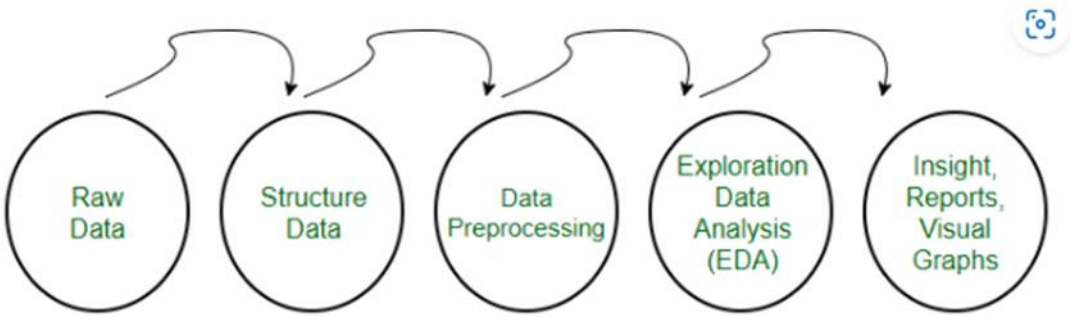
WEEK -03 : DATA PREPROCESSING AND REGRESSION

Data pre-processing is a basic requirement of any good machine learning model. Pre-processing the data implies using the data which is easily readable by the machine learning model. In this week, we are going to discuss the basics of data pre-processing and how to make the data suitable for machine learning models.



Data Preprocessing

What is data pre-processing?

Data pre-processing is the process of preparing the raw data and making it suitable for machine learning models. Data pre-processing includes data cleaning for making the data ready to be given to machine learning model.

After data cleaning, data pre-processing requires the data to be transformed into a format that is understandable to the machine learning model.

Why is data pre-processing required?

Data pre-processing is mainly required for the following:

- **Accurate data:**For making the data readable for machine learning model, it needs to be accurate with no missing value, redundant or duplicate values.
- **Trusted data:**The updated data should be as accurate or trusted as possible.
- **Understandable data:**The data updated needs to be interpreted correctly.

All in all, data pre-processing is important for the machine learning model to learn from such data which is correct in order to lead the model to the right predictions/outcomes.

Examples of data preprocessing for different data set types with Python

Since data comes in various formats, let us discuss how different data types can be converted into a format that the ML model can read accurately. Let us see how to feed correct features from datasets with:

- Missing values
- Outliers
- Overfitting
- Data with no numerical values
- Different date formats

▪ *Missing values*

Missing values are a common problem while dealing with data! The values can be missed because of various reasons such as human errors, mechanical errors, etc.

Data cleansing is an important step before you even begin the algorithmic trading process, which begins with historical data analysis for making the prediction model as accurate as possible.

Based on this prediction model you create the trading strategy. Hence, leaving missed values in the data set can wreak havoc by giving faulty predictive results that can lead to erroneous strategy creation and further the results can not be great to state the obvious.

There are three techniques to solve the missing values' problem in order to find out the most accurate features, and they are:

Dropping
Numerical imputation
Categorical imputation

Dropping

Dropping is the most common method to take care of the missed values. Those rows in the data set or the entire columns with missed values are dropped in order to avoid errors to occur in data nalysis.

There are some machines that are programmed to automatically drop the rows or columns that include missed values resulting in a reduced training size. Hence, the dropping can lead to a decrease in the model performance.

A simple solution for the problem of a decreased training size due to the dropping of values is to use imputation. We will discuss the interesting imputation methods further. In case of dropping, you can define a threshold to the machine.

For instance, the threshold can be anything. It can be 50%, 60% or 70% of the data. Let us take 60% in our example, which means that 60% of data with missing out values will be accepted by the model/algorithm as the training data set, but the features with more than 60% missing values will be dropped.

For dropping the values, following Python codes are used:

```
#Dropping columns in the data higher than 60% threshold
data = data[data.columns[data.isnull().mean() < threshold]]
#Dropping rows in the data higher than 60% threshold
data = data.loc[data.isnull().mean(axis=1) < threshold]
```

By using the above Python codes, the missed values will be dropped and the machine learning model will learn on the rest of the data.

Numerical imputation

The word imputation implies replacing the missing values with such a value that makes sense. And, numerical imputation is done in the data with numbers.

