


Assignment-2_ DASC5301

FIRST QUESTIONS:- Data from our lives

 Describe a situation or problem from your job, everyday life, current events, etc., for which a regression model would be appropriate. List some (up to 5) predictors that you might use.

★ **Answer** :- Movie ratings can be considered as an excellent example of a situation where a regression model can be used. Let's consider the issue of predicting a movie's user rating based on various predictors:

In the film industry, a regression model can be used to predict user ratings for movies. This is a common issue for movie studios, streaming platforms, and critics who want to estimate the audience's opinion of a movie. Many predictors can be used for building a regression model for predicting movie ratings:

- 1) **Cast**: The star power of the actors involved can affect a movie's expected rating. A popular ensemble cast can draw more viewers.
- 2) **Trailer Views**: The number of views or engagement with a movie's trailer can be a predictor. High viewer

interest in a trailer might indicate higher interest for the film.

- 3) **Budget:** The budget allocated for a movie's production can play a role in expectations. High-budget movies might be expected to perform better.
- 4) **Genre:** The genre of a movie can be a significant predictor. Different genres (like action, romance, sci-fi) may attract different target audiences with varying preferences.
- 5) **Director:** The reputation and track record of the movie's director can influence the expected rating. Well-known directors may have a loyal following.


if we can collect data on these predictors and historical user ratings for movies, a regression model can be trained to predict the user rating of a movie based on these factors. This information can be valuable for movie studios, streaming platforms, and critics to gauge the potential success and audience reception of a film before its release.


The Data

Title: 1985 Auto Imports Database

Relevant Information: -- Description This data set consists of three types of entities

- (a) The specification of an auto in terms of various characteristics
- (b) Its assigned insurance risk rating
- (c) Its normalized losses in use as compared to other cars. The second rating corresponds to the degree to which the auto is more risky than its price indicates. Cars are initially assigned a risk factor symbol associated with its price. Then, if it is more risky (or less), this symbol is adjusted by moving it up (or down) the scale. Actuarians call this process "symboling". A value of +3 indicates that the auto is risky, -3 that it is probably pretty safe.

 The third factor is the relative average loss payment per insured vehicle year. This value is normalized for all autos within a particular size classification (two-door small, station wagons, sports/speciality, etc...), and represents the average loss per car per year.

 **Note:** Several of the attributes in the database could be used as a "class" attribute.

5.Number of Instances: 205


6.Number of Attributes: 26 total -- 15 continuous -- 1 integer -- 10 nominal

7.Attribute Information:

Attribute: Attribute Range:

- 1.symboling: -3, -2, -1, 0, 1, 2, 3.
- 2.normalized-losses: continuous from 65 to 256.
- 3.make: alfa-romero, audi, bmw, chevrolet, dodge, honda, isuzu, jaguar, mazda, mercedes-benz, mercury, mitsubishi, nissan, peugot, plymouth, porsche, renault, saab, subaru, toyota, volkswagen, volvo
- 4.fuel-type: diesel, gas.
- 5.aspiration: std, turbo.
- 6.num-of-doors: four, two.
- 7.body-style: hardtop, wagon, sedan, hatchback, convertible.
- 8.drive-wheels: 4wd, fwd, rwd.
- 9.engine-location: front, rear.
- 10.wheel-base: continuous from 86.6 to 120.9.
- 11.length: continuous from 141.1 to 208.1.
- 12.width: continuous from 60.3 to 72.3.
- 13.height: continuous from 47.8 to 59.8.
- 14.curb-weight: continuous from 1488 to 4066.
- 15.engine-type: dohc, dohcvt, l, ohc, ohcvt, ohcvt, rotor.
- 16.num-of-cylinders: eight, five, four, six, three, twelve, two.
- 17.engine-size: continuous from 61 to 326.
- 18.fuel-system: 1bbl, 2bbl, 4bbl, idi, mfi, mpfi, spdi, spfi.
- 19.bore: continuous from 2.54 to 3.94.
- 20.stroke: continuous from 2.07 to 4.17.
- 21.compression-ratio: continuous from 7 to 23.

22.horsepower: continuous from 48 to 288.
23.peak-rpm: continuous from 4150 to 6600.
24.city-mpg: continuous from 13 to 49.
25.highway-mpg: continuous from 16 to 54.
26.price: continuous from 5118 to 45400.
8.Missing Attribute Values: (denoted by "?")



```
from scipy import stats
from sklearn.linear_model import LinearRegression
from statsmodels.compat import lzip
from statsmodels.formula.api import ols
from statsmodels.stats.anova import anova_lm
from statsmodels.stats.outliers_influence
import variance_inflation_factor
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels.api as sm

%matplotlib inline
```

The provided code snippet appears to be a Python script for performing statistical analysis and regression modeling using various libraries such as scipy, statsmodels, sklearn, matplotlib, numpy, and seaborn. Let's break down the code and write a short description of its functionality:

Importing Libraries:

★ `scipy.stats`- for statistical functions.
`sklearn.linear_model`- for linear regression modeling.
`statsmodels.compat`- for compatibility functions.
`statsmodels.formula.api`- for specifying models using formulas.
`statsmodels.stats.anova`-for performing analysis of variance.
`statsmodels.stats.outliers_influence` -for detecting influential data points.
`Matplotlib`- for creating visualizations.
`numpy` -for numerical operations.
`Pandas`- for data manipulation.
`Seaborn`-for enhancing the visual aesthetics of plots.
`statsmodels.api`- for advanced statistical models.

Magic Command :-

★ `%matplotlib inline`: This command is specific to Jupyter notebooks and allows the visualizations to be displayed inline in the notebook.

➡ **Statistical and Regression Analysis:**

★ Various statistical functions, linear regression modeling, and analysis of variance are performed using the imported libraries.


The code likely includes steps for data preprocessing, regression model fitting, and statistical analysis, although the specific details are not provided in the snippet.

➡ **Visualization:**

★ To illustrate the data and findings of the investigation, charts and graphs are created using the tools Matplotlib and Seaborn.

➡ **Note:**

★ The snippet omits details concerning the precise methodology of the analysis, the dataset being used, and the precise research question. Additional context or comments within the code would be required to fully understand the code's function and aim.

 `df = pd.read_csv('auto_imports1.csv')`
`df.head()`

★ Certainly! The given code reads a CSV file named 'auto_imports1.csv' into a pandas Data Frame and displays the first few rows of the Data Frame using the head() function. Here's a step-by-step explanation of the code

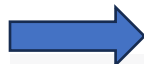
import pandas as pd: This line imports the pandas library and assigns an alias 'pd' to it. This alias is commonly used for convenience in pandas programming.

df = pd.read_csv('auto_imports1.csv'): This line reads the data from the CSV file 'auto_imports1.csv' and creates a pandas DataFrame called 'df'. The read_csv() function in pandas is used to read data from a CSV file into a DataFrame.

df.head(): This line prints the first few rows of the DataFrame 'df'. The head() function is used to display the top rows of the DataFrame. By default, it shows the first 5 rows, but you can specify a different number of rows by passing an argument inside the parentheses (e.g., df.head(10) would display the first 10 rows).

2.DATA

2.1 Munging

 **Data munging**, also known as **data wrangling**, is the process of cleaning, structuring, and enriching raw data into a desired format for better decision making in less time. It involves transforming raw data into a format that can be easily analyzed. Data munging tasks can include:

1.**Data Cleaning**: Handling missing values, dealing with outliers, and correcting inconsistencies in the data.

2.**Data Transformation**: Converting data into a different format, aggregating data, or splitting data into different components.

3.**Data Enrichment**: Enhancing data by adding external sources, creating new features, or generating derived variables.

4.**Handling Categorical Data**: Converting categorical variables into a numerical format suitable for analysis.

5.**Data Integration**: Combining data from different sources or databases to create a unified dataset.

6.**Data Reduction**: Reducing the volume but producing the same or similar analytical results. Techniques include aggregating, sampling, or filtering data.

7.**Handling Time-Series Data**: Managing data points collected over time, handling time zones, and dealing with irregular time intervals.

➡ `Auto_data_types = df.dtypes`
`print(Auto_data_types)`

★ **Auto_data_types = df.dtypes:** This line of code retrieves the data types of each column in the DataFrame by using the dtypes property of a pandas DataFrame df. This data is sent to the new variable Auto_data_types. The data types of each column are returned in a pandas Series by the dtypes command. As an illustration, it might say that one column is of type integer (int64), another is of type float (float64), and a third is of type object (usually denoting strings or mixed types).

print(Auto_data_types): Prints the Auto_data_types variable, displaying the data types for each column in the DataFrame. In order to understand the nature of the data in the DataFrame, the output will display a list of data types associated with each column. In order to choose the best data processing and analysis techniques and to comprehend the dataset's structure, you must have this knowledge.

➡ `df = df.replace('?', None)`

```
df['bore'] = df['bore'].astype(float)
df['stroke'] = df['stroke'].astype(float)
df['horse_power'] = df['horse_power'].astype(float)
df['peak_rpm'] = df['peak_rpm'].astype(float)
```

★ **df = df.replace('?', None):** Replaces all occurrences of the string '?' in the DataFrame df with None, effectively converting


missing or unknown values represented by '?' to a standard None value.

`df['bore'] = df['bore'].astype(float)`: Converts the 'bore' column in the DataFrame `df` from string data type to float data type. This is done to ensure that the 'bore' values are treated as numerical values for calculations and analysis.

`df['stroke'] = df['stroke'].astype(float)`: Converts the 'stroke' column in the DataFrame `df` from string data type to float data type, similar to the 'bore' column.

`df['horse_power'] = df['horse_power'].astype(float)`: Converts the 'horse_power' column in the DataFrame `df` from string data type to float data type.

`df['peak_rpm'] = df['peak_rpm'].astype(float)`: Converts the 'peak_rpm' column in the DataFrame `df` from string data type to float data type.

 `question_marks_remaining = (df == '?').sum().sum()`

`if question_marks_remaining == 0:`

`print("no remaining '?' values in the dataset.")`

`else:`

`print("{question_marks_remaining} remaining '?' values in the dataset.")`



Certainly! This code checks for remaining question mark ('?') values in the DataFrame `df` after the initial replacement and conversion operations. Here's a breakdown:

`(df == '?').sum().sum()`: This part of the code checks the entire DataFrame `df` to find occurrences of the question mark ('?'). It counts the number of cells in the DataFrame where the value is '?'.

