Machine Learning HW 1

32190984 김이수

Task 0.

Environment

The code that was submitted with this report was tested and checked it working in following environment.

- Python 3.9
- **Numpy 1.12.5**: For processing math and matrix operations.
- **Pandas 1.4.2**: For loading csv file easily.
- **Matplotlib 3.5.1**: For visualization.
- With Jupyter Notebook

Files

The files that are attached with this document is like below:

- **Task1.ipynb**: The solution code for Task 1.
- **Task2.ipynb**: The solution code for Task 2.

You can run my code by opening **jupter notebook** in installed in your PC and run line by line (Shift + Enter) to get results and see how results are represented.

Why Python?

I personally love C++ and C over Python due to the fact that Python is painfully slow compared to C++ and C. However, since Python provides us with powerful libraries such as Numpy (for matrix operations), Pandas (for processing csv data) and Matplotlib (for visualizing data), I chose Python over C++ or C this time. Also, don't forget the fact that with C++ we have to implement everything from ground up which will take us time to just implement matrix operations. Let's try not to reinvent the wheel.

Task 1.

Requirements

- Model = $y = ax + b$
- Approach: Gradient Descent.

Assumption

- Using MSE for cost function.
- We will use a and b instead of θ_1 and θ_0 respectively.

Design and Theory

It is well known fact that gradient descent uses following equation for updating weights:

$$\theta_j := \theta_j - \alpha * \frac{\partial}{\partial\theta_j} J(\theta_0, \theta_1)\ where\ \alpha = Learning\ Rate$$

In our case, we use following equations.

$$h(x) = ax + b$$

$$J(a,b) = \frac{1}{2m}\sum_{i=1}^{m}(h(x_i) + y_i)^2$$

$$\because a = \theta_1, b = \theta_0$$

Therefore, with those two formulas combined, we can derive an update rule for weights.

$$a := a - \alpha\frac{\partial}{\partial a}\frac{1}{2m}\sum_{i=1}^{m}(h(x_i) + y_i)^2 = a - \alpha * \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i) * x_i$$

$$b := b - \alpha\frac{\partial}{\partial b}\frac{1}{2m}\sum_{i=1}^{m}(h(x_i) + y_i)^2 = b - \alpha * \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)$$

And we will update those update rules that was mentioned right before until a and b converges. (Meaning that this procedure will be iterated until a and b does not change any more)

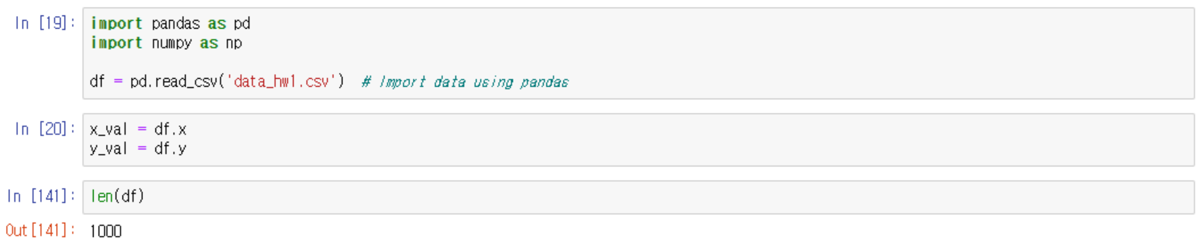
Now, let's implement this equation using Python. The code that was used can be found in "**Task1.ipynb**" that was attached with this document.

Process

The code will be executed in following sequence

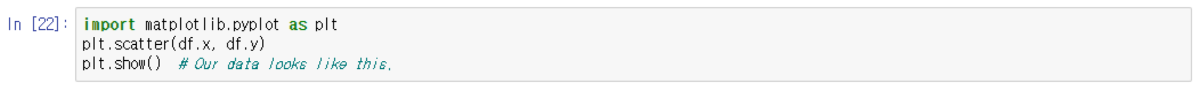
1. Read csv file using Pandas.
2. From the csv, generate Pandas.DataFrame for storing csv data.
3. From the Pandas.DataFrame, process Gradient Descent using formula mentioned before.
4. If Gradient Descent was finished (might have been converged, or maxed out iteration count), represent results.

