USED CARS PRICE PREDICTION USING MACHINE LEARNING

Used Cars Price Data Prediction

Contents

1. **Data Collection**
2. **Data Exploration**
3. **Data Preprocessing**
4. **Feature Engineering**
5. **Model Training**
6. **Model Assessment**

Introduction

Vehicles with one or more previous retail owners are referred to as used cars, this type of car is quite popular in Saudi Arabia as Saudi Arabia is the largest automotive market in the Gulf region with a population estimated to be 34.14 million, the latest annual surveys conducted by relevant authorities in Saudi Arabia assure that most Saudi families now have at least two cars, especially since women have now been permitted to drive and work, which has resulted in high demand for cars. Cars are employed for a wide range of activities, including daily activities due to the lack of public transportation and the unbearable outside weather. Furthermore, the VAT has been raised in the market from 5% to 15% and the manufacturer sets a fixed price of new cars becasue government incurring some additional costs on them in the form of taxes, making the budget one of the most significant constraints. As a consequence, Customers spend the majority of their time looking for a car, and owners change their cars every 2-4 years for all of the reasons stated above and the demand on the used car market has increased as the VAT does not affect this type of market since it is a peer-to-peer market.

As the cost of a used car is determined by a number of factors and features, people trying to buy a used car usually have difficulty finding one that fits their budget as well as determines the price of a particular used car.

In this notebook, we investigate this issue and construct a supervised prediction model (based on machine learning techniques) to enable the customer estimating the price of a used car.

The dataset is collected from Kaggle (https://www.kaggle.com/code/abdelrahmankhalil/audi-car-price-prediction-95-score/data) that contains information about all the features that could help predict the price for a specific Audi car model. By using and utilizing this data, we can get a great deal on purchasing a used car and how much the customer should pay for a specific type of car, especially since sellers can take advantage of this high demand to request excessive prices.

Problem Statement

The used cars market is become a large and important market in most regions. More cars are being sold than ever before. However, due to financial and budget constraints and the high cost of new cars, the sale of used cars is expanding at an exponential rate and most people prefer to buy the used cars. As a result, car owners (sellers) often take advantage of this situation by listing exaggerated prices. As a result, the need for such a predictive model to predict and assign a reasonable price to a vehicle based on its attributes that helps the customer effectively determine if the posted price is worth paying for the wanted car by taking several attributes into consideration.

Part 1: Data Collection

First, we collect the data about Audi used cars, do exploratory data analysis (EDA) trying to identify the important features that reflect the price.

Step 1: Import important libraries

Before starting our EDA, we need to import some tools, libraries, and modules to make our EDA process seamless.

Here are some of the most important libraries that will be used during the building of the end-to-end machine learning pipeline.

1. `Pandas`: This will be useful for data analysis and manipulation.

2. `numpy`: This allows to perform a wide range of mathematical operations on numerical arrays.

3. `Sklearn`: This will have an impact on our machine learning models by allowing for different functions to be used within the model.

4. `matplotlib`, `seaborn`: Visualize our results.

5. `mpl_toolkits`: To improve 3D visualization.

[1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib

import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
import mpl_toolkits.mplot3d

import sklearn
import sklearn.tree
import sklearn.metrics
import sklearn.ensemble

import sklearn.preprocessing
import sklearn.linear_model
import sklearn.model_selection

from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.model_selection import  train_test_split

import warnings
%matplotlib inline
```

By the help of `warning` module, `simplefilter` function can be used by passing the action that needs to be performed in the upcoming warning to control them by passing `ignore`

[2]:
```
warnings.simplefilter(action = "ignore")
```

Step 2: Load the data

`CSV file` is the most common file type to store data.

Data from the CSV file will be loaded into the system and stored in a data frame to help access the dataset easily and understand the structure of the dataset using the `read csv()` function with the dataset file's directory as a parameter.

Below, basic operations will be performed to check what the data consists of.

- **`Head of the dataset.`**
- **`Shape of the dataset.`**
- **`info of the dataset.`**
- **`Summary of the entire dataset.`**

[3]:
```
df = pd.read_csv("../datasets/audi.csv")
```

Start exploring dataset and its features by printing the first five rows from our data frame to have a general view of the selected dataset by performing the following command.

[4]:
```
df.head()
```

[4]:

	model	year	price	transmission	mileage	fuelType	tax	mpg	engineSize
0	A1	2017	12500	Manual	15735	Petrol	150	55.4	1.4
1	A6	2016	16500	Automatic	36203	Diesel	20	64.2	2.0
2	A1	2016	11000	Manual	29946	Petrol	30	55.4	1.4
3	A4	2017	16800	Automatic	25952	Diesel	145	67.3	2.0
4	A3	2019	17300	Manual	1998	Petrol	145	49.6	1.0

From the names of the columns, it can be observed that the data is actually about Audi car features and its price. Let's check the last five rows of the dataset by using the following command.

[5]:
```
df.tail()
```

[5]:

	model	year	price	transmission	mileage	fuelType	tax	mpg	engineSize
10663	A3	2020	16999	Manual	4018	Petrol	145	49.6	1.0
10664	A3	2020	16999	Manual	1978	Petrol	150	49.6	1.0
10665	A3	2020	17199	Manual	609	Petrol	150	49.6	1.0
10666	Q3	2017	19499	Automatic	8646	Petrol	150	47.9	1.4
10667	Q3	2016	15999	Manual	11855	Petrol	150	47.9	1.4

The column **`Price Column`** is our **`Target variable`**.

Now, start implementing specific steps and techniques to explore and find the best features in the data frame.

Check for the dataset dimension and the number of features and attributes that exist by using **`shape`** function to find out how many rows and columns there are.

[6]:
```
print("Dataset Rows:", df.shape[0])
print("Dataset Columns:", df.shape[1])
```

```
Dataset Rows: 10668
Dataset Columns: 9
```

The dataset has a total of 10668 observation rows and a total of 9 variable columns with different data types.

We can't assume that all the data was loaded into the correct data types, so types will be used to check the data types for all the data. So let's find out what each column's data type is by using the **df.dtypes** attribute.

[7]:
```
df.dtypes
```

[7]:
```
model            object
year              int64
price             int64
transmission     object
mileage           int64
fuelType         object
tax               int64
mpg             float64
engineSize      float64
dtype: object
```

Insights:

As can be seen from the output result, data types for dataset columns have been successfully returned, and it can be observed that all the columns are of the correct data type. No changes need to be made.

Print a brief summary of the data frame that shows the column names, datatype, memory usage, and the number of cells in each column, as well as the range index.

[8]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10668 entries, 0 to 10667
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   model         10668 non-null  object
 1   year          10668 non-null  int64
 2   price         10668 non-null  int64
 3   transmission  10668 non-null  object
 4   mileage       10668 non-null  int64
 5   fuelType      10668 non-null  object
 6   tax           10668 non-null  int64
 7   mpg           10668 non-null  float64
 8   engineSize    10668 non-null  float64
dtypes: float64(2), int64(4), object(3)
memory usage: 750.2+ KB
```

Observations

- There is `six numerical data` (2 as floats, 4 as integers) and `three categorical values` (object) in the data frame.
- The data frame columns consume 750.2+ KB of total memory.
- The data frame has `10668 non-null values`, which means there are no null/missing values.

Description of dataframe attributes:

Descriptions and details of the dataset are described in table below.

Attribute Name	Description
Model	The model of the car
Year	Car model year
Price	Price of the car
Transmission	Most important feature of the car (Automatic / Manual)
Mileage	Kilometer run by the car
Fuel_Type	The fuel type that the car used
Tax	Car tax

Attribute Name	Description
mpg	Mile per galloon, the represent how much the car is supposed to travel or move by 1 galloon of fuel
EngineSize	Engine size of the car

For the puprose of this Notebook, the target variable that we are attempting to predict is the `price`, whereas the other variables are the `attributes.`

Part 2: Data Exploration

Visual exploration was used to acquire ideas about the model that might be applied to the data as well as understand the distribution in the data set by using bar charts, box plot, and distribution graphs to explore the features and relationships between the target label and other features.

Step 1: Data Inspection

In this part, some techniques will be used to drive deep into the dataset in order to describe the characterizations of the dataset, such as missing values, unique values, duplicates and outliers to strategize and identify the relationships between the target variable and different variables, as well as defining the variables' distribution for better understanding the data and get better insights to effectively build a regressor model that fits the data.

`1.1 Null Values`

Missing values can cause problems and errors and directly affect what we need to do in the data cleaning step as well as affect the final insights. Therefore, in this section, the main step is to check for any missing (NULL) values to decide what actions will be taken on them if there are any. However, this will be much easier to interpret and more comprehensible when it comes to quickly identifying the details of missing values, which leads to having a clear reason for the actions that need to be performed to treat the missing values.

Check for any missing values separately by using **isnull()** functions with **sum()** to return the number of missing values.

[9]:
```
print("Number of empty Rows:")
print(df.isnull().any(axis=1).sum())

print("Number of empty Columns:")
print(df.isnull().any(axis=0).sum())
```

```
Number of empty Rows:
0
Number of empty Columns:
0
```

Luckily, the dataset is cleaned from any missing values.

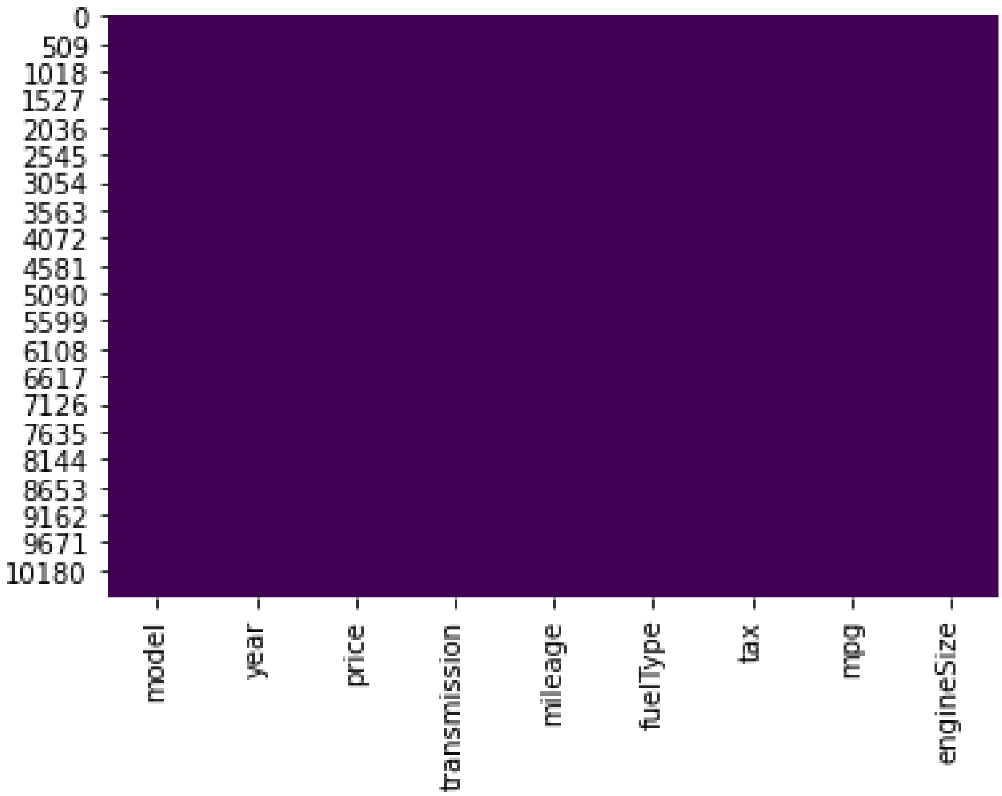
For a double check.ins>

Use `heatmap` from seaborn library to visualize the missing values in each variable and pass the following parameters.