For instance, if there is a tabular data set with the number of stocks, commodities and derivatives traded in a month as the columns, it is better to replace the missed value with a "0" than leaving them as it is.

With numerical imputation, the data size is preserved and hence, predictive models like linear regression can work better to predict in the most accurate manner.

A linear regression model can not work with missing values in the data set since it is biased toward the missed values and considers them "good estimates". Also, the missed values can be replaced with the median of the columns since median values are not sensitive to outliers unlike averages of columns.

Let us see the Python codes for numerical imputation, which are as follows:

```
#For filling all the missed values as 0
data = data.fillna(0)
#For replacing missed values with median of columns
data = data.fillna(data.median())
```

Categorical imputation

This technique of imputation is nothing but replacing the missed values in the data with the one which occurs the maximum number of times in the column. But, in case there is no such value that occurs frequently or dominates the other values, then it is best to fill the same as "NAN".

The following Python code can be used here:

```
#Categorical imputation
data['column_name'].fillna(data['column_name'].value_counts().idxmax(), inplace=True)
```

- *Outliers*

An outlier differs significantly from other values and is too distanced from the mean of the values. Such values that are considered outliers are usually due to some systematic errors or flaws.

Let us see the following Python codes for identifying and removing outliers with standard deviation:

```
#For identifying the outliers with the standard deviation method
outliers = [x for x in data if x < lower or x > upper]
print('Identified outliers: %d' % len(outliers))
#Remove outliers
outliers_removed = [x for x in data if x >= lower and x <= upper]
print('Non-outlier observations: %d' % len(outliers_removed))
```

In the codes above, “lower” and “upper” signify the upper and lower limit in the data set.

- *Overfitting*

In both machine learning and statistics, overfitting occurs when the model fits the data too well or simply put when the model is too complex.

Overfitting model learns the detail and noise in the training data to such an extent that it negatively impacts the performance of the model on new data/test data.

The overfitting problem can be solved by decreasing the number of features/inputs or by increasing the number of training examples to make the machine learning algorithms more generalised.

The most common solution is regularisation in an overfitting case. Binning is the technique that helps with the regularisation of the data which also makes you lose some data every time you regularise it.

For instance, in the case of numerical binning, the data can be as follows:

Stock value	Bin
100-250	Lowest
251-400	Mid
401-500	High

Here is the Python code for binning:

```
data['bin'] = pd.cut(data['value'], bins=[100,250,400,500], labels=["Lowest", "Mid", "High"])
```

Your output should look something like this:

	Value	Bin
0	102	Low
1	300	Mid
2	107	Low
3	470	High

- *Data with no numerical values*

In the case of the data set with no numerical values, it becomes impossible for the machine learning model to learn the information.

The machine learning model can only handle numerical values and thus, it is best to spread the values in the columns with assigned binary numbers “0” or “1”. This technique is known as one-hot encoding.

In this type of technique, the grouped columns already exist. For instance, below I have mentioned a grouped column:

Infected	Covid variants
2	Delta
4	Lambda
5	Omicron
6	Lambda
4	Delta
3	Omicron
5	Omicron
4	Lambda
2	Delta

Now, the above-grouped data can be encoded with the binary numbers ”0” and “1” with one hot encoding technique. This technique subtly converts the categorical data into a numerical format in the following manner:

Infected	Delta	Lambda	Omicron
2	1	0	0
4	0	1	0
5	0	0	1
6	0	1	0
4	1	0	0
3	0	0	1
5	0	0	1
4	0	1	0
2	1	0	0

Hence, it results in better handling of grouped data by converting the same into encoded data for the machine learning model to grasp the encoded (which is numerical) information quickly.

Problem with the approach

Going further, in case there are more than three categories in a data set that is to be used for feeding the machine learning model, the one-hot encoding technique will create as many columns. Let us say, there are 2000 categories, then this technique will create 2000 columns and it will be a lot of information to feed to the model.