Data

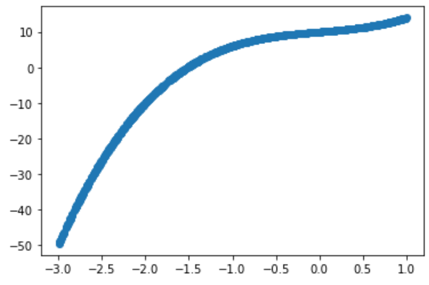


```
In [19]:  import pandas as pd
          import numpy as np

          df = pd.read_csv('data_hw1.csv')  # Import data using pandas

In [20]:  x_val = df.x
          y_val = df.y

In [141]: len(df)

Out[141]: 1000
```

data_hw1.csv has 1000 pairs of x and y. Now, let's see how overall data is plotted.



```
In [22]:  import matplotlib.pyplot as plt
          plt.scatter(df.x, df.y)
          plt.show()  # Our data looks like this.
```



With matplotlib, we can plot how it looks like this.

Gradient Descent

For Gradient Descent, I wrote a simple code that does everything and returns best a and b values named "do_gradient_descent". The function takes 4 parameters as inputs.

```
def do_gradient_descent(max_iter, learning_rate, data_x, data_y):
```

The function's parameters are as it follows below:

```
@param max_iter: The maximum iteration count for training.
@param learning_rate: The learning rate for gradient descent.
@param data_x: The x data.
@param data_y: The y data.
@return list object that contains (a, b)
```

For example, if we would like to perform linear regression with data provided, we will call the function like below.

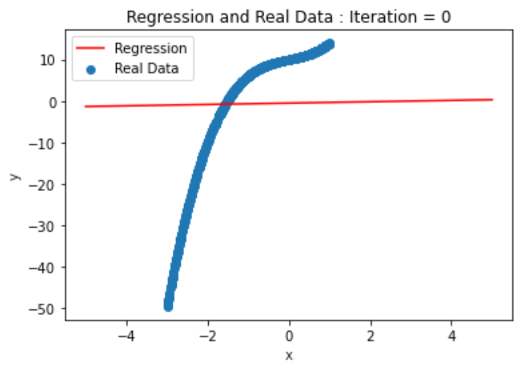
```
a, b = do_gradient_descent(1000, 0.1, df.x, df.y)  # Run 1000 times with learning rate of 0.1
```

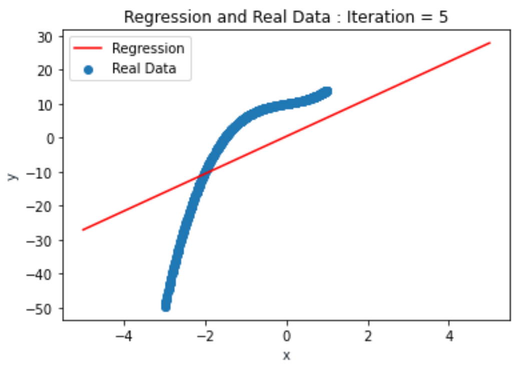
Since, the function has enough comments and documentation, in this report those code details will not be mentioned.

