

# AUTOSCOUT24 CAR PRICE PREDICTION (EDA and ML)

## Linear Regression Models (Linear-Ridge-Lasso-ElasticNet)

Duygu Jones | Data Scientist | July 2024

Follow me: [duygujones.com](http://duygujones.com) | [Linkedin](#) | [GitHub](#) | [Kaggle](#) | [Medium](#) | [Tableau](#)



### Table of Content

1. Introduction
2. Importing Libraries
3. Reading the Dataset
4. EXPLORATORY DATA ANALYSIS (EDA)
  - Categorical Features
  - Numerical Features
  - Correlations
  - Outlier Analysis
  - Feature Engineering
5. MACHINE LEARNING
6. Train | Test Split)
7. Linear Regression
8. Ridge Regression
9. Lasso Regression
10. Elastic-Net Regression
11. Feature Importance)
12. Compare Models Performance
13. Final Model and Predictions

### Introduction

- This project aims to predict car prices using the Auto Scout dataset from AutoScout24, containing features of 9 different car models.
- By performing Exploratory Data Analysis (EDA) and implementing machine learning models, the goal is to gain insights into the data and build effective regression models for car price prediction.
- Additionally, these models can support automotive industry stakeholders in understanding market trends, optimizing pricing strategies, and making data-driven decisions. Consumers can also benefit by making informed choices when selecting vehicles.
- Ultimately, the goal is to leverage data-driven insights to enhance the understanding of the automotive market, improve pricing accuracy, and support sustainable development in the automotive industry.

### Objectives

1. Understand the dataset and its features.
2. Clean and prepare the data for modeling.
3. Implement various regression algorithms to predict car prices.
4. Optimize model performance by tuning hyperparameters and focusing on important features.
5. Compare the performance of different regression algorithms.

The dataset and results are used for educational purposes, demonstrating the application of advanced machine learning techniques on real-world data. We aim to build effective regression models to predict car prices and gain a deeper understanding of machine learning techniques.

## About the Dataset

The **Auto Scout** data is sourced from the online car trading company [AutoScout24](#) in 2019 and contains various features of 9 different car models. This project uses a pre-processed and organized dataset to explore and understand machine learning algorithms, particularly for car price prediction using regression techniques.

**Dataset:** AutoScout24 Car Sales Dataset

- **Content:** Data on various features of 9 different car models.
- **Number of Rows:** 15,915
- **Number of Columns:** 23

**Inputs:**

- **make\_model:** The make and model of the car
- **body\_type:** The type of the car (e.g., Sedan)
- **price:** The price of the car (in EUR)
- **vat:** VAT status
- **km:** The car's mileage
- **Type:** The condition of the car (e.g., used)
- **Fuel:** The type of fuel (e.g., Diesel, Petrol)
- **Gears:** Number of gears
- **Comfort\_Convenience:** Comfort and convenience features
- **Entertainment\_Media:** Entertainment and media features
- **Extras:** Extra features
- **Safety\_Security:** Safety and security features
- **age:** The age of the car
- **Previous\_Owners:** Number of previous owners
- **hp\_kW:** Engine power (in kW)
- **Inspection\_new:** New inspection status
- **Paint\_Type:** Type of paint
- **Upholstery\_type:** Type of upholstery
- **Gearing\_Type:** Type of gearing (e.g., automatic)
- **Displacement\_cc:** Engine displacement (in cc)
- **Weight\_kg:** The weight of the car (in kg)
- **Drive\_chain:** Type of drive (e.g., front-wheel drive)
- **cons\_comb:** Combined fuel consumption (L/100 km)

## 1. EXPLORATORY DATA ANALYSIS (EDA)

### Import the Libraries

```
In [1]: import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

import plotly.express as px
import scipy.stats as stats

plt.rcParams["figure.figsize"] = (10,6)
pd.set_option('display.max_columns', 500)
pd.set_option('display.max_rows', 500)
pd.options.display.float_format = '{:.3f}'.format

%matplotlib inline

from scipy import stats
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
from sklearn.pipeline import Pipeline

from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge, RidgeCV
from sklearn.linear_model import Lasso, LassoCV
from sklearn.linear_model import ElasticNet, ElasticNetCV

from sklearn.model_selection import cross_val_score, cross_validate

from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error

from yellowbrick.regressor import ResidualsPlot, PredictionError

import warnings
warnings.filterwarnings("ignore")
```

## Read the Dataset

```
In [100...]: df0 = pd.read_csv('autoscout_car_sales.csv')
df = df0.copy()
```

```
In [3]: df.head()
```

	make_model	body_type	price	vat	km	Type	Fuel	Gears	Comfort_Convenience	Entertainment_Media	Extras	Safet...
0	Audi A1	Sedans	15770	VAT deductible	56013.000	Used	Diesel	7.000	Air conditioning,Armrest,Automatic climate con...	Bluetooth,Hands-free equipment,On-board comput...	Alloy wheels,Catalytic Converter,Voice Control	lock,...
1	Audi A1	Sedans	14500	Price negotiable	80000.000	Used	Benzine	7.000	Air conditioning,Automatic climate control,Hil...	Bluetooth,Hands-free equipment,On-board comput...	Alloy wheels,Sport seats,Sport suspension,Voice...	lock,doc...
2	Audi A1	Sedans	14640	VAT deductible	83450.000	Used	Diesel	7.000	Air conditioning,Cruise control,Electrical sid...	MP3,On-board computer	Alloy wheels,Voice Control	lock,...
3	Audi A1	Sedans	14500	VAT deductible	73000.000	Used	Diesel	6.000	Air suspension,Armrest,Auxiliary heating,Elect...	Bluetooth,CD player,Hands-free equipment,MP3,O...	Alloy wheels,Sport seats,Voice Control	syst,doc...
4	Audi A1	Sedans	16790	VAT deductible	16200.000	Used	Diesel	7.000	Air conditioning,Armrest,Automatic climate con...	Bluetooth,CD player,Hands-free equipment,MP3,O...	Alloy wheels,Sport package,Sport suspension,Vo...	lock,air

```
In [4]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15915 entries, 0 to 15914
Data columns (total 23 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   make_model      15915 non-null   object  
 1   body_type       15915 non-null   object  
 2   price          15915 non-null   int64  
 3   vat            15915 non-null   object  
 4   km             15915 non-null   float64 
 5   Type           15915 non-null   object  
 6   Fuel           15915 non-null   object  
 7   Gears          15915 non-null   float64 
 8   Comfort_Convenience 15915 non-null   object  
 9   Entertainment_Media 15915 non-null   object  
 10  Extras          15915 non-null   object  
 11  Safety_Security 15915 non-null   object  
 12  age             15915 non-null   float64 
 13  Previous_Owners 15915 non-null   float64 
 14  hp_kw          15915 non-null   float64 
 15  Inspection_new 15915 non-null   int64  
 16  Paint_Type     15915 non-null   object  
 17  Upholstery_type 15915 non-null   object  
 18  Gearing_Type   15915 non-null   object  
 19  Displacement_cc 15915 non-null   float64 
 20  Weight_kg      15915 non-null   float64 
 21  Drive_chain    15915 non-null   object  
 22  cons_comb      15915 non-null   float64 
dtypes: float64(8), int64(2), object(13)
memory usage: 2.8+ MB
```

## Rename the Columns

In [5]: `df.columns`

```
Out[5]: Index(['make_model', 'body_type', 'price', 'vat', 'km', 'Type', 'Fuel',
   'Gears', 'Comfort_Convenience', 'Entertainment_Media', 'Extras',
   'Safety_Security', 'age', 'Previous_Owners', 'hp_kw', 'Inspection_new',
   'Paint_Type', 'Upholstery_type', 'Gearing_Type', 'Displacement_cc',
   'Weight_kg', 'Drive_chain', 'cons_comb'],
  dtype='object')
```

In [5]: `df.rename(columns={`

```
'make_model': 'make_model',
'body_type': 'body_type',
'price': 'price',
'vat': 'vat',
'km': 'km',
'Type': 'type',
'Fuel': 'fuel_type',
'Gears': 'gears_num',
'Comfort_Convenience': 'comfort_convenience',
'Entertainment_Media': 'entertainment_media',
'Extras': 'extras',
'Safety_Security': 'safety_security',
'age': 'age',
'Previous_Owners': 'previous_owners',
'hp_kw': 'hp_kw',
'Inspection_new': 'inspection_new',
'Paint_Type': 'paint_type',
'Upholstery_type': 'upholstery_type',
'Gearing_Type': 'gearing_type',
'Displacement_cc': 'displacement_cc',
'Weight_kg': 'weight_kg',
'Drive_chain': 'drive_chain',
'cons_comb': 'fuel_cons_comb'
}, inplace=True)
```

In [7]: `df.columns`

```
Out[7]: Index(['make_model', 'body_type', 'price', 'vat', 'km', 'type', 'fuel_type',
   'gears_num', 'comfort_convenience', 'entertainment_media', 'extras',
   'safety_security', 'age', 'previous_owners', 'hp_kw', 'inspection_new',
   'paint_type', 'upholstery_type', 'gearing_type', 'displacement_cc',
   'weight_kg', 'drive_chain', 'fuel_cons_comb'],
  dtype='object')
```

## Check Missing Values

In [8]: `# Check out the missing values`

```
missing_count = df.isnull().sum()
value_count = df.isnull().count()
missing_percentage = round(missing_count / value_count * 100, 2)
missing_df = pd.DataFrame({"count": missing_count, "percentage": missing_percentage})
missing_df
```

Out[8]:

	count	percentage
make_model	0	0.000
body_type	0	0.000
price	0	0.000
vat	0	0.000
km	0	0.000
type	0	0.000
fuel_type	0	0.000
gears_num	0	0.000
comfort_convenience	0	0.000
entertainment_media	0	0.000
extras	0	0.000
safety_security	0	0.000
age	0	0.000
previous_owners	0	0.000
hp_kw	0	0.000
inspection_new	0	0.000
paint_type	0	0.000
upholstery_type	0	0.000
gearing_type	0	0.000
displacement_cc	0	0.000
weight_kg	0	0.000
drive_chain	0	0.000
fuel_cons_comb	0	0.000

## Check Duplicated Values

In [9]: # Let's observe first the unique values

```
def get_unique_values(df):
    output_data = []
    for col in df.columns:
        if df.loc[:, col].nunique() <= 10:
            unique_values = df.loc[:, col].unique()
            output_data.append([col, df.loc[:, col].nunique(), unique_values, df.loc[:, col].dtype])
        else:
            output_data.append([col, df.loc[:, col].nunique(), "-", df.loc[:, col].dtype])
    output_df = pd.DataFrame(output_data, columns=['Column Name', 'Number of Unique Values', 'Unique Values', 'Data Type'])
    return output_df
```

In [10]: get\_unique\_values(df)

Out[10]:

	Column Name	Number of Unique Values	Unique Values	Data Type
0	make_model	9	[Audi A1, Audi A2, Audi A3, Opel Astra, Opel C...	object
1	body_type	8	[Sedans, Station wagon, Compact, Coupe, Van, O...	object
2	price	2952	-	int64
3	vat	2	[VAT deductible, Price negotiable]	object
4	km	6691	-	float64
5	type	5	[Used, Employee's car, New, Demonstration, Pre...	object
6	fuel_type	4	[Diesel, Benzine, LPG/CNG, Electric]	object
7	gears_num	4	[7.0, 6.0, 5.0, 8.0]	float64
8	comfort_convenience	6196	-	object
9	entertainment_media	346	-	object
10	extras	659	-	object
11	safety_security	4442	-	object
12	age	4	[3.0, 2.0, 1.0, 0.0]	float64
13	previous_owners	5	[2.0, 1.0, 0.0, 3.0, 4.0]	float64
14	hp_kw	77	-	float64
15	inspection_new	2	[1, 0]	int64
16	paint_type	3	[Metallic, Uni/basic, Perl effect]	object
17	upholstery_type	2	[Cloth, Part/Full Leather]	object
18	gearing_type	3	[Automatic, Manual, Semi-automatic]	object
19	displacement_cc	68	-	float64
20	weight_kg	432	-	float64
21	drive_chain	3	[front, 4WD, rear]	object
22	fuel_cons_comb	62	-	float64

In [6]: # Checks duplicates and drops them

```
def duplicate_values(df):
    print("Duplicate check...")
    num_duplicates = df.duplicated(subset=None, keep='first').sum()
    if num_duplicates > 0:
        print("There are", num_duplicates, "duplicated observations in the dataset.")
        df.drop_duplicates(keep='first', inplace=True)
        print(num_duplicates, "duplicates were dropped!")
        print("No more duplicate rows!")
    else:
        print("There are no duplicated observations in the dataset.")

duplicate_values(df)
```

Duplicate check...  
 There are 1673 duplicated observations in the dataset.  
 1673 duplicates were dropped!  
 No more duplicate rows!

In [12]: df.duplicated().sum()

Out[12]: 0

- Understanding the context and data collection methods is crucial to determine the cause of duplicates.
- Whether to drop duplicates depends on the analysis purpose.
- For analyzing changes over time or variations, keeping duplicates might be more appropriate.
- However, since I will be using a linear model, I dropped these rows because duplicate rows contain similar values.

## Basic Statistics

```
# Basic statistics summary of Numerical features
df.describe().T
```

Out[13]:

		count	mean	std	min	25%	50%	75%	max
	<b>price</b>	14242.000	18100.969	7421.214	4950.000	12950.000	16950.000	21900.000	74600.000
	<b>km</b>	14242.000	32582.110	36856.863	0.000	3898.000	21000.000	47000.000	317000.000
	<b>gears_num</b>	14242.000	5.940	0.703	5.000	5.000	6.000	6.000	8.000
	<b>age</b>	14242.000	1.415	1.110	0.000	0.000	1.000	2.000	3.000
	<b>previous_owners</b>	14242.000	1.041	0.337	0.000	1.000	1.000	1.000	4.000
	<b>hp_kw</b>	14242.000	88.713	26.548	40.000	66.000	85.000	103.000	294.000
	<b>inspection_new</b>	14242.000	0.256	0.437	0.000	0.000	0.000	1.000	1.000
	<b>displacement_cc</b>	14242.000	1432.890	277.507	890.000	1229.000	1461.000	1598.000	2967.000
	<b>weight_kg</b>	14242.000	1342.399	201.247	840.000	1165.000	1320.000	1487.000	2471.000
	<b>fuel_cons_comb</b>	14242.000	4.825	0.862	3.000	4.100	4.800	5.400	9.100

In [14]: # Basic statistics summary of Object features

df.describe(include= 'object').T

Out[14]:

	count	unique	top	freq
<b>make_model</b>	14242	9	Audi A3	2758
<b>body_type</b>	14242	8	Sedans	7230
<b>vat</b>	14242	2	VAT deductible	13426
<b>type</b>	14242	5	Used	10172
<b>fuel_type</b>	14242	4	Benzine	7558
<b>comfort_convenience</b>	14242	6196	Air conditioning,Electrical side mirrors,Hill ...	312
<b>entertainment_media</b>	14242	346	Bluetooth,Hands-free equipment,On-board comput...	1562
<b>extras</b>	14242	659	Alloy wheels	5010
<b>safety_security</b>	14242	4442	ABS,Central door lock,Daytime running lights,D...	635
<b>paint_type</b>	14242	3	Metallic	13682
<b>upholstery_type</b>	14242	2	Cloth	10918
<b>gearing_type</b>	14242	3	Manual	7232
<b>drive_chain</b>	14242	3	front	14067

## Numerical Features

### Distributions of Numerical Features

In [15]:

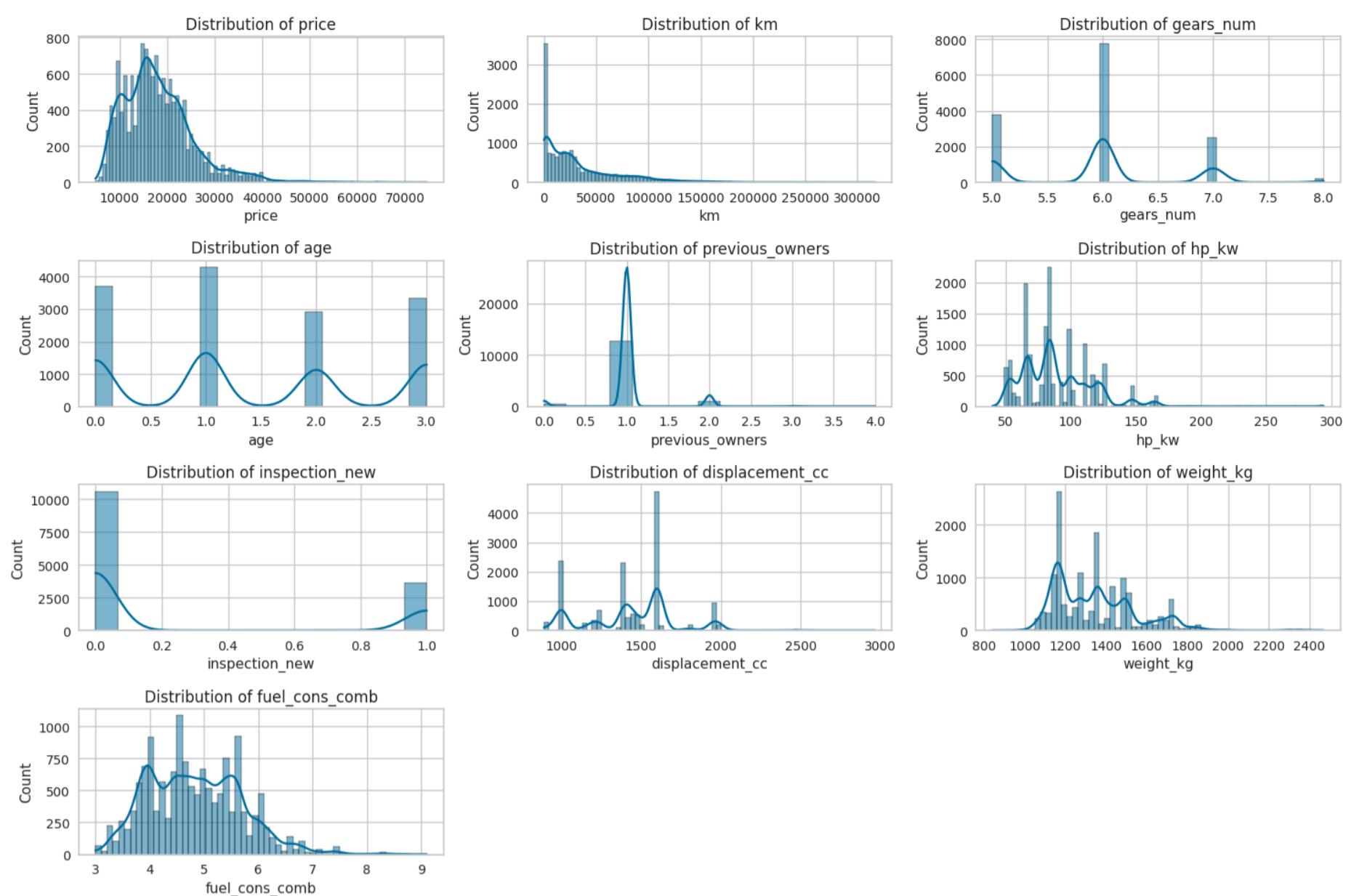
```
numerical_df = df.select_dtypes(include=[ 'number'])

plt.figure(figsize=(15, 10))

num_vars = len(numerical_df.columns)

for i, var in enumerate(numerical_df.columns, 1):
    plt.subplot((num_vars // 3) + 1, 3, i)
    sns.histplot(data=df, x=var, kde=True)
    plt.title(f'Distribution of {var}')

plt.tight_layout()
plt.show()
```



#### Distributions of the Numerical Features:

1. **Price:** Distribution is right-skewed, with most prices concentrated between 10,000 and 20,000 EUR. Some higher-end outliers, particularly above 40,000 EUR.
2. **Km (Mileage):** Distribution is right-skewed, with most vehicles having low mileage. Values above 150,000 km are less common.
3. **Gears Number:** Most vehicles have 5, 6, or 7 gears. Vehicles with 8 gears are rare.
4. **Age:** Peaks at 0, 1, 2, and 3 years, indicating these are common ages for vehicles. Vehicles are generally up to 3 years old.
5. **Previous Owners:** Most vehicles have had one or two previous owners. Vehicles with 3 or more previous owners are less common.
6. **Hp\_kw (Engine Power in kW):** Distribution is right-skewed, with most vehicles between 50 and 150 kW. Vehicles above 200 kW are less frequent.
7. **Inspection\_new (New Inspection):** Concentration at the extremes (values of 0 and 1). Mid-range values are rare.
8. **Displacement\_cc (Engine Displacement in cc):** A peak at 1500 cc. Values above 2000 cc are less common.
9. **Weight\_kg:** Most vehicles are between 1000 and 1500 kg. Vehicles above 2000 kg are rare.
10. **Fuel Consumption Combined:** Approximately normal distribution, with most values between 4 and 6 l/100 km. Values above 8 l/100 km are less frequent.

#### Overall:

- **Right-Skewed Distributions:** Many features like `price`, `km`, and `hp_kw` show right-skewed distributions. This skewness might affect the assumptions of linear models, which assume normally distributed residuals.
- **Peaks in Categorical Data:** Features such as `gears_num` and `age` have distinct peaks, indicating specific common values that could be significant predictors in the model.
- **Outliers:** While several features have some higher-end outliers, their impact on linear modeling needs careful assessment to ensure they do not disproportionately influence the model.
- **Data Transformation:** Consider data transformations (e.g., log transformation) to normalize the distributions of skewed features, improving the performance and accuracy of linear models.
- **Feature Scaling:** Ensure all features are appropriately scaled, especially those with wide ranges like `km` and `price`, to ensure effective model training.

