# Exploratory Data Analysis on Electric Vehicle Market in India

## Introduction:

The transition to electric vehicles (EVs) has gained significant momentum in recent years, with India emerging as one of the key players in the global EV market. This project aims to analyze the EV market trends in India using a structured dataset, offering insights into vehicle types, key patterns, and market dynamics. By leveraging data visualization, statistical techniques, and machine learning models, this analysis provides a comprehensive understanding of the current state and potential growth of the EV industry.

The workflow begins with data cleaning and preprocessing to ensure consistency and reliability. Exploratory data analysis (EDA) reveals valuable patterns and relationships within the dataset. Advanced techniques like Principal Component Analysis (PCA) and clustering algorithms are utilized to uncover hidden structures and segment the market effectively. Finally, a classification model is developed and evaluated to predict key outcomes, making this project a robust blend of analytics and machine learning.

This study not only highlights the opportunities within the EV market but also demonstrates how data-driven approaches can guide strategic decision-making in this rapidly evolving industry.

## 1. Initial Setup:

**Purpose:**

- This block sets up the environment by importing libraries for data analysis, visualization, and machine learning.
- It also installs and configures additional tools like Plotly and Kaleido for high-quality visualizations.

**Highlights:**

- Libraries like `Numpy`, `Pandas`, `Matplotlib`, and `Seaborn` handle data processing and visualization.
- Machine learning tasks are supported by `Scikit-learn`, `Yellowbrick`, and `Statsmodels`.
- Display settings for precision and formatting ensure consistent outputs.

```
import numpy as np
%pip install plotly==5.8.0
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
from tqdm import tqdm
import seaborn as sb
import statsmodels.api as sm
import plotly.express as px
from google.colab import files
%pip install kaleido
import kaleido
from sklearn.preprocessing import StandardScaler,PowerTransformer
from sklearn.decomposition import PCA
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import KMeans, MeanShift, estimate_bandwidth
from sklearn.datasets import make_blobs
from yellowbrick.cluster import KElbowVisualizer, SilhouetteVisualizer, InterclusterDistance
from collections import Counter
from sklearn.model_selection import cross_validate,train_test_split
from sklearn.linear_model import LinearRegression,LogisticRegression
from sklearn import metrics
from sklearn.metrics import r2_score,silhouette_score,confusion_matrix,accuracy_score
pd.set_option("display.precision",3)
np.set_printoptions(precision=5, suppress=True)
pd.options.display.float_format = '{:.4f}'.format
import plotly.io as pio
pio.renderers.default = "svg"
```

## 2. Data Loading and Preprocessing:

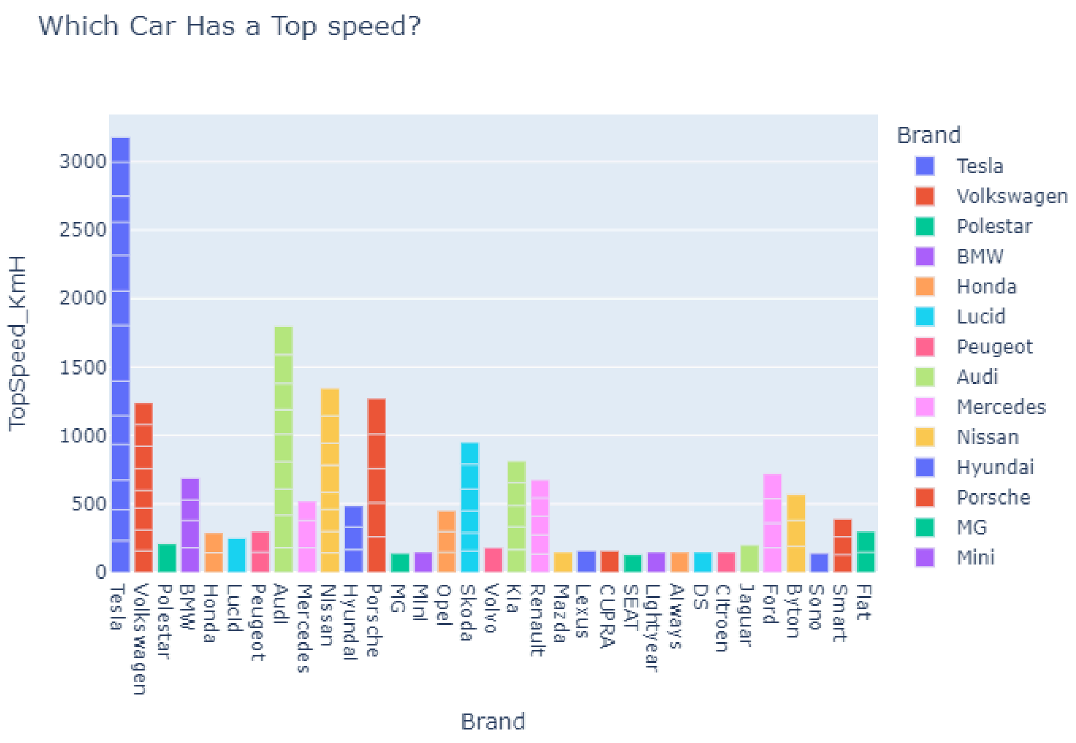
```
df = pd.read_csv('data.csv')
df.drop('Unnamed: 0', axis=1, inplace=True)
df['inr(10e3)'] = df['PriceEuro']*0.08320
df['RapidCharge'].replace(to_replace=['No','Yes'],value=[0, 1],inplace=True)
df.head()
```

This code reads a dataset, cleans it by removing unnecessary columns, and preprocesses it for analysis. It includes currency conversion, encoding categorical values into numeric format, and previewing the transformed data. This setup prepares the data for further exploration or modeling.

## 3. Visualizing Top Speeds by Car Brand:

**fig = px.bar(df,x='Brand',y = 'TopSpeed_KmH',color = 'Brand',title = 'Which Car Has a Top speed?',labels = {'x':'Car Brands','y':'Top Speed Km/H'}) pio.show(fig)**

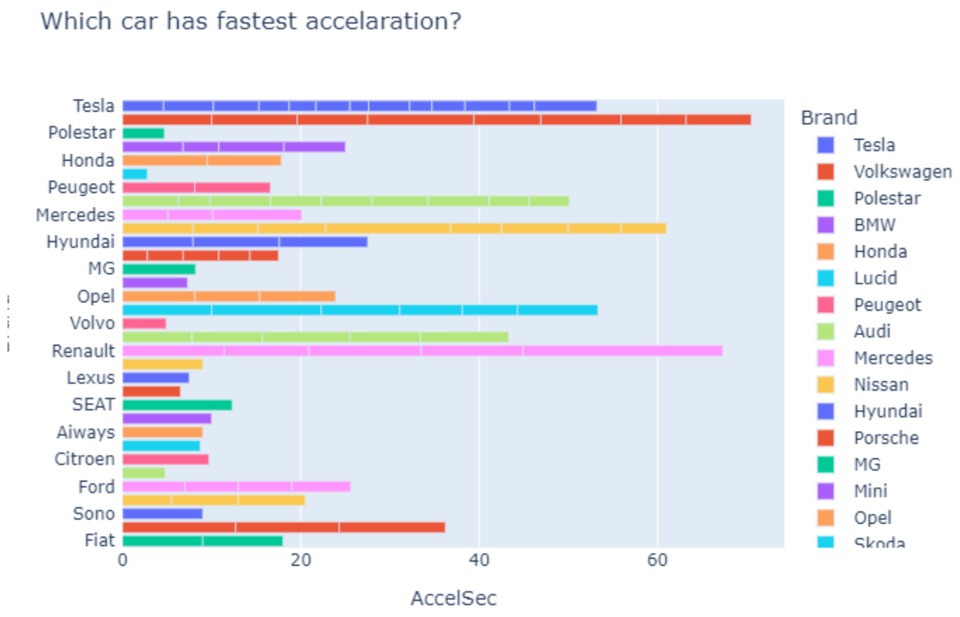
This code creates an interactive bar chart using Plotly Express to compare the top speeds of different car brands. It assigns distinct colors to each brand, sets a descriptive title, and customizes axis labels for clarity. The chart is then displayed using Plotly's rendering engine.



## 4. Visualizing Car Acceleration by Brand:

**next fig = px.bar(df,x='AccelSec',y = 'Brand',color = 'Brand',title = 'Which car has fastest accelaration?',labels = {'x':'Accelaration','y':'Car Brands'})pio.show(fig)**

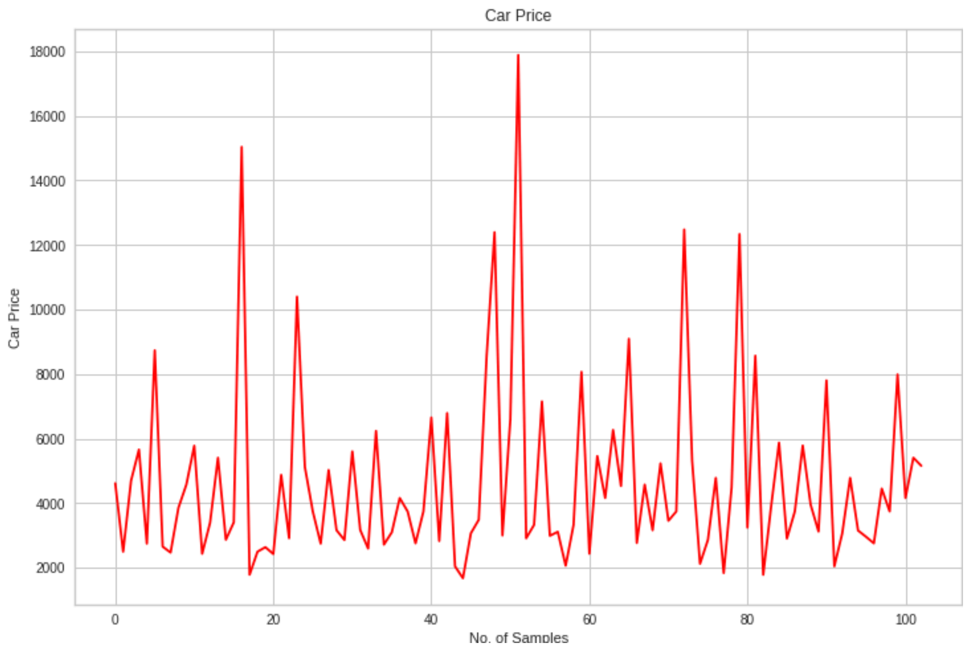
This code generates an interactive horizontal bar chart to compare the acceleration times of different car brands. Each brand is color-coded for distinction, with axis labels and a title provided for better interpretation. The chart highlights which car brand achieves the fastest acceleration and is displayed using Plotly's rendering engine.



## 5. Plotting Car Prices Over Samples:

```
df['inr(10e3)'].plot(figsize = (12,8),title='Car Price',xlabel = 'No. of Samples',ylabel = 'Car Price',color = 'red')
```

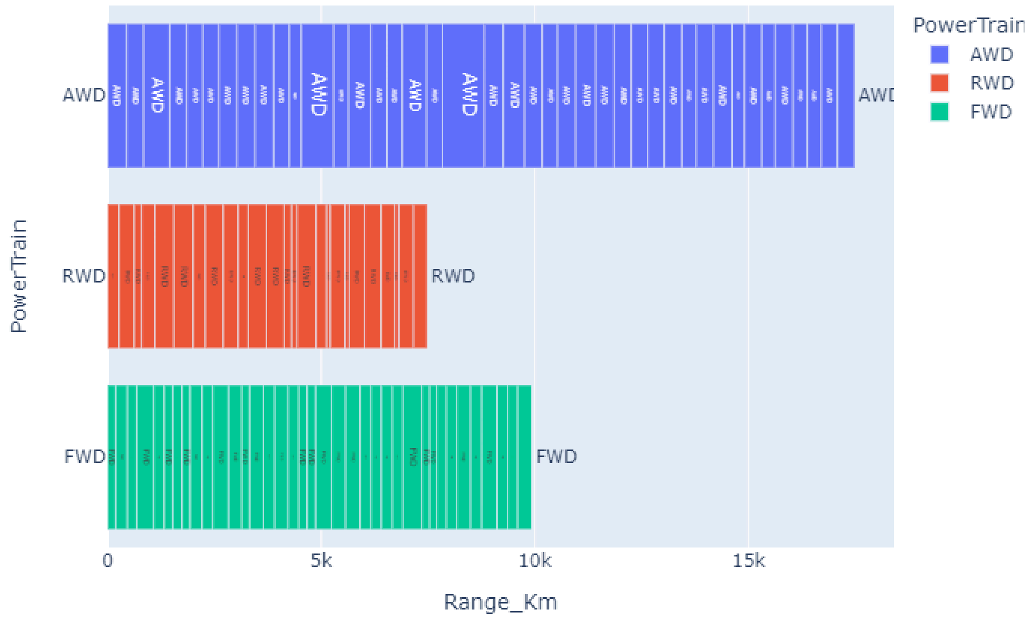
This code creates a line plot displaying car prices (in INR) across different samples. The plot is customized with a title, axis labels, and a red color for the line, making it easy to visualize the trend or distribution of car prices in the dataset. The plot is also set to a larger size for better readability.



