_runner(points, initial_w0, initial_w1, learning_rate, num_
```

```
Iteration 1: w0=0.01447, w1=0.01502, mse=54.56336
Iteration 2: w0=0.02888, w1=0.02998, mse=54.13273
Iteration 3: w0=0.04324, w1=0.04488, mse=53.70551
Iteration 4: w0=0.05754, w1=0.05972, mse=53.28169
Iteration 5: w0=0.07178, w1=0.07449, mse=52.86122
Iteration 6: w0=0.08597, w1=0.08921, mse=52.44409
Iteration 7: w0=0.10010, w1=0.10387, mse=52.03026
Iteration 8: w0=0.11418, w1=0.11847, mse=51.61972
Iteration 9: w0=0.12820, w1=0.13301, mse=51.21243
Iteration 10: w0=0.14217, w1=0.14750, mse=50.80837
Iteration 11: w0=0.15609, w1=0.16192, mse=50.40752
Iteration 12: w0=0.16994, w1=0.17629, mse=50.00984
Iteration 13: w0=0.18375, w1=0.19060, mse=49.61532
Iteration 14: w0=0.19750, w1=0.20485, mse=49.22392
Iteration 15: w0=0.21120, w1=0.21904, mse=48.83563
Iteration 16: w0=0.22484, w1=0.23318, mse=48.45042
Iteration 17: w0=0.23843, w1=0.24726, mse=48.06826
Iteration 18: w0=0.25197, w1=0.26128, mse=47.68913
Iteration 19: w0=0.26545, w1=0.27525, mse=47.31301
Iteration 20: w0=0.27888, w1=0.28915, mse=46.93987
Iteration 21: w0=0.29226, w1=0.30301, mse=46.56969
Iteration 22: w0=0.30559, w1=0.31681, mse=46.20245
Iteration 23: w0=0.31887, w1=0.33055, mse=45.83811
Iteration 24: w0=0.33209, w1=0.34423, mse=45.47667
Iteration 25: w0=0.34526, w1=0.35786, mse=45.11809
Iteration 26: w0=0.35838, w1=0.37144, mse=44.76235
Iteration 27: w0=0.37145, w1=0.38496, mse=44.40944
Iteration 28: w0=0.38447, w1=0.39843, mse=44.05932
Iteration 29: w0=0.39743, w1=0.41184, mse=43.71198
Iteration 30: w0=0.41035, w1=0.42520, mse=43.36740
Iteration 31: w0=0.42322, w1=0.43850, mse=43.02554
Iteration 32: w0=0.43603, w1=0.45175, mse=42.68640
Iteration 33: w0=0.44880, w1=0.46495, mse=42.34994
Iteration 34: w0=0.46151, w1=0.47809, mse=42.01616
Iteration 35: w0=0.47418, w1=0.49118, mse=41.68502
Iteration 36: w0=0.48680, w1=0.50422, mse=41.35650
Iteration 37: w0=0.49936, w1=0.51721, mse=41.03059
Iteration 38: w0=0.51188, w1=0.53014, mse=40.70727
Iteration 39: w0=0.52435, w1=0.54302, mse=40.38651
Iteration 40: w0=0.53677, w1=0.55585, mse=40.06829
Iteration 41: w0=0.54915, w1=0.56862, mse=39.75259
Iteration 42: w0=0.56147, w1=0.58135, mse=39.43940
Iteration 43: w0=0.57375, w1=0.59402, mse=39.12869
Iteration 44: w0=0.58598, w1=0.60665, mse=38.82044
Iteration 45: w0=0.59816, w1=0.61922, mse=38.51464
Iteration 46: w0=0.61029, w1=0.63174, mse=38.21127
Iteration 47: w0=0.62238, w1=0.64421, mse=37.91029
```

```
Iteration 48: w0=0.63442, w1=0.65663, mse=37.61171
Iteration 49: w0=0.64641, w1=0.66900, mse=37.31549
Iteration 50: w0=0.65835, w1=0.68132, mse=37.02162
Iteration 51: w0=0.67025, w1=0.69359, mse=36.73008
Iteration 52: w0=0.68210, w1=0.70580, mse=36.44085
Iteration 53: w0=0.69391, w1=0.71798, mse=36.15392
Iteration 54: w0=0.70567, w1=0.73010, mse=35.86926
Iteration 55: w0=0.71738, w1=0.74217, mse=35.58686
Iteration 56: w0=0.72905, w1=0.75419, mse=35.30669
Iteration 57: w0=0.74068, w1=0.76617, mse=35.02875
Iteration 58: w0=0.75225, w1=0.77809, mse=34.75301
Iteration 59: w0=0.76379, w1=0.78997, mse=34.47946
Iteration 60: w0=0.77527, w1=0.80180, mse=34.20807
Iteration 61: w0=0.78672, w1=0.81358, mse=33.93884
Iteration 62: w0=0.79812, w1=0.82532, mse=33.67174
Iteration 63: w0=0.80947, w1=0.83701, mse=33.40676
Iteration 64: w0=0.82078, w1=0.84865, mse=33.14388
Iteration 65: w0=0.83205, w1=0.86024, mse=32.88309
Iteration 66: w0=0.84327, w1=0.87178, mse=32.62436
Iteration 67: w0=0.85445, w1=0.88328, mse=32.36769
Iteration 68: w0=0.86558, w1=0.89474, mse=32.11305
Iteration 69: w0=0.87667, w1=0.90614, mse=31.86042
Iteration 70: w0=0.88772, w1=0.91750, mse=31.60980
Iteration 71: w0=0.89873, w1=0.92882, mse=31.36117
Iteration 72: w0=0.90969, w1=0.94009, mse=31.11451
Iteration 73: w0=0.92061, w1=0.95131, mse=30.86981
Iteration 74: w0=0.93149, w1=0.96249, mse=30.62704
Iteration 75: w0=0.94232, w1=0.97362, mse=30.38620
Iteration 76: w0=0.95311, w1=0.98470, mse=30.14727
Iteration 77: w0=0.96387, w1=0.99575, mse=29.91023
Iteration 78: w0=0.97457, w1=1.00674, mse=29.67507
Iteration 79: w0=0.98524, w1=1.01770, mse=29.44178
Iteration 80: w0=0.99587, w1=1.02861, mse=29.21033
Iteration 81: w0=1.00645, w1=1.03947, mse=28.98072
Iteration 82: w0=1.01700, w1=1.05029, mse=28.75294
Iteration 83: w0=1.02750, w1=1.06107, mse=28.52695
Iteration 84: w0=1.03796, w1=1.07180, mse=28.30276
Iteration 85: w0=1.04838, w1=1.08249, mse=28.08035
Iteration 86: w0=1.05876, w1=1.09313, mse=27.85970
Iteration 87: w0=1.06910, w1=1.10374, mse=27.64079
Iteration 88: w0=1.07940, w1=1.11430, mse=27.42363
Iteration 89: w0=1.08966, w1=1.12481, mse=27.20818
Iteration 90: w0=1.09988, w1=1.13529, mse=26.99445
Iteration 91: w0=1.11006, w1=1.14572, mse=26.78240
Iteration 92: w0=1.12020, w1=1.15611, mse=26.57204
Iteration 93: w0=1.13030, w1=1.16646, mse=26.36335
Iteration 94: w0=1.14037, w1=1.17676, mse=26.15631
Iteration 95: w0=1.15039, w1=1.18703, mse=25.95091
Iteration 96: w0=1.16037, w1=1.19725, mse=25.74714
Iteration 97: w0=1.17032, w1=1.20743, mse=25.54499
Iteration 98: w0=1.18022, w1=1.21757, mse=25.34444
Iteration 99: w0=1.19009, w1=1.22767, mse=25.14548
Iteration 100: w0=1.19992, w1=1.23773, mse=24.94809
```

Exercise 1.1

As you can see, the mean squared error decreases every iteration of gradient descent. Currently the gradient descent runs for 20 iterations. Making changes in the above code so that it now runs for 100 iterations. Observing the mean squared error now. You will see that the mean squared error stops decreasing any further

by an appreciable amount after 50 iterations. But since you have specified that the gradient descent should run for 100 iterations, therefore, the program will run for 100 iterations, although the last 50 iterations would be pretty much useless and a waste of your compute resources. In our case, we have a very small dataset, and each iteration takes a very small amount of time, but real-world datasets can be huge, and each iteration of gradient descent will take a considerable amount of time. So you can't afford to run gradient descent if it no longer yields a decrease in the loss.

Your task is to now modify `gradient_descent_runner` function to implement an early stop such that if the improvement in error loss between two consecutive iterations of gradient descent steps is less than a certain user-specified threshold then the algorithm stops, otherwise the algorithm runs for a maximum of `num_iterations`. Name your function as `gradient_descent_runner_early_stop`

Paste the code for modified `gradient_descent_runner_early_stop` function in the block provided below (the code can be Python or R):

In [39]:

```
# TODO:
# Paste your code below for the modified gradient_descent_runner_early_stop function

def gradient_descent_runner_early_stop(points, starting_w0, starting_w1, learning_rate, num
    w0 = starting_w0
    w1 = starting_w1
    old_mse = float('inf')
    for i in range(num_iterations):
            w0, w1 = step_gradient(w0, w1, points, learning_rate)
            mse = compute_error_for_line_given_points(w0, w1, points)
            check = old_mse - mse
            if (check < threshold):
                print(f'Iteration {i+1}: w0={w0:0.5f}, w1={w1:0.5f}, mse={mse:0.5f}')
                print('Stopped')
                break
            old_mse = mse
            print(f'Iteration {i+1}: w0={w0:0.5f}, w1={w1:0.5f}, mse={mse:0.5f}')
    return [w0, w1, mse]

num_iterations = 20
learning_rate = 0.01
initial_w0 = 0 # initial y-intercept guess
initial_w1 = 0 # initial slope guess
threshold = 0.0001
[w0, w1, mse] = gradient_descent_runner_early_stop(points, initial_w0, initial_w1, learning
```

```
Iteration 1: w0=0.14469, w1=0.15019, mse=50.73447
Iteration 2: w0=0.28372, w1=0.29433, mse=46.80348
Iteration 3: w0=0.41732, w1=0.43266, mse=43.17860
Iteration 4: w0=0.54571, w1=0.56540, mse=39.83600
Iteration 5: w0=0.66908, w1=0.69279, mse=36.75368
Iteration 6: w0=0.78764, w1=0.81503, mse=33.91137
Iteration 7: w0=0.90158, w1=0.93233, mse=31.29038
Iteration 8: w0=1.01109, w1=1.04488, mse=28.87346
Iteration 9: w0=1.11633, w1=1.15287, mse=26.64473
Iteration 10: w0=1.21748, w1=1.25649, mse=24.58952
Iteration 11: w0=1.31470, w1=1.35591, mse=22.69432
Iteration 12: w0=1.40814, w1=1.45130, mse=20.94666
Iteration 13: w0=1.49797, w1=1.54281, mse=19.33506
Iteration 14: w0=1.58431, w1=1.63060, mse=17.84891
Iteration 15: w0=1.66731, w1=1.71482, mse=16.47845
Iteration 16: w0=1.74710, w1=1.79562, mse=15.21465
Iteration 17: w0=1.82381, w1=1.87312, mse=14.04923
Iteration 18: w0=1.89755, w1=1.94745, mse=12.97450
Iteration 19: w0=1.96846, w1=2.01876, mse=11.98342
Iteration 20: w0=2.03663, w1=2.08715, mse=11.06945
```

Exercise 1.2

In the program above we had set the learning rate to 0.1. Using the original `gradient_descent_runner` function, first set the number of iterations to 100. Then try to run the code with two different values of learning rates:

1. a learning rate of 0.001

2. a learning rate of 1

Explain what you observe.

```
1. learning rate of 0.001
    - mse with 5 decimals from first iteration to last iteration
2. learning rate of 1
    - mse with incredible large numbers from first iteration to last iteration

Conclusion: Very large error with high learning rate
```

Exercise 1.3

Earlier in the document, you learned about the closed-form solution of linear regression:

$$\hat{w} = (X^T X)^{-1} X^T y$$

where y is a vector of output values, and X is matrix of input vectors. It is important to note that the y-intercept (w_0) is represented by a column of 1's in the X matrix.

This equation is called the Normal equation, and it can work only if the columns of the matrix X are linear independent, or in other words, the matrix X is invertible.

Once you solve the normal equation, the vector $\bar{w}$ will contain the parameter estimates (w_0 and w_1) of the line.

Your task is to implement the normal equation, and give it the same values of `X` and `y` as we have used above, and see what the matrix $\bar{w}$ contains. It should contain the parameter estimates of the fit.

How do the parameter estimates obtained by solving the normal equation compare to the ones obtained using the gradient descent algorithm?

Hints:

1. Add a column of 1's to X to using `np.hstack` . This column of 1's models for the y-intercept (w_0).
2. Use `inv` function in numpy to find the inverse of a matrix. inv is defined in `numpy.linalg` module so will need to import it from it.
3. To multiply two matrices `a` and `b` , use the following notation in python `a.dot(b)`
4. To transpose a matrix `c` use `c.T`

Students who wish to do this in R instead of Python would find this resource (https://www.statmethods.net/advstats/matrix.html) useful.

In [40]:

```
# TODO
# Write your solution here

#extract size of row and column
n,m = X.shape

#create column of ones
ones = np.ones((n, 1))

#append column with ones to the matrix
newX = np.hstack((X, ones))

#transpose matrix
XT = newX.T
#multiply two matrix
XTX = XT.dot(newX)
#find inverse of matrix
invXTX = np.linalg.inv(XTX)

XTy = XT.dot(y)
w = invXTX.dot(XTy)

print(w)
```

```
[[3.02129039]
 [4.45478709]]
```

Exercise 1.4

To brush up your calculus skills, derive the partial derivate of MSE that has *L2* penalty term included in it. In other words, we want you compute the following partial derivatives:

$$\frac{\partial}{\partial w_0}\left(\frac{1}{n}\sum_{i=1}^{n}[y_i-(w_1x_i+w_0)]^2+\lambda {w_0}^2\right)$$

$$\frac{\partial}{\partial w_1}\left(\frac{1}{n}\sum_{i=1}^{n}[y_i-(w_1x_i+w_0)]^2+\lambda {w_1}^2\right)$$

What to submit

A derivation of both the gradient equations.

$$\frac{\partial}{\partial w_0}\left(\frac{1}{n}\sum_{i=1}^{n}[y_i-(w_1x_i+w_0)]^2+\lambda {w_0}^2\right)=-\frac{2}{n}\left(\sum_{i=1}^{n}y_i-(w_1x_i+w_0)+\lambda w_0\right)$$

$$\frac{\partial}{\partial w_1}\left(\frac{1}{n}\sum_{i=1}^{n}[y_i-(w_1x_i+w_0)]^2+\lambda {w_1}^2\right)=-\frac{2}{n}\left(\sum_{i=1}^{n}x_i[y_i-(w_1x_i+w_0)]+\lambda w_1\right)$$

Linear Regression with *sklearn* Machine Learning Library

In the section above, we implemented our own linear regression with gradient descent. However, instead of re-inventing the wheel and writing the code for each machine learning algorithm ourselves, we can also follow the easier path of using someone else's code to do machine learning. That's the path that most people follow. Having said that, knowing how a particular algorithm works internally is a very important for understanding its nuances.

In this section, we will use a machine learning library called sklearn to perform linear regression on the same dataset and see what kind of results we get this time round.

In [41]:

```
# install sklearn if it isn't installed already
!pip install sklearn
```

```
Requirement already satisfied: sklearn in c:\users\phili\appdata\local\progr
ams\python\python37-32\lib\site-packages (0.0)
Requirement already satisfied: scikit-learn in c:\users\phili\appdata\local
\programs\python\python37-32\lib\site-packages (from sklearn) (0.19.2)
```

In [42]:

```
# import LinearRegression class from sklearn.linear_model module
from sklearn.linear_model import LinearRegression

# make a lin_reg object form the LinearRegression class
lin_reg = LinearRegression()

# use the fit method of LinearRegression class to fit a straight line through the data
lin_reg.fit(X, y)
```

Out[42]:

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

In [43]:

```
print('y-intercept w0:', lin_reg.intercept_)
print('slope w1:', lin_reg.coef_)
```

```
y-intercept w0: [4.45478709]
slope w1: [[3.02129039]]
```

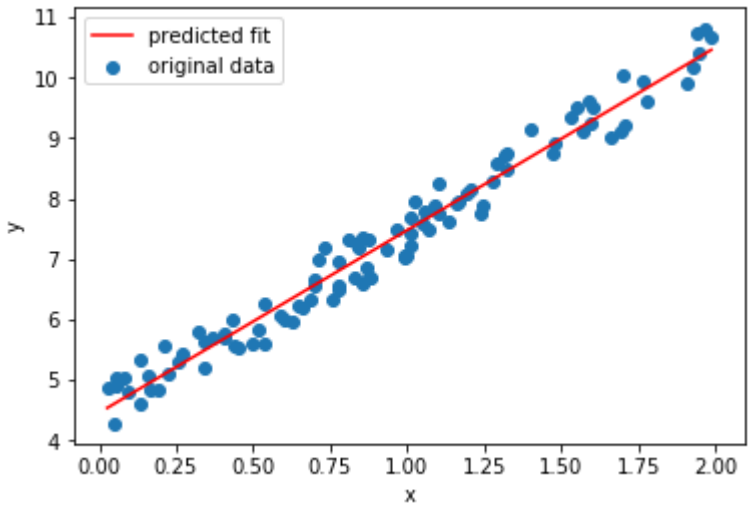
You can see that the estimates for w_0 and w_1 are very close to the estimates we obtained by using our gradient descent algorithm. Now lets plot the original data along with the fitted line.

In [44]:

```
import matplotlib.pyplot as plt
%matplotlib inline

# plot the original data points as a scatter plot
plt.scatter(X, y, label='original data')

# plot the line that fits these points. Use the values of m and b as provided by the fit me
y_ = lin_reg.coef_*X + lin_reg.intercept_

# you can also get y_ by using the predict method. Uncomment the line below:
#y_ = lin_reg.predict(X)

plt.plot(X, y_, color='r', label='predicted fit')
plt.xlabel('x'); plt.ylabel('y')
plt.legend(loc='best')
```

Out[44]:

```
<matplotlib.legend.Legend at 0x141f05d0>
```



Exercise 1.5

1. How does fit you got from sklearn library compare to the one you got form your implementation of the Normal equation.
2. Suppose you have a new data point `x=3` . Use the `predict` method provided by the `LinearRegression` class to find its corresponding y value.

N.B. Student who prefer R can implement linear regression using `lm()` method.