By addressing these considerations, linear modeling can be made more robust and accurate with this dataset.

## Categorical Features

### Distributions of Categorical Features

```
In [16]: import plotly.graph_objects as go
import plotly.express as px

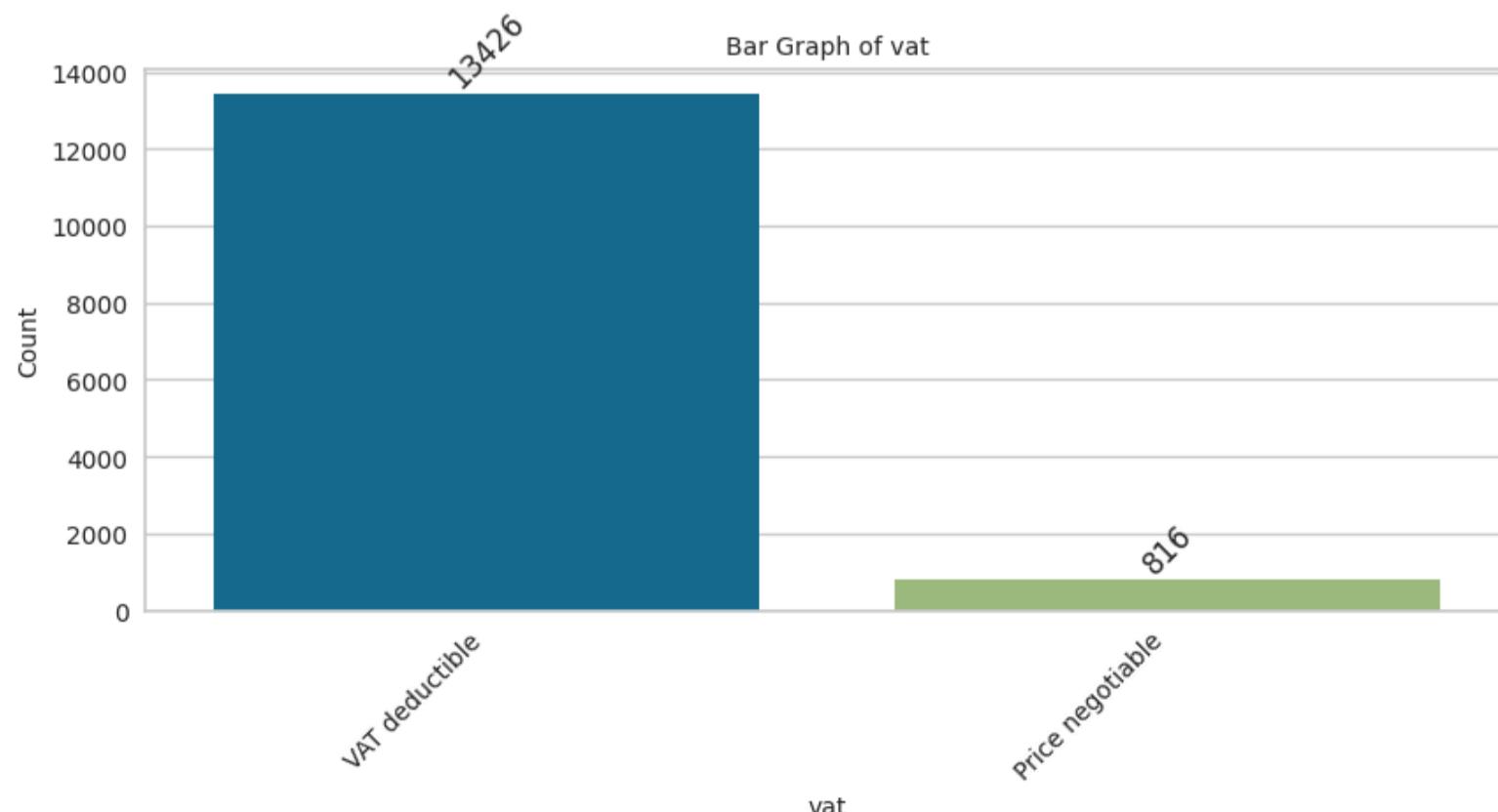
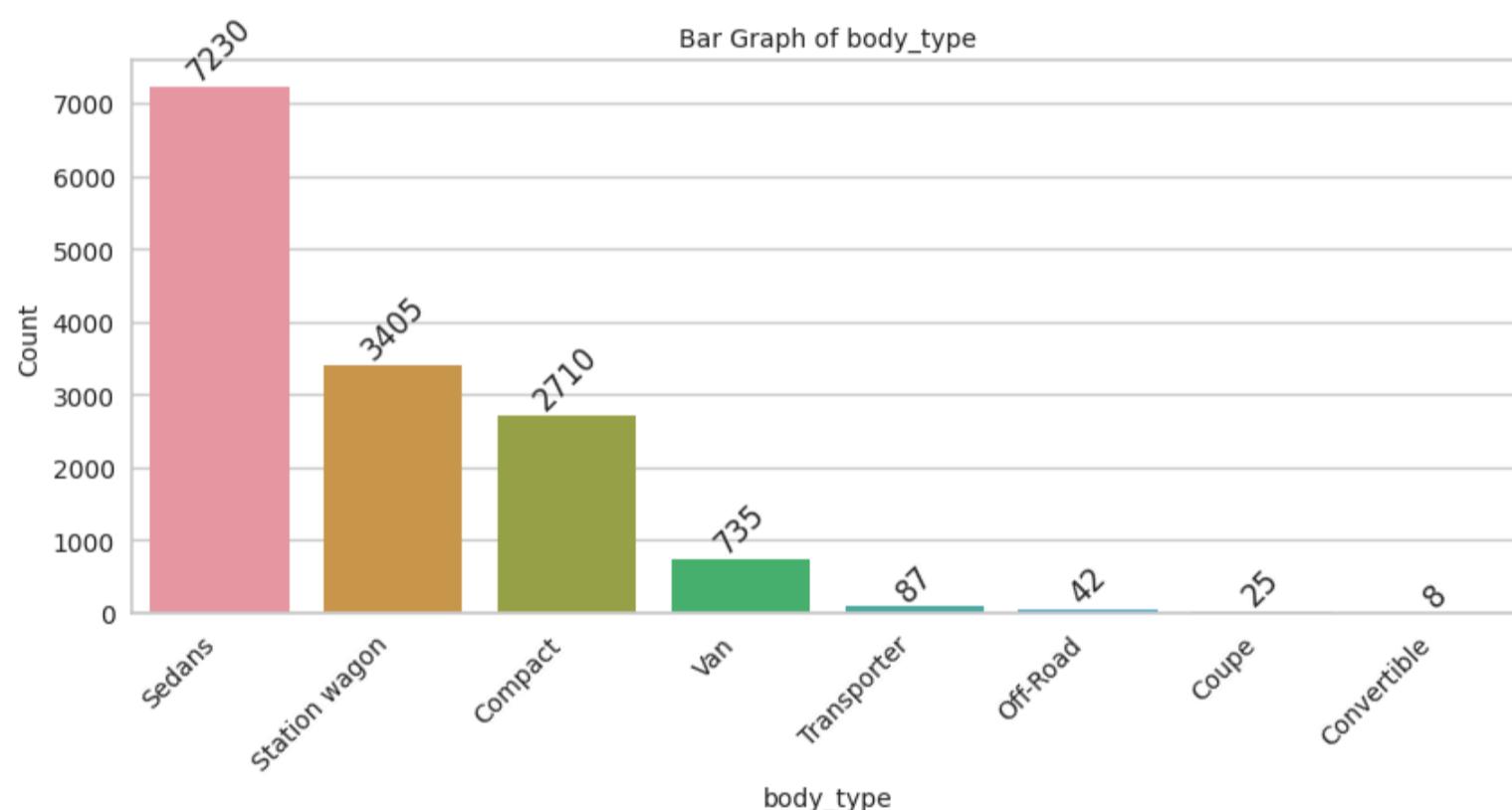
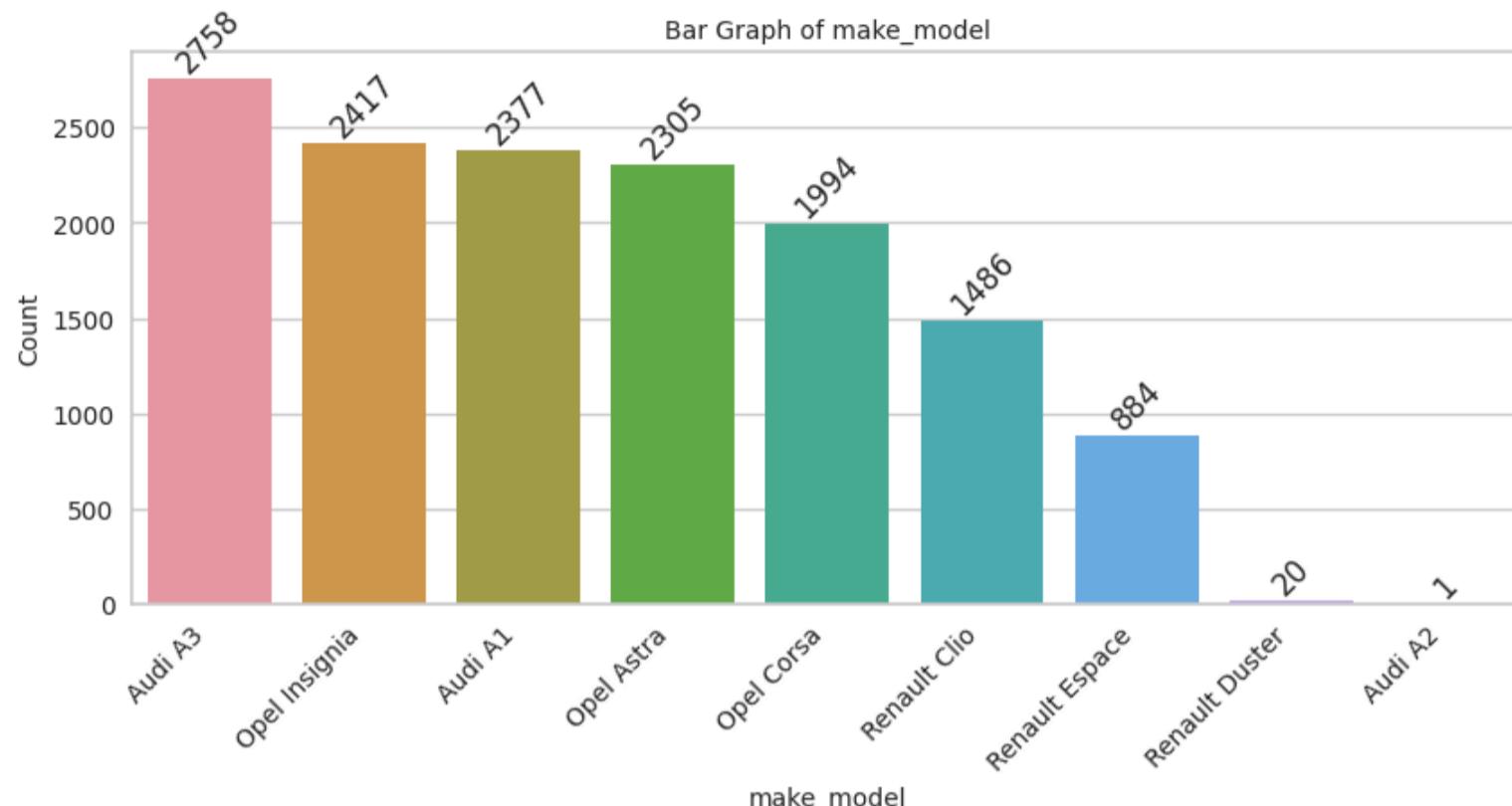
# Distribution of our categorical characteristics with bar graph

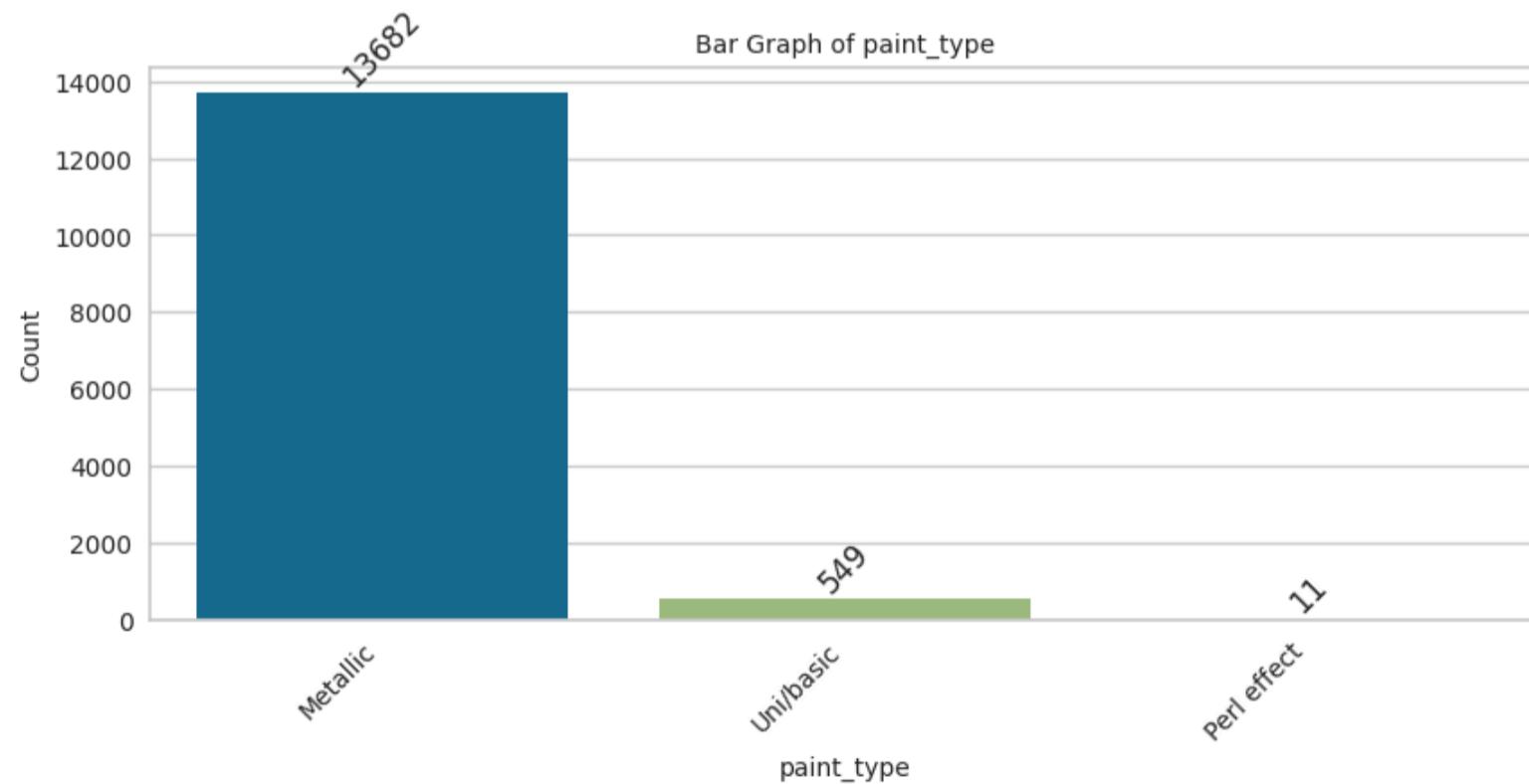