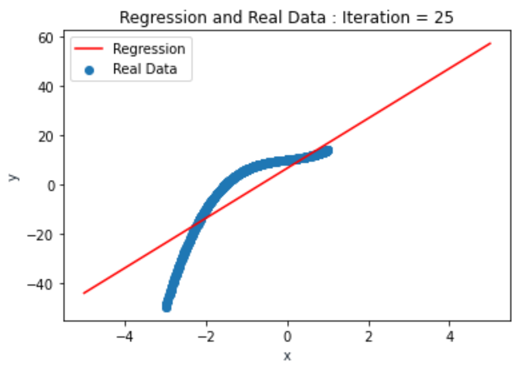
Results

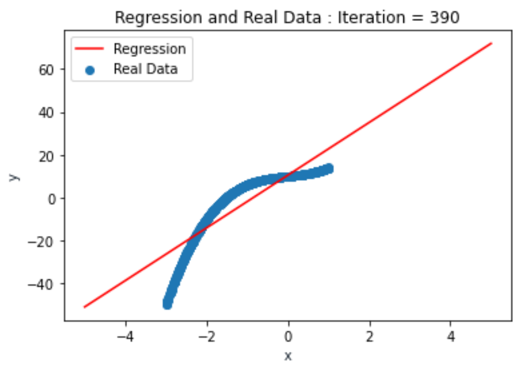
```
[+] Iter : 665 / a = 12.288069059865048 / b = 10.493209154984159
[+] Iter : 670 / a = 12.288069059865057 / b = 10.493209154984175
[+] Iter : 675 / a = 12.288069059865064 / b = 10.493209154984184
[+] Iter : 680 / a = 12.28806905986507 / b = 10.493209154984193
[+] Iter : 685 / a = 12.288069059865073 / b = 10.493209154984202
[+] Iter : 690 / a = 12.288069059865077 / b = 10.49320915498421
Convergence!!! Stop training.
```

After iterations, Gradient Descent function stopped since convergence occured. Meaning that the value a and b stayed the same. Now, Let's see how it went during those iterations.

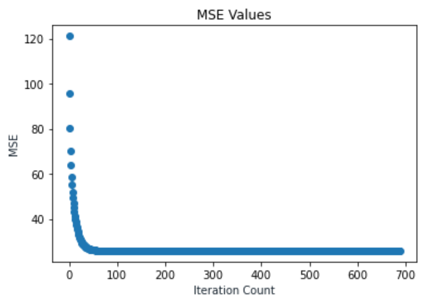




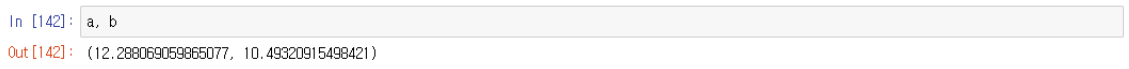




As iteration went on, our gradient descent model was fitting the graph into the data.



MSE values and error goes down as iteration goes down as well.



```
In [142]: a, b
Out[142]: (12.288069059865077, 10.49320915498421)
```

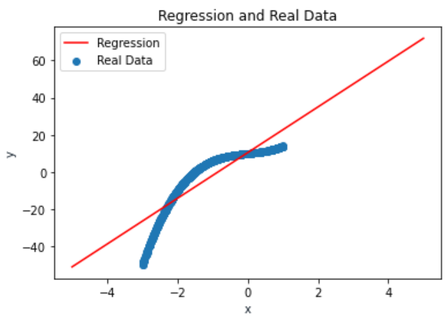
The final values of a and b is as it follows:

a	b
12.288069059865077	10.49320915498421

a roughly had value of 12.2880 and b roughly had value of 10.4932. Thus, the original hypothesis will be as it follows:

$$h(x) = 12.2880x + 10.4932$$

Let's see how it is plotted with the original data.

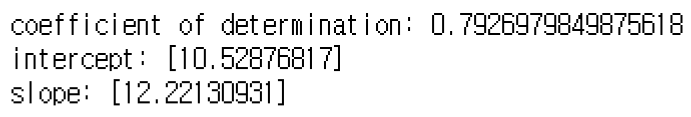


Comparison with SKlearn

For fun and for comparison between our regression model and the model generated by professional libraries, let's check how sklearn.linear_model.SGDRegressor returns a and b. (This is using SGD regressor which is slightly different from just GD, however this will let us know rough information about a and b)

The original code that implemented using Sklearn with SGD Regressor can be found https://www.activestate.com/resources/quick-reads/how-to-run-linear-regressions-in-python-scikit-learn/. I modified the code a bit so that it can use SGD Regressor.

The regression result is like below:


```
coefficient of determination: 0.7926979849875618
intercept: [10.52876817]
slope: [12.22130931]
```

Which meant following a and b.

a	b
12.22130931	10.52876817

Conclusion & Comparision

Method	a	b
Gradient Descent (We made!)	12.288069059865077	10.49320915498421
SGDRegressor (From Sklearn)	12.22130931	10.52876817

By comparing both two results, it seems like our gradient descent algorithm was not too bad finding out a and b. Also, the total MSE that our gradient descent generated was roughly 25.8568.