```
1. The fit we got from sklearn gives the same result as our implementation of the Normal
Equation

Result from Normal Equation:
    [[-0.03541352]
    [ 2.4189864 ]]
```

```
Result from sklearn library:
    y-intercept w0: [2.4189864]
    slope w1: [[-0.03541352]]
```

In [45]:

```
# TODO
# write you solution below
lin_reg.predict(3)
```

Out[45]:

```
array([[13.51865826]])
```

2. Multivariate Linear Regression

So far we we were using univariate linear regression. Let us now discuss multivariate linear regression. In multivariate regression, the dependent variable is modeled as a linear combination of multiple independent variables.

To study multivariate regression, we are going to use the Hollywood movies dataset. This dataset is in `movies.csv` file. It has four columns and their description is as following:

- `revenue` = Total revenue obtained in the first year of box office release in millions
- `production_cost` = Total cost in million in producing the movie
- `promotional_cost` = Total cost in millions in promoting the movies
- `book_sales` = Total sales in millions of the movie's book

We will now try to model the revenue as a linear combination of the production_cost, promotional_cost, and book_sales using Multivariate Linear Regression.

In [46]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# make a dataframe of the data
df = pd.read_csv('movies.csv')

# show first five rows of df
df.head(n=5)
```

Out[46]:

	revenue	production_cost	promotional_cost	book_sales
0	85.099998	8.5	5.100000	4.7
1	106.300003	12.9	5.800000	8.8
2	50.200001	5.2	2.100000	15.1
3	130.600006	10.7	8.399999	12.2
4	54.799999	3.1	2.900000	10.6

In [47]:

```
# Extract the first column and set it to the output or dependent varaible y
y = df[['revenue']]

# Remove the first column and set the rest of the dataframe to X. This is the set of indepe
X = df.drop(columns=['revenue'])

# show first five rows of X
X.head(n=5)
```

Out[47]:

	production_cost	promotional_cost	book_sales
0	8.5	5.100000	4.7
1	12.9	5.800000	8.8
2	5.2	2.100000	15.1
3	10.7	8.399999	12.2
4	3.1	2.900000	10.6

In [48]:

```
# show first five rows of y
y.head(n=5)
```

Out[48]:

	revenue
0	85.099998
1	106.300003
2	50.200001
3	130.600006
4	54.799999

Now we will fit a Multivariate Linear Regression model to it:

In [49]:

```
from sklearn.linear_model import LinearRegression

# make a lin_reg object form the LinearRegression class
lin_reg = LinearRegression()

# use the fit method of LinearRegression class to fit a straight line through the data
lin_reg.fit(X, y)

# Display the learned parameters
lin_reg.intercept_, lin_reg.coef_
```

Out[49]:

```
(array([7.67602854]), array([[3.66160401, 7.62105126, 0.82846807]]))
```

Exercise 2.1

What would be the first year box office revenue of a movie which costed 23 million dollars to make, 12 million dollars to promote, and had total book sales of 10 million dollars.

N.B: Students you want to attempt this exercise in R can use `lm()` function to do multiple linear regression.

In [50]:

```
## TODO
## Write your code here
lin_reg.predict([[23, 12, 10]])
```

Out[50]:

```
array([[191.6302165]])
```

Multivariate Regression with Polynomial basis

What if your data is actually more complex than a simple straight line, or a plane? Surprisingly, you can actually use a linear model to fit nonlinear data as well. A simple way to do this is to add powers of each feature as new features, and then train a linear model on this extended set of features. This technique is called Polynomial Regression. Because we are modeling the response variable as linear combination of the polynomial features, therefore this regression is classed as multivariate regression.

Lets look at an example. First, lets generate some nonlinear data, based on a simple quadratic equation (plus some noise).

$$y = 0.5X^2 + X + 2 + noise$$

In [51]:

```
# define the number of points to generate as k
k = 100

# define a seed value. It is important to define the seed value
# so that the random numbers generated are the same every time
# this code is executed.
np.random.seed(10)

# generate k x-axis values from -3 to +3
X = 6 * np.random.rand(k, 1) - 3

# sort the numbers in ascending order. This helps when we are plotting the data.
# Without this line, your plots will be all jumbled up
X.sort(axis=0)

# generate k y-axis values
y = 0.5 * X**2 + X + 2 + np.random.rand(k, 1)
```

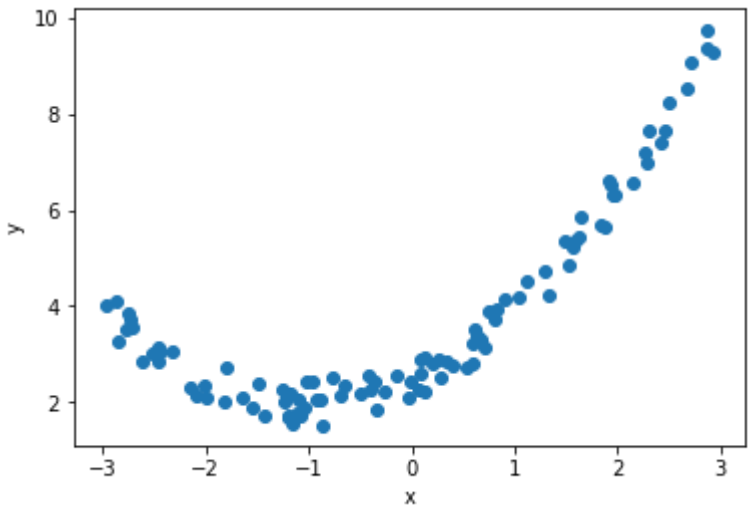
Let us now plot the data:

In [52]:

```
plt.scatter(X, y)
plt.xlabel('x'); plt.ylabel('y')
```

Out[52]:

```
Text(0,0.5,'y')
```



Clearly, a straight line will never fit this data properly. So lets use Scikit-Learn's `PolynomialFeatures` class to transform our training data by adding the square of X in the training set as new feature.

In [53]:

```
from sklearn.preprocessing import PolynomialFeatures

poly_features = PolynomialFeatures(degree=2, include_bias=False)

# generate polyonimal features upto degree 2 from the vector X
X_poly = poly_features.fit_transform(X)
```

In [54]:

```
# display 4 original data points
X[1:5]
```

Out[54]:

```
array([[-2.8754883 ],
       [-2.84760131],
       [-2.7643094 ],
       [-2.76024475]])
```

In [55]:

```
# Display the transformed data.
# You will now see the original X data alongside its corresponding 2nd-degree polynomial fe
X_poly[1:5]
```

Out[55]:

```
array([[-2.8754883 ,  8.26843299],
       [-2.84760131,  8.10883321],
       [-2.7643094 ,  7.64140644],
       [-2.76024475,  7.61895107]])
```

Now we fit a linear regression model to the transformed data:

In [56]:

```
lin_reg = LinearRegression()

# Now we fit a linear model to the X_poly (the transformed features set) and y
lin_reg.fit(X_poly, y)

# show the values of intercept and learned co-efficients
lin_reg.intercept_, lin_reg.coef_
```

Out[56]:

```
(array([2.48183064]), array([[0.99509291, 0.49139048]]))
```

So the fit estimated by polynomial regression has the following form:

$$\bar{y} = 0.49X^2 + 0.99X + 2.48$$

which is pretty close the function we used to generate the original the original data:

$$y = 0.5X^2 + X + 2 + noise$$

(N.B.: The values that you might get for `lin_reg.intercept_` and `lin_reg.coef_` may be a bit different. This is because you might be using a different computer with a different operating system, different precision etc. All this might lead a slightly different values of intercept and co-efficients.)

