Car Prices Prediction

1.1 Introduction

If we look at car companies, we see that they set different prices of cars based on a few set of factors such as horse power and other parameters. If a new company is planning to invest in manufacturing cars, it would be really useful for the company to understand some of the factors that affect the prices of the car respectively. It would be really useful if we are able to use machine learning in the prediction of the prices of cars and also predict the prices of different cars. With the help of machine learning and data science, it is possible to explore some useful insights about the car prices and other important features for the manufacture of the cars respectively. Once we understand that data, we would be able to provide insights to new companies that are willing to invest in the manufacture of different cars respectively.

1.2 Metrics

1. Mean Squared Error (MSE)
2. Mean Absolute Error (MAE)

1.3 Source

The data which is used can be downloaded from the repository. This is a fun data to work with and we would be coming through some of the key insights that would help us get the predictions for different cars.

https://www.kaggle.com/CooperUnion/cardataset

There are some cars such as Bugatti and Lamborghini whose prices are beyond what an average Joe would be willing to buy. There are many other affordable car brands such as Ford and Toyota. The exploration of the data that we are about to work with will give us a very good idea about all the cars and their average prices for a particular car brand. We would be working with the data that contains the most accurate information about cars and so on.

We would start by reading the data and then, visualizing the plots followed by using various machine learning algorithms for predicting the prices of cars. Moreover, we would be comparing those machine learning algorithms and check the best algorithm and use it for predictions for new cars that we are about to enter. So without much delay, let's go!

Table of Contents

1.4 Importing libraries

We would be importing some of the libraries for understanding the data, visualizing and getting a good idea about the machine learning models. Below are some of the libraries that would be imported.

In [1]:

```
import pandas as pd
import seaborn as sns
import matplotlib as plt
import numpy as np
from sklearn.preprocessing import OneHotEncoder, StandardScaler, MinMaxScaler
from sklearn.model_selection import GridSearchCV, train_test_split, KFold, cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_squared_error, accuracy_score, mean_absolute_error, mean_squared_error
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
import missingno as msno
from category_encoders import TargetEncoder, OneHotEncoder
import warnings
warnings.filterwarnings("ignore")
sns.set(rc = {'figure.figsize':(20,20)})
%matplotlib inline
```

In [2]:

```
data = pd.read_csv(r"C:\Users\rohan\Downloads\data.csv", index_col = False)
```

In [3]:

```
data.shape
```

Out[3]:

```
(11914, 16)
```

In [4]:

```
data.head()
```

Out[4]:

	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors	Market Category	Vehicle Size	Vehicle Style	higl
0	BMW	1 Series M	2011	premium unleaded (required)	335.0	6.0	MANUAL	rear wheel drive	2.0	Factory Tuner,Luxury,High-Performance	Compact	Coupe	
1	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,Performance	Compact	Convertible	
2	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,High-Performance	Compact	Coupe	
3	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,Performance	Compact	Coupe	
4	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Luxury	Compact	Convertible	

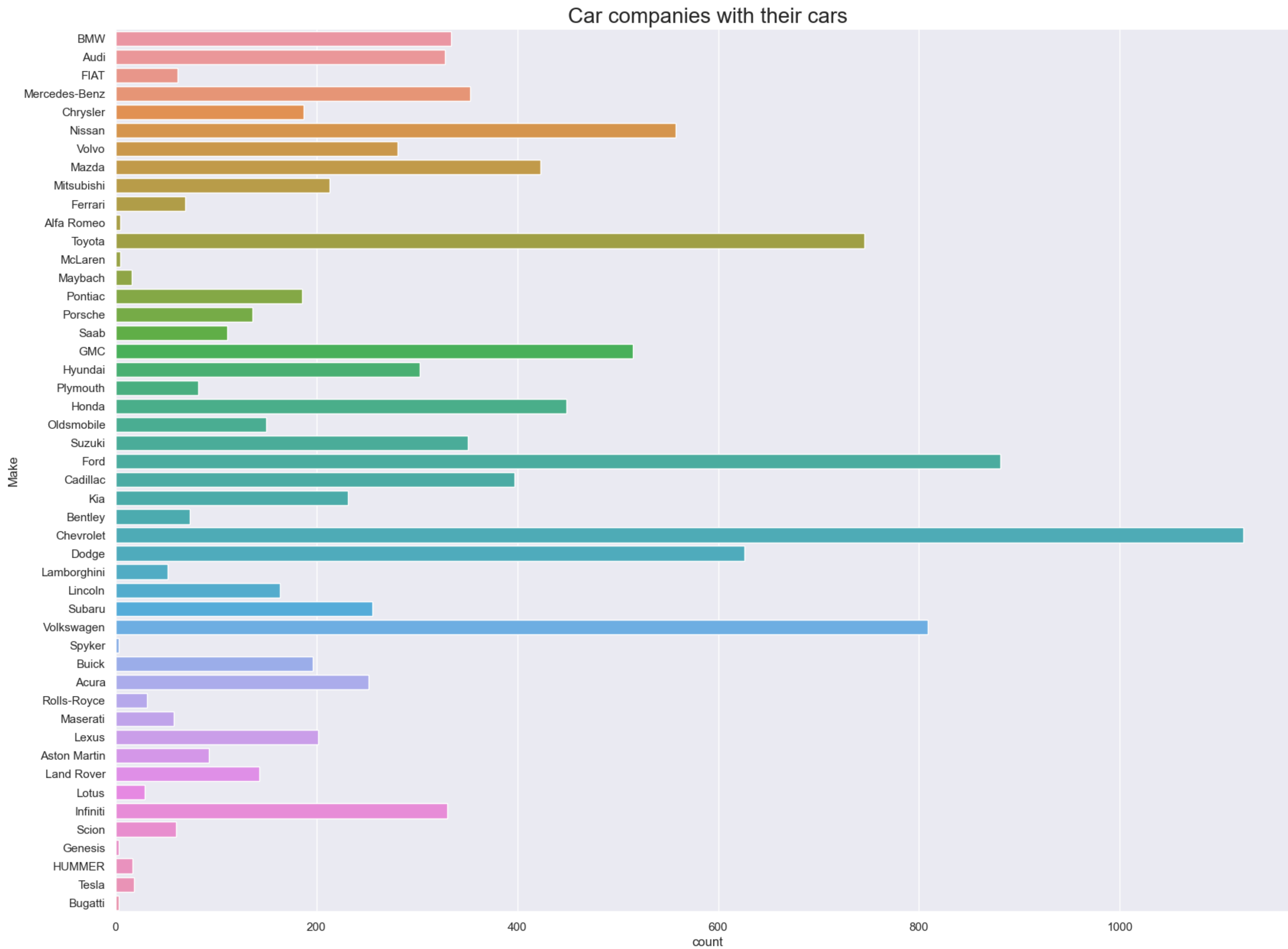
Countplot

Countplots are used with the help of seaborn library in python. These plots give us a good understanding of the total number of elements present in a particular feature that we have considered. Below are a list of countplots for different features of interest which would help in understanding the overall distribution of data based on different features. Therefore, taking a look at these plots would ensure that one is familiar with the data along with the total number of classes for different features respectively.

Countplot of different car companies

In [5]:

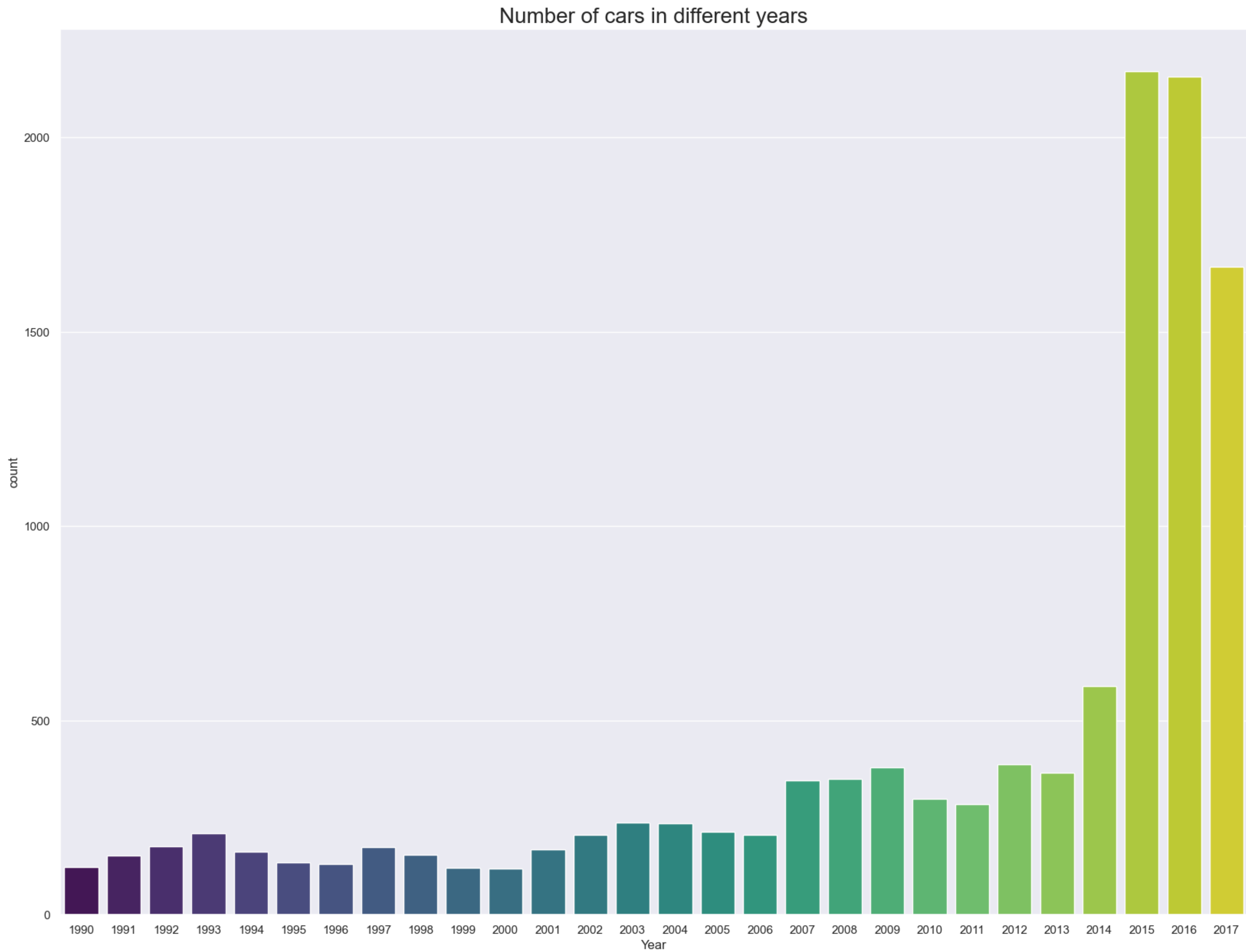
```
from matplotlib import figure
import matplotlib.pyplot as plt
plt.figure(figsize = (20, 15))
sns.countplot(y = data.Make)
plt.title("Car companies with their cars", fontsize = 20)
plt.show()
```


Car companies with their cars
BMW
Audi
FIAT
Mercedes-Benz
Chrysler
Nissan
Volvo
Mazda
Mitsubishi
Ferrari
Alfa Romeo
Toyota
McLaren
Maybach
Pontiac
Porsche
Saab
GMC
Hyundai
Plymouth
Honda
Oldsmobile
Suzuki
Ford
Cadillac
Kia
Bentley
Chevrolet
Dodge
Lamborghini
Lincoln
Subaru
Volkswagen
Spyker
Buick
Acura
Rolls-Royce
Maserati
Lexus
Aston Martin
Land Rover
Lotus
Infiniti
Scion
Genesis
HUMMER
Tesla
Bugatti
Make
0
200
400
600
800
1000
count

We would be using seaborn's countplot to check the total number of cars per company that we have in our dataset. We see that there are more than 1000 cars for the company 'Chevrolet' in our dataset. We also see that there just a few cars for companies such as 'Bugatti' and 'Genesis' which is reflective of the real-world as we don't find different models of these cars. A countplot could really give us a good understanding of the data that we are working.

CountPlot of total cars per different years

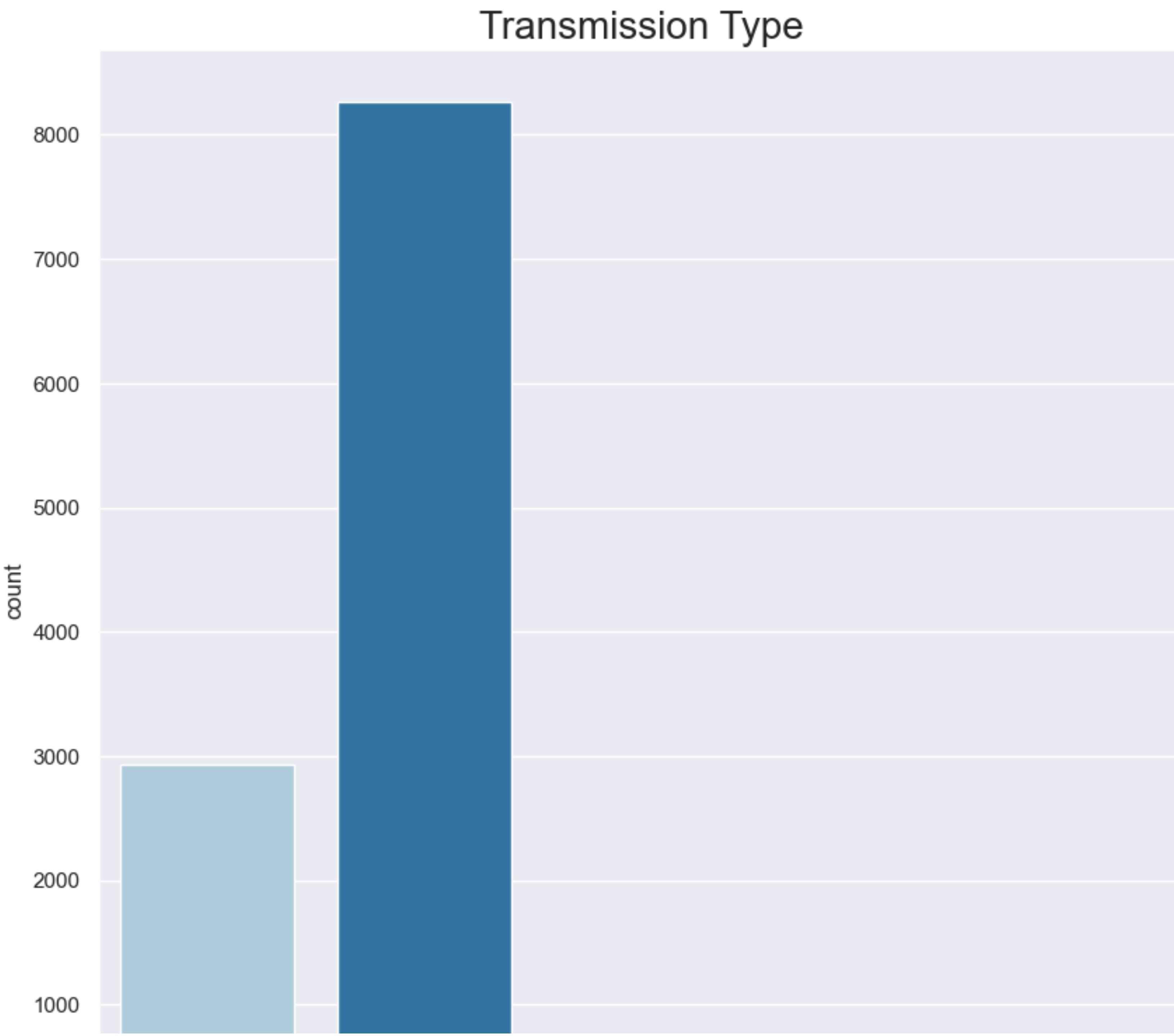
In [6]:
```
plt.figure(figsize = (20, 15))
sns.countplot(data.Year, palette = 'viridis')
plt.title("Number of cars in different years", fontsize = 20)
plt.show()
```

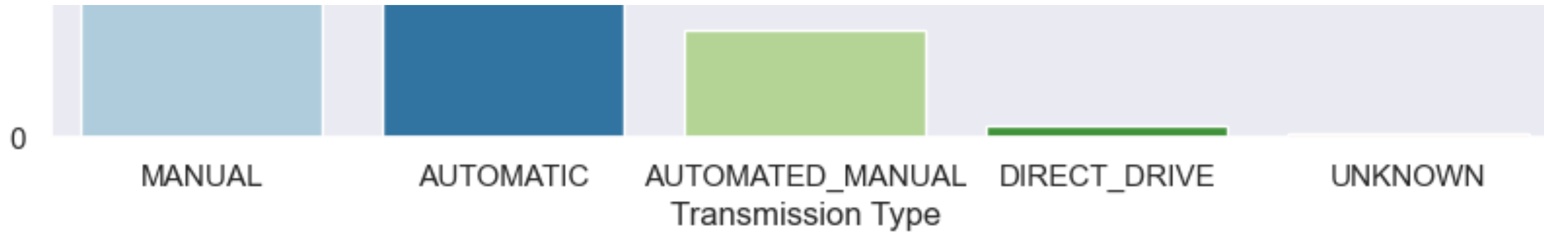

Number of cars in different years
count
2000
1500
1000
500
0
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
Year

We would be checking the total number of cars per year just to understand the data. We find that there are many cars in the years 2015 to 2017 compared to the other years in our dataset. From this visualization, we can get an understanding that most of our data contains recent values. This is a good dataset as we are more interested in the prices of the future cars. It would be better if we have the most recent values as they would help us well in our predictions.

Counting the cars based on transmission type

In [7]:
```
plt.figure(figsize = (10,10))
sns.countplot(data['Transmission Type'], palette = 'Paired')
plt.title("Transmission Type", fontsize = 20)
plt.show()
```


Transmission Type
count
8000
7000
6000
5000
4000
3000
2000
1000



We are all interested in cars that are automatic as they are really easy to handle and efficient. In addition to this, most of the manual cars are being replaced by automatic cars and thus, we don't have a lot of demand for manual cars. That is being reflected here in the dataset. We see that when we see the total number of cars based on transmission type, we find that there are many automatic cars as compared to the cars that are manual. There are a few automatic_manual cars that is second option for the buyer of the cars. Thus, we could see that most of the cars that we have chosen in our dataset are automatic.

Getting the unique elements from the data

In [8]:
```
data.nunique()
```

Out[8]:
```
Make                   48
Model                 915
Year                   28
Engine Fuel Type       10
Engine HP             356
Engine Cylinders        9
Transmission Type       5
Driven_Wheels           4
Number of Doors         3
Market Category        71
Vehicle Size            3
Vehicle Style          16
highway MPG            59
city mpg               69
Popularity             48
MSRP                 6049
dtype: int64
```

We see from the below that there are a few categories for features such as 'Number of Doors', 'Vehicle Size', 'Driven_Wheels' and so on. That is what is expected as we should not have a lot of categories for the above mentioned features. In addition to this, we see that there are a few features that contain a lot of categories. Some of the features include 'Model', 'Engine HP' and so on. That is also what is expected in real life as we should have different models for cars and also different values of horsepower (hp).

In [9]:
```
data.head()
```

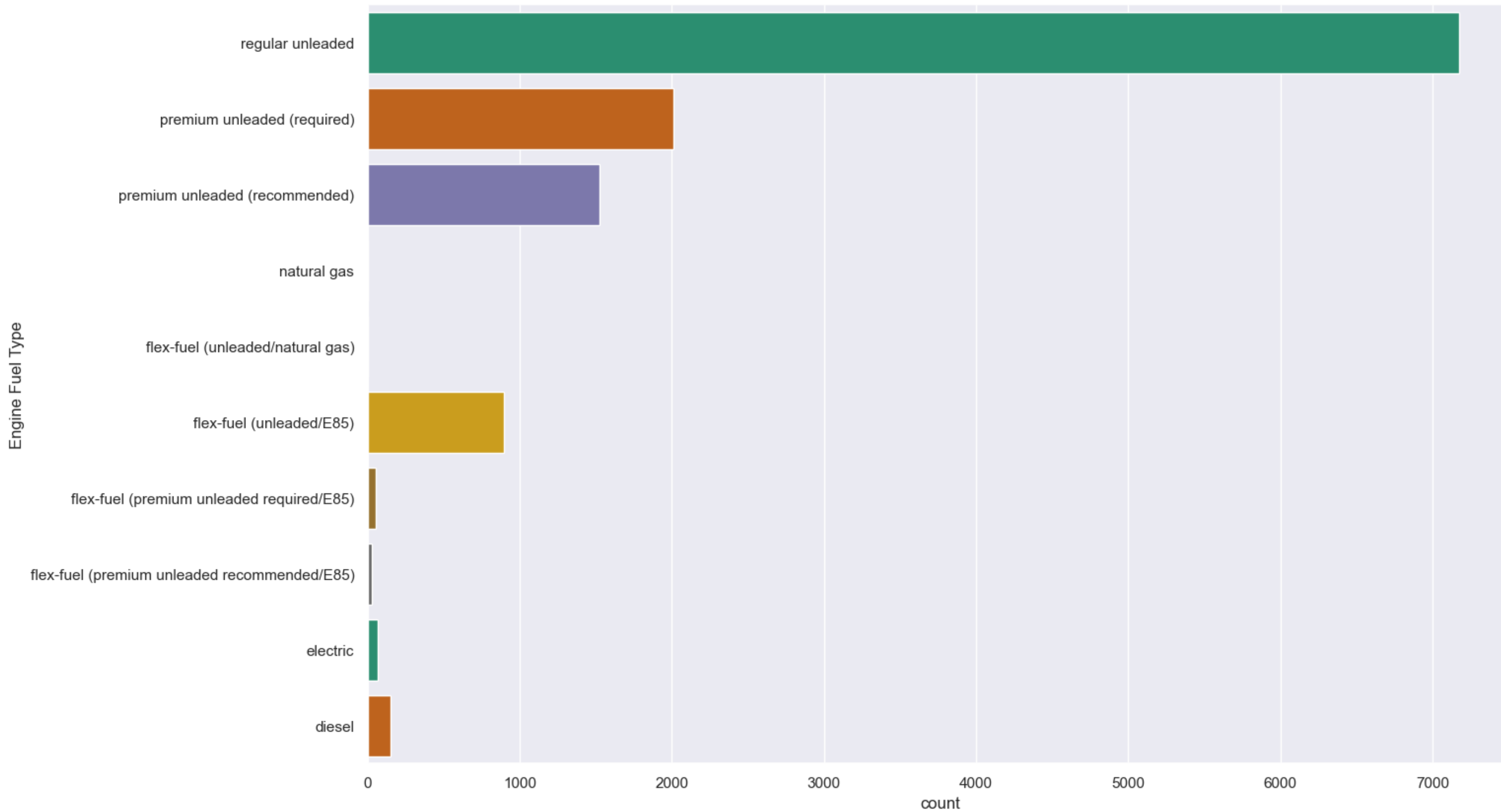
Out[9]:

	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors	Market Category	Vehicle Size	Vehicle Style	higl
0	BMW	1 Series M	2011	premium unleaded (required)	335.0	6.0	MANUAL	rear wheel drive	2.0	Factory Tuner,Luxury,High-Performance	Compact	Coupe	
1	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,Performance	Compact	Convertible	
2	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,High-Performance	Compact	Coupe	
3	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,Performance	Compact	Coupe	
4	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Luxury	Compact	Convertible	

Countplot of Engine Fuel Type

In [10]:
```
plt.figure(figsize = (15, 10))
sns.countplot(y = data['Engine Fuel Type'].sort_values(ascending = False), palette = 'Dark2')
```

Out[10]:
```
<AxesSubplot:xlabel='count', ylabel='Engine Fuel Type'>
```

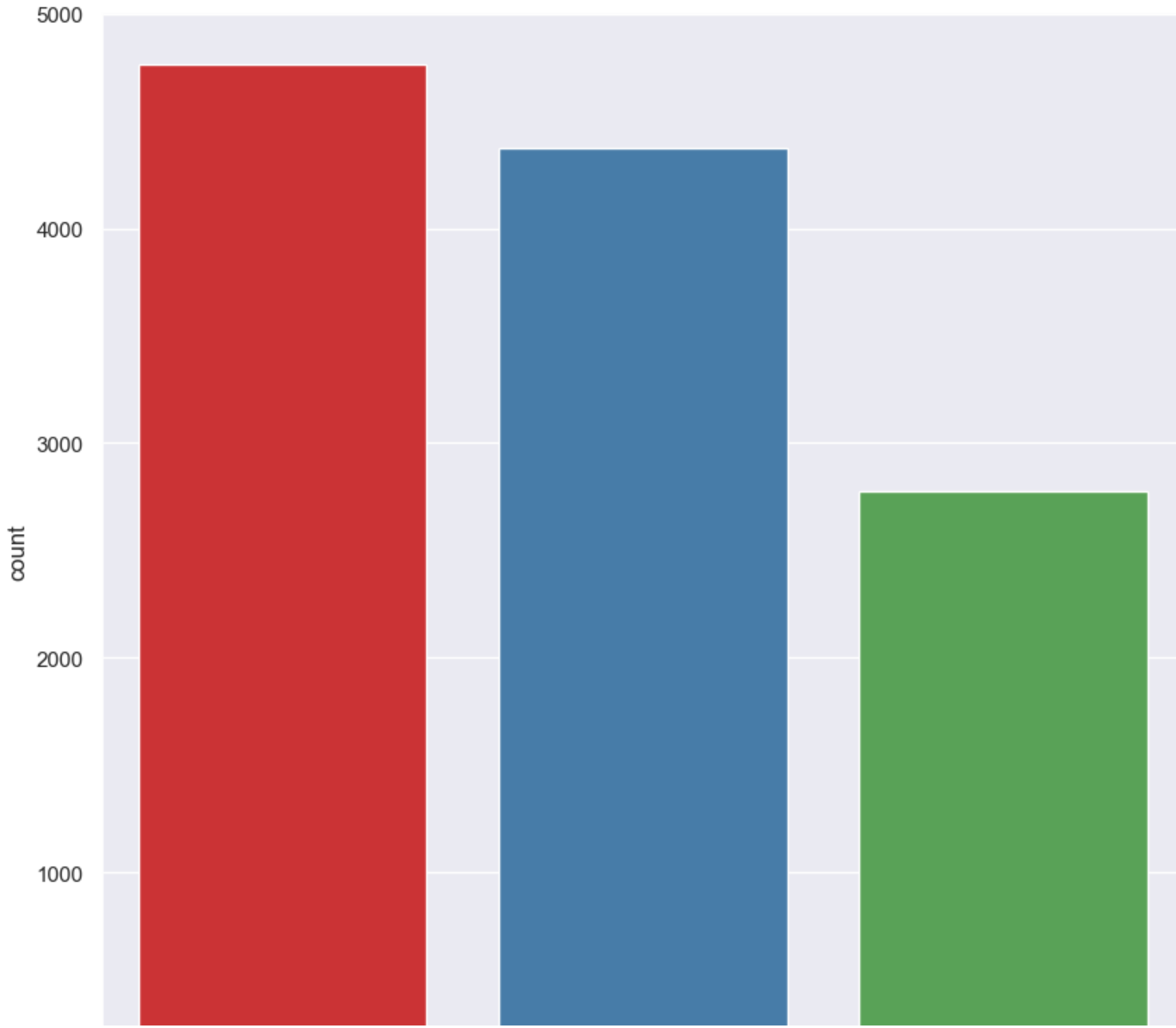


We would be counting the total number of values for 'Engine Fuel Type' feature. We see that there are more than a majority of cars that have 'regular unleaded' as their category. Apart from this, there are other categories such as 'premium unleaded (required)' and 'premium unleaded (recommended)' which could also be taken into consideration. Moreover, we have a few few cars that are electric in our data. That is what is expectd as in real-life, we don't find a lot of electric cars (hope they replace our conventional cars leading to safer environment! Just kidding).