Solution:
To solve this problem, while using this technique, we can apply the target encoding technique which implies calculating the "mean" of each predictor category and using the same mean for all other rows with the same category under the predictor column. This will convert the categorical column into the numeric column and that is our main aim.
Let us understand this with the same example as above but this time we will use the "mean" of the values under the same category in all the rows. Let us see how.
In Python, we can use the following code:

```
#Convert data into numerical values with mean
Infected = [2, 4, 5, 6, 4, 3]
Predictor = ['Delta', 'Lambda', 'Omicron', 'Lambda', 'Delta', 'Omicron']
Infected_df = pd.DataFrame(data={'Infected':Infected, 'Predictor':Predictor})
means = Infected_df.groupby('Predictor')['Infected'].mean()
Infected_df['Predictor_encoded'] = Infected_df['predictor'].map(means)
Infected_df
```

Output:

Infected	Predictor	Predictor_encoded
2	Delta	3
4	Lambda	5
5	Omicron	4
6	Lambda	5
4	Delta	3
3	Omicron	4

In the output above, the Predictor column depicts the Covid variants and the Predictor_encoded column depicts the "mean" of the same category of Covid variants which makes 2+4/2 = 3 as the mean value for Delta, 4+6/2 = 5 as the mean value for Lambda and so on.
Hence, the machine learning model will be able to feed the main feature (converted to a number) for each predictor category for the future.

- *Different date formats*

With the different date formats such as "25-12-2021", "25th December 2021" etc. the machine learning model needs to be equipped with each of them. Or else, it is difficult for the machine learning model to understand all the formats.

With such a data set, you can preprocess or decompose the data by mentioning three different columns for the parts of the date, such as Year, Month and Day.
In Python, the preprocessing of the data with different columns for the date will look like this:

```
#Convert to datetime object
df['Date'] = pd.to_datetime(df['Date'])
#Decomposition
df['Year'] = df['Date'].dt.year
df['Month'] = df['Date'].dt.month
df['Day'] = df['Date'].dt.day
df[['Year','Month','Day']].head()
```

Output:

Year	Month	Day
2019	1	5
2019	3	8
2019	3	3
2019	1	27
2019	2	8

In the output above, the data set is in date format which is numerical. And because of decomposing the date in different parts such as Year, Month and Day, the machine learning model will be able to learn the date format.

REGRESSION

Regression is a supervised learning technique that supports finding the correlation among variables.
A regression problem is when the output variable is a real or continuous value.
What is a Regression?

In Regression, we plot a graph between the variables which best fit the given data points. The machine learning model can deliver predictions regarding the data. In naïve words, **"Regression shows a line or curve that passes through all the data points on a target-predictor graph in such a way that the vertical distance between the data points and the regression line is minimum."** *It is used principally for prediction, forecasting, time series modeling, and determining the causal-effect relationship between variables.*

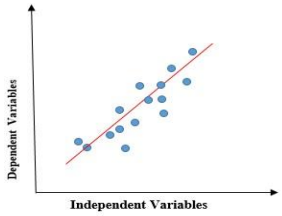
TYPES OF REGRESSION MODELS

1. Linear Regression
2. Polynomial Regression
3. Logistics Regression

1. Linear Regression:

Linear regression is a quiet and simple statistical regression method used for predictive analysis and shows the relationship between the continuous variables. Linear regression shows the linear relationship between the independent variable (X-axis) and the dependent variable (Y-axis), consequently called linear regression. If there is a single input variable (x), such linear regression is called **simple linear regression**. And if there

is more than one input variable, such linear regression is called **multiple linear regression.** The linear regression model gives a sloped straight line describing the relationship within the variables.



The above graph presents the linear relationship between the dependent variable and independent variables. When the value of x (**independent variable**) increases, the value of y (**dependent variable**) is likewise increasing. The red line is referred to as the best fit straight line. Based on the given data points, we try to plot a line that models the points the best.

To calculate best-fit line linear regression uses a traditional slope intercept form.

$$y = mx+b \quad \Longrightarrow \quad y = a_0+a_1x$$

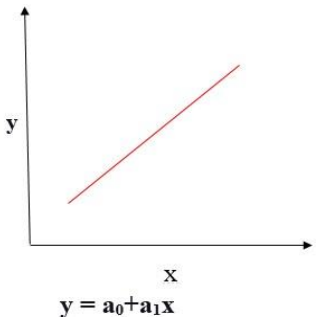
y= Dependent Variable.
x= Independent Variable.
a0= intercept of the line.
a1 = Linear regression coefficient.
Need of a Linear regression:

Linear regression estimates the relationship between a dependent variable and an independent variable. Let's say we want to estimate the salary of an employee based on year of experience. You have the recent company data, which indicates that the relationship between experience and salary. Here year of experience is an independent variable, and the salary of an employee is a dependent variable, as the salary of an employee is dependent on the experience of an employee. Using this insight, we can predict the future salary of the employee based on current & past information.