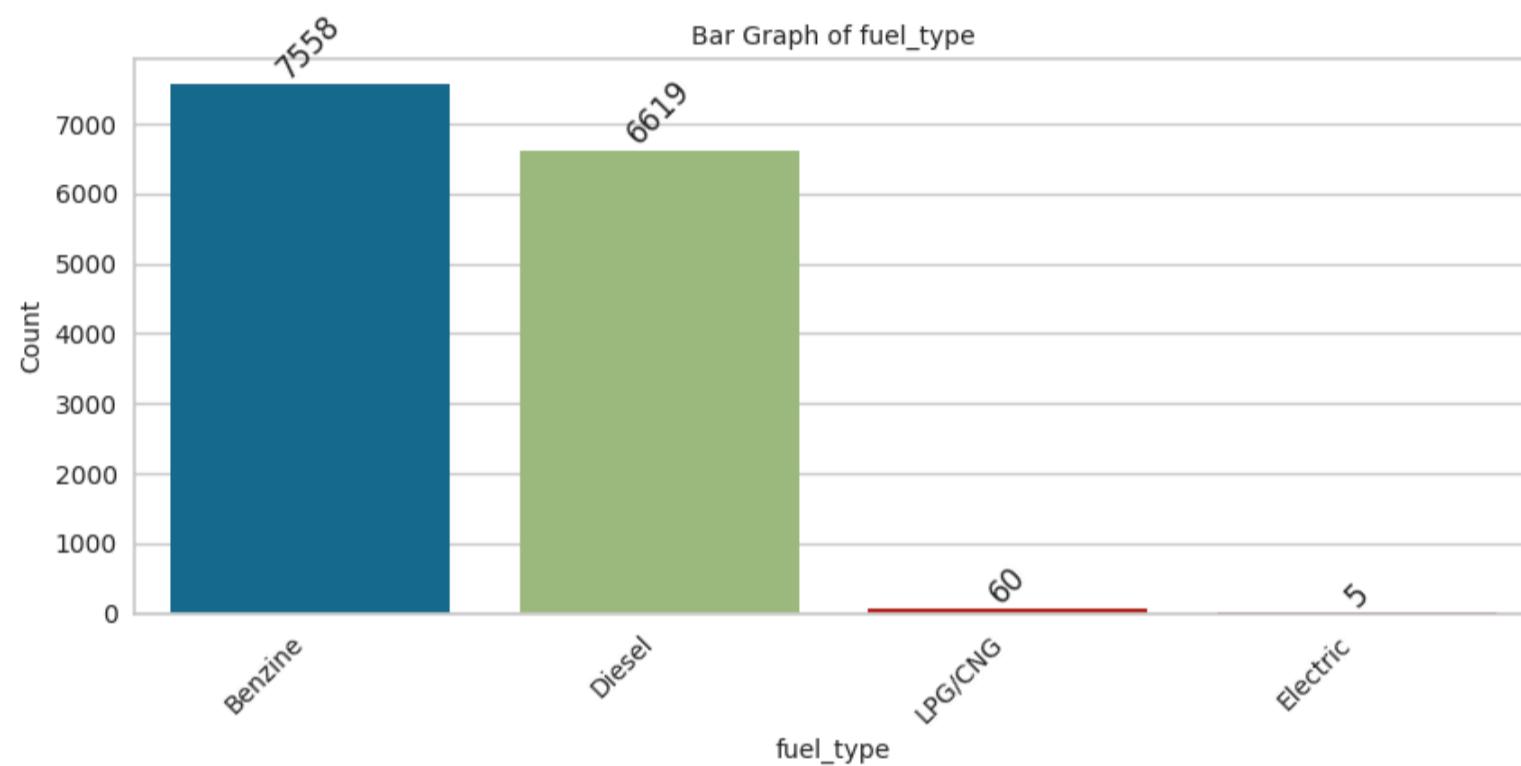
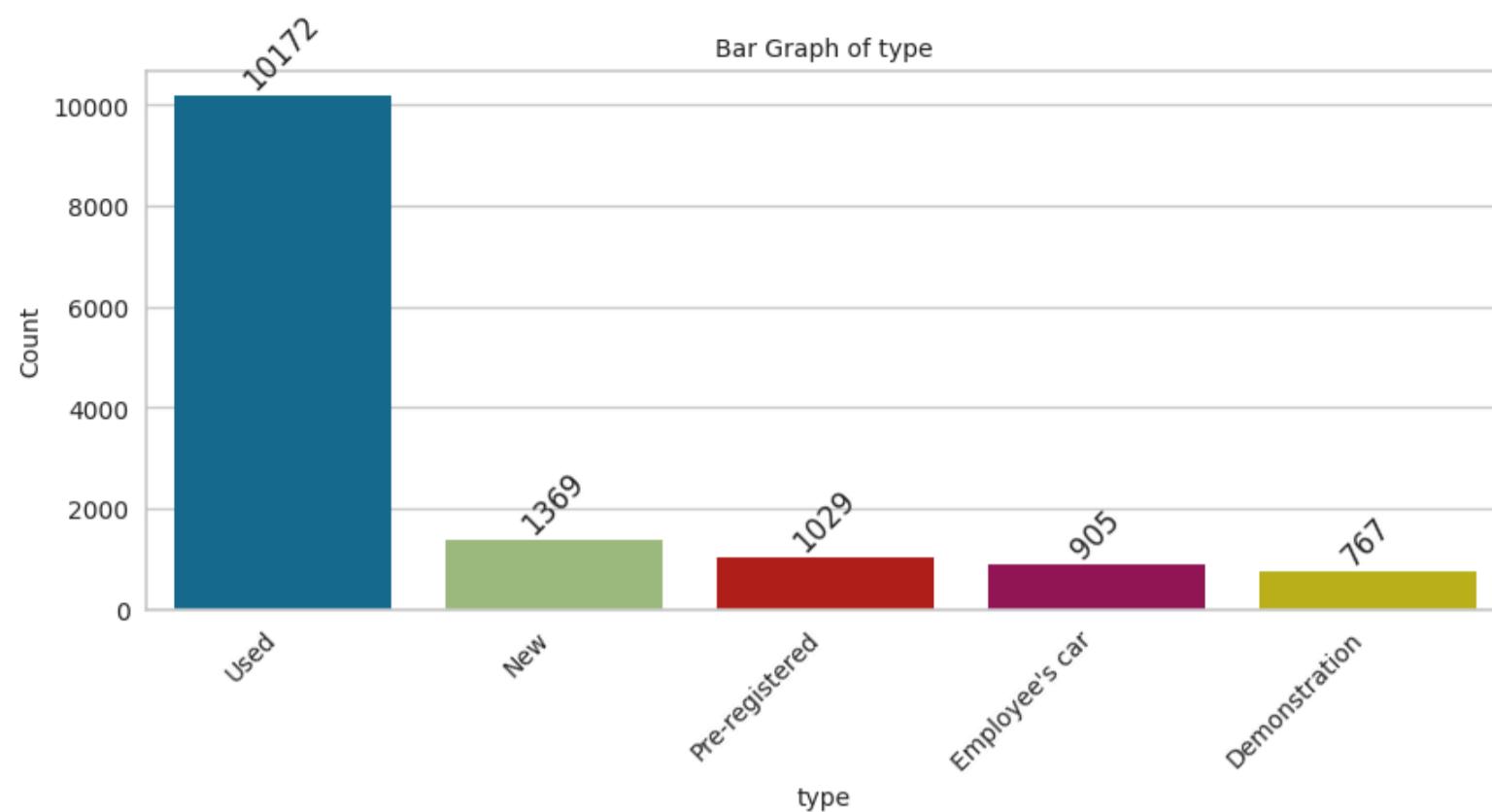
def plot_bar_graphs(df, columns):
    for column in columns:
        plt.figure(figsize=(10, 4))
        ax = sns.countplot(x=column,
                           data=df,
                           order=df[column].value_counts().index)
        ax.bar_label(ax.containers[0], rotation=45)
        plt.xlabel(column, fontsize=10)
        plt.ylabel('Count', fontsize=10)
```