Countplot of Vehicle Size

In [11]:
```
plt.figure(figsize = (10,10))
sns.countplot(x= 'Vehicle Size', data = data, palette = 'Set1')
```

Out[11]:
```
<AxesSubplot:xlabel='Vehicle Size', ylabel='count'>
```

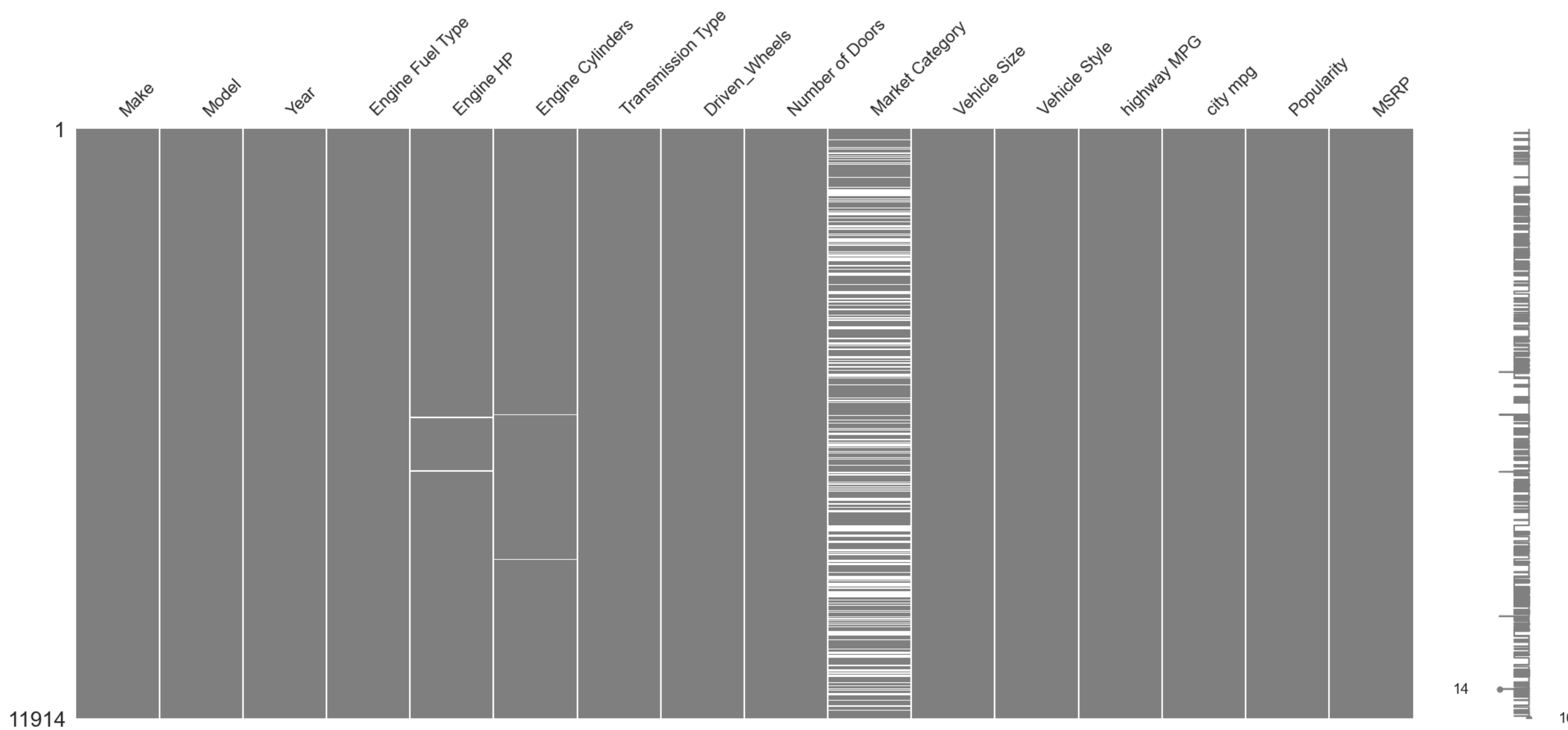

5000
4000
3000
2000
1000
count

There are mostly compact cars in our data followed by Midsize cars. There are just a few cars that are large compared to compact and midsize cars. This is typical of the real world data as we don't have a lot of cars that are large. We see a lot of cars to be compact and midsize in real life too! Great we are doing a good job selecting this data.

Missingno

In [12]:
```
msno.matrix(data, color = (0.5, 0.5, 0.5))
```

Out[12]: <AxesSubplot:>



We would be making the use of Missingno library from python. It is a very good graphical representation of missing values in our data. We see that there are many missing values in 'Market Category' feature. There are also a few missing values in 'Engine HP', 'Engine Cylinders' and 'Number of Doors' respectively. This library could be used to plot the missing values present in our data even when there is a huge data present. Therefore, we could use this library for understanding the missing values in our data.

Groupby

Groupby is a good useful function in python where the values are grouped based on feature (or) features that we give to the groupby function and things such as mean, median, mode or other aggregate functions could be performed once the data values are grouped together. Below are some of the plots which are made possible with the help of groupy function in python. For demonstration, I have created a groupby table below which shows the grouping of cars based on their make. In addition to this, there are various features that I have taken after grouping the values which are "Engine HP", "Engine Cylinders", "highway MPG" and "city mpg" features respectively.

Groupby with 'Make' feature

We would be making use of groupby which would take into consideration the feature that would be grouped on and it would perform different operations after grouping such as finding the minimum element in particular group, maximum element in a particular group and so on. Therefore, we would be making use of this in groupby as it makes life simple in python. Here, we see that we have grouped the data on the basis of 'Make' and considered a few features such as 'Engine HP', 'Engine Cylinders', 'highway MPG' and 'city mpg'. We would be then looking at the maximum values, minimum values and mean of the data. We could see a very good depiction of the result below.

In [13]:
```
data.groupby('Make')[['Engine HP', 'Engine Cylinders', 'highway MPG', 'city mpg']].agg(['min', 'max', 'mean'])
```

Out[13]:

	Engine HP			Engine Cylinders			highway MPG			city mpg		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
Make												
Acura	111.0	573.0	244.797619	4.0	6.0	5.333333	17	38	28.111111	13	39	19.940476
Alfa Romeo	237.0	237.0	237.000000	4.0	4.0	4.000000	34	34	34.000000	24	24	24.000000
Aston Martin	420.0	568.0	484.322581	8.0	12.0	10.623656	15	22	18.892473	9	14	12.526882
Audi	108.0	610.0	277.695122	4.0	12.0	5.557927	18	354	28.823171	11	31	19.585366
BMW	170.0	600.0	326.907186	0.0	12.0	5.958084	18	111	29.245509	10	137	20.739521
Bentley	400.0	631.0	533.851351	8.0	12.0	9.729730	14	25	18.905405	9	15	11.554054
Bugatti	1001.0	1001.0	1001.000000	16.0	16.0	16.000000	14	14	14.000000	8	8	8.000000
Buick	138.0	310.0	219.244898	4.0	8.0	5.316327	19	36	26.948980	14	28	18.704082
Cadillac	140.0	640.0	332.309824	4.0	8.0	6.433249	18	33	25.236776	12	22	17.355164
Chevrolet	55.0	650.0	246.972247	0.0	8.0	5.908118	15	110	25.815672	11	128	19.021371
Chrysler	100.0	385.0	229.139037	4.0	8.0	5.593583	17	36	26.368984	12	23	17.759358
Dodge	92.0	707.0	244.415335	4.0	10.0	6.258786	12	41	22.345048	10	28	16.065495
FIAT	101.0	180.0	143.559322	0.0	4.0	3.806452	29	108	37.338710	21	122	30.645161
Ferrari	400.0	731.0	511.956522	8.0	12.0	9.797101	12	23	15.724638	7	16	10.565217
Ford	63.0	662.0	243.097926	0.0	8.0	5.914869	13	99	24.006810	11	110	17.960272
GMC	105.0	420.0	259.844660	4.0	8.0	6.454369	13	32	21.403883	10	22	15.813592
Genesis	311.0	420.0	347.333333	6.0	8.0	6.666667	23	28	25.333333	15	18	16.333333
HUMMER	239.0	300.0	261.235294	5.0	8.0	6.058824	16	18	17.294118	13	14	13.529412
Honda	62.0	280.0	195.749441	0.0	6.0	4.659243	18	105	32.574610	14	132	25.443207
Hyundai	81.0	429.0	201.917492	4.0	8.0	4.666667	21	45	30.392739	15	40	22.343234
Infiniti	145.0	420.0	310.066667	4.0	8.0	6.151515	17	36	24.778788	12	29	17.827273
Kia	125.0	420.0	206.827434	0.0	8.0	4.588745	20	92	30.653680	15	120	23.848485

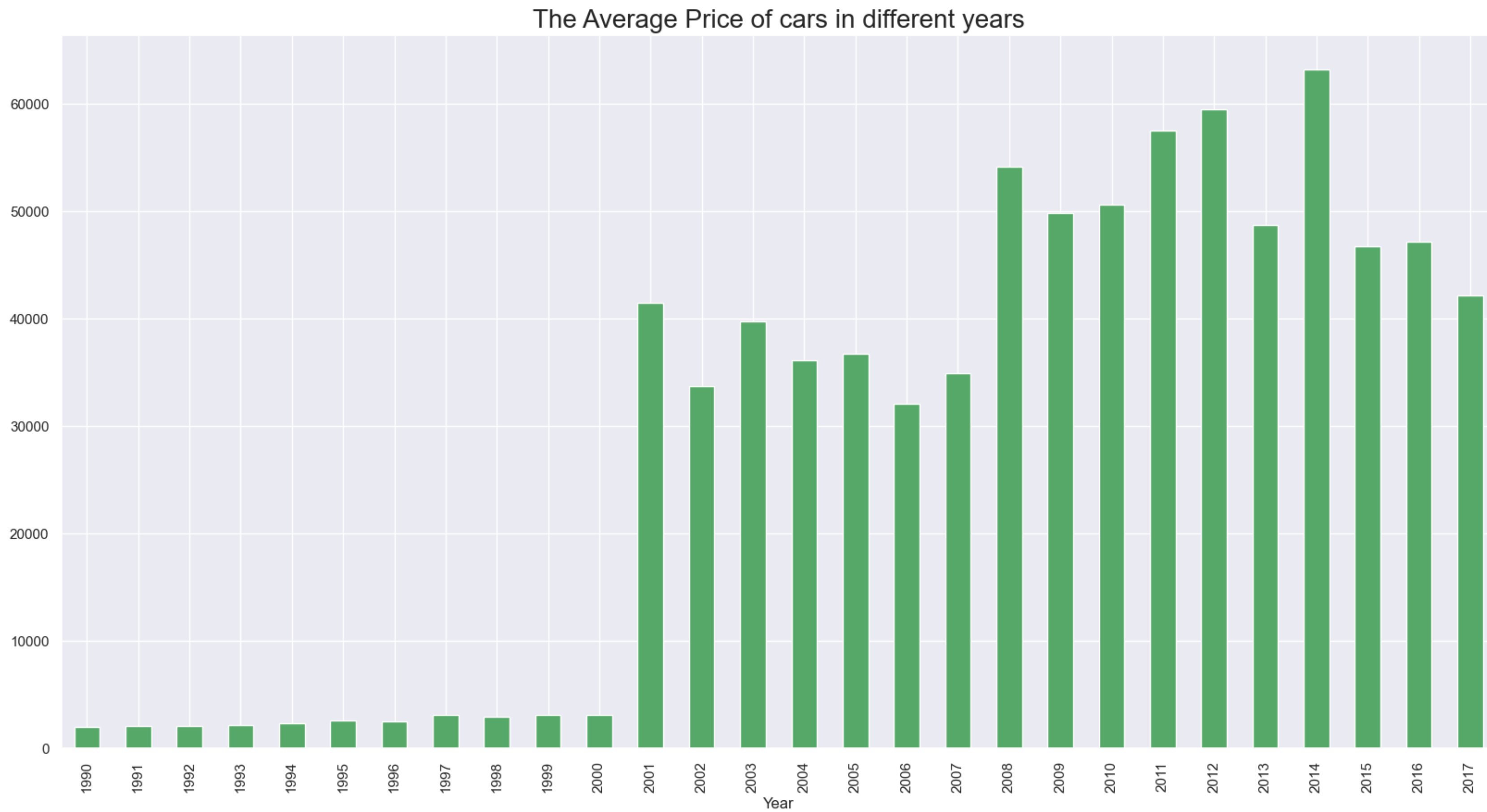
	Engine HP			Engine Cylinders			highway MPG			city mpg		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
Make												
Lamborghini	550.0	750.0	614.076923	10.0	12.0	10.884615	12	21	18.019231	8	15	11.519231
Land Rover	174.0	550.0	322.097902	4.0	8.0	6.125874	14	30	22.125874	11	22	16.230769
Lexus	134.0	552.0	277.415842	4.0	10.0	6.247525	14	40	25.876238	11	43	20.311881
Lincoln	188.0	380.0	284.910256	4.0	8.0	6.073171	15	39	24.487805	11	41	17.890244
Lotus	189.0	400.0	275.965517	4.0	8.0	5.241379	21	39	26.551724	14	21	18.758621
Maserati	345.0	523.0	420.793103	6.0	8.0	7.344828	15	25	20.293103	10	17	13.327586
Maybach	543.0	631.0	590.500000	12.0	12.0	12.000000	16	16	16.000000	10	10	10.000000
Mazda	82.0	274.0	171.992908	4.0	6.0	4.615385	17	41	27.851064	14	34	21.245863
McLaren	562.0	641.0	610.400000	8.0	8.0	8.000000	22	23	22.200000	15	16	15.600000
Mercedes-Benz	121.0	641.0	350.181818	0.0	12.0	6.711048	13	82	24.830028	10	85	18.181303
Mitsubishi	66.0	320.0	173.429245	3.0	8.0	4.680952	17	102	27.544601	13	126	21.910798
Nissan	90.0	600.0	239.921533	0.0	8.0	5.336918	17	101	27.799283	12	126	21.874552
Oldsmobile	110.0	275.0	177.466667	4.0	8.0	5.573333	19	31	26.233333	13	22	17.606667
Plymouth	92.0	253.0	131.560976	4.0	6.0	4.390244	21	36	27.963415	15	28	20.792683
Pontiac	74.0	415.0	190.295699	4.0	8.0	5.483871	19	37	27.069892	13	27	18.682796
Porsche	208.0	605.0	392.794118	4.0	10.0	6.132353	15	30	25.367647	9	21	17.470588
Rolls-Royce	322.0	624.0	487.548387	8.0	12.0	11.870968	15	21	19.129032	10	13	11.838710
Saab	150.0	390.0	220.522523	4.0	8.0	4.540541	16	33	26.351351	12	21	17.765766
Scion	94.0	200.0	154.433333	4.0	4.0	4.000000	28	42	32.300000	22	36	25.316667
Spyker	400.0	400.0	400.000000	8.0	8.0	8.000000	18	18	18.000000	13	13	13.000000
Subaru	66.0	305.0	197.308594	3.0	6.0	4.367188	21	38	28.683594	15	30	21.789062
Suzuki	66.0	261.0	160.287749	4.0	6.0	4.547009	19	39	26.034188	14	33	19.914530

	Engine HP			Engine Cylinders			highway MPG			city mpg		
	min	max	mean	min	max	mean	min	max	mean	min	max	mean
Make												
Tesla	NaN	NaN	NaN	0.0	0.0	0.000000	90	107	98.944444	86	102	94.111111
Toyota	93.0	381.0	236.147849	0.0	8.0	5.597315	17	74	26.453083	13	78	21.554960
Volkswagen	81.0	444.0	189.757726	4.0	12.0	4.365217	15	105	32.128554	11	126	23.580964
Volvo	114.0	345.0	230.971530	4.0	6.0	4.722420	19	38	27.202847	15	26	19.583630

Grouping the data on the basis of Year

We would now be grouping the data on the basis of year and check the average prices of cars for the years of cars. Looking at the plot below, we see that the average prices of cars was the highest in the year 2014 followed by the year 2012. The average prices of cars that are in the year 2000 and below are pretty low as can be easily seen from the plot. On average, we also find an interesting trend. As the years increase, we could see that the average prices of cars keep increasing but not in a steady way.

In [14]:
```
plt.figure(figsize=(20,10))
data.groupby('Year')['MSRP'].mean().plot(kind='bar', color='g')
plt.title("The Average Price of cars in different years", fontsize = 20)
plt.show()
```

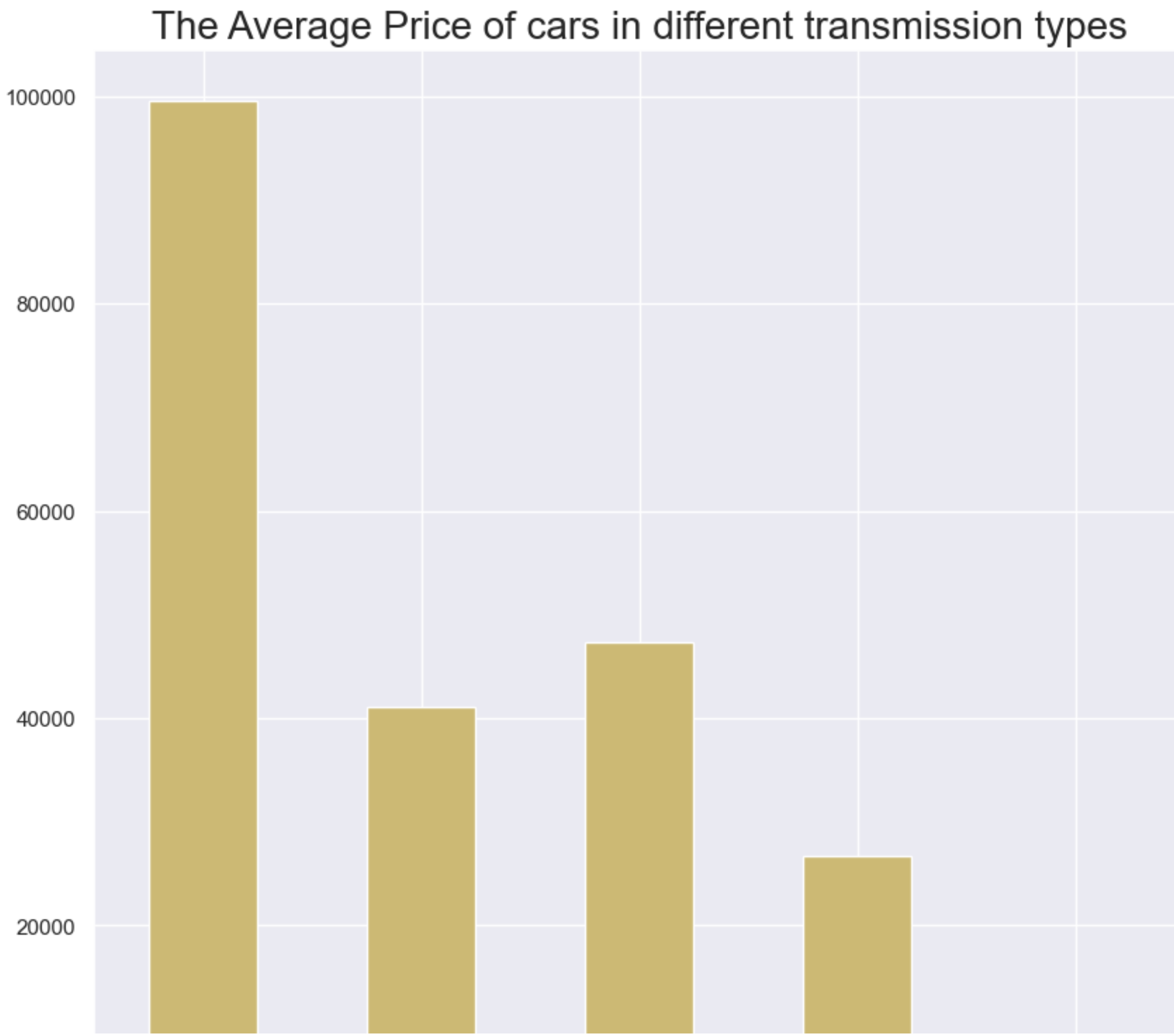


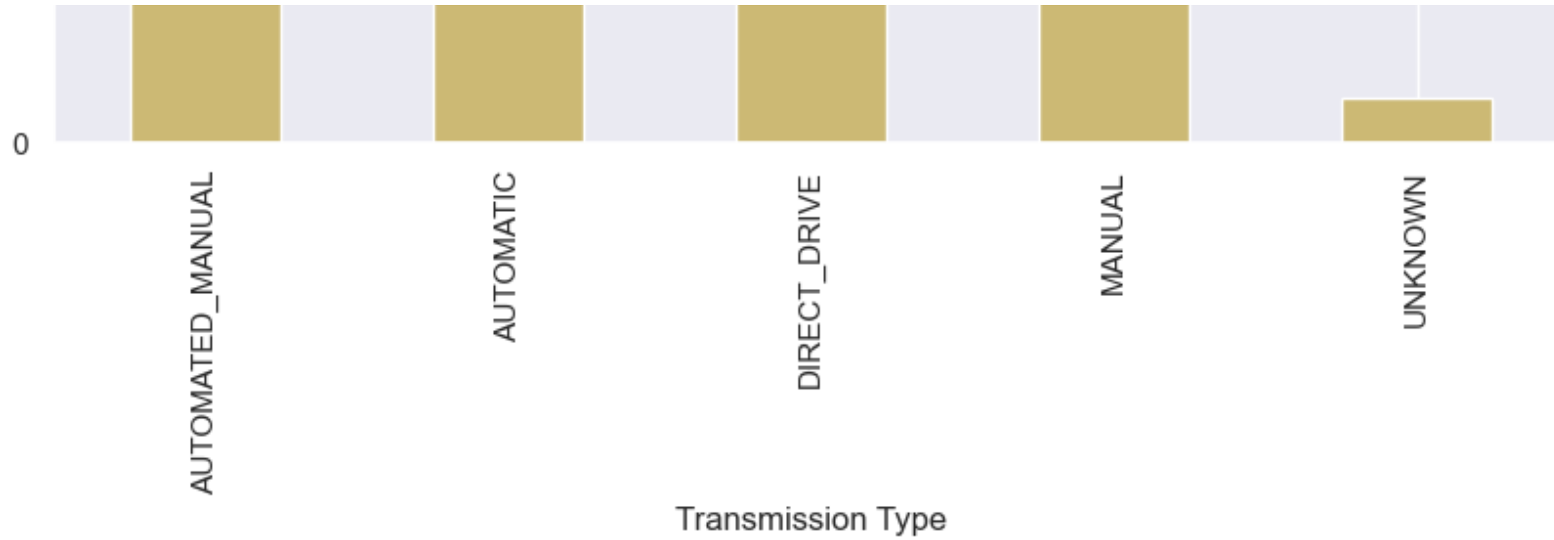
Grouping on the basis of Transmission Type

We would be grouping the data on the basis of transmission type and check the average prices of cars. We see that automated_manual cars have the highest average price. That is being followed by automatic cars. We expect the prices of cars that are manual to be low compared to the prices of cars that are automatic. That is being reflected in the graph below.

In [15]:

```
plt.figure(figsize=(10,10))
data.groupby('Transmission Type')['MSRP'].mean().plot(kind='bar', color='y')
plt.title("The Average Price of cars in different transmission types", fontsize = 20)
plt.show()
```


The Average Price of cars in different transmission types
100000
80000
60000
40000
20000

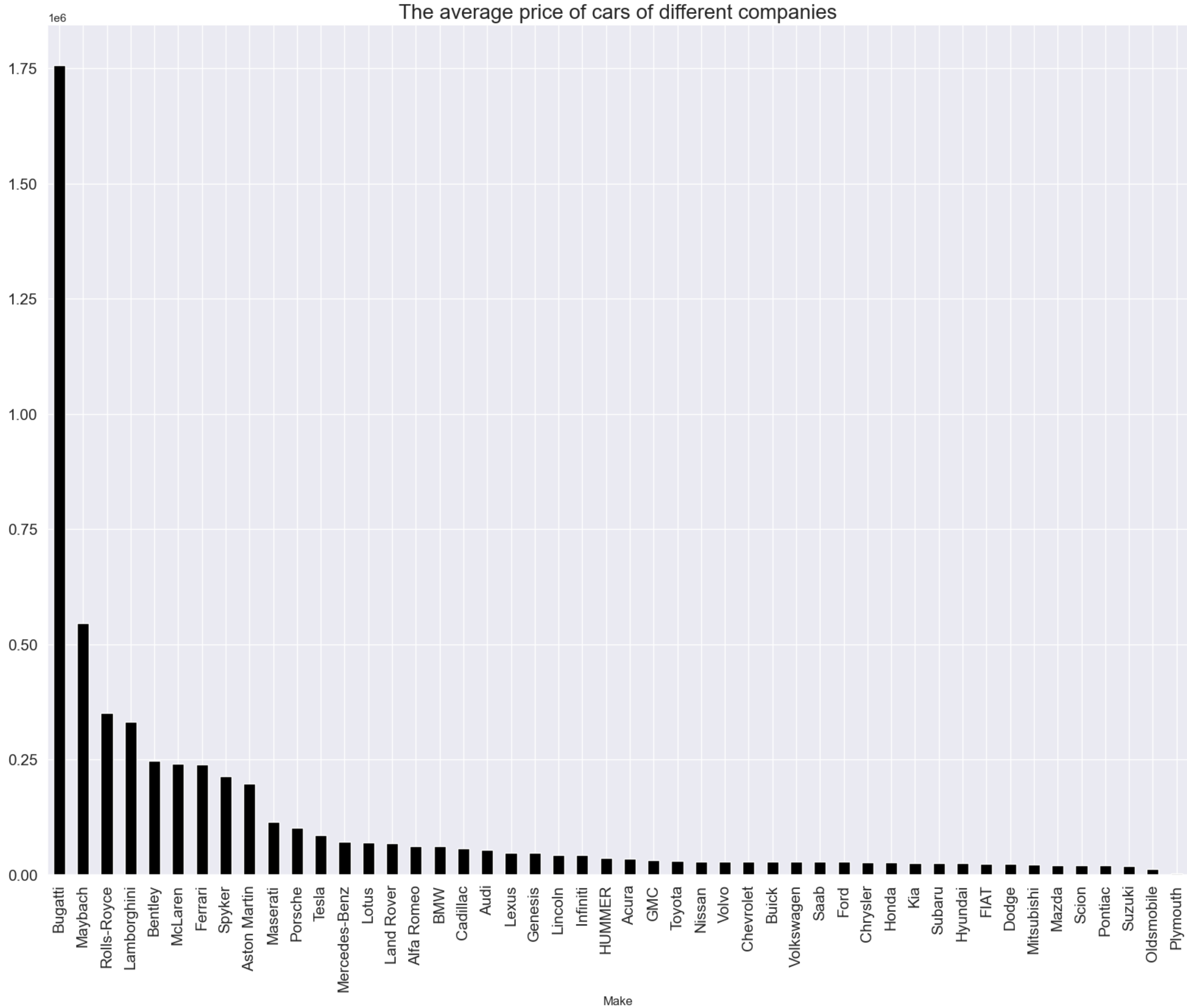


Grouping on basis of Make with 'MSRP' values

We would now be grouping on the basis of make and check the average prices of cars of particular makes. We should surely be expecting Bugatti to the most expensive car. In fact, it is the most expensive car in the world. Hope we buy the car anytime soon! (Just kidding). We see that the average price of Bugatti Veyron is about 1.75 million dollars. It is way too expensive compared to the other cars. There are other cars such as Maybach and Rolce-Royce that are also expensive if we remove Bugatti from our list. We see that the least expensive car is Plymouth.