Since gradient descent tried to fit a “line” into a “curved” data, there are some flaws. In order for us to fit a model better into the “curved” data, it is suggested to use other functions such as “quadratic” functions. Let’s see how “quadratic” function fits our “curved” data in Task 2.

Task 2.

Requirements

- Model = $y = ax^2 + bx + c$
- Approach: Normal Equation

Assumption

- Using MSE for cost function. (Which means Euclidian norm)

Design and Theory

So, even if the model that we are dealing with is a quadratic function, the same rules and equations can be applied as if we were using a linear function. We can use following expression in order to represent our data with a, b and c.

$$\begin{bmatrix} (-1.572851214)^2 & -1.572851214 & 1 \\ \vdots & \vdots & \vdots \\ (0.675706147)^2 & 0.675706147 & 1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} -0.927732791 \\ \vdots \\ 0.675706147 \end{bmatrix}$$

Let

$A = \begin{bmatrix} (-1.572851214)^2 & -1.572851214 & 1 \\ \vdots & \vdots & \vdots \\ (0.675706147)^2 & 0.675706147 & 1 \end{bmatrix}$ (1000×3 since we have 1000 entries for x)

$X = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ (3×1 since we have a, b and c.)

$b = \begin{bmatrix} -0.927732791 \\ \vdots \\ 0.675706147 \end{bmatrix}$ (1000×1 since we have 1000 entries which includes data y)

We can make a simple formula out of those three matrices

$$A \cdot X = b$$

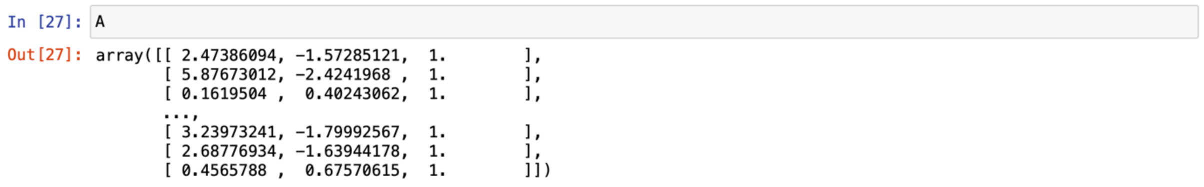
In page 36 of lecture "[Slide] 02. Linear Regression.pdf", we know that in order for us to get X which minimizes cost (Euclidean norm), we can perform following formula to get X.

$$X = (A^T A)^{-1} A^T b$$

Now, let's implement this equation using Python. Since Python has a wonderful library named Numpy, we will use Numpy to calculate matrix operations. The code that was used can be found in "**Task2.ipynb**" that was attached with this document.

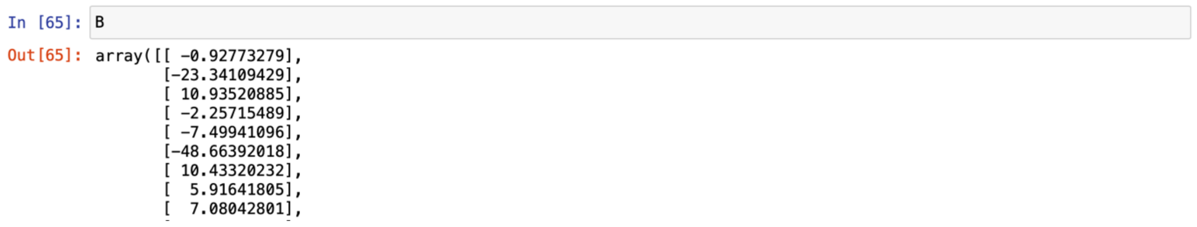
Process

1. Read csv file using Pandas.
2. From the csv, generate Pandas.DataFrame for storing csv data.
3. Generate matrix A and b from the data from csv file.
4. Apply normal equation and retrieve X just like magic.



```
In [27]: A
Out[27]: array([[ 2.47386094, -1.57285121,  1.        ],
                [ 5.87673012, -2.4241968 ,  1.        ],
                [ 0.1619504 ,  0.40243062,  1.        ],
                ...,
                [ 3.23973241, -1.79992567,  1.        ],
                [ 2.68776934, -1.63944178,  1.        ],
                [ 0.4565788 ,  0.67570615,  1.        ]])
```

Matrix A will look like this. In each row, first column will store squared value of original x value, second column will store original x value, and third column will store 1.



```
In [65]: B
Out[65]: array([[ -0.92773279],
                [-23.34109429],
                [ 10.93520885],
                [ -2.25715489],
                [ -7.49941096],
                [-48.66392018],
                [ 10.43320232],
                [  5.91641805],
                [  7.08042801],
```

Matrix B will look like this. It has dimension of 1000×1 and will just store values of original y.