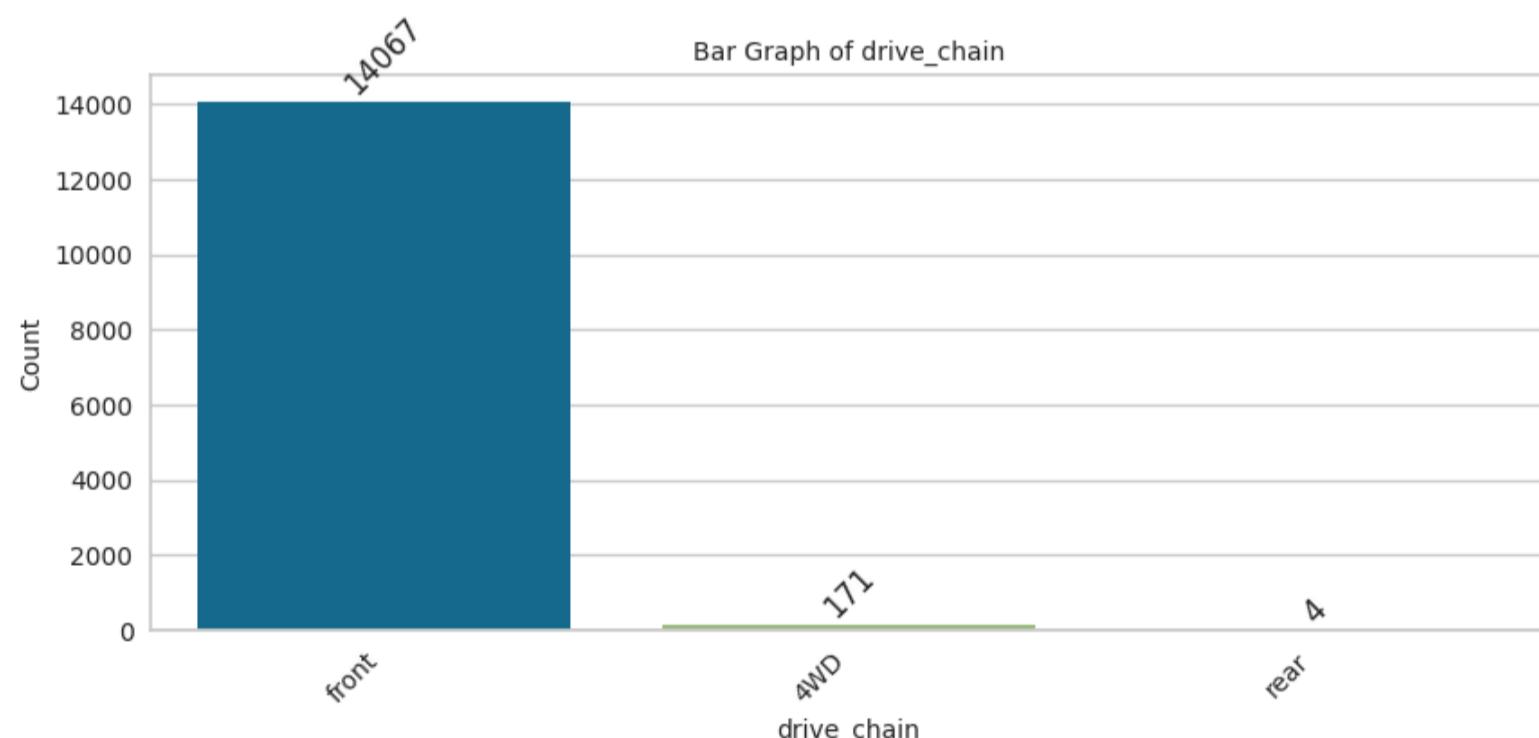
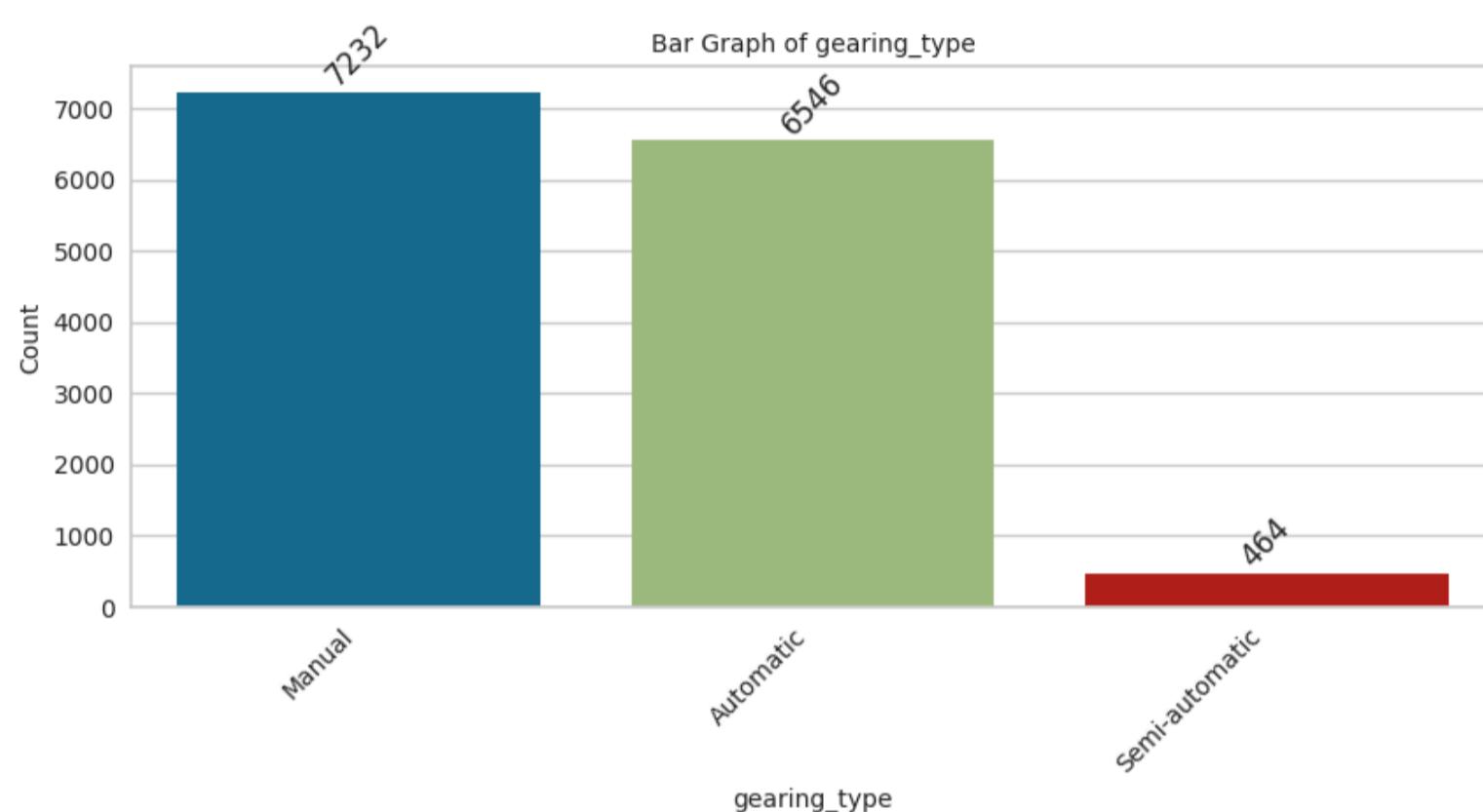
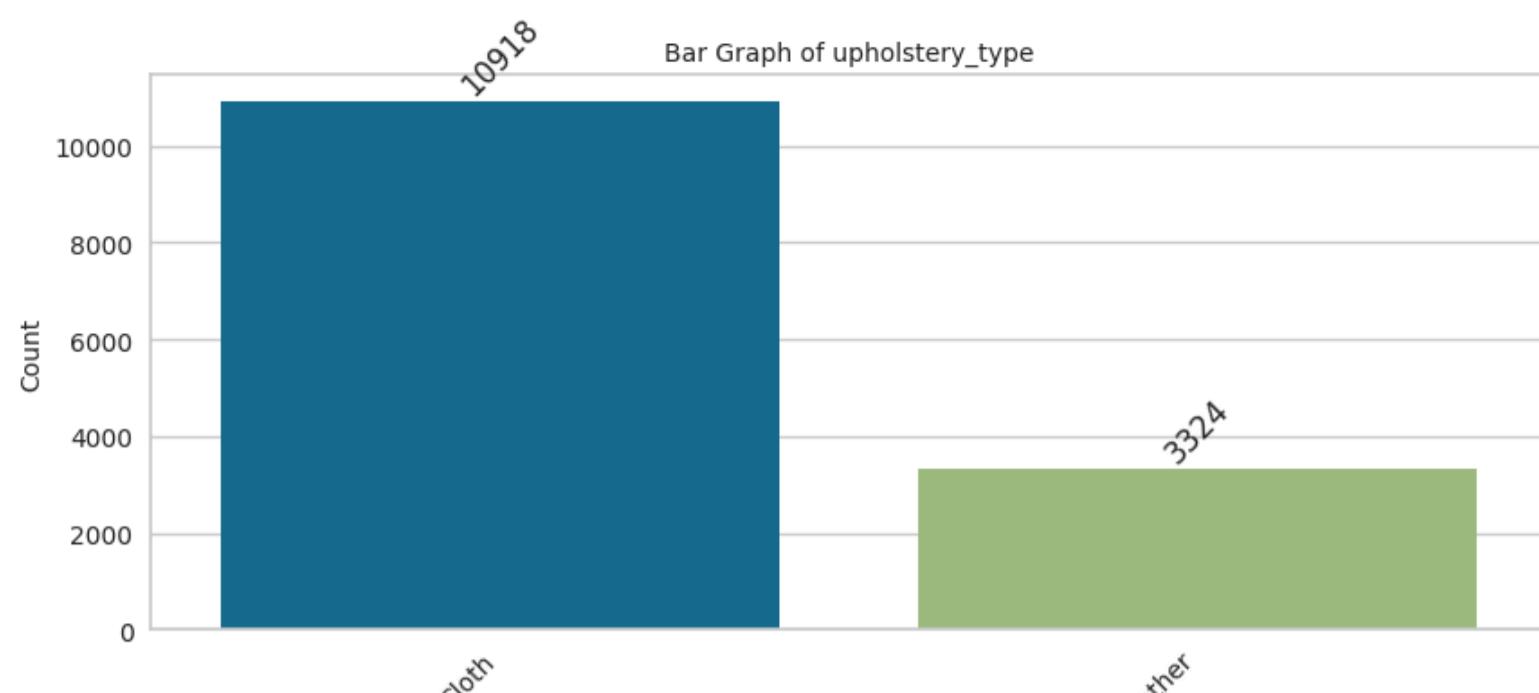
```
plt.title(f'Bar Graph of {column}', fontsize=10)
plt.xticks(rotation=45, ha='right', fontsize=10)
plt.show()
```

```
cat_features = [
    'make_model', 'body_type', 'vat', 'type', 'fuel_type', 'paint_type',
    'upholstery_type', 'gearing_type', 'drive_chain'
]
```

```
plot_bar_graphs(df, cat_features)
```