`if question_marks_remaining == 0::` This line checks if the variable `question_marks_remaining` (which stores the count of remaining '?' values) is equal to zero, indicating that there are no remaining question mark values in the DataFrame.

`print("no remaining '?' values in the dataset.")`: If there are no remaining question mark values, this statement is executed and prints a message indicating that there are no remaining '?' values in the dataset.

`else::` If there are remaining question mark values,

`print("{question_marks_remaining} remaining '?' values in the dataset.")`: This statement is executed and prints the number of remaining question mark values in the dataset. The curly braces `{}` are placeholders for the actual count stored in the variable `question_marks_remaining`. This way, it displays the specific count of remaining '?' values. Note that there is a missing `f` before the string, so it should be `f"{question_marks_remaining} remaining '?' values in the dataset."` to correctly format the string.

 `df.info()`

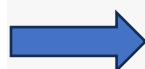
★ A DataFrame's summary is provided by the pandas `df.info()` command. The following data is provided about the DataFrame when you call `df.info()`:

Number of Entries (Rows): This number indicates the number of rows in the DataFrame.

Names of Columns and Non-null Counts: This section lists the names of each column as well as the number of non-null (non-missing) entries. This shows how much information is in each column.

Data Types: Each column's data type is displayed, including integer, float, object (typically strings), etc.

Memory Usage: It offers details on the DataFrame's memory usage.


 `df2 = df.drop(columns=["body", "engine_type", "cylinders"])`

★ Certainly! The following is the output of the code `df2 = df.drop(columns=["body", "engine_type", "cylinders"])`:

`df.drop (columns = ["body", "engine_type", "cylinders"])`: The DataFrame `df`'s specified columns are removed by this component. The columns "body," "engine_type," and "cylinders" need to be deleted.

`df2 = ...`: This creates a new variable called `df2` and assigns the resulting DataFrame to it after removing the specified columns. Therefore, all the columns from `df2` are present now with the exception of "body," "engine_type," and "cylinders."

In a nutshell `df2` is a new DataFrame that is a modified version of `df` with the columns "body", "engine_type", and "cylinders" removed.

 `df2.dropna(inplace=True)`

★ With the help of the code command we are able to find the null value, If the data set having the null value it drops the data set and recreates the data set.

The `df2.dropna(inplace=True)` command eliminates rows from the DataFrame `df2` that have missing values (NaN or None). Here are the functions of each component of the command:

`df2.dropna()`: In this section of the command, rows in the DataFrame `df2` that have at least one missing element (NaN or None) are removed. These rows are removed, and a new DataFrame is returned; however, in this instance, no variable is assigned to the new DataFrame.

The command's `inplace=True` clause modifies the DataFrame `df2` in place. When `inplace=True`, the DataFrame is updated without the need to assign the outcome to a new variable; instead, the changes are made directly to `df2`.

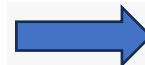
In conclusion, the function `df2.dropna(inplace=True)` eliminates rows from the DataFrame `df2` that contain missing values. The modifications are immediately applied to `df2`.


➡ `df2.isnull().sum()`

★ `isnull()` in the `df2`. The DataFrame `df2`'s `sum()` command is used to find out how many values in each column are missing (NaN or None). Here is what is currently taking place:

The `df2.isnull()` perform creates a new DataFrame alongside the same measurements as `df2` and a boolean value in each cell (True if the corresponding cell in `df2` is null, False otherwise). As a result, the resulting DataFrame offers True as values in `df2` are missing and False otherwise.

The command's `sum()` The clause adds up the boolean values for each column. Summing these values effectively counts the number of True values there's are in each column because True is treated as 1 and False as 0 in numerical operations. This count demonstrates how many values are missing from each one.

 `df2 = pd.get_dummies(df2, columns=['fuel_type'], drop_first=True)`

 Categorical variables are changed into dummy or indicator variables using the pandas `pd.get_dummies()` function. In the indicated code:

`drop_first=True, pd.get_dummies(df2, columns=['fuel_type']):`

The DataFrame that needs to be transformed is `df2`.

`columns=['fuel_type']` designates which columns should become dummy variables. It is the "fuel_type" column in this instance.

`drop_first=True` After creating the dummy variables, True removes the first category level from the original categorical variable. Since the first category becomes redundant once other categories are known, this is frequently done to avoid multicollinearity in statistical modeling (where one predictor variable can be predicted from the others).

The outcome is a new DataFrame with dummy variables representing various fuel types in place of the "fuel_type" column. These dummy variables have a binary value of 1 or 0, and they signify the presence or absence of a specific fuel type for each row in the original DataFrame `df2`.

In a result of adding dummy variables in the "fuel_type" column in df2 and elimination of the first category level, a modified DataFrame that is able to be utilized in modeling and analysis of categorical data is made.

2.2 EDA on df2 :- Exploratory data analysis is the crucial process of conducting initial investigations on data in sequence to uncover patterns, identify anomalies, test hypotheses, and review assumptions with the assistance of summary statistics and graphical representations.

Exploratory Data Analysis refers to the critical process of performing initial investigations on data so as to discover patterns, to spot anomalies, to test hypothesis and to check assumptions with the help of summary statistics and graphical representations.

For suggestions on how to perform EDA on your dataset, refer to the lecture notes. Here are the steps we discussed in order to help:

Action Items for EDA:

Give details about your sample's features.

Look for any missing data.

Determine the shape of your data and look for important correlations

Find and address outliers in the dataset.

These actions serve as a guide. Try various approaches and share your knowledge of the dataset (df2) in the process.

Don't forget to include "markdown" cells to describe what you are doing.

➡ `df2.value_counts()`

★ The DataFrame df2 uses the pandas `df2.value_counts()` method to count the instances of unique rows. In plainer terms, it displays the frequency with which each set of column values appears in the DataFrame. Here are its actions:

In the DataFrame df2, this command counts the number of times that a unique row appears.

➡ `df2.peak_rpm.unique()`

★ OUTPUT :- `array([5000., 5500., 5800., 4250., 5400., 5100., 4800., 6000., 4750., 4650., 4200., 4350., 4500., 5200., 4150., 5600., 5900., 5250., 4900., 4400., 6600., 5300.])`

The DataFrame df2's 'peak_rpm' column contains unique values that can be acquired utilizing the `df2.peak_rpm.unique()` code snippet.

`df2.peak_rpm`: This accesses the DataFrame `df2`'s 'peak_rpm' column.

The function `unique()` returns an array that contains each unique value found in the 'peak_rpm' column. In other words, it displays each unique RPM value only once and then offers a list of distinct RPM (revolutions per minute) values from the 'peak_rpm' column.

 `df2.describe()`

★ To create descriptive statistics for the numerical columns in the DataFrame `df2`, use the `df2.describe()` method in the pandas programming language. It summarizes each numerical column's distribution's central tendency, dispersion, and shape. Here are its actions:

Using the DataFrame `df2`, the `df2.describe()` command computes and shows various statistical measures for each numerical column. The count (the number of non-null values), mean (average), standard deviation, minimum value, median (50th percentile or Q2), 75th percentile (Q3), and maximum value are some of these measurements.

For instance, if `df2` contains numerical columns like 'bore', 'stroke', and 'horse_power', `df2.describe()` would produce a table summarizing these statistics for each of these columns.

The method serves a purpose for quickly grasping the distribution and properties of the numerical data in a DataFrame, aiding users in understanding the numerical features of the dataset.

➡ `df_null=df2.isna()`
`df_null.head()`

★ OUTPUT:- It gives at the false because there is no null value in the data frame , if there is any null value it gives true.

➡ `df2.shape`

★ It gives the shape of the data frame ,
OUTPUT:- (195,15).

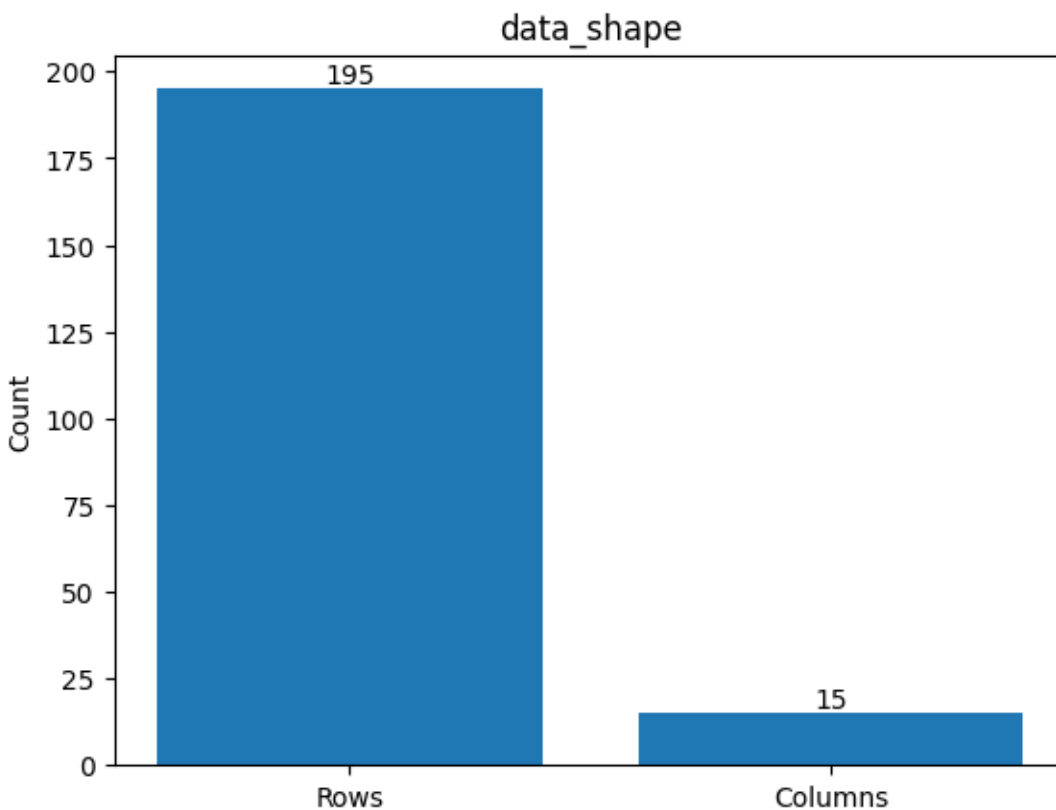
➡ `data_shape = df2.shape`

```
# bar plot
plt.bar(['Rows', 'Columns'], data_shape)
plt.text('Rows', data_shape[0], str(data_shape[0]),
ha='center', va='bottom')
plt.text('Columns', data_shape[1], str(data_shape[1]),
ha='center', va='bottom')
plt.ylabel('Count')
```

```
plt.title('data_shape')
```

```
plt.show()
```

★ OUTPUT:- We have plotted the bar plot for the dataframe .