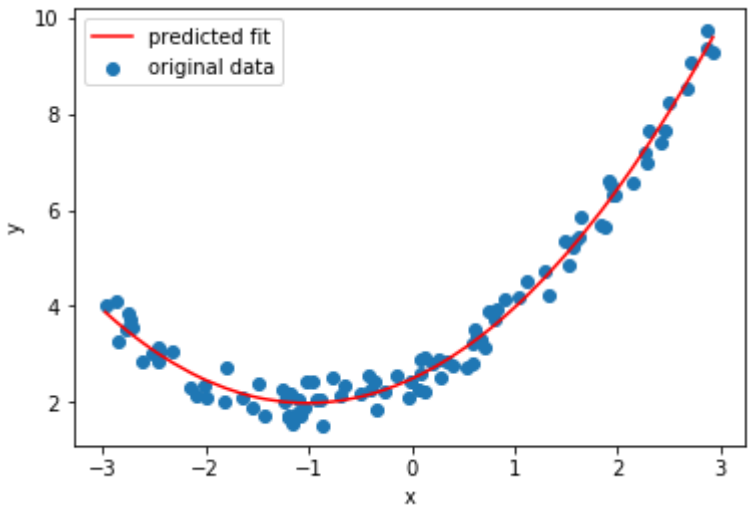
Let's now plot the original data (in blue) and the predicted fit (in red):

In [57]:

```
y_ = lin_reg.predict(X_poly)

plt.scatter(X, y, label='original data')
plt.plot(X, y_, color='r', label='predicted fit')
plt.legend(loc='best')

plt.xlabel('x')
plt.ylabel('y')
```

Out[57]:

```
Text(0,0.5,'y')
```



Try it yourself

Change the value of polynomial degree, and see what happens to the fitted line.

Try values of 5, 10, and 15. You don't need to submit anything for this one.

Regularization with Ridge Penalty

As you might have noticed from the above task, as the degree of polynomial increases, the model starts to overfit. Model overfitting often happens when you have:

- lots of features, and
- too little data per feature.

The model starts memorizing the data, rather than generalizing.

To prevent model from overfitting, we can do regularization. In regularization, we add a penalty term to the loss function. This penalty term prevents the model from overfitting.

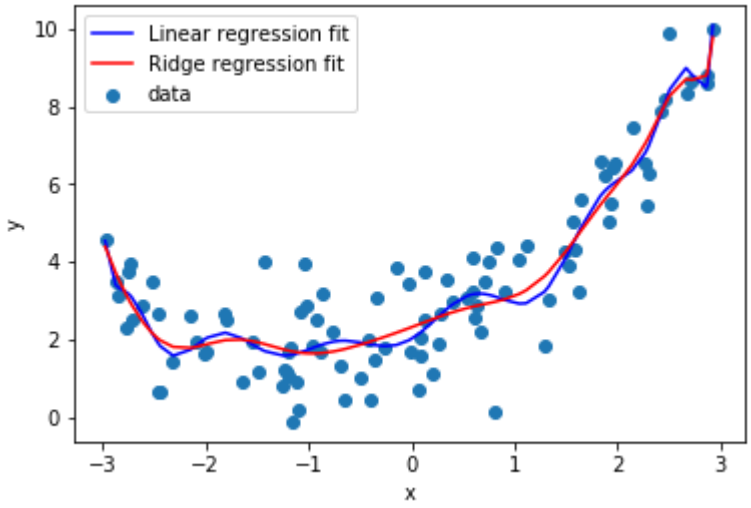
Here, we will write the code for Ridge regression.

In [58]:

```
from sklearn.linear_model import Ridge
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

# define the number of points to generate
k = 100
np.random.seed(10)

# generate k x-axis values from -3 to +3
X = 6 * np.random.rand(k, 1) - 3
X.sort(axis=0)

# generate k y-axis values
y = 0.5 * X**2 + X + 2 + np.random.randn(k, 1)

# Create polynomial feature (degree 15)
poly_features = PolynomialFeatures(degree=15, include_bias=False)
X_poly = poly_features.fit_transform(X)

# Create Ridge regression object from Ridge class
#ridge_reg = Ridge(alpha=5) -- Original
ridge_reg = Ridge(alpha=0.5)

# Fit data using Ridge regression
ridge_reg.fit(X_poly, y)

# Create Linear regression object from LinearRegress class (this is just for comparison)
lin_reg = LinearRegression()
# Fit data using Linear regression
lin_reg.fit(X_poly, y)

y_predict_ridge = ridge_reg.predict(X_poly)
y_predict_linear = lin_reg.predict(X_poly)

plt.scatter(X, y, label='data')
plt.plot(X, y_predict_linear, color='b', label='Linear regression fit')
plt.plot(X, y_predict_ridge, color='r', label='Ridge regression fit')

plt.xlabel('x')
plt.ylabel('y')

plt.legend(loc='best')
```

Out[58]:

```
<matplotlib.legend.Legend at 0x142e6ef0>
```



As you can see in the plot above, linear regression leads to overfitting (jiggly blue fit in the plot above).

Ridge regression with `alpha=5` yields a fit that isn't overfitted (smooth red curve).

Exercise 2.2

Change the value of regularization parameter to 0.05 and see what happens. Explain what you observe?

N.B. Students who wish to do this exercise in R can use the `lm.ridge()` function.

TODO

Write you answer here:

```
Changing the regularization parameter to 0.05 (alpha=0.05) will result in a change in
Ridge regression fit to be more like the Linear regression fit.

With 0.05 as a parameter, the Ridge regression fit will also lead to overfitting.
A smaller alpha will change the Ridge regression to become much more like the Linear
regression fit
```

Type *Markdown* and LaTeX: α^2

Linear Regression with Radial Basis Functions

In the lecture, you learned about the radial basis functions (RBF). A radial basis function is a real-valued function whose value depends only on the distance from some point `c` called a center. We can use each point of our data as center of an RBF and then use a weighted sums of these radial basis functions to approximate the fit for the original data. In this case multivariate linear regression can be used to find the weights (or coefficients) of this weighted sum of `m` radial basis functions, where `m` are the number of data points in the dataset for which we want to find a fit.