- `Data`: Our data frame.
- `cbar`: To hide the color bar of the heatmap, set it to ”false‘.
- `cmap`: The color of the filled area corresponding to the missing values.

[10]:
```
sns.heatmap(df.isnull(), cbar=False, cmap="viridis")
```

[10]: <AxesSubplot:>



As the heatmap conclude, the dataset is clear of any explicit missing values.

1.2 Unique Values

As the dataset is consists of two types of data, categorical and numerical data, we will look for the unique values in both types of data individually by selecting first the categorical data and using the `nunique` function to get the number of unique values, as well as the same step for the numerical type of data and use `format` function to format the specified value.

[11]:
```
categorical_data = df.select_dtypes(["object"]).columns
```

[12]:
```
for col in categorical_data:
    print("{} : {} unique value(s)".format(col, df[col].nunique()))
```

```
model : 26 unique value(s)
transmission : 3 unique value(s)
fuelType : 3 unique value(s)
```

Observing the highest number of categorical unique values in `model` column with 26 unique values, let's explore the unique values by defining the model column and using "unique()' as follows.

[13]:
```
df["model"].unique()
```

[13]:
```
array([' A1', ' A6', ' A4', ' A3', ' Q3', ' Q5', ' A5', ' S4', ' Q2',
       ' A7', ' TT', ' Q7', ' RS6', ' RS3', ' A8', ' Q8', ' RS4', ' RS5',
       ' R8', ' SQ5', ' S8', ' SQ7', ' S3', ' S5', ' A2', ' RS7'],
      dtype=object)
```

print out the number of occurrences for each value and sort them ascending by using `value_counts`.

[14]:
```
df["model"].value_counts()
```

[14]:
```
 A3      1929
 Q3      1417
 A4      1381
 A1      1347
 A5       882
 Q5       877
 Q2       822
 A6       748
 Q7       397
 TT       336
 A7       122
 A8       118
 Q8        69
 RS6       39
 RS3       33
 RS4       31
 RS5       29
 R8        28
 S3        18
 SQ5       16
 S4        12
 SQ7        8
 S8         4
 S5         3
 A2         1
 RS7        1
Name: model, dtype: int64
```

[15]:
```
numerical_data = df.select_dtypes(["int", "float"]).columns
```

[16]:
```
for col in numerical_data:
    print("{} : {} unique value(s)".format(col, df[col].nunique()))
```

```
year : 21 unique value(s)
price : 3260 unique value(s)
mileage : 7725 unique value(s)
tax : 37 unique value(s)
mpg : 104 unique value(s)
engineSize : 19 unique value(s)
```

[18]:
```
df["mileage"].unique()
```

[18]:
```
array([15735, 36203, 29946, …,  4018,  1978,  8646])
```

1.3 Duplicates

Duplicate values in a dataset need to be figured out and handled since having the same records within the dataset would not help the buyer make the best decisions. So it's significant to practice in this step to check for the number of duplicates and remove any duplicates by deciding to keep only the first record or last, or to not keep any and drop all the duplicates.

Check for the number of duplicated values as follows: * Use.duplicated() to count the number of duplicated rows in the pandas data frame.

[19]:
```
print("Total No. of duplicated rows:{}".format(df.duplicated().sum()))
```

```
Total No. of duplicated rows:103
```

103 duplicated values need to be handled. Let's print out the duplicated rows in the dataset.

[20]:
```
df[df.duplicated()]
```

[20]:

	model	year	price	transmission	mileage	fuelType	tax	mpg	engineSize
273	Q3	2019	34485	Automatic	10	Diesel	145	47.1	2.0
764	Q2	2019	22495	Manual	1000	Diesel	145	49.6	1.6
784	Q3	2015	13995	Manual	35446	Diesel	145	54.3	2.0
967	Q5	2019	31998	Semi-Auto	100	Petrol	145	33.2	2.0
990	Q2	2019	22495	Manual	1000	Diesel	145	49.6	1.6
…	…	…	…	…	…	…	…	…	…
9508	A4	2019	26990	Automatic	2250	Diesel	145	50.4	2.0
9521	Q3	2019	26990	Manual	10	Petrol	145	40.9	1.5
9529	Q5	2019	44990	Automatic	10	Diesel	145	36.2	2.0
9550	Q3	2019	29995	Manual	10	Petrol	145	39.8	1.5
9597	Q3	2019	28490	Manual	10	Diesel	145	42.8	2.0

[103 rows x 9 columns]

Using Pandas' `describe` function, display a statistical summary for the dataset.

[21]:
```
df.describe()
```

[21]:

	year	price	mileage	tax	mpg \
count	10668.000000	10668.000000	10668.000000	10668.000000	10668.000000
mean	2017.100675	22896.685039	24827.244001	126.011436	50.770022
std	2.167494	11714.841888	23505.257205	67.170294	12.949782
min	1997.000000	1490.000000	1.000000	0.000000	18.900000
25%	2016.000000	15130.750000	5968.750000	125.000000	40.900000
50%	2017.000000	20200.000000	19000.000000	145.000000	49.600000
75%	2019.000000	27990.000000	36464.500000	145.000000	58.900000
max	2020.000000	145000.000000	323000.000000	580.000000	188.300000

```
        engineSize
count  10668.000000
mean       1.930709
std        0.602957
min        0.000000
25%        1.500000
50%        2.000000
75%        2.000000
max        6.300000
```

As the statistical summary of the dataset shows, some statistical data for numerical values like (percentile, mean, max, min, and std), There are some outliers identified in some columns where there is a difference between the max value and the third quartile of 75%.

1.4 Outliers

Outliers are data points that differ significantly from the remainder of the dataset's data points. They might appear for a variety of causes, including errors made during measuring or entering data. However, in certain cases, outliers are critical and must be detected and managed, while in others, they aren't as critical and can be ignored. According to our dataset.

To find outliers in the data frame:

- **Visualise the outliers.**
- **Define a function to find the outliers in the data frame.**
- **Apply some strategies to handle the outliers, either by removing them or accepting them.**

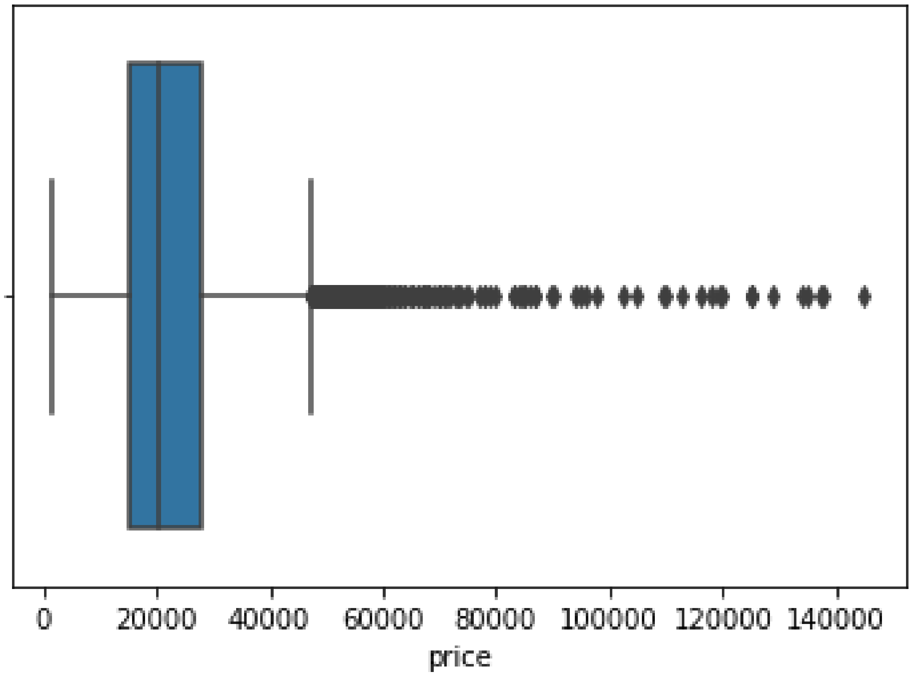
I will start by visualizing the outliers following these techniques:

1. Create an array of quantitative data that will be checked for outliers.

2. use the `boxplot` from the `seaborn` library to visualize the distribution of the points in the numerical columns and pass each column by its index.

[22]:
```
num_cols = ["price", "mileage", "tax", "mpg", "engineSize"]
```

[23]:
```
sns.boxplot(df[num_cols[0]]);
```



Insights:

As above, the box plot displays the price distribution based on five main numbers:

1. The `first quartile (Q1), the 25%th percentile on the left side of the median in the box (the number between the`smallest`and the`median' value.

2. The `third quartile (Q3), the 75%th percentile, is on the right side of the median in the box ( the number that is between the`highest`and`median' value).

3. `Median` which is the middle value of the dataset with a (50%th percentile), which is the line in the center of the box.

4. Interquartile range (IQR) starting from Q1 to Q3 (25th to 75th)

___5.___Minimum (Q1-1.5*IQR) value

6. Highest value (Q3 + 1.5*IQR)

7. Whiskers are the lines that connect the Q3 and maximum value, as well as the Q1 and minimum value.

Based on the boxplot numbers and as the boxplot shown:

1. The first quartile (Q1) is around 15,000.

2. The Third Quartile Q3 is approximately 28,000.

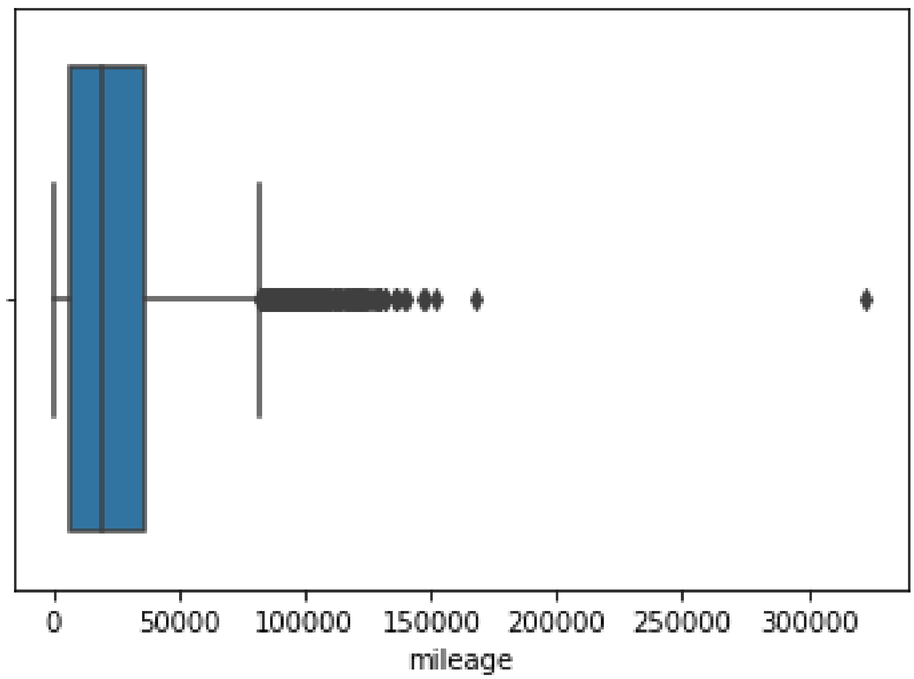
3. "Maximum Value" is around 50,000.

4. All the prices values lies in the range of 1000a and 50,000

- Clearly, it's observed that there are a lot of outliers in the price feature since the dots are overlapped in the box blot.
- The maximum outliers were identified because the maximum price value is approximately 145,000 versus the Q3 rice value of approximately 28,000.
- According to these outliers, they are acceptable outliers because price variation in cars makes sense because there is a well-known rich level that only buys luxuary cars with high prices.

