

Assignment On  
**“Linear Regression in Machine Learning”**

Course Title: Data Mining and Data Warehousing Lab

Course Code: CSEL-4108

**Submitted To,**  
Dr. Md. Manowarul Islam  
Associate Professor  
Department of CSE  
Jagannath University, Dhaka

**Submitted By,**  
Momtaj Hossain Mow  
(B180305025)  
&  
Md. Abu Yousuf Siam  
(B180305012)



**Department of Computer Science & Engineering**  
**Jagannath University, Dhaka**  
**Date of Submission: 16/05/2023**

# Linear Regression in Python

## What is Linear Regression?

Linear regression is a basic and commonly used type of predictive analysis. The overall idea of regression is to examine two things:

- (1) Does a set of predictor variables do a good job in predicting an outcome (dependent) variable?
- (2) Which variables in particular are significant predictors of the outcome variable, and in what way do they—indicated by the magnitude and sign of the beta estimates—impact the outcome variable?

These regression estimates are used to explain the relationship between one dependent variable and one or more independent variables. The simplest form of the regression equation with one dependent and one independent variable is defined by the formula:

$$y = mx + c$$

Where,

y = estimated dependent variable score,

m = regression coefficient,

x = score on the independent variable,

and c = constant (the cutting point in y axis)

*Three major uses for regression analysis are:*

- (1) determining the strength of predictors
- (2) forecasting an effect and
- (3) trend forecasting.

First, the regression might be used to identify the strength of the effect that the independent variable(s) have on a dependent variable. Typical questions are what is the strength of relationship between dose and effect, sales and marketing spending, or age and income.

Second, it can be used to forecast effects or impact of changes. That is, the regression analysis helps us to understand how much the dependent variable changes with a change in one or more independent variables. A typical question is, “how much additional sales income do I get for each additional \$1000 spent on marketing?”

Third, regression analysis predicts trends and future values. The regression analysis can be used to get point estimates. A typical question is, “what will the price of gold be in 6 months?”

**Types of Linear Regression:**

- a. *Simple linear regression*  
1 dependent variable (interval or ratio), 1 independent variable (interval or ratio or dichotomous)
- b. *Multiple linear regression*  
1 dependent variable (interval or ratio) , 2+ independent variables (interval or ratio or dichotomous)
- c. *Logistic regression*  
1 dependent variable (dichotomous), 2+ independent variable(s) (interval or ratio or dichotomous)
- d. *Ordinal regression*  
1 dependent variable (ordinal), 1+ independent variable(s) (nominal or dichotomous)
- e. *Multinomial regression*  
1 dependent variable (nominal), 1+ independent variable(s) (interval or ratio or dichotomous)
- f. *Discriminant analysis*  
1 dependent variable (nominal), 1+ independent variable(s) (interval or ratio)

The kind of data type that can have any intermediate value (or any level of granularity) is known as continuous data. The kind of data type that cannot be partitioned or defined more granularly is known as discrete data.

Regression is performed on continuous data while classification is performed on discrete data. Regression can be anything like predicting someone's age, the price of a house or value of any variable. Classification includes predicting what class something belongs to (such as whether a tumor is benign or malignant).

## Exploratory Data Analysis:

Let's start with exploratory data analysis. We want to get to know our data first - this includes loading it in, visualizing features, exploring their relationships and making hypotheses based on your observations. The dataset is a CSV (comma-separated values) file, which contains the hours studied and the scores obtained based on those hours. We'll load the data into a DataFrame using Pandas:

```
from google.colab import files
import io
import pandas as pd
```

Let's read the CSV file and package it into a DataFrame:

```
from google.colab import files
import io
import pandas as pd
uploaded = files.upload()
df = pd.read_csv(io.BytesIO(uploaded['student_scores.csv']))
```

Choose Files student\_scores.csv

- student\_scores.csv(text/csv) - 214 bytes, last modified: 5/9/2023 - 100% done

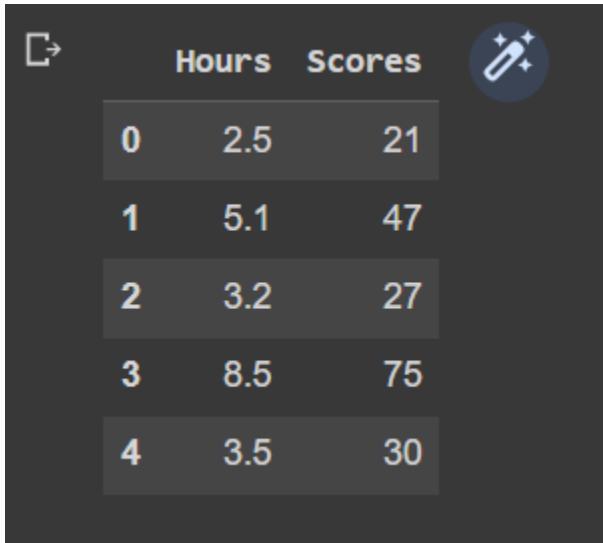
Saving student\_scores.csv to student\_scores (1).csv

```
# This is the code for local file uploading in anaconda
(spyder) but in colab, reading csv file is different, which is
attached in the previous screenshot.
# Substitute the path_to_file content by the path to your
student_scores.csv file
path_to_file = 'home/projects/datasets/student_scores.csv'
df = pd.read_csv(path_to_file)
```

Once the data is loaded in, let's take a quick peek at the first 5 values using the `head()` method:

```
#it will show the first 5 values
df.head()
```

This results in:



|   | Hours | Scores |
|---|-------|--------|
| 0 | 2.5   | 21     |
| 1 | 5.1   | 47     |
| 2 | 3.2   | 27     |
| 3 | 8.5   | 75     |
| 4 | 3.5   | 30     |

We can also check the shape of our dataset via the **shape** property:

```
#let's check the shape of given dataset in (row, column) format
df.shape
```

(25, 2)

Knowing the shape of your data is generally pretty crucial to being able to both analyze it and build models around it.