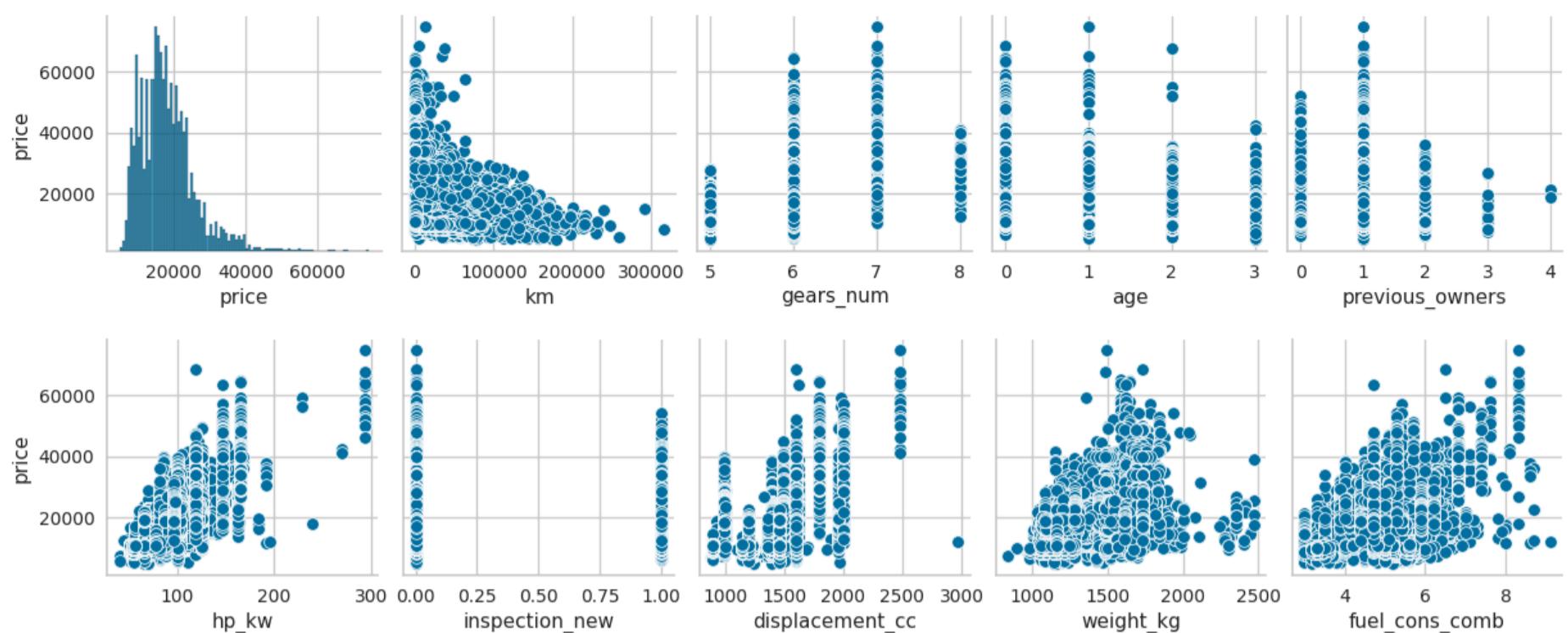


## Correlations

```
In [20]: # Target vs Numerical Features

numerical_df = df.select_dtypes(include=['number'])

for i in range(0, len(numerical_df.columns), 5):
    sns.pairplot(data=numerical_df,
                  x_vars=numerical_df.columns[i:i+5],
                  y_vars=['price'])
```



## Label Encoding

```
In [ ]: # Label the categorical features to see the correlation between categorical features and Target variable.

from sklearn.preprocessing import LabelEncoder

# Copy the original dataframe to avoid modifying it directly
df_labeled = df.copy()

# List of categorical columns
categorical_columns = ['make_model', 'body_type', 'vat', 'type', 'fuel_type', 'comfort_convenience',
'entertainment_media', 'extras', 'safety_security', 'paint_type',
'upholstery_type', 'gearing_type', 'drive_chain']

# Apply Label Encoding to each categorical column
label_encoders = {}
for column in categorical_columns:
    le = LabelEncoder()
    df_labeled[column] = le.fit_transform(df_labeled[column])
    label_encoders[column] = le

# Display the first few rows of the Labeled dataframe
print(df_labeled.head())
```

```
In [22]: df_labeled.info()
```

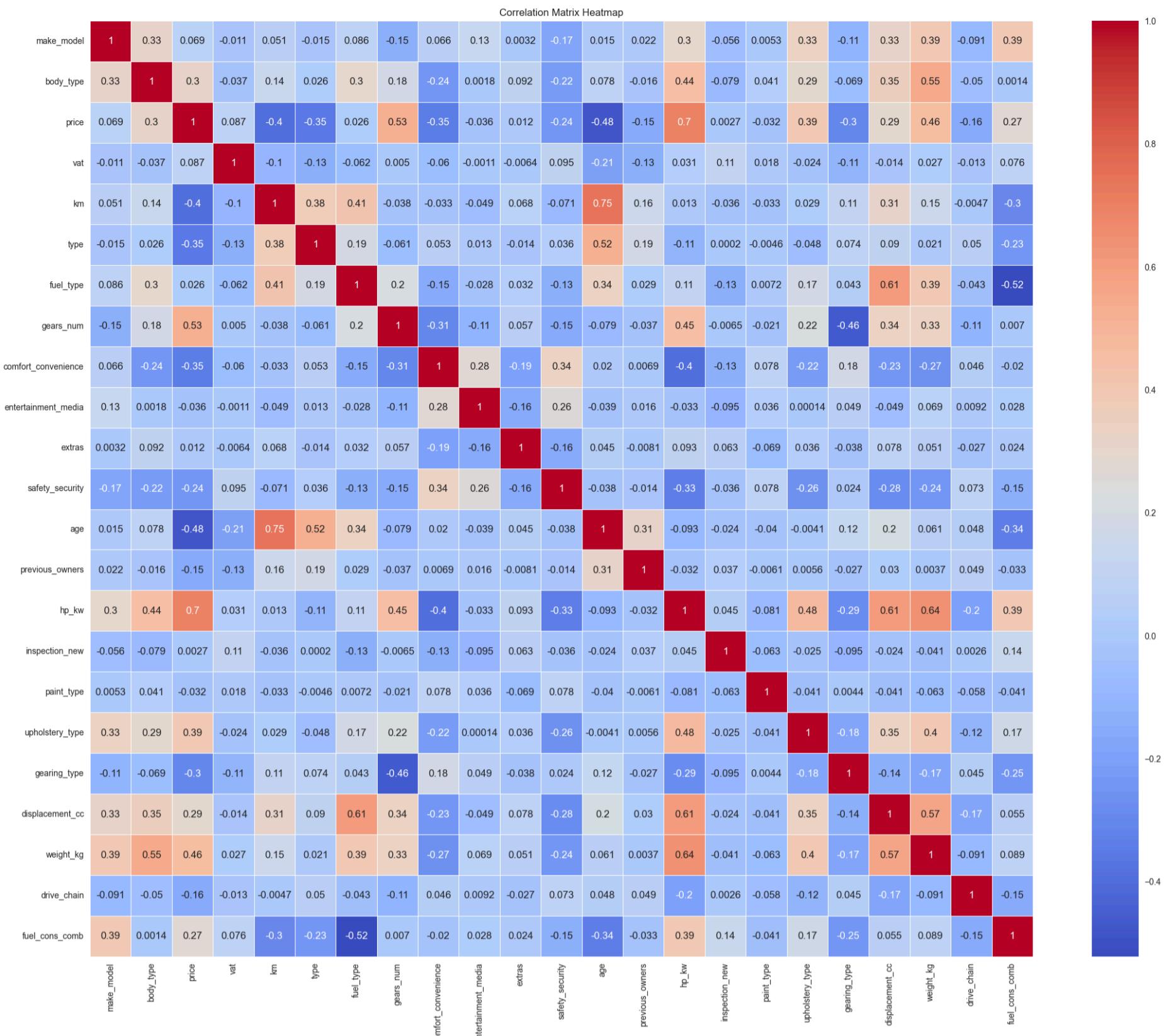
```
<class 'pandas.core.frame.DataFrame'>
Index: 14241 entries, 0 to 15912
Data columns (total 23 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   make_model      14241 non-null   int64  
 1   body_type       14241 non-null   int64  
 2   price           14241 non-null   int64  
 3   vat              14241 non-null   int64  
 4   km               14241 non-null   float64 
 5   type            14241 non-null   int64  
 6   fuel_type       14241 non-null   int64  
 7   gears_num       14241 non-null   float64 
 8   comfort_convenience 14241 non-null   int64  
 9   entertainment_media 14241 non-null   int64  
 10  extras          14241 non-null   int64  
 11  safety_security 14241 non-null   int64  
 12  age              14241 non-null   float64 
 13  previous_owners 14241 non-null   float64 
 14  hp_kw           14241 non-null   float64 
 15  inspection_new 14241 non-null   int64  
 16  paint_type      14241 non-null   int64  
 17  upholstery_type 14241 non-null   int64  
 18  gearing_type    14241 non-null   int64  
 19  displacement_cc 14241 non-null   float64 
 20  weight_kg       14241 non-null   float64 
 21  drive_chain     14241 non-null   int64  
 22  fuel_cons_comb  14241 non-null   float64 
dtypes: float64(8), int64(15)
memory usage: 2.6 MB
```