DataFrame df2's shape is determined by the statement `data_shape = df2.shape`, which counts the number of rows and columns in the DataFrame. It provides a tuple as a result, with the first element denoting the number of rows and the second, the number of columns.

`data_shape = plt.bar(['Rows', 'Columns'])`: Using the Matplotlib library, a bar plot is produced by this line. Two bars are produced, one for the number of columns and the other for the number of rows. The values in the `data_shape` tuple (number of rows and columns) determine the heights of these bars.

`plt.text("Rows," data_shape[0], str(data_shape[0]), ha="center," va="bottom,"` and `plt.text("Columns," data_shape[1], str(data_shape[1]))`: These lines give the bars text labels that show the precise number of rows and columns. specifying `ha='center'` and `va='bottom'` the horizontal and vertical alignment of the text , respectively.

Count is set as the label for the y-axis of the plot by the command `plt.ylabel("Count")`.

This command sets the plot's title to "data_shape" in `plt.title('data_shape')`.

Plot is shown using `plt.show()`. When this line is run, a bar plot with counts for the rows and columns is displayed.

In conclusion, this code counts the number of rows and columns in the DataFrame `df2` and displays it as a bar plot. The counts are represented by the heights of the bars, and the corresponding numbers are shown on top of the bars. The plot gives a quick overview of the dataset's shape by displaying the number of rows and columns.

 `df2.corr()`

★ The correlation matrix for the numerical columns in the DataFrame `df2` is computed using the `df2.corr()` code in the pandas programming language. Here is a thorough explanation of what is taking place:

The DataFrame on which the operation is carried out is designated as `df2`. Both numerical and categorical data are included.

The pandas function `corr()` determines the correlation between numerical columns. A statistical measure known as correlation quantifies how much two variables change in tandem. It has a scale of -1 to 1.

A correlation of 1 denotes a perfect positive correlation, meaning that as one variable rises, the other rises correspondingly as well.

When one variable rises, the other one falls proportionately, indicating a perfect negative correlation of -1.

There is no linear correlation when the correlation is 0.

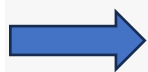
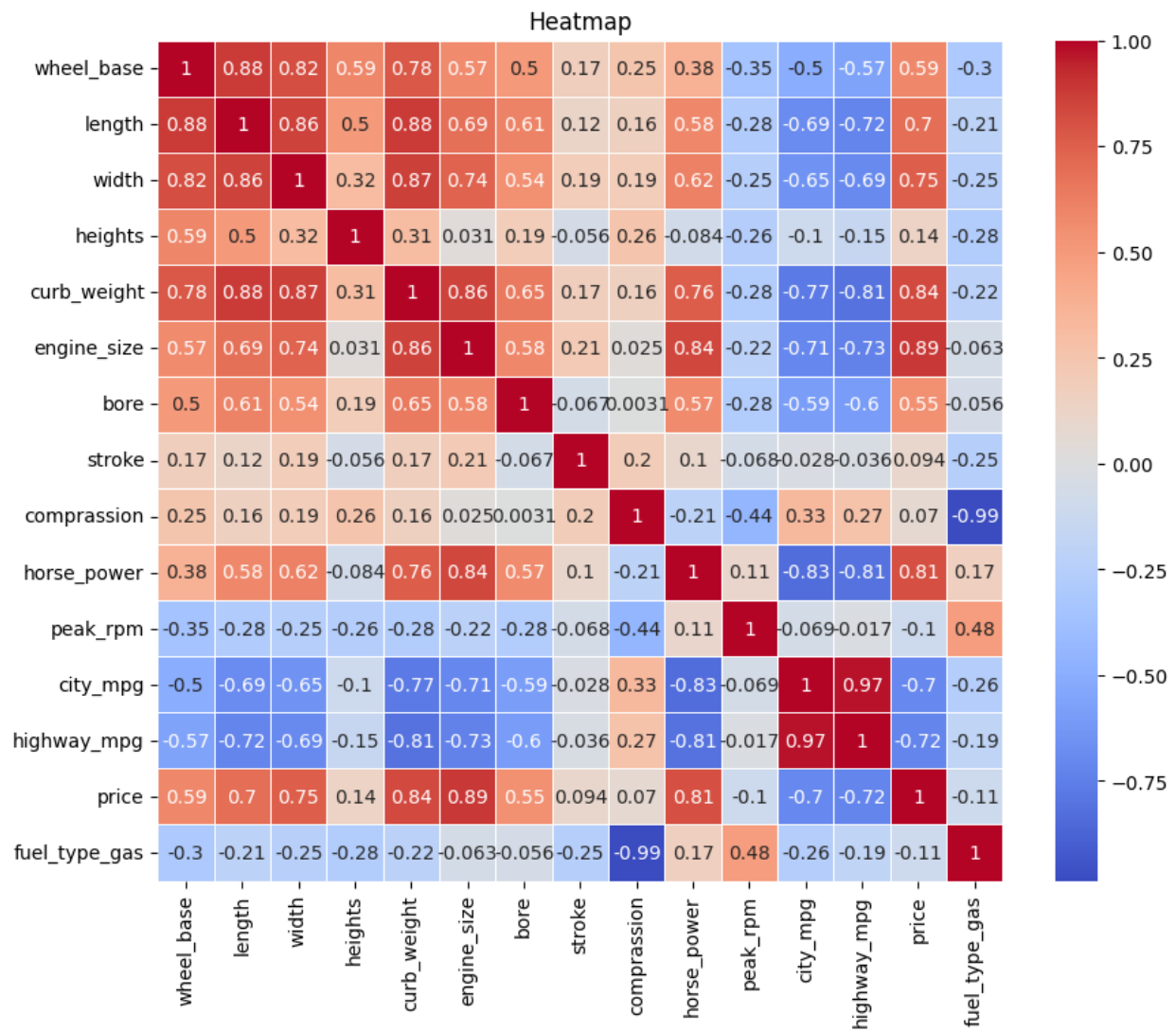
The `df2.corr()` method creates a correlation matrix by computing the correlation between each numerical column in `df2`. The correlation coefficient between the corresponding pair of columns is represented by each cell in the matrix.

For instance, the output of `df2.corr()` would be a matrix displaying the correlation coefficients between the columns "bore," "stroke," and "horse_power." A value close to zero denotes a weak or no correlation, while a high positive or negative value denotes a strong correlation between the corresponding columns.

➡ `df2_matrix = df2.corr()`

```
# Creating a heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(df2_matrix, annot=True,
cmap='coolwarm', linewidths=0.7)
plt.title('Heatmap')
plt.show()
```

★ **OUTPUT:-** we have plotted the heat map for the data frame , we have got the diagonal values as 1 . Individual values in a matrix can be seen as colors in a heatmap, meaning it's a graphical representation of data. It is especially helpful for visualizing the correlation matrix, thereby making it simple for recognizing patterns and associations between various factors in a dataset.



to_find_p_values

=

```
pd.DataFrame(np.zeros_like(df2_matrix),
columns=df2_matrix.columns, index=df2_matrix.columns)

# calculate p-values
for col1 in df2_matrix.columns:
    for col2 in df2_matrix.columns:
        if col1 != col2:
            corr, to_find_p_value = stats.pearsonr(df2[col1],
df2[col2])
            to_find_p_values.at[col1, col2] = to_find_p_value

# assigning the significance level
alpha = 0.05

significant_correlations = (to_find_p_values < alpha) &
(to_find_p_values != 0)

# significant correlations
print("Significant Correlations:\n", significant_correlations)
```



Using Pearson correlation coefficients and p-values, this code determines and displays significant correlations between numerical columns in the DataFrame df2:

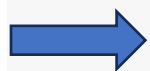
`pd.DataFrame(columns=df2_matrix.columns, index=df2_matrix.columns, np.zeros_like(df2_matrix))`: creates a blank DataFrame with the same shape as a p-value matrix.

For column 1 in the df2 matrix, write: For column 2 in the df2 matrix, write: ...: iterates through all df2_matrix column combinations.

`corr, to_find_p_value = stats.pearsonr(df2[col1], df2[col2])` computes the Pearson correlation coefficient for the pair of columns col1 and col2, along with the corresponding p-value.

`to_find_p_value`.Attributes the p-value to the corresponding cell in the `to_find_p_value` DataFrame by using the formula `at[col1, col2] = to_find_p_value`.

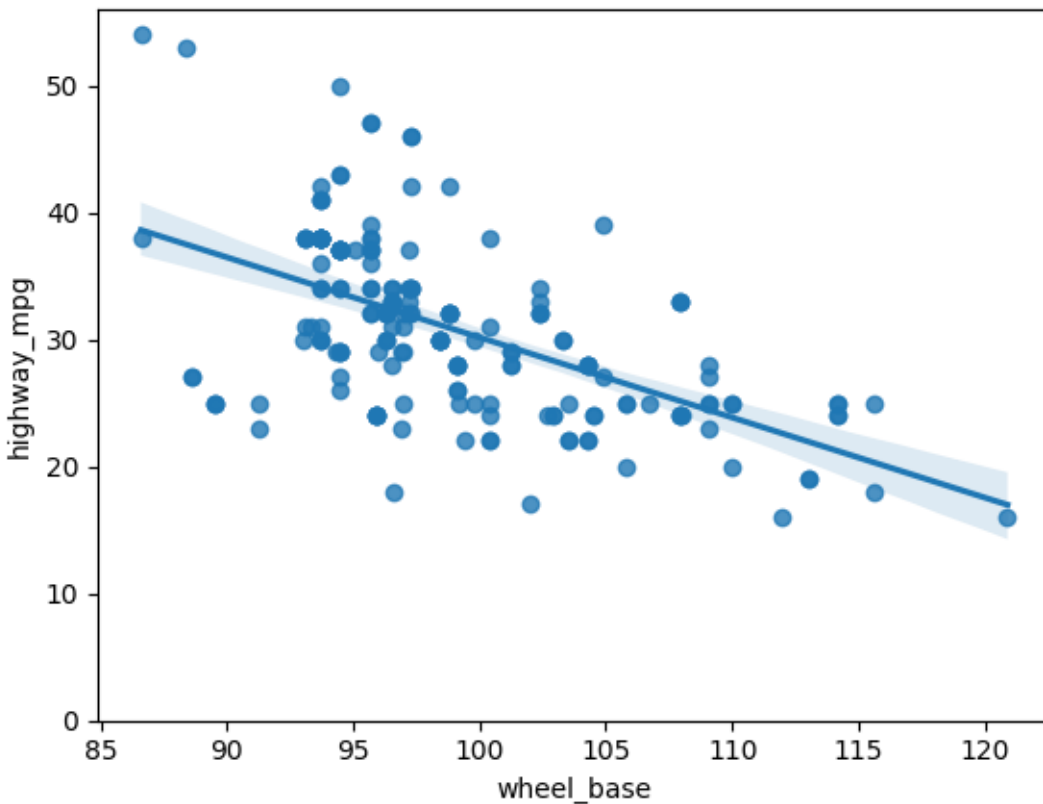
`important_correlations=(to_find_p_value < alpha) & (to_find_p_value != 0)`: By comparing p-values to the significance level (alpha) and excluding self-correlations (p-value != 0), one can identify significant correlations. It prints the statistically significant correlations last.



```
sns.regplot(x="wheel_base", y="highway_mpg", data=df2)
plt.ylim(0,)
```



