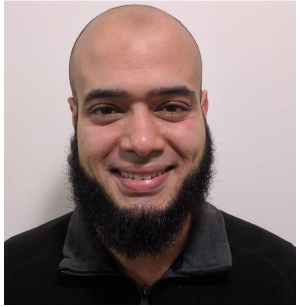
Machine Learning

Linear Regression Homework 2

Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / MSc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



Tips for this assignment

- Overall, this is an easy assignment
- However, it consists of many requests, and that can be overwhelming!
- My advice:
    - First, see all the requirements
    - Start with the first task, because this is the CORE of the assignment. Everything else depends on it

Use **ArgumentParser**

- It is important to have a good coding that allows easy experimentation
- Use Python's ArgumentParser

```
parser = argparse.ArgumentParser(description='Linear Regression Homework')

parser.add_argument('--dataset', type=str, default='dataset_200x4_regression.csv')

parser.add_argument('--preprocessing', type=int, default=1,       # p4
                    help='0 for no processing, '
                         '1 for min/max scaling and '
                         '2 for standrizing')

parser.add_argument('--choice', type=int, default=2,
                    help='0 for linear verification, '           # p0
                         '1 for training with all features, '     # p1 / p3 / p7
                         '2 for training with the best feature, ' # p5
                         '3 for normal equations, '               # p6
                         '4 for sikit)')

# Use below to explore for p2
parser.add_argument('--step_size', type=float, default=0.01, help='Learning rate (default: 0.01)')
parser.add_argument('--precision', type=float, default=0.0001, help='Requested precision (default: 0.0001)')
parser.add_argument('--max_iter', type=int, default=10000, help='number of epochs to train (default: 1000)')

args = parser.parse_args()
```

P1: Implement LR using Gradient Descent

- We already coded the GD algorithm for function minimization
- Now, we just add a **few lines of code** for linear regression
- We just need to code the **derivative of MSE** over N examples
  - Try to do that in a Pythonic way (vectorization - *2-4 lines of code*)
- **def** gradient_descent_linear_regression(X, t, step_size = 0.01, precision = 0.0001, max_iter = 1000):
  - X is an N * D dimensions input vector (first column with ones for the intercept)

$$cost(W) = \frac{1}{2N} \sum_{n=1}^{N} (y(X^n, W) - t^n)^2$$

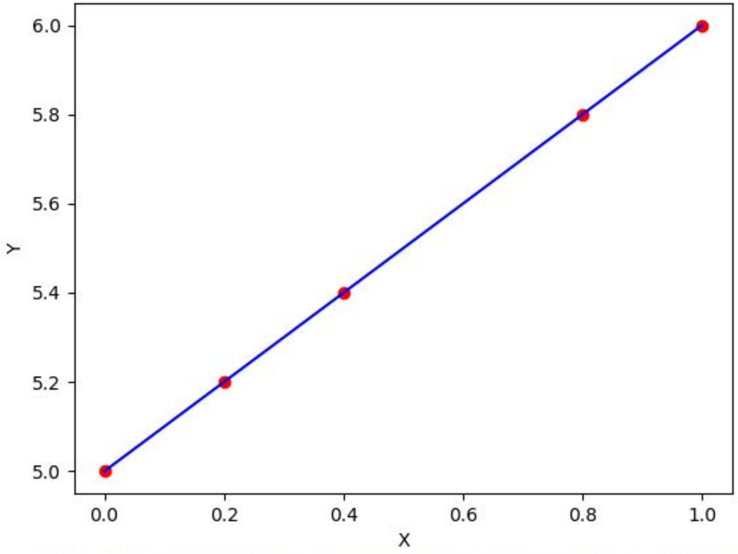
$$\frac{\partial cost(W)}{\partial W_j} = \frac{1}{N} \sum_{n=1}^{N} (y(X^n, W) - t^n) * X_j^n$$

Early Verifications

- You should always think how to verify your code
  - I take baby steps to compile, build and verify
- The easiest way is to start with a simple data for a **line (no noise)**
  - E.g. a 45 degree line
  - You should be able to perfectly fit it
- Consider the following data:
  - x = np.array([0, 0.2, 0.4, 0.8, 1.0])
  - t = 5 + x
  - Clearly the solution for such data is: slope = 1 and intercept = 5
- We already implemented its cost function and derivative

Visual Verification

- Draw your **found** line. It should perfectly fit the data
- I used the following configuration:
  - step_size=0.1, precision = 0.00001, max_iter=1000
- Start your weights from a random position
  - cur_weights = **np.random.rand**(features)
- Note: try other learning rates (step size)
  - E.g. 0.01 0.001



Dataset

- Now, you verified the algorithm. Time to work on the requested dataset
- Attached a simple dataset: **dataset_200x4_regression.csv**
- It consists of 200 x 4 matrix
  - The first 3 columns represents **3 features** (x1, x2, x3)
  - The last column represents **our target** that we want to regress it
- The data is numeric and very clean
  - The real datasets has non-numeric values (e.g. city name)
  - The real datasets are not clean (e.g. missing values)
- Start with using MinMax Scaler for the data
- We will use the 200 examples for training