A regression line can be a Positive Linear Relationship or a Negative Linear Relationship.

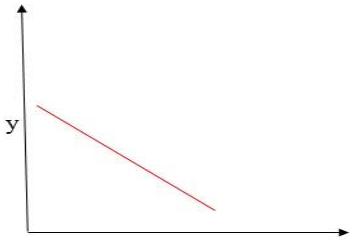
Positive Linear Relationship:

If the dependent variable expands on the Y-axis and the independent variable progress on X-axis, then such a relationship is termed a Positive linear relationship.



Negative Linear Relationship:

If the dependent variable decreases on the Y-axis and the independent variable increases on the X-axis, such a relationship is called a negative linear relationship.



$y = -a_0 + a_1 x$

The goal of the linear regression algorithm is to get the best values for a0 and a1 to find the best fit line. The best fit line should have the least error means the error between predicted values and actual values should be minimized.

Cost function

The cost function helps to figure out the best possible values for a0 and a1, which provides the best fit line for the data points.

Cost function optimizes the regression coefficients or weights and measures how a linear regression model is performing. The cost function is used to find the accuracy of the **mapping function** that maps the input variable to the output variable. This mapping function is also known as **the Hypothesis function**.

In Linear Regression, **Mean Squared Error (MSE)** cost function is used, which is the average of squared error that occurred between the predicted values and actual values.

By simple linear equation y=mx+b we can calculate MSE as:

Let's y = actual values, y_i = predicted values

$$MSE = \frac{1}{N}\sum_{i=1}^{n}(y_i - (mx_i + b))^2$$

Using the MSE function, we will change the values of a0 and a1 such that the MSE value settles at the minima. Model parameters ***xi, b (a_0,a_1)*** can be manipulated to minimize the cost function. These parameters can be determined using the gradient descent method so that the cost function value is minimum.

SCIKIT LEARN

- It is mainly used in machine learning
- It has lot of statistics related tools
- It is open source.
- By using the Scikit library the efficiency will improve tremendously as it is quite accurate.
- It is very useful in algorithms which are very famous in machine learning like K-mean, K-nearest, clustering etc.
- It is available to everybody so any programmer if he or she feels like utilizing it then can use it.
- Scikit requires Numpy

Features of Scikit learn are as follows:

- Clustering: Scikit can be applied in clustering algorithm, in clustering the grouping is done on the basis of similarities like eg: age, color etc.
- Cross validation
    - *Feature selection*

Example:

```
from sklearn.datasets import load_iris
iris = load_iris()
A= iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names
print("Feature names:", feature_names)
```

```
print("Target names:", target_names)
print("\nFirst 10 rows of A:\n", A[:10])
```

How to split a Dataset into Train and Test Sets using Python

Scikit-learn alias **sklearn** is the most useful and robust library for machine learning in Python. The **scikit-learn library** provides us with the model_selection module in which we have the splitter function train_test_split().

Syntax:

train_test_split(*arrays, test_size=None, train_size=None, random_state=None, shuffle=True, stratify=None)

Parameters:

1. *arrays: inputs such as lists, arrays, data frames, or matrices
2. test_size: this is a float value whose value ranges between 0.0 and 1.0. it represents the proportion of our test size. its default value is none.
3. train_size: this is a float value whose value ranges between 0.0 and 1.0. it represents the proportion of our train size. its default value is none.
4. random_state: this parameter is used to control the shuffling applied to the data before applying the split. it acts as a seed.
5. shuffle: This parameter is used to shuffle the data before splitting. Its default value is true.
6. stratify: This parameter is used to split the data in a stratified fashion.

Example

```
# read the dataset
# import modules
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
df = pd.read_csv('/home/student/Downloads/Real-estate.csv')
 # get the locations
X = df.iloc[:, :-1]
y = df.iloc[:, -1]
# split the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.05, random_state=0)
```

In the above example, **We import the pandas package and sklearn package**. after that to import the CSV file we use the read_csv() method. The variable df now contains the data frame. in the example “house price” is the column we’ve to predict so we take that column as y and the rest of the columns as our X variable. test_size = 0.05 specifies only 5% of the whole data is taken as our test set, and 95% as our train set. The random state helps us get the same random split each time.

