Assignment # 1

Date	@September 6, 2022
Course	Machine Learning
Finished?	☑
Tags	Lab writing

```
# ------------------------ #
# << HOMEWORK GROUP :: 3 >> #
# ------------------------ #
# <<   GROUP MEMBERS  >>    #
# ------------------------ #
# 1. DHRUV PATEL
# 2. BHAVANASI APURVA
# 3. SAMUEL HELMRATH
# 4. ABRAR AHMED MOHAMMED
# ------------------------ #
```


Problem 1 : Design Algorithm to play `Tic-Tac-Toe`

- The basic idea behind the learning system is that the system should be able to improve its performance (P) with respect to a set of tasks (T) by learning from training experience (E). In this case,
    - Task (T) : Playing Tic-Tac-Toe
    - Performance (P) : Percentage of Games won against Humans
    - Experience (E) : Feedback by games history generated from games played against a clone
- To do so, we can create a linear function that evaluates board state at a given time.
- The linear function we came up with uses 6 features and 7 weights (including the bias term). These features are extracted from the board state b. When playing the game the learner gets all of the legal moves and applies the evaluation function (target/cost function) to the new states to learn which move gets the highest rating. After the evaluation, the learner takes an action resulting in subsequent board states. This process can be repeated until the game is won, lost or drawn.
- In the context of this problem, we will define a row as 3 subsequent squares - the rows, the columns, and the diagonals. Tic-Tac-Toe problem has 3 rows, 3 columns and 2 diagonals. We will refer to all of them as rows for extracting the features. The 6 features are as below :
    - $x1(b)$ = Number of instances where there are 2 X's in a row with an open subsequent square
    - $x2(b)$ = Number of instances where there are 2 O's in a row with an open subsequent square
    - $x3(b)$ = Number of instances where there is an X in a completely open row
    - $x4(b)$ = Number of instances where there is an O in a completely open row
    - $x5(b)$ = Number of instances of 3 X's in a row (value of 1 would signify the game has ended and player with X has won)
    - $x6(b)$ = Number of instances of 3 O's in a row (value of 1 would signify the game has ended and player with O has won)
- We then can give the learner some randomly initialized weights ($\omega0, \omega1, \omega2, ..., \omega6$) or give equal importance to each feature for the evaluation. Here, we would like to keep all the weights (including the bias $\omega0$) equal to 1. This would signify equal importance to all features.
- The Learned Function $\hat{V}(b)$ then can be represented as :

$$\hat{V}(b) = \omega0 + \omega1 * x1 + \omega2 * x2 + \omega3 * x3 + \omega4 * x4 + \omega5 * x5 + \omega6 * x6$$

- Now, that we have our Learned Function, we can generate training data (training examples) by playing several games. The obtained training examples will then be used to refine the weights that would give the best estimation for the given board state (b). Training examples can be obtained by the following conditions.

- If game is ended and we won, $V_{train}(b)$ = 100
- If game is ended and we lost, $V_{train}(b)$ = -100
- If game is ended and is drawn, $V_{train}(b)$ = 0
- If game is not ended, $V_{train}(b) = V_{estimate}(successor(b))$

- Using this generated training data, we update the weights using the Least Mean Squares (LMS) method, which is given in brief as below :

```
# LMS Weight Update Rule
1. Do repeatedly until end condition
    1.1 Select a training example (b) at random
        1.1.1 Compute Error
            error(b) = V_train(b) - V_hat(b)
        1.1.2 For each board feature x_i, update weight w_i
            w_i <---- w_i + lambda * x_i * error(b)
```

where, λ is the learning rate which dictates how fast or slow the weights are updated and is constant (e.g. 0.4).

Problem Set-up for programming

- Before we can start training and have a learned model, we first need to setup the aforementioned algorithm into code.
- First we need to represent the board state. We can set rules such as : 0 represents an unfilled square, 1 represents an X and 2 represents an O for the given square.
- After setting up the board, we need to write a `function` that would update the board state after each player turn. There also needs to be another `function` that would extract above mentioned features from x_1 to x_6 after every board state update.
- We would then check for the game end condition that would fall between 4 categories : Game Won, Lost or Drawn and Game is yet to end.
- If the game has not ended then we take update our game history (adding values to the training dataset) and take a move and repeat this process till end.
- After playing the game for several hundred times we should have an adequate amount of data so that we can train a model on it.
- The main bottleneck would be to get every possible board state in a given number of play runs. We might miss some data points in the training which the game state never encounters. This may lead to imperfect training of the model.