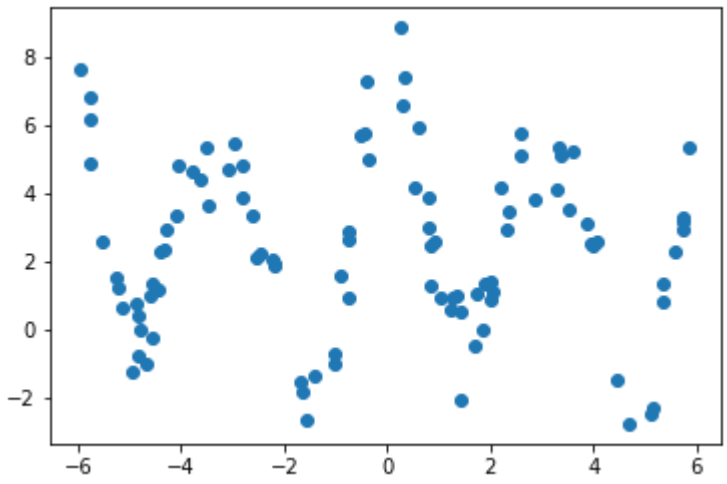
Here you will see an implementation of multivariate linear regression with radial basis functions.

In [59]:

```
# Set random seed
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
m = 100

# Create random set of m x values between -6 and +6
X = np.random.rand(m, 1)*12 - 6
X.sort(axis =0)

# Create a non-linear dataset with random noise
y = 0.5*np.cos(X) + np.sin(X) + 4*np.cos(2*X) + np.exp(np.cos(3*X)) + 3*np.random.rand(m,1)

# plot it
plt.scatter(X, y)
```

Out[59]:

```
<matplotlib.collections.PathCollection at 0x1432bf30>
```

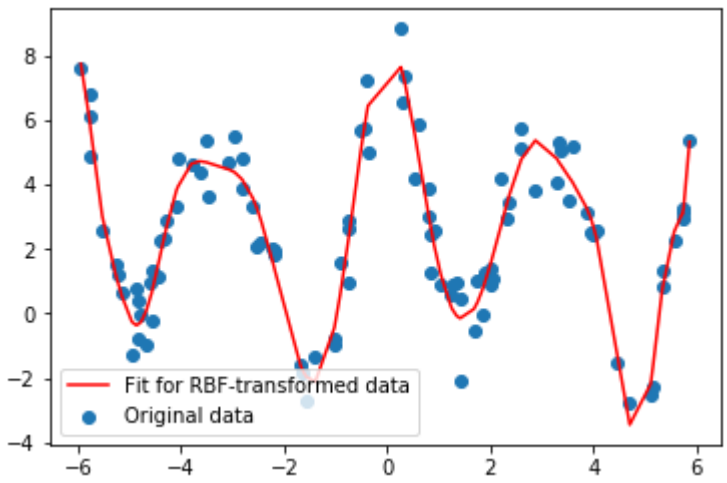


In [60]:

```
from sklearn.metrics.pairwise import rbf_kernel
from sklearn.linear_model import LinearRegression

# find the transformation of X using Radial Basis Functions
# Each point in X is now modeled as vector of 100 values.
# See the X_RBF.shape and X_RBF to find how rbf_kernel transformed
# the original datapoints
X_RBF = rbf_kernel(X, X, gamma=0.1)

# Fit a linear regression model to the RBF-transformed data
clf = LinearRegression()
clf.fit(X_RBF, y)

# find the predicted values
y_= clf.predict(X_RBF)

# plot original data and predicted fit
plt.scatter(X, y, label='Original data')
plt.plot(X, y_, color='r', label='Fit for RBF-transformed data')
plt.legend(loc='best')
```

Out[60]:

```
<matplotlib.legend.Legend at 0x143669b0>
```



Exercise 2.3

Plot the first 0th, 99th, and 49th radial basis of X on the same plot as a line graph.

HINT:

X_RBF contains 100 radial basis functions, corresponding to each of the 100 data points. All you need to do is to index them, and then plot them.

N.B.: Students who wish to do this exercise in R should refer to this resource for RBF (http://www.di.fc.ul.pt/~jpn/r/rbf/rbf.html).

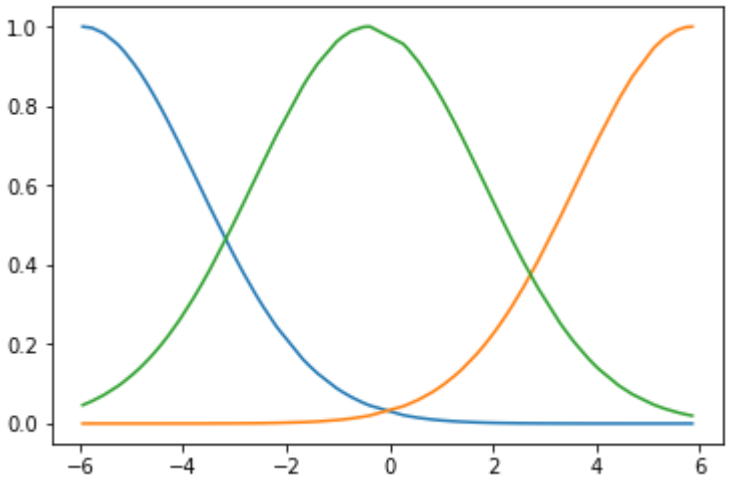
Paste your code in the cell below:

In [61]:

```
# TODO
# Paste your solution here
plt.plot(X, X_RBF[0], label = '0th')
plt.plot(X, X_RBF[99], label = '99th')
plt.plot(X, X_RBF[49], label = '49th')
```

Out[61]:

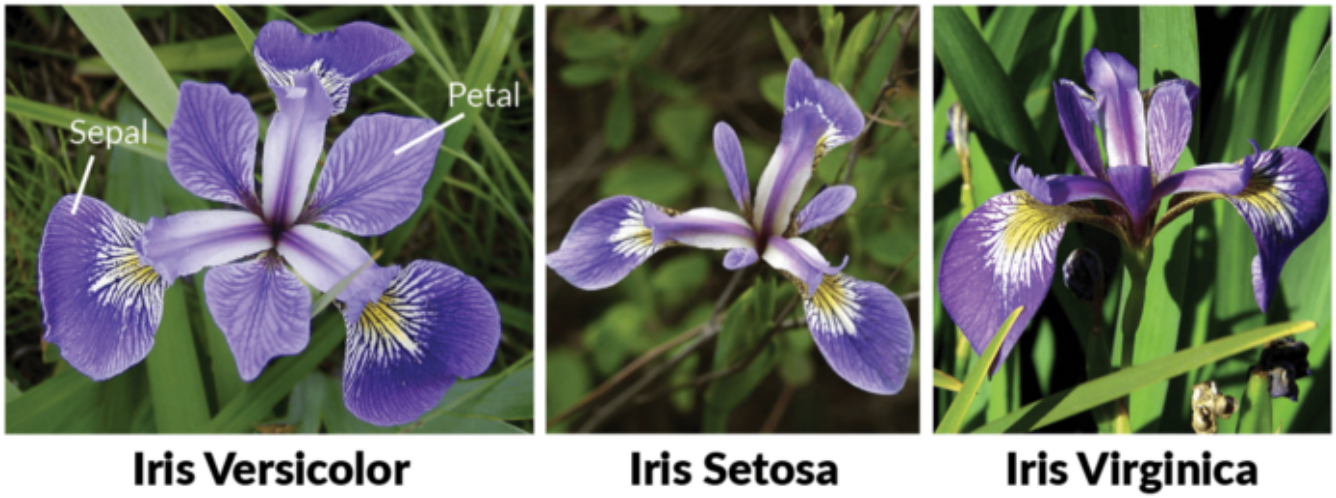
```
[<matplotlib.lines.Line2D at 0x143af570>]
```

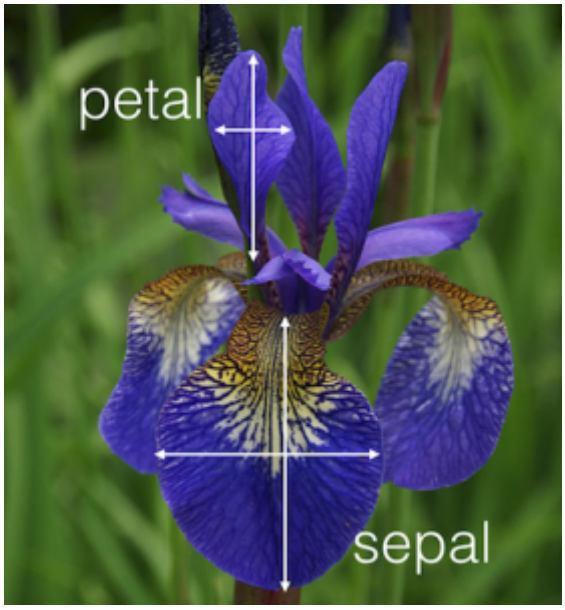


3. Logistic Regression

Logistic Regression (also called Logit Regression) is a generalized linear model which is commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?). If the estimated probability is greater than 0.5, then the model predicts that the instance belongs to that class (called the positive class, labeled “1”), or else it predicts that it does not (i.e., it belongs to the negative class, labeled “0”). This makes it a binary classifier.

Let’s use the iris dataset to illustrate Logistic Regression. This is a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: Iris-Setosa, Iris-Versicolor, and Iris-Virginica.





Iris data is already present in the sklearn library, and is essentially the same as the one you used during your first weekly exercise. All we need to do is to import it.

In [62]:

```
from sklearn import datasets
iris = datasets.load_iris()

# iris is a dictionary of key-value pairs. Each key-value pairs contains some information a
# Lets display a list of these keys and see what they hold
list(iris.keys())
```

Out[62]:

```
['data', 'target', 'target_names', 'DESCR', 'feature_names']
```

- `data` : holds the data of sepal and petal lengths and widths in four columns,
- `target` : holds the class of each flower. These class are encoded as 0, 1, and 2,
- `target_names` : holds the names of each of the flower classes,
- `DESCR` : contains a detailed description of the dataset, and
- `feature_names` : contains a list of name of the columns of data

In [63]:

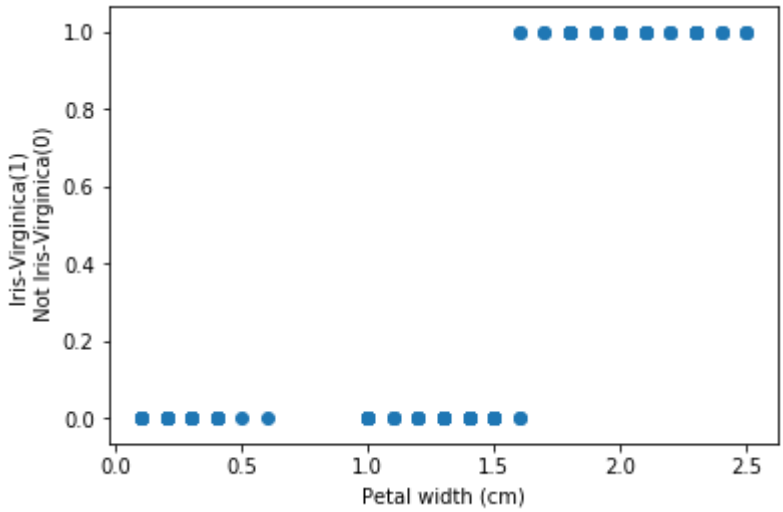
```
# let us get the petal width. It is present in the 4th column of data
X = iris["data"][:, 3:]
X.sort(axis=0)

# lets define a binaray variable that encodes whether a flower is Iris-Virginca or not
# Iris_virginca flower is encoded as a 2 in target
y = (iris["target"] == 2).astype(np.int) # 1 if Iris-Virginica, else 0
```

In [64]:

```
plt.scatter(X, y)
plt.xlabel('Petal width (cm)')
plt.ylabel('Iris-Virginica(1) \n Not Iris-Virginica(0)')
```

Out[64]:

```
Text(0,0.5,'Iris-Virginica(1) \n Not Iris-Virginica(0)')
```



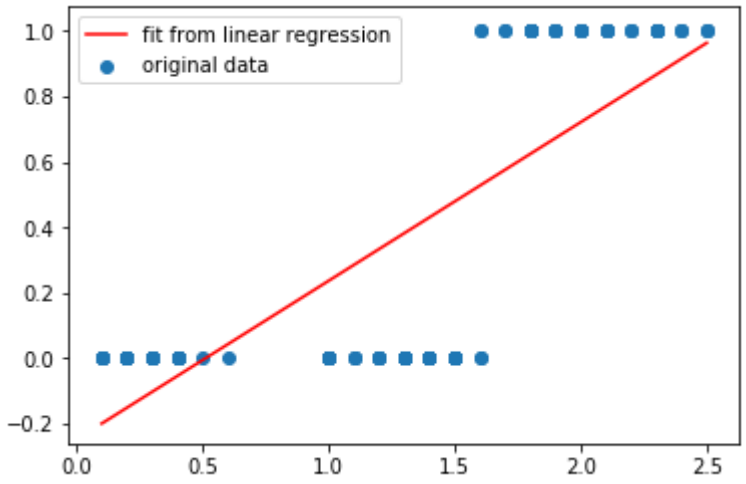
Let us first try the naive thing of fitting a linear model to this data.