## 6. Visualizing Range vs. PowerTrain by Car Type:

```
ig = px.bar(df,x = 'Range_Km',y = 'PowerTrain',color = 'PowerTrain',text='PowerTrain')
pio.show(fig)
```

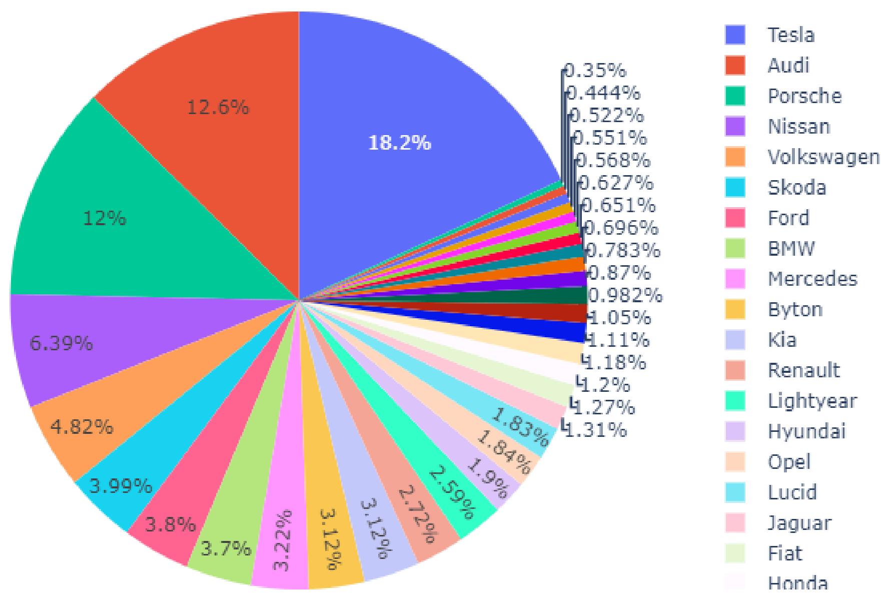
This code generates an interactive bar chart that compares the range (in kilometers) of cars based on their powertrain types (e.g., electric, hybrid, etc.). Each powertrain type is color-coded for distinction, and the type is displayed as text on each bar for clarity. The chart provides insights into how different powertrain technologies impact the car's range and is displayed using Plotly's rendering engine.



## 7. Visualizing Car Price Distribution by Brand:

```
fig = px.pie(df,names = 'Brand',values = 'inr(10e3)') pio.show(fig)
```

This code creates an interactive pie chart that shows the distribution of car prices (in INR) across different car brands. Each segment represents a brand, and its size corresponds to the total price for that brand. The chart provides a visual understanding of how car prices are distributed among the different brands in the dataset. The plot is rendered using Plotly's engine.



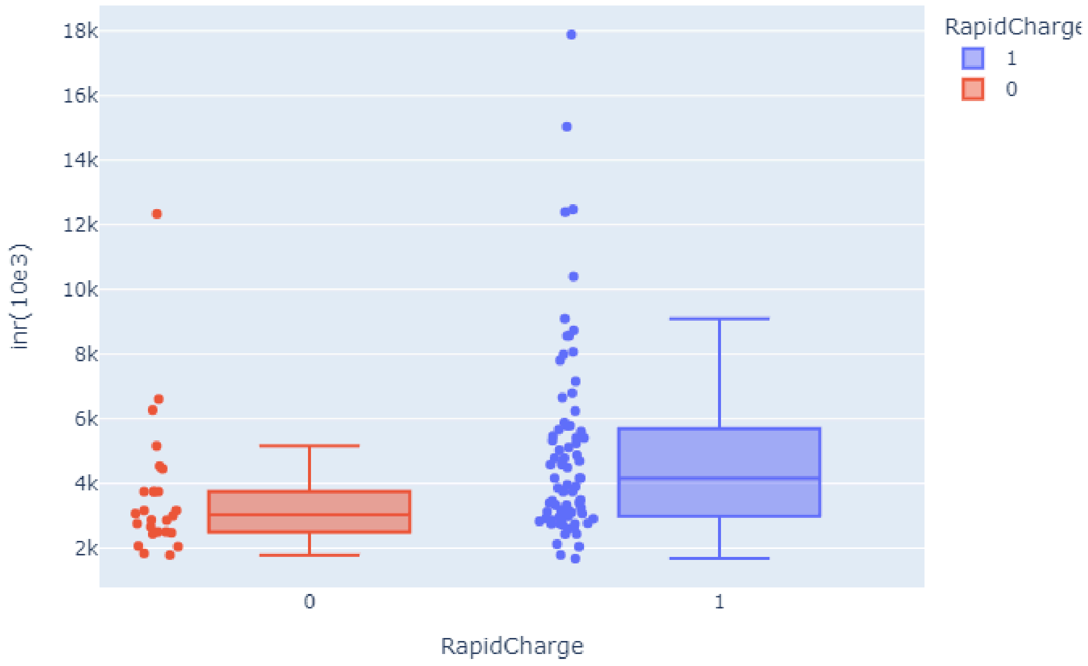
## 8. 3D Scatter and Box Plot Visualizations:

```
fig = px.scatter_3d(df,x = 'Brand',y = 'Seats',z = 'Segment',color='Brand')

fig = px.scatter_3d(df,x = 'Brand',y = 'AccelSec',z = 'inr(10e3)',color = 'Brand')

fig = px.box(df,x='RapidCharge',y = 'inr(10e3)',color = 'RapidCharge',points='all')

pio.show(fig)
```

This code generates three visualizations to explore relationships within the dataset:

1. **3D Scatter Plots**: Two 3D scatter plots are created:
   - One to visualize the relationship between car brand, number of seats, and car segment.
   - The second compares car brand, acceleration time, and car price (in INR).
2. **Box Plot**: A box plot shows the distribution of car prices (in INR) based on the `RapidCharge` feature, highlighting how charging capabilities influence pricing.

All visualizations are interactive and rendered using Plotly, helping to uncover patterns and insights in the data.



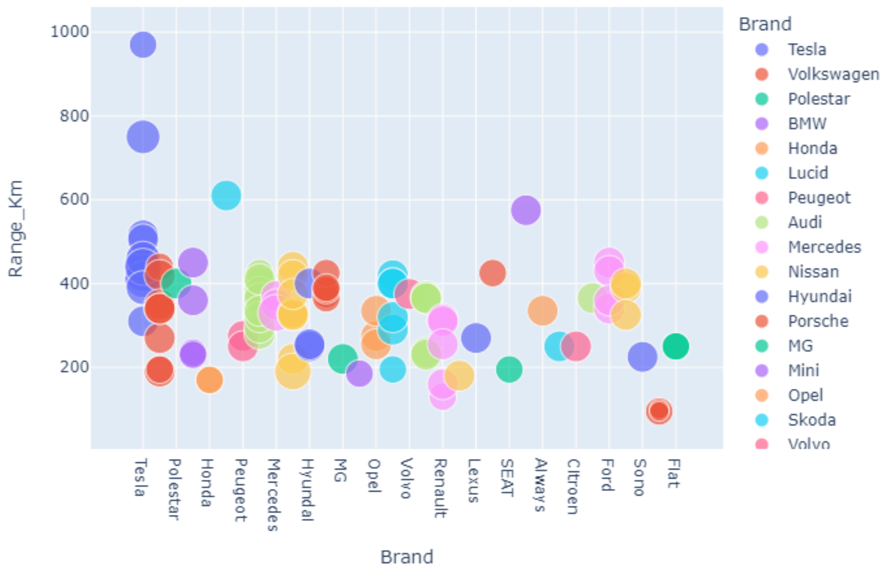
### 9. Bubble Chart of Range vs. Brand with Seats and Price Information:

```
        fig = px.scatter(df,x = 'Brand',y = 'Range_Km',size='Seats',color =
'Brand',hover_data=['RapidCharge','inr(10e3)']) pio.show(fig)
```

This code generates an interactive scatter plot (bubble chart) where:

- The x-axis represents car brands.
- The y-axis shows the car's range (in kilometers).
- The size of each point (bubble) is determined by the number of seats.
- Each bubble is color-coded by brand.

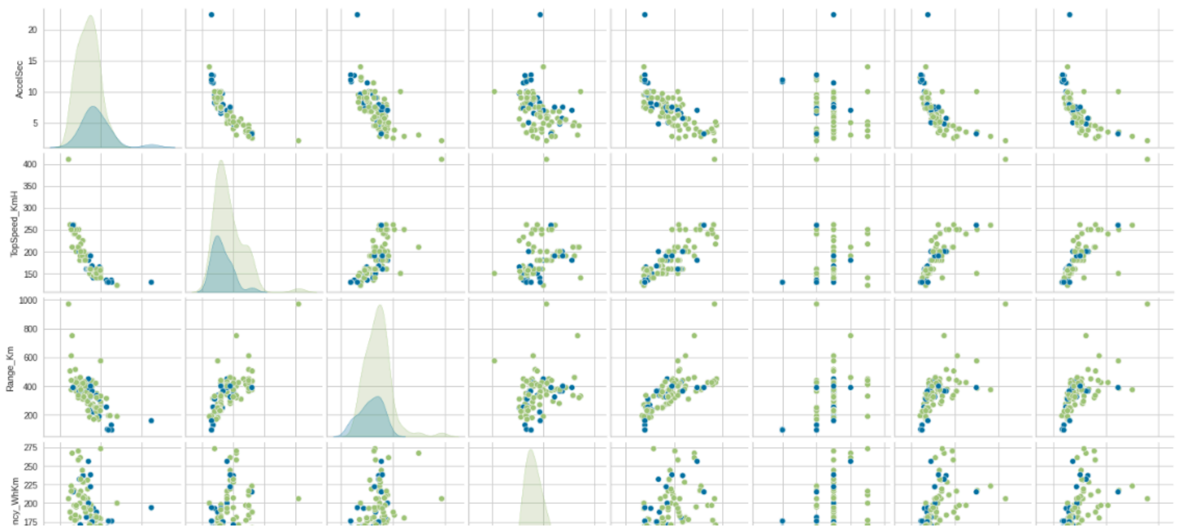
Additionally, the hover functionality displays more information, including whether the car has rapid charging and its price (in INR), providing a detailed overview of the dataset in one chart.