Normal Equation

Now it is time to get matrix X using normal equation. As a reminder, the equation looks like below:

$$X = (A^T A)^{-1} A^T b$$

So, in order for us to process "inverse", "transpose", and matrix multiplication we will be using "np.linalg.inv", "np.transpose", "np.matmul" respectively. Thus, the implementation will look like this:

```
X = np.matmul(np.transpose(A), A)  # Calculate Transpose A mul A
X = np.linalg.inv(X)  # Calculate inverse of (Transpose A mul A)
X = np.matmul(X, np.transpose(A))  # Calculate inverse mul Transpose A
X = np.matmul(X, B)  # Calculate mul B
```

This whole process is implemented as a single function named "normal_equation".

```
def normal_equation(A, B):
```

This function has arguments of following:

```
@param A: The vector A (Which contains 1000 x 3 size)
@param B: The vector B (Which literally just stores 1000 x 1 size which contains y values.)
```

After running the function, this will return a matrix that represents values of X.

Results

```
In [53]: X = normal_equation(A, B)

In [54]: X
Out[54]: array([[-5.82074893],
                [ 1.05708197],
                [12.70482255]])
```

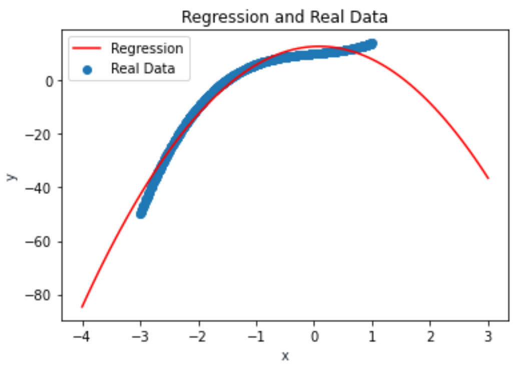
Thus, the result will be as it follows:

a	b	c
-5.82074893	1.05708197	12.70482255

Meaning following equation

$$h(x) = -5.82074893 * x^2 + 1.05708197 * x + 12.70482255$$

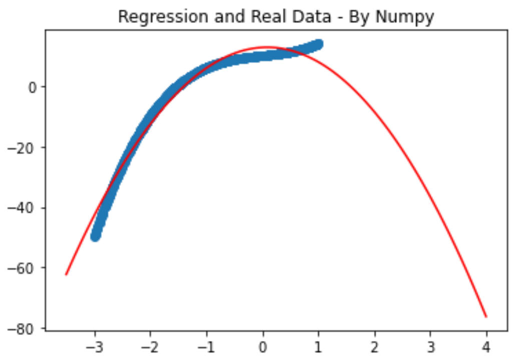
Now, let's plot the graph.



Wow, without any iterations like gradient descent had to, using normal equation made it possible for us to retrieve values of a, b and c. Also, as you can see, the graph is fit to the original data compared to linear regression!

Comparison

Just for fun and for comparing results with professional libraries, let's see how Numpy solved the same problem. Numpy provides us a function named numpy.polyfit for fitting polynomial equation into data. For more information about numpy.polyfit, please check link https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html for more information. When Numpy solved the problem with numpy.polyfit, the graph was represented like below.