In [16]:

```
plt.figure(figsize=(20,15))
data.groupby(['Make']).mean()['MSRP'].sort_values(ascending = False).plot(kind = 'bar', fontsize = 15, color = 'black
plt.title("The average price of cars of different companies", fontsize = 20)
plt.show()
```


The average price of cars of different companies
1e6
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Bugatti
Maybach
Rolls-Royce
Lamborghini
Bentley
McLaren
Ferrari
Spyker
Aston Martin
Maserati
Porsche
Tesla
Mercedes-Benz
Lotus
Land Rover
Alfa Romeo
BMW
Cadillac
Audi
Lexus
Genesis
Lincoln
Infiniti
HUMMER
Acura
GMC
Toyota
Nissan
Volvo
Chevrolet
Buick
Volkswagen
Saab
Ford
Chrysler
Honda
Kia
Subaru
Hyundai
FIAT
Dodge
Mitsubishi
Mazda
Scion
Pontiac
Suzuki
Oldsmobile
Plymouth
Make

Grouping on the basis of 'Make' with 'Engine Horse Power' Values

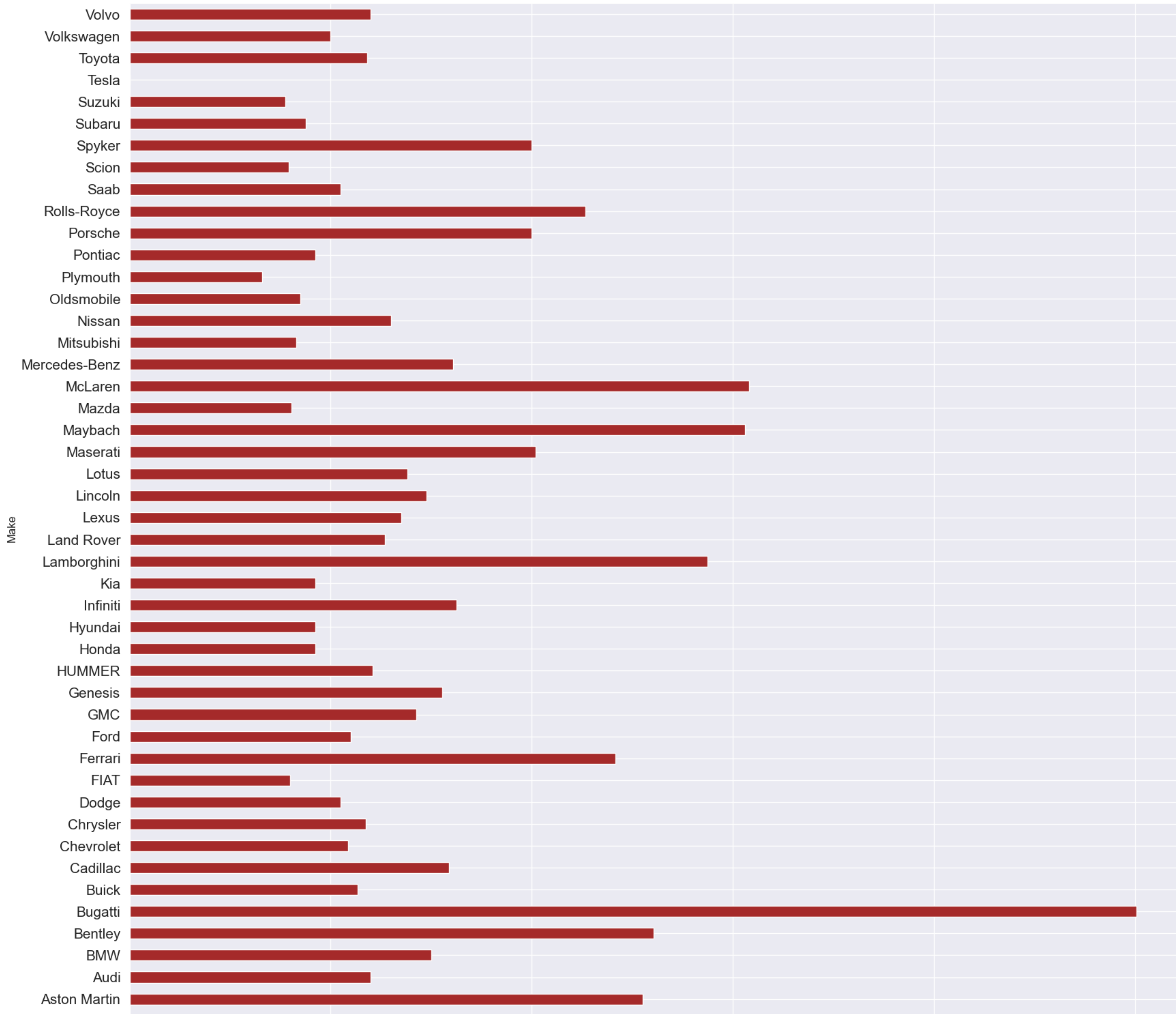
We would be grouping the data on the basis of Make and consider the 'Engine HP'. We should expect Bugatti to have the highest horse power (hp). We see that in the below graph. In addition, there are other car makers such as McLaren and Maybach which also contain a good horse power (hp). Since the horse power of 'tesla' is not known, we don't have a bar depicted below.

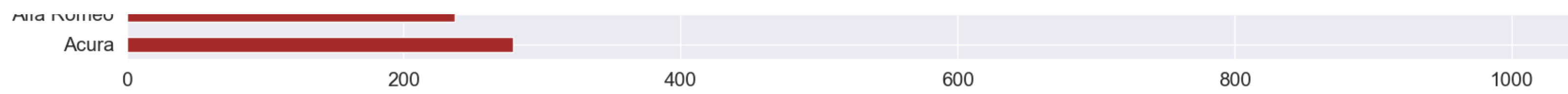
In [17]:

```
plt.figure(figsize = (20,20))
data.groupby('Make').median()['Engine HP'].plot(kind= 'barh', fontsize = 15, color = 'brown')
```

Out[17]:

```
<AxesSubplot:ylabel='Make'>
```


Volvo
Volkswagen
Toyota
Tesla
Suzuki
Subaru
Spyker
Scion
Saab
Rolls-Royce
Porsche
Pontiac
Plymouth
Oldsmobile
Nissan
Mitsubishi
Mercedes-Benz
McLaren
Mazda
Maybach
Maserati
Lotus
Lincoln
Lexus
Land Rover
Lamborghini
Kia
Infiniti
Hyundai
Honda
HUMMER
Genesis
GMC
Ford
Ferrari
FIAT
Dodge
Chrysler
Chevrolet
Cadillac
Buick
Bugatti
Bentley
BMW
Audi
Aston Martin
Make

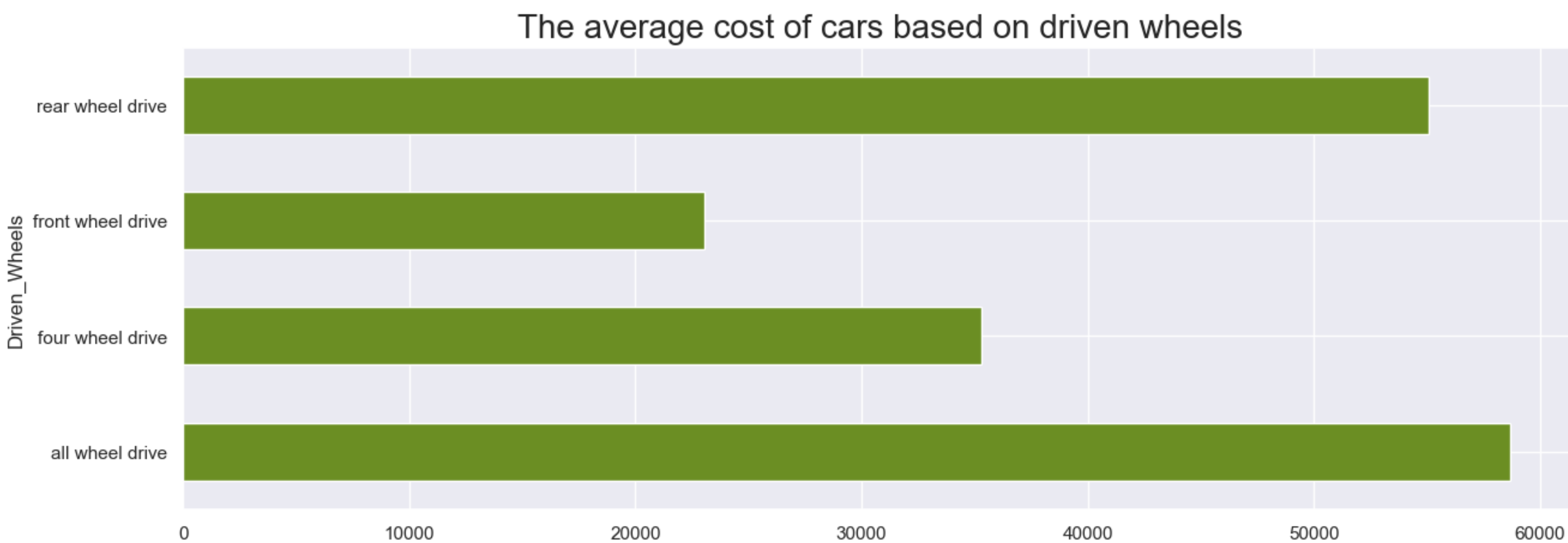


Grouping on the basis of Driven Wheels

We would now be dividing the data on the basis of Driven Wheels. We would then calculate the average prices of each group. We see that the average price of 'all wheel drive' cars is the highest followed by 'rear wheel drive' cars respectively. That is what should be expected as 'all wheel drive' cars are powerful and their cost is high. The 'front wheel drive' cars on the other hand are not that expensive as they don't have a lot of power. Thus, the data is reflective of the real-world data.

In [18]:

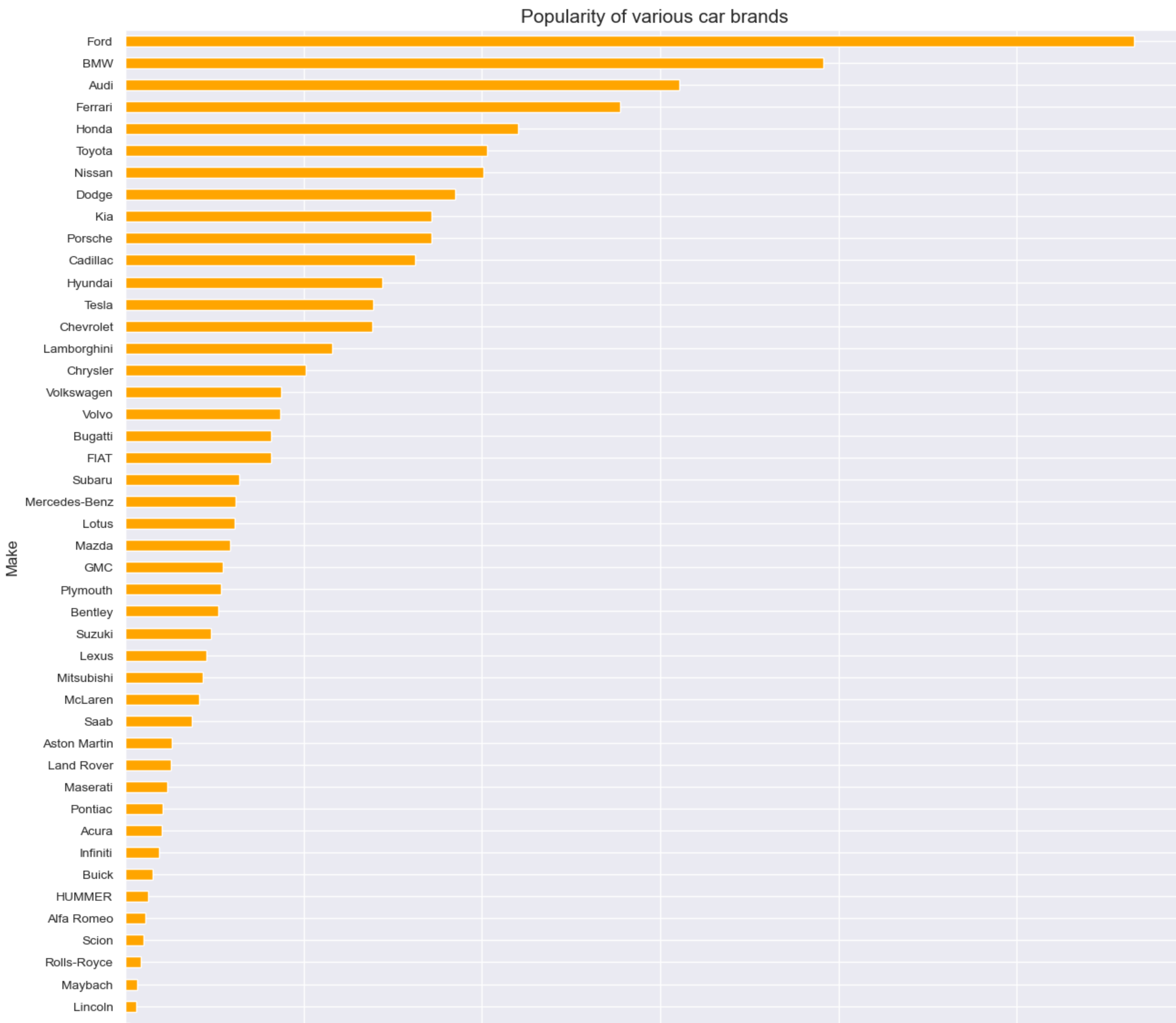
```
plt.figure(figsize = (15, 5))
data.groupby('Driven_Wheels').mean()['MSRP'].plot(kind= 'barh', color = 'olivedrab')
plt.title("The average cost of cars based on driven wheels", fontsize = 20)
plt.show()
```

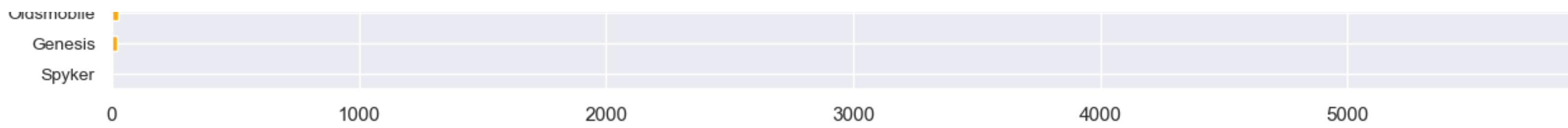


Grouping on the basis of Make with 'Popularity' values

We would group the data on the basis of make and find the average popularity of different cars. We see that 'Ford' is very popular all around our data. It is being followed by 'BMW' and 'Audi' respectively. We see that there are other car makers such as 'Lincoln' and 'Genesis' that are not so popular. 'Toyota' is also a popular brand and is present in the plot below.

In [19]:
```
plt.figure(figsize = (15, 15))
data.groupby('Make').mean()['Popularity'].sort_values(ascending = True).plot(kind = 'barh', color = 'orange')
plt.yticks(fontsize = 10)
plt.title("Popularity of various car brands", fontsize = 15)
plt.show()
```

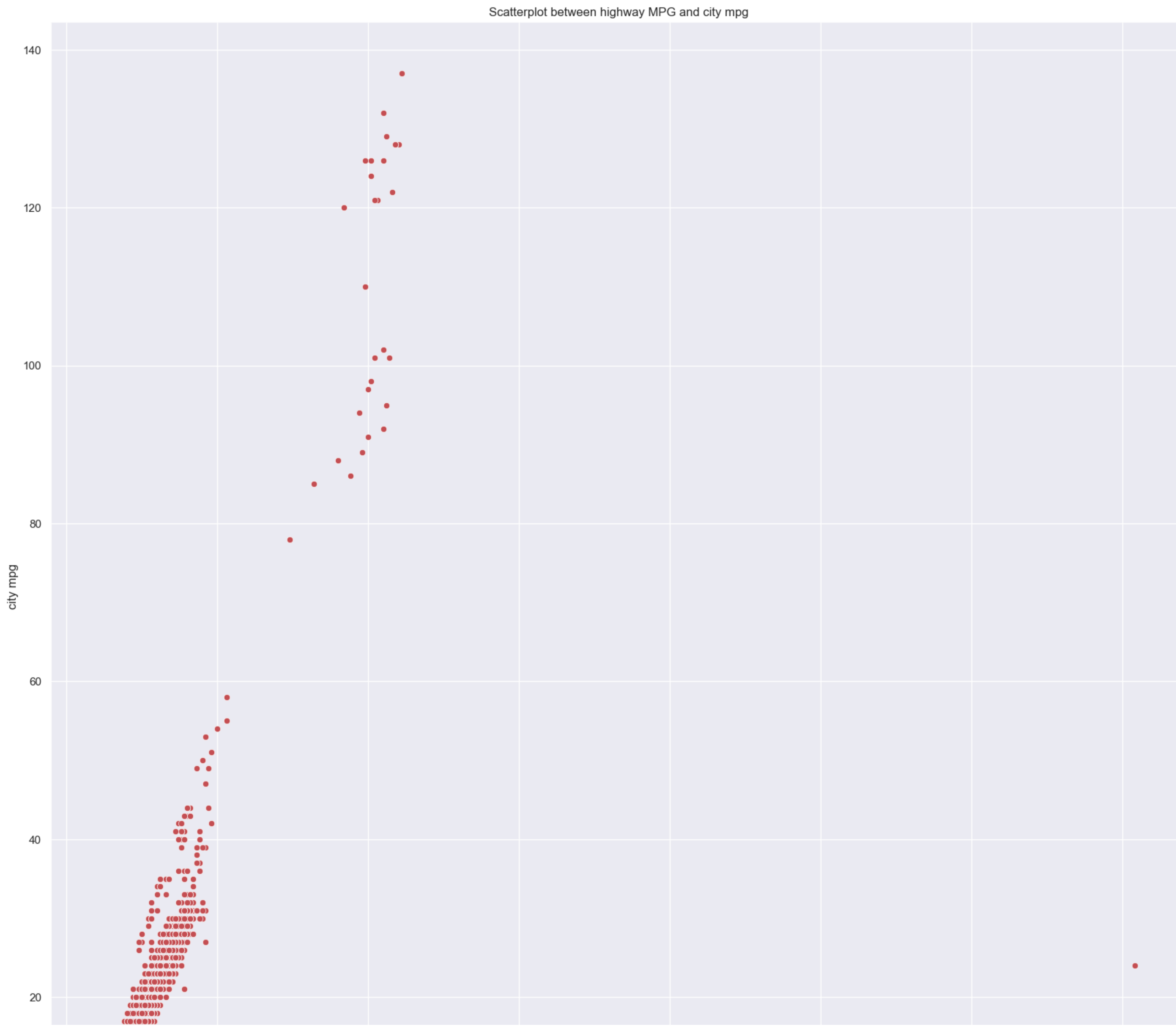

Popularity of various car brands
Make
Ford
BMW
Audi
Ferrari
Honda
Toyota
Nissan
Dodge
Kia
Porsche
Cadillac
Hyundai
Tesla
Chevrolet
Lamborghini
Chrysler
Volkswagen
Volvo
Bugatti
FIAT
Subaru
Mercedes-Benz
Lotus
Mazda
GMC
Plymouth
Bentley
Suzuki
Lexus
Mitsubishi
McLaren
Saab
Aston Martin
Land Rover
Maserati
Pontiac
Acura
Infiniti
Buick
HUMMER
Alfa Romeo
Scion
Rolls-Royce
Maybach
Lincoln



2.4 Scatterplot between 'highway MPG' and 'city mpg'

We should be expecting a linear relationship between 'highway MPG' and 'city mpg' as they are very much correlated with each other. We cannot have cars, in general, that have a city mileage that is very much different from highway mileage. In the plot below, we see that there is one outlier where the highway MPG is about 350. There are no cars that have that high mileage. We can remove the outlier as it would affect our results as errors in the data are costly when performing the machine learning operations.

In [20]:
```
sns.scatterplot(x = 'highway MPG', y = 'city mpg', data = data, color ='r')
plt.title("Scatterplot between highway MPG and city mpg")
plt.show()
```


Scatterplot between highway MPG and city mpg
city mpg
140
120
100
80
60
40
20

We would be removing the outlier in our data where the highway MPG is about 350.

In [21]:
```
data[data['highway MPG'] > 350]
```

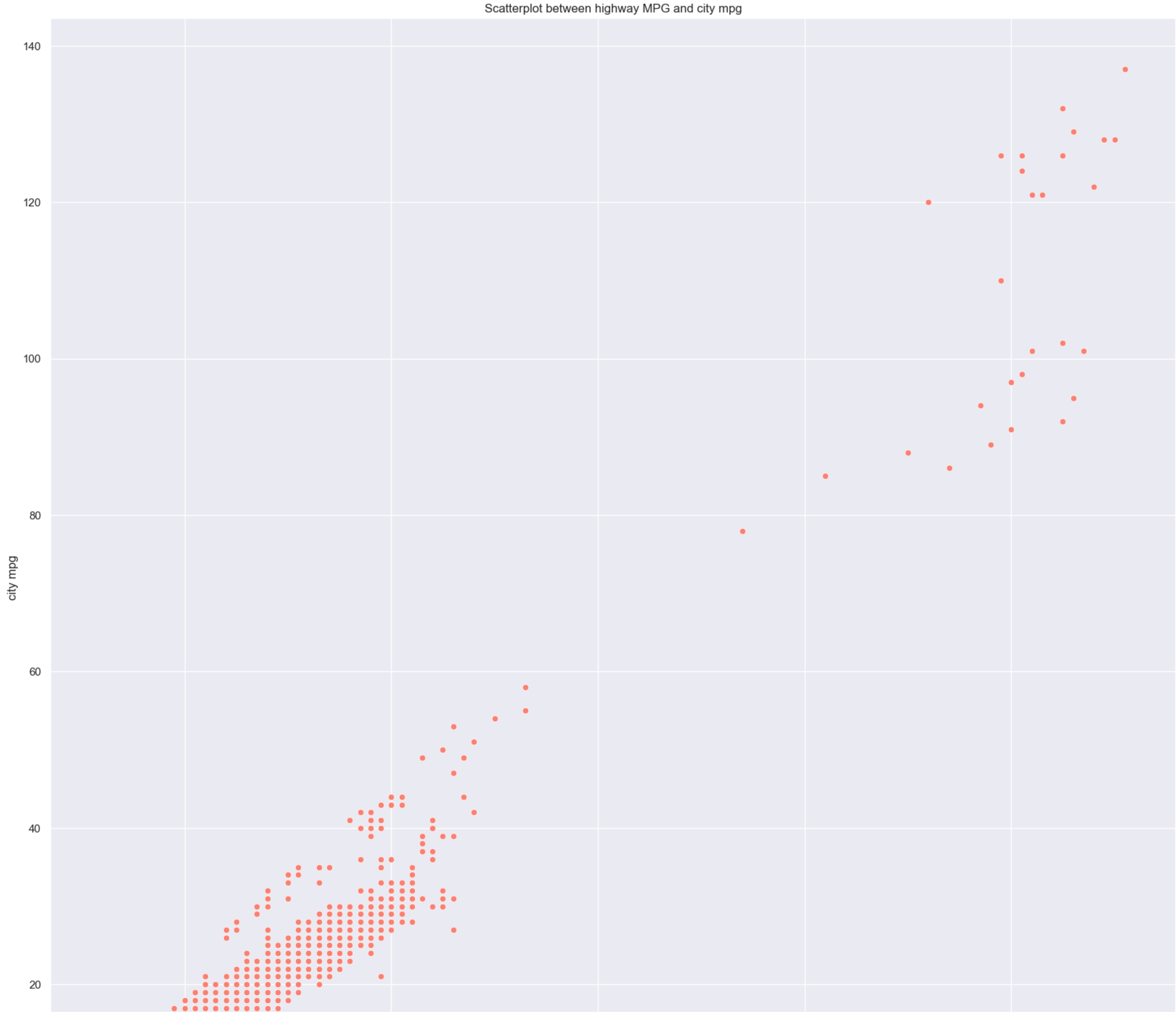
Out[21]:

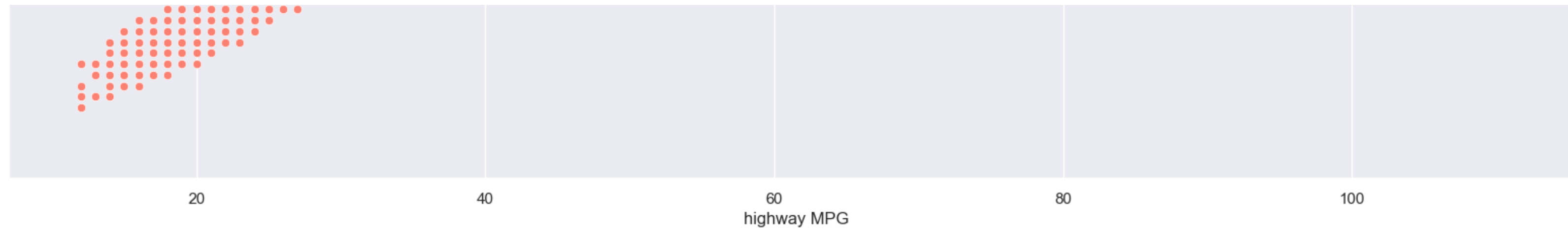
	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors	Market Category	Vehicle Size	Vehicle Style
1119	Audi	A6	2017	premium unleaded (recommended)	252.0	4.0	AUTOMATED_MANUAL	front wheel drive	4.0	Luxury	Midsize	Sedan

In [22]:
```
data = data[data['highway MPG'] < 350]
```

We would now be using a scatterplot as the above but with removing the outliers. We see almost a linear line between the two features that we have considered and that is what is expected.