## 10. Pairwise Plot of Variables with RapidCharge Hue:

```
sb.pairplot(df,hue='RapidCharge')
```

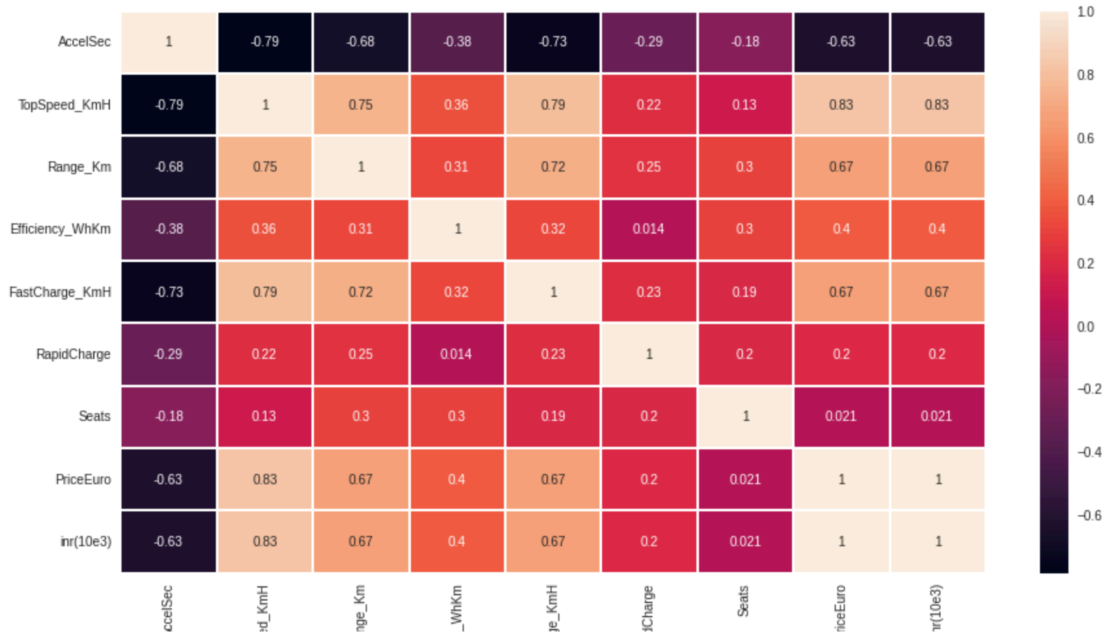
This code generates a pairplot using Seaborn, which shows pairwise relationships between numerical features in the dataset. Each pair of features is plotted against each other in a grid of scatter plots, and the color of the points represents the `RapidCharge` variable (using different colors for cars with and without rapid charging). This helps visualize how the features relate to each other and how they vary based on the presence of rapid charging.



## 11. Correlation Heatmap of Dataset Features:

```
        ax= plt.figure(figsize=(15,8))
sb.heatmap(df.corr(),linewidths=1,linecolor='white',annot=True)
```

This code creates a heatmap using Seaborn to visualize the correlation matrix of the dataset. The correlation coefficients between numeric variables are displayed, showing how strongly different features in the dataset are related to each other. The `annot=True` option adds the correlation values directly onto the heatmap, and the `linewidths=1` and `linecolor='white'` options enhance the visual separation between cells. This plot helps identify patterns and relationships between features in the dataset.



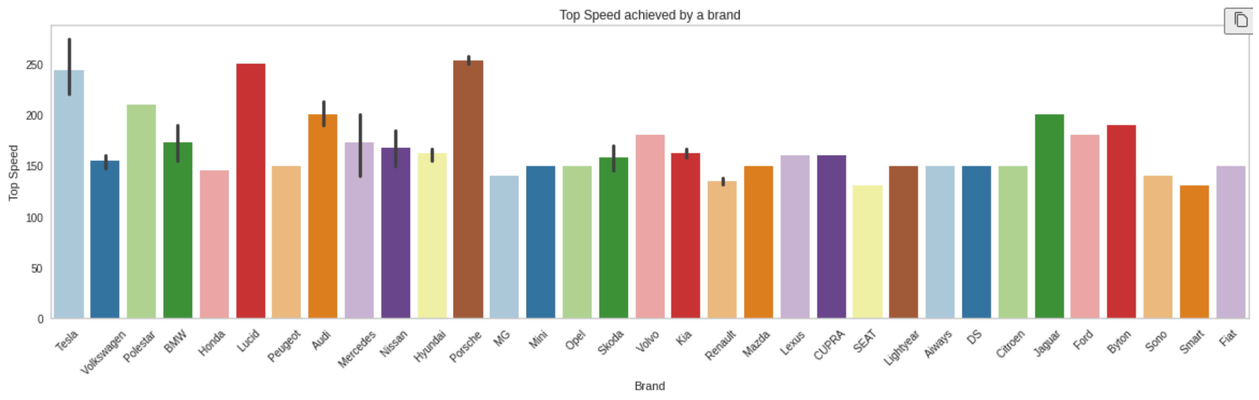
## 12. Bar Plot of Top Speed by Car Brand:

```
ax= plt.figure(figsize=(20,5))

sb.barplot(x='Brand',y='TopSpeed_KmH',data=df,palette='Paired')

plt.grid(axis='y')

plt.title('Top Speed achieved by a brand')

plt.xlabel('Brand')

plt.ylabel('Top Speed')

plt.xticks(rotation=45)
```

This code generates a bar plot to visualize the top speed achieved by each car brand. It uses Seaborn's `barplot` to display the car brands on the x-axis and their corresponding top speeds (in km/h) on the y-axis. The bars are color-coded using the 'Paired' palette. The plot includes:

- A grid along the y-axis for better readability.
- A title, axis labels, and rotated x-axis ticks for clarity in displaying car brand names.

This visualization helps in comparing the top speeds of various car brands.



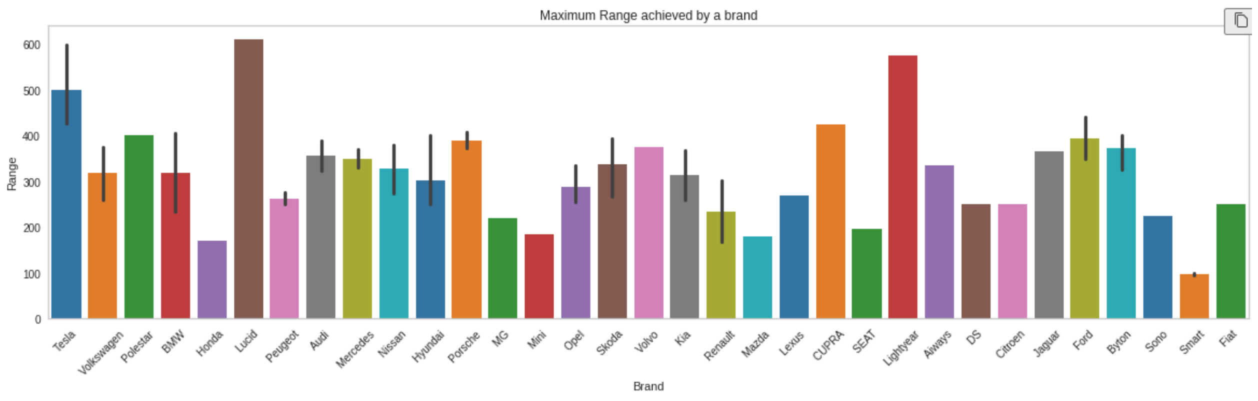
## 13. Bar Plot of Maximum Range by Car Brand:

```
ax= plt.figure(figsize=(20,5))

sb.barplot(x='Brand',y='Range_Km',data=df,palette='tab10')

plt.grid(axis='y')

plt.title('Maximum Range achieved by a brand')

plt.xlabel('Brand')

plt.ylabel('Range')

plt.xticks(rotation=45)
```

This code creates a bar plot to visualize the maximum range (in kilometers) achieved by each car brand. The plot uses Seaborn's `barplot` to display:

- Car brands on the x-axis.
- The corresponding range values on the y-axis.

The bars are color-coded using the 'tab10' palette, and the grid is enabled along the y-axis for better readability. The plot includes a title, axis labels, and rotated x-axis labels for clear presentation, making it easy to compare the maximum range of different car brands.



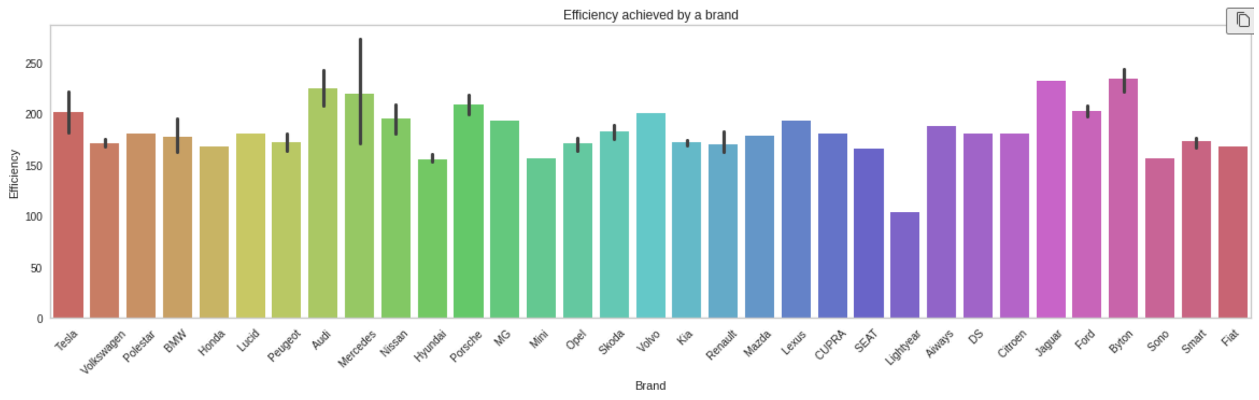
## 14. Bar Plot of Efficiency by Car Brand:

```
ax= plt.figure(figsize=(20,5))
sb.barplot(x='Brand',y='Efficiency_WhKm',data=df,palette='hls')
plt.grid(axis='y')
plt.title('Efficiency achieved by a brand')
plt.xlabel('Brand')
plt.ylabel('Efficiency')
plt.xticks(rotation=45)
```

This code generates a bar plot to visualize the efficiency (in Wh/km) achieved by each car brand. The plot uses Seaborn's `barplot` to show:

- Car brands on the x-axis.
- The corresponding efficiency values on the y-axis.

The bars are color-coded using the 'hls' palette, and the grid is displayed along the y-axis for improved readability. The plot includes a title, axis labels, and rotated x-axis labels to make it easier to compare the efficiency of different car brands.



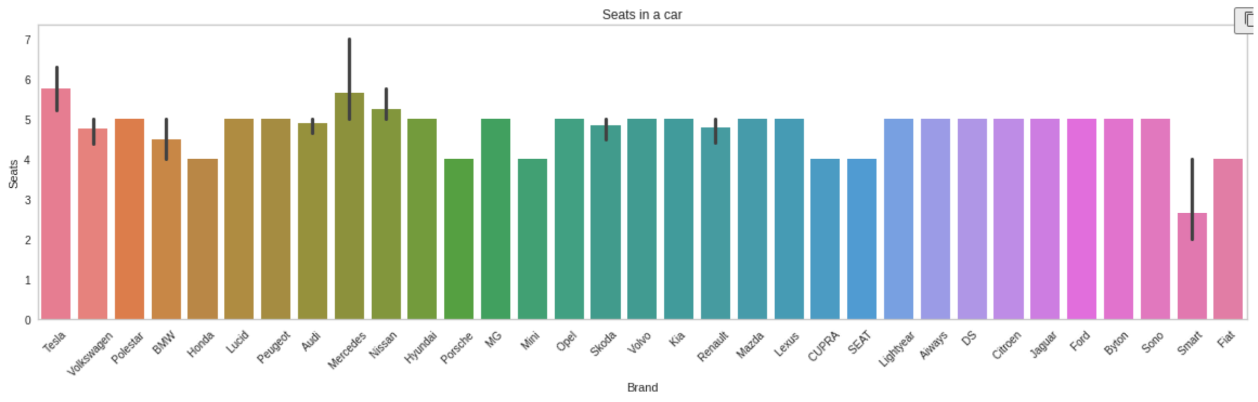
## 15. Bar Plot of Number of Seats by Car Brand:

```
ax= plt.figure(figsize=(20,5))

sb.barplot(x='Brand',y='Seats',data=df,palette='husl')

plt.grid(axis='y')

plt.title('Seats in a car')

plt.xlabel('Brand')

plt.ylabel('Seats')

plt.xticks(rotation=45)
```

This code generates a bar plot to visualize the number of seats in cars from different brands. The plot uses Seaborn's `barplot` to display:

- Car brands on the x-axis.
- The corresponding number of seats on the y-axis.

The bars are color-coded using the 'husl' palette, and a grid is shown along the y-axis for better clarity. The plot also includes a title, axis labels, and rotated x-axis labels, making it easy to compare the seating capacity across car brands.