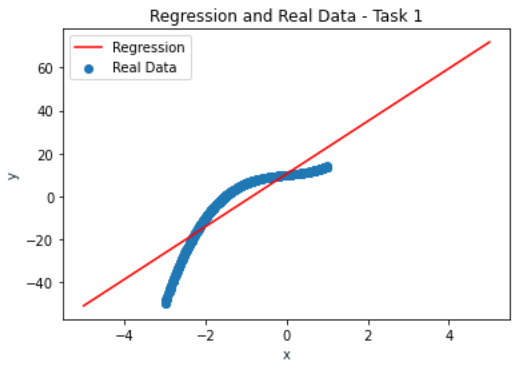


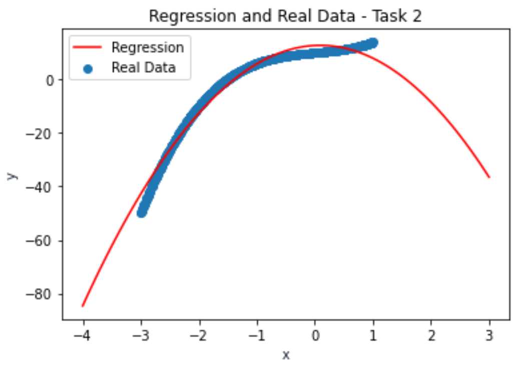
Also, let's compare values of a, b and c.

Method	a	b	c
Normal Equation (We made!)	-5.82074893	1.05708197	12.70482255
Library	-5.821	1.057	12.7

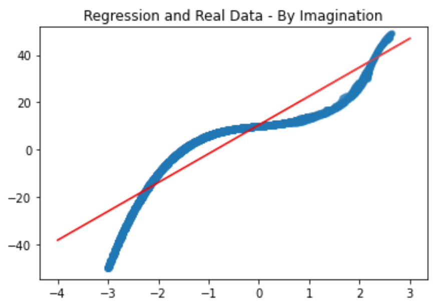
Those two values are almost the same! Meaning that our model got values almost correct.

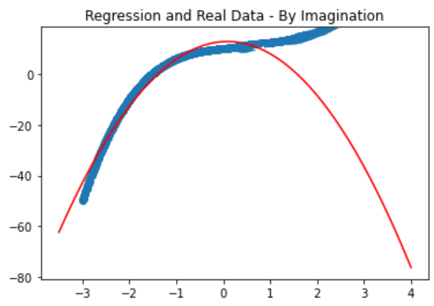
Conclusion





Comparing Task 1 and Task 2, Task 2 was better when it comes to costs **<u>within</u>** the data provided. By **<u>within</u>**, I mean that if there is more data found besides the data that those models were trained with, Task 2 has potential of becoming a "bad" model compared to Task 2. Let's suppose that we found more data and the data looked like





We actually did not have any data after the provided data in csv. So, we do not know how the future data might had been. Since Task 2's model uses a quadratic function which hits the top of the graph and goes down, there is potential that Task 2 might be a bad model compared to Task 1 if the data was actually like the graph above.

Therefore, **<u>within</u>** the data that was provided by the csv file that we got, Task 2 was better fitted. However, we cannot just tell that Task 2 is a better model over Task 1. Since we do not know other data.

It was a fun project implementing gradient descent with normal equation. Normal equation was interesting since it just calculated a, b and c like magic (without any iterations or any kinds of external operations) with just help of inverse, transpose and matrix multiplication.

Since normal equation was like magic, I looked into normal equation's time complexity and if it was really that efficient. For $X = (A^T A)^{-1} A^T b$ we use following functions from Numpy:

- numpy.matmul
- numpy.linag.inv
- numpy.transpose

As I searched further information, Numpy uses methods with highly-optimized algorithms to implement matrix operations. However, in typical cases, matrix operations are quite costly.

Operation	Size	Complexity
Matrix Multiplication	$n \times m$ with $m \times p$	$O(nmp)$
Matrix Transpose	$n \times m$	$O(nm)$
Matrix Inversion	$n \times n$	$O(n^3)$

However, in gradient descent, it is fixed to $O(knd)$ where:

- k is the number of iterations
- n is the number of samples
- d is number of features

It is difficult to say gradient descent is better than normal equation, vice versa. However, with big data set with large number of samples with lots of features, it is well known fact that normal equation is not ideal. Since matrix operation is costly.

To sum up, **within** the data that was provided, normal equation with quadratic function was good. However, we cannot generalize that normal equation with quadratic function is always better than gradient descent with linear function.