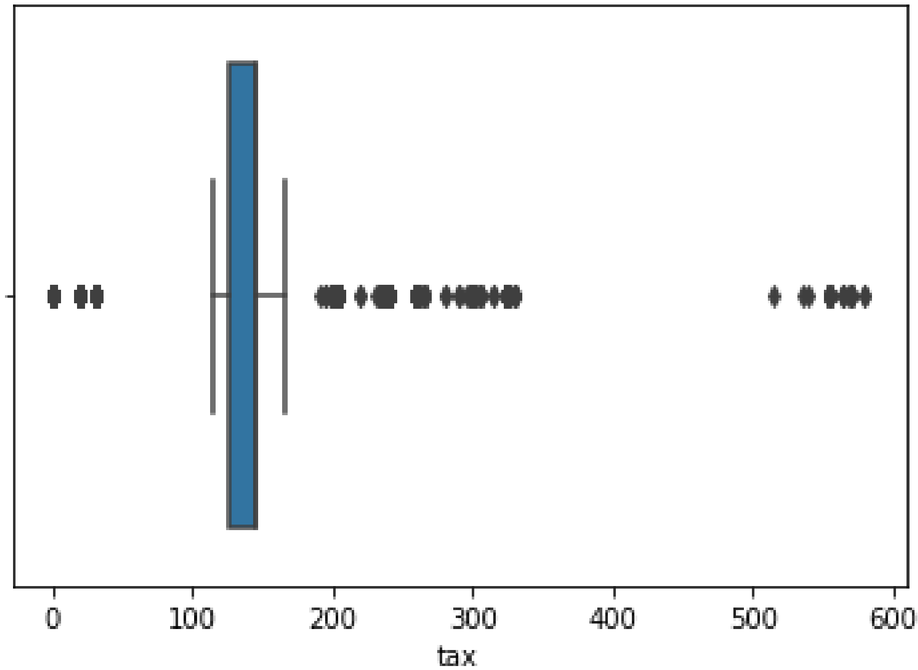
Let's see what it looks like for the second feature.

[24]:
```
sns.boxplot(df[num_cols[1]]);
```



Clearly, there is a maximum outlier in the mileage column with various values, such as a value near 350000.

[25]:
```
sns.boxplot(df[num_cols[2]]);
```



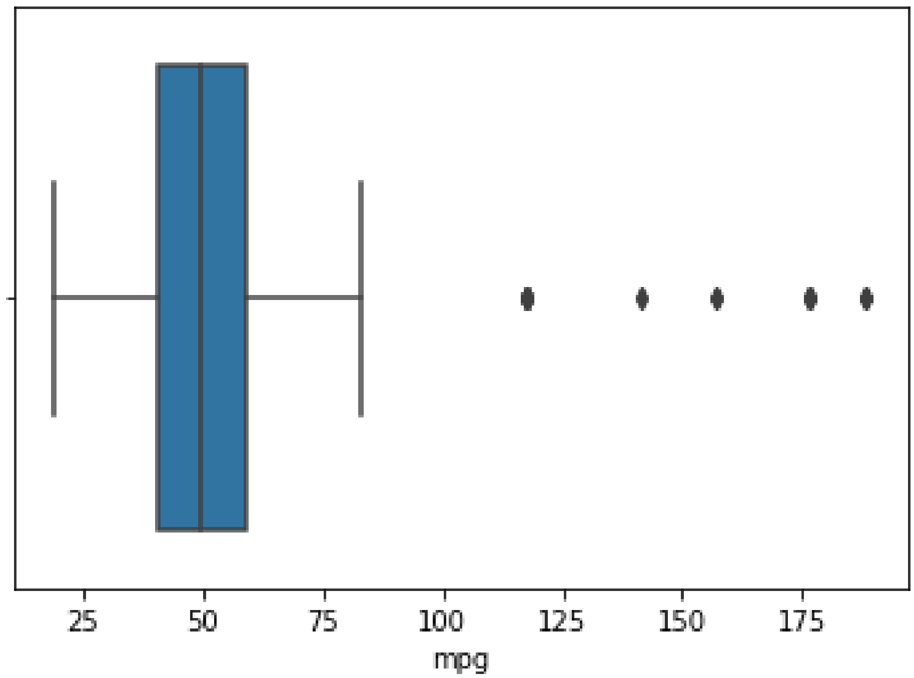
Minimum value is around 95.

Maximum value around 170.

When it comes to the tax feature, maximum and minimum outliers were found, where there was a minimum value of 0. That means there are implicit values that appear in these features that need to be handled while cleaning the data.

We will look at this point in the next sections.

[26]:
```
sns.boxplot(df[num_cols[3]]);
```

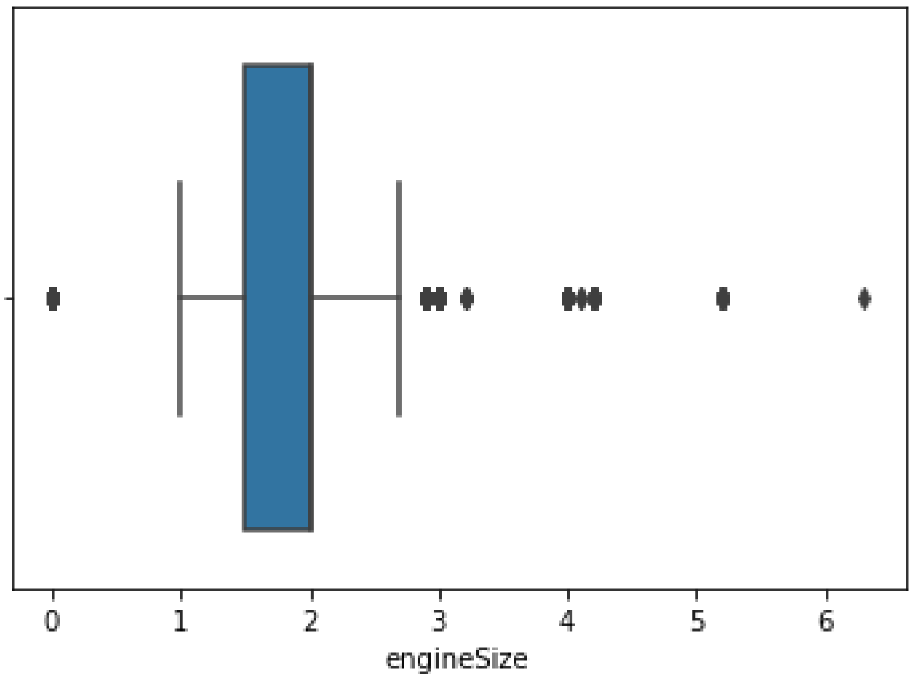


Fewer outliers were identified in the mpg feature based on the following values that appear in the box plot:

1. The **First Quartile Q1** is approximately 40.5.
2. The **Third Quartile Q3** value is around 58.
3. "Maximum Value" is around 80-90.

Some outliers are clearly defined, and some of these outliers appear multiple times. These maximum outliers range from 115 to 180.

[27]:
```
sns.boxplot(df[num_cols[4]]);
```



As well as for the last feature, there are also maximum and minimum outlier values, but according to our dataset, it makes sense, so we will keep the maximum outliers, whereas we will remove the minimum outliers.

To clearly have a deeper view of the outliers that we have in the dataset, we will define a function to find and return a list of indexes of outliers in a specific column and handle them by defining a function to get rid of the unwanted outliers (in the data preprocessing part).

The following steps will be used to determine the outliers :

1. Extract all the outliers.

2. Use the retrieved outliers' indexes to later remove them from the original data frame.

3. Define `find_outlier` function with 1 input argument x.

4. Calculate the first quartile, `Q1` by using `np.percentile` funtion in Numpy and the input x with the value of the percentile for Q1, which is 25.

5. Follow the same steps to calculate the Q3, but change the Q3 percentile value to 75.

3. Compute the interquartile range (IQR) where it is equal to the third quartile-first quartile using this formula (`IQR = Q3 - Q1`)

4. Use this formula to calculate the lower-quartile, which will yield all values less than Q1 (Q1–1.5*IQR), and the upper-quartile, which will yield all values greater than Q3 (Q3+1.5*IQR).

5. Store the index of all the values that are not in the lower quartile's range or upper quartile in a list object `outlier_list`.

6. Store the outlier values in another list based on the outlier index in the outlier list.

[34]:
```
def find_outliers(x):
    Q1 = np.percentile(x, 25)
    Q3 = np.percentile(x, 75)
    IQR = Q3 - Q1

    lower_quartile = Q1 - 1.5 * IQR
    upper_quartile = Q3 + 1.5 * IQR

    outliers_list = list(x.index[(x < lower_quartile) | (x > upper_quartile)])
    outliers_values = list(x[outliers_list])

    return outliers_list, outliers_values
```

Let's start by extracting the outliers in the mbg column as follows:

- Define the mpg_list and mpg_values objects so that the mpg_list object contains all the values indices that are above the minimum lower and upper quartiles, whereas the mpg_value object contains the outlier values for these indices.
- Sort them for a better understanding of the data.

[35]:
```
mpg_list, mpg_values = find_outliers(df["mpg"])
print(np.sort(mpg_values))
```

```
[117.7 117.7 117.7 117.7 117.7 117.7 117.7 117.7 117.7 117.7 117.7 117.7
 117.7 117.7 117.7 141.3 141.3 156.9 156.9 156.9 176.6 176.6 176.6 176.6
 176.6 176.6 176.6 176.6 188.3 188.3 188.3 188.3 188.3]
```

The above result indicates that there are a few outliers in this feature, and examining them by values makes sense because the data is being processed for the KSA used market, and these are actual metrics to keep in such a large metropolis that relies on personal cars for every daily activity.

[36]:
```
eng_list, eng_values = find_outliers(df["engineSize"])
print(np.sort(eng_values))
```

```
[0.  0.  0.  … 5.2 5.2 6.3]
```

[37]:
```
df['engineSize'].value_counts()
```

[37]:
```
2.0    5169
1.4    1594
3.0    1149
1.6     913
1.5     744
1.0     558
4.0     154
1.8     126
2.5      61
```

```
0.0    57
2.9    49
1.2    31
4.2    25
5.2    23
3.2     5
1.9     4
2.7     3
4.1     2
6.3     1
Name: engineSize, dtype: int64
```

Returning to engineSize, we've noticed that there are some implicit values that must be removed throughout the data cleaning process.

Step 2: Visualizing data points and features

We'll show some graphs and plots at this stage to acquire some key insights and correlations.

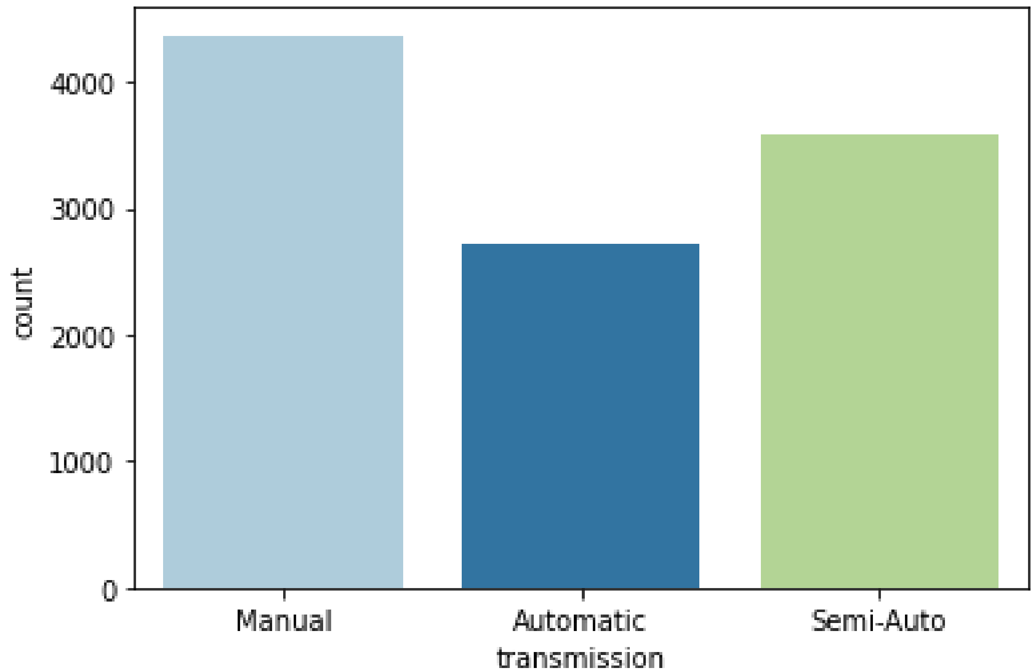
Let us start to explore the variables in-depth, examining the relationships between them and between features and target variables as well.