In [23]:
```
sns.scatterplot(x = 'highway MPG', y = 'city mpg', data = data, color = 'salmon')
plt.title("Scatterplot between highway MPG and city mpg")
plt.show()
```


Scatterplot between highway MPG and city mpg
140
120
100
80
60
40
20
city mpg



We would check all the unique values in 'Market Category'. We see that there are so many different unique values.

In [24]:
```
data['Market Category'].unique()
```

Out[24]:
```
array(['Factory Tuner,Luxury,High-Performance', 'Luxury,Performance',
       'Luxury,High-Performance', 'Luxury', 'Performance', 'Flex Fuel',
       'Flex Fuel,Performance', nan, 'Hatchback',
       'Hatchback,Luxury,Performance', 'Hatchback,Luxury',
       'Luxury,High-Performance,Hybrid', 'Diesel,Luxury',
       'Hatchback,Performance', 'Hatchback,Factory Tuner,Performance',
       'High-Performance', 'Factory Tuner,High-Performance',
       'Exotic,High-Performance', 'Exotic,Factory Tuner,High-Performance',
       'Factory Tuner,Performance', 'Crossover', 'Exotic,Luxury',
       'Exotic,Luxury,High-Performance', 'Exotic,Luxury,Performance',
       'Factory Tuner,Luxury,Performance', 'Flex Fuel,Luxury',
       'Crossover,Luxury', 'Hatchback,Factory Tuner,Luxury,Performance',
       'Crossover,Hatchback', 'Hybrid', 'Luxury,Performance,Hybrid',
       'Crossover,Luxury,Performance,Hybrid',
       'Crossover,Luxury,Performance',
       'Exotic,Factory Tuner,Luxury,High-Performance',
       'Flex Fuel,Luxury,High-Performance', 'Crossover,Flex Fuel',
       'Diesel', 'Hatchback,Diesel', 'Crossover,Luxury,Diesel',
       'Crossover,Luxury,High-Performance',
       'Exotic,Flex Fuel,Factory Tuner,Luxury,High-Performance',
       'Exotic,Flex Fuel,Luxury,High-Performance',
       'Exotic,Factory Tuner,Luxury,Performance', 'Hatchback,Hybrid',
       'Crossover,Hybrid', 'Hatchback,Luxury,Hybrid',
       'Flex Fuel,Luxury,Performance', 'Crossover,Performance',
       'Luxury,Hybrid', 'Crossover,Flex Fuel,Luxury,Performance',
       'Crossover,Flex Fuel,Luxury', 'Crossover,Flex Fuel,Performance',
       'Hatchback,Factory Tuner,High-Performance', 'Hatchback,Flex Fuel',
       'Factory Tuner,Luxury',
       'Crossover,Factory Tuner,Luxury,High-Performance',
       'Crossover,Factory Tuner,Luxury,Performance',
       'Crossover,Hatchback,Factory Tuner,Performance',
       'Crossover,Hatchback,Performance', 'Flex Fuel,Hybrid',
       'Flex Fuel,Performance,Hybrid',
       'Crossover,Exotic,Luxury,High-Performance',
       'Crossover,Exotic,Luxury,Performance', 'Exotic,Performance',
       'Exotic,Luxury,High-Performance,Hybrid', 'Crossover,Luxury,Hybrid',
       'Flex Fuel,Factory Tuner,Luxury,High-Performance',
       'Performance,Hybrid', 'Crossover,Factory Tuner,Performance',
       'Crossover,Diesel', 'Flex Fuel,Diesel',
       'Crossover,Hatchback,Luxury'], dtype=object)
```

2.5 Boxplot

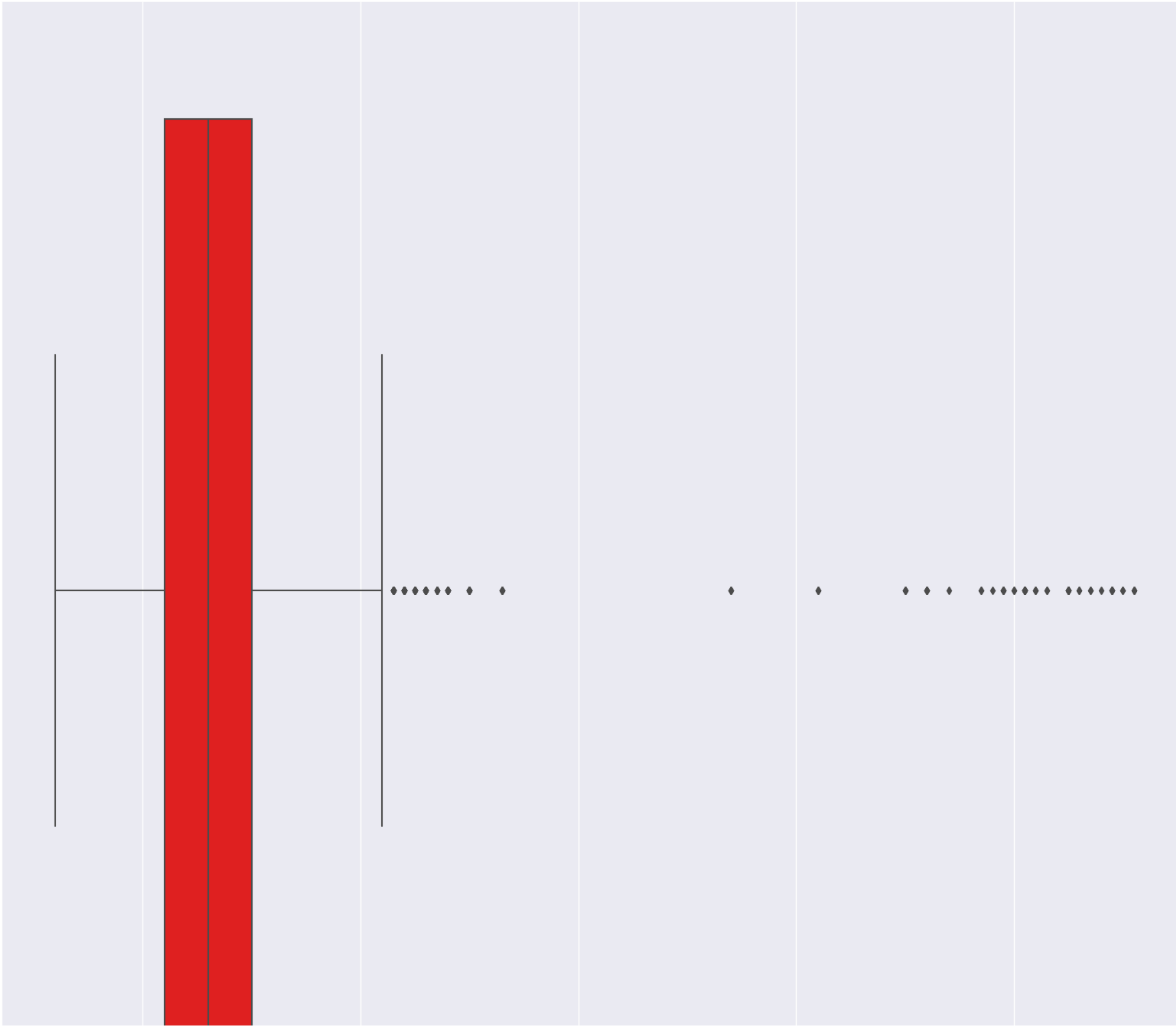
Boxplots give us a good understanding of how the data values are spread for different features. We could get to know the 25th, 50th and 75th percentile values present in different features. In addition, the outliers could also be detected by making use of a formula and considering the interquartile range which is the difference between the 75th percentile and 25th percentile respectively.

2.5.1 Boxplot of highway MPG

We would now be using the boxplot of highway MPG and calculate the average values and how the distribution is spread. We see that the average values are about 25 for highway MPG and we see the maximum value being equal to about 40 and the points above that to be outliers. We see that the data is not so spread as most of the values lie between 21 and 30 respectively.

In [25]:
```
sns.boxplot(x = 'highway MPG', data = data, color = 'red')
```

Out[25]:
```
<AxesSubplot:xlabel='highway MPG'>
```



Calculating percentiles of highway MPG

We would now print the highway percentile values to get a better understanding of the outliers in our data. We see that we cannot distinguish the outliers in the below print statements. We would be using more granularity and then calculate the percentile values and spot the outliers in our data.

In [26]:
```
for i in range(90, 100):
    print("The {:.1f}th percentile value is {:.2f}".format(i, np.percentile(data['highway MPG'], i)))
```

```
The 90.0th percentile value is 35.00
The 91.0th percentile value is 36.00
The 92.0th percentile value is 36.00
The 93.0th percentile value is 37.00
The 94.0th percentile value is 37.00
The 95.0th percentile value is 38.00
The 96.0th percentile value is 39.00
The 97.0th percentile value is 40.00
The 98.0th percentile value is 42.00
The 99.0th percentile value is 46.00
```

In [27]:
```
for i in [x*0.1 for x in range (990, 1000)]:
    print("The {:.1f}th percentile value is {:.2f}".format(i, np.percentile(data['highway MPG'], i)))
```

```
The 99.0th percentile value is 46.00
The 99.1th percentile value is 46.00
The 99.2th percentile value is 48.00
The 99.3th percentile value is 48.00
The 99.4th percentile value is 50.00
The 99.5th percentile value is 85.52
The 99.6th percentile value is 97.35
The 99.7th percentile value is 101.00
The 99.8th percentile value is 103.35
The 99.9th percentile value is 107.09
```

We see that 99.5th percentile values and so on have very high values and can be considered as outliers. Therefore, we have to remove those outliers so that they don't disturb our data and machine learning algorithms perform well with the data once the outliers are removed.

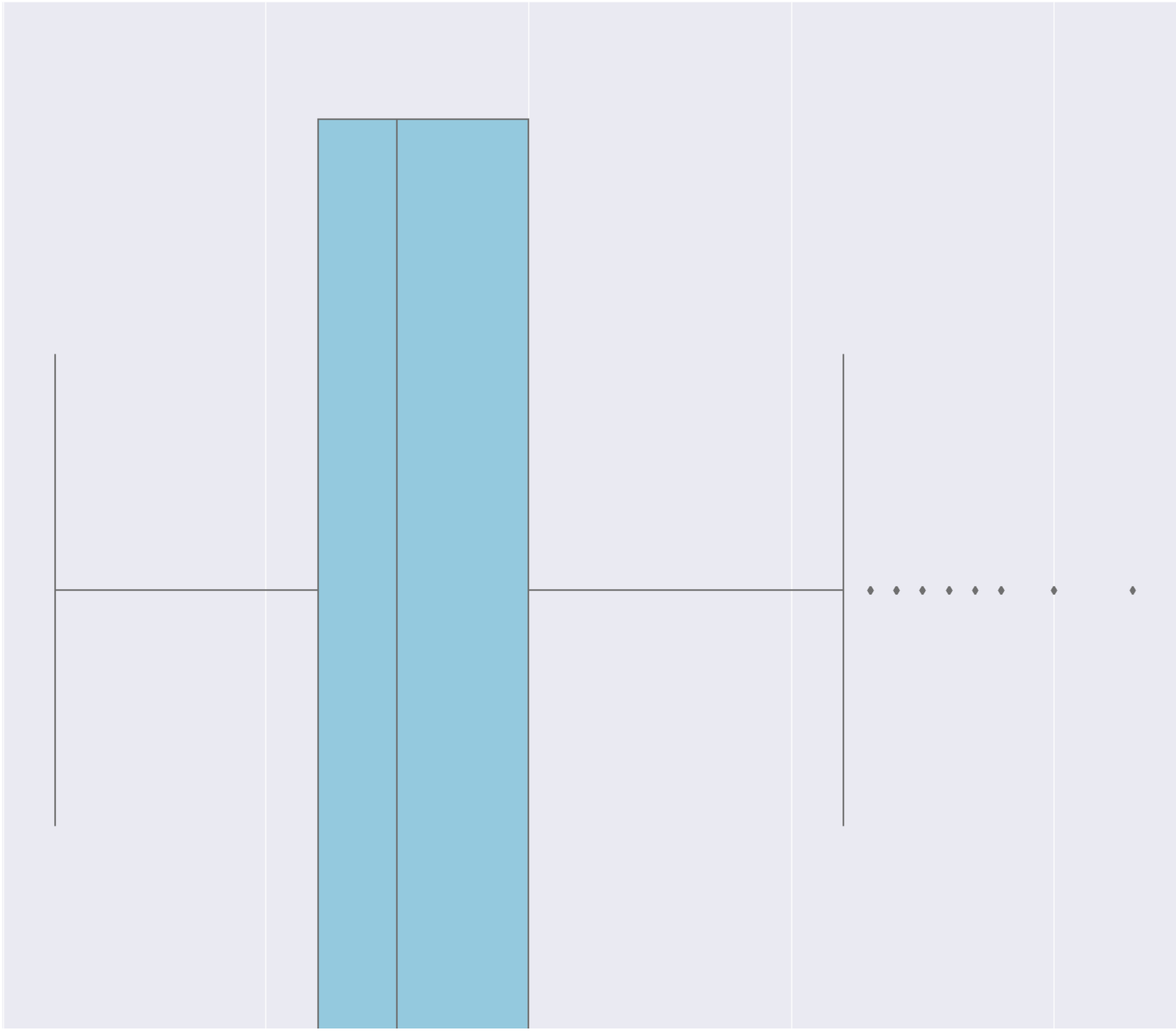
We would be removing the outlier values in our data. We would set the bar to be equal to about 60 respectively.

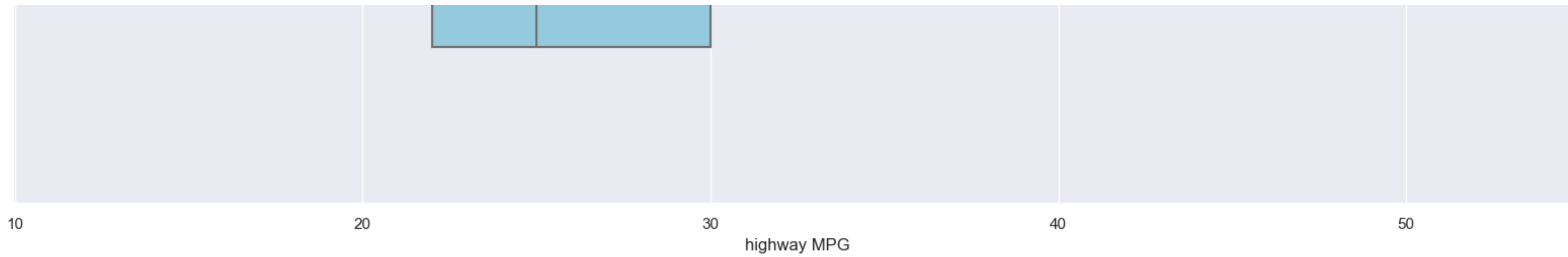
In [28]:
```
data = data[data['highway MPG'] < 60]
```

We would once again plot the boxplot and see how the values are split for highway MPG. We find that the highway MPG is more skewed towards the right. We see a lot of values to the right of the mean. What this means is that more than 50 percent of the values are above 24 (mean).

In [29]:
```
sns.boxplot(x= 'highway MPG', data = data, color = 'skyblue')
```

Out[29]:
```
<AxesSubplot:xlabel='highway MPG'>
```



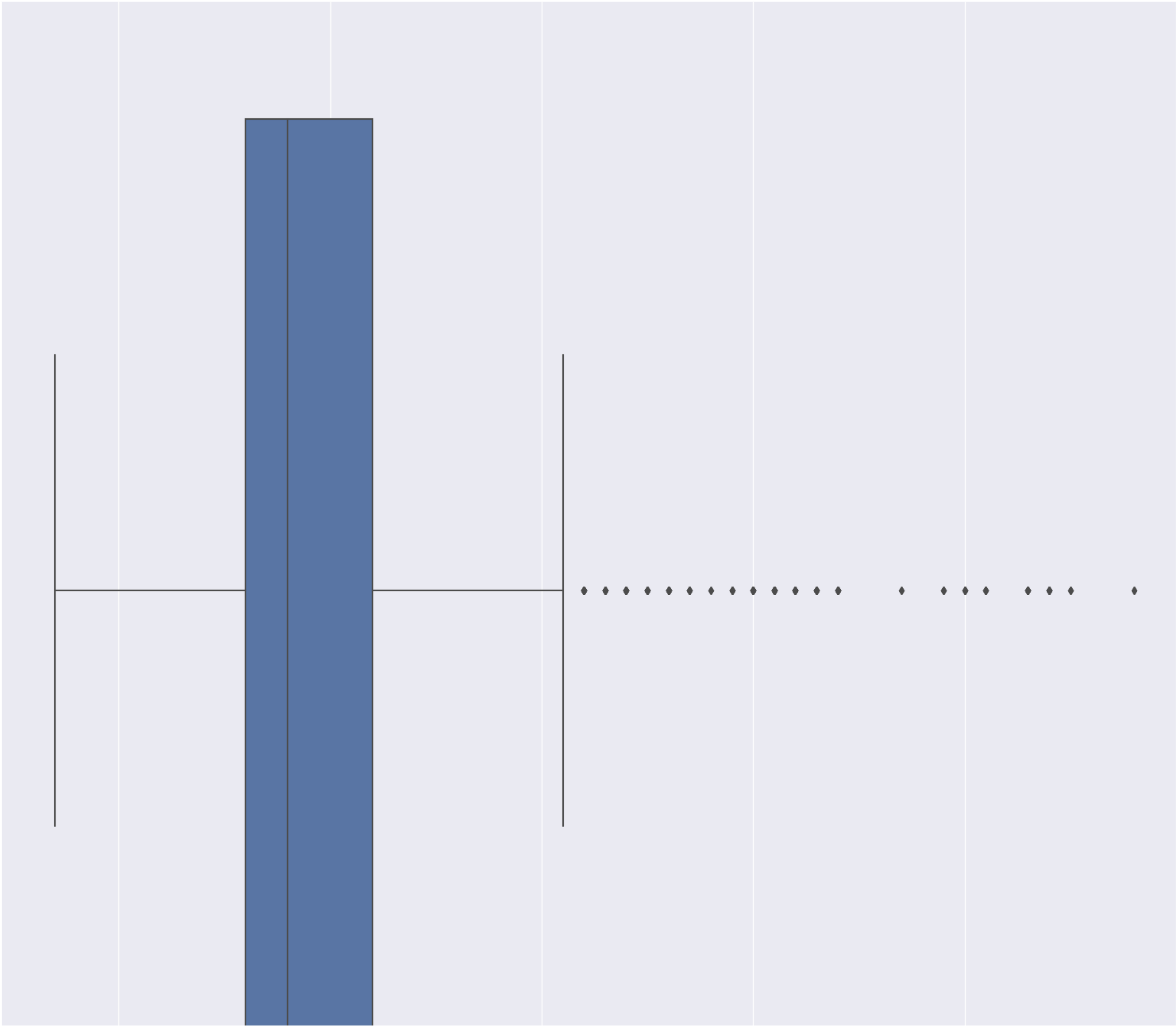


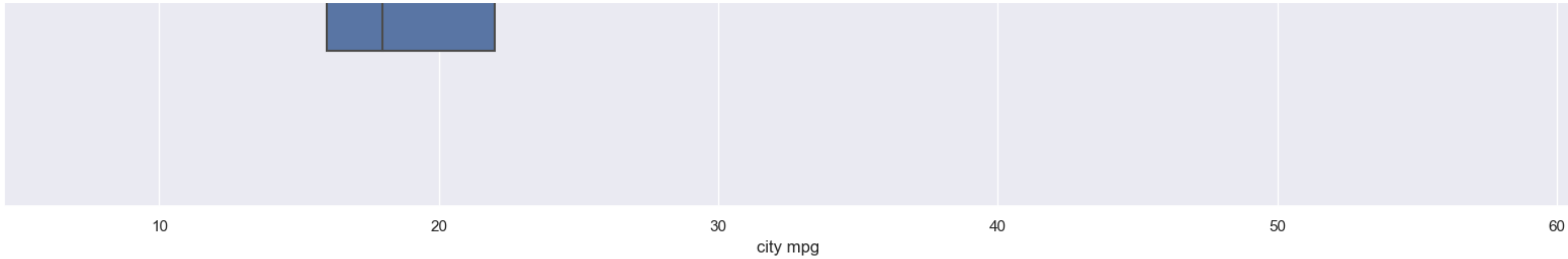
Boxplot of city mpg

We would do the similar above operation for 'city mpg' respectively. We also see below some outliers that might interfere in our predictions. Therefore, we would delete those values.

In [30]:
```
sns.boxplot(x = 'city mpg', data=data)
```

Out[30]:
```
<AxesSubplot:xlabel='city mpg'>
```





We would now be getting percentile values and check the outliers

In [31]:
```
for i in range (90, 100):
    print("The {:.1f}th percentile value is {:.2f}".format(i, np.percentile(data['city mpg'], i)))
```

```
The 90.0th percentile value is 26.00
The 91.0th percentile value is 26.00
The 92.0th percentile value is 27.00
The 93.0th percentile value is 27.00
The 94.0th percentile value is 28.00
The 95.0th percentile value is 29.00
The 96.0th percentile value is 30.00
The 97.0th percentile value is 31.00
The 98.0th percentile value is 32.00
The 99.0th percentile value is 41.00
```

We would be checking the outliers and calculate their values

In [32]:
```
for i in [x * 0.1 for x in range (990, 1000)]:
    print("The {:.1f}th percentile value is {:.2f}".format(i, np.percentile(data['city mpg'], i)))
```

```
The 99.0th percentile value is 41.00
The 99.1th percentile value is 41.00
The 99.2th percentile value is 41.00
The 99.3th percentile value is 42.00
The 99.4th percentile value is 43.00
The 99.5th percentile value is 44.00
The 99.6th percentile value is 44.00
The 99.7th percentile value is 50.00
The 99.8th percentile value is 53.00
The 99.9th percentile value is 54.00
```

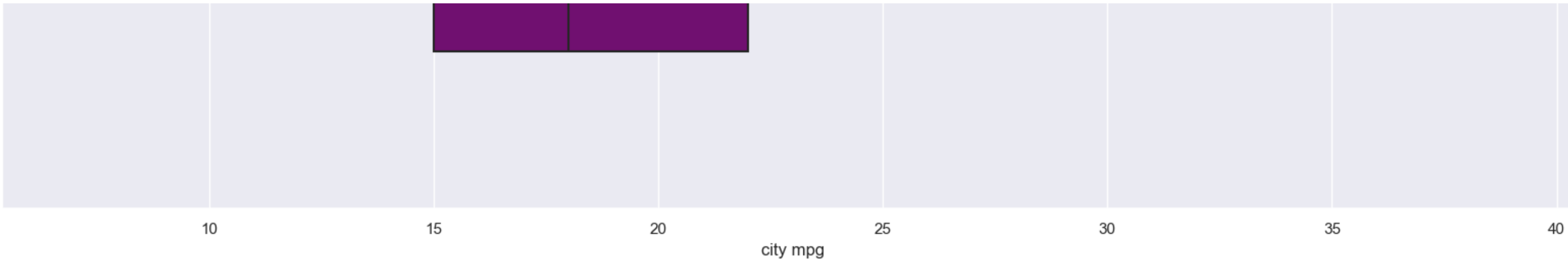
We would be removing the outliers and put the bar equal to 40 respectively.

In [33]:
```
data = data[data['city mpg'] < 40]
```

We would once again plot the boxplot of 'city mpg' respectively. We see again that the data is right skewed.

In [34]:
```
sns.boxplot(x = 'city mpg', data = data, color = 'purple')
```

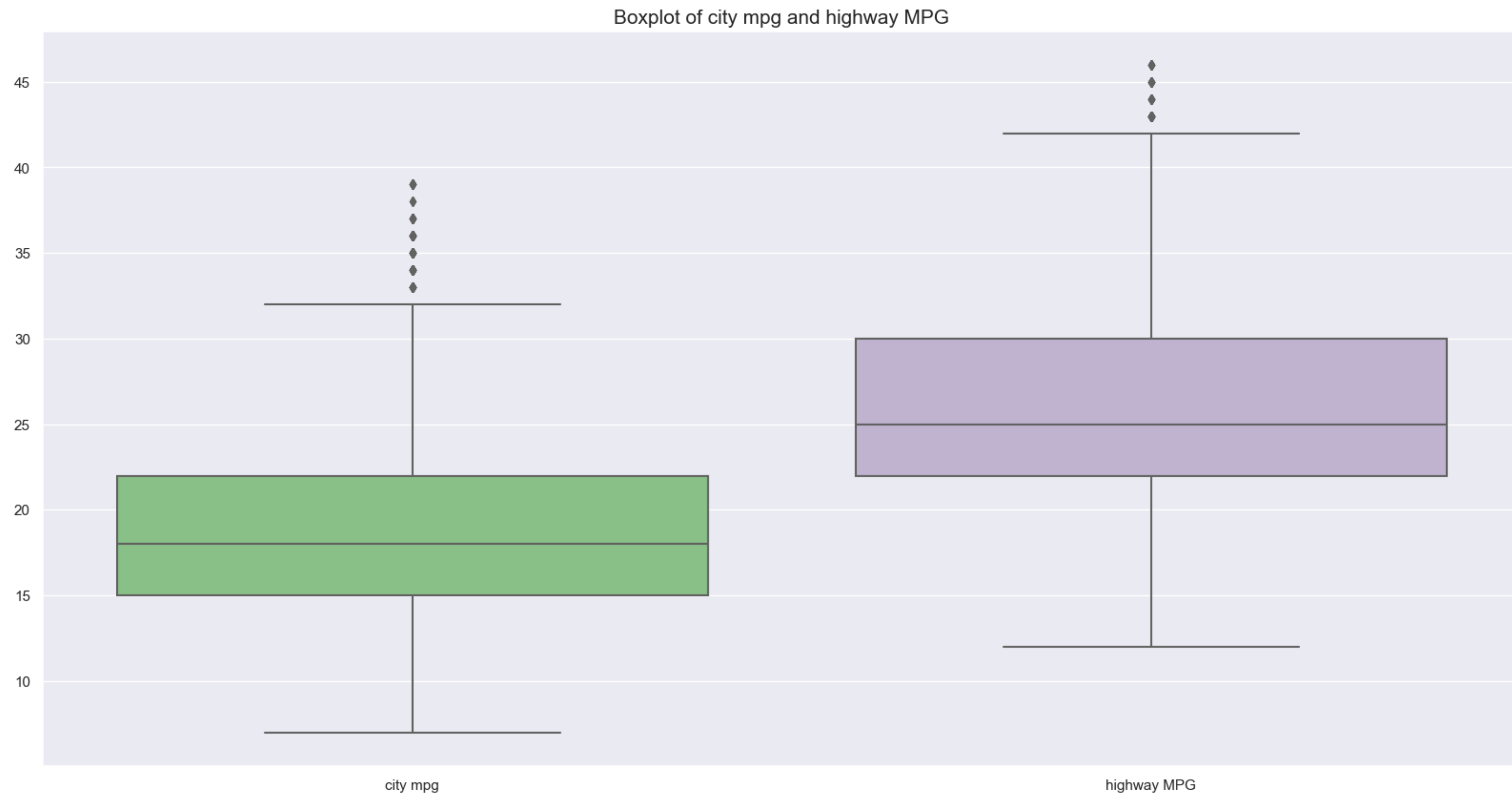
Out[34]:
```
<AxesSubplot:xlabel='city mpg'>
```



2.5.3 Boxplot of 2 features 'city mpg' and 'highway MPG'

We would be looking at the 2 features 'city mpg' and 'highway MPG' respectively. We see that in terms of 'city mpg' most of the values that are present are in the range between 15 to 22 respectively. On the other hand, we find that most of the values that are present in 'highway MPG' are in the range 22 to 30 respectively. Therefore, we can see how the values are spread in the boxplot and see there the spread actually take place by comparing the features.

