CMSC 197 Problem Set
Implementing Linear Regression with Gradient Descent as Optimization Function

Deadline: September 24, 2024 @ 11:59 pm

At the end of the activity, the student is expected to:

- Understand and apply gradient descent as optimization for multiple linear regression
- Compare the results of linear regression with sklearn and ordinary least squares approach compared to using the gradient descent approach
- Determine the impact of the learning rates and the number of iterations on the accuracy of predictions

Instructions for Submission:

- Submit a github link of the jupyter notebook of your code solution (make sure you have executed each code cell before saving)
- Submit a pdf report that addresses the questions (at the end of this page). If you have additional analysis and experiments, you could include that in the report. Filename should be hw3-surname.pdf. Examples: hw3-ambita.pdf. hw3-delaCruz.pdf.

  You can use the sklearn package for

  - preprocessing the data (standardization and dividing into train and test sets)
  - checking MSE and r2 values.
  - For comparison of your implementation (from scratch) and the OLS results since sklearn uses least squares for linear regression

  But you can't use it for

  - Fitting the model

  Packages you're allowed to use: numpy, pandas, matplotlib.
  Once you understand the breadth of gradient descent, then your machine learning journey will be easier!

Main goal: Predict Sales from TV, radio, and newspaper advertising expenditures.
Dataset: Advertising.csv
Features/Predictors X: TV, Radio, and Newspaper
Response Y: Sales

Implement multiple linear regression with gradient descent as optimization. Essentially, the general form of multiple linear regression is

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n = \theta^\top x$$

where $x_1, x_2, \ldots, x_n$ represent he model predictors and $\theta_1, \theta_2, \ldots, \theta_n$ represent the weights associated to each predictor. We can represent the model in matrix form using $\theta^\top x$.

The weights/parameters $\theta_1, \theta_2, \ldots, \theta_n$ are unknown and generally, we can estimate these using the ordinary least squares approach or gradient descent. While least squares approach is a closed analytical form, gradient descent is an iterative method that determines the optimal parameters.

The main idea is to start with some random initial weights for $\theta_1, \theta_2, \ldots, \theta_n$ and iteratively nudge/change it until we hopefully achieve the minimum cost $J(\theta)$. In essence, a cost function is a measure of how wrong the model is in terms of its ability to estimate the relationship between x and y.

In gradient descent, the common cost function is mean squared error $J(\theta)$ given by:

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

Where $h_\theta\left(x^{(i)}\right) = \widehat{y^{(i)}}$ represent the predictions made by the model.

But how do we actually change the values of parameters θ in order to minimize $\boldsymbol{J(\theta_0,\ \theta_1)}$? We do this by computing the **value of the first partial derivative** of the cost function $J(\theta_0,\ \theta_1)$ with respect to a parameter θ_j.

The equation below shows the formula on updating all the parameters/weights in the model.

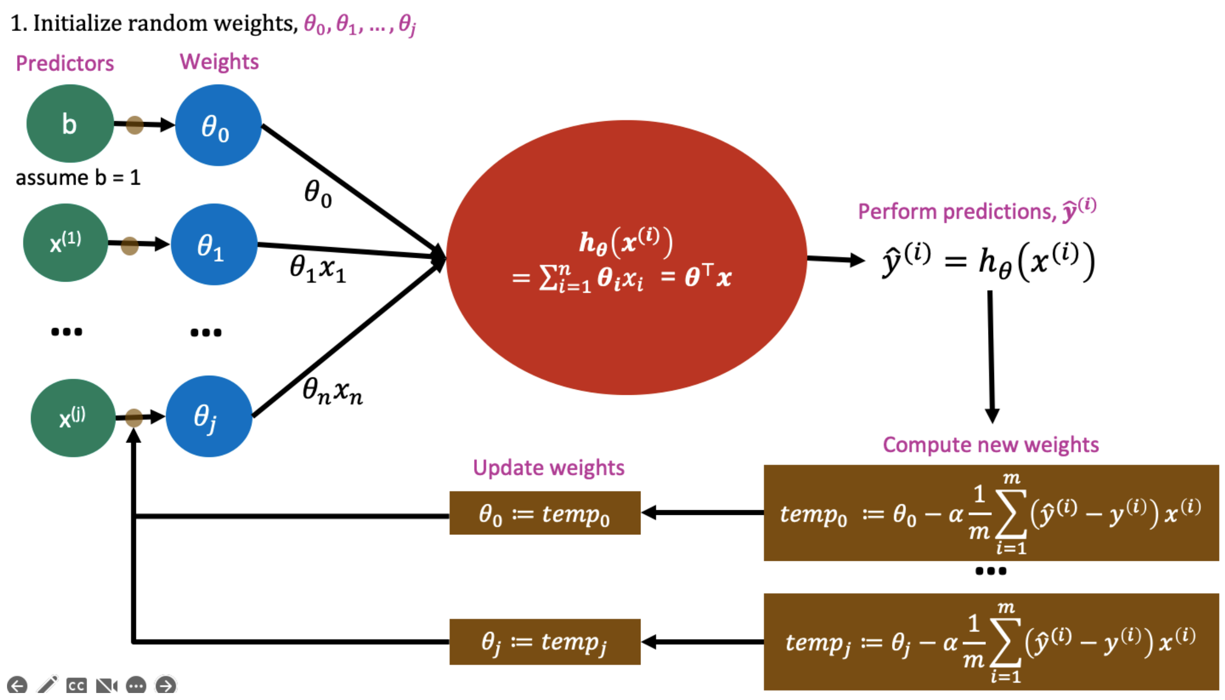
Repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)} \right) x_j^{(i)}$$