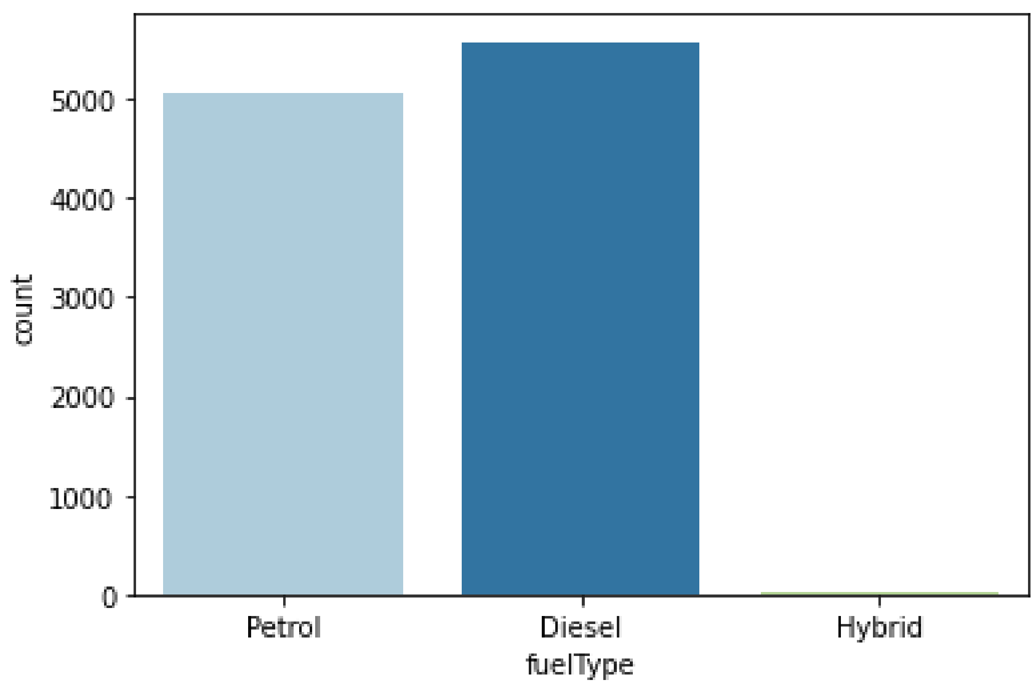
Starting by visualizing the categorical features using searborn's `countblox` as follows:

- Iterate over the categorical features array
- For plotting, pass the feature parameters, data frame, and palette name to set the colors for variables.
- Show the visualization.

[38]:
```
features = ["transmission", "fuelType"]
```

[39]:
```
for ft in features:
    sns.countplot(x=ft, data=df, palette="Paired")
    plt.show()
```





Observations were obtained

- It's clearly obvious that manual cars are the most purchased cars, followed by semi-auto and automatic cars coming last.

- In terms of fuel type, it reveals that diesel cars are the most in-demand, followed by petrol cars, and finally, the least amonunt of demand is given to the hybrid cars.

Relationship between the Price and other features

Let's have a look at and examine the relationship between features and target value in the dataset:

- Use `df.corr()` to find the relationship between the variables and the strength of this relationship among the features.
- Store this result in `corr object` to be used later on while visualizing the correlation.
- Use `sort_values` and pass the target variable to sort the result in `descending order`.

[40]:
```python
print("Best Correlated Features")
corr = df.corr()
corr.sort_values(["price"], ascending=False, inplace=True)
print(corr.price)
```

```
Best Correlated Features
price          1.000000
year           0.592581
engineSize     0.591262
tax            0.356157
mileage       -0.535357
mpg           -0.600334
Name: price, dtype: float64
```

Observations:

1. Year and engineSize are the two most correlated variables with the price.

2. The "tax" feature has not had that strong of a correlation with price.

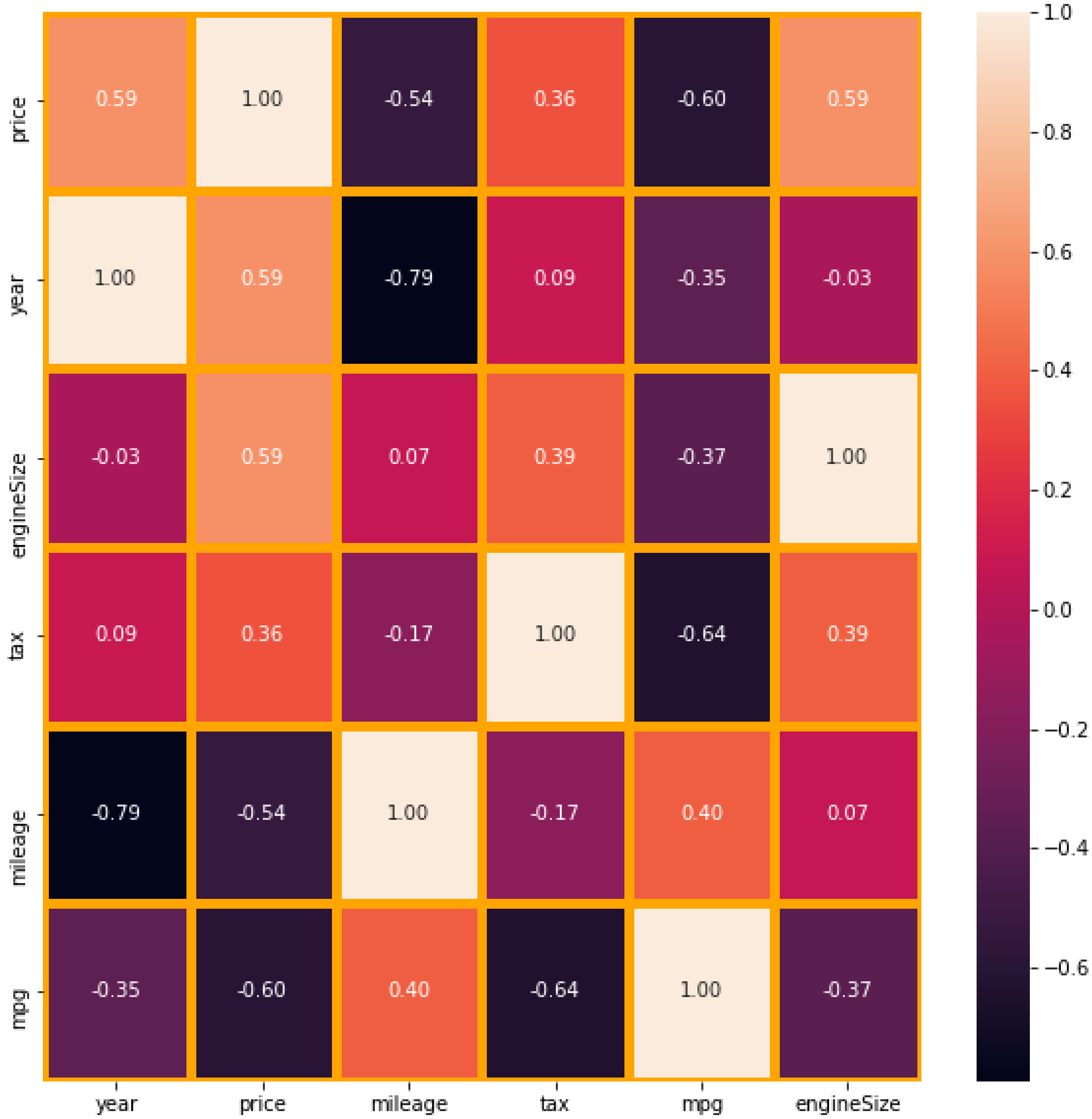
3. mileage and mpg have a negative correlation with price.

Plot the results by using `seaborn's heatmap` function and passing the following parameters:

- Correlation data.
- `annot`: If True, correlation values will be represented in each cell.
- `fmt`: This function formats the string that will be used to store annotation values.
- `linewidth`: the width of the dividing line between cells.
- `line color`: to choose the line color.

[41]:
```python
plt.figure(figsize=(10, 10))

sns.heatmap(corr, annot=True, fmt="0.2f", linewidth=5, linecolor="orange")
plt.show()
```



Observations

It is evident from the heatmap that features are highly correlated with each other and are dependent where the darker color denotes a high correlation where the ligheter color denotes low correlation between variables.

Positive Correlation:

- Positive and engine size have a positive and high correlation with price."
- Engine size is highly correlated with price and has a low correlation with mileage and tax.
- There is a positive correlation between mileage and mpg.

Negative Correlation:

- Year, price, tax, and engine size all have a negative correlation with MB.
- Mileage was inversely related to the year, price, and tax.
- Price has a negative correlation with the mileage.

Hence, all these features negatively affecting the price

As observed, diesel cars have the highest demand, while petrol cars, and hybrids, have the least fuel-type car demand.

Using `piechart` will explore the percentage of the fuel type demand.

Pie chart as it is a statistical graphic chart which divides the pie (circle) into multiple slices based on the given data to show the numerical proportion, as these types of charts are widely used in business and analysis, fuel type analysis will be analyzed by pie chart using the imported module `plotly graph object` following this process:

1. Create `figure object` that will be used to show the plot.
2. Call the `Figure` method of the `graph_object` submodule.
3. Create the plot `Pie` chart inside the figure object and pass the arguments:

* `labels`: Data to be displayed and passed by the column name to the fuel type.

* `values`: Show the plot based on the values in the price column.

`* `hole`: A hole is used to cut the pie chart into donut charts.`

```
[42]: fig = go.Figure(data=[go.Pie(labels=df["fuelType"], values=df["price"], hole=0.
       ↪4)])
      fig.show()
```

Observations

- **Diesel** is the most popular fuel type among customers, accounting for 53.1 percent of all requests.
- **Petrol** is the next most wanted with a percentage of **45.6%**
- **Hybrid** vehicles have the lowest proportion and are nearly unwelcome due to their low percentage of **0.35 percent** .

`Visualize the relationship between Year, Price and Kilometer-Driven by the car`

As Matplotlib is mainly used to plot 2D plotting graphs, `3D` visualization can be plotted by importing the submodule mplot3d `mpl_toolkits.mplot3d` as well as importing other libraries for plotting data, and following these steps to visualize the relationship as a 3D plot.

- Set the figure size.
- Pass the keyword `projection = 3d` to create and enable the three-dimensional axes.
- Use the most commonly used 3D plot, `scatter plot` by creating the plot object and creating the scatter plot by using the `scatter3D()` method and the 3 values to be plotted (year of the car, price of the car, and the kilometers driven by the car).
- Modify the plot's linewidth, edge color, and marker size.

- Set the 3 axes (x-axis: year, y-axis: price, z-axis: mileage)
- As the price is the target label, insights will be gotten from the scatter plot based on the price. By using 'fig. color bar we will set the label to price variable and shrink it for better visualization.