We have 25 rows and 2 columns - that's 25 entries containing a pair of an hour and a score.

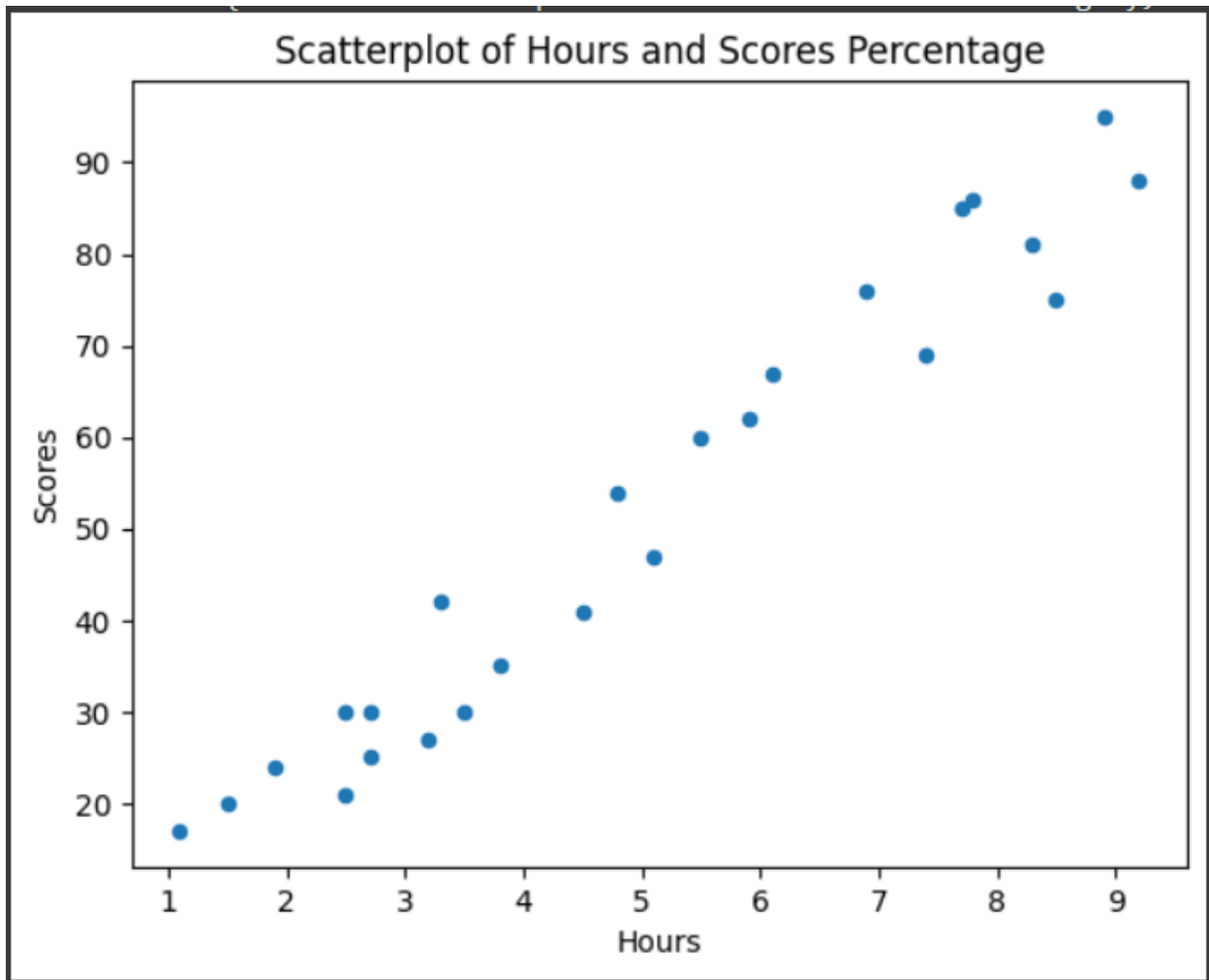
A great way to explore relationships between variables is through Scatterplots. We'll plot the hours on the X-axis and scores on the Y-axis, and for each pair, a marker will be positioned based on their values:

```
[25] df.plot.scatter(x='Hours', y='Scores', title='Scatterplot of Hours and Scores Percentage')
```

The code in text:

```
df.plot.scatter(x='Hours', y='Scores', title='Scatter Plot of
hours and scores percentages');
```

This results in:



As the hours increase, so do the scores. We say that there's a positive linear correlation between the Hours and Scores variables.

The **corr() method** calculates and displays the correlations between numerical variables in a DataFrame:

```
print(df.corr())
```

Output:

|        | Hours    | Scores   |
|--------|----------|----------|
| Hours  | 1.000000 | 0.976191 |
| Scores | 0.976191 | 1.000000 |

In this table, Hours and Hours have a 1.0 (100%) correlation, just as Scores have a 100% correlation to Scores, naturally. Any variable will have a 1:1 mapping with itself! However, the correlation between Scores and Hours is 0.97. Anything above 0.8 is considered to be a strong positive correlation.

Pandas also ships with a great helper method for statistical summaries, and we can `describe()` the dataset to get an idea of the mean, maximum, minimum, etc. values of our columns:

```
#we can use the 'describe()' method to describe the mean, maximum, minimum etc. values of the columns of a dataset.
print(df.describe())
```

|       | Hours     | Scores    |
|-------|-----------|-----------|
| count | 25.000000 | 25.000000 |
| mean  | 5.012000  | 51.480000 |
| std   | 2.525094  | 25.286887 |
| min   | 1.100000  | 17.000000 |
| 25%   | 2.700000  | 30.000000 |
| 50%   | 4.800000  | 47.000000 |
| 75%   | 7.400000  | 75.000000 |
| max   | 9.200000  | 95.000000 |

## Linear Regression with Python's Scikit-learn Library

With the theory under our belts - let's get to implementing a Linear Regression algorithm with Python and the Scikit-Learn library! We'll start with a simpler linear regression and then expand onto multiple linear regression with a new dataset.