Problem 2 : A straw man learning algorithm for `Tic-Tac-Toe`

- According to the Straw man algorithm we need to construct a dataset to get informed about the baseline performance of the algorithm
- In order to win a game, there must be three ‘X’s in a row or three ‘O’s in a row (Row, Columns or Diagonals). If the main player is ‘X’ then there must be three ‘X’s in a row and vice versa.

- Construction of the Dataset:
  - There are exactly 8 possible ways for X to win.
  - The attributes in this dataset are :
    - Top-left-square : x_1
    - Top-middle-square : x_2

- Top-right-square : x_3
- Middle-left-square : x_4
- Middle-middle-square : x_5
- Middle-right-square : x_6
- Bottom-left-square : x_7
- Bottom-middle-square : x_8
- Bottom-right-square : x_9

- The possible values for all these attributes are X, O, and 'void' (blank space).
- The class output that we can achieve in this game are 'WIN' or 'LOSE' and as discussed in Problem 1 we can also assign real-value by assigning +100 for WIN condition and -100 for LOSE condition.
- As possible number of values for each cell is 3. Then the total number of values in the dataset would be $3^9 - 1$ (i.e. subtracting the blank board state).
- The training table would look as follows :

Cell1	Cell2	Cell3	Cell4	Cell5	Cell6	Cell7	Cell8	Cell9	Result
x	o	b	x	o	x	x	b	o	WIN
x	o	b	x	o	o	x	x	b	WIN
x	o	b	x	o	o	x	b	x	WIN
x	o	b	x	o	b	x	x	o	WIN
x	o	b	x	o	b	x	b	b	WIN
x	o	b	x	b	x	x	o	o	WIN
x	o	b	x	b	o	x	x	o	WIN
x	o	b	x	b	o	x	o	x	WIN
x	o	b	x	b	o	x	b	b	WIN
x	o	b	x	b	b	x	o	b	WIN
x	x	o	o	o	x	x	o	x	LOSE
x	o	x	x	x	o	o	x	o	LOSE
x	o	x	x	o	x	o	x	o	LOSE
x	o	x	x	o	o	o	x	x	LOSE
x	o	x	o	x	x	o	x	o	LOSE
x	o	x	o	o	x	x	x	o	LOSE
x	o	o	o	x	x	x	x	o	LOSE
o	x	x	x	x	o	o	o	x	LOSE
o	x	x	x	o	o	x	o	x	LOSE
o	x	x	x	o	o	o	x	x	LOSE
o	x	o	x	x	o	x	o	x	LOSE
o	x	o	x	o	x	x	o	x	LOSE
o	x	o	o	x	x	x	o	x	LOSE
o	o	x	x	x	o	o	x	x	LOSE

- From the dataset above we can observe that based on all cell values we are making decisions about whether the game is lost or won.

  1) **As a function of the size of the training set , that is, for example how accurate**

  would expect the algorithm to be given the training set (table) size

  - If the algorithm is given less data ,then when a new instance is given to the model,according to the straw man algorithm, if win occurs more from the sample data set we have chosen, then win will be predicted even though it is not necessarily true.
  - This can lead to Poor performance of the model and Baseline performance also varies based on the data we have taken into account.

  2) **As a function of the size of a data point (i.e., number of features)**

  - If the algorithm considers fewer variables then only a few cell positions are taken into account and the model prediction is damaged. In this game we have taken every cell into account in the data set to predict its result.
  - Consider the data set below as an example :

cell1	cell2	cell3	cell4	cell5	cell6	cell7	cell8	cell9	result
b	b	x	o	b	x	b	o	x	WIN
b	b	x	o	b	x	o	b	x	WIN
b	b	x	o	o	o	b	x	x	LOSE
b	b	x	o	o	o	x	b	x	LOSE

- If only the cells c1-c4 are considered and others are ignored then the result we get the model is ambiguous.
- So if attributes which contribute to deciding the game result , we need to consider the appropriate number of attributes, using correlation, information share measures.

Problem 3 : Linear Regression Programming using Normal Equation

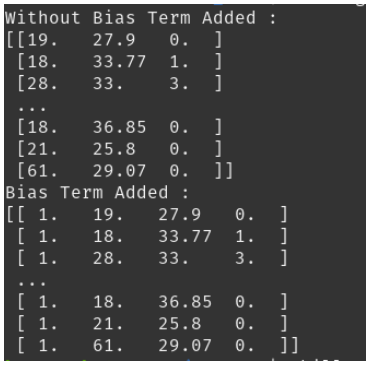
- Here, a dataset is provided on which we have to apply normal equation to implement linear regression. This will include pre-processing of the dataset, splitting the data into various train-test split, applying the normal equation to obtain various θ values (hypothesis/weights).
- First of all, let us explain the folder structure to run the program successfully.

```
---> Root Folder/              # Folder where the python file is saved
  |
  | ---> assignment_1.py       # This is the python file to execute
  | ---> insurance.csv         # data file should be in the same folder as pythong file
  |
    |
    | ---> Plots/              # sub-directory inside root folder to save plots
```