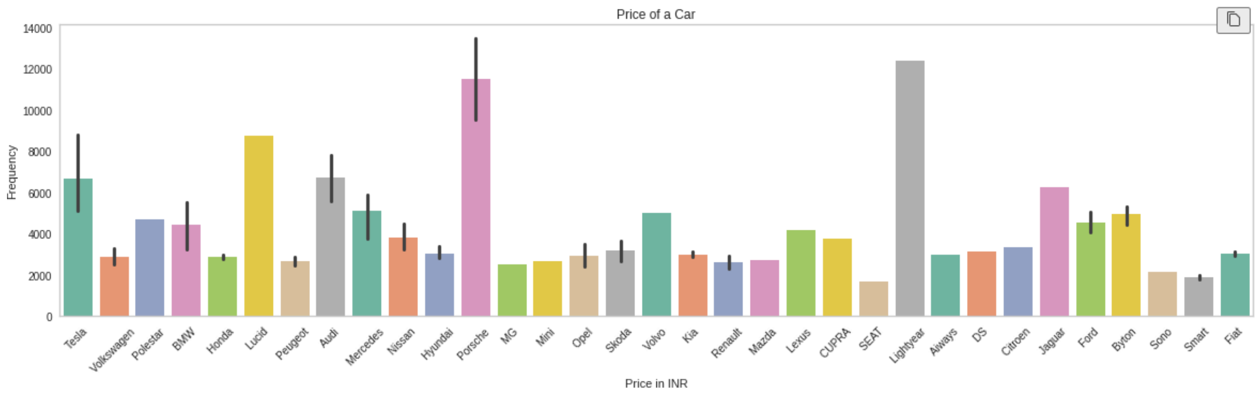
## 16. Bar Plot of Car Price Distribution by Brand:

```
ax= plt.figure(figsize=(20,5))

sb.barplot(x='Brand',y='inr(10e3)',data=df,palette='Set2')

plt.title('Price of a Car')

plt.xlabel('Price in INR')

plt.grid(axis='y')

plt.ylabel('Frequency')

plt.xticks(rotation=45)
```

This code generates a bar plot to visualize the distribution of car prices (in INR) across different car brands. The plot uses Seaborn's `barplot` to display:

- Car brands on the x-axis.
- The corresponding car price (in INR) on the y-axis.

The bars are color-coded using the 'Set2' palette. A grid is shown along the y-axis to improve readability. The plot includes a title, axis labels, and rotated x-axis labels for clarity, helping to compare the prices of cars from different brands.



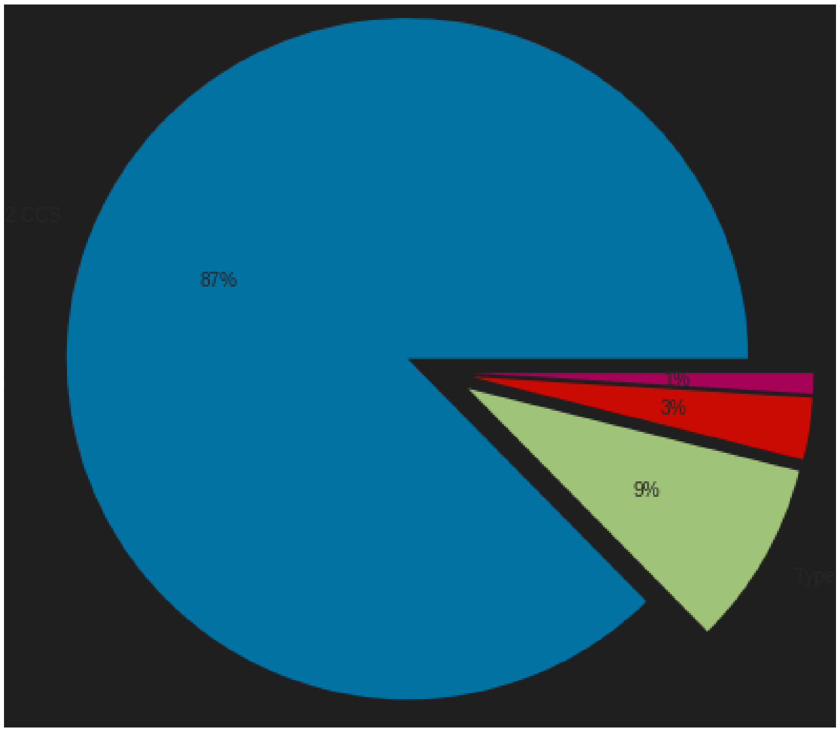
## 17. Pie Chart of Plug Type Distribution:

```
df['PlugType'].value_counts().plot.pie(figsize=(8,15),autopct='%.0f%%',explode=(.1,.1,.1,.1))
plt.title('Plug Type')
```

This code generates a pie chart to visualize the distribution of different plug types in the dataset. It uses the `value_counts()` method to count the occurrences of each plug type in the `PlugType` column, and then plots the data as a pie chart. The chart displays:

- Each plug type as a segment.
- The percentage share of each plug type (`autopct='%.0f%%'` displays the percentage without decimals).
- The `explode` parameter slightly offsets each segment to make the chart more visually distinct.

The chart includes a title to indicate that it represents the distribution of plug types.



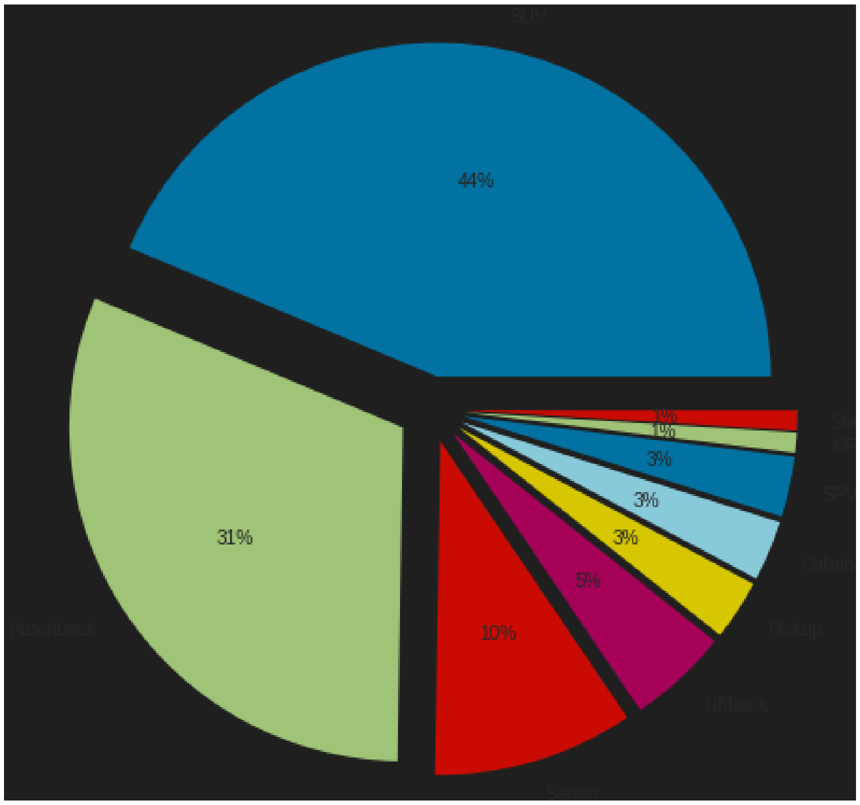
### 18. Pie Chart of Body Style

```
Distribution:df['BodyStyle'].value_counts().plot.pie(figsize=(8,15),autopct='%.0f%%',explode=(0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1))

plt.title('Body Style')
```

This code generates a pie chart to visualize the distribution of different body styles in the dataset. It uses the `value_counts()` method to count how many times each body style appears in the `BodyStyle` column and then plots these values as a pie chart. The chart displays:

- Each body style as a segment of the pie.
- The percentage share of each body style, displayed without decimals (`autopct='%.0f%%'`).
- The `explode` parameter slightly offsets each segment, enhancing visibility for each body style.

The chart includes a title that clearly indicates it represents the distribution of body styles across the dataset.



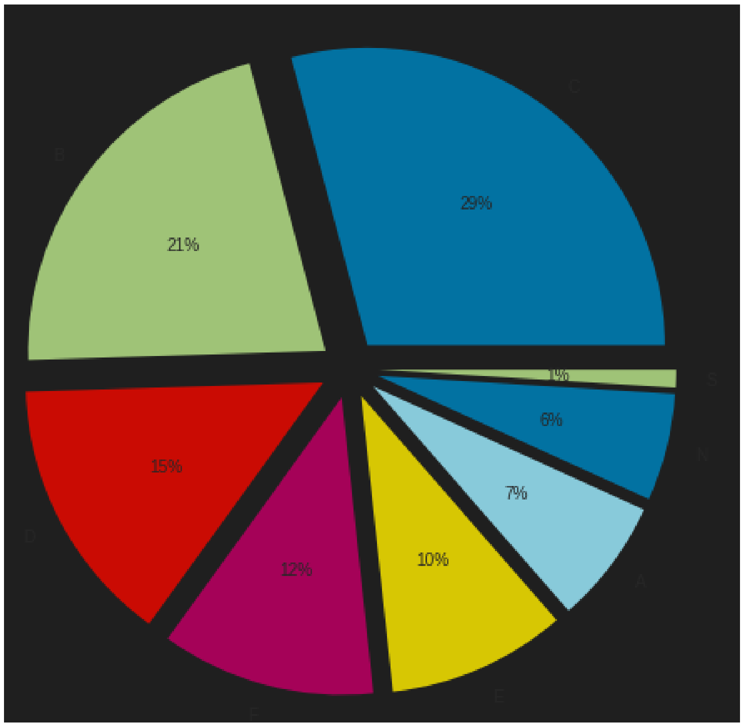
## 19. Pie Chart of Car Segment Distribution:

**df['Segment'].value_counts().plot.pie(figsize=(8,15),autopct='%.0f%%',explode=(0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1)) plt.title('Segment')**

This code generates a pie chart to visualize the distribution of car segments in the dataset. It uses the `value_counts()` method to count how many times each segment appears in the `Segment` column and then plots the data as a pie chart. The chart shows:

- Each car segment as a segment of the pie.
- The percentage share of each segment, displayed without decimals (`autopct='%.0f%%'`).
- The `explode` parameter is applied to slightly offset each segment for better visual clarity.

The chart also includes a title indicating that it represents the distribution of car segments in the dataset.



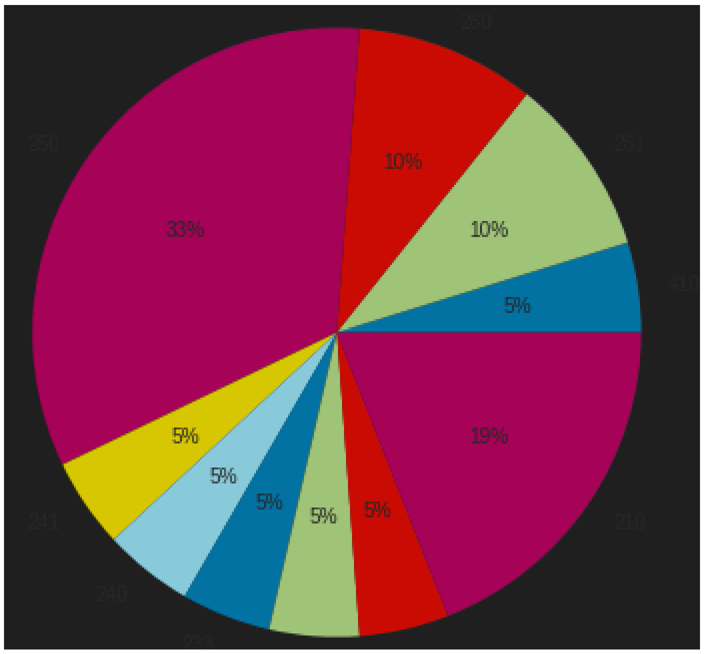
## 20. Pie Chart of Cost Based on Top Speed:

```
df3= df[["TopSpeed_KmH", "inr(10e3)"]].groupby("TopSpeed_KmH").count()

df4= df[["Range_Km", "inr(10e3)"]].groupby("Range_Km").count()

df5= df[["Range_Km", "TopSpeed_KmH"]].groupby("Range_Km").count()

df5.head()

df3=df3.sort_values("TopSpeed_KmH",ascending = False).head(10)

df4=df4.sort_values("Range_Km",ascending = False).head(10)

df5=df5.sort_values("Range_Km",ascending = False).head(10)

plt.figure(figsize=(10,7))

plt.title('Cost based on top speed')

plt.pie(x=df3["inr(10e3)"],labels=df3.index,autopct='%1.0f%%')

plt.show()
```

This code generates a pie chart to visualize the distribution of car prices based on the top speed of the cars in the dataset. Here's the breakdown:

- The dataset is first grouped by `TopSpeed_KmH` and `inr(10e3)` to count the number of cars at each top speed, using `groupby` and `count`.
- Then, the data is sorted to get the top 10 top speeds with the highest counts.
- A pie chart is created with the top 10 top speeds (`df3["inr(10e3)"]`), where the slices represent the proportion of cars at each top speed.