### Data Preprocessing

In the previous section, we have already imported Pandas, loaded our file into a DataFrame and plotted a graph to see if there was an indication of a linear relationship. Now, we can divide our data in two arrays - one for the dependent feature and one for the independent, or target feature. Since we want to predict the score percentage depending on the hours studied, our y will be the "Score" column and our X will be the "Hours" column.

To separate the target and features, we can attribute the dataframe column values to our y and X variables:

```
#To separate the target and features, we can attribute the dataframe column values to our y and X variables:
#df['Column_Name'] returns a pandas Series
y = df['Scores'].values.reshape(-1, 1)
x = df['Hours'].values.reshape(-1, 1)
print(df['Hours'].values)
print(df['Hours'].values.shape)
```

```
[2.5 5.1 3.2 8.5 3.5 1.5 9.2 5.5 8.3 2.7 7.7 5.9 4.5 3.3 1.1 8.9 2.5 1.9
 6.1 7.4 2.7 4.8 3.8 6.9 7.8]
(25,)
```

Some libraries can work on a Series just as they would on a NumPy array, but not all libraries have this awareness. In some cases, you'll want to extract the underlying NumPy array that describes your data. This is easily done via the values field of the Series.

We could already feed our x and y data directly to our linear regression model, but if we use all of our data at once, how can we know if our results are any good? Just like in learning, what we will do is use a part of the data to train our model and another part of it, to test it.

This is easily achieved through the helper `train_test_split()` method, which accepts our X and y arrays (also works on DataFrames and splits a single DataFrame into training and testing sets), and a test\_size. The test\_size is the percentage of the overall data we'll be using for testing:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.2)
```

The method randomly takes samples respecting the percentage we've defined, but respects the X-y pairs, lest the sampling would totally mix up the relationship. Some common train-test splits are 80/20 and 70/30.

Since the sampling process is inherently random, we will always have different results when running the method. To be able to have the same results, or reproducible results, we can define a constant called SEED that has the value of the meaning of life (42):

```
SEED = 42
```



**Note:** The seed can be any integer, and is used as the seed for the random sampler. The seed is usually random, netting different results. However, if you set it manually, the sampler will return the same results. It's convention to use 42 as the seed as a reference to the popular novel series "The Hitchhiker's Guide to the Galaxy".

We can then pass that **SEED** to the **random\_state** parameter of our **train\_test\_split** method:

```
from sklearn.model_selection import train_test_split
SEED = 42
X_train, X_test, Y_train, Y_test = train_test_split(x, y, test_size = 0.2, random_state = SEED)
print(X_train)
print(Y_train)
```

Now, if we print our **X\_train** array - we'll find the study hours, and **Y\_train** contains the score percentages:

```
[[2.7]
 [3.3]
 [5.1]
 [3.8]
 [1.5]
 [3.2]
 [4.5]
 [8.9]
 [8.5]
 [3.5]
 [2.7]
 [1.9]
 [4.8]
 [6.1]
 [7.8]
 [5.5]
 [7.7]
 [1.1]
 [7.4]
 [9.2]]
[[25]
 [42]
 [47]
 [35]
 [20]
 [27]
 [41]
 [95]
 [75]
 [30]]
```

## Training a Linear Regression Model

We have our train and test sets ready. Scikit-Learn has a plethora of model types we can easily import and train, **LinearRegression** being one of them:

```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
```

Now, we need to fit the line to our data, we will do that by using the `.fit()` method along with our `X_train` and `y_train` data:

```
regressor.fit(X_train, y_train)
```

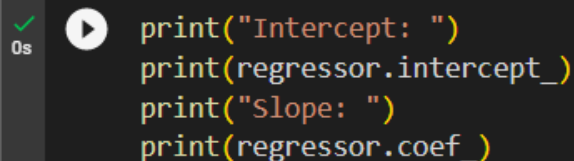
If no errors are thrown - the regressor found the best fitting line! The line is defined by our features and the intercept/slope. In fact, we can inspect the intercept and slope by printing the `regressor.intercept_` and `regressor.coef_` attributes, respectively:

```
print(regressor.intercept_)
```

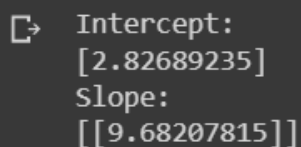
For retrieving the slope (which is also the coefficient of `x`):

```
print(regressor.coef_)
```

The result should be:

A terminal window with a dark background. On the left, there is a green checkmark icon and the text '0s'. To the right of the icon is a play button icon. The terminal contains the following code:

```
print("Intercept: ")
print(regressor.intercept_)
print("Slope: ")
print(regressor.coef_)
```

The output of the code from the previous block, displayed in the terminal. It shows the intercept and slope values.

```
Intercept:
[2.82689235]
Slope:
[[9.68207815]]
```

This can quite literally be plugged in into our formula from before :

$$\text{score} = 9.68207815 * \text{hours} + 2.82689235$$

Let's check real quick whether this aligns with our guesstimation:

hours=5

score= 9.68207815\*hours+2.82689235

score = 51.2672831

With 5 hours of study, you can expect around 51% as a score! Another way to interpret the intercept value is - if a student studies one hour more than they previously studied for an exam, they can expect to have an increase of 9.68% considering the score percentage that they had previously achieved.

**Note:** In other words, the slope value shows what happens to the dependent variable whenever there is an increase (or decrease) of one unit of the independent variable.