In [35]:
```
plt.figure(figsize = (20, 10))
sns.boxplot(data = data[['city mpg', 'highway MPG']], palette = 'Accent')
plt.title("Boxplot of city mpg and highway MPG", fontsize = 15)
plt.show()
```

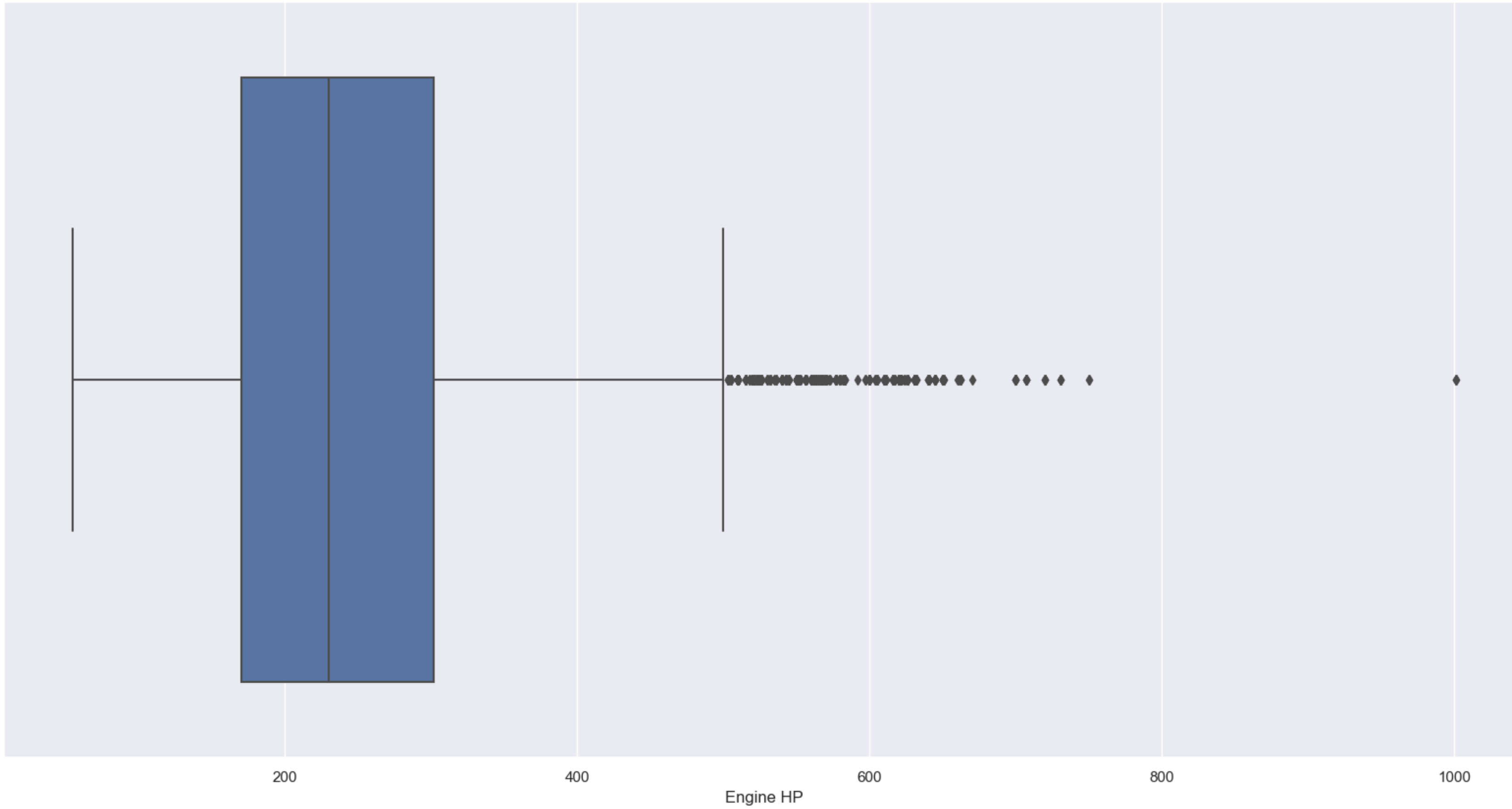


2.5.4 Boxplot of 'Engine HP'

We would be making use of the boxplot and then seeing the spread of the values 'Engine HP' and get an idea about how the values are spread. We see that most of the values of 'Engine HP' would lie between 150 to 300 respectively. The maximum value of the engine horsepower would be something like 500 while the remaining values that are higher are considered to be outliers. The boxplot also looks as though it is right skewed where there are more number of values of 'Engine HP' that are higher than the mean value of about 250 (mean) respectively.

In [36]:
```
plt.figure(figsize = (20, 10))
sns.boxplot(data['Engine HP'])
```

Out[36]: <AxesSubplot:xlabel='Engine HP'>

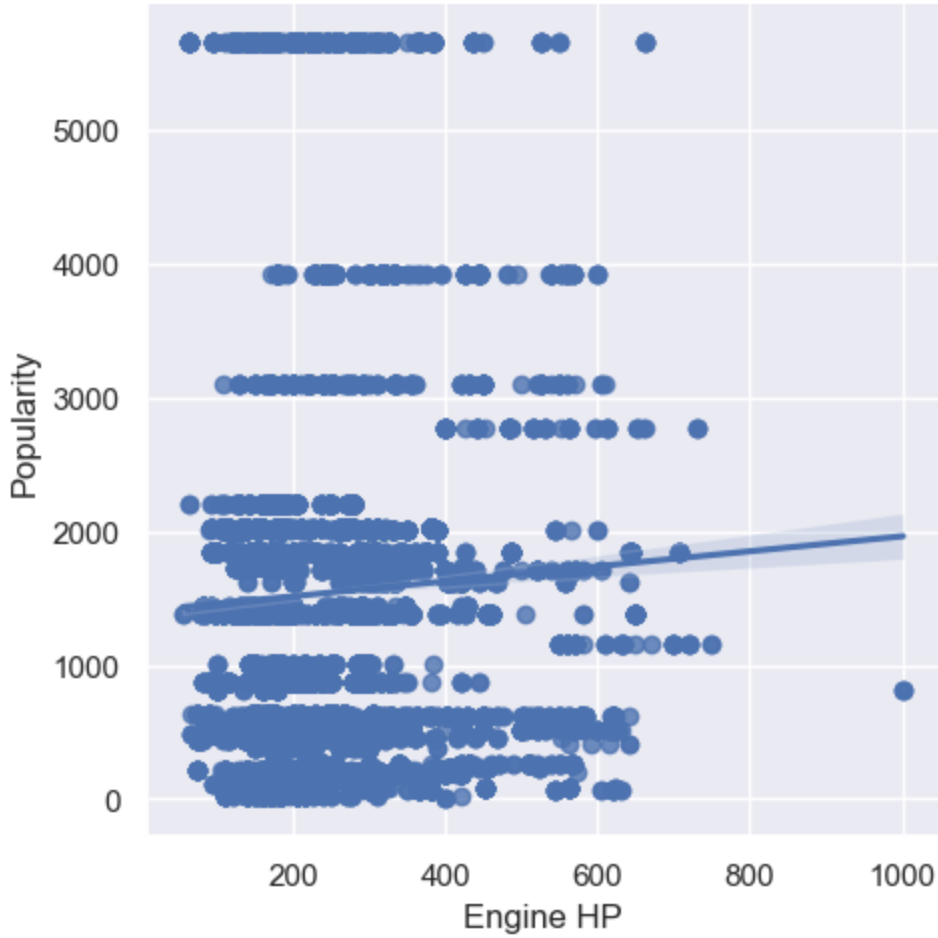


2.6.1 lmplot between 'Engine HP' and 'Popularity'

We would be using lmplot and checking the relationship between 'Engine HP' and 'Popularity' respectively. We see that most of the data is not related. One thing to note, however, is that there is a linear line which has a positive slope. What this means is that with the increase in 'Engine HP', there is a higher chance of increase in 'Popularity' respectively. This need not be true in all the cases as correlation need not always be equal to causation.

In [37]:
```
sns.set(rc = {'figure.figsize': (20, 20)})
sns.lmplot(x = 'Engine HP', y = 'Popularity', data = data)
```

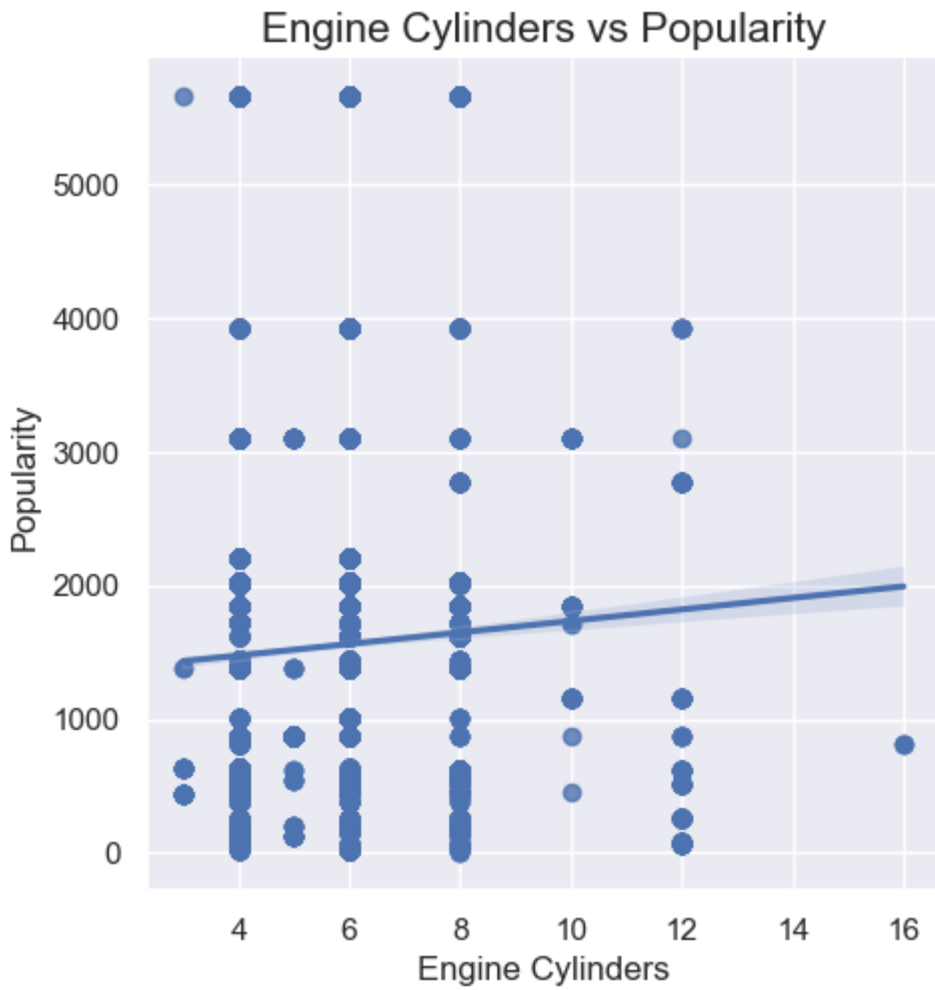
Out[37]:
```
<seaborn.axisgrid.FacetGrid at 0x1ed3d162e50>
```



2.6.2 lmplot between 'Engine Cylinders' and 'Popularity'

We would now be plotting the lmplot between 'Engine Cylinders' and 'Popularity' and see if there is any relationships between the parameters taken into consideration. We see that there is a relationship between the 'Popularity' and 'Engine Cylinders' and there is a positive relationship between parameters. We see that the data and the features that we have considered are correlated with each other. But that should again not be confused with causation as having higher number of 'Engine Cylinders' does not cause the car to be more popular and increase the 'Popularity'.

In [38]:
```
sns.lmplot(x = 'Engine Cylinders', y = 'Popularity', data = data)
plt.title("Engine Cylinders vs Popularity", fontsize = 15)
plt.show()
```

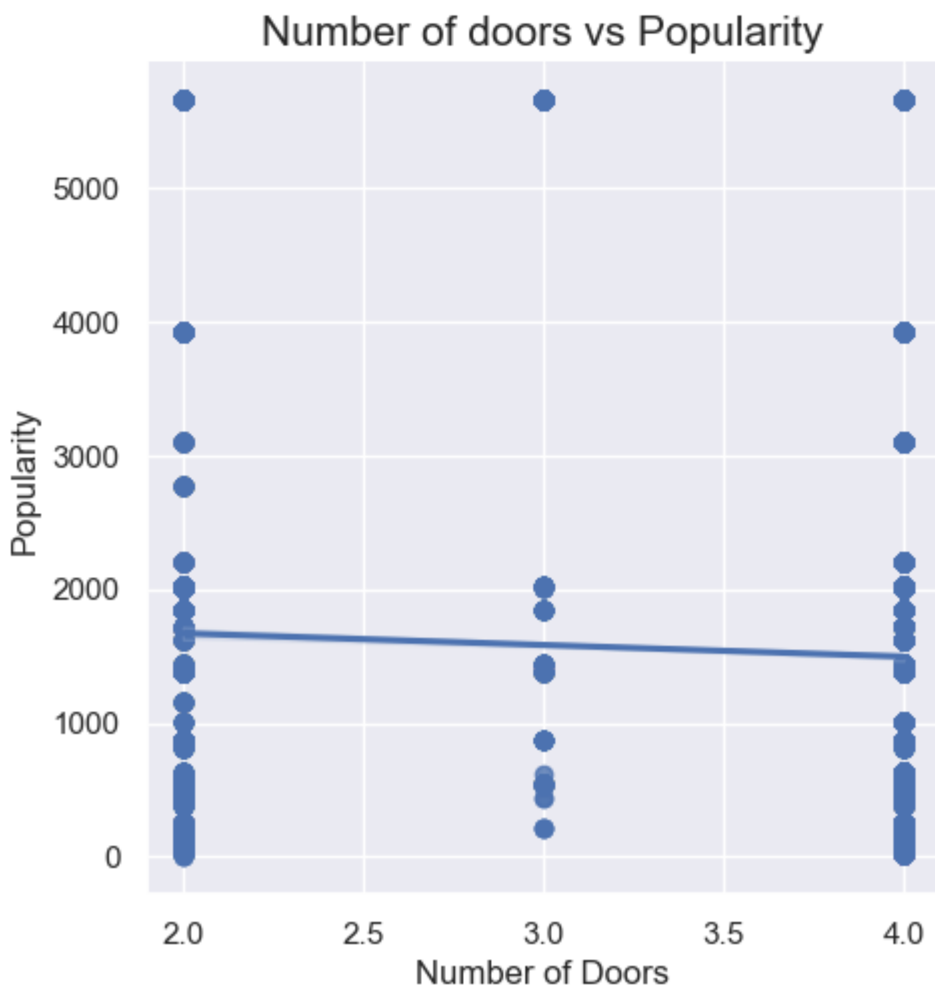


2.6.3 lmplot between 'Number of Doors' and 'Popularity'

We see that there is a line that has a negative slope on the relationship between the parameters. We see that 'Popularity' and the 'Number of Doors' are not related with each other. In general, we see that the more the number of doors of the car, the less the popularity. That is true in real life as well as we see that the cars that are highly popular have just 2 doors. Some of the cars include Bugatti Veyron and Lamborghini Gallardo. Thus, this data is reflective of the real world data set that we have taken into consideration.

In [39]:

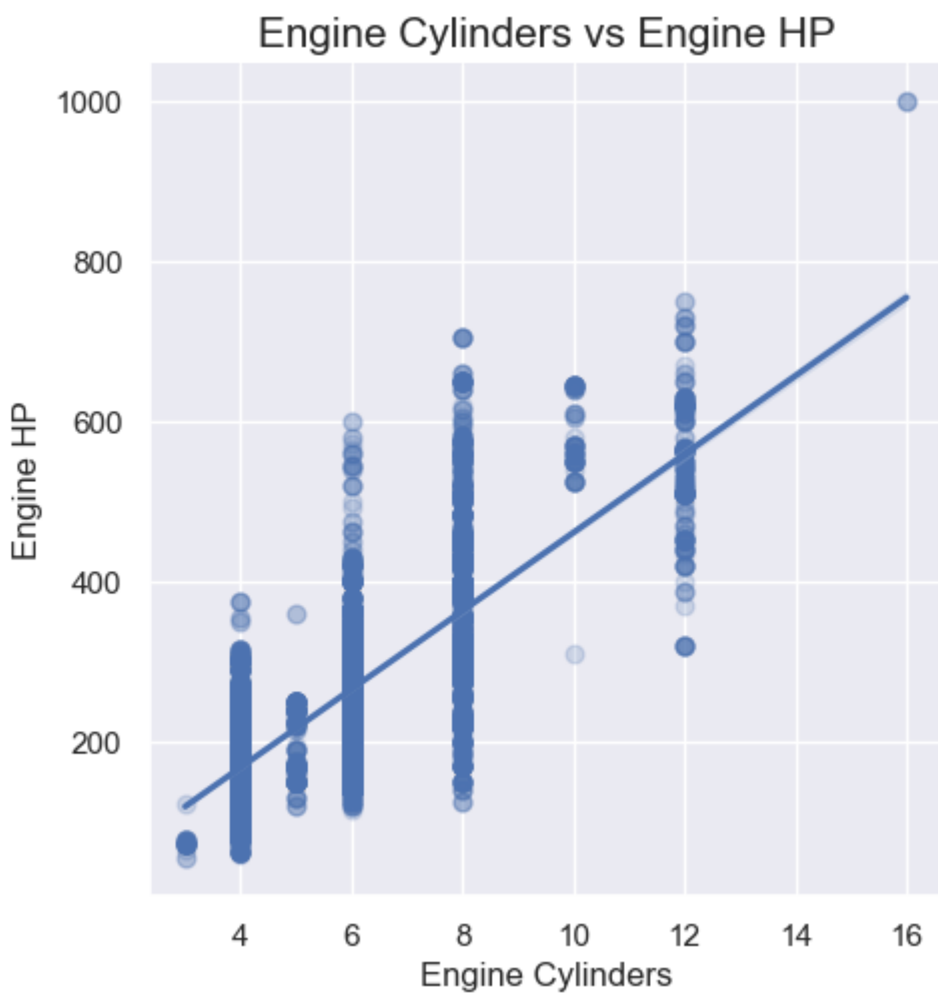
```
sns.lmplot(x = 'Number of Doors', y = 'Popularity', data = data)
plt.title("Number of doors vs Popularity", fontsize = 15)
plt.show()
```



2.6.4 lmplot between 'Engine Cylinders' and 'Engine HP'

We see that there is a very good correlation between 'Engine Cylinders' and 'Engine HP' as can be seen from the plot below. That's the reason we gave an almost perfect linearly drawn line. Therefore, lmplot could be used to see the linear relationship or the correlation between the 2 features under consideration respectively.

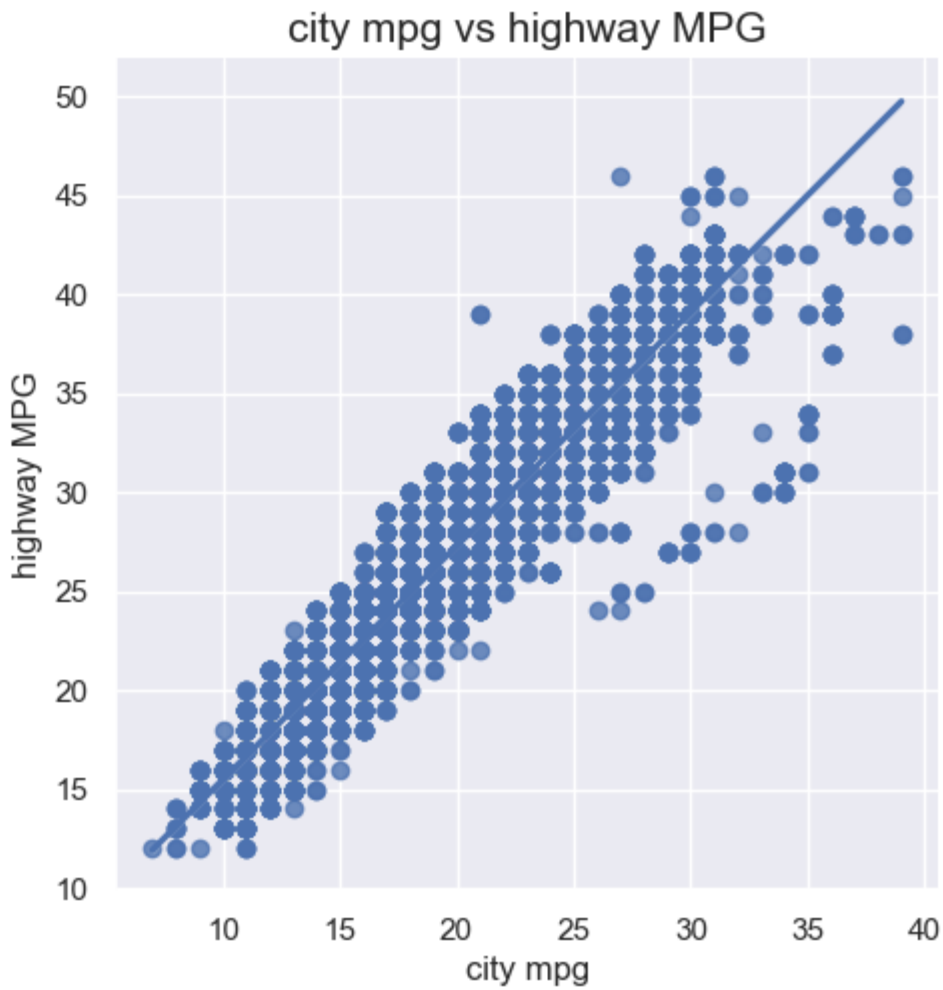
In [40]:
```
sns.lmplot(x = 'Engine Cylinders', y = 'Engine HP', scatter_kws = {"s": 40, "alpha": 0.2}, data = data)
plt.title("Engine Cylinders vs Engine HP", fontsize = 15)
plt.show()
```



2.6.5 lmplot between 'city mpg' and 'highway MPG'

We see that there is a linear relationship between features that we have taken into consideration. In real life, we see the same being reflected. Therefore, we are working with a real world data set as most of the features are what we find in the real-world.

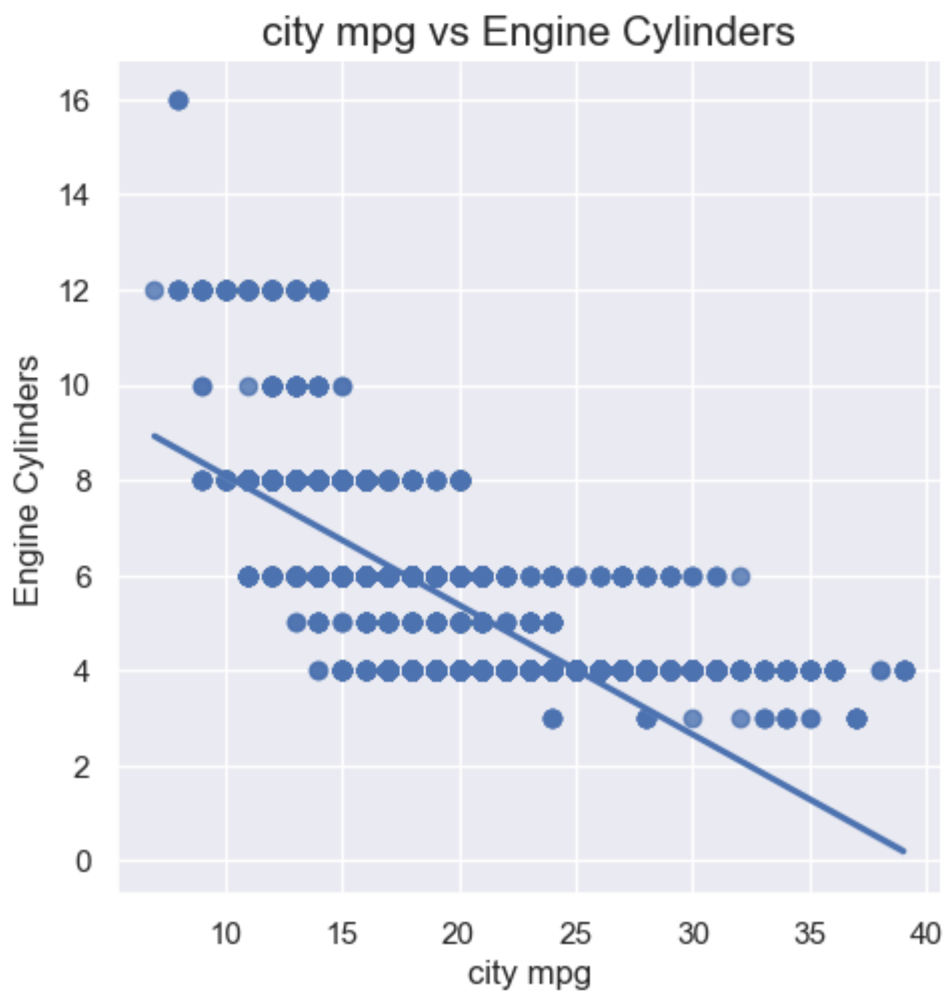
In [41]:
```
sns.lmplot(x = 'city mpg', y = 'highway MPG', data = data)
plt.title("city mpg vs highway MPG", fontsize = 15)
plt.show()
```



2.6.6 lmplot between 'city mpg' and 'Engine Cylinders'

We see that there is an inverse relationship between 'city mpg' and 'Engine Cylinders' respectively. That is what we typically find in real-life. We see that as there is an increase in the number of engine cylinders, there is a higher possibility for the car under consideration to be lower in terms of city mileage. That is what is being reflected in our plot.