- The give dataset contains several real-valued attributes as well as some categorical attributes. Since, we are only interested in doing linear regression, we can drop the categorical attributes and just work with the real-valued features. `age`, `bmi`, and `number_of_children` are the those three features.
- The normal equation for linear regression is given as

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

- Here, θ is hypothesis parameters (weights). X is the input feature vector/array. While, y is the output target vector/array.
- It is important to note that, the input vector X has to have a bias column at column index 0 for the correct results. Hence, we add a bias column with every instance equal to 1. However, while solving the normal equation this value will change. This is shown in the Image below.



- After we get all of the required input arrays, we then can split the input-output vector into various training set size ranging from 20% to 80% with a 10% increment step.
- After implementing the normal equation, we would get the θ array which would include the bias term and the weights for each attribute. Size of the θ vector is [4,1].

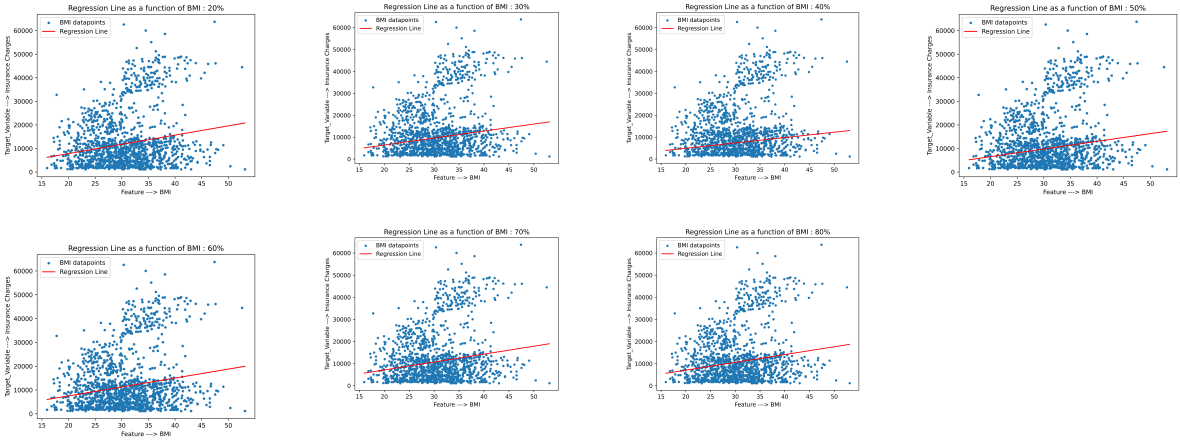
- For each training set size, we would get different θ values, modeling power as well as generalization power. All of the results are tabulated in the below table.

Training Set Size	Test Set Size	θ_0 : Bias Term	θ_1 : Weight for Age	θ_2 : Weight for BMI	θ_3 : Weight for Children	Modeling Power	Generalization Power
20 %	80 %	- 7683.24	198.28	391.81	899.01	119989862.88	132501707.71
30 %	70 %	- 7269.04	230.53	319.50	907.93	103333130.49	141958457.34
40 %	60 %	- 4887.37	234.55	244.93	725.12	126507588.30	131984395.57
50 %	50 %	- 7760.21	252.48	325.87	696.07	133727128.86	124729418.55
60 %	40 %	- 8557.70	242.08	375.56	680.59	120975246.14	142323413.66
70 %	30 %	- 8010.23	243.32	357.65	612.98	127311914.77	132988794.94
80 %	20 %	- 6966.94	223.94	352.66	527.86	128596025.12	130705925.43

- As we can see from the table above, the linear hypothesis model gives negative intercept for every train set, and the highest weight is given to the `number_of_children` attribute. However, just looking at this raw data is somewhat misleading. To better understand the results, let us take a look at the plots for
  1. Regression line as a function of BMI
  2. Regression line as a function of AGE
  3. Regression line as a function of number of children
  4. Modeling Loss and Generalization Loss Plot as a function of train set size