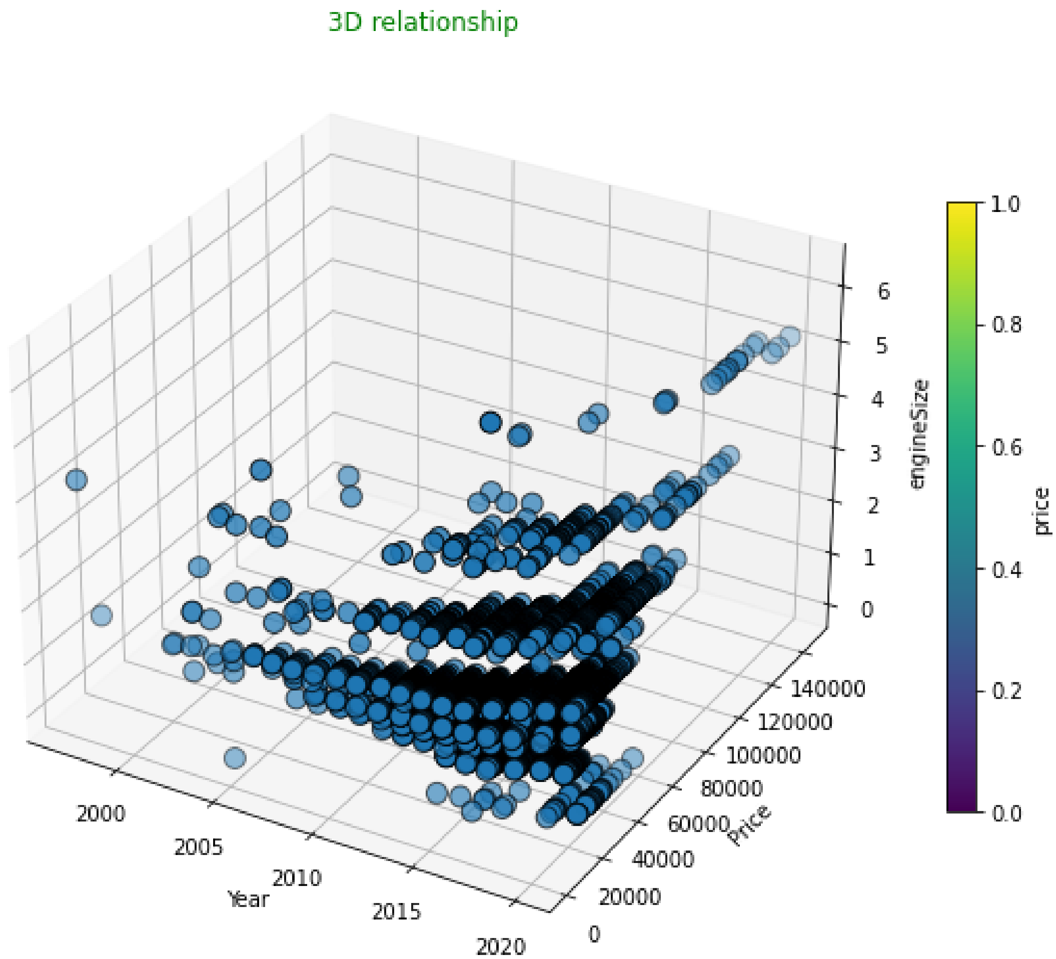
[43]:
```
fig = plt.figure(figsize=(10, 9))
axes = fig.gca(projection="3d")

plot = axes.scatter(
    df["year"],
    df["price"],
    df["engineSize"],
    s=100,
    linewidth=1,
    edgecolor="k",
)

axes.set_xlabel("Year")
axes.set_ylabel("Price")
axes.set_zlabel("engineSize")

fig_label = fig.colorbar(plot, shrink=0.6)
fig_label.set_label("price", fontsize=10)

plt.title("3D relationship", color="green")
plt.show()
```



Observations

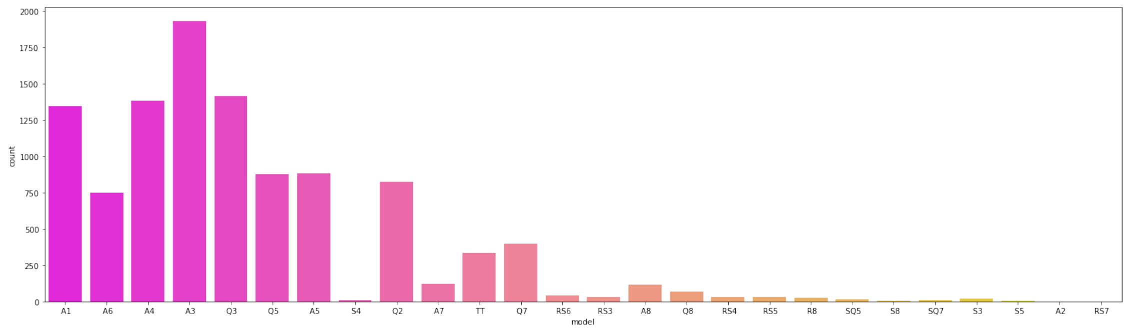
The 3D Scatter Plot helps explore and identify the strength and direction of the relationship between the 3 variables mentioned. As the price is predictor value, the scatter plot shows the following:

1. Some data points are far from the group of data points that can be considered outliers.
2. A strong and positive relationship between price and year, as the data points tend to rise in unison.
3. Datapoints in engine size tend to decline where the price and year data points tend to rise together, which reflects the negative relationship (negative correlation).

Visulaize the most wanted car model

[44]:
```
plt.figure(figsize=(25, 7))
sns.countplot("model", data=df, palette="spring")
```

[44]: <AxesSubplot:xlabel='model', ylabel='count'>



Insights:

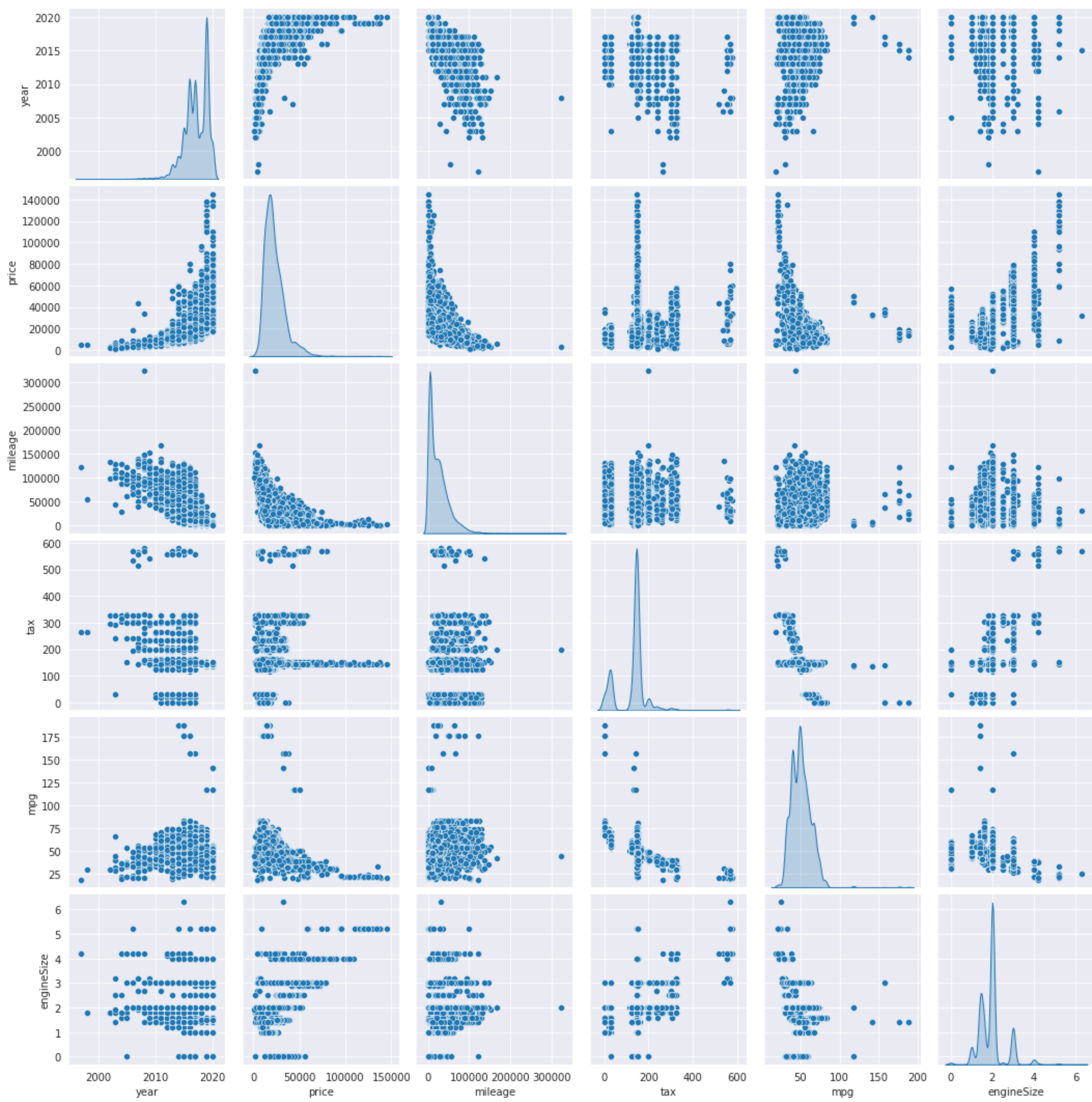
- The Audi A3 model is the most popular car model among customers, followed by the Audi Q3 model.

[45]:
```
sns.set_style("darkgrid")
```

Use the'seaborn library', the 'pairplot()' function, and the data frame to help understand the exploratory data by building an axis grid to present the data across the X and Y-axis for this machine learning project.

[46]:
```
sns.pairplot(df, diag_kind="kde")
```

[46]: <seaborn.axisgrid.PairGrid at 0x7fe8348a8730>



What sns.pairplot(df) has done is create different multiple figures based on the numerical data frame variables (year, price, mileage, tax, MPG, and engine size).

Taking a look at some figures

1. `Histogram figure` in the upper left corner of the graph that corresponds to `year` feature in the `Y-axis` as well as the `X-axis` and having a kind of histogram figure along the diagonal
2. `Scatter Plots` that show the relationship

`Relationship between Car Options and Price`

Create a figure and subplots.

- Set the figure size.