```

[34] #Making Predictions
def calc(slope, intercept, hours):
    return slope*hours + intercept

score = calc(regressor.coef_, regressor.intercept_, 9.5)
print(score)

[[94.80663482]]

```

However, a much handier way to predict new values using our model is to call on the **predict** function:

```

#passing 9.5 in double brackets to have a 2 dimensional array
score = regressor.predict([[9.5]])
print(score)

[[94.80663482]]

```

We can predict the scores according to  $X_{\text{test}}$  data by using **predict()** method:

```
#To make predictions on the test data, we pass the X_test values to the predict() method.  
#We can assign the results to the variable y_pred:  
y_pred = regressor.predict(X_test)
```

We can also see the how the predicted values are actual values are varying by using **squeeze()** method:

```
df_preds = pd.DataFrame({'Actual': Y_test.squeeze(), 'Predicted': y_pred.squeeze()})  
print(df_preds)
```

|   | Actual | Predicted |
|---|--------|-----------|
| 0 | 81     | 83.188141 |
| 1 | 30     | 27.032088 |
| 2 | 21     | 27.032088 |
| 3 | 76     | 69.633232 |
| 4 | 62     | 59.951153 |

## Evaluating the Model

After looking at the data, seeing a linear relationship, training and testing our model, we can understand how well it predicts by using some metrics. For regression models, three evaluation metrics are mainly used:

### 1. Mean Absolute Error (MAE):

When we subtract the predicted values from the actual values, obtaining the errors, sum the absolute values of those errors and get their mean. This metric gives a notion of the overall error for each prediction of the model, the smaller (closer to 0) the better.

$$mae = \left(\frac{1}{n}\right) \sum_{i=1}^n |Actual - Predicted|$$

### 2. Mean Squared Error (MSE):

It is similar to the MAE metric, but it squares the absolute values of the errors. Also, as with MAE, the smaller, or closer to 0, the better. The MSE value is squared so as to make large errors even larger. One thing to pay close attention to, it that it is usually a hard metric to interpret due to the size of its values and of the fact that they aren't in the same scale of the data.

$$mse = \sum_{i=1}^D (Actual - Predicted)^2$$

### 3. Root Mean Squared Error (RMSE):

Tries to solve the interpretation problem raised with the MSE by getting the square root of its final value, so as to scale it back to the same units of the data. It is easier to interpret and good when we need to display or show the actual value of the data with the error. It shows how much the data may vary.

$$rmse = \sqrt{\sum_{i=1}^D (Actual - Predicted)^2}$$

Luckily, we don't have to do any of the metrics calculations manually. The Scikit-Learn package already comes with functions that can be used to find out the values of these metrics for us. Let's find the values for these metrics using our test data. First, we will import the necessary modules for calculating the MAE and MSE errors. Respectively, the `mean_absolute_error` and `mean_squared_error`:

```
0s ✓ from sklearn.metrics import mean_absolute_error, mean_squared_error
import numpy as np
```

For the metrics calculations:

```
0s ✓ [39] mae = mean_absolute_error(Y_test, y_pred)
      mse = mean_squared_error(Y_test, y_pred)
      rmse = np.sqrt(mse)
```

We will also print the metrics results using the f string and the 2 digit precision after the comma with `:.2f`:

```
0s ✓ print(f'Mean absolute error: {mae:.2f}')
      print(f'Mean squared error: {mse:.2f}')
      print(f'Root mean squared error: {rmse:.2f}')

☞ Mean absolute error: 3.92
   Mean squared error: 18.94
   Root mean squared error: 4.35
```

## Multiple Linear Regression

Multiple linear regression is a regression model that estimates the relationship between a quantitative dependent variable and two or more independent variables using a straight line.

### Exploratory Data Analysis

First, we can import the data with pandas `read_csv()` method:

```
[41] uploaded = files.upload()
      df = pd.read_csv(io.BytesIO(uploaded['petrol_consumption.csv']))
```

Choose Files petrol\_consumption.csv

- petrol\_consumption.csv(text/csv) - 1211 bytes, last modified: 10/1/2019 - 100% done

Saving petrol\_consumption.csv to petrol\_consumption.csv

We can now take a look at the first five rows with `df.head()`:

```
df.head()
```

This results is:

|   | Petrol_tax | Average_income | Paved_Highways | Population_Driver_licence(%) | Petrol_Consumption |
|---|------------|----------------|----------------|------------------------------|--------------------|
| 0 | 9.0        | 3571           | 1976           | 0.525                        | 541                |
| 1 | 9.0        | 4092           | 1250           | 0.572                        | 524                |
| 2 | 9.0        | 3865           | 1586           | 0.580                        | 561                |
| 3 | 7.5        | 4870           | 2351           | 0.529                        | 414                |
| 4 | 8.0        | 4399           | 431            | 0.544                        | 410                |

We can see the how many rows and columns our data has with `shape`:

```
[43] df.shape
```

Which displays:

```
(48, 5)
```

There is no consensus on the size of our dataset. Let's keep exploring it and take a look at the descriptive statistics of this new data. This time, we will facilitate the

comparison of the statistics by rounding up the values to two decimals with the `round()` method, and transposing the table with the `T` property:

```
print(df.describe().round(2).T)
```

Our table is now column-wide instead of being row-wide:

```
count      mean      std      min      25%  \
Petrol_tax      48.0      7.67      0.95      5.00      7.00
Average_income  48.0  4241.83  573.62  3063.00  3739.00
Paved_Highways  48.0  5565.42  3491.51  431.00  3110.25
Population_Driver_licence(%)  48.0      0.57      0.06      0.45      0.53
Petrol_Consumption  48.0  576.77  111.89  344.00  509.50