Simple Linear Regression With scikit-learn

You’ll start with the simplest case, which is simple linear regression. There are five basic steps when you’re implementing linear regression:

1. Import the packages and classes that you need.
2. Provide data to work with, and eventually do appropriate transformations.
3. Create a regression model and fit it with existing data.
4. Check the results of model fitting to know whether the model is satisfactory.
5. Apply the model for predictions.

These steps are more or less general for most of the regression approaches and implementations. Throughout the rest of the tutorial, you'll learn how to do these steps for several different scenarios.

Step 1: Import packages and classes

The first step is to import the package numpy and the class LinearRegression from sklearn.linear_model:

```
>>> import numpy as np
>>> from sklearn.linear_model import LinearRegression
```

Now, you have all the functionalities that you need to implement linear regression.

The fundamental data type of NumPy is the array type called numpy.ndarray. The rest of this tutorial uses the term array to refer to instances of the type numpy.ndarray.

You'll use the class sklearn.linear_model.LinearRegression to perform linear and polynomial regression and make predictions accordingly.

Step 2: Provide data

The second step is defining data to work with. The inputs (regressors, x) and output (response, y) should be arrays or similar objects. This is the simplest way of providing data for regression:

:

```
>>> x = np.array([5, 15, 25, 35, 45, 55]).reshape((-1, 1))
>>> y = np.array([5, 20, 14, 32, 22, 38])
```

Now, you have two arrays: the input, x, and the output, y. You should call .reshape() on x because this array must be **two-dimensional**, or more precisely, it must have **one column** and **as many rows as necessary**. That's exactly what the argument (-1, 1) of .reshape() specifies.

This is how x and y look now:

```
>>> x
array([[ 5],
       [15],
       [25],
       [35],
       [45],
       [55]])

>>> y
array([ 5, 20, 14, 32, 22, 38])
```

As you can see, x has two dimensions, and x.shape is (6, 1), while y has a single dimension, and y.shape is (6,).

Step 3: Create a model and fit it

The next step is to create a linear regression model and fit it using the existing data.

Create an instance of the class LinearRegression, which will represent the regression model:

```
>>> model = LinearRegression()he
```

This statement creates the variable model as an instance of LinearRegression. You can provide several optional parameters to LinearRegression:

- fit_intercept is a Boolean that, if True, decides to calculate the intercept b_0 or, if False, considers it equal to zero. It defaults to True.
- normalize is a Boolean that, if True, decides to normalize the input variables. It defaults to False, in which case it doesn't normalize the input variables.
- copy_X is a Boolean that decides whether to copy (True) or overwrite the input variables (False). It's True by default.
- n_jobs is either an integer or None. It represents the number of jobs used in parallel computation. It defaults to None, which usually means one job. -1 means to use all available processors.

Your model as defined above uses the default values of all parameters.

It's time to start using the model. First, you need to call .fit() on model:

```
>>> model.fit(x, y)
LinearRegression()
```

With .fit(), you calculate the optimal values of the weights b_0 and b_1, using the existing input and output, x and y, as the arguments. In other words, .fit() **fits the model**. It returns self, which is the variable model itself. That's why you can replace the last two statements with this one:

```
>>> model = LinearRegression().fit(x, y)
```

This statement does the same thing as the previous two. It's just shorter.

Step 4: Get results

Once you have your model fitted, you can get the results to check whether the model works satisfactorily and to interpret it.

You can obtain the coefficient of determination, R^2, with .score() called on model:

```
>>> r_sq = model.score(x, y)
>>> print(f"coefficient of determination: {r_sq}")
coefficient of determination: 0.7158756137479542
```

When you're applying .score(), the arguments are also the predictor x and response y, and the return value is R^2.

The attributes of model are .intercept_, which represents the coefficient b_0, and .coef_, which represents b_1:

```
>>> print(f"intercept: {model.intercept_}")
intercept: 5.633333333333329

>>> print(f"slope: {model.coef_}")
slope: [0.54]
```

The code above illustrates how to get b_0 and b_1. You can notice that .intercept_ is a scalar, while .coef_ is an array.

Note: In scikit-learn, by convention, a trailing underscore indicates that an attribute is estimated. In this example, .intercept_ and .coef_ are estimated values.