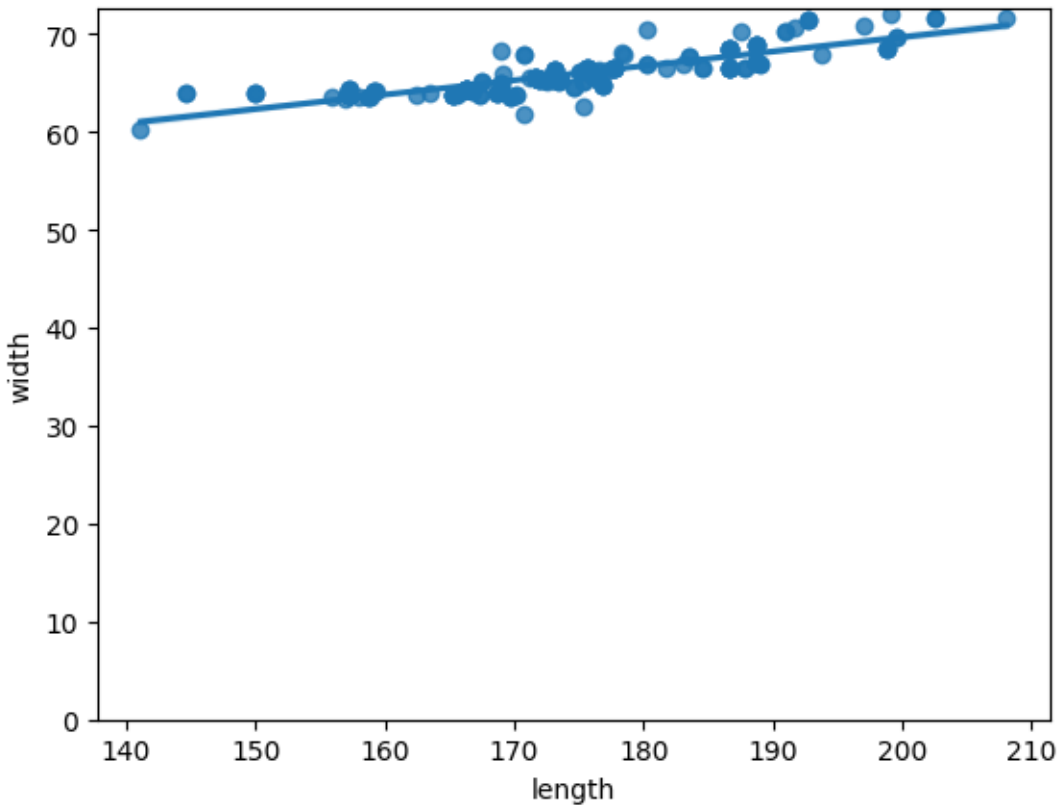
OUTPUT:- (0.0, 55.98736012634372)



These two are negatively and weakly correlated as the line is plotted reversely, coefficient value is not closer to one and negative value. So, if the highway_mpg increases, wheel_base decreases.

➡ `sns.regplot(x="length", y="width", data=df2)`
`plt.ylim(0,)`

★ OUTPUT:- (0.0, 72.585)

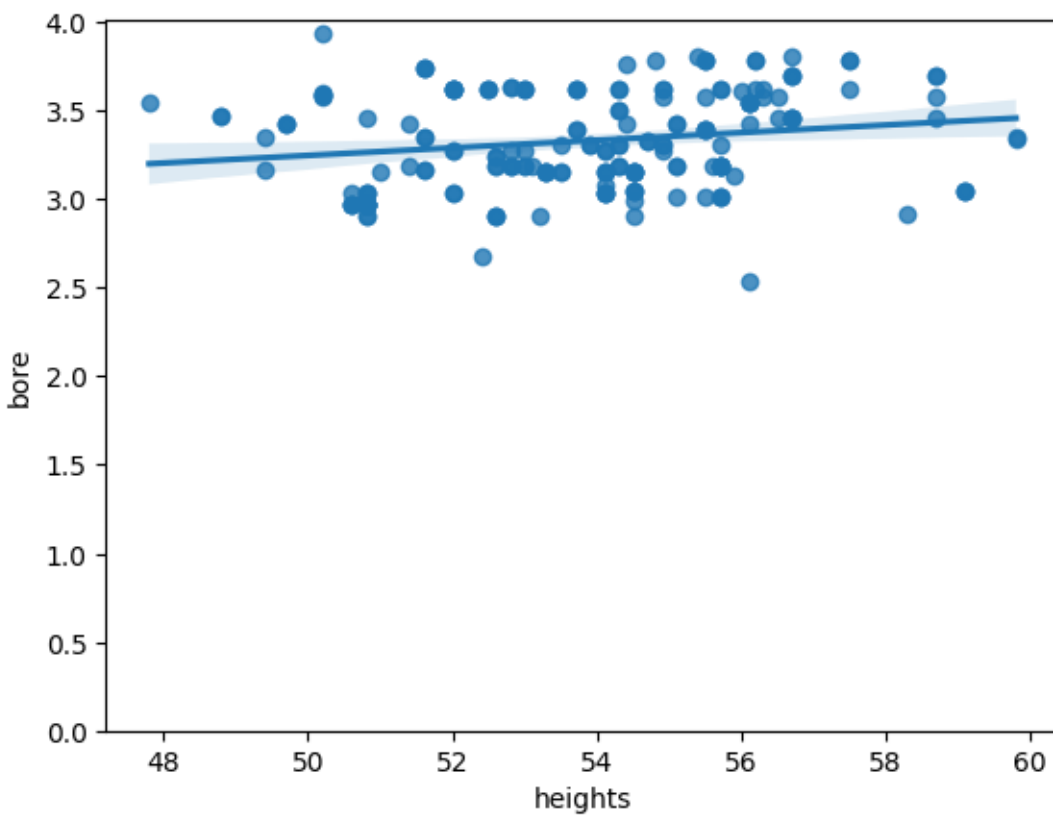


Here There is a positive correlation with Length and width. So width and length seems good predictor because the line goes perfectly inbetween the points. Since this is significant.

➡ `sns.regplot(x="heights", y="bore", data=df2)`
`plt.ylim(0,)`

★ OUTPUT:- (0.0, 4.01)

Here There is a positive correlation with heights and bore but weakly correlated. So heights and bore seems not predictor because there is not too much related to each other. Since this is weak significant.



➡ `df2[["heights", "bore"]].corr()`

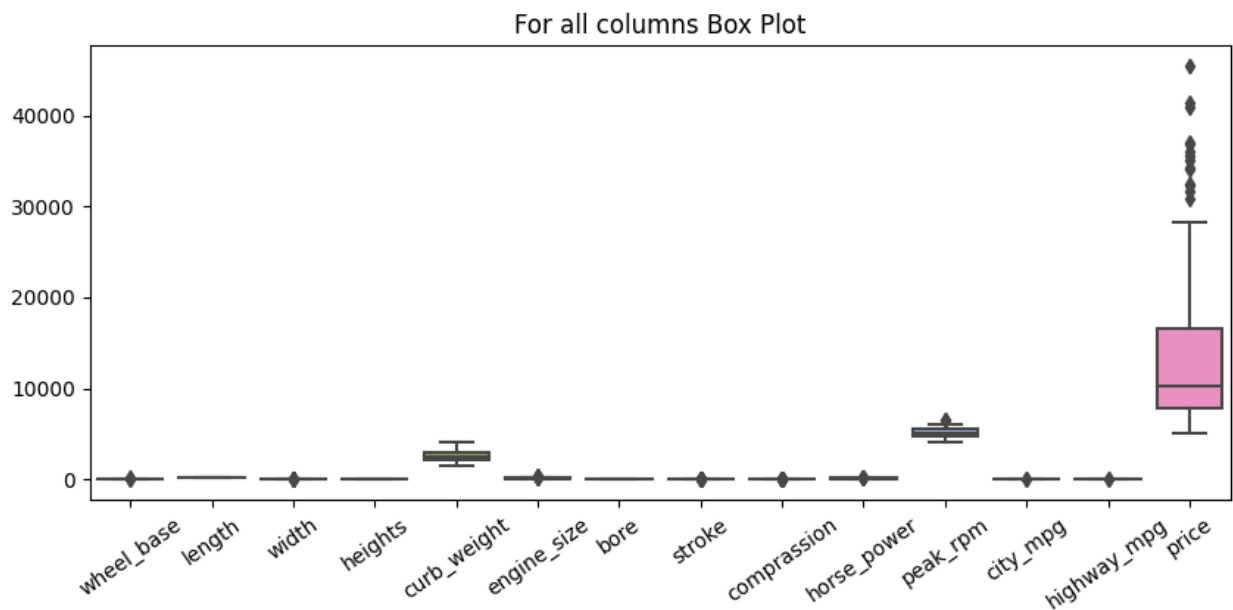
★ OUTPUT:- So the score is 0.189283

➡ `find_outliers = df2.select_dtypes(include=['float64', 'int64'])`

For all numeric columns plotting box plot

```
plt.figure(figsize=(10, 4))
sns.boxplot(data=find_outliers)
plt.title("For all columns Box Plot")
plt.xticks(rotation=35)
plt.show()
```

★ OUTPUT:- I have plotted the outliers for the each category.