              50%      75%      max
Petrol_tax      7.50      8.12     10.00
Average_income  4298.00  4578.75  5342.00
Paved_Highways  4735.50  7156.00 17782.00
Population_Driver_licence(%)  0.56      0.60      0.72
Petrol_Consumption  568.50  632.75   968.00
```

By looking at the min and max columns of the described table, we see that the minimum value in our data is 0.45, and the maximum value is 17,782. This means that our data range is 17,781.55 ( $17,782 - 0.45 = 17,781.55$ ), very wide - which implies our data variability is also high. Also, by comparing the values of the mean and std columns, such as 7.67 and 0.95, 4241.83 and 573.62, etc., we can see that the means are really far from the standard deviations. That implies our data is far from the mean, decentralized - which also adds to the variability.

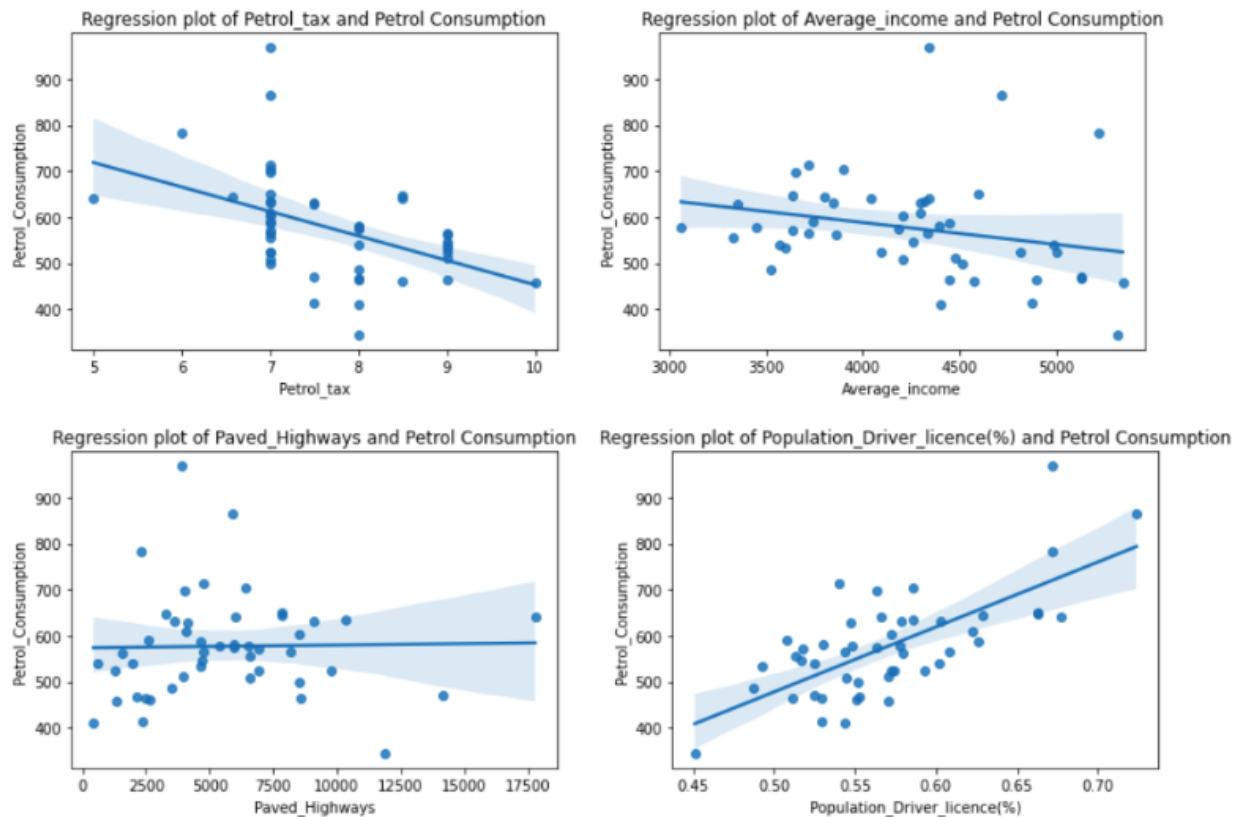
Now we will use Seaborn, an extension of Matplotlib which Pandas uses under the hood when plotting:

```
import seaborn as sns
import matplotlib.pyplot as plt
variables = ['Petrol_tax', 'Average_income', 'Paved_Highways', 'Population_Driver_licence(%)']