In [65]:

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X, y)
y_ = lin_reg.predict(X)

plt.scatter(X, y, label='original data')
plt.plot(X, y_, color='r', label='fit from linear regression')
plt.legend(loc='best')
```

Out[65]:

```
<matplotlib.legend.Legend at 0x145101d0>
```



A linear regression model cannot be an optimal fit for such dichotomous data. Lets try now to fit a logistic regression model to this data.

In [66]:

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)
```

Out[66]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

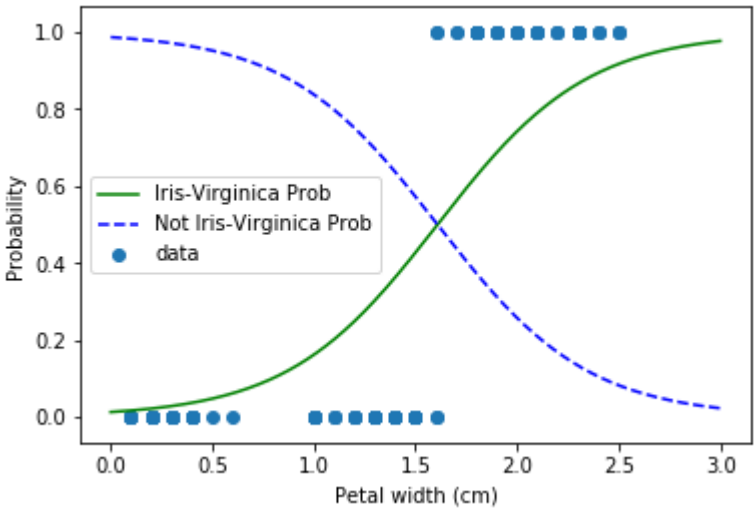
Let’s now look at the model’s estimated probabilities for flowers with petal widths varying from 0 to 3 cm

In [67]:

```
# we generate X_new which is vector of closely spaced points form 0 to 3
# This vector will help us plot the model
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)

# make a vector of prediction probablity values for all datapoints in X_new
y_proba = log_reg.predict_proba(X_new)

plt.plot(X_new, y_proba[:, 1], "g-", label="Iris-Virginica Prob")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris-Virginica Prob")
plt.scatter(X, y, label='data')

plt.xlabel('Petal width (cm)')
plt.ylabel('Probability')
plt.legend(loc='best')
```

Out[67]:

```
<matplotlib.legend.Legend at 0x14558ad0>
```



The petal width of Iris-Virginica flowers ranges from 1.4 cm to 2.5 cm, while the other iris flowers generally have a smaller petal width, ranging from 0.1 cm to 1.8 cm.

Notice that there is a bit of overlap. Above about 2 cm the classifier is highly confident that the flower is an Iris-Virginica (it outputs a high probability to that class), while below 1 cm it is highly confident that it is not an Iris-Virginica (high probability for the “Not Iris-Virginica” class). In between these extremes, the classifier is unsure. However, if you ask it to predict the class (using the `predict()` method rather than the `predict_proba()` method), it will return whichever class is the most likely. Therefore, there is a decision boundary at around 1.6 cm where both probabilities are equal to 0.5 (or 50%): if the petal width is higher than 1.6 cm, the classifier will predict that the flower is an Iris-Virginica, or else it will predict that it is not (even if it is not very confident).

Let's try to predict the class of a flower that has a petal width of 1.7.

In [68]:

```
log_reg.predict([[1.7]])
```

Out[68]:

```
array([1])
```

So the class predicted is 1 or Iris-Virginica. Let us find out how sure the classifier was while making this decision.

In [69]:

```
log_reg.predict_proba([[1.7]])
```

Out[69]:

```
array([[0.43834057, 0.56165943]])
```

So the classifier was about 56% sure that the flower is Iris-Virginica.

Exercise 3.1

In the example above, we discussed a two class classification problem i.e, we built a logistic regression classifier that was able to distinguish between Iris-Virginica and non-Iris-Virginica flowers based on just a single feature: the petal width.

Your task now is build a multi-class classifier that can distinguish between Iris-Virginca, Iris-Setosa, and Iris-Versicolor. Furthermore, instead of using one feature, now you have to use two features — petal length and petal width — to train your model.

Using the model that you trained, predict the most probable class for a flower that has petal length and width of 1 and 0.1 cm, respectively. What is probability value of this most probable class.

For those who prefer to work in R, you can use the `glm()` function to perform logistic regression. More on it here (https://www.datacamp.com/community/tutorials/logistic-regression-R).

Hint:

1. Make an object from LogisticRegression class in sklearn as following:

`multiclass_logreg_obj = LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10)` .

`multiclass_logreg_obj` is just a name. It could be any (appropriate) name you like.

Read more about the Logistic Regression parameters in the online documentation (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html). Understanding C, solver, and multi_class is important for this assignment.

In [70]:

```
multiclass_logreg_obj = LogisticRegression(multi_class="multinomial", solver="lbfgs", C=10)

#data: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
X_iris = iris["data"][:, [2,3]]
#target: 0=Iris Versicolor, 1=Iris Setosa, 2=Iris-Virginica
y_iris = (iris["target"])

multiclass_logreg_obj.fit(X_iris, y_iris)

flower = multiclass_logreg_obj.predict([[1, 0.1]])[0]
print(flower)
probability = multiclass_logreg_obj.predict_proba([[1, 0.1]])[0][0]
print(probability)
```

```
0
0.9995796959204409
```

The flower we get with petal length and width of 1 and 0. cm resepctively is Iris Versicolor. The probability for this flower is 99%

Sources

1. https://github.com/mattnedrich/GradientDescentExample/blob/master/gradient_descent_example.py (https://github.com/mattnedrich/GradientDescentExample/blob/master/gradient_descent_example.py)
2. Hands-On Machine Learning with Scikit-Learn and TensorFlow by Aurélien Géron

For further details on linear regression, we recommend watching the Linear Regression lectures in the Machine Learning Course by Andrew Ng on Coursera.