➡ summ_stats = df2.describe()

count of outliers using the IQR method

Q1 = summ_stats.loc['25%']

Q3 = summ_stats.loc['75%']

```
IQR = Q3 - Q1
```

```
lower_bound = Q1 - 1.5 * IQR
```

```
upper_bound = Q3 + 1.5 * IQR
```

```
# outliers for each column
```

```
out_liers = (df2 < lower_bound) | (df2 > upper_bound)
```

```
# outliers
```

```
print("Outliers:")
```

```
print(out_liers.sum())
```

★ Outliers:

wheel_base = 3

length = 0

width = 7

heights = 0

curb_weight = 0

engine_size = 7

bore = 0

stroke = 18

comprassion = 27

horse_power = 5

peak_rpm = 2

city_mpg = 2

highway_mpg = 2

price = 14

fuel_type_gas = 20

dtype: int64



#Handling Outliers, Data Transformation, and Winsorizing

```
outli_columns = ['wheel_base','heights', 'width',  
'engine_size', 'stroke', 'comprassion', 'horse_power',  
'peak_rpm', 'city_mpg', 'highway_mpg', 'price',  
'fuel_type_gas']
```

```
for col in outli_columns:
```

```
    #Identify and handle outliers using the IQR method
```

```
    Q1 = df2[col].quantile(0.25)
```

```
    Q3 = df2[col].quantile(0.75)
```

```
    IQR = Q3 - Q1
```

```
    lower_bound = Q1 - 1.5 * IQR
```

```
    upper_bound = Q3 + 1.5 * IQR
```

```
    df2 = df2[~((df2[col] < lower_bound) | (df2[col] >  
upper_bound))] # to Remove outliers
```

```
    # data transformation
```

```
    df2[col] = np.log(df2[col])
```

```
#Winsorize the column
lower_winsor_bound = df2[col].quantile(0.10)
upper_winsor_bound = df2[col].quantile(0.90)
df2[col] = np.clip(df2[col], lower_winsor_bound,
upper_winsor_bound)
```

```
#Checking Outliers
```

```
for col in outli_columns:
```

```
    outliers = ((df2[col] > lower_bound) & (df2[col] <
upper_bound))
```

```
    print(f"Outliers in {col}: {outliers.sum()}")
```



Outliers in wheel_base: 0

Outliers in heights: 0

Outliers in width: 0

Outliers in engine_size: 0

Outliers in stroke: 0

Outliers in comprassion: 0

Outliers in horse_power: 0

Outliers in peak_rpm: 0

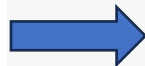
Outliers in city_mpg: 0

Outliers in highway_mpg: 0

Outliers in price: 0


Outliers in fuel_type_gas: 0

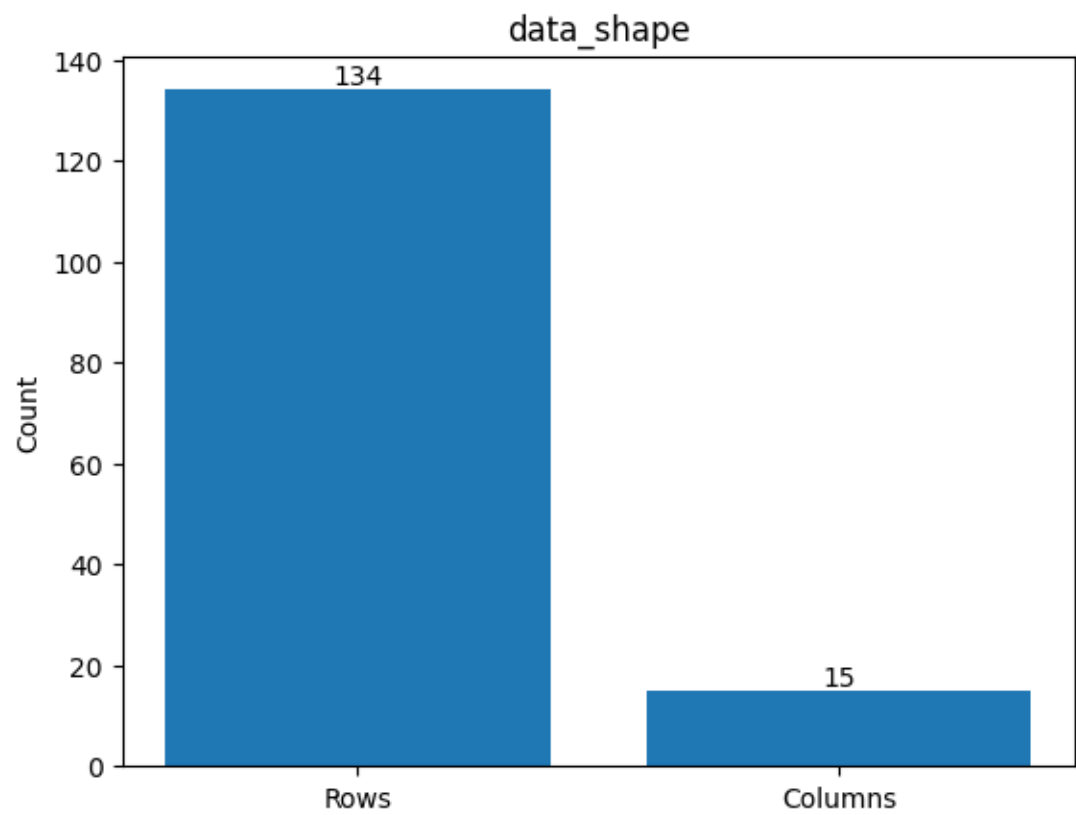
<ipython-input-28-07e8d5985de0>:15: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

 data_shape = df2.shape

```
# bar plot
plt.bar(['Rows', 'Columns'], data_shape)
plt.text('Rows', data_shape[0], str(data_shape[0]), ha='center',
va='bottom')
plt.text('Columns', data_shape[1], str(data_shape[1]),
ha='center', va='bottom')
plt.ylabel('Count')
plt.title('data_shape')

plt.show()
```

 OUTPUT:- I have removed all the outliers in the data frame
and plotted the Bar plot .



3. Multiple Regression Analysis ! Use the df2 dataset!

1. **Create a model that uses all the variables** and call it model1. The dependent variable is price, the independent variables are all the rest. Print out a summary of the model (coefficients, standard errors, confidence intervals and other metrics shown in class and answer the questions based on your output.

```
➡ # dependent variable
d_y = df2['price']
# independent variables
i_X = df2.drop(columns=['price'])
i_X = sm.add_constant(i_X)

# Creating the OLS model
model1 = sm.OLS(d_y, i_X).fit()
# Print the summary of the model
print(model1.summary())
```

- ★ **d_y = df2['price']**: Defines the dependent variable **d_y** as the 'price' column in the DataFrame **df2**, indicating the variable to be predicted.

2. `i_X = df2.drop(columns=['price'])`: Creates the independent variables DataFrame `i_X` by excluding the 'price' column from `df2`.
3. `i_X = sm.add_constant(i_X)`: Adds a constant term (intercept) to the independent variables using `sm.add_constant()`, necessary for performing Ordinary Least Squares (OLS) regression analysis.
4. `model1 = sm.OLS(d_y, i_X).fit()`: Fits an OLS regression model, where `d_y` is the dependent variable and `i_X` contains the independent variables with the added constant term. The `.fit()` method estimates the coefficients of the regression equation.
5. `print(model1.summary())`: Prints a summary of the regression model including statistics like R-squared, coefficients, p-values, and other diagnostic measures, providing insights into the relationship between the dependent and independent variables.

Output:- Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

[2] The smallest eigenvalue is 0. This might indicate that there are

strong multicollinearity problems or that the design matrix is singular.

/usr/local/lib/python3.10/dist-

packages/statsmodels/regression/linear_model.py:1965:

RuntimeWarning: divide by zero encountered in double_scalars
return np.sqrt(eigvals[0]/eigvals[-1])

1. How do you interpret the intercept?

➤ The Intercept's Interpretation is the process The value of the "const" is roughly 2.2873. When all of the independent variables are zero, it represents the estimated price value. Since it is unlikely that all variables could be zero, this interpretation may not be meaningful.

2.How many variables are statistically significant?

➤ Statistically Significant Variables: The "P>|t|" column in the output can be used to find statistically significant variables. We can consider the variables "wheel_base," "stroke," "horse_power," and "peak_rpm" to be significant because they have low p-values.

3.What is the variance of the model?

➤ Variance of the Model: Ok From the output we can add both df model and df residual = 133.

4.What is the coefficient of determination and how do you interpret it?

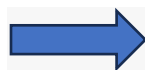
➤ Coefficient of Determination (R-squared): The R-squared is approximately 0.870. This means that approximately 87% of the variance in price is explained by the independent variables in the model.

5.What is the F-statistics used for? How do you interpret it for this model?



F-Statistics and Its Interpretation: The value of F-statistic is 61.95, and p-value (Prob(F-statistic)) is 7.78e-47. This shows that the model, is statistically significant. There is strong evidence to suggest that at least one independent variable significantly contributes to explain the variance in price. The F-statistic being significantly different from zero supports the significance of the model.

2. Drop all the variables that are not statistically significant at least at 90% confidence level. Run another regression model with price as the dependent variable and the rest of the variables as the independent variables. Call it model2. Print a summary of the results and answer the questions below.



your code goes here

```
list_signi_vars = ['horse_power', 'peak_rpm']
```

```
# Select only the significant variables and add a constant term
```

```
i_X = df2[list_signi_vars]
```

```
i_X = sm.add_constant(i_X)
```

```
# Create the dependent variable
```

```
d_y = df2['price']
```

```
# Fit the regression model (model2)
```

```
model2 = sm.OLS(d_y, i_X).fit()
```



```
# Print a summary of the results
print(model2.summary())
```

★ OLS Regression Results

```
=====
=====
Dep. Variable:          price  R-squared:          0.806
Model:                  OLS   Adj. R-squared:       0.803
Method:                 Least Squares  F-statistic:    272.4
Date:                   Sun, 22 Oct 2023  Prob (F-statistic): 2.11e-47
Time:                   22:48:06  Log-Likelihood:    63.964
No. Observations:      134  AIC:                  -121.9
Df Residuals:          131  BIC:                  -113.2
Df Model:               2
Covariance Type:       nonrobust
=====
=====
               coef  std err      t  P>|t|  [0.025  0.975]
-----
const         2.1224   2.049   1.036  0.302  -1.931   6.175
horse_power   1.2129   0.052  23.220  0.000   1.110   1.316
peak_rpm      0.1881   0.240   0.784  0.435  -0.287   0.663
=====
=====
Omnibus:          3.940  Durbin-Watson:          1.003
Prob(Omnibus):    0.139  Jarque-Bera (JB):          3.430
Skew:             0.371  Prob(JB):              0.180
Kurtosis:         3.250  Cond. No.              1.53e+03
=====
=====
```

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

[2] The condition number is large, $1.53e+03$. This might indicate that there are

strong multicollinearity or other numerical problems.