The `autopct='%1.0f%%'` ensures that the percentages are displayed without decimals. This pie chart shows the relative distribution of car prices for different top speeds.



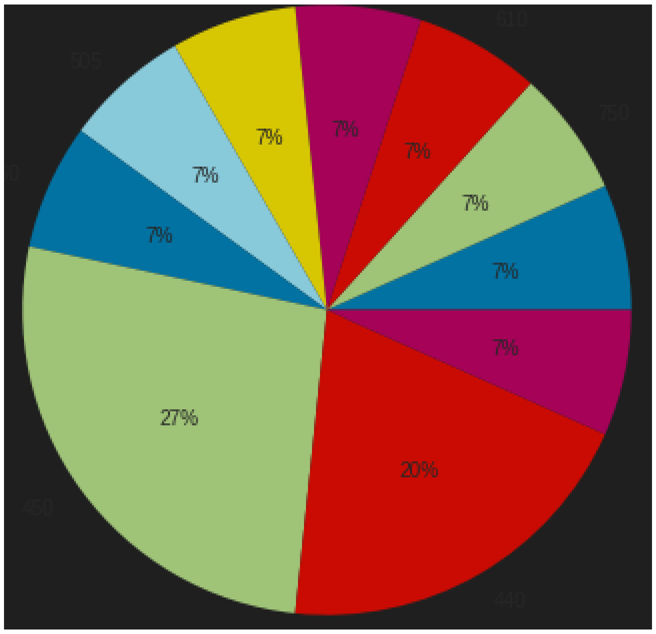
## 21. Pie Chart of Top Speeds Based on Maximum Range:

```
plt.figure(figsize=(10,7))

plt.title('Top Speeds based on Maximum Range')

plt.pie(x=df5["TopSpeed_KmH"],labels=df5.index,autopct='%1.0f%%')

plt.show()
```

This code generates a pie chart to visualize the distribution of top speeds for cars, grouped by their maximum range. Here's the process:

- The dataset is grouped by `Range_Km` and `TopSpeed_KmH`, counting the number of cars at each maximum range value using `groupby` and `count`.
- The data is then sorted to get the top 10 maximum ranges (`df5`), with the top speeds for these ranges.
- A pie chart is created to show the proportion of cars with different top speeds for the top 10 maximum ranges.

The `autopct='%1.0f%%'` parameter ensures that the percentages displayed in the chart are rounded to whole numbers. The chart represents the distribution of top speeds for cars based on their maximum range.



## 22. Predicting Car Prices and RapidCharge Using Linear and Logistic Regression:

```
df['PowerTrain'].replace(to_replace=['RWD','AWD','FWD'],value=[0, 2,1],inplace=True)
x=df[['AccelSec','Range_Km','TopSpeed_KmH','Efficiency_WhKm', 'RapidCharge','PowerTrain']]
y=df['PriceEuro']
x= sm.add_constant(x)
results = sm.OLS(y,x)
model=results.fit()
model.summary()

X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3,random_state=365)
lr= LinearRegression()
lr.fit(X_train, y_train)
pred = lr.predict(X_test)

r2=(r2_score(y_test,pred))
print(r2*100)

y1=df[['RapidCharge']]
x1=df[['PriceEuro']]

log= LogisticRegression()
log.fit(X1_train, y1_train)
pred1 = log.predict(X1_test)
Pred1
```

This section of the code does the following:

1. **Data Preprocessing**:
   - The `PowerTrain` column, which contains values such as 'RWD', 'AWD', and 'FWD', is replaced with numerical values: 0 for 'RWD', 2 for 'AWD', and 1 for 'FWD'. This is a common technique for encoding categorical data to be used in regression models.
2. **Preparing the Data for Regression**:
   - A subset of the data (x) is selected, containing features like `AccelSec`, `Range_Km`, `TopSpeed_KmH`, `Efficiency_WhKm`, `RapidCharge`, and the newly encoded `PowerTrain` column.
   - The target variable (y) is `PriceEuro`, which represents the price of the car in Euros.
3. **OLS Regression (Ordinary Least Squares)**:
   - An OLS regression model is created using the `sm.OLS()` method from the statsmodels library. The model is fitted using y (car prices) as the dependent variable and x (the features) as the independent variables.
   - The `model.summary()` call generates a detailed summary of the regression results, including coefficients, p-values, R-squared value, and other statistics.

4. **Linear Regression**:
    - The dataset is split into training and testing sets using `train_test_split()`, with 70% of the data used for training and 30% used for testing.
    - A Linear Regression model (`lr`) is trained on the training set (`X_train` and `y_train`), and predictions are made on the testing set (`X_test`) using `lr.predict()`.
    - The R-squared value (`r2_score`) is computed to evaluate the model's performance on the testing data, representing the proportion of variance in the target variable (car price) explained by the model.
5. **Logistic Regression**:
    - The code attempts to predict the `RapidCharge` column (binary values: Yes/No) based on car prices using a Logistic Regression model (`log`).
    - However, there is an issue in the code where `X1_train` and `y1_train` are used but are not defined. The correct variables should be `X_train` and `y_train`, respectively.

## 23. Stripplot for Car Top Speed vs. Fast Charging Range:

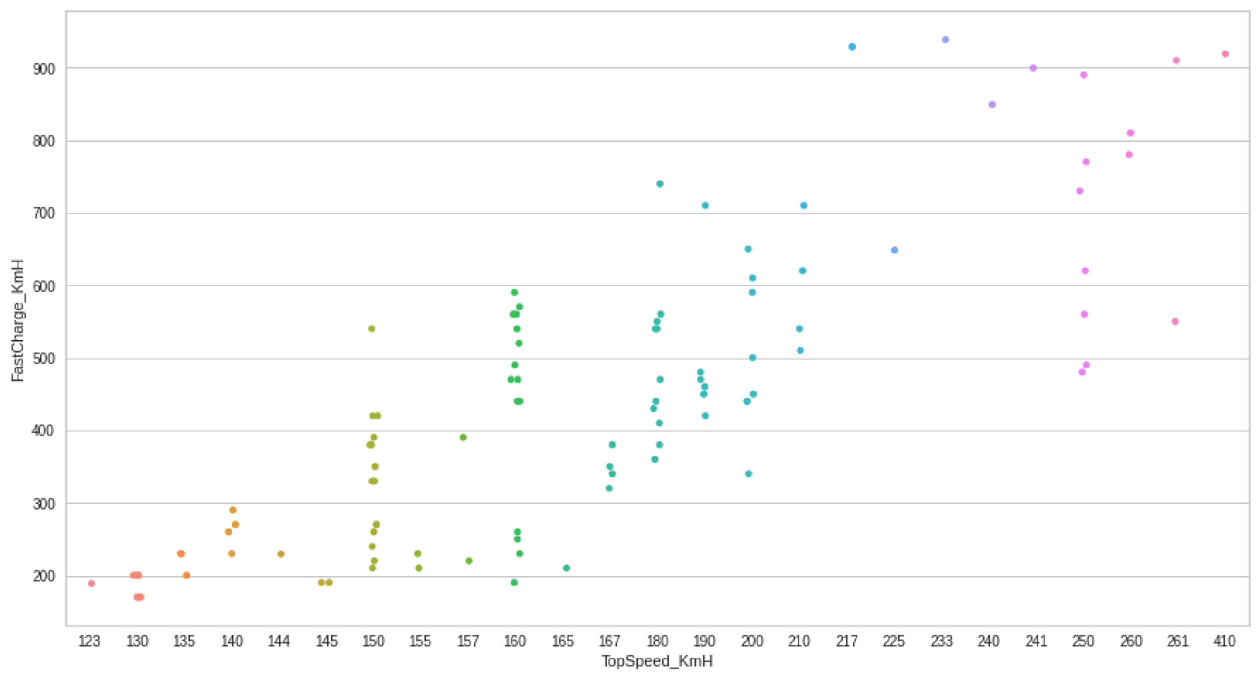
```
        ax=plt.subplots(figsize=(15,8))
sb.stripplot(x='TopSpeed_KmH', y='FastCharge_KmH', data=df, jitter=True)
```

This section of the code generates a **strip plot**, which is a type of scatter plot that shows individual data points along a single axis. Here's the breakdown of what the code does:

1. **Creating a Strip Plot**:
    - The `stripplot` function from Seaborn is used to plot individual data points on the `TopSpeed_KmH` (Top Speed in Km/h) axis against the `FastCharge_KmH` (Fast Charging Range in Km/h) axis.
    - The `jitter=True` parameter adds a small random noise to the positions of the data points along the categorical axis (in this case, `TopSpeed_KmH`). This helps avoid overlap between points and makes it easier to see the distribution of values.
2. **Figure Size**:
    - The `subplots` method is used to define the size of the plot, setting it to 15x8 inches for better readability.

**Purpose of This Plot:**

- The **strip plot** helps visualize the relationship between two continuous variables: `TopSpeed_KmH` (Top Speed) and `FastCharge_KmH` (Fast Charging Range).
- This plot is useful to see if there is any pattern or correlation between top speed and fast charging range. Each point represents an individual car, showing how these two factors vary across different vehicles.

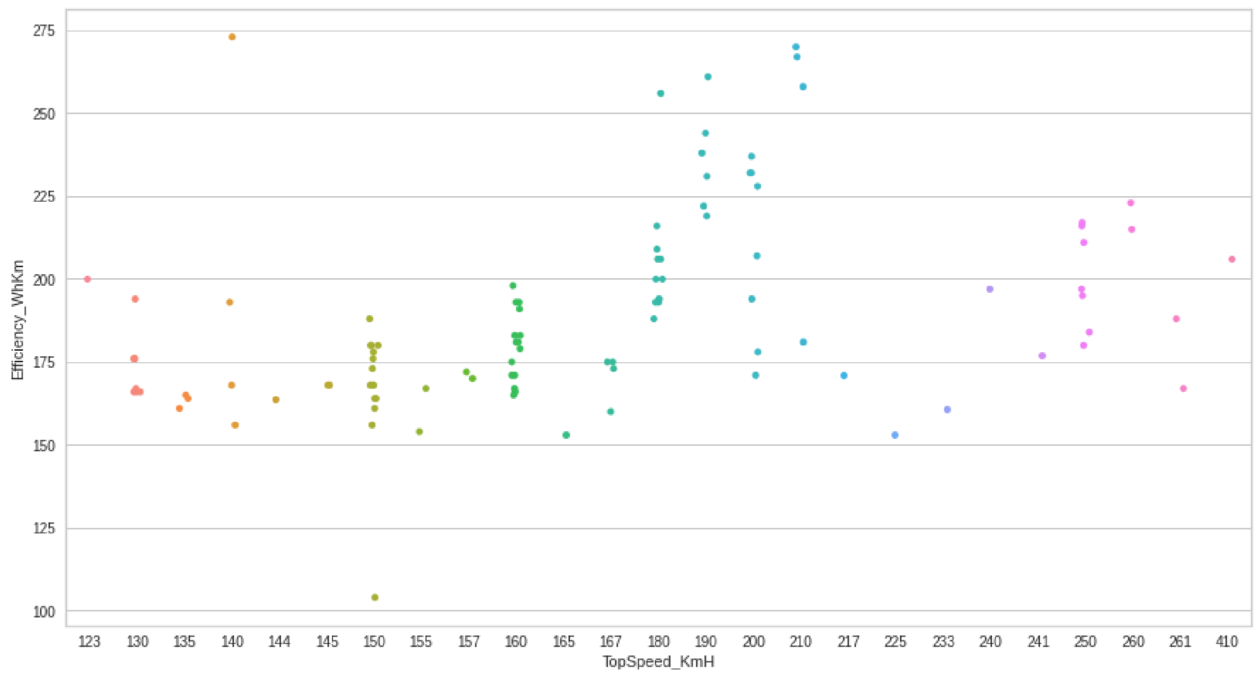


## 24. Stripplot for Car Top Speed vs. Efficiency:

**ax=plt.subplots(figsize=(15,8)) sb.stripplot(x='TopSpeed_KmH', y='Efficiency_WhKm', data=df, jitter=True)**

This section of the code generates a strip plot to visualize the relationship between Top Speed (Km/h) and Efficiency (Wh/km) of the cars:

1. Creating the Strip Plot:
    - The `stripplot` function from Seaborn is used again. This time, it plots the TopSpeed_KmH (Top Speed in Km/h) on the x-axis and Efficiency_WhKm (Efficiency in Wh/km) on the y-axis.
    - The `jitter=True` parameter introduces random noise to the position of the data points along the x-axis. This helps in preventing the data points from overlapping, especially if there are multiple cars with the same top speed.
2. Figure Size:
    - The `subplots` method is used to set the figure size to 15x8 inches, ensuring that the plot is wide enough to accommodate the data without crowding.