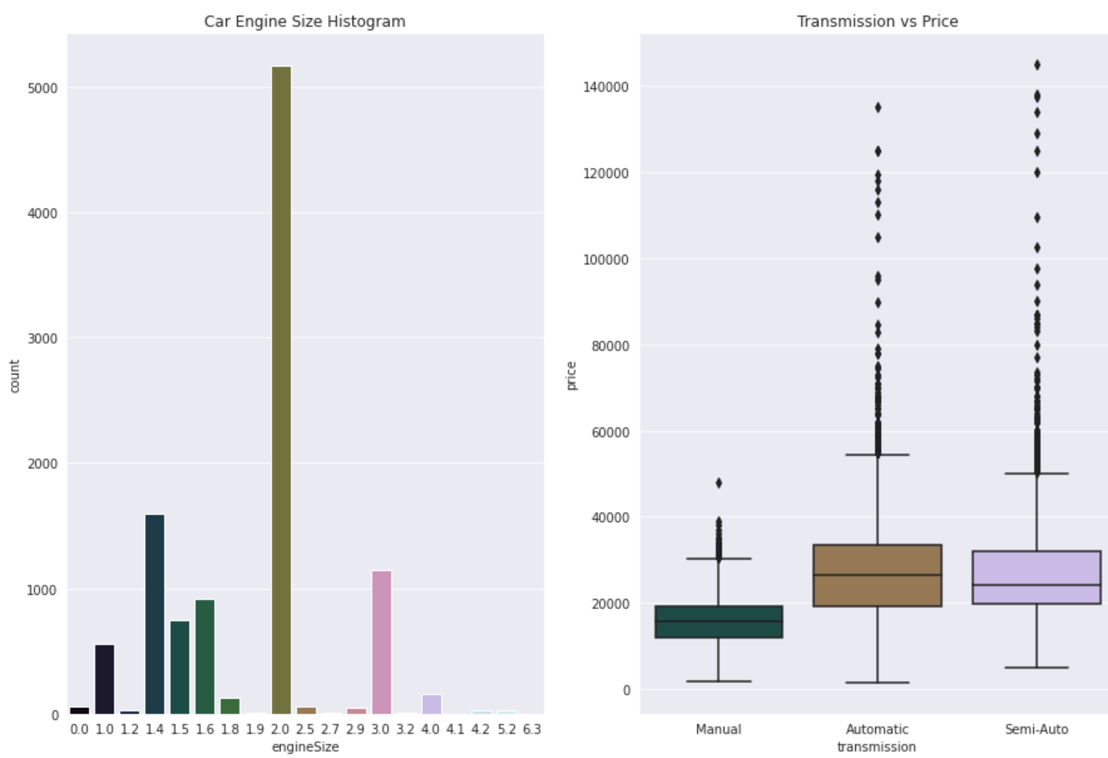
- Make the first subplot by omitting the number of rows (1 row) and columns (2 columns) required to plot the first feature.
- In the first subplot, call countplot() and pass the engine size row as well as the palette color.
- Create a second subplot and pass the columns (transmission and price) as a boxplot.

[48]:
```
plt.figure(figsize=(15, 10))

plt.subplot(1, 2, 1)
plt.title("Car Engine Size Histogram")
sns.countplot(df.engineSize, palette=("cubehelix"))

plt.subplot(1, 2, 2)
plt.title("Transmission vs Price")
sns.boxplot(x=df.transmission, y=df.price, palette=("cubehelix"))

plt.show()
```



Observations:

For the first subplot (Engine Size):

Clearly, most of the engine sizes are requested, as the engine size 2.0 is the most requested, followed by 1.4.

For second suplot (Transimission):

Semi-auto cars are the most expensive cars, automatic cars come next, and the cheapest one is for manual cars, which now it can be explained why manual cars are the most wanted and requested cars by customers. That refers to that the prices for manual cars are the most affordable prices that can range from 10,000 to 50,000.

`Analyze the average amount on money spent for a car model based on the year model`

- Create a model_year object to hold the result and pass it to the visualizing method to be plotted.
- Use groupby to group the data based on the year of the car to get the average price using.mean().
- Display the first ten sorted values for the average price result in numbers.

[49]:
```
model_year = df
model_year = model_year[["price", "year"]].groupby("year").mean()
model_year.sort_values(by=["price"], inplace=True, ascending=False)
model_year = model_year.head(10)
pd.DataFrame(model_year)
```

[49]:

year	price
2020	35967.067039
2019	30410.752268
2018	26296.707176
2017	19951.624289
2016	16908.725051
2015	15128.235235
2014	13890.659955
2013	11690.790378
2012	9860.811765
2011	8944.808511

[50]:
```
plt.figure(figsize=(14, 7))
plt.title("Average Money Spend for a model, by Year")
sns.heatmap(data=model_year, annot=True)
plt.xlabel("Model price")
```

[50]: Text(0.5, 42.0, 'Model price')

Insights

As we could see from the heatmap, the newer cars' prices would be the most expensive as well as the most wanted, as people would spend more on 2020 cars than 2019 ones, as 2011 cars are the least expensive. As that explains the variety of customers in the used car market, they find the prices in the new car market unreliable and it's worth buying a used car, so they tend to buy the newest cars as used cars.

Part 3: Data Preprocessing

Preprocessing the data to be prepared for models following some steps

Step 1: Data cleaning

After exploring the data, relationships, and distribution for all the variables and features, it's time to prepare our data for further steps to start building the machine learning models.

- Begin by removing any implicit or explicit missing values, duplicates, and outliers from the data.
- Drop some columns that may be unnecessary for our process, rename some features, etc.

`1.1 : Remove Duplicates`

Because the dataset contained 103 duplicates, the duplicates were removed and only the first row of duplicates was retained.

Using `drop_duplicates` to achieve this mission, as well as passing the parameters keep = first`and`in place =True‘ to make the change in the original DF.

[51]:
```
df.drop_duplicates(keep="first", inplace=True)
```

Double check.

[52]:
```
df.duplicated().sum()
```

[52]:
```
0
```

No more duplicates in the dataframe.

`1.2: Handling outliers`

We have reviewed all the outliers in the numerical features and have decided to accept some of the outliers and eliminate others.

As the engine size column has a `minimum outliers` with 52 `0` values repeated, where this is not real, a function will be defined to remove the outliers based on `lower quartile`.

[53]:
```
df["engineSize"].value_counts()
```

[53]:
```
2.0    5120
1.4    1589
3.0    1145
1.6     908
1.5     718
1.0     550
4.0     154
1.8     126
2.5      61
0.0      52
2.9      48
1.2      31
4.2      25
5.2      23
3.2       5
1.9       4
2.7       3
4.1       2
6.3       1
Name: engineSize, dtype: int64
```

- Define a function to find out the first quartile and third quartile for the engine size column as calculating the IQR range and extract only the minimum outliers that falls below the first quartile.
- Concat the temporary dataframe with the outlier dataframe that has the outlier values.

- Return first the temporary dataframe with the the outlier dataframe.
- Dorp the temporary dataframe.
- Pass the dataframe to check extract and remove the engine size outliers.

[54]:
```
def remove_outlier(df):
    df_temp = pd.DataFrame()
    df_engineSize = df["engineSize"]

    Q1 = df_engineSize.quantile(0.25)
    Q3 = df_engineSize.quantile(0.75)
    IQR = Q3 - Q1

    df_outlier = df_engineSize[(df_engineSize < (Q1 - 1.5 * IQR))]
    df_temp = pd.concat([df_temp, df_outlier])

    return df.drop(df_temp.index)


df = remove_outlier(df)
```

Check if the minimum outliers were successfully eliminated.

[55]:
```
df["engineSize"].value_counts()
```

[55]:
```
2.0    5120
1.4    1589
3.0    1145
1.6     908
1.5     718
1.0     550
4.0     154
1.8     126
2.5      61
2.9      48
1.2      31
4.2      25
5.2      23
3.2       5
1.9       4
2.7       3
4.1       2
6.3       1
Name: engineSize, dtype: int64
```

No more minimum outliers.

Part 4: Feature Engineering

[93]:
```
df.head()
```

[93]:

	model	year	price	transmission	mileage	fuelType	tax	mpg	engineSize
0	0	2017	12500	1	15735	2	150	55.4	1.4
1	5	2016	16500	0	36203	0	20	64.2	2.0
2	0	2016	11000	1	29946	2	30	55.4	1.4
3	3	2017	16800	0	25952	0	145	67.3	2.0
4	2	2019	17300	1	1998	2	145	49.6	1.0

Step 1: Label Encoder

For transmission, fuel type, manual, automatic, or semi-automatic options, and the same go for the other two features.

Since our process is a regression model rather than a classification into some class, the procedure that has been taken to convert them to numerical type is by using `label encoder` to simply label features from `0 to n-1`.

This process is important for machine learning models in terms of accuracy and evaluation metrics since these models cannot work with categorical types.

Process:

- Create a label encoder object, call `LabelEncoder class using the scikit-learn library` to create an instance of LabelEnoder and store it in the created object.
- "Fit" and "transform" the categorical characteristics.
- To skip dropping the original columns after encoding these columns, assign the result in the original data frame column.

[56]:
```
le = sklearn.preprocessing.LabelEncoder()
```

[57]:
```
le.fit(df["model"])
le.transform(df["model"])
df["model"] = le.transform(df["model"])
```

[58]:
```
le.fit(df["transmission"])
le.transform(df["transmission"])
df["transmission"] = le.transform(df["transmission"])
```

[59]:
```
le.fit(df["fuelType"])
le.transform(df["fuelType"])
df["fuelType"] = le.transform(df["fuelType"])
```

[60]:
```python
df.head()
```

[60]:

	model	year	price	transmission	mileage	fuelType	tax	mpg	engineSize
0	0	2017	12500	1	15735	2	150	55.4	1.4
1	5	2016	16500	0	36203	0	20	64.2	2.0
2	0	2016	11000	1	29946	2	30	55.4	1.4
3	3	2017	16800	0	25952	0	145	67.3	2.0
4	2	2019	17300	1	1998	2	145	49.6	1.0

[61]:
```python
df.shape
```

[61]: `(10513, 9)`

It can be observed that the features are fitted and transformed successfully, as well as the number of rows in the data frame is increased to 10513 observations.

Using iloc to separate the data by row numbers, divide the **features matrix** and **target vector** into train_set and test_set variables.

Split the train set to have the first `10000 observations` and the test set to have the `last 513 observations`. by accessing the indices `iloc`.

[62]:
```python
train_set = df.iloc[:10000, :]
test_set = df.iloc[10000:, :]
```

Print the shapes of train_set and test_set to be sure that we have not missed anything.

[63]:
```python
print("train_set shape:", np.shape(train_set))
print("test_set shape:", np.shape(test_set))
```

```
train_set shape: (10000, 9)
test_set shape: (513, 9)
```

Prepare the sets for X and Y in the next step to drop the target value from the test set.

[64]:
```python
test_set = test_set.drop(columns="price")
test_set.shape
```

[64]: `(513, 8)`

It's observed that 1 column has been dropped, so the set has 8 columns now, which means that the test set has all the features except the target variable.

Step 2: Splitting data

Split the dataset into `train data` and 'test data to evaluate the performance of supervised machine learning algorithms.

The procedure for splitting the dataset was as follows:

1. Divide the dataset into two subsets.

2. `Training Dataset` to "fit" our model.
3. `Test Dataset` to `evaluate` the fitted model and check the `accuracy` of the model.
4. As we have set the percentage data for both subsets as follows: 70% for training, 30% for testing

Split the orginal dataset to `X` and `y` input and output columns

Use `train_test_split() function` that the scikit-learn Python machine learning library provides to split the data and pass x and y as well as the size of the split. Not to forget to mention that, when we want to compare the machine learning algorithms, later on, they require us to fit and evaluate the same subset of the dataset. According to that, `random _state` will be passed through the function to get an identical split of the original dataset.

[65]:
```
X = train_set.drop(columns="price")
y = train_set["price"]

print(X.shape)
print(y.shape)
```

```
(10000, 8)
(10000,)
```

Now X and Y variables are prepared.

Step 3: Scalling data

To get better performance from the built machine learning model, scaling the data is an excellent idea to get the data closer to each other so the algorithms can be fitted well and trained faster, Thus, all the variables will be generalized, so the distance between them will be lower. Using StandarScaler () to scale the data is the most common scale technique for the data points. This means that the mean of these scaled data points will be valued between 0 and 1 .