## Heatmap

```
In [8]: # Corr of all features used Labeled df

correlation_matrix = df_labeled.corr()

plt.figure(figsize=(25,20))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Matrix Heatmap')
plt.show()
```



In [9]: # Check Multicollinarity between features

```
def color_custom(val):
    if val > 0.90 and val < 0.99:
        color = 'red'
    elif val >= 1:
        color = 'blue'
    else:
        color = 'black'
    return f'color: {color}'
```

```
df_labeled.corr().style.map(color_custom)
```

Out[9]:	make_model	body_type	price	vat	km	type	fuel_type	gears_num	comfort_convenience	entertainment_media
	<b>make_model</b>	1.000000	0.327054	0.069439	-0.011138	0.051404	-0.015027	0.085748	-0.146270	0.065943
	<b>body_type</b>	0.327054	1.000000	0.304434	-0.036554	0.139391	0.026279	0.298733	0.182115	-0.236887
	<b>price</b>	0.069439	0.304434	1.000000	0.087051	-0.402373	-0.350389	0.025583	0.528101	-0.347764
	<b>vat</b>	-0.011138	-0.036554	0.087051	1.000000	-0.104631	-0.127659	-0.062094	0.005016	-0.060231
	<b>km</b>	0.051404	0.139391	-0.402373	-0.104631	1.000000	0.379356	0.413952	-0.038146	-0.049121
	<b>type</b>	-0.015027	0.026279	-0.350389	-0.127659	0.379356	1.000000	0.193455	-0.060956	0.052919
	<b>fuel_type</b>	0.085748	0.298733	0.025583	-0.062094	0.413952	0.193455	1.000000	0.204353	-0.151186
	<b>gears_num</b>	-0.146270	0.182115	0.528101	0.005016	-0.038146	-0.060956	0.204353	1.000000	-0.307914
	<b>comfort_convenience</b>	0.065943	-0.236887	-0.347764	-0.060231	-0.033351	0.052919	-0.151186	-0.307914	1.000000
	<b>entertainment_media</b>	0.129709	0.001831	-0.036164	-0.001129	-0.049121	0.013009	-0.028285	-0.106910	0.277904
	<b>extras</b>	0.003234	0.091602	0.012074	-0.006407	0.068500	-0.014204	0.031707	0.057241	-0.194022
	<b>safety_security</b>	-0.168685	-0.215044	-0.244648	0.095352	-0.071110	0.036312	-0.127852	-0.154906	0.344883
	<b>age</b>	0.014540	0.077671	-0.481426	-0.207585	0.748734	0.521802	0.338200	-0.079298	0.020332
	<b>previous_owners</b>	0.022334	-0.015783	-0.148630	-0.134057	0.158776	0.190820	0.028637	-0.036680	0.006863
	<b>hp_kw</b>	0.300697	0.443754	0.697984	0.031094	0.013366	-0.114630	0.113218	0.451227	-0.402756
	<b>inspection_new</b>	-0.055886	-0.079364	0.002714	0.111503	-0.036464	0.000203	-0.130849	-0.006485	-0.127837
	<b>paint_type</b>	0.005281	0.041088	-0.031693	0.018431	-0.032642	-0.004602	0.007209	-0.021477	0.078492
	<b>upholstery_type</b>	0.333009	0.288684	0.394886	-0.023963	0.029223	-0.048139	0.174516	0.224129	-0.215277
	<b>gearing_type</b>	-0.109929	-0.068857	-0.299150	-0.114237	0.114087	0.074165	0.043201	-0.463721	0.179809
	<b>displacement_cc</b>	0.327960	0.353084	0.287056	-0.014385	0.305729	0.090366	0.612020	0.337763	-0.234556
	<b>weight_kg</b>	0.387076	0.549287	0.460120	0.026726	0.152804	0.021155	0.387654	0.332270	-0.270643
	<b>drive_chain</b>	-0.091013	-0.050190	-0.160692	-0.012521	-0.004722	0.049896	-0.042669	-0.110661	0.045509
	<b>fuel_cons_comb</b>	0.387220	0.001373	0.273551	0.076376	-0.295297	-0.234526	-0.522230	0.006956	-0.019664

In [25]: # Corr between Target vs other Features

```
correlation_matrix = df_labeled.corr() # Labeled df

price_corr = correlation_matrix['price'].sort_values(ascending=False)
price_corr
```

Out[25]:

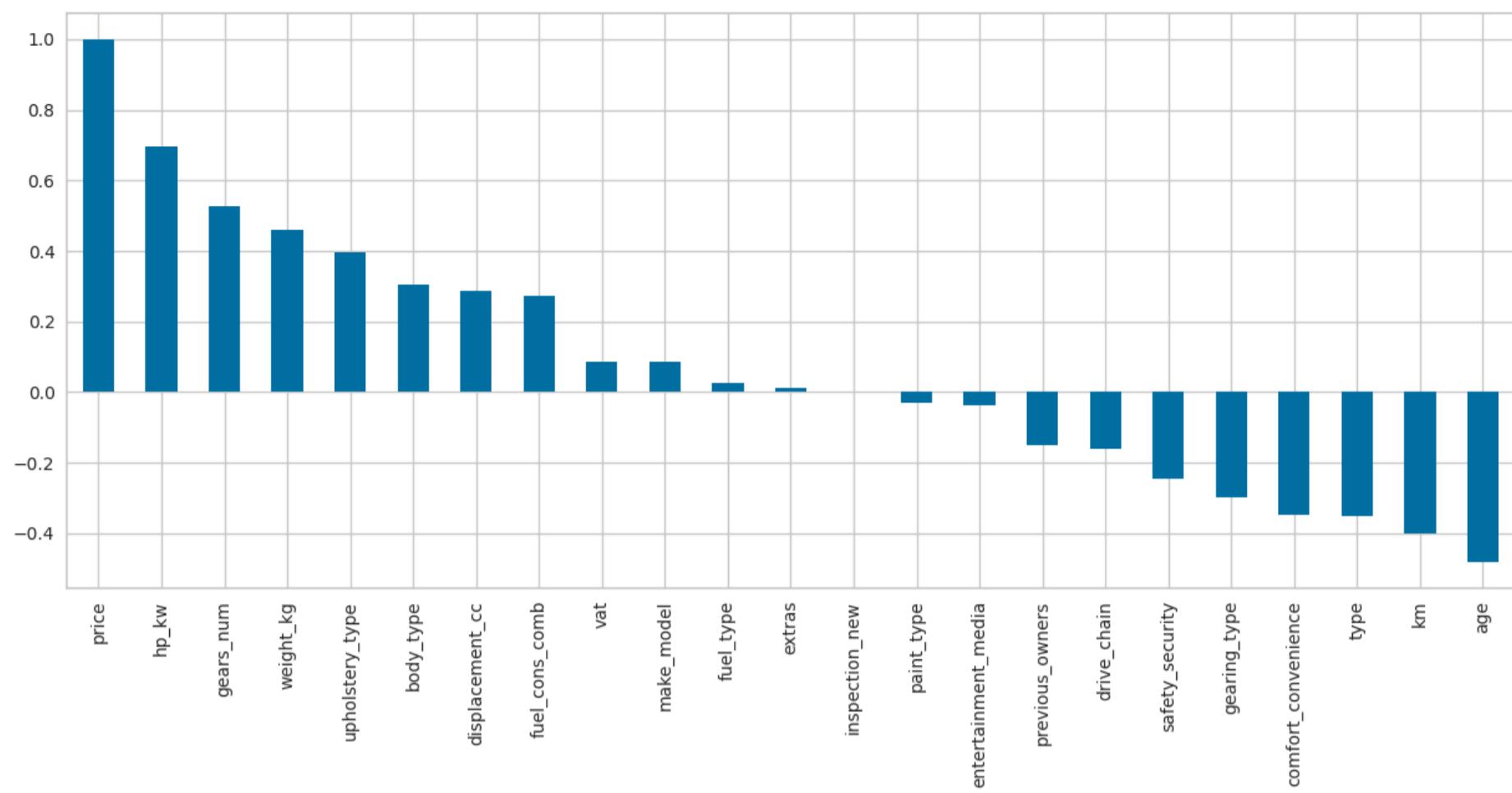
price	1.000
hp_kw	0.698
gears_num	0.528
weight_kg	0.460
upholstery_type	0.395
body_type	0.304
displacement_cc	0.287
fuel_cons_comb	0.274
vat	0.087
make_model	0.085
fuel_type	0.025
extras	0.012
inspection_new	0.003
paint_type	-0.032
entertainment_media	-0.036
previous_owners	-0.149
drive_chain	-0.161
safety_security	-0.245
gearing_type	-0.299
comfort_convenience	-0.348
type	-0.350
km	-0.402
age	-0.481

Name: price, dtype: float64

In [26]: # Target vs other Features

```
price_corr.plot(kind='bar', figsize=(15,6))
```

Out[26]: <Axes: >

**NOTE:****1. Strong Positive Correlations with Price:**

- **Gears Number ( gears\_num )**: Shows a strong positive correlation with `price` (0.53), indicating that cars with more gears tend to be more expensive.
- **Engine Power ( hp\_kw )**: Exhibits a moderate positive correlation with `price` (0.45), suggesting that higher-powered cars are generally more expensive.
- **Engine Displacement ( displacement\_cc )**: Has a moderate positive correlation with `price` (0.36), indicating that cars with larger engine displacements tend to be more expensive.

**2. Negative Correlations with Price:**

- **Mileage ( km )**: Displays a negative correlation with `price` (-0.35), suggesting that cars with higher mileage tend to be cheaper.
- **Age ( age )**: Exhibits a negative correlation with `price` (-0.21), indicating that older cars tend to be cheaper.
- **Fuel Consumption ( fuel\_cons\_comb )**: Shows a moderate negative correlation with `price` (-0.30), indicating that cars with higher fuel consumption tend to be cheaper.

**3. Multicollinearity Considerations:**

- **Gears Number ( gears\_num ), Engine Power ( hp\_kw ), and Engine Displacement ( displacement\_cc )**: These features show strong correlations with each other and with `price`, suggesting potential multicollinearity.
- **Mileage ( km ) and Age ( age )**: These features also show a strong correlation (0.76) with each other, indicating redundancy.