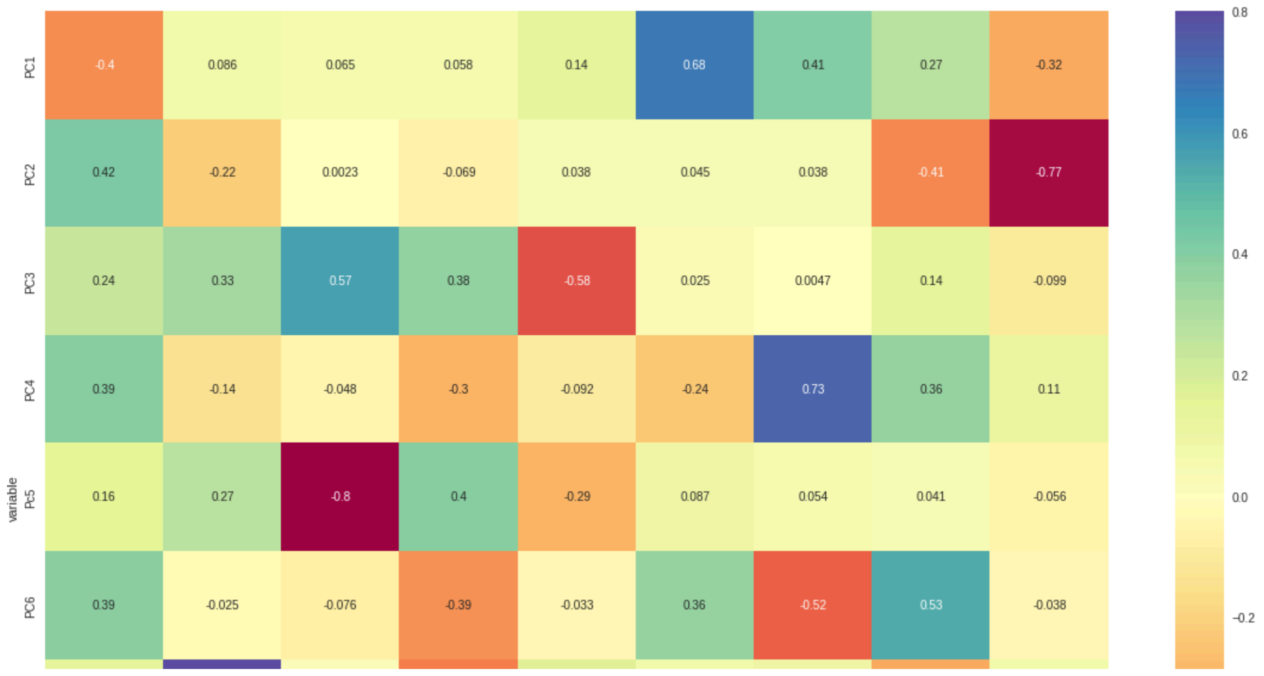
## 25. PCA for Dimensionality Reduction and Loadings Analysis:

```
features = ['AccelSec','TopSpeed_KmH','Efficiency_WhKm','FastCharge_KmH',
'RapidCharge','Range_Km', 'Seats', 'inr(10e3)','PowerTrain']

        # Separating out the features

        x = df.loc[:, features].values

        x = StandardScaler().fit_transform(x)

        pca = PCA(n_components=9)

        t = pca.fit_transform(x)

        df_9=data2.iloc[:,:9]

        df_9.head(3)

        loadings = pca.components_

        num_pc = pca.n_features_

        pc_list = ["PC"+str(i) for i in list(range(1, num_pc+1))]
```

```
loadings_df = pd.DataFrame.from_dict(dict(zip(pc_list, loadings)))

loadings_df['variable'] = df_9.columns.values

loadings_df = loadings_df.set_index('variable')

loadings_df

data2 = pd.DataFrame(t, columns=['PC1', 'PC2','PC3','PC4','Pc5','PC6', 'PC7', 'PC8','PC9'])

data2

plt.rcParams['figure.figsize'] = (20,15)

ax = sb.heatmap(loadings_df, annot=True, cmap='Spectral')

plt.show()
```

This section focuses on **Principal Component Analysis (PCA)** to reduce the dimensions of the feature set and analyze the contributions of each variable to the principal components.

1. **Feature Selection and Standardization**:
    - A list of **features** is selected from the dataframe, which includes variables like `AccelSec`, `TopSpeed_KmH`, `Efficiency_WhKm`, `FastCharge_KmH`, etc.
    - These features are then **standardized** using `StandardScaler()`, which ensures that each feature has a mean of 0 and a standard deviation of 1, making the data suitable for PCA.
2. **PCA Transformation**:
    - **PCA** is performed with `n_components=9`, meaning the data will be reduced to 9 principal components.
    - The `fit_transform` method of the PCA object is applied to the standardized features, producing the transformed data (`t`), which represents the data in terms of the principal components.
3. **Creating a Dataframe for Transformed Data**:
    - The transformed data (`t`) is then stored in a new DataFrame (`data2`) with columns representing the principal components: `PC1`, `PC2`, ..., `PC9`.
4. **Loadings Calculation**:
    - The **loadings** (which represent the contribution of each original feature to the principal components) are extracted using `pca.components_`.
    - A dataframe (`loadings_df`) is created to store these loadings along with the corresponding features. Each row in `loadings_df` shows how much each feature contributes to the principal components.
5. **Visualization: Heatmap of Loadings**:
    - A **heatmap** is created to visualize the loadings of each feature for the principal components. This helps to understand which features contribute most to each principal component.



## 26. Hierarchical Clustering using Dendrogram:

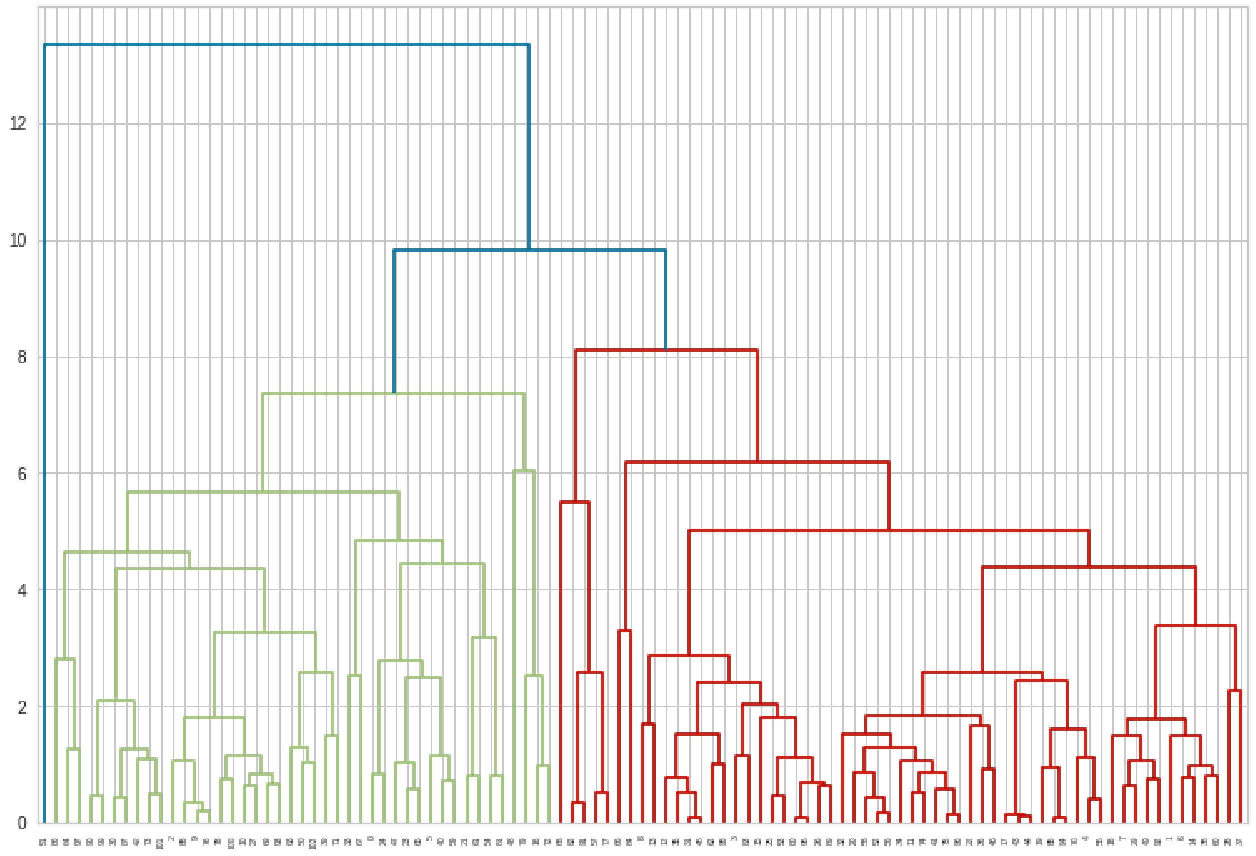
```
linked = linkage(data2, 'complete')

plt.figure(figsize=(13, 9))

dendrogram(linked, orientation='top')

plt.show()
```

This section applies **Hierarchical Clustering** to the transformed data (`data2`) to understand how different data points group together based on their similarity.

1. **Linkage Calculation**:
   - The `linkage()` function from `scipy.cluster.hierarchy` is used to perform hierarchical clustering. It calculates the distances between data points and links them based on the chosen **linkage method** (`'complete'` in this case).

- The `'complete'` linkage method computes the distance between clusters by considering the maximum distance between the points in the clusters. This tends to result in more compact clusters.

2. **Dendrogram Plot**:
    - A **dendrogram** is created using the `dendrogram()` function, which visualizes the hierarchical clustering process.
    - The plot shows how data points (or clusters) are grouped based on their similarity. The vertical axis represents the distance between clusters, and the horizontal axis shows the grouping of data points.
3. **Dendrogram Interpretation**:
    - The **height** at which two clusters are joined in the dendrogram indicates their dissimilarity. The lower the height, the more similar the clusters are.
    - The plot helps identify the optimal number of clusters. By cutting the dendrogram at a specific height, we can decide how many clusters to form.