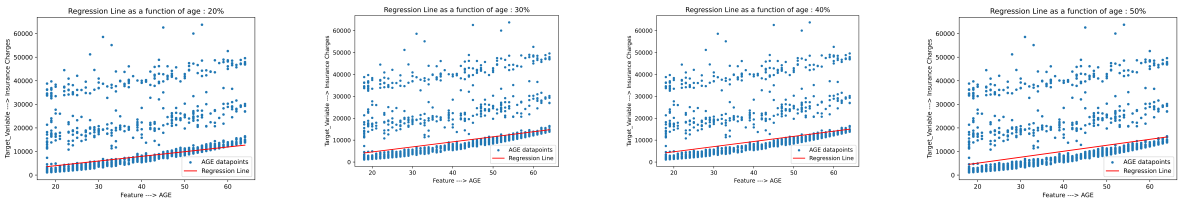
Plots

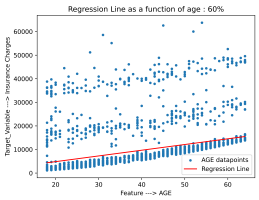
1. Regression plot as a function of `BMI` plots

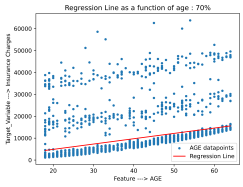


- From the above plots, we can see how the regression line changes the slope when more or less training data is provided. The dense data-points between the target value of 0 to 15,000 keeps the regression line more or less close to the bottom of the plot. However, when we provide 80% of the dataset, the scattered target value above 30,000 does increase the slope of the regression line as it is trying to reduce the MSE Loss.

2. Regression plot as a function of `AGE` plots

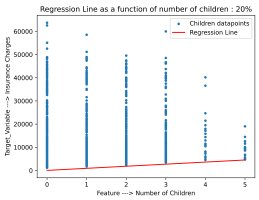


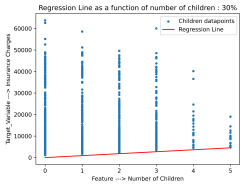


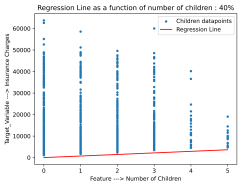


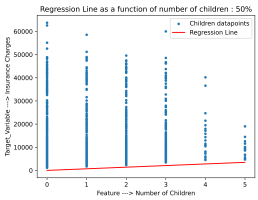
- From the above plots, we can see how the regression line changes the slope when more or less training data is provided. The dense data-points between the target value of 0 to 10,000 keeps the regression line more or less close to the bottom of the plot. The scattered data points for all age values and above $20,000 target values are so scattered, the regression line stays more or less similar to each other with very minimal slope change.

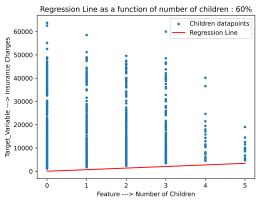
3. Regression plot as a function of `number_of_children`

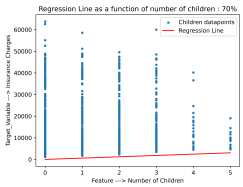


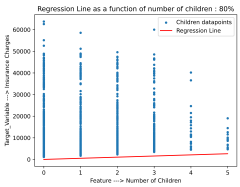






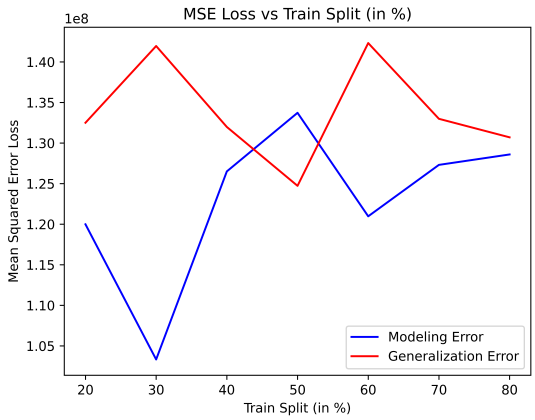






- From the plots above, we can see that the feature `number_of_children` has high negative co-relation with the target value. This is the reason why the regression line fails to capture the data-points information and has relatively small slope. After the initial analysis, it would be better to not use this attribute in order to make a robust regression model.

4. Loss Plot



- In the above plot, the Modeling Error is shown in the `blue` line and the Generalization error is shown in `red` line. The error is taken as MSE (Mean Squared Error) and calculated for each train set size (from 20% to 80%). As we can see from the plot, the modeling

error is least for the train split of 30%, while the Generalization error reaches its minima at 50% train split. Also, the generalization error is more than Modeling Error except one instance (i.e. 50% split).

- We can come to the conclusion that because of the less correlative nature of the `number_of_children` attribute, as we provide more data to the model; it worsens in performance.