Managing multicollinearity among these highly correlated features is crucial to ensure model stability and performance. By addressing these correlations, we can build a more reliable and accurate price prediction model.

## Outlier Analysis

Linear models are generally sensitive to outliers because they seek linear relationships between data points and employ MSE-like loss functions that amplify large errors. Some models, such as decision trees and robust regression, are more resilient to outliers and are less affected by such data. Therefore, when choosing a model, the characteristics of the dataset and the presence of outliers should be considered.

- However, we will not intervene with outliers at the moment, but we can take action later according to the model's forecasting performance.
- Let's observe the outliers for now.

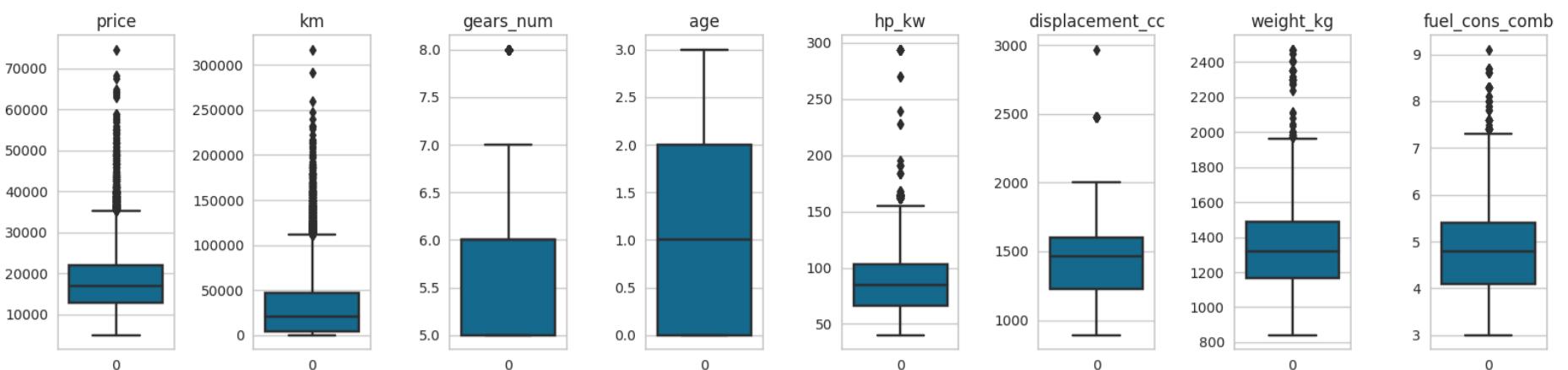
```
In [27]: # Checking Outliers
# Initialize the subplot counter
x = 0

#Numerical features;
numerical_columns = ['price', 'km', 'gears_num', 'age', 'hp_kw', 'displacement_cc', 'weight_kg', 'fuel_cons_comb']

# Create a figure with specified size
plt.figure(figsize=(16, 4))

# Loop through each numerical column and create a boxplot
for col in numerical_columns:
    x += 1
    plt.subplot(1, 8, x)
    sns.boxplot(data=df[col])
    plt.title(col)

# Show the plots
plt.tight_layout() # Adjust subplots to fit in the figure area.
plt.show()
```



```
In [28]: # Calculate skewness for numeric features

# A skewness value greater than 1 indicates positive skewness,
# a skewness value less than -1 indicates negative skewness,
# and a skewness value close to zero indicates a relatively symmetric distribution.

num_cols = df.select_dtypes('number').columns

skew_limit = 0.75 # define a limit above which we will log transform
skew_vals = df[num_cols].skew()

# Showing the skewed columns
skew_cols = (skew_vals
    .sort_values(ascending=False)
    .to_frame()
    .rename(columns={0:'Skew'})
    .query('abs(Skew) > {}'.format(skew_limit)))

skew_cols
```

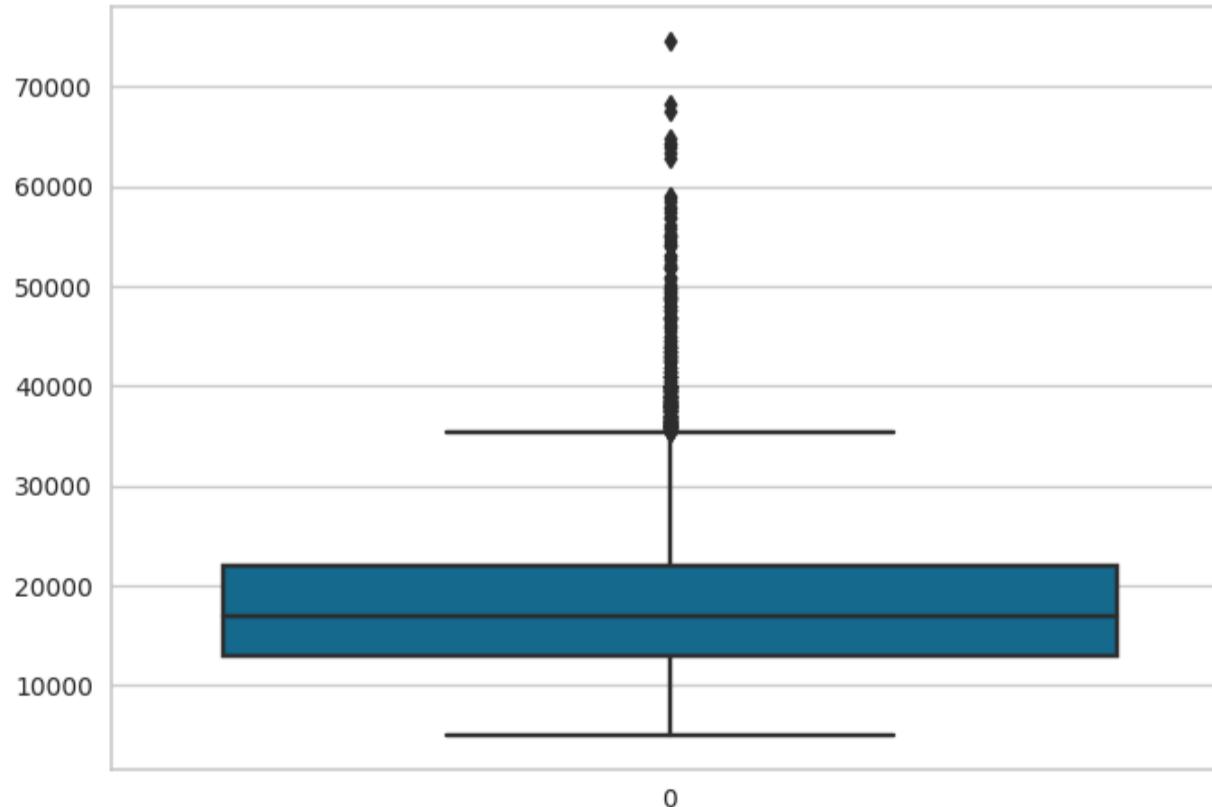
Out[28]:

Skew	
km	1.651
hp_kw	1.330
price	1.269
inspection_new	1.116
previous_owners	1.103
weight_kg	1.069

## Target Variable

```
In [29]: # Checking outliers for target variable 'price' with boxplot

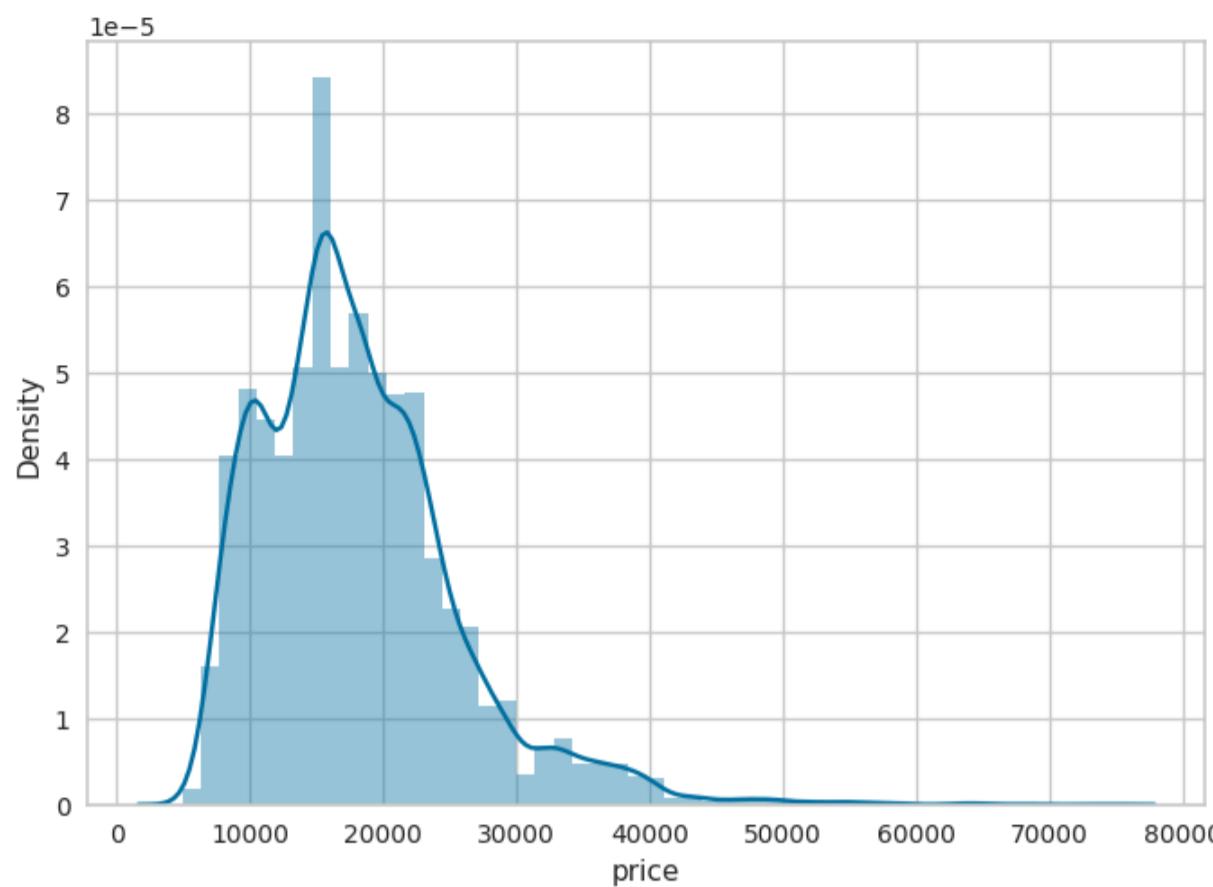
sns.boxplot(df['price']);
```



```
In [30]: #Skewness of the target variable
print("Skewness: %f" % df['price'].skew())

# Distribution of target variable before log transformation
price_untransformed = sns.distplot(df['price'])
```