[66]:
```
scaler = sklearn.preprocessing.StandardScaler()
X = scaler.fit_transform(X)
print(X.shape)
```

```
(10000, 8)
```

[67]:
```
X
```

[67]:
```
array([[-1.11957328, -0.05795055, -0.1580572 , …,  0.380771  ,
         0.34616957, -0.92599675],
       [-0.15575331, -0.54046972, -1.48071162, …, -1.5802166 ,
         1.03603149,  0.10368939],
       [-1.11957328, -0.54046972, -0.1580572 , …, -1.4293714 ,
         0.34616957, -0.92599675],
       …,
       [-0.3485173 , -2.47054641, -0.1580572 , …, -1.4293714 ,
         0.71461855,  0.10368939],
```

```
       [-1.11957328, -0.05795055, -1.48071162, …, -1.5802166 ,
         1.03603149, -1.61245418],
       [-0.54128129, -1.98802724, -0.1580572 , …, -1.4293714 ,
         1.03603149,  0.10368939]])
```

- Split the dataset into two parts: a `training set` and `test set`. The following command will be performed for splitting to `X_train`, `X_test`, `y_train`, and `y_test`

[68]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=44, shuffle=True
)
```

[69]:
```
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(6700, 8)
(6700,)
(3300, 8)
(3300,)
```

Part 5: Model Training

Step 1: Agorithm Selection and Hyperparameter tuning

The most crucial phase is to begin considering the optimal algorithm models that will be trained on the dataset.

Since the main purpose of the dataset is prediction, that means regression algorithms must be used and trained over the dataset.

For the best prediction, as identifying that the main objective of our progress is:

- A price prediction model.
- `Inputs` are the variables like (car model, Year, Mile of age for the car, FuelType, etc...)

Data was collected and prepared for the models.

Based on the prediction, a regression model that belongs to the "supervised algorithms" will then be used to predict the output label (dependent variable) on the basis of input features `(independent variables)` by using the `Scikit-learn` library.

`Model 1: Linear Regression`

Because of its simplicity and short training period, linear regression was chosen as the first model by using the **linear regression()** function of Scikit-learn.

The command below will create a `linear regressor object`.

[70]:
```
linear_reg = sklearn.linear_model.LinearRegression()
```

We are now ready to implement the **GridSearch** by selecting the **Best Parameters** that will effectively influence and maximize the performance of the algorithm model while reducing errors.

The linear regression algorithm expects that the input and output variables have a 'linear relationship.'

A hyperparameter tuning process will be performed to identify the best parameter for the linear regression algorithm as follows:

- Define a set of hyperparameter values to search for and find the best hyperparameter values.
- Make a **`dictionary`** and pass the names of the hyperparameters as the **`keys`** and their values.

For linear regression hyperparameter tuning:

- Pass the **`normalize`** : is the default value is **`False`**. Bypassing the parameter, we will figure out if the model is best normalized before regression or not by working on the **`mean`** value.
- Use `GridSearchCv` to pick each value for each passed parameter, bypassing the parameter_grid and the model, as well as

Cross-validation for each passed parameter set (**CV**, **n_jobs =-1**) to train the model as quickly as possible to determine which value works best for the chosen model.

(Following this step, the grid object is ready)

- Fit the model to train on the train set.
- Print the best score and best parameters for this model.

For the case of these hyperparameters, two models will be trained since we passed two values in the parameters_grid dictionary.

[71]:
```
parameters_grid = {"normalize": [True, False]}
```

[72]:
```
linear_reg_grid = sklearn.model_selection.GridSearchCV(
    linear_reg, param_grid=parameters_grid, cv=2, n_jobs=-1, scoring="r2"
)
```

[73]:
```
linear_reg_grid.fit(X_train, y_train)
```

[73]:
```
GridSearchCV(cv=2, estimator=LinearRegression(), n_jobs=-1,
             param_grid={'normalize': [True, False]}, scoring='r2')
```

[74]:
```python
best_score_1 = linear_reg_grid.best_score_
best_params_1 = linear_reg_grid.best_params_

print("Results from Grid Search")
print("Best score along searched params:\n", best_score_1)
print("Best parameters are:\n", best_params_1)
```

```
Results from Grid Search
Best score along searched params:
 0.8157198760587283
Best parameters are:
 {'normalize': True}
```

`linear_reg_grid object` returned the best model with the best set of hyperparameters.

- (normalize: True)
- We used a 5-fold cross-validation because the score decreased when we tried less than 5, so the best k-fold cross-validation was 5 and more.
- The best model has achieved an R2 value of 0.82, which is pretty good.

`Model 2: Decision Tree Regressor`

Given a data point, a decision tree will be run through the entire tree except the leaf nodes, making splits based on asking boolean (True/False) questions are asked till the result is the leaf node. The final prediction employs Mean Absolute Error/Mean Square Error to average the values of the dependent variables in those leaf nodes. It constructs the forest using an ensemble of Decision Trees as the result is enhanced because it employs the ensemble method.

Here, we will use the Decision Tree Regressor`to build the regression model as a`Tree Structure‘ to solve different kinds of regression problems.

Create **`Decision Tree regressor object.`** to use GridSearchCV.

[75]:
```python
dtree_reg = sklearn.tree.DecisionTreeRegressor()
```

When it comes to training the model with all possible hyperparameters, questions may be asked like what should be the best range of values that we should try for the `maximum depth`, what is the minimum amount of samples requires to splitting an internal node?

Answers to these kinds of questions are hard to find, and they are not straightforward since every change would affect the model’s performance and score.

- Define a set of hyperparameters.
- `criterion`: The default value is ’mse’ for mean squared error, which measures the quality of the split.
- `splitter`: This hyperparameter is used to select the type of split at each node: ’best’ for the BEST split and ’random’ for the BEST RANDOM split. Both values will be presented to the model to see which one performs better.

- `max_depth`: indicates the depth of the tree. Trying to increase this value will make our model more likely to overfit.
- `'max_features'`: By default, it's `None`, We will try to pass (auto, None: in this case, the max_features will be n_features). (log2: max_features=**log2**(n_features). (log2: max_features=**log2**(n_features). (sqrt: max_features=**sqrt**(n_features).
- `"min_samples_split"`: Denotes the minimum number of samples needed to split the child node.

[76]:
```
parameters_grid = {
    "criterion": ["mse"],
    "splitter": ["best", "random"],
    "max_depth": [1, 3, 5, 7, 9, 11, 12],
    "min_samples_split": [10, 20, 40],
    "max_features": ["auto", "log2", "sqrt", None],
}
dtree_reg_grid = sklearn.model_selection.GridSearchCV(
    dtree_reg, param_grid=parameters_grid, cv=5, n_jobs=-1
)
dtree_reg_grid.fit(X_train, y_train)

best_score_2 = dtree_reg_grid.best_score_
best_params_2 = dtree_reg_grid.best_params_

print("Best score along searched params:\n", best_score_2)
print("The best parameters = {}".format(best_params_2))
```

```
Best score along searched params:
 0.9272669836291023
The best parameters = {'criterion': 'mse', 'max_depth': 11, 'max_features':
'auto', 'min_samples_split': 10, 'splitter': 'random'}
```

When attempting to adjust the CV parameter, a score of less than 5 will result in a lower score, while a score of 5 or more will yield better results. The score for the hyperparameter 'max feature' reduced to 92 percent when it wasn't used, but it improved after the max feature parameter was added and changed.

`Model 3: Lasso`

As this type of model assumptions that the input variables and the target variable have a linear relationship and it uses shrinkage as the datapoint shrinks, using the L1 penalty will minimize the size of all coefficients and allow them to go to the value of zero.

Create **`Lasso object.`** to use GridSearchCV.

[77]:
```
lasso_reg = sklearn.linear_model.Lasso()
```

As the default value for `alpha` is 1, we will grid search more alpha values to discover what works best with the model.

[78]:
```
parameters_grid = {"alpha": [1, 2, 10, 50], "selection": ["random", "cyclic"]}


lasso_reg_grid = sklearn.model_selection.GridSearchCV(
    lasso_reg, param_grid=parameters_grid, cv=10, n_jobs=-1
)
lasso_reg_grid.fit(X_train, y_train)

best_score_3 = lasso_reg_grid.best_score_
best_params_3 = lasso_reg_grid.best_params_

print("Best score along searched params:\n", best_score_3)
print("Best parameters = {}".format(best_params_3))
```

```
Best score along searched params:
 0.8147124494379643
Best parameters = {'alpha': 10, 'selection': 'cyclic'}
```

Model 4: Random Forest Regressor

In order to generate a regression model, the RandomForestRegssor model employs numerous decision trees. Because each tree predicts the value of the target variable, these numerous trees act as an ensemble. Finally, all of these predictions are aggregated to generate a more specific and accurate prediction.

[79]:
```
rforest_reg = sklearn.ensemble.RandomForestRegressor()
```

There are various parameters to tune in Random Forest, some of these parameters are important and are discussed below:

1. Number of Estimators (`"n_estimators"`): This is the number of decision trees, high number of trees high the model will overfit.

2. Maximum number of features (`"max_features"`): Present the maximum number of features that should be trained in a single tree.

As imported another features to best train the model by following features: * `max_depth`: Indicates the depth of the tree that will be the longest path between the root node and the leaf node: Increase this value will increase the model performance.

`min_sample_leaf`: The smallest number of samples that can be found in newly generated leaves after splitting,

- `min_sample_split`: The minum amoint of samples (observations) required to split the given node, it's vary to set it at least 1 sample so trying to set them greater than 1, as the default value for this parameter is 2 so it's a good idea to tune the hyperparameter with a values more than 2 trying to reduce number of splits in the node to prevent the overfitting
- `bootstrap`: used the random forest techniques (bagging and aggregation) to redure the variance to produce robust and trees.

[84]:
```
parameters_grid = {
    "bootstrap": [True],
    "max_depth": [80, 90, 100, 110],
    "max_features": [2, 3],
    "min_samples_leaf": [3, 4, 5],
    "min_samples_split": [8, 10, 12],
    "n_estimators": [100, 200, 300, 1000],
}
rforest_reg_grid = sklearn.model_selection.GridSearchCV(
      rforest_reg, param_grid=parameters_grid, cv=5, n_jobs=-1, verbose=2
)
rforest_reg_grid.fit(X_train, y_train)

best_score_4 = rforest_reg_grid.best_score_
best_params_4 = rforest_reg_grid.best_params_

print("Best Accuary = {}".format(best_score_4))
print("Best found hyperparameters = {}".format(best_params_4))
```

```
Fitting 5 folds for each of 288 candidates, totalling 1440 fits
Best Accuary = 0.9514710458651219
Best found hyperparameters = {'bootstrap': True, 'max_depth': 80,
'max_features': 3, 'min_samples_leaf': 3, 'min_samples_split': 8,
'n_estimators': 300}
```

To determine the optimal number of trees, a GridSearch Algorithm was used.

- Best training score was found out to be 95% when 300 trees were used.
- The performance of the model is the highest close to 3 value of the max features.
- As max_feature parameter increased, the model perforamance is increased

[85]:
```
models = pd.DataFrame(
    {
        "Model": [
            "Linear Regression",
            "Decision Tree",
            "Lasso",
            "RandomForestRegresso",
        ],
        "Score": [best_score_1, best_score_2, best_score_3, best_score_4],
    }
)

models.sort_values(by="Score", ascending=False)
```

[85]:

	Model	Score
3	RandomForestRegresso	0.951471