}
Simultaneously update θ_j, for $j = 0,\ 1,\ \cdots, n$

θ_j – represent the parameter/weight associated with each predictor
α – represent the learning rate (or dictates how large the step/change) in the theta
$\hat{y}^{(i)}$ – predicted y
$y^{(i)}$ – actual y
$x_j^{(i)}$ – data points
m – number of samples

The process is as follows:


1. Initialize random weights, $\theta_0, \theta_1, \ldots, \theta_j$
Predictors
Weights
b
assume b = 1
$x^{(1)}$
$x^{(j)}$
θ_0
θ_1
θ_j
θ_0
$\theta_1 x_1$
$\theta_n x_n$
$h_\theta(x^{(i)}) = \sum_{i=1}^{n} \theta_i x_i = \theta^\top x$
Perform predictions, $\hat{y}^{(i)}$
$\hat{y}^{(i)} = h_\theta(x^{(i)})$
Compute new weights
$temp_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)}) x^{(i)}$
$temp_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)}) x^{(i)}$
Update weights
$\theta_0 := temp_0$
$\theta_j := temp_j$

Instructions for the application part:

1. Load `Advertising.csv` dataset using pandas
2. Standardize each column of the dataset

   For each predictor x_j, for $j = 0, 1, \cdots, n$, compute for the standardized values:

$$x_{scaled} = \frac{x_j - mean(x_j)}{standard\ deviation(x_j)}$$

   This ensures that the data is centered around zero and that standard deviation is always 1. Standardizing your data will help your model converge faster and better. Alternatively, check sklearn's `preprocessing.scale` function.
3. Note that

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1K} \\ 1 & x_{21} & \cdots & x_{2K} \\ 1 & x_{31} & \cdots & x_{3K} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & \cdots & x_{NK} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_K \end{bmatrix}$$

   So you must add an extra column composing of **all ones** to X.
4. Divide the dataset into training and testing, with 0.85 and 0.25 ratio, respectively.

5. Fit the model on the training set. Essentially, you have to optimize the model using the training set, and not including the test set. (Instruction 5 elaborated below)
6. Predict the quantitative response y on the train set. Evaluate performance. You can use the MSE cost function defined for the gradient descent.
7. Predict the quantitative response y on the test set. Evaluate performance. Similarly, you can use the MSE cost function defined for the gradient descent.
8. Note: Since the data is standardized, you might be surprised that the predictions differ from the original data. In order to revert back a standardized data into the original form, we simply have to equate the previous equation:
$$x_{scaled} * standard\ deviation\ (x_j) + mean\ (x_j) = x_j$$
9. Observe the cost results and analyse.

Required Functions (Elaborated version of item 5 above):

1. `initialize_weights`: returns a vector `init_w` composing of 4 uniformly distributed numbers between 0 and 1. This serves as the initial weights θ_j, for $j = 0,\ 1,\ 2, 3$. You can set a random seed so you can objectively assess if your model is working correctly. *Seed* function is used to save the state of a *random* function, so that it can generate same *random* numbers on multiple executions of the code. Alternatively, you can just set `init_w = np.array([0.0, 0.0, 0.0, 0.0])`. Note that the first weight refers to the weight of the bias and the rest represents the weights of the predictors.
2. `predict`: returns a vector of the predicted values $\hat{y}^{(i)}$
3. `compute_cost`**:** returns a scalar value that tells us how accurate the model is. This is represented by $J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$
4. `compute_gradient`: returns a matrix *w* that represents the partial derivative of the cost function $J(\theta_0,\ \theta_1)$ with respect to each parameter θ_j. $\alpha\frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)x_j^{(i)}$. We have 3 predictors and 1 bias, so w should have a shape of 4x1.
5. `update_weights`: returns a 4x1 matrix that contains the updated weights. $\theta_j := \theta_j - \alpha w x_j^{(i)}$ where the matrix *w* contains the gradients for a specific iteration.
6. `grad_descent`**:** returns 2 matrices: one matrix for the weights, and one matrix for the cost values per iteration. *grad_descent* calls the functions 1-5 until the number of iterations is reached.
7. `plot_costs`**:** plot the costs as a function of iteration.
8. Predict y for train set and calculate the cost.
9. Predict y for test set and calculate the cost

Questions:

Note that the weights are generated randomly and the results may vary across execution. While you have a function for generating random weights, assume that the initial weights are all 0s.

```
init_w = np.array([0.0, 0.0, 0.0, 0.0])
```

1. What are the optimal weights found by your implemented gradient descent? Plug it into the linear model:
$$h_\theta(x) = \theta_0 + \theta_1 TV + \theta_2 Radio + \theta_3 Newspaper$$
What are your interpretations regarding the formed linear model?
2. Provide a scatter plot of the $\left(\widehat{y^{(i)}}\right)$ $and\ y^{(i)}$ for both the train and test set. Is there a trend? Provide an r2 score (also available in sklearn).
3. What happens to the error, r2, and cost as the number of iterations increase? Show your data and proof. You can alternatively plot your result data for visualization and check until 50000 iterations or more (actually).
4. Once you determine the optimal number of iterations, check the effect on the cost and error as you change the learning rate. The common learning rates in machine learning include 0.1, 0.01, 0.001, 0.0001, 0.2 but you have the option to include others. Visualize the cost function (vs the optimal number of iterations) of each learning rate in ONLY ONE PLOT. Provide your analysis.
5. Is there a relationship on the learning rate and the number of iterations?
6. Compare the results with the results of ordinary least squares function.