More Verifications

- Use the following information for the verification step
  - Process all the data using **min-max scaler**
  - Start from **initial state: [1, 1, 1, 1]**
    - 3 input features ⇒ learn 4 parameters (includes intercept)
  - Use **step_size = 0.1 and max_iter = 3**

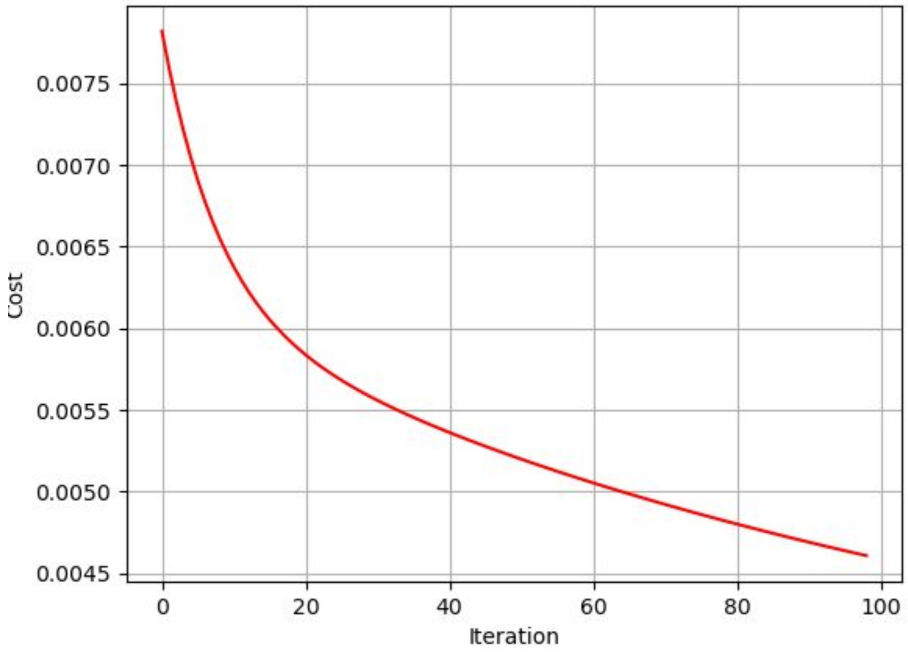
```
weights: [1. 1. 1. 1.]
    cost: 1.5127403213757413 - gradient: [1.69732353 0.87760002 0.88852874 0.50521329]
weights: [0.83026767 0.91223997 0.9111471  0.9494787 ]
    cost: 1.0800099415415019 - gradient: [1.42904069 0.73688578 0.75333759 0.42991259]
weights: [0.6873636  0.8385514  0.83581334 0.9064874 ]
    cost: 0.7724470467820816 - gradient: [1.20289468 0.61829309 0.63933402 0.36641601]
```

P2: Optimizing the **hyper**parameters

- Let's fix the maximum number of iterations to 10,000
- Consider the following 2 hyperparameter sets:
  - Step sizes: {0.1, 0.01, 0.001, 0.0001, 0.00001, 0.0000001}
  - Precision: {0.01, 0.001, 0.0001, 0.00001}
  - For each combination, run 3 times of your program to try different **initial** startings
- For all the possible setups, investigate:
  - What is the minimum error you achieved among them?
  - How many iterations before the program stops?
  - What is your best combination?

P3: Visualization

- Visualize the iterations versus the cost function of each state your program passed with
- How can such a graph help us investigate the performance?

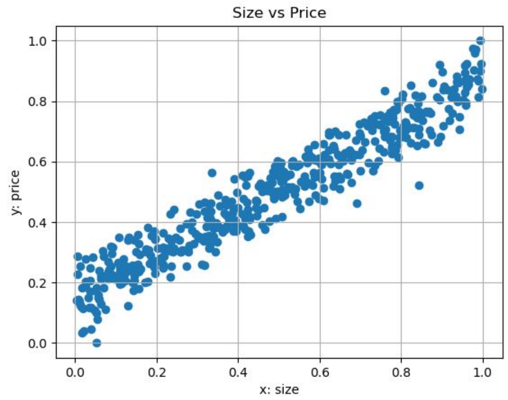


P4: Exploration

- What will happen if we did not preprocess the data (e.g. scaling)?
- What about standardizing the data (e.g. scikit StandardScaler)?