```
1          Decision Tree  0.927267
0      Linear Regression  0.815720
2                  Lasso  0.814712
```

Insights:

The best score was for **`Random Forset Regressor`** algorithm, followed by ”Decision Tree” and then lasso, Linear regression respectively.

Step 2: Evaluation Metrics

The dataset was trained with 4 different regressors, to determine the best regressor model performs the best, the 4 models will be evaluated using a variety of evaluation metrics: 1. **`R2 score`**.

2. **`Mean Suqare Error`**.
3. **`Mean Absolute Error`**.

The R2 score will be used to determine how well the data fits the regression line, whereas Root Mean Squared Error is the standard deviation of the prediction errors, while Mean Absolute Error is the average distance between the train data and the predicted data

As MSE is a single value that indicates how good a regression line is, smaller MSE value better the model is which mean the value has a small errors, where MAE represents the dataset’s average residual..

`1.1 Linear Regrission Prediction`

[86]:
```
y_pred1 = linear_reg_grid.predict(X_test)

score_1 = sklearn.metrics.r2_score(y_test, y_pred1) * 100

MAE_1 = sklearn.metrics.mean_absolute_error(y_test, y_pred1)
MSE_1 = sklearn.metrics.mean_squared_error(y_test, y_pred1)
```

`1.2 Decision Tree Regressor Pridiction`

[87]:
```
y_pred2 = dtree_reg_grid.predict(X_test)

score_2 = sklearn.metrics.r2_score(y_test, y_pred2) * 100

MAE_2 = sklearn.metrics.mean_absolute_error(y_test, y_pred2)
MSE_2 = sklearn.metrics.mean_squared_error(y_test, y_pred2)
```

`1.3 Lasso Pridiction`

[88]:
```
y_pred3 = lasso_reg_grid.predict(X_test)

score_3 = sklearn.metrics.r2_score(y_test, y_pred3) * 100

MAE_3 = sklearn.metrics.mean_absolute_error(y_test, y_pred3)
MSE_3 = sklearn.metrics.mean_squared_error(y_test, y_pred3)
```

1.4 Random Forest Prediction

[89]:
```
y_pred4 = rforest_reg_grid.predict(X_test)

score_4 = sklearn.metrics.r2_score(y_test, y_pred4) * 100


MAE_4 = sklearn.metrics.mean_absolute_error(y_test, y_pred4)
MSE_4 = sklearn.metrics.mean_squared_error(y_test, y_pred4)
```

Step 3: Models Comparision

The **R2 score** of our predictions was used to quantify the results of our tests. This score is a statistical measure of how near the data are to the fitted regression lineand the score comparission for the used models are represnted and sorted as a dataframe as follows:

[90]:
```
test_models = pd.DataFrame(
    {
        "Model": [
            "Linear Regression",
            "Decision Tree",
            "Lasso",
            "RandomForestRegresso",
        ],
        "R2 Score": [score_1, score_2, score_3, score_4],
        "MAE": [MAE_1, MAE_2, MAE_3, MAE_4],
        "MSE": [MSE_1, MSE_2, MSE_3, MSE_4],
    }
)

test_models.sort_values(by="R2 Score", ascending=False)
```

[90]:

	Model	R2 Score	MAE	MSE
3	RandomForestRegresso	95.805404	1601.740812	5.978734e+06
1	Decision Tree	92.795684	2098.699724	1.026862e+07
0	Linear Regression	80.590270	3352.080733	2.766551e+07
2	Lasso	80.580727	3348.767105	2.767911e+07

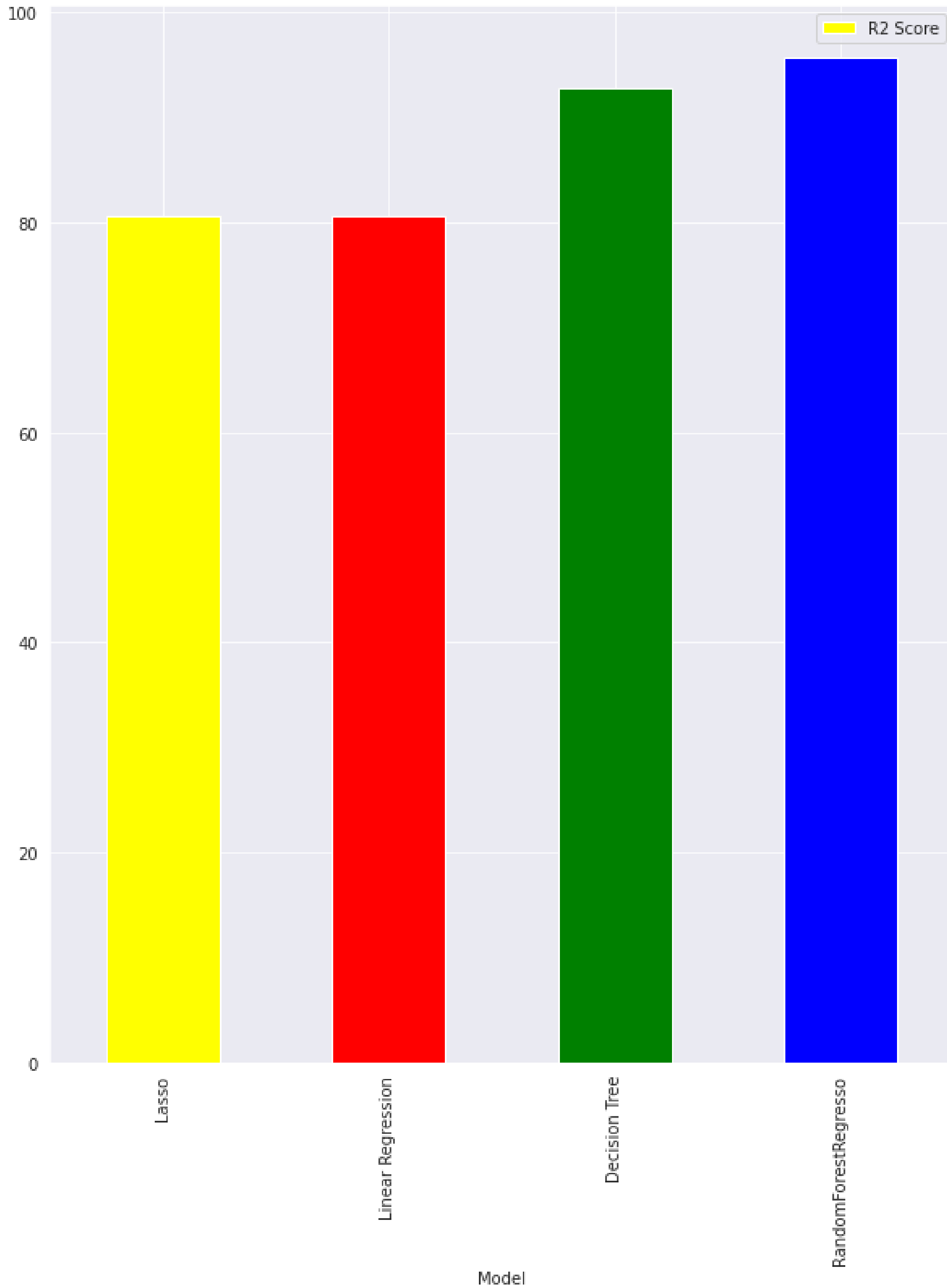
Random Forest Regressor has the highest accuracy with 95.7% of the other three algorithms and has the lower error in all three-evaluation metrics. So the top model to be selected is the Random Forset Regrosser, followed by the Decision Tree algorithm with 91.4% accuaracy.

[91]:
```
test_models = test_models.sort_values("R2 Score")

test_models.plot(
    x="Model",
    y="R2 Score",
    kind="bar",
    figsize=(10, 12),
```

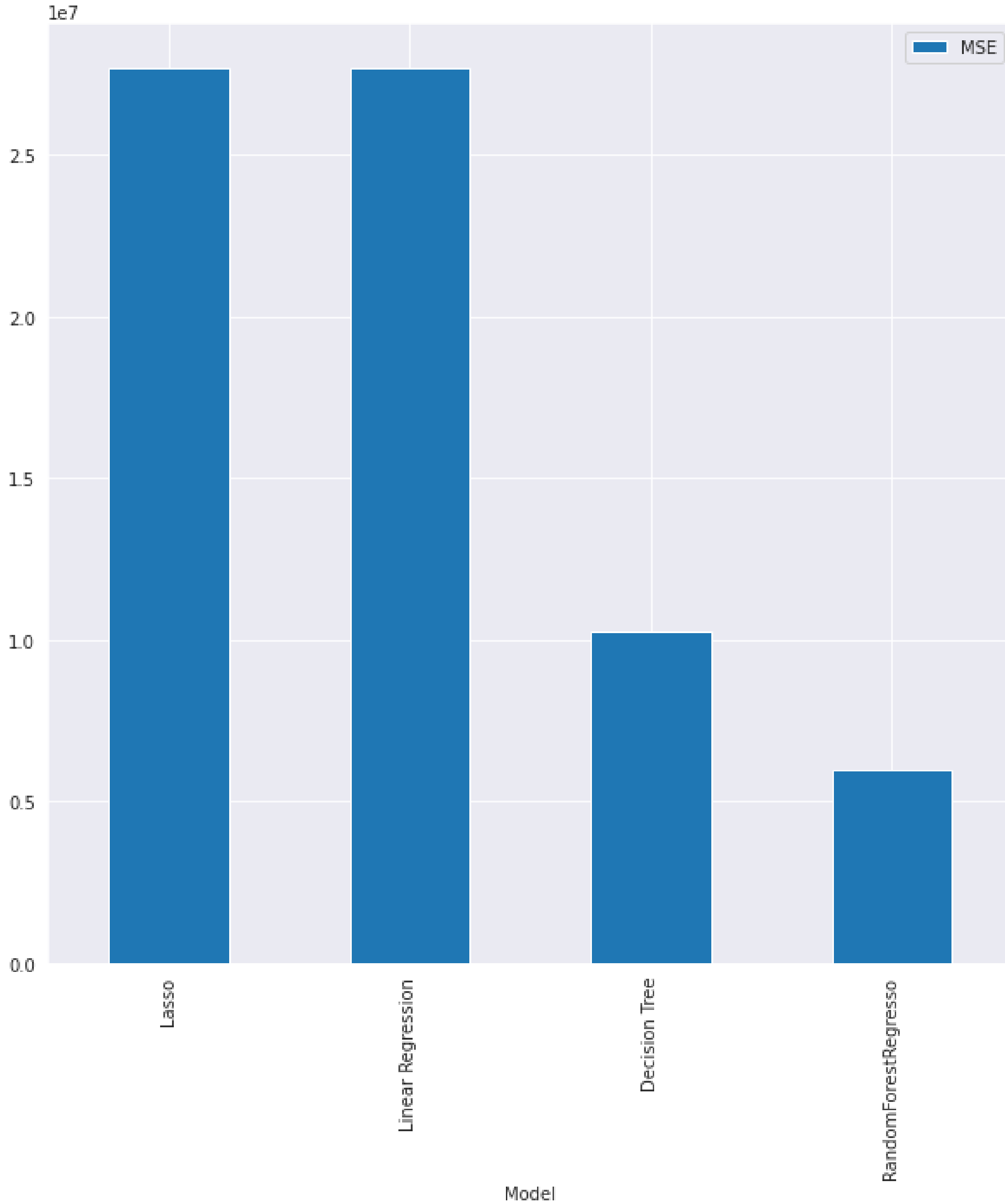
```
    color=["yellow", "red", "green", "blue", "cyan"],
)
```

[91]: <AxesSubplot:xlabel='Model'>



[94]:
```
test_models = test_models.sort_values("R2 Score")
test_models.plot(x="Model", y=["MSE"], kind="bar", figsize=(10, 10))
```

[94]: <AxesSubplot:xlabel='Model'>



Conclusion

A set of data is gathered and pre-processed. An exploratory data analysis was carried out, and the best parameters were determined for each regressor model using the hyperparameter tuning method, GridSearchCV.

Scores for each regressor were identified and five-fold cross-validation is used to measure the overall performance for the some machine learning regression algorithms such as linear regression, LASSO regression, decision tree, random forest were used.

following an in-depth exploratory data study to determine the impact of each feature on pricing,The best-performing model (**Random Forest Regressor**) was picked after each method's performance was evaluated as it's performed much better than other models.

Picked model gaves an accuracy of 95% on train data and approximately 96% on test data.

As a recommendation, to deploy the model in the future and create a completely automated, interactive system on the web that includes a database of used cars and their prices to be available for the end users later on.

`Recommendations and Future Work`

Finally, the model has been deployed as a web application on a local system as well as mobile application with the intention of making it available to end users later.