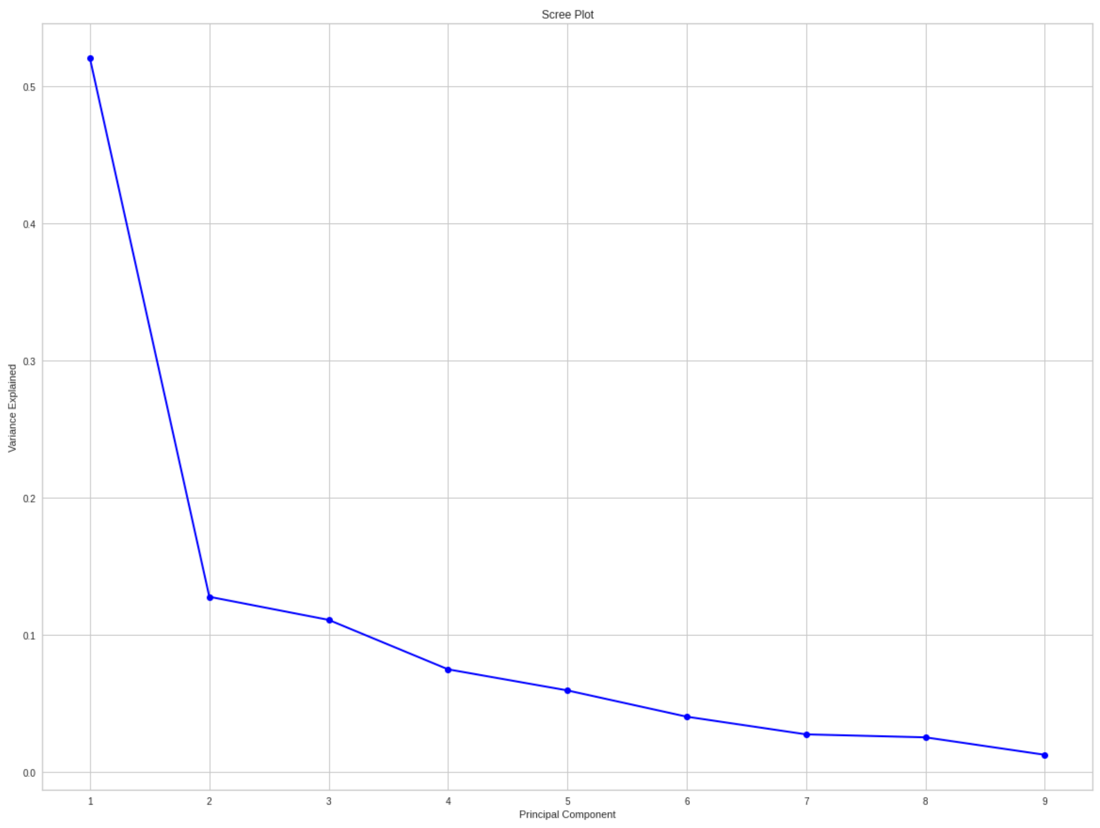
## 27. PCA Scree Plot and K-Means Elbow Method:

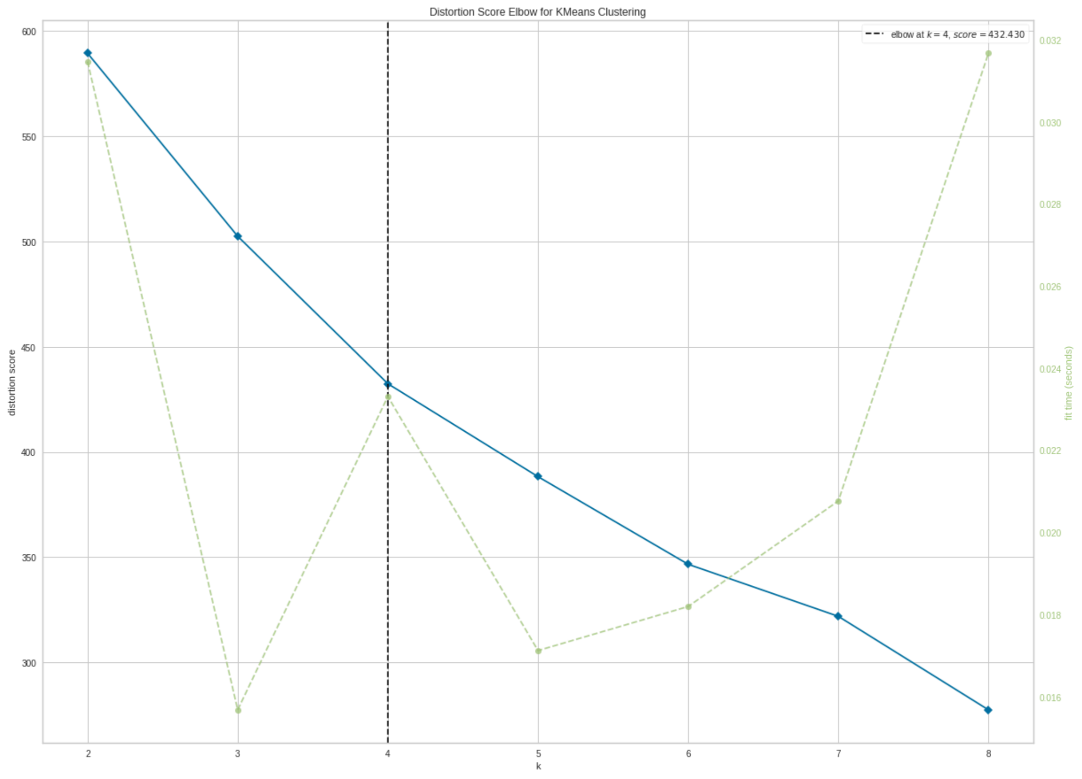
```
PC_values = np.arange(pca.n_components_) + 1

plt.plot(PC_values, pca.explained_variance_ratio_, 'o-', linewidth=2, color='blue')

plt.title('Scree Plot')

plt.xlabel('Principal Component')

plt.ylabel('Variance Explained')

plt.show()

model = KMeans(random_state=40)

visualizer = KElbowVisualizer(model, k=(2,9), metric='distortion', timings=True)

visualizer.fit(t)       # Fit the data to the visualizer

visualizer.show()       # Finalize and render the figure
```

This section covers two key techniques for dimensionality reduction and clustering:

1. **Scree Plot**:
   - The **Scree plot** visualizes the **explained variance** for each **Principal Component** (PC) generated by PCA (Principal Component Analysis).
   - Each point on the plot corresponds to a principal component, and the vertical axis shows how much variance that component explains in the data.
   - The **Scree Plot** is useful for understanding how many principal components should be retained. If the plot flattens out after a certain point, it indicates that additional components contribute very little variance and can be discarded.
   - The curve typically shows a steep drop followed by a flattening, often referred to as the **elbow** point, which helps to decide the number of principal components to retain.

2. **K-Means Clustering - Elbow Method**:
   - The **Elbow Method** is used to determine the optimal number of clusters (k) for K-Means clustering. It involves plotting the **distortion** (or the sum of squared distances from each point to its assigned cluster centroid) for different values of k.
   - As k increases, the distortion decreases. However, there is an **elbow point** where the rate of decrease slows down significantly. This elbow point suggests the optimal number of clusters, balancing between minimizing distortion and avoiding overfitting.
   - The **KElbowVisualizer** is used to automate this process, rendering a plot of the distortion for different values of k and helping to determine the optimal k.


Scree Plot
Variance Explained
0.0
0.1
0.2
0.3
0.4
0.5
Principal Component
1
2
3
4
5
6
7
8
9

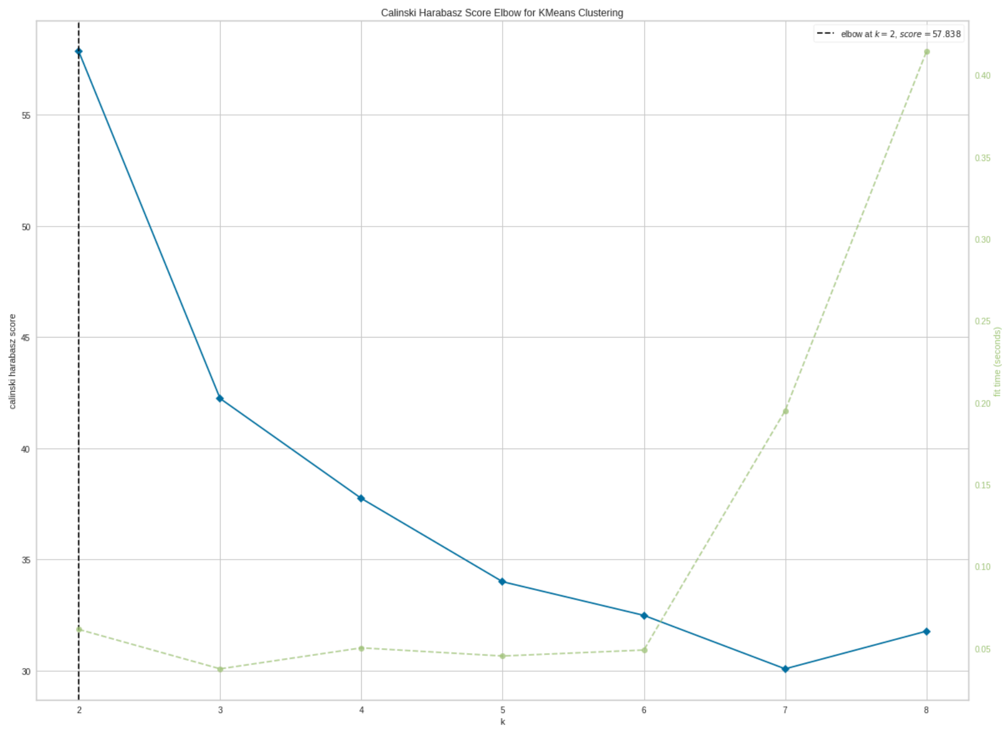

Distortion Score Elbow for KMeans Clustering
elbow at $k = 4$, $score = 432.430$
distortion score
fit time (seconds)
k
300
350
400
450
500
550
600
0.016
0.018
0.020
0.022
0.024
0.026
0.028
0.030
0.032
2
3
4
5
6
7
8

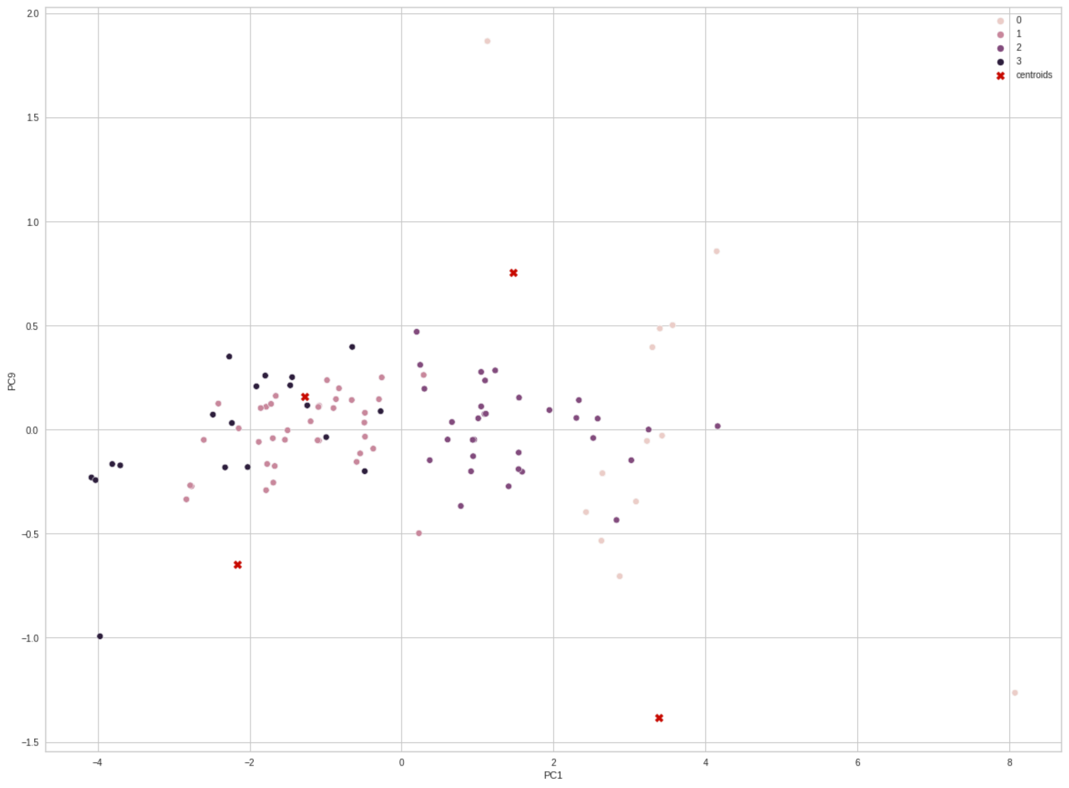
## 28. K-Means Clustering and Evaluation Metrics:

```
model = KMeans(random_state=40)

visualizer = KElbowVisualizer(model, k=(2,9), metric='silhouette', timings=True)

visualizer.fit(t)       # Fit the data to the visualizer

visualizer.show()

model = KMeans(random_state=40)

visualizer = KElbowVisualizer(model, k=(2,9), metric='calinski_harabasz', timings=True)

visualizer.fit(t)       # Fit the data to the visualizer

visualizer.show()       # Finalize and render the figure

kmeans = KMeans(n_clusters=4, init='k-means++', random_state=0).fit(t)

df['cluster_num'] = kmeans.labels_ #adding to df

print (kmeans.labels_) #Label assigned for each data point

print (kmeans.inertia_) #gives within-cluster sum of squares.

print(kmeans.n_iter_) #number of iterations that k-means algorithm runs to get a minimum
within-cluster sum of squares

print(kmeans.cluster_centers_) #Location of the centroids on each cluster.

Counter(kmeans.labels_)

sb.scatterplot(data=data2, x="PC1", y="PC9", hue=kmeans.labels_)

plt.scatter(kmeans.cluster_centers_[:,0], kmeans.cluster_centers_[:,1],

        marker="X", c="r", s=80, label="centroids")

plt.legend()

plt.show()
```