The value of b_0 is approximately 5.63. This illustrates that your model predicts the response 5.63 when x is zero. The value $b_1 = 0.54$ means that the predicted response rises by 0.54 when x is increased by one.

You'll notice that you can provide y as a two-dimensional array as well. In this case, you'll get a similar result. This is how it might look:

```
>>> new_model = LinearRegression().fit(x, y.reshape((-1, 1)))
>>> print(f"intercept: {new_model.intercept_}")
intercept: [5.63333333]

>>> print(f"slope: {new_model.coef_}")
slope: [[0.54]]
```

As you can see, this example is very similar to the previous one, but in this case, .intercept_ is a one-dimensional array with the single element b_0, and .coef_ is a two-dimensional array with the single element b_1.

Step 5: Predict response

Once you have a satisfactory model, then you can use it for predictions with either existing or new data. To obtain the predicted response, use .predict():

```
>>> y_pred = model.predict(x)
>>> print(f"predicted response:\n{y_pred}")
predicted response:
[ 8.33333333 13.73333333 19.13333333 24.53333333 29.93333333 35.33333333]
```

When applying .predict(), you pass the regressor as the argument and get the corresponding predicted response. This is a nearly identical way to predict the response:

```
>>> y_pred = model.intercept_ + model.coef_ * x
>>> print(f"predicted response:\n{y_pred}")
predicted response:
[[ 8.33333333]
 [13.73333333]
 [19.13333333]
 [24.53333333]
 [29.93333333]
 [35.33333333]]
```

In this case, you multiply each element of x with model.coef_ and add model.intercept_ to the product.

The output here differs from the previous example only in dimensions. The predicted response is now a two-dimensional array, while in the previous case, it had one dimension.

If you reduce the number of dimensions of x to one, then these two approaches will yield the same result. You can do this by replacing x with x.reshape(-1), x.flatten(), or x.ravel() when multiplying it with model.coef_.

In practice, regression models are often applied for forecasts. This means that you can use fitted models to calculate the outputs based on new inputs:

```
>>> x_new = np.arange(5).reshape((-1, 1))
>>> x_new
array([[0],
       [1],
       [2],
       [3],
```

[4]])

```
>>> y_new = model.predict(x_new)
>>> y_new
array([5.63333333, 6.17333333, 6.71333333, 7.25333333, 7.79333333])
```

Here .predict() is applied to the new regressor x_new and yields the response y_new. This example conveniently uses arange() from numpy to generate an array with the elements from 0, inclusive, up to but excluding 5—that is, 0, 1, 2, 3, and 4.

Ref: https://realpython.com/linear-regression-in-python/

Question

1. Consider the hepatitis/ pima-indians-diabetes csv file, perform the following date pre-processing.
    1. Load data in Pandas.
    2. Drop columns that aren't useful.
    3. Drop rows with missing values.
    4. Create dummy variables.
    5. Take care of missing data.
    6. Convert the data frame to NumPy.
    7. Divide the data set into training data and test data.

2. a. Construct a CSV file with the following attributes:
Study time in hours of ML lab course (x)
Score out of 10 (y)
The dataset should contain 10 rows.
b. Create a regression model and display the following:
Coefficients: B0 (intercept) and B1 (slope)
RMSE (Root Mean Square Error)
Predicted responses
c. Create a scatter plot of the data points in red color and plot the graph of x vs. predicted y in blue color.
d. Implement the model using two methods:
Pedhazur formula (intuitive)
Calculus method (partial derivatives, refer to class notes)
e. Compare the coefficients obtained using both methods and compare them with the analytical solution.
f. Test your model to predict the score obtained when the study time of a student is 10 hours.
Note: Do not use scikit-learn.

Additional Question

1. a. Consider the hepatitis/diabetes CSV file. Create a regression model and display the following:

- Coefficients: B0 (intercept) and B1 (slope)
- RMSE (Root Mean Square Error)
- Predicted responses

b. Create a scatter plot of the data points in red color and plot the graph of x vs. predicted y in blue color.
c. Implement the model using two methods:
    1. Pedhazur formula (intuitive)
    2. Calculus method (partial derivatives, refer to class notes)

d. Compare the coefficients obtained using both methods. For a given data point, check the predicted y value.
Note: Do not use scikit-learn.