P5: Data Linearity Investigation

- So far, we just used the whole data as they are
- When we started linear regression, we assumed data seems coming from a line
- Is that true?
- Draw 3 plots, each one is a **feature versus the target**
    - Which feature seems really came from a line?
- Retrain your **model only using this feature** (mx+c case)



P6: Normal Equations

- The Normal equation is a closed-form solution (OLS) for the linear regression problem
- Implement function: `def normal_equations_solution(X, y):` $\Theta = (X^T X)^{-1} X^T y$
- Given the data, the function computes the parameter
- Compute the results **with and without scaling**
- Compute the actual predictions
- Any thoughts?
- There are 2 ways to implement this expression. Can you find the most efficient way?

My results

- Let me share some results for the sake of verification
- Let's fix: step_size = 0.01, precision = 0.0001, max_iter = 1000
- Scaling + Gradient Descent + 3 features
  - *Number of iterations ended at **2021** - with cost 0.0033275517970179813 -*
  - *Optimal weights [0.15874904 0.54368119 0.10489429 0.22107806]*
  - *The actual cost function on the original domain is **116**.5076769533999*
- Scaling + Normal Equations + 3 features
  - *Optimal weights: [0.12060381 0.6338432 0.20894728 0.00150253]*
  - *The actual cost function on the original domain is **118**.11925272653018*

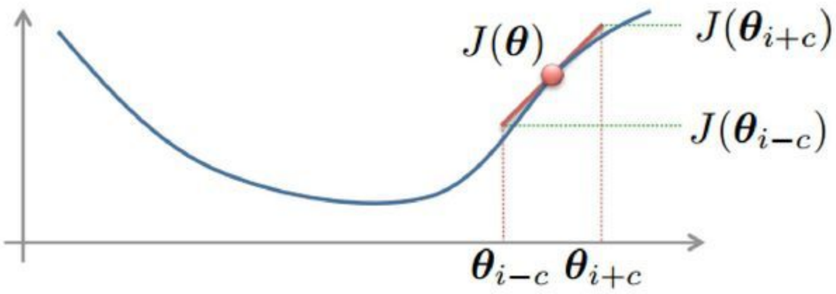
P7: Gradient Check

- *This question comes in **interviews***
- Luckily, in this homework I give you output samples so that you can **verify your derivative code**
- In practice, you might compute the derivative wrongly. Or compute it correctly and code it wrongly
- Luckily, in programming, there's a way to check if we coded the derivative correctly or not - using a simple formula for the gradient!
- Tip: this feature is implemented in the SciPy library

Gradient Check

- From calculus, we know we can compute the **gradient numerically** from the cost function (let's call it J)
- What does this formula say?
- Consider a segment of length 2C centred around the current weights

$$\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta}) \approx \frac{J(\boldsymbol{\theta}_{i+c}) - J(\boldsymbol{\theta}_{i-c})}{2c}$$



Img src

Gradient Check

- It says if you want to compute the derivative of your function with respect to the i-th variable, you should consider the following:
  - Use a small epsilon variable C, e.g. = 1e-4
  - Compute the cost function at 2 locations: weigh[i] + c and weight[i] - c (other weights fixed)
  - Compute the above formula. This is the derivative estimate for the i-th weight
  - Verify this gradient is the same as your gradient from the partial derivatives
  - Do this for every variable
- Enhance your gradient code with a **gradient check**
- Question: why not simply employ this gradient alternative in practice?

Code Organization

- In real projects, we don’t just write a lot of code in the same file
- We need to partition the code to files, packages, classes and functions
- Think how you will structure your code
- Here is mine

Name	Size	Type
data	1 item	Folder
assignment_driver.py	3.1 kB	Text
data_helper.py	856 bytes	Text
gradient_descent_linear_reg.py	1.7 kB	Text
normal_eq.py	158 bytes	Text
regression_utilites.py	990 bytes	Text

P8: Simple Linear Regression Derivation

- We mentioned the 2D line fitting y=mx+c has a simple formula. In this task, you will start from the cost function and derive the formula
- Let our line formula be: $\hat{Y}_i = a + Bx_i$ (here, a is like c, and B like m)
- Our cost function is $$S = \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$
- To find the minimum:

  compute derivative = 0

  Then extract the target variable

Simple Linear Regression Derivation : Steps

- First compute the partial derivative dS/da
- You should end with the formula:
  - The bar means the mean(Y)

$$a = \bar{Y} - B\bar{x}$$

- Then compute dS/dB
  - Substitute a with the formula we derived
  - Rearrange to end with the formula

$$B = \frac{\sum_{i=1}^{n} (x_i Y_i - \bar{Y} x_i)}{\sum_{i=1}^{n} (x_i^2 - \bar{x} x_i)}$$

P9: Time Complexity

- Assume that:
  - k is number of iterations
  - n is number of training examples
  - d is number of features
- Analyze the **time complexity** for
  - Batch Gradient Descent Algorithm
  - Normal Equation
  - Compare them
- Recall: Time complexity for matrix multiplication
  - [A x B] * [B x C] is O(ABC). If A=B=C, then is $O(A^3)$

“Acquire knowledge and impart it to the people.”

“Seek knowledge from the Cradle to the Grave.”