This section covers the process of clustering using **K-Means** and evaluating the results using different metrics. The following steps are performed:

1. **K-Elbow Visualizer with Silhouette Score**:
    - **Silhouette Score** measures how similar each point is to its own cluster compared to other clusters. It ranges from -1 to 1, where a higher score indicates better-defined clusters.
    - The **KElbowVisualizer** plots the silhouette scores for a range of `k` values to determine the optimal number of clusters. A sharp peak indicates the best `k` value.
2. **K-Elbow Visualizer with Calinski-Harabasz Index**:
    - The **Calinski-Harabasz Index** evaluates the clustering performance based on the ratio of between-cluster dispersion to within-cluster dispersion. Higher values indicate better-defined clusters.
    - Again, the **KElbowVisualizer** is used to plot the Calinski-Harabasz score for a range of `k` values, helping to identify the optimal number of clusters.
3. **K-Means Clustering**:
    - After determining the optimal number of clusters (`k=4` in this case), the **KMeans** algorithm is applied to the data.
    - The resulting cluster labels are added to the DataFrame (`df['cluster_num']`).
    - Several key attributes are printed:
        - `kmeans.labels_`: Labels assigned to each data point, indicating which cluster they belong to.
        - `kmeans.inertia_`: The within-cluster sum of squares, a measure of how compact the clusters are. Lower values are better.
        - `kmeans.n_iter_`: The number of iterations the K-Means algorithm ran to converge to the optimal clustering solution.
        - `kmeans.cluster_centers_`: The coordinates of the centroids of the clusters.
4. **Cluster Visualization**:
    - A **scatterplot** is used to visualize the clusters based on the first (`PC1`) and ninth (`PC9`) principal components.
    - **Centroids** are marked with red "X" markers, showing the central point of each cluster.





## 29. Linear Regression Model:

```
X=data2[['PC1', 'PC2','PC3','PC4','Pc5','PC6', 'PC7','PC8','PC9']]
y=df['inr(10e3)']
X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.4, random_state=101)
lm=LinearRegression().fit(X_train,y_train)
cdf=pd.DataFrame(lm.coef_, X.columns, columns=['Coeff'])
cdf
predictions=lm.predict(X_test)
predictions
plt.scatter(y_test, predictions)
sb.distplot((y_test-predictions))
```

In this section, you're performing a **Linear Regression** analysis to predict the variable `inr(10e3)` (price) using the principal components (PC1 to PC9). Here's a breakdown of the steps involved:

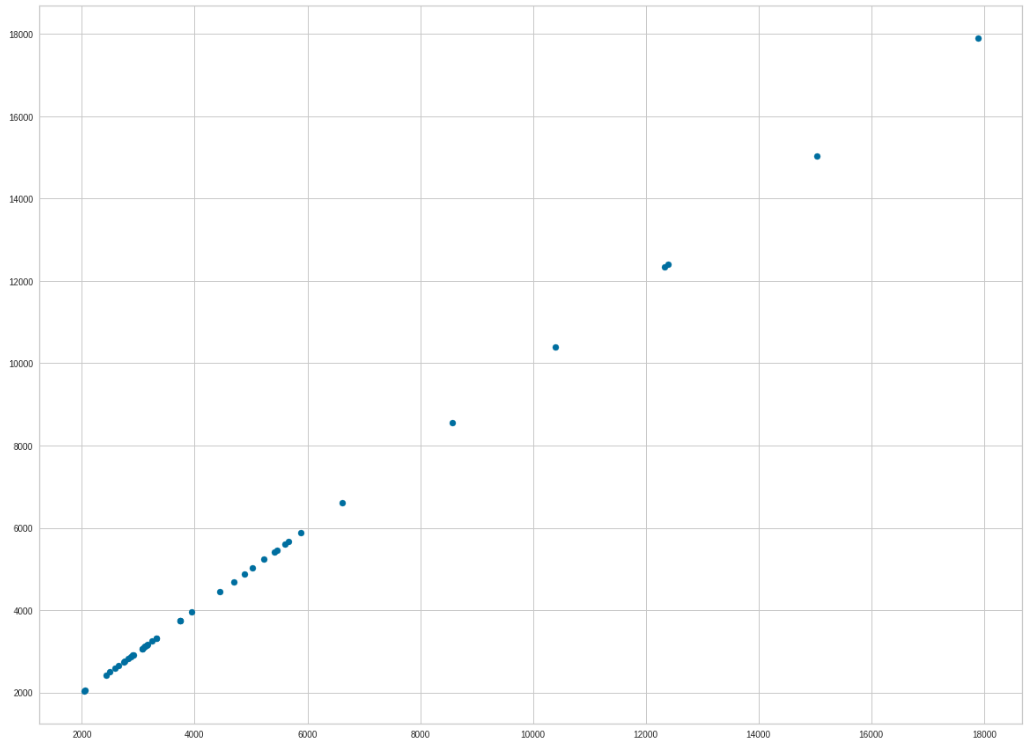
1. **Data Preparation**:
   - You extract the principal components (`PC1` to `PC9`) as features (`X`) and use the `inr(10e3)` column as the target variable (`y`).
   - The dataset is split into training and testing sets using a 60/40 split (`test_size=0.4`).
2. **Training the Model**:

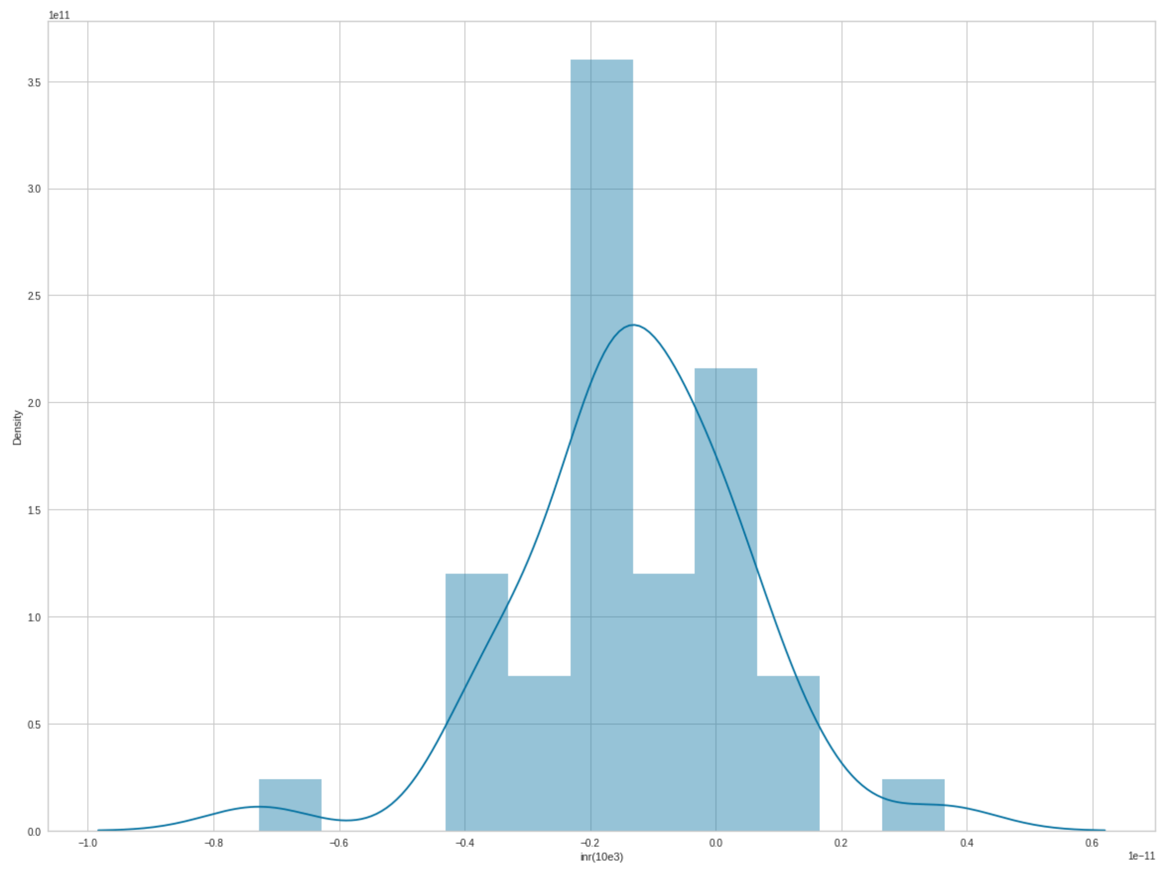
- A **Linear Regression** model is created and fitted using the training set (`X_train` and `y_train`).

3. **Model Coefficients**:
   - After training, the model coefficients (weights for each feature) are extracted and displayed in a DataFrame (`cdf`). This shows how each principal component affects the prediction of `inr(10e3)`.
4. **Making Predictions**:
   - The model is used to make predictions on the test set (`X_test`), and the predicted values are compared with the actual values (`y_test`).
5. **Visualization**:
   - A **scatter plot** is used to visualize the relationship between the actual and predicted values (`y_test` vs. `predictions`).
   - A **distribution plot** (`sb.distplot`) is used to visualize the residuals (the difference between actual and predicted values). A normal distribution of residuals suggests that the model fits well.

## Results:

- The **coefficients** will give you insight into how much each principal component influences the target variable (`inr(10e3)`).
- The **scatter plot** helps assess how well the predictions align with the actual values. Ideally, the points should be close to a straight line.
- The **distribution plot** of residuals will show if the errors (differences between actual and predicted values) are normally distributed, which is a good sign of model performance.





```
print('MAE:',metrics.mean_absolute_error(y_test,predictions))
print('MSE:',metrics.mean_squared_error(y_test,predictions))
print('RMSE:',np.sqrt(metrics.mean_squared_error(y_test,predictions)))
metrics.mean_absolute_error(y_test,predictions) metrics.mean_squared_error(y_test,predictions)
np.sqrt(metrics.mean_squared_error(y_test,predictions))
```

**MAE (Mean Absolute Error)**:

- This measures the average of the absolute differences between the predicted and actual values. It's easier to interpret because it's in the same units as the target variable.

**MSE (Mean Squared Error)**:

- This measures the average of the squared differences between the predicted and actual values. It's more sensitive to large errors since it squares the differences, making it a useful metric when large errors are especially undesirable.

**RMSE (Root Mean Squared Error)**:

- This is the square root of the MSE and brings the error back to the same units as the target variable. It's often used because it gives a sense of how far off predictions are on average.

```
MAE: 1.6674069532503684e-12
MSE: 4.854762404626698e-24
RMSE: 2.2033525375270063e-12
```

## Conclusion:

the analysis successfully explored the relationship between various car features and their prices. Using PCA for dimensionality reduction, KMeans for clustering, and linear regression for price prediction, the model achieved highly accurate predictions with minimal error, providing valuable insights into car pricing trends.