In [42]:
```
sns.lmplot(x = 'city mpg', y = 'Engine Cylinders', data = data)
plt.title("city mpg vs Engine Cylinders", fontsize = 15)
plt.show()
```



2.7 Heatmap

One of the cool features of python is the heatmap. We would be able to consider some of the important values that are present such as 'Engine HP', 'Engine Cylinders' and 'Number of Doors'. We would be taking the features that are numerical and we would be using the plots and see the correlation between them. We see that 'highway MPG' and 'city mpg' are highly correlated. That is the reason that we got a value of about 0.94 respectively. In addition to this, we see that 'Engine Horsepower' and 'Engine Cylinders' are correlated. That is true as having a higher number of cylinders would ensure that there is a high horsepower on a car.

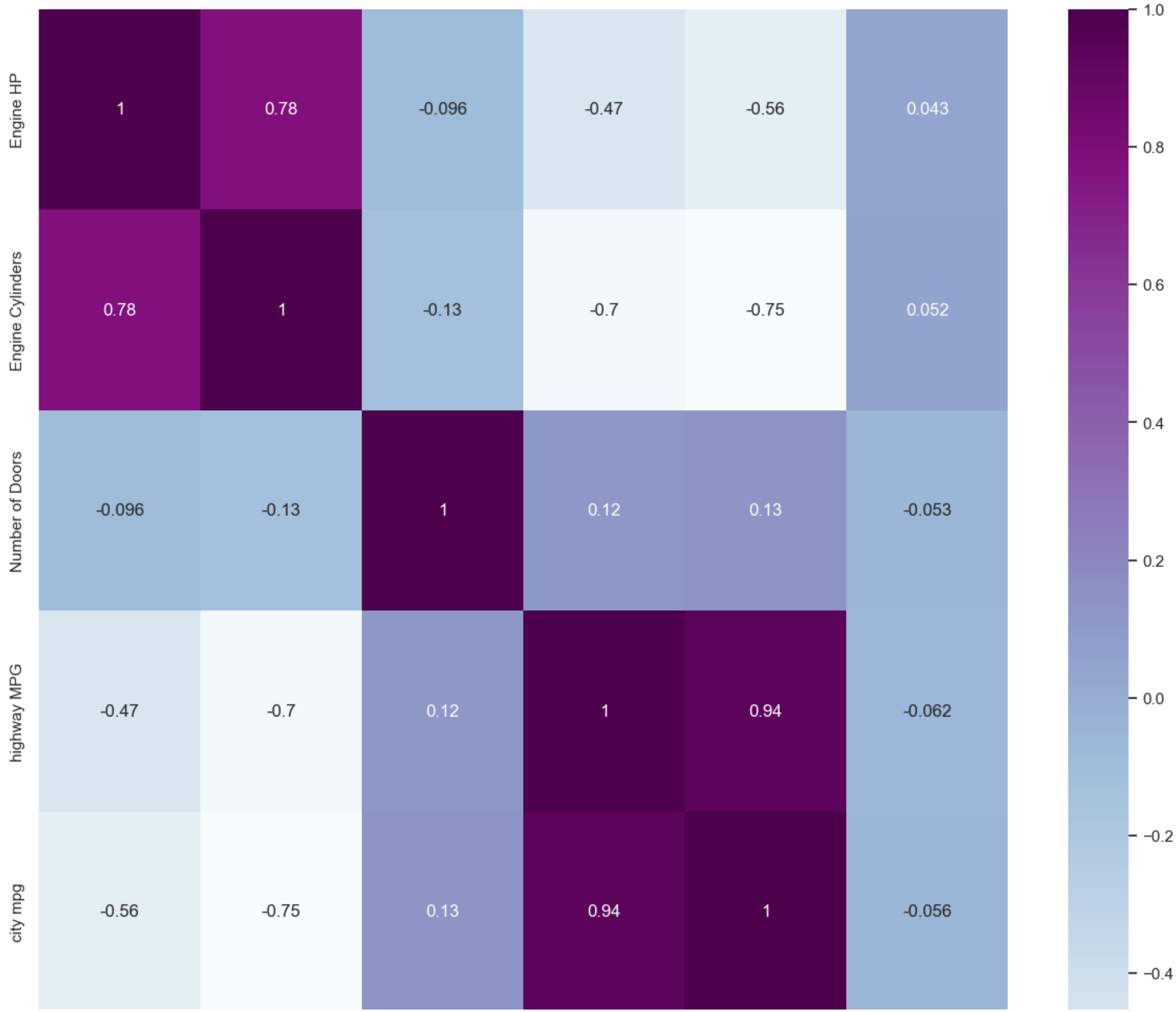
In [43]:

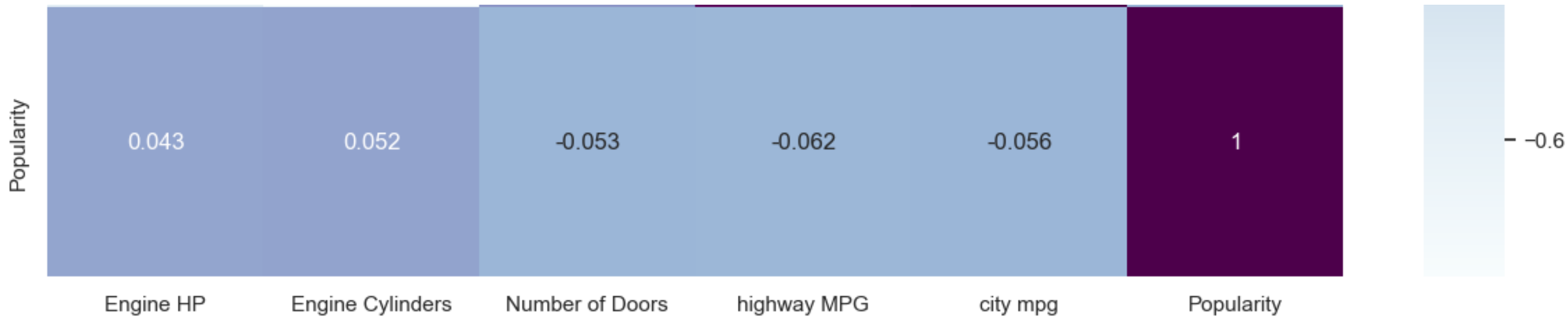
```
plt.figure(figsize = (15, 15))
numeric_columns = ['Engine HP', 'Engine Cylinders', 'Number of Doors', 'highway MPG', 'city mpg', 'Popularity']
heatmap_data = data[numeric_columns].corr()

sns.heatmap(heatmap_data, cmap = 'BuPu', annot = True)
```

Out[43]:

```
<AxesSubplot:>
```

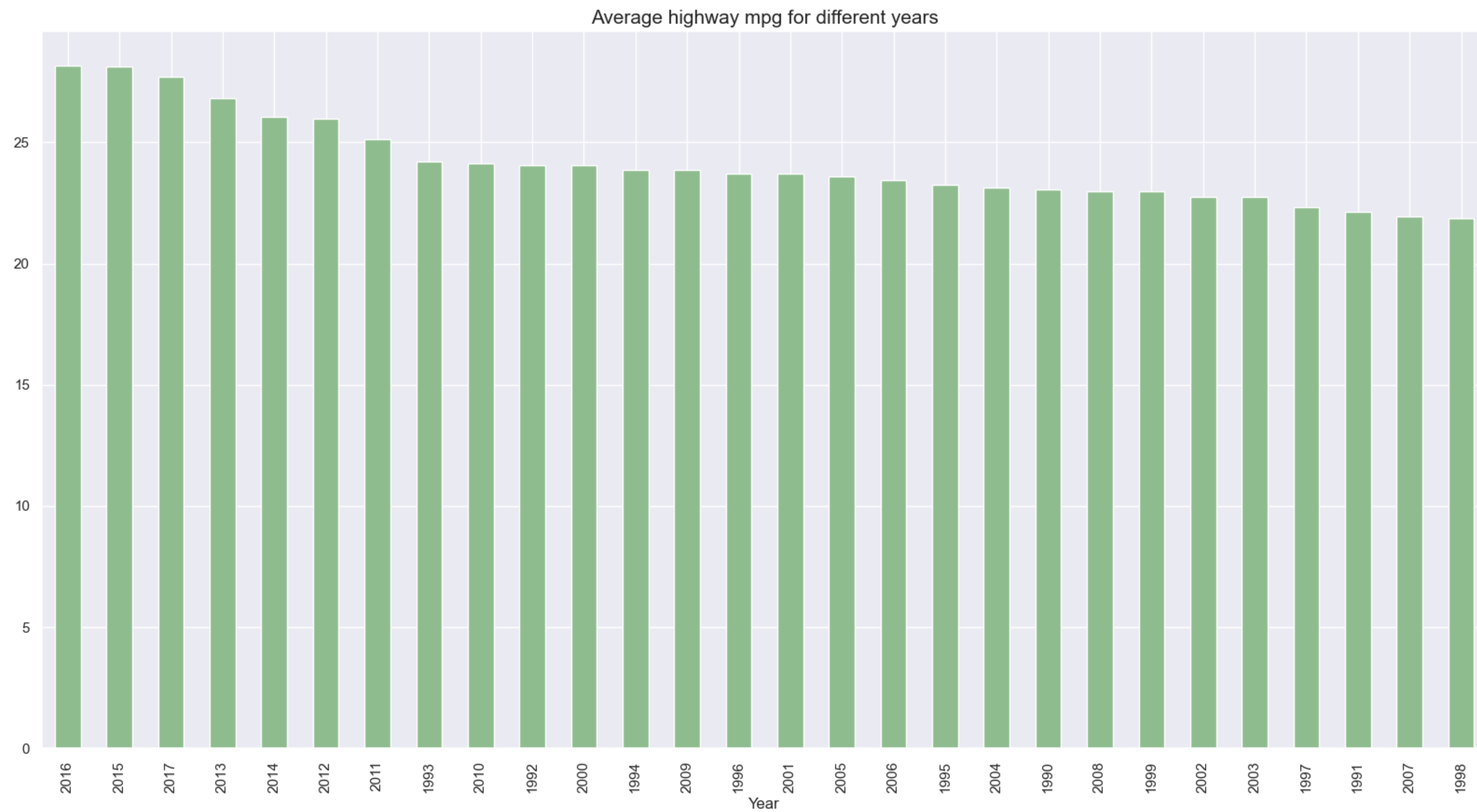

Engine HP
1
0.78
-0.096
-0.47
-0.56
0.043
Engine Cylinders
0.78
1
-0.13
-0.7
-0.75
0.052
Number of Doors
-0.096
-0.13
1
0.12
0.13
-0.053
highway MPG
-0.47
-0.7
0.12
1
0.94
-0.062
city mpg
-0.56
-0.75
0.13
0.94
1
-0.056
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4



2.8 Grouping on the basis of 'Year'

We would now be grouping on the basis of the year and check the 'highway MPG' in descending order so that we can know which year there was a good 'highway MPG' and so on. We see from the plot that in the year 2016, there is a higher value of 'highway MPG' respectively. We cannot easily seperate the values on the basis of year as there are so many years to be taken into consideration respectively. We were able to separate the average prices of the car on the basis of the year as we found that there are some cars of the years below 2000 that were considered to be low. Here, we cannot find such cases as there are cars even in the 90's that had high values of 'highway MPG' respectively.

In [44]:
```
plt.figure(figsize = (20, 10))
data.groupby('Year').mean()['highway MPG'].sort_values(ascending = False).plot(kind = 'bar', color = 'darkseagreen')
plt.title("Average highway mpg for different years", fontsize = 15)
plt.show()
```



Checking the NULL values

Now, it is time to check the null values and see if there are any missing data values. We see that there are a few features that have missing values. We see some features such as 'Engine Fuel Type' and 'Engine HP' that are missing. We have to fill those missing values as our machine learning model cannot deal with missing values though there are some algorithms that can solve the problem.

In [45]:
```
data.isnull().sum()
```

Out[45]:
```
Make                    0
Model                   0
Year                    0
Engine Fuel Type        3
Engine HP              21
Engine Cylinders       20
Transmission Type       0
Driven_Wheels           0
Number of Doors         1
Market Category      3737
Vehicle Size            0
Vehicle Style           0
highway MPG             0
city mpg                0
Popularity              0
MSRP                    0
dtype: int64
```

We would be calculating the median values of 'Number of doors' so that we can fill the missing values with the median value.

In [46]:
```
data['Number of Doors'].median()
```

Out[46]:
```
4.0
```

Here, we would be filling the missing values with the value 4 which is the median of the number of doors.

In [47]:
```
data['Number of Doors'].fillna(4.0, inplace = True)
```

In [48]:
```
data['Number of Doors'].isnull().sum()
```

Out[48]:
```
0
```

In [49]:
```
data
```

Out[49]:

	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors	Market Category	Vehicl Siz
0	BMW	1 Series M	2011	premium unleaded (required)	335.0	6.0	MANUAL	rear wheel drive	2.0	Factory Tuner,Luxury,High-Performance	Compa
1	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,Performance	Compa
2	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,High-Performance	Compa
3	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,Performance	Compa
4	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Luxury	Compa
...	...	...	...	...	...	...	...	...	...	...	.
11909	Acura	ZDX	2012	premium unleaded (required)	300.0	6.0	AUTOMATIC	all wheel drive	4.0	Crossover,Hatchback,Luxury	Midsiz
11910	Acura	ZDX	2012	premium unleaded (required)	300.0	6.0	AUTOMATIC	all wheel drive	4.0	Crossover,Hatchback,Luxury	Midsiz
11911	Acura	ZDX	2012	premium unleaded (required)	300.0	6.0	AUTOMATIC	all wheel drive	4.0	Crossover,Hatchback,Luxury	Midsiz
11912	Acura	ZDX	2013	premium unleaded (recommended)	300.0	6.0	AUTOMATIC	all wheel drive	4.0	Crossover,Hatchback,Luxury	Midsiz
11913	Lincoln	Zephyr	2006	regular unleaded	221.0	6.0	AUTOMATIC	front wheel drive	4.0	Luxury	Midsiz

11705 rows × 16 columns

Creating a new column

We would now try to add a new feature which is used to calculate the difference between the present year and the year of manufacture of the car so that we can take into consideration the depreciation amount which can be done by the machine learning models. Therefore, we create a new column called 'Present Year' and we make it equal to 2023 respectively. We would then subtract the 'Year of manufacture' values with these values of the car so that we get the total number of years the car has been out.

In [50]:
```
data['Present Year'] = 2023
```

We would be printing the head of the dataframe just to check the values that are present in it.

In [51]:
```
data.head()
```

Out[51]:

	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors	Market Category	Vehicle Size	Vehicle Style	higl
0	BMW	1 Series M	2011	premium unleaded (required)	335.0	6.0	MANUAL	rear wheel drive	2.0	Factory Tuner,Luxury,High-Performance	Compact	Coupe	
1	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,Performance	Compact	Convertible	
2	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,High-Performance	Compact	Coupe	
3	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,Performance	Compact	Coupe	
4	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Luxury	Compact	Convertible	

Now it is the time to create a new column called 'Year of Manufacture' respectively. We would be doing the subtraction of the 'Present Year' from the 'Year' which is nothing but the year of manufacture. It would be better to plot the graph and see how the graph looks like in the notebook.

In [52]:
```
data['Years of Manufacture'] = data['Present Year'] - data['Year']
```

Once we have the information, there is no need to have an additional column called 'Present Year' as that value is a constant. We would, therefore, delete the column as it is no longer needed.

In [53]:
```
data.drop(['Present Year'], inplace = True, axis = 1)
```

In [54]:
```
data
```

Out[54]:

	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors	Market Category	Vehicl Siz
0	BMW	1 Series M	2011	premium unleaded (required)	335.0	6.0	MANUAL	rear wheel drive	2.0	Factory Tuner,Luxury,High-Performance	Compac
1	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,Performance	Compac
2	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,High-Performance	Compac
3	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,Performance	Compac
4	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Luxury	Compac
...	...	...	...	...	...	...	...	...	...	...	.
11909	Acura	ZDX	2012	premium unleaded (required)	300.0	6.0	AUTOMATIC	all wheel drive	4.0	Crossover,Hatchback,Luxury	Midsiz
11910	Acura	ZDX	2012	premium unleaded (required)	300.0	6.0	AUTOMATIC	all wheel drive	4.0	Crossover,Hatchback,Luxury	Midsiz
11911	Acura	ZDX	2012	premium unleaded (required)	300.0	6.0	AUTOMATIC	all wheel drive	4.0	Crossover,Hatchback,Luxury	Midsiz
11912	Acura	ZDX	2013	premium unleaded (recommended)	300.0	6.0	AUTOMATIC	all wheel drive	4.0	Crossover,Hatchback,Luxury	Midsiz
11913	Lincoln	Zephyr	2006	regular unleaded	221.0	6.0	AUTOMATIC	front wheel drive	4.0	Luxury	Midsiz

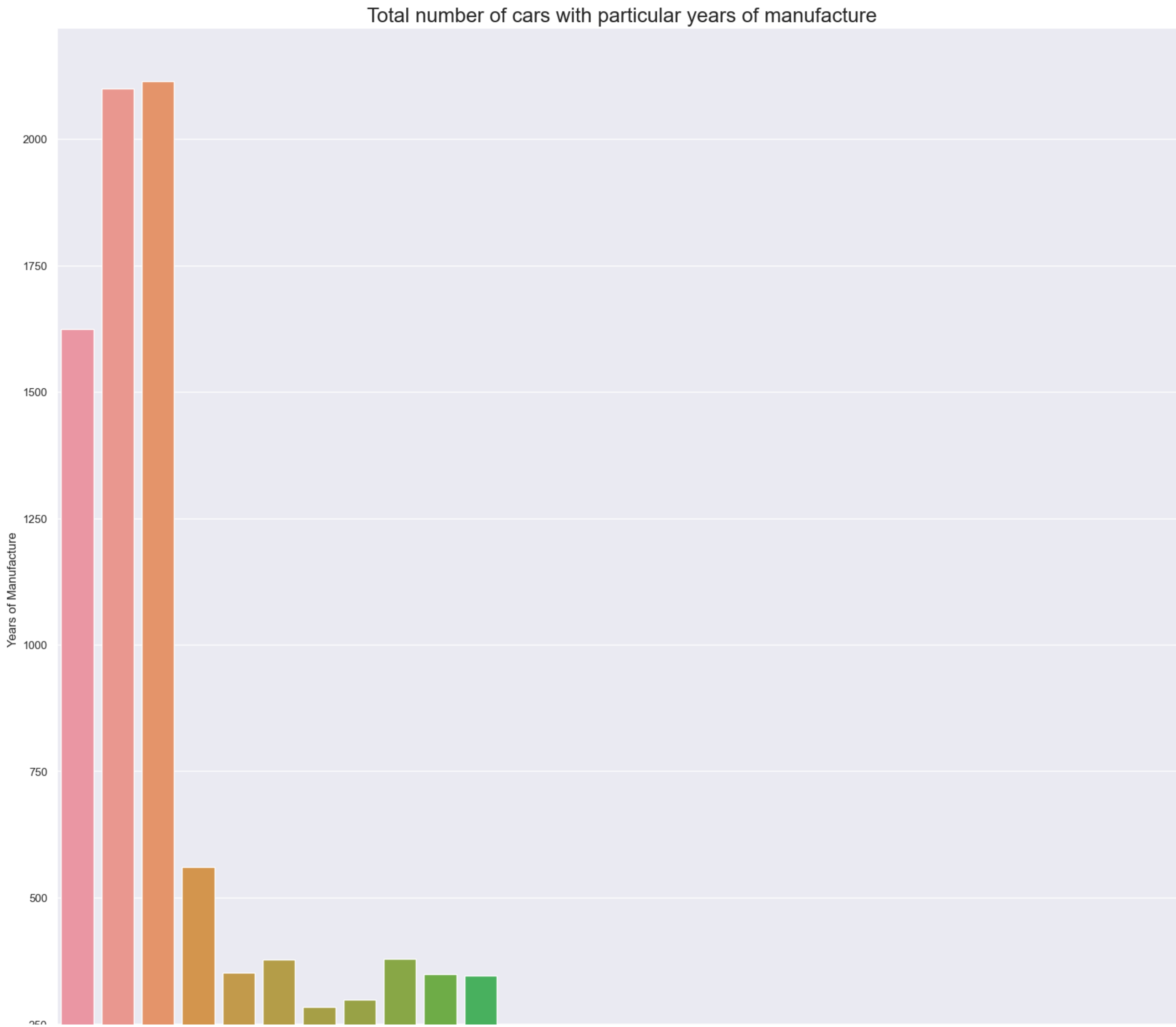
11705 rows × 17 columns

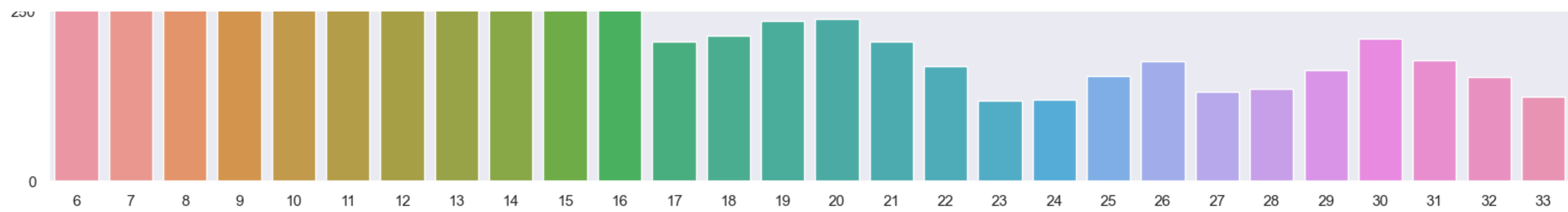
2.9 Plotting the barplot of 'Years of Manufacture'

We see that most of the values are about 8 years old. Therefore, we are working with young cars as there are some other cars in our data that are about 33 years old. These cars are very few in number. It is good to work with the most recent data points as the future would also be more relying and would be following the trend of the most recent data points into consideration.

In [55]:

```
sns.barplot(y = data['Years of Manufacture'].value_counts(), x = data['Years of Manufacture'].value_counts().index)
plt.title("Total number of cars with particular years of manufacture", fontsize = 20)
plt.show()
```


Total number of cars with particular years of manufacture
Years of Manufacture
2000
1750
1500
1250
1000
750
500
250



Unique values in 'Engine Fuel Type'

We see that there are a few values that are present in 'Engine Fuel Types' and we would be taking those values into consideration respectively.

In [56]:
```
data['Engine Fuel Type'].unique()
```

Out[56]:
```
array(['premium unleaded (required)', 'regular unleaded',
       'premium unleaded (recommended)', 'flex-fuel (unleaded/E85)',
       'diesel', 'flex-fuel (premium unleaded recommended/E85)',
       'natural gas', 'flex-fuel (premium unleaded required/E85)',
       'flex-fuel (unleaded/natural gas)', nan], dtype=object)
```

In [57]:
```
type("data['Engine Fuel Type'].mode()")
```

Out[57]:
```
str
```

In [58]:
```
data['Engine Fuel Type'].fillna("data['Engine Fuel Type'].mode()", inplace = True)
```

In [59]:
```
data['Engine Fuel Type'].isnull().sum()
```

Out[59]:
```
0
```

We would now be calculating the mean value of 'Engine HP' and understand it better. As could be seen from the graph, the 'Engine HP' is about 250 as can also be seen below.

In [60]:
```
data['Engine HP'].mean()
```

Out[60]:
```
250.75316672372475
```

We would now be checking the median values of 'Engine HP' and understand them better. We see that the value is about '230' respectively which is nothing but the median value.

In [61]:
```
data['Engine HP'].median()
```

Out[61]:
```
230.0
```

We are now going to fill the missing values with the median value so that it is more appropriate and accurate. And we have to make sure that inplace = True which means that the values that are replaced are permanent rather than getting temporary solutions in a variable

In [62]:
```
data['Engine HP'].fillna(data['Engine HP'].median(), inplace = True)
```

In [63]:
```
data['Engine HP'].isnull().sum()
```

Out[63]:
```
0
```

In [64]:
```
data.isnull().sum()
```

Out[64]:
```
Make                       0
Model                      0
Year                       0
Engine Fuel Type           0
Engine HP                  0
Engine Cylinders          20
Transmission Type          0
Driven_Wheels              0
Number of Doors            0
Market Category         3737
Vehicle Size               0
Vehicle Style              0
highway MPG                0
city mpg                   0
Popularity                 0
MSRP                       0
Years of Manufacture       0
dtype: int64
```

It is good to know the values that are present in 'Engine Cylinders' and see if there is any replacement. We see that there is an 'nan' value present in our data. We have to remove that point and replace it with some other value as missing values in machine learning could be costly and lead to some errors. Moreover, some machine learning algorithms cannot perform well too if there are any missing values.

In [65]:
```
data['Engine Cylinders'].unique()
```

Out[65]:
```
array([ 6.,  4.,  5.,  8., 12., 10.,  3., nan, 16.])
```

In [66]:
```
data['Engine Cylinders'].fillna(4, inplace = True)
```

We would once again check the missing values and see if there are values present in our features. We see that there is one feature with missing values namely 'Market Category' respectively. We would be removing that feature as there are many missing values and it seems as though we cannot make use of this feature anyways as it contains complex texts. Note that we might have used some natural language processing techniques (NLP) to solve the problem. However, it is not feasible here as the text is too complex and cannot find much of a value in it.

In [67]:
```
data.isnull().sum()
```

Out[67]:
```
Make                      0
Model                     0
Year                      0
Engine Fuel Type          0
Engine HP                 0
Engine Cylinders          0
Transmission Type         0
Driven_Wheels             0
Number of Doors           0
Market Category        3737
Vehicle Size              0
Vehicle Style             0
highway MPG               0
city mpg                  0
Popularity                0
MSRP                      0
Years of Manufacture      0
dtype: int64
```

Below we are dropping the 'Market Category' feature and making it inplace = True which shows that the feature is removed.

In [68]:
```
data.drop(['Market Category'], inplace = True, axis = 1)
```

We once again check the missing values and see if there are any missing values in our data. We see that there are no missing values in our features. Therefore, now is the time to convert all these features into a mathematical format so that we would be able to perform the machine learning operations.

In [69]:
```
data.isnull().sum()
```

Out[69]:
```
Make                   0
Model                  0
Year                   0
Engine Fuel Type       0
Engine HP              0
Engine Cylinders       0
Transmission Type      0
Driven_Wheels          0
Number of Doors        0
Vehicle Size           0
Vehicle Style          0
highway MPG            0
city mpg               0
Popularity             0
MSRP                   0
Years of Manufacture   0
dtype: int64
```

We would see the information about the data and consider the type of feature that we are going to be dealing in machine learning respectively. We find that there are a few object features which must be converted to a mathematical form for the machine learning algorithm to read and understand them.