for var in variables:
    plt.figure()
    sns.regplot(x=var, y='Petrol_Consumption', data=df).set(title=f'Regression plot of {var} and Petrol Consumption');
```



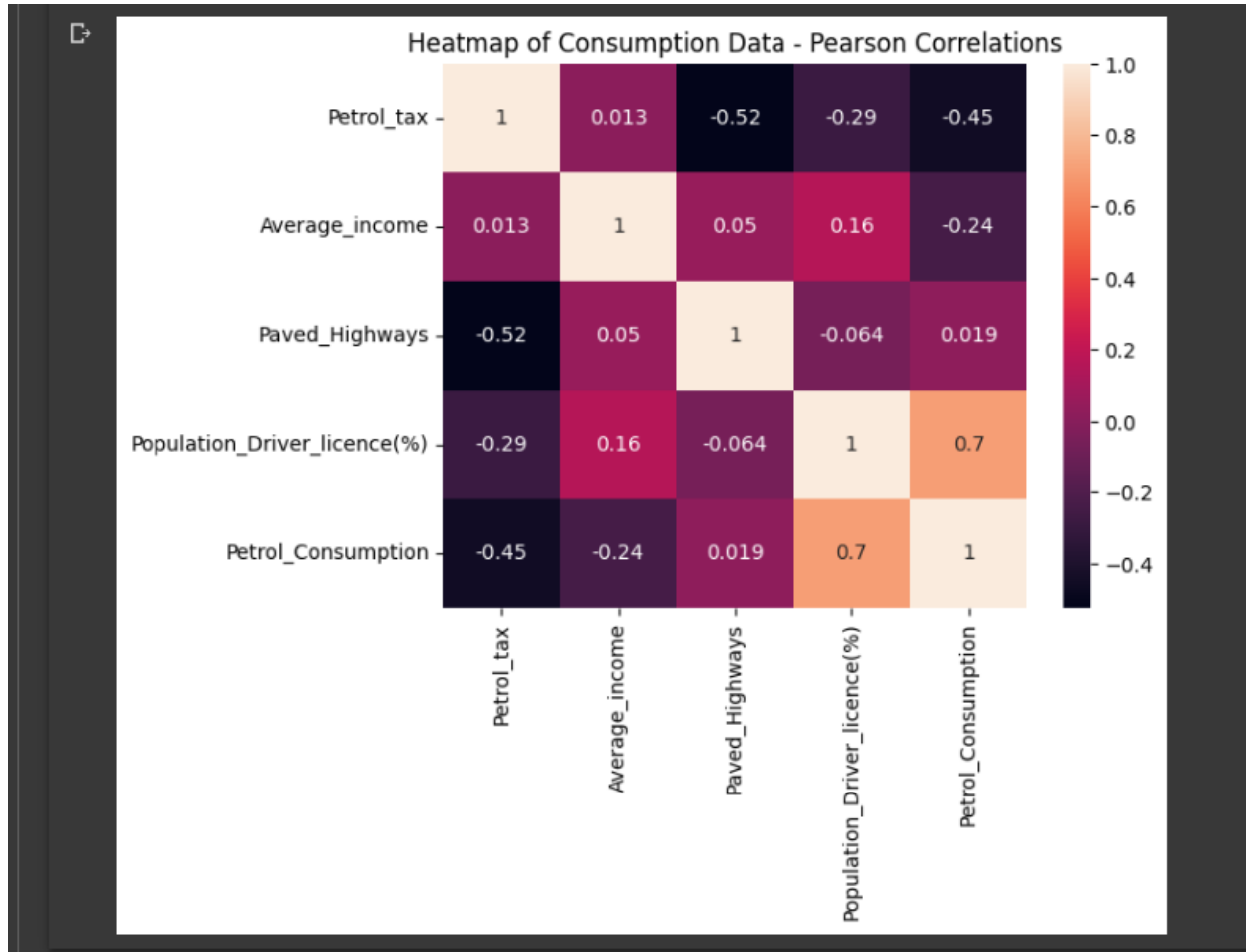
These are our four plots:



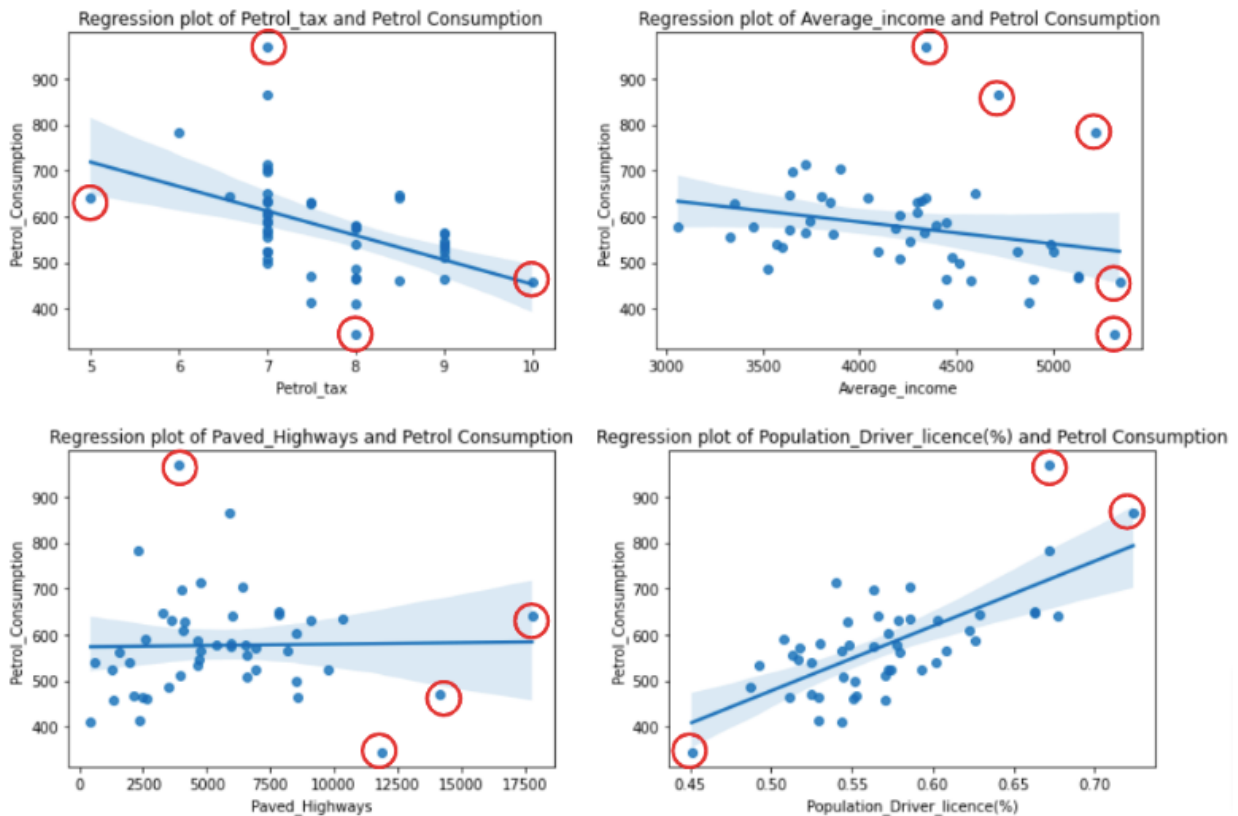
Seeing the regplots, it seems the Petrol\_tax and Average\_income have a weak negative linear relationship with Petrol\_Consumption. It also seems that the Population\_Driver\_license(%) has a strong positive linear relationship with Petrol\_Consumption, and that the Paved\_Highways variable has no relationship with Petrol\_Consumption.

We can also calculate the correlation of the new variables, this time using Seaborn's heatmap() to help us spot the strongest and weaker correlations based on warmer (reds) and cooler (blues) tones:

```
correlations = df.corr()
# annot=True displays the correlation values
sns.heatmap(correlations, annot=True).set(title='Heatmap of Consumption Data - Pearson Correlations');
```



It seems that the heatmap corroborates our previous analysis! `Petrol_tax` and `Average_income` have a weak negative linear relationship of, respectively, -0.45 and -0.24 with `Petrol_Consumption`. `Population_Driver_license(%)` has a strong positive linear relationship of 0.7 with `Petrol_Consumption`, and `Paved_Highways` correlation is of 0.019 - which indicates no relationship with `Petrol_Consumption`. Another important thing to notice in the regplots is that there are some points really far off from where most points concentrate, we were already expecting something like that after the big difference between the mean and std columns - those points might be data outliers and extreme values.



## Preparing the Data

Following what has been done with the simple linear regression, after loading and exploring the data, we can divide it into features and targets.

The main difference is that now our features have 4 columns instead of one. We can use double brackets `[[ ]]` to select them from the dataframe:

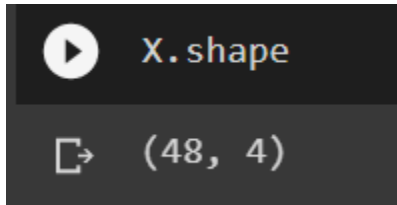
```
[50] y = df['Petrol_Consumption']
      X = df[['Average_income', 'Paved_Highways',
              'Population_Driver_licence%', 'Petrol_tax']]
```

After setting our X and y sets, we can divide our data into train and test sets. We will be using the same seed and 20% of our data for training:

```
[51] X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=SEED)
```

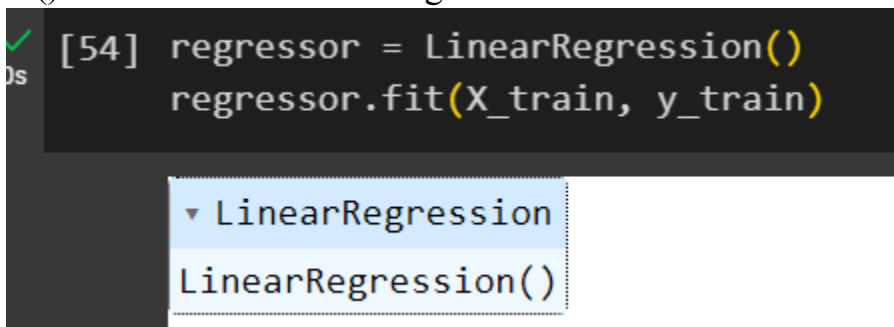
## Training the Multivariate Model

After splitting the data, we can train our multiple regression model. Notice that now there is no need to reshape our X data, once it already has more than one dimension:



```
▶ X.shape
Out: (48, 4)
```

To train our model we can execute the same code as before, and use the `fit()` method of the `LinearRegression` class:



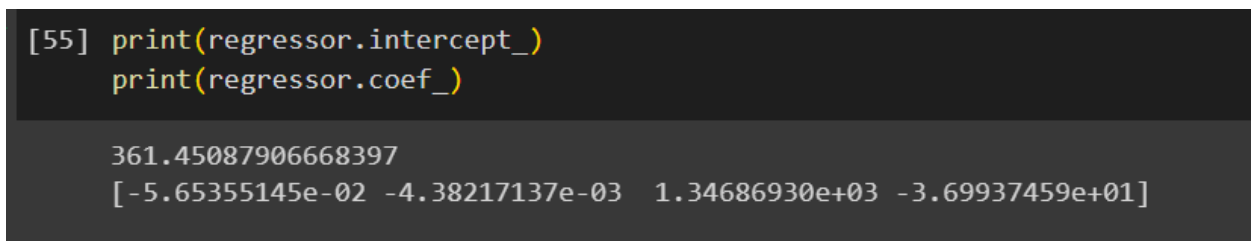
```
[54] regressor = LinearRegression()
      regressor.fit(X_train, y_train)
```

▼ LinearRegression

LinearRegression()

After fitting the model and finding our optimal solution, we can also look at the intercept:

And at the coefficients of the features:



```
[55] print(regressor.intercept_)
      print(regressor.coef_)

361.45087906668397
[-5.65355145e-02 -4.38217137e-03  1.34686930e+03 -3.69937459e+01]
```

Those four values are the coefficients for each of our features in the same order as we have them in our X data. To see a list with their names, we can use the `dataframe columns` attribute:

Considering it is a little hard to see both features and coefficients together like this, we can better organize them in a table format.

To do that, we can assign our column names to a `feature_names` variable, and our coefficients to a `model_coefficients` variable. After that, we can create a dataframe with our features as an index and our coefficients as column values called `coefficients_df`:

```

▶ feature_names = X.columns
  model_coefficients = regressor.coef_

  coefficients_df = pd.DataFrame(data = model_coefficients,
                                index = feature_names,
                                columns = ['Coefficient value'])

  print(coefficients_df)

```

|                              | Coefficient value |
|------------------------------|-------------------|
| Average_income               | -0.056536         |
| Paved_Highways               | -0.004382         |
| Population_Driver_license(%) | 1346.869298       |
| Petrol_tax                   | -36.993746        |

Here we can see that ,unit increase in the average income, there is a decrease of 0.06 dollars in gas consumption.

Similarly, for a unit increase in paved highways, there is a 0.004 decrease in miles of gas consumption; and for a unit increase in the proportion of population with a driver's license, there is an increase of 1,346 billion gallons of gas consumption.

And, lastly, for a unit increase in petrol tax, there is a decrease of 36,993 million gallons in gas consumption.

By looking at the coefficients dataframe, we can also see that, according to our model, the `Average_income` and `Paved_Highways` features are the ones that are closer to 0, which means they have had the least impact on the gas consumption. While the `Population_Driver_license(%)` and `Petrol_tax`, with the coefficients of 1,346.86 and -36.99, respectively, have the biggest impact on our target prediction. In other words, the gas consumption is mostly explained by the percentage of the population with driver's license and the petrol tax amount, surprisingly (or unsurprisingly) enough.

**Making Predictions with the Multivariate Regression Model** To understand if and how our model is making mistakes, we can predict the gas consumption using our

test data and then look at our metrics to be able to tell how well our model is behaving.

In the same way we had done for the simple regression model, let's predict with the test data:

```
✓ 0s ▶ y_pred = regressor.predict(X_test)
```

Now, that we have our test predictions, we can better compare them with the actual output values for X\_test by organizing them in a DataFrame format:

```
▶ results = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})  
print(results)
```

The output should look like this:

```
☞
```

|    | Actual | Predicted  |
|----|--------|------------|
| 27 | 631    | 606.692665 |
| 40 | 587    | 673.779442 |
| 26 | 577    | 584.991490 |
| 43 | 591    | 563.536910 |
| 24 | 460    | 519.058672 |
| 37 | 704    | 643.461003 |
| 12 | 525    | 572.897614 |
| 19 | 640    | 687.077036 |
| 4  | 410    | 547.609366 |
| 25 | 566    | 530.037630 |

## Evaluating the Multivariate Model

After exploring, training and looking at our model predictions - our final step is to evaluate the performance of our multiple linear regression. We want to understand if our predicted values are too far from our actual values. We'll do this in the same way we had previously done, by calculating the MAE, MSE and RMSE metrics. So, let's execute the following code:

```
[61] mae = mean_absolute_error(y_test, y_pred)
     mse = mean_squared_error(y_test, y_pred)
     rmse = np.sqrt(mse)

     print(f'Mean absolute error: {mae:.2f}')
     print(f'Mean squared error: {mse:.2f}')
     print(f'Root mean squared error: {rmse:.2f}')
```

The output of our metrics should be:

```
Mean absolute error: 53.47
Mean squared error: 4083.26
Root mean squared error: 63.90
```

Here RMSE is 63.90, which means that our model might get its prediction wrong by adding or subtracting 63.90 from the actual value. It would be better to have this error closer to 0, and 63.90 is a big number - this indicates that our model might not be predicting very well.

Our MAE is also distant from 0. We can see a significant difference in magnitude when comparing our previous simple regression where we had a better result.

Let's consider another matrix  $R^2$ .  $R^2$  quantifies how much of the variance of the dependent variable is being explained by the model.

$$R^2 = 1 - \frac{\sum (Actual - Predicted)^2}{\sum (Actual - Actual\ Mean)^2}$$

The  $R^2$  metric varies from 0% to 100%. The closer to 100%, the better. If the  $R^2$  value is negative, it means it doesn't explain the target at all.

Lets calculate R2 in Python:

```

actual_minus_predicted = sum((y_test - y_pred)**2)
actual_minus_actual_mean = sum((y_test - y_test.mean())**2)
r2 = 1 - actual_minus_predicted/actual_minus_actual_mean
print('R²:', r2)

```

➞ R²: 0.39136640014305457

R2 also comes implemented by default into the score method of Scikit-Learn's linear regressor class. We can calculate it like this:

```

regressor.score(X_test, y_test)

```

➞ 0.39136640014305457

So far, it seems that our current model explains only 39% of our test data which is not a good result, it means it leaves 61% of the test data unexplained.

Let's also understand how much our model explains of our train data:

```

[64] regressor.score(X_train, y_train)

```

0.7068781342155135

We have found an issue with our model. It explains 70% of the train data, but only 39% of our test data, which is more important to get right than our train data. It is fitting the train data really well, and not being able to fit the test data - which means, we have an overfitted multiple linear regression model.

There are many factors that may have contributed to this, a few of them could be:

1. Need for more data: we have only one year worth of data (and only 48 rows), which isn't that much, whereas having multiple years of data could have helped improve the prediction results quite a bit.



2. Overcome overfitting: we can use a cross validation that will fit our model to different shuffled samples of our dataset to try to end overfitting.
3. Assumptions that don't hold: we have made the assumption that the data had a linear relationship, but that might not be the case.  
Visualizing the data using boxplots, understanding the data distribution, treating the outliers, and normalizing it may help with that.
4. Poor features: we might need other or more features that have stronger relationships with values we are trying to predict.

## References:

- [1] What is Linear Regression? - Statistics Solutions  
<https://www.statisticssolutions.com/free-resources/directory-of-statistical-analyses/what-is-linear-regression/>
- [2] Google Colab File  
[https://colab.research.google.com/drive/1KY\\_VVwzSq\\_9vWUqPomMcibwM-vdv3IAP?usp=sharing](https://colab.research.google.com/drive/1KY_VVwzSq_9vWUqPomMcibwM-vdv3IAP?usp=sharing)