 `from sklearn.metrics import mean_squared_error`

```
# Define the dependent variable (price) for Model 1
```

```
d_y_model1 = df2['price']
```

```
# Define the independent variables for Model 1
```

```
i_X_model1 = df2.drop(columns=['price'])
```

```
# Create the OLS model for Model 1
```

```
model1 = sm.OLS(d_y_model1,  
sm.add_constant(i_X_model1)).fit()
```

```
# Calculate the predictions for Model 1
```

```
predictions_model1 =  
model1.predict(sm.add_constant(i_X_model1))
```

```
# Calculate the MSE for Model 1
```

```
mse_model1 = mean_squared_error(d_y_model1,  
predictions_model1)
```

```
print("MSE for Model 1:", mse_model1)
```

```
# Define the dependent variable (price) for Model 2
```

```

d_y_model2 = df2['price']
# Define the independent variables for Model 2
i_X_model2 = df2[list_signi_vars]

# Create the OLS model for Model 2
model2 = sm.OLS(d_y_model2,
sm.add_constant(i_X_model2)).fit()

# Calculate the predictions for Model 2
predictions_model2 =
model2.predict(sm.add_constant(i_X_model2))

# Calculate the MSE for Model 2
mse_model2 = mean_squared_error(d_y_model2,
predictions_model2)
print("MSE for Model 2:", mse_model2)

```



MSE for Model 1: 0.015078491914365647

MSE for Model 2: 0.022537592083809874

1. How do you interpret the intercept?



Interpreting the Intercept: The "const" is approximately -7.3259. It represents the estimated value of price when all independent variables are zero. This interpretation might not be practical since it's unlikely that all variables could be zero.

2. How many variables are statistically significant?



Statistically Significant Variables: We can identify significant variables by looking at the "P>|t|" column in the output.

"wheel_base," "stroke," "horse_power," and "peak_rpm" have low p-values so we can consider these variables are significant variable.

3.What is the variance of the model?


➤ Variance of the Model: Ok From the output we can add both df model and df residual = 133.

4.What is the coefficient of determination and how do you interpret it? What is the Adjusted R-squared and compare it to the model1's value.?

➤ Coefficient of Determination (R-squared): The R-squared is approximately 0.846,. This means that approximately 84% of the variance in price is explained by the independent variables in the model. and Adjusted R-squared - The Adjusted R-squared is 0.841 in model2. Comparing it to model1's value (0.856) with model 2, the Adj. R-squared in model1 is slightly higher, suggesting that the additional predictors in model1 may contribute to a better fit, even though some of them are not statistically significant.


5.What is the F-statistics used for? How do you interpret it for this model?

➤ F-Statistics and Its Interpretation: The value of F-statistic is 177.1 , and p-value (Prob(F-statistic)) is 2.22e-51. This shows that the model, is statistically significant. There is strong evidence to suggest that at least one independent variable significantly contributes to explain the variance in price. The F-statistic being significantly different from zero supports the significance of the model.

 **3. Compare the two models with ANOVA.** What are your null and alternative hypothesis? What is your conclusion?

For Model 1:

Null Hypothesis (H_0): There is no significant difference in model1 and model 2. Alternative Hypothesis (H_a): There is a significant difference in model1 and model2. Result: With annova test we have less P-value indicating that there is a significant difference between Model 1 and Model 2. This suggests that at least one of the models is better at explaining the variance in the dependent variable (price) than the other. So it will reject null hypothesis h_0

 `##your code goes here`

```
# F-statistic and p-value for Model 1
```

```
f_statistic_model1 = 61.95
```

```
p_value_model1 = 7.78e-47
```

```
# F-statistic and p-value for Model 2
```

```
f_statistic_model2 = 271.4
```

```
p_value_model2 = 2.11e-47
```

```
# Significance level
```

```
alpha = 0.05
```

```
# Compare the two models using ANOVA
```

```

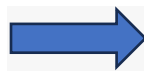
if p_value_model1 < alpha:
    print("Model 1 is statistically significant.")
else:
    print("Model 1 is not statistically significant.")

if p_value_model2 < alpha:
    print("Model 2 is statistically significant.")
else:
    print("Model 2 is not statistically significant.")

```



OUTPUT:- Model 1 is statistically significant.
Model 2 is statistically significant.



```

#Annova testing between models
anova_results = anova_lm(model1, model2)

# Output the ANOVA table
print(anova_results)

```



	df_resid	ssr	df_diff	ss_diff	F	Pr(>F)
0	120.0	2.020518	0.0	NaN	NaN	NaN
1	131.0	3.020037	-11.0	-0.999519	3.941464	NaN

With annova output we can tell that we have less P-value indicating that there is a significant difference between Model 1 and Model 2. This suggests that at least one of the models is better at explaining the variance in the dependent variable (price) than the other. So it will reject null hypothesis H_0 .

4. Checking the assumptions:

-What are the assumptions?

-Do they hold?

Linearity: The relationship between the independent variables and the dependent variable should be linear. 1 nm 2 m

Independence of Errors: The errors should be independent of each other. 1 m 2 nm

Homoscedasticity: The variance of the errors should be constant across all levels of the independent variables. 1 nm, 2 m

Normality of Errors: The errors should follow a normal distribution. 1 m 2 m

Multicollinearity: Independent variables should not be highly correlated with each other. 1 m 2 m

Now, let's discuss whether these assumptions hold for your models (Model 1 and Model 2):

1) Model 1: Linearity: independent variables against the dependent variable in your data is NOT MET because residuals did not randomly scatter around zero for each independent variable.

Independence of Errors: The residuals are randomly scattered around zero, this assumption is met.

Homoscedasticity: The spread of residuals is not consistent across all levels of the fitted values, homoscedasticity is NOT Met.

Normality of Errors: The residuals are approximately normally distributed, this assumption is met.

Multicollinearity: For Model 1, you can calculate the VIF for each independent variable to check for multicollinearity it is >5 so it is met


2) Model 2: Linearity: independent variables against the dependent variable in your data is not MET because residuals randomly scattered around zero for each independent variable.

Independence of Errors: The residuals are not randomly scattered around zero, this assumption is NOT Met.

Homoscedasticity: The spread of residuals is consistent across all levels of the fitted values, homoscedasticity is met.

Normality of Errors: The residuals are approximately normally distributed, this assumption is met.

Multicollinearity: For Model 2, we can calculate the VIF for each independent variable to check for multicollinearity it is >5 so it is met

 ##your code goes here

```
#Model 1
```

```
model1 = sm.OLS(df2['price'], df2.drop(columns=['price']))
```

```
output1 = model1.fit()
```

```
# residuals calculation
```

```
residuals1 = output1.resid
```

```
# scatterplots
```

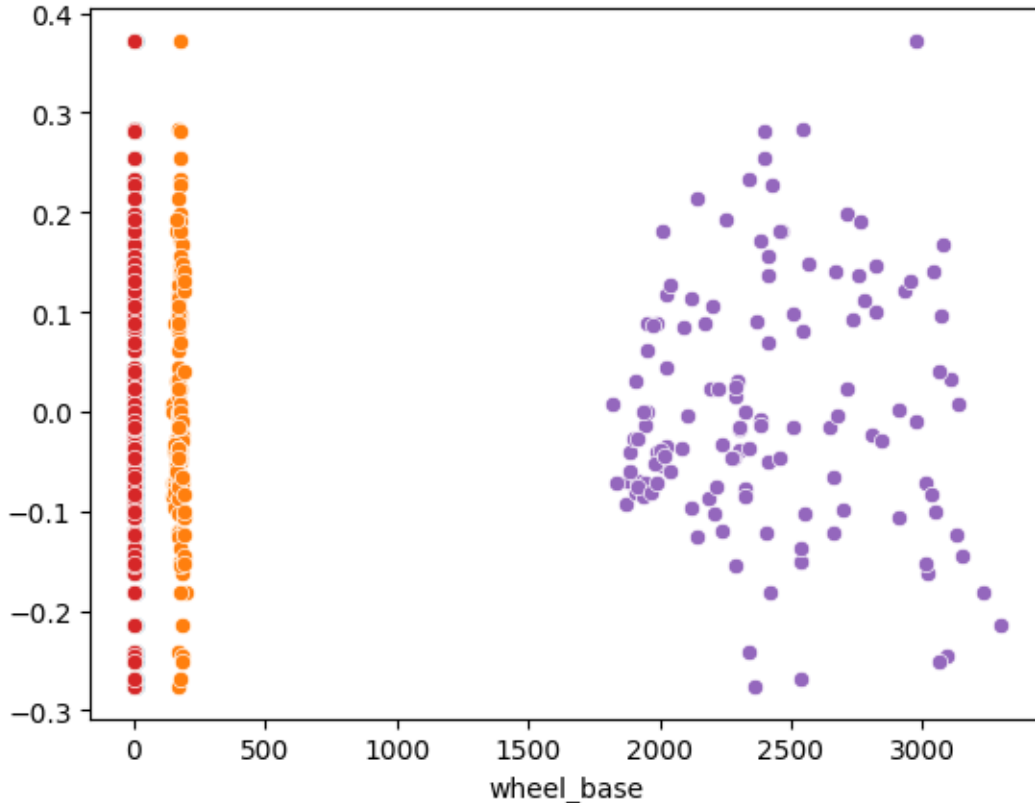
```
for col in df2.columns:
```

```
    if col != 'price':
```



```
sns.scatterplot(x=df2[col], y=residuals1)
```