In [70]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 11705 entries, 0 to 11913
Data columns (total 16 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   Make                  11705 non-null  object 
 1   Model                 11705 non-null  object 
 2   Year                  11705 non-null  int64  
 3   Engine Fuel Type      11705 non-null  object 
 4   Engine HP             11705 non-null  float64
 5   Engine Cylinders      11705 non-null  float64
 6   Transmission Type     11705 non-null  object 
 7   Driven_Wheels         11705 non-null  object 
 8   Number of Doors       11705 non-null  float64
 9   Vehicle Size          11705 non-null  object 
 10  Vehicle Style         11705 non-null  object 
 11  highway MPG           11705 non-null  int64  
 12  city mpg              11705 non-null  int64  
 13  Popularity            11705 non-null  int64  
 14  MSRP                  11705 non-null  int64  
 15  Years of Manufacture  11705 non-null  int64  
dtypes: float64(3), int64(6), object(7)
memory usage: 1.5+ MB
```

We would find the unique values of 'Vehicle Size' and see the values that are associated with them. We see just 3 categorical features such as 'Compact', 'Midsize' and 'Large' respectively.

In [71]:
```
data['Vehicle Size'].unique()
```

Out[71]:
```
array(['Compact', 'Midsize', 'Large'], dtype=object)
```

We would also see the vehicle style and the categories that are associated with them. We see a lot of categories and we would have to be working with them and understand and convert them into the form of integers before working with them.

In [72]:
```
data['Vehicle Style'].unique()
```

Out[72]:
```
array(['Coupe', 'Convertible', 'Sedan', 'Wagon', '4dr Hatchback',
       '2dr Hatchback', '4dr SUV', 'Passenger Minivan', 'Cargo Minivan',
       'Crew Cab Pickup', 'Regular Cab Pickup', 'Extended Cab Pickup',
       '2dr SUV', 'Cargo Van', 'Convertible SUV', 'Passenger Van'],
      dtype=object)
```

3. Manipulation of Data

Now, it is time to manipulate the data and convert it in the forms where we could give it for the machine learning models for predictions. We use various libraries in python such as shuffle that are used to choose various data values that would later be given to the machine learning models. There is a requirement to also encode the text information that is present so that those values are converted to mathematical vectors that would ensure that they would be understood by the algorithms respectively.

3.1 Shuffling the data

Most of the machine learning projects that I've seen do not make use of shuffle feature in python. It is very important to shuffle the data randomly so that we get outputs differently and we would be dealing with data without any particular order or a particular timeframe.

In [73]:
```
from sklearn.utils import shuffle
```

In [74]:
```
shuffled_data = shuffle(data, random_state = 100)
X = shuffled_data.drop(['MSRP'], axis = 1)
y = shuffled_data['MSRP']
```

In [75]:
```
X
```

Out[75]:

	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors	Vehicle Size	Vehicle Style	highw M
10070	Pontiac	Sunbird	1994	regular unleaded	110.0	4.0	MANUAL	front wheel drive	4.0	Compact	Sedan	
4342	Ford	Expedition	2017	regular unleaded	365.0	6.0	AUTOMATIC	four wheel drive	4.0	Large	4dr SUV	
8749	Chevrolet	S-10	2002	regular unleaded	190.0	6.0	MANUAL	four wheel drive	3.0	Compact	Extended Cab Pickup	
6681	Chevrolet	Malibu	2016	premium unleaded (recommended)	250.0	4.0	AUTOMATIC	front wheel drive	4.0	Midsize	Sedan	
8064	Dodge	RAM 150	1992	regular unleaded	230.0	8.0	MANUAL	rear wheel drive	2.0	Large	Extended Cab Pickup	
...	...	...	...	...	...	...	...	...	...	...	...	
350	Nissan	370Z	2017	premium unleaded (required)	332.0	6.0	AUTOMATIC	rear wheel drive	2.0	Compact	Coupe	
79	Chrysler	200	2017	flex-fuel (unleaded/E85)	295.0	6.0	AUTOMATIC	all wheel drive	4.0	Midsize	Sedan	
8233	Dodge	Ramcharger	1992	regular unleaded	230.0	8.0	MANUAL	four wheel drive	2.0	Midsize	2dr SUV	
7084	Ford	Mustang	2016	premium unleaded (recommended)	435.0	8.0	MANUAL	rear wheel drive	2.0	Midsize	Convertible	
5712	Chevrolet	HHR	2009	premium unleaded (recommended)	260.0	4.0	MANUAL	front wheel drive	4.0	Compact	Wagon	

11705 rows × 15 columns

In [76]:

```
y
```

Out[76]:
```
10070     2000
4342     62860
8749     19757
6681     30920
8064      2000
          ...
350      39270
79       31785
8233      2000
7084     41895
5712     24815
Name: MSRP, Length: 11705, dtype: int64
```

3.2 Dividing the data into training and testing set

We would be dividing the data into training data and testing data. Since we have a lot of data points, it would be better to randomly divide the data so that the test set just contains 20 percent of the values. Since the total number of data points that we have taken into consideration are about 10000, it would be wise to divide the training and testing set in the ratio 80:20 percent. In general, we would be diving the training and testing set so that the value that is present in the training set is about 30 percent of the total data.

In [77]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 100)
```

We would be printing the format of the data and see how the values are divided. We see that the total number of rows on the training set are about 9364 respectively. We also see that the total number of rows on the test set are about 2341 respectively. These values can be seen here.

In [78]:
```
print("The size of the input train data is: {}".format(X_train.shape))
print("The size of the output train data is: {}".format(y_train.shape))
print("The size of the input test data is: {}".format(X_test.shape))
print("The size of the output test data is: {}".format(y_test.shape))
```

```
The size of the input train data is: (9364, 15)
The size of the output train data is: (9364,)
The size of the input test data is: (2341, 15)
The size of the output test data is: (2341,)
```

3.3 Encoding the data

When we are doing any machine learning applications, it is important to encode the data so that we would be able to convert the data in the form of categorical features so that we would be working on the data that is mathematical rather than categorical. Therefore, we would be converting the categorical feature into numerical features so that we are going to be using the mathematical vectors for our machine learning applications.

There are different encoding techniques that we would be taking into consideration and we are making sure that we get the best output values associated with each of them. You can check out the link below to see the different encoding techniques to convert the cateogrical values into numerical features respectively.

https://www.analyticsvidhya.com/blog/2020/08/types-of-categorical-data-encoding/

In [79]:
```
encoder = TargetEncoder(cols = 'Year')
```

In [80]:
```
X_train.head()
```

Out[80]:

	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors	Vehicle Size	Vehicle Style	highway MPG	c m
1354	Oldsmobile	Alero	2003	regular unleaded	140.0	4.0	AUTOMATIC	front wheel drive	4.0	Midsize	Sedan	30	
896	Saab	900	1997	regular unleaded	185.0	4.0	MANUAL	front wheel drive	2.0	Compact	2dr Hatchback	25	
2635	Chevrolet	C/K 1500 Series	1997	regular unleaded	200.0	6.0	MANUAL	four wheel drive	2.0	Large	Regular Cab Pickup	18	
11165	Aston Martin	V8 Vantage	2015	premium unleaded (required)	430.0	8.0	MANUAL	rear wheel drive	2.0	Compact	Convertible	19	
2554	Honda	Civic	2015	regular unleaded	143.0	4.0	AUTOMATIC	front wheel drive	4.0	Compact	Sedan	39	

We would be doing the target encoding here where we would replace the values with the average values of the 'MSRP' whenever we find a value associated with it. This would make it easier for the machine learning model as we are already giving the output values to it so that there is no need to encode further.

In [81]:
```
from category_encoders import TargetEncoder, OneHotEncoder
```

In [82]:
```
encoder.fit(X_train['Year'], y_train.to_frame()['MSRP'])
```

Out[82]:
```
TargetEncoder(cols=['Year'])
```

We would be transforming the value so that we are now converting the column so that we get the most desired output respectively.

In [83]:
```
X_train['Year'] = encoder.transform(X_train['Year'])
```

We should also make sure that the values that we have taken into consideration and transforming should be with respect to the training set. We should not replace those values with the test output as it would lead to data leakage respectively.

In [84]:
```
X_test['Year'] = encoder.transform(X_test['Year'])
```

In [85]:
```
X_train
```

Out[85]:

	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors	Vehicle Size	Vehicle Style	high
1354	Oldsmobile	Alero	36784.190660	regular unleaded	140.0	4.0	AUTOMATIC	front wheel drive	4.0	Midsize	Sedan	
896	Saab	900	2558.613101	regular unleaded	185.0	4.0	MANUAL	front wheel drive	2.0	Compact	2dr Hatchback	
2635	Chevrolet	C/K 1500 Series	2558.613101	regular unleaded	200.0	6.0	MANUAL	four wheel drive	2.0	Large	Regular Cab Pickup	
11165	Aston Martin	V8 Vantage	46953.929157	premium unleaded (required)	430.0	8.0	MANUAL	rear wheel drive	2.0	Compact	Convertible	
2554	Honda	Civic	46953.929157	regular unleaded	143.0	4.0	AUTOMATIC	front wheel drive	4.0	Compact	Sedan	
...	...	...	...	...	...	...	...	...	...	...	...	
1013	Porsche	968	2669.242559	regular unleaded	236.0	4.0	MANUAL	rear wheel drive	2.0	Compact	Convertible	
9906	Kia	Spectra	51694.325658	regular unleaded	138.0	4.0	MANUAL	front wheel drive	4.0	Compact	Sedan	
2034	Ford	Bronco	2669.242559	regular unleaded	205.0	8.0	MANUAL	four wheel drive	2.0	Midsize	2dr SUV	
1456	Bentley	Arnage	56411.258064	premium unleaded (required)	500.0	8.0	AUTOMATIC	rear wheel drive	4.0	Large	Sedan	
2564	Honda	Civic	46953.929157	premium unleaded (required)	205.0	4.0	MANUAL	front wheel drive	2.0	Compact	Coupe	

9364 rows × 15 columns

We would be doing the same set of steps for other models and we would be taking those values into consideration from our data set.

In [86]:
```
encoder = TargetEncoder(cols = 'Model')
encoder.fit(X_train['Model'], y_train.to_frame()['MSRP'])
X_train['Model'] = encoder.transform(X_train['Model'])
X_test['Model'] = encoder.transform(X_test['Model'])
```

Same operation is being performed here as well as can be seen below.

In [87]:
```
encoder = TargetEncoder(cols = 'Make')
encoder.fit(X_train['Make'], y_train.to_frame()['MSRP'])
X_train['Make'] = encoder.transform(X_train['Make'])
X_test['Make'] = encoder.transform(X_test['Make'])
```

In [88]:
```
X_train.head()
```

Out[88]:

	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors	Vehicle Size	Vehi Sty
1354	10812.757938	30176.543012	36784.190660	regular unleaded	140.0	4.0	AUTOMATIC	front wheel drive	4.0	Midsize	Sed
896	28423.023983	25245.937696	2558.613101	regular unleaded	185.0	4.0	MANUAL	front wheel drive	2.0	Compact	2 Hatchba
2635	28230.392090	5061.892819	2558.613101	regular unleaded	200.0	6.0	MANUAL	four wheel drive	2.0	Large	Regu Cab Pick
11165	196884.138144	97860.899828	46953.929157	premium unleaded (required)	430.0	8.0	MANUAL	rear wheel drive	2.0	Compact	Converti
2554	26660.798742	22687.787683	46953.929157	regular unleaded	143.0	4.0	AUTOMATIC	front wheel drive	4.0	Compact	Sed

In [89]:
```
X_train["Engine Fuel Type"].unique()
```

Out[89]:
```
array(['regular unleaded', 'premium unleaded (required)', 'diesel',
       'premium unleaded (recommended)', 'flex-fuel (unleaded/E85)',
       'flex-fuel (premium unleaded required/E85)',
       'flex-fuel (unleaded/natural gas)',
       'flex-fuel (premium unleaded recommended/E85)',
       "data['Engine Fuel Type'].mode()", 'natural gas'], dtype=object)
```

3.4 One Hot Encoding

Now we would be making use of the one hot encoding. One hot encoding is a technique where each category in a feature is converted into a feature and set to 1 once the particular value is present in the data.

In [90]:
```
encoder = OneHotEncoder()
encoder.fit(X_train[['Engine Fuel Type', 'Transmission Type', 'Driven_Wheels', 'Vehicle Size', 'Vehicle Style']])
one_hot_encoded_output_train = encoder.transform(X_train[['Engine Fuel Type', 'Transmission Type', 'Driven_Wheels',
one_hot_encoded_output_test = encoder.transform(X_test[['Engine Fuel Type', 'Transmission Type', 'Driven_Wheels', 'Ve
```

We would concatenate the features with the X_train and X_test and remove the actual categorical features as they should not be given to the machine learning algorithms respectively.

In [91]:
```
X_train = pd.concat([X_train, one_hot_encoded_output_train], axis = 1)
X_test = pd.concat([X_test, one_hot_encoded_output_test], axis = 1)
```

In [92]:
```
X_train.drop(['Engine Fuel Type', 'Transmission Type', 'Driven_Wheels', 'Vehicle Size', 'Vehicle Style'], axis = 1, i
X_test.drop(['Engine Fuel Type', 'Transmission Type', 'Driven_Wheels', 'Vehicle Size', 'Vehicle Style'], axis = 1, in
```

We would check the info of the data and see the values that are present in the data. We see that there are only float and int values rather than objects. We see that we can give this to the machine learning algorithm for implementation.

In [93]:
```
X_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 9364 entries, 1354 to 2564
Data columns (total 47 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   Make                  9364 non-null   float64
 1   Model                 9364 non-null   float64
 2   Year                  9364 non-null   float64
 3   Engine HP             9364 non-null   float64
 4   Engine Cylinders      9364 non-null   float64
 5   Number of Doors       9364 non-null   float64
 6   highway MPG           9364 non-null   int64  
 7   city mpg              9364 non-null   int64  
 8   Popularity            9364 non-null   int64  
 9   Years of Manufacture  9364 non-null   int64  
 10  Engine Fuel Type_1    9364 non-null   int64  
 11  Engine Fuel Type_2    9364 non-null   int64  
 12  Engine Fuel Type_3    9364 non-null   int64  
 13  Engine Fuel Type_4    9364 non-null   int64  
 14  Engine Fuel Type_5    9364 non-null   int64  
 15  Engine Fuel Type_6    9364 non-null   int64  
 16  Engine Fuel Type_7    9364 non-null   int64  
 17  Engine Fuel Type_8    9364 non-null   int64  
 18  Engine Fuel Type_9    9364 non-null   int64  
 19  Engine Fuel Type_10   9364 non-null   int64  
 20  Transmission Type_1   9364 non-null   int64  
 21  Transmission Type_2   9364 non-null   int64  
 22  Transmission Type_3   9364 non-null   int64  
 23  Transmission Type_4   9364 non-null   int64  
 24  Driven_Wheels_1       9364 non-null   int64  
 25  Driven_Wheels_2       9364 non-null   int64  
 26  Driven_Wheels_3       9364 non-null   int64  
 27  Driven_Wheels_4       9364 non-null   int64  
 28  Vehicle Size_1        9364 non-null   int64  
 29  Vehicle Size_2        9364 non-null   int64  
 30  Vehicle Size_3        9364 non-null   int64  
 31  Vehicle Style_1       9364 non-null   int64  
 32  Vehicle Style_2       9364 non-null   int64  
 33  Vehicle Style_3       9364 non-null   int64  
 34  Vehicle Style_4       9364 non-null   int64  
 35  Vehicle Style_5       9364 non-null   int64  
 36  Vehicle Style_6       9364 non-null   int64  
 37  Vehicle Style_7       9364 non-null   int64  
 38  Vehicle Style_8       9364 non-null   int64  
```

```
 39  Vehicle Style_9       9364 non-null   int64
 40  Vehicle Style_10      9364 non-null   int64
 41  Vehicle Style_11      9364 non-null   int64
 42  Vehicle Style_12      9364 non-null   int64
 43  Vehicle Style_13      9364 non-null   int64
 44  Vehicle Style_14      9364 non-null   int64
 45  Vehicle Style_15      9364 non-null   int64
 46  Vehicle Style_16      9364 non-null   int64
dtypes: float64(6), int64(41)
memory usage: 3.4 MB
```

3.5 Standardization and Normalization of data

We would be considering the values of our data and perform some operations such as standardization and normalization before giving the data to the machine learning algorithms. We would be transforming the features that are present in the data and convert the values using the minmaxscaler respectively.

In [94]:
```
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_new = scaler.transform(X_train)
X_test_new = scaler.transform(X_test)
```

In [95]:
```
X_train_new.shape
```

Out[95]:
```
(9364, 47)
```

In [96]:
```
y_train.shape
```

Out[96]:
```
(9364,)
```

We would create an empty list and we would be appending the values later so that we can analyze different machine learning algorithms for deployment.

In [97]:
```
error_mean_square = []
error_mean_absolute = []
```

4. Machine Learning Analysis

In this section, we are going to be performing the machine learning analysis where we use different machine learning models and see how well they perform on the test set. Let us consider their performances and plot them using various plots such as regplots and barplots respectively. With this analysis, we can conclude the performance of different machine learning models and select the best machine learning model for our problem. Therefore, let us dive into this section and see the overall performance of the models for car prices prediction.

4.1 Linear Regression

We would now be working with linear regression model and understand the data fully. We see that one of the best ways for predicting the regression values or the continuous output is to use linear regression as it is straightforward. We have to first give the training data including the training output. We have to first fit the model with that data and understand the parameters. After we fit the model, we have to train the model using the machine learning predictions to get the output. We have to later compare the values from the actual values with the predicted values to get the output. We have to be using various machine learning metrics what are used for evaluation.

In the same way, we would be working with a few machine learning models and get their outputs and compare the values using the metrics to see which algorithm performs the best.

In [98]:
```
model = LinearRegression()
model.fit(X_train_new, y_train)
```

Out[98]:
```
LinearRegression()
```

Here, we would be using predict to predict the test set values and store those values in y_predict which would later be used for comparison.

In [99]:
```
y_predict = model.predict(X_test_new)
```

We would be storing the results in error_mean_square and error_mean_absolute as they are lists. We would later be plotting the outputs and see how well the machine learning models did in the test set.

In [100…
```
error_mean_square.append(int(mean_squared_error(y_predict, y_test)))
error_mean_absolute.append(int(mean_absolute_error(y_predict, y_test)))
```

We see that there is a value appended in the list below.

In [101…
```
error_mean_absolute
```

Out[101]:
```
[13253]
```

One of the interesting things that we would be doing is to create a dataframe containing the predicted valeus and the actual values and draw a plot so that we can see how the output is actually different from the predictions.

In [102…
```
y_predict = pd.DataFrame(y_predict, columns = ['Predicted Output'])
```

We would be looking at the head of the dataframe and understand the data better

In [103…
```
y_predict.head()
```

Out[103]:

	Predicted Output
0	11932.269053
1	18180.110310
2	71540.949096
3	29414.645783
4	307259.662158

We would be also testing the set values and see how they actually are respectively

In [104…
```
y_test.to_frame().head()
```

Out[104]:

	MSRP
8780	24660
674	2000
6569	49770
11368	20875
3548	284976

Here, we would be concatenating the y_predict values an the y_test values and see how well the machine learning models perform.

In [105…

```
results = pd.concat([y_predict, y_test.to_frame().reset_index(drop = True)], axis = 1, ignore_index = False)
```

Below we can see the concantenated output and see the output

In [106…

```
results.head()
```

Out[106]:

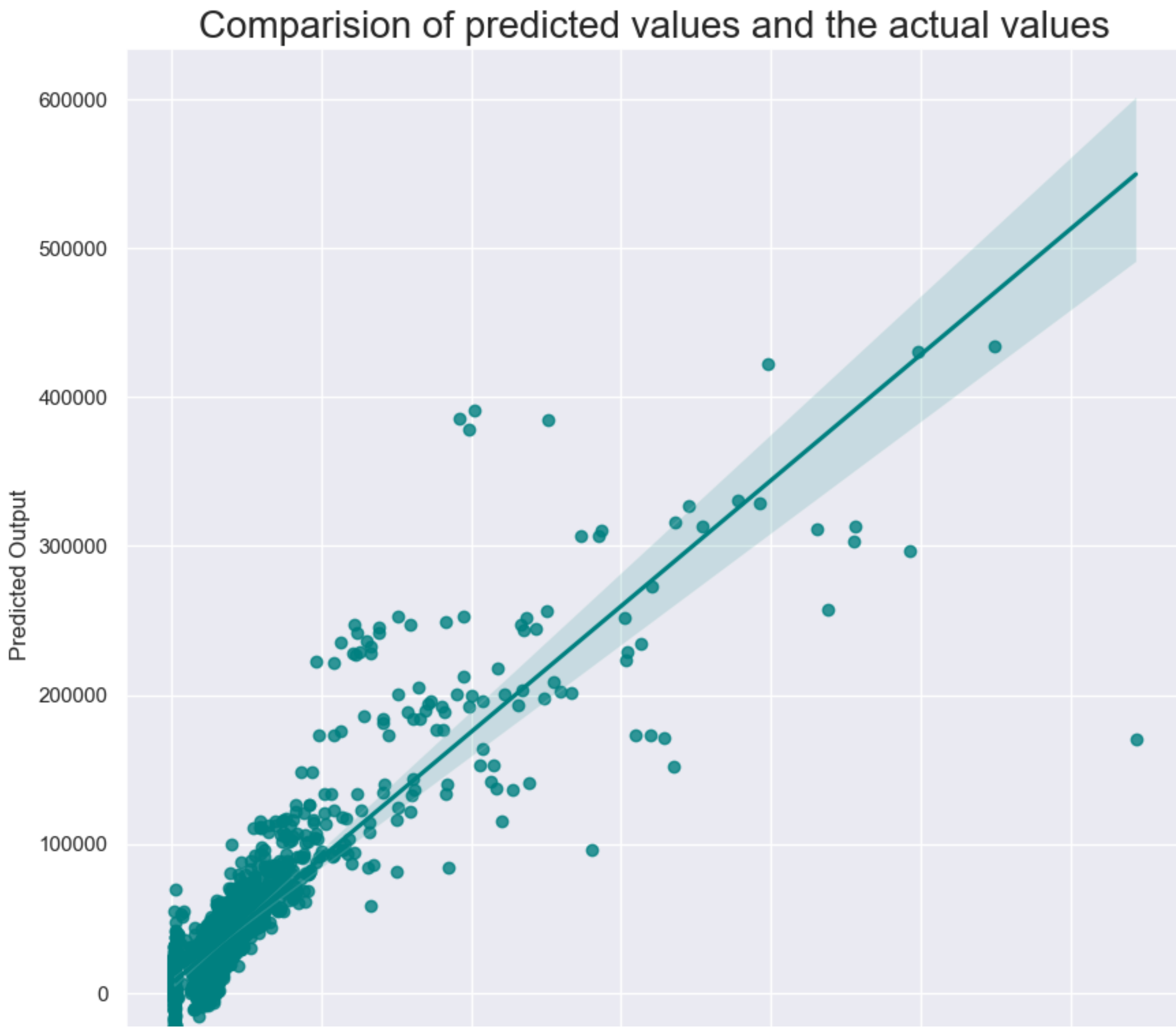
	Predicted Output	MSRP
0	11932.269053	24660
1	18180.110310	2000
2	71540.949096	49770
3	29414.645783	20875
4	307259.662158	284976

4.1.1 Regplot for Linear Regression Output

We would be using the seaborn's regplot to better understand how the data is spread. We see how the values are spread out and get a good understanding. We can understand from the plot that the predictions were very close to the actual values that we have considered. Therefore, linear regression did a good job in giving the regression values and can be used for predictions in the future. However, it is also better to test other machine learning models and see how well they do so that we can finally decide the best model that could be used for deployment.

In [107…

```
plt.figure(figsize = (10, 10))
sns.regplot(data = results, y = 'Predicted Output', x = 'MSRP', color = 'teal', marker = 'o')
plt.title("Comparision of predicted values and the actual values", fontsize = 20)
plt.show()
```


Comparision of predicted values and the actual values
Predicted Output
600000
500000
400000
300000
200000
100000
0

4.3 K - Neighbors Regressor

We would be using the knn regressor and understand the output. We would be drawing a regplot to get an understanding of how the data is spread out.

In [108…

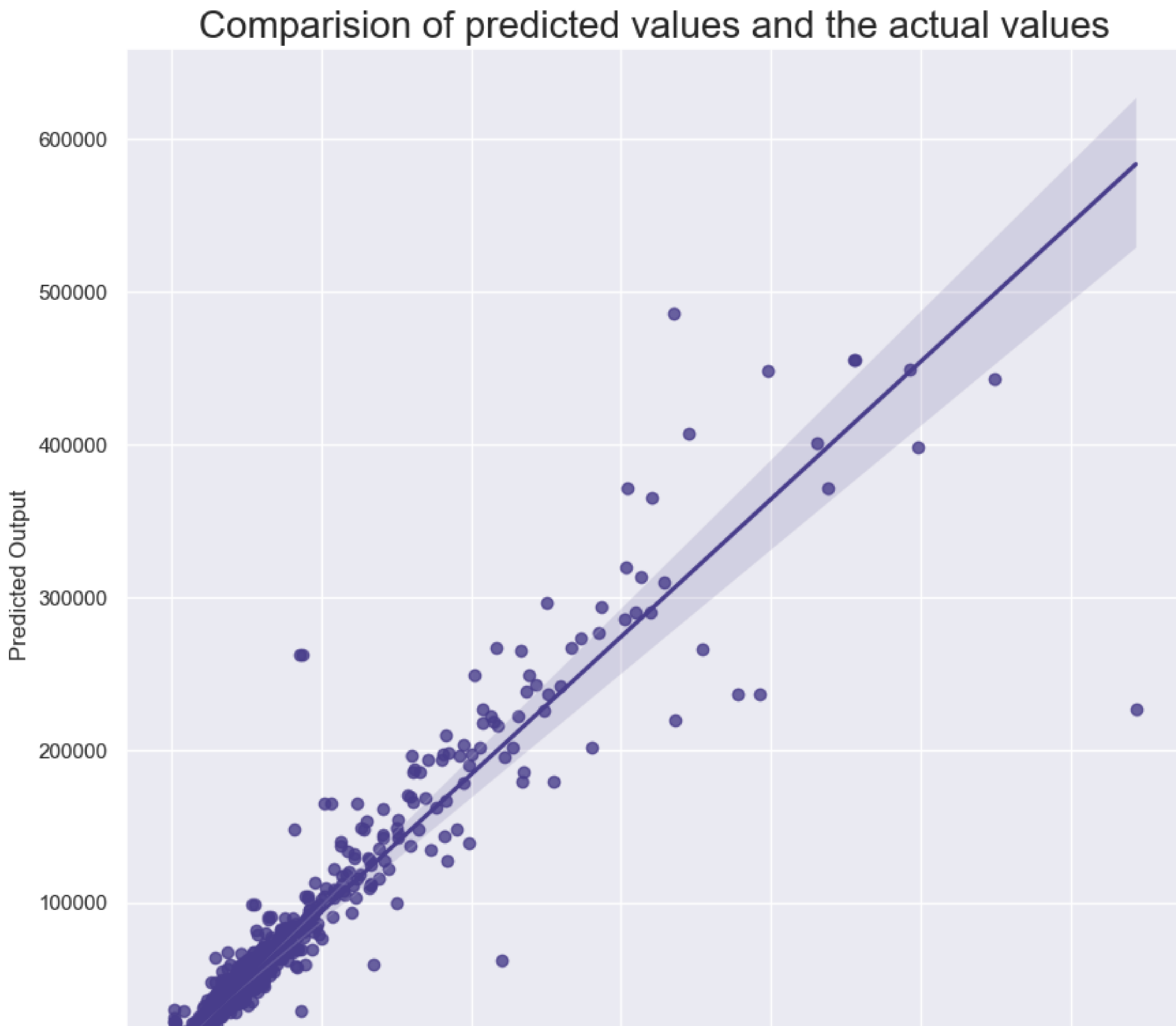
```
model = KNeighborsRegressor(n_neighbors = 2)
model.fit(X_train_new, y_train)
y_predict = model.predict(X_test_new)
y_predict = pd.DataFrame(y_predict, columns = ['Predicted Output'])
results = pd.concat([y_predict, y_test.to_frame().reset_index(drop = True)], axis = 1, ignore_index = False)
```

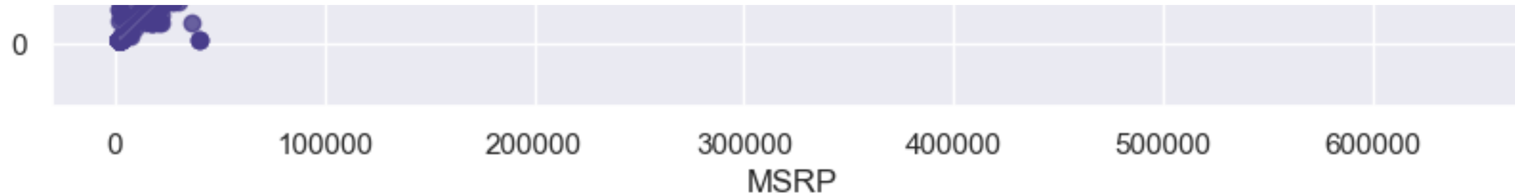
4.3.1 Regplot for K - Neighbors Regressor

We would be making use of the regplot again and plotting the predicted values and the actual predicted output values. We see that K - Neighbors Regressor did well in the testing set as compared to Support Vector Regressor respectively.We see that most of the predictions are close to the actual outputs in the plot below. There are just a few points that were not completely accurate and the margin error is high. But ther is not a lot of error for the remaining predictions as can be seen below.

In [109…

```
plt.figure(figsize = (10, 10))
sns.regplot(data = results, y = 'Predicted Output', x = 'MSRP', color = 'darkslateblue', marker = 'o')
plt.title("Comparision of predicted values and the actual values", fontsize = 20)
plt.show()
```


Comparision of predicted values and the actual values
600000
500000
400000
300000
200000
100000
Predicted Output



We would be using the metrics that we have seen and storing those values in a list. We would append the elements and form a list.

In [110…

```
error_mean_square.append(int(mean_squared_error(y_predict, y_test)))
error_mean_absolute.append(int(mean_absolute_error(y_predict, y_test)))
```

4.5 Decision Tree Regressor

We would making use of decision tree regressor and make the split to be random. We would be fitting the training data to it and make the predictions later for the test data to get an understanding of how the algorithm did in the test set.

In [111…

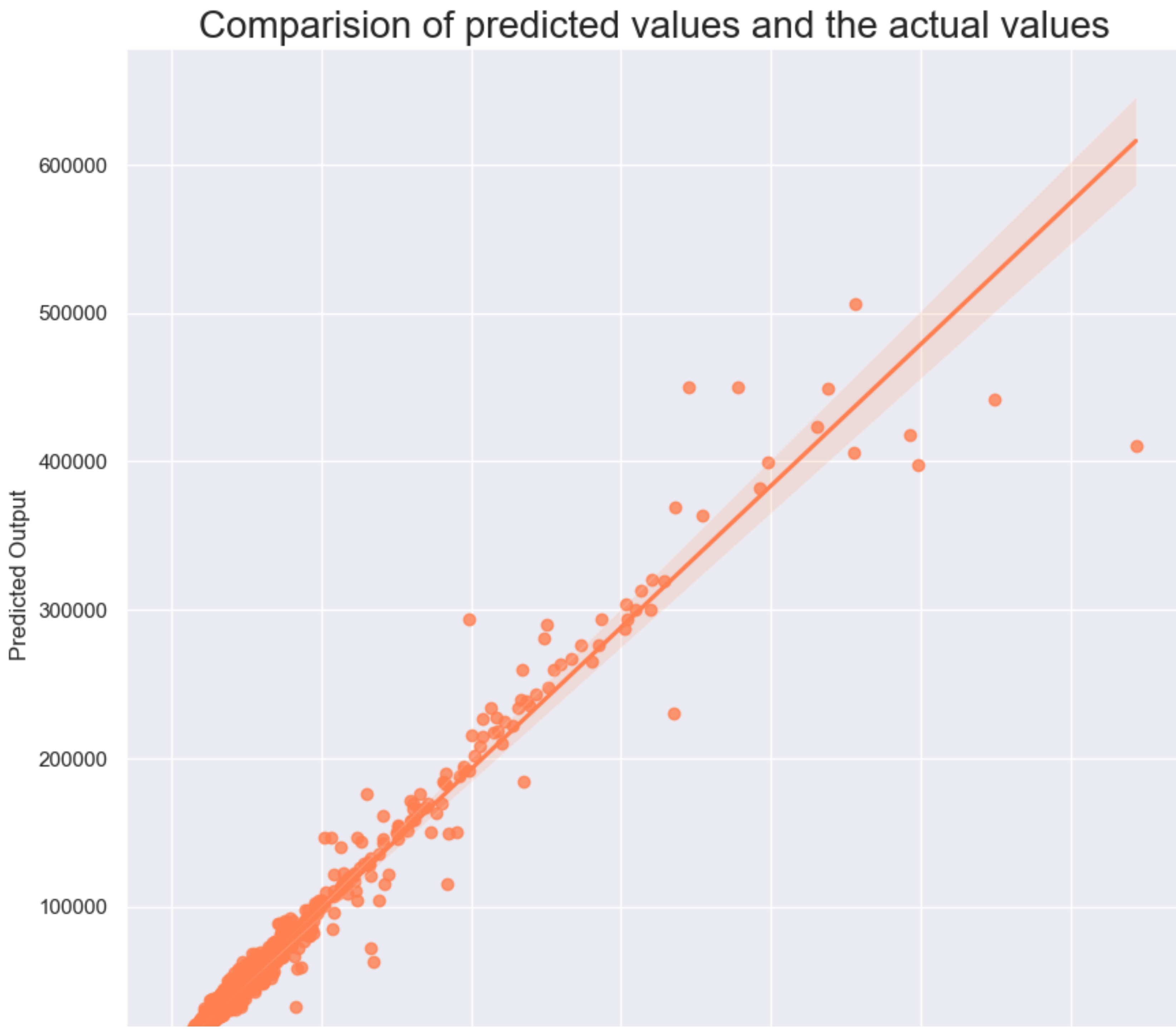
```
model = DecisionTreeRegressor(splitter = 'random')
model.fit(X_train_new, y_train)
y_predict = model.predict(X_test_new)
y_predict = pd.DataFrame(y_predict, columns = ['Predicted Output'])
results = pd.concat([y_predict, y_test.to_frame().reset_index(drop = True)], axis = 1, ignore_index = False)
```

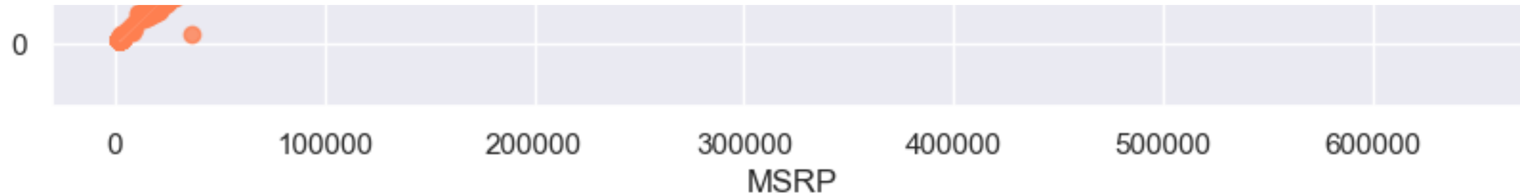
4.5.1 Regplot for Decision Tree Regressor

We would be making use of Decision Tree Regressor and understand the outputs respectively. We see that decision tree regressor also does a very good job of predicting the right outputs for the test inputs. Therefore, this model can be deployed in production. In additon to this, we have to do the hyperparameter tuning so that we would be able to get the best output for this model.

In [112…

```
plt.figure(figsize = (10, 10))
sns.regplot(data = results, y = 'Predicted Output', x = 'MSRP', color = 'coral', marker = 'o')
plt.title("Comparision of predicted values and the actual values", fontsize = 20)
plt.show()
```


Comparision of predicted values and the actual values
Predicted Output
600000
500000
400000
300000
200000
100000



We would be appending the values to the list that we have created before.

In [113…

```
error_mean_square.append(int(mean_squared_error(y_predict, y_test)))
error_mean_absolute.append(int(mean_absolute_error(y_predict, y_test)))
```

4.6 Gradient Boosting Regressor

We would be making use of gradient boosting regressor respectively. We would follow the same procedure of traning the data and getting test output and see how well the model did on the test set. There can be a few hyperparameters that we would need to tune. But it would be better to see how the model actually performs with it's default values of hyperparameters respectively.

In [114…

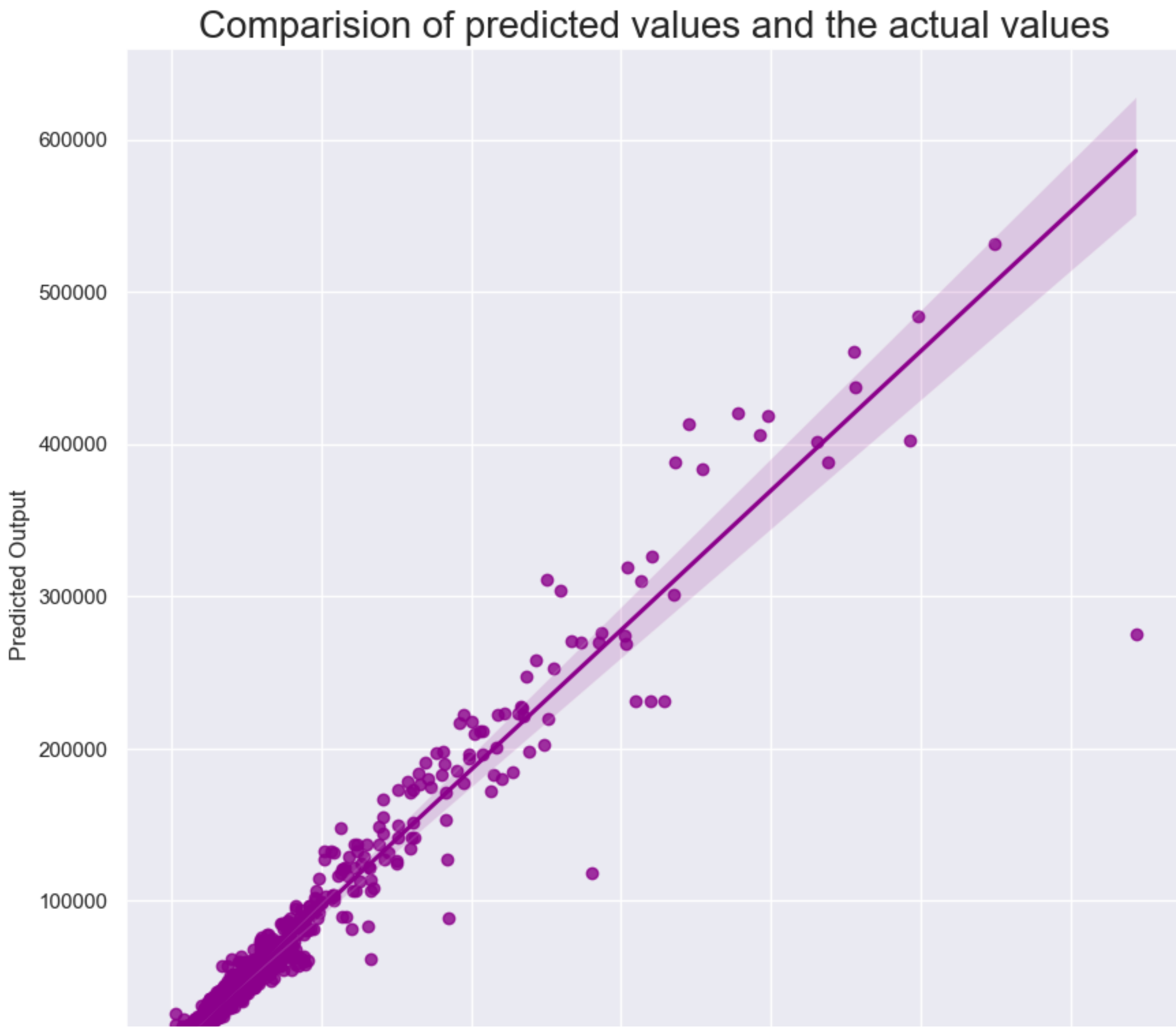
```
model = GradientBoostingRegressor()
model.fit(X_train_new, y_train)
y_predict = model.predict(X_test_new)
y_predict = pd.DataFrame(y_predict, columns = ['Predicted Output'])
results = pd.concat([y_predict, y_test.to_frame().reset_index(drop = True)], axis = 1, ignore_index = False)
```

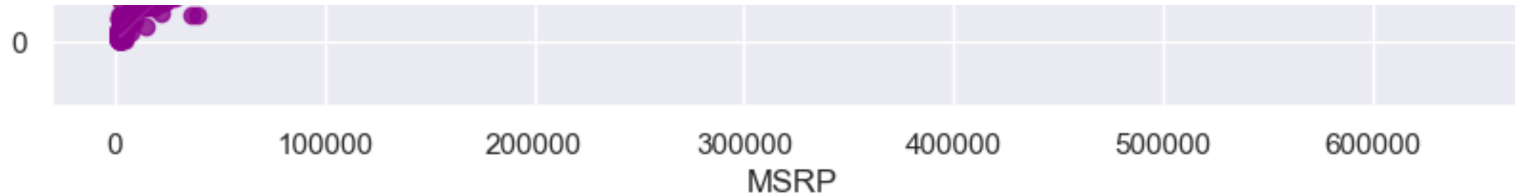
4.6.1 Regplot of Gradient Boosting Regressor

We would now be using the gradient boosting regressor and plot the values and get a scatterplot respectively. We see that the gradient boosting regressor also did a fine job in getting the most accurate predictions. There could be a few outliers in the predictions but they are few in number. Most of the points were accurately predicted with small errors in them. Therefore, this is also a good model that could be used for predictions.

In [115…

```
plt.figure(figsize = (10, 10))
sns.regplot(data = results, y = 'Predicted Output', x = 'MSRP', color = 'darkmagenta', marker = 'o')
plt.title("Comparision of predicted values and the actual values", fontsize = 20)
plt.show()
```


Comparision of predicted values and the actual values
Predicted Output
600000
500000
400000
300000
200000
100000



We would make use of the list and append the errors so that we could plot them later.

In [116…

```
error_mean_square.append(int(mean_squared_error(y_predict, y_test)))
error_mean_absolute.append(int(mean_absolute_error(y_predict, y_test)))
```

4.8 Dataframe of Machine Learning Models

Now it is time to get to the end. We would now be using the models that we have just created and making a dataframe. We would append the list values that we have been appending the error values and make a dataframe containing the models and the errors associated with them.

In [117…

```
data = {'Models': ['Linear Regression', 'K Nearest Regressor', 'Decision Tree Regressor', 'Gradient Boosting Regress
model_dataframe = pd.DataFrame(data)
```

We could have a look at the machine learning models dataframe that we have just created respectively.

In [118…

```
model_dataframe
```

Out[118]:

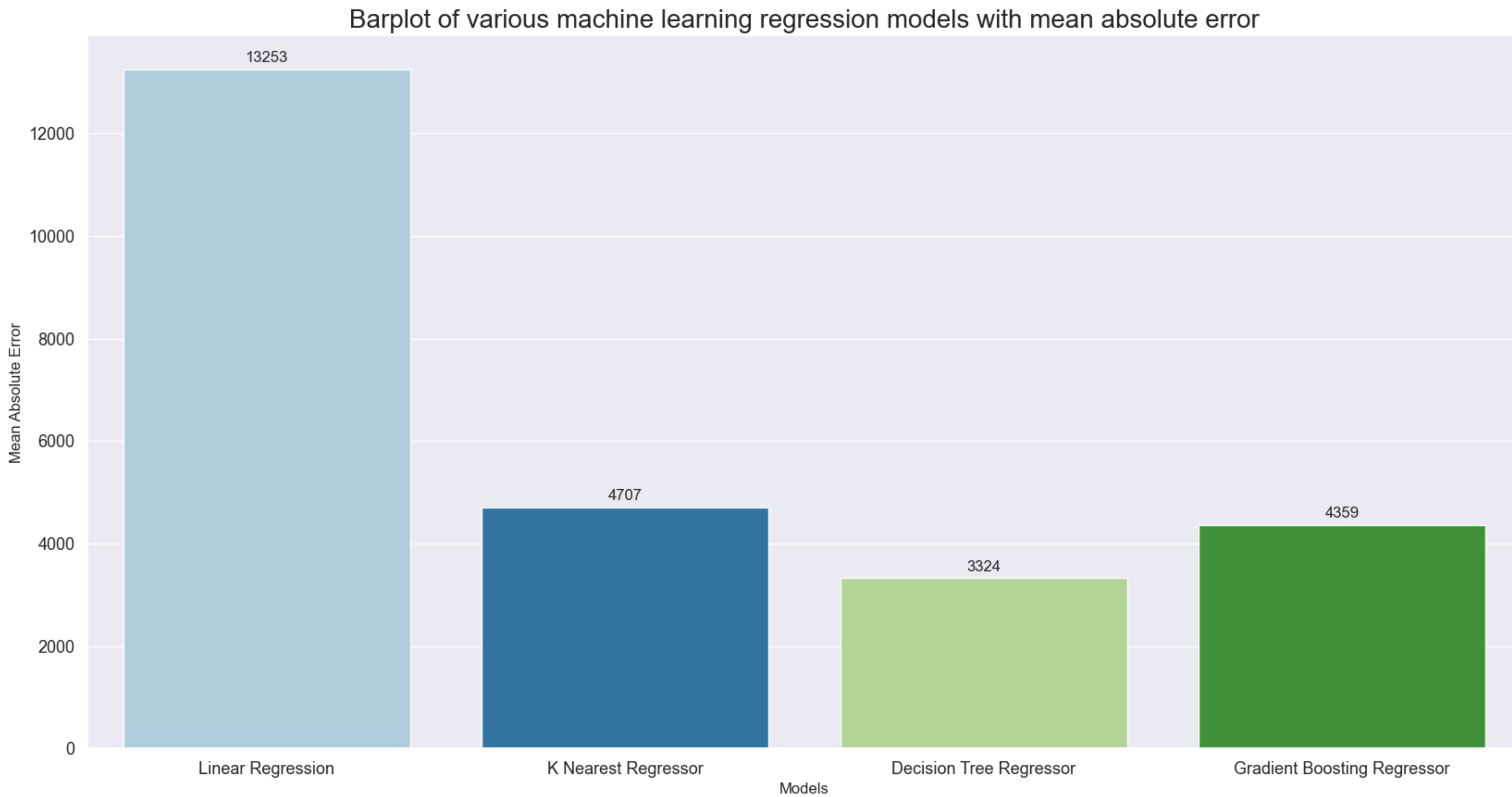
	Models	Mean Absolute Error	Mean Squared Error
0	Linear Regression	13253	596724778
1	K Nearest Regressor	4707	226658292
2	Decision Tree Regressor	3324	83498684
3	Gradient Boosting Regressor	4359	135991988

4.9 (a) Barplot of machine learning models with mean absolute error

We would be making use of the mean absolute error and understand the data fully. We can see from the graph that the 'Decision Tree Regressor' has the lowest mean absolute error. We can conclude that it is better to use the 'Decision Tree Regressor' for deploying and for predictions in the future as it has the lowest mean absolute error.

In [119…

```
plt.figure(figsize = (20, 10))
splot = sns.barplot(data = model_dataframe, x = 'Models', y = 'Mean Absolute Error', palette = 'Paired')
for p in splot.patches:
    splot.annotate(format(p.get_height(), '.0f'),
                   (p.get_x() + p.get_width() / 2., p.get_height()),
                   ha = 'center', va = 'center',
                   xytext = (0, 9),
                   textcoords = 'offset points')
plt.xticks(fontsize = 13)
plt.yticks(fontsize = 13)
plt.title("Barplot of various machine learning regression models with mean absolute error", fontsize = 20)
plt.show()
```

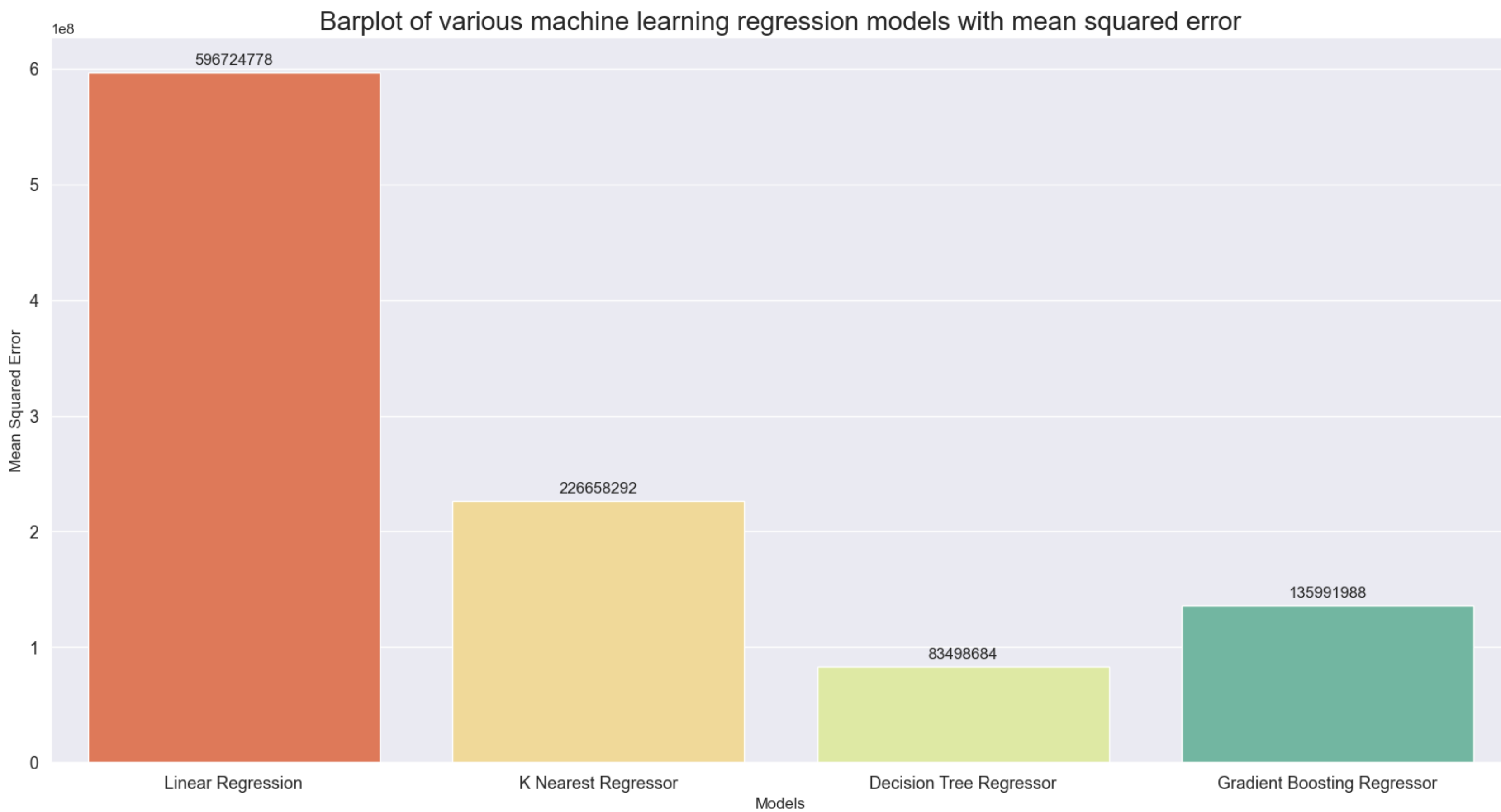


4.9 (b) Barplot of machine learning models with mean squared error

We would now be plotting the barplot of various machine learning models by taking into consideration the mean squared error respectively. We can see from the graph that 'Decision Tree Regressor' has the lowest mean squared error respectively. Therefore, it is one of the best models to use as there is low error for the testing set. We have to always compare different machine learning models and understand how the values are shaped respectively. There might be different machine learning models that would perform differently for different scenarios and different data sets respectively.

In [120…

```
plt.figure(figsize = (20, 10))
splot = sns.barplot(data = model_dataframe, x = 'Models', y = 'Mean Squared Error', palette = 'Spectral')
for p in splot.patches:
    splot.annotate(format(p.get_height(), '.0f'),
                   (p.get_x() + p.get_width() / 2., p.get_height()),
                   ha = 'center', va = 'center',
                   xytext = (0, 9),
                   textcoords = 'offset points')
plt.xticks(fontsize = 13)
plt.yticks(fontsize = 13)
plt.title("Barplot of various machine learning regression models with mean squared error", fontsize = 20)
plt.show()
```



5. Conclusion

1. We can see that using different machine learning models would lead to different values of mean absolute error and mean squared error respectively.
2. We would have to first convert all the categorical features into numerical features before we give those data points to the machine learning models for prediction. If we just give categorical features directly, there would be an error in the machine learning models respectively.
3. It is always good to shuffle the data before we split the data into training and testing set. This is done so that we have more randomness in the training data so that the machine learning models would work well on new data.
4. We have to always ensure that there are no missing values in our data. We have to replace those values so that there is no problem when we are using different machine learning models for prediction.
5. We have to also remove the outliers in our data as they would completely change some of the important predictions and lead to an increase in the error respectively.

In [ ]:

```

```