★ OUTPUT:-

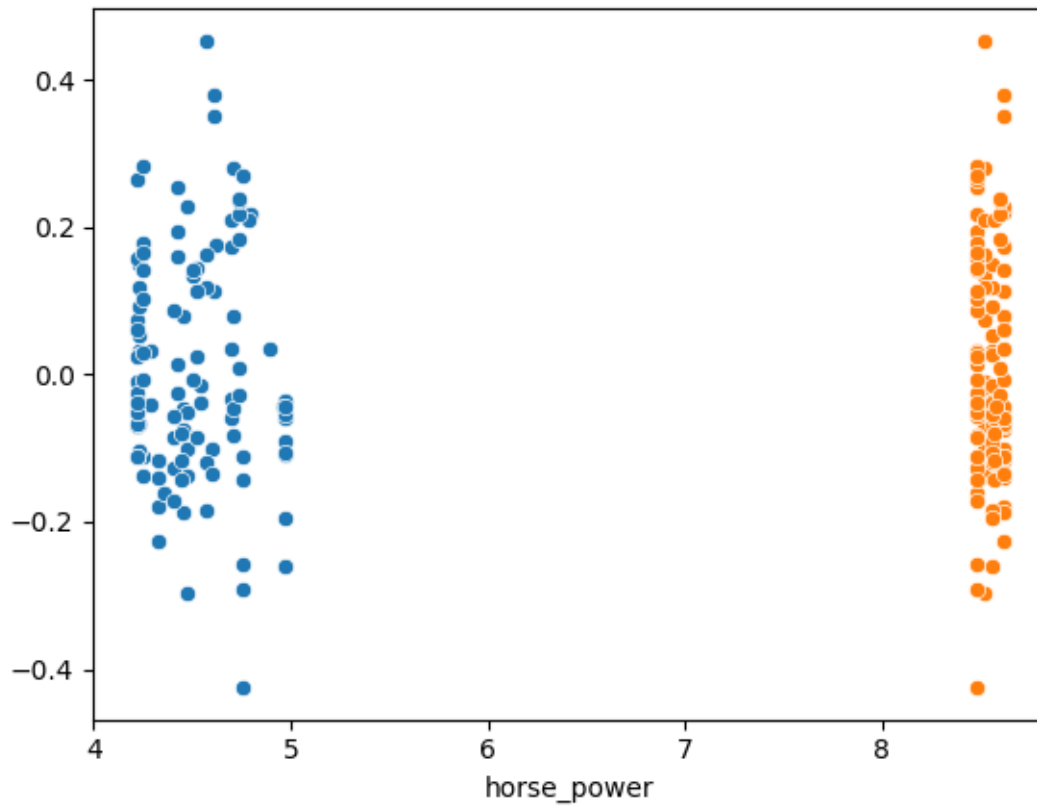


➡ # Model 2

```
model2 = sm.OLS(df2['price'], df2[['horse_power', 'peak_rpm']])  
output2 = model2.fit()
```

```
# Calculate residuals  
residuals2 = output2.resid
```

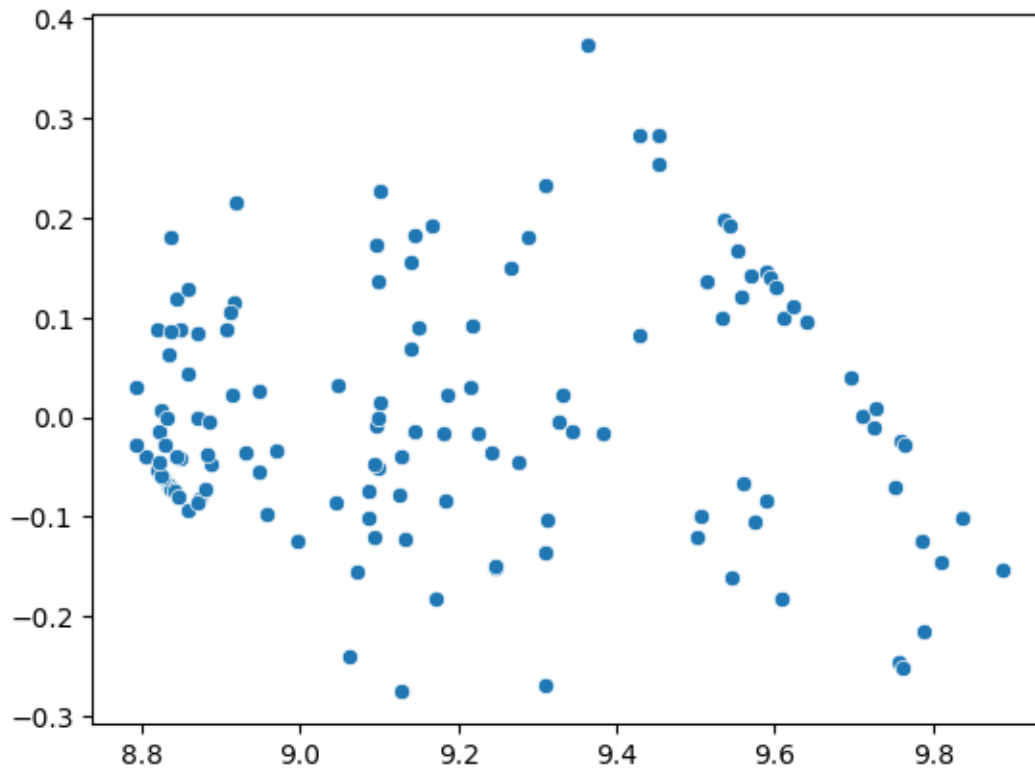
```
# Create scatterplots  
for col in df2[['horse_power', 'peak_rpm']]:  
    sns.scatterplot(x=df2[col], y=residuals2)
```



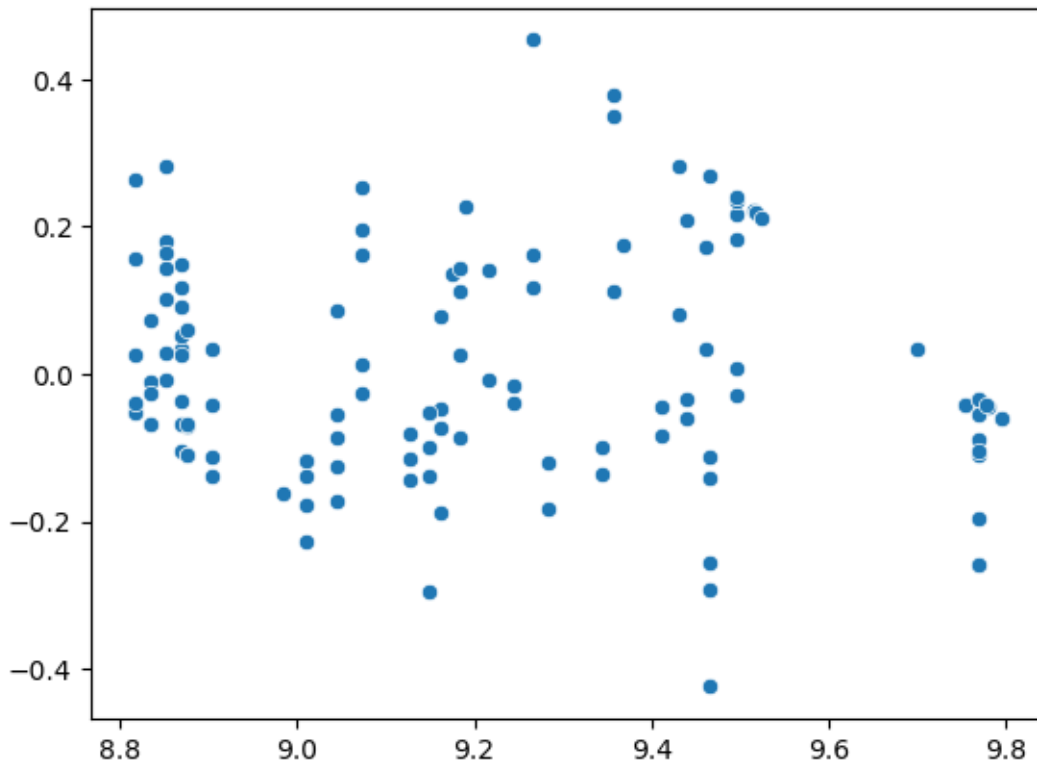
```
sns.scatterplot(x=output1.fittedvalues, y=residuals1)
```




OUTPUT:-



➡ `sns.scatterplot(x=output2.fittedvalues, y=residuals2)`



 `plt.hist(residuals1, bins=20, alpha=0.5, label='Model 1')`

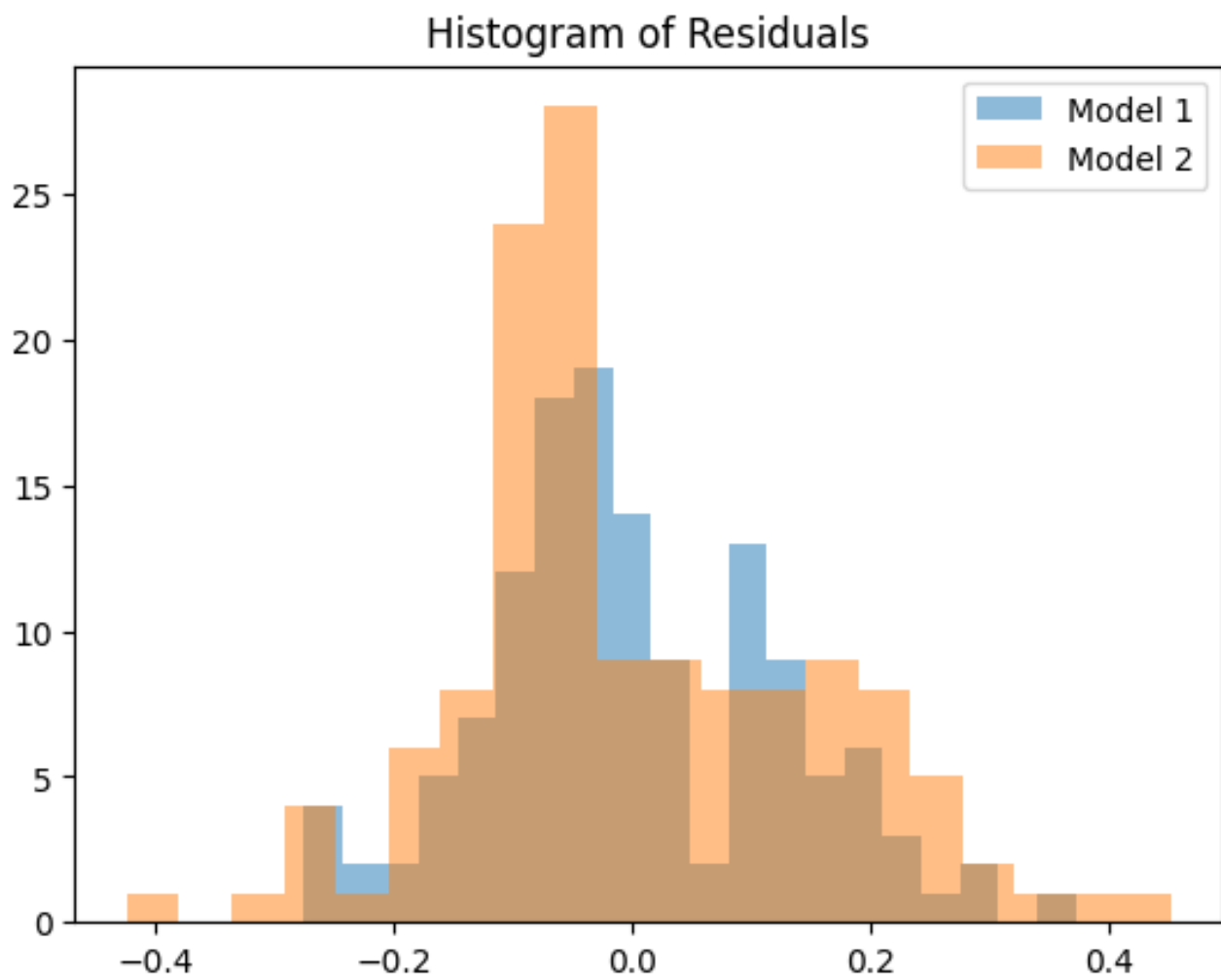
```
plt.hist(residuals2, bins=20, alpha=0.5, label='Model 2')
plt.legend()
plt.title('Histogram of Residuals')
plt.show()
```

```
stats.probplot(residuals1, dist="norm", plot=plt)
plt.title("Q-Q Plot for Model 1 Residuals")
plt.show()
```

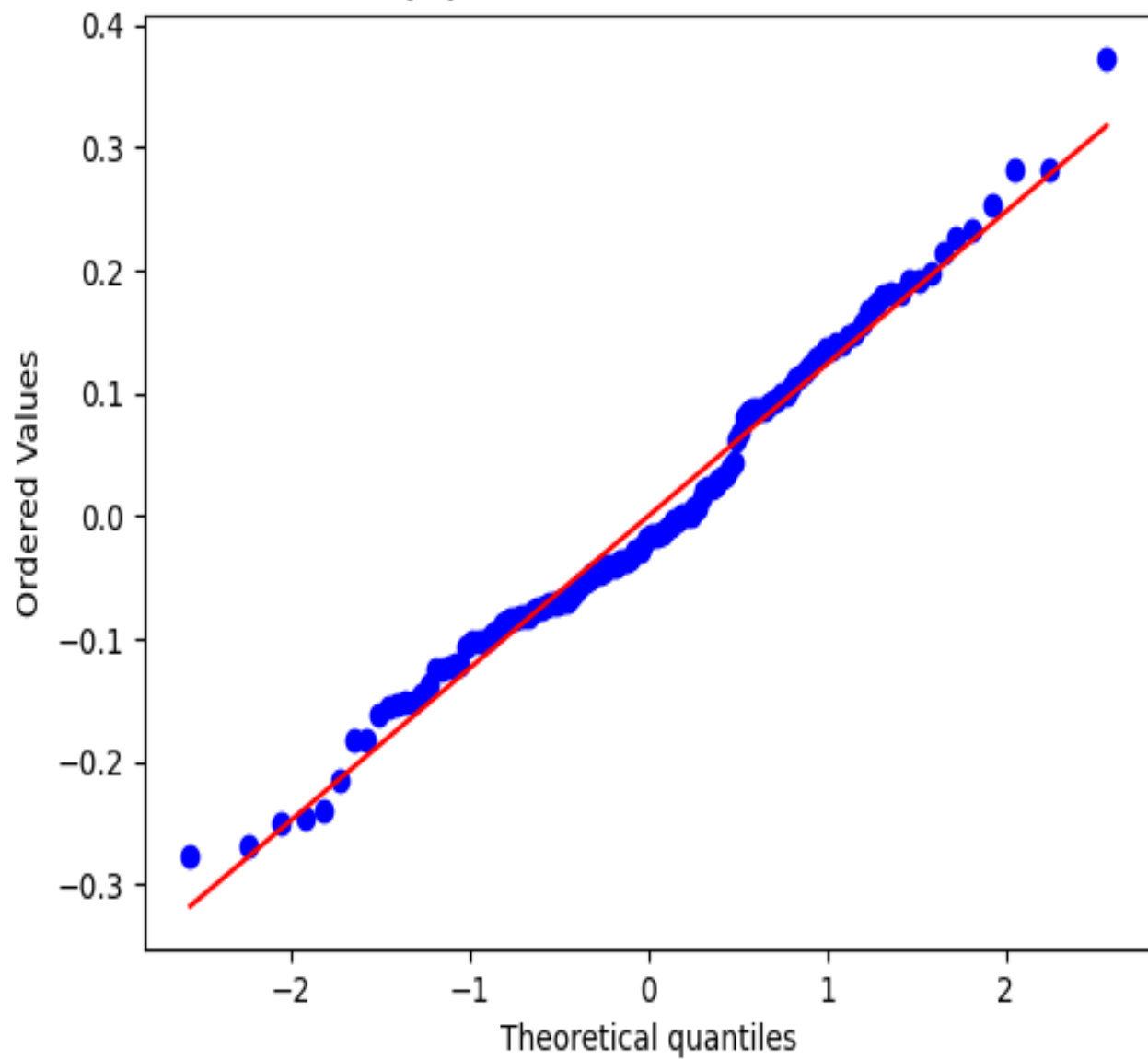
```
stats.probplot(residuals2, dist="norm", plot=plt)
plt.title("Q-Q Plot for Model 2 Residuals")
```

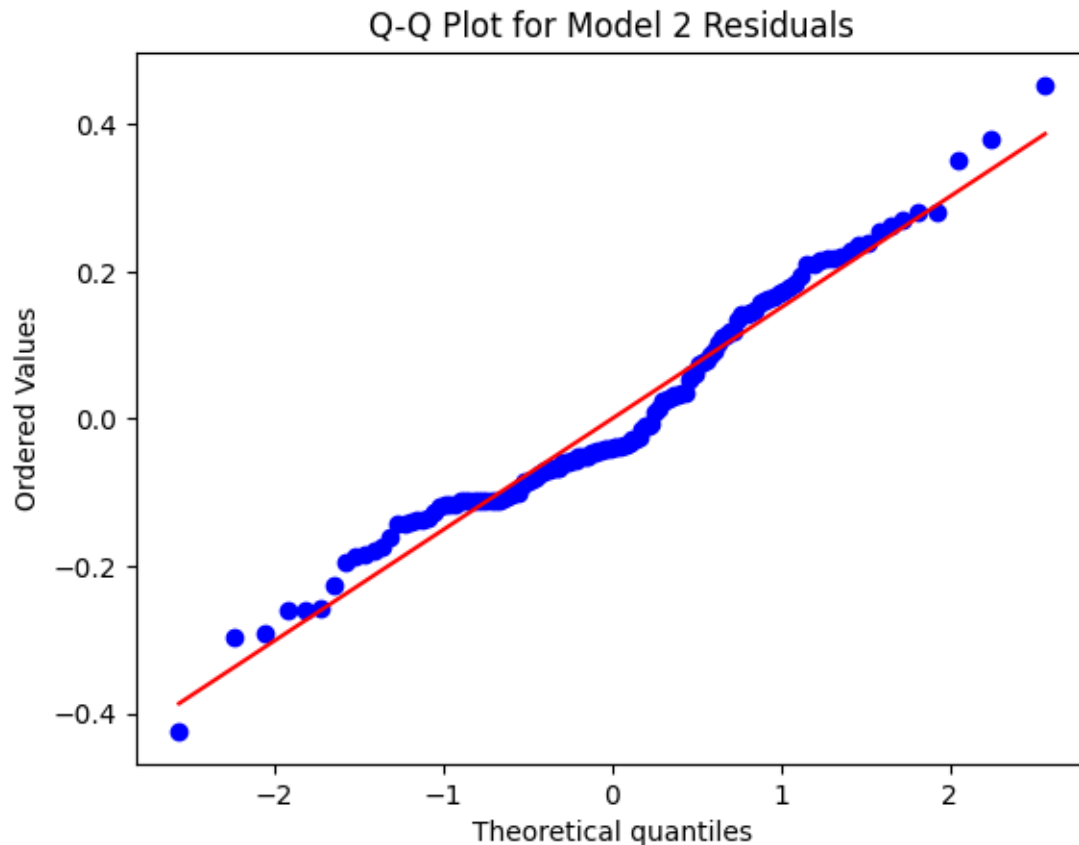
```
plt.show()
```

★ OUTPUT:-



Q-Q Plot for Model 1 Residuals





5. Is there Multicollinearity in your data?

➡ ##your code goes here

```
# VIF for Model 1
```

```
X1 = df2.drop(columns=['price'])
```

```
vif = pd.DataFrame()
```

```
vif["column_Variable"] = X1.columns
```

```
vif["VIF1"] = [variance_inflation_factor(X1.values, i) for i in  
range(X1.shape[1])]
```

```
print(vif)
```

★ OUTPUT:- column_Variable VIF1

0	wheel_base	66078.792478
1	length	1637.090800
2	width	93611.623286
3	heights	22684.724931
4	curb_weight	419.278980
5	engine_size	9570.069438
6	bore	553.682819
7	stroke	865.094873
8	comprassion	7657.538651
9	horse_power	4197.806191
10	peak_rpm	47303.164884
11	city_mpg	8733.109789
12	highway_mpg	9227.241974
13	fuel_type_gas	NaN

➡ # VIF for Model 2

```
X2 = df2[['horse_power', 'peak_rpm']]
vif2 = pd.DataFrame()
vif2["column_Variable"] = X2.columns
vif2["VIF2"] = [variance_inflation_factor(X2.values, i) for i in
range(X2.shape[1])]
print(vif2)
```

★ OUTPUT:-

column_Variable	VIF2
-----------------	------

0 horse_power 322.063696

1 peak_rpm 322.063696

FINAL RESULT :- Yes there is strong Multicollinearity in the data because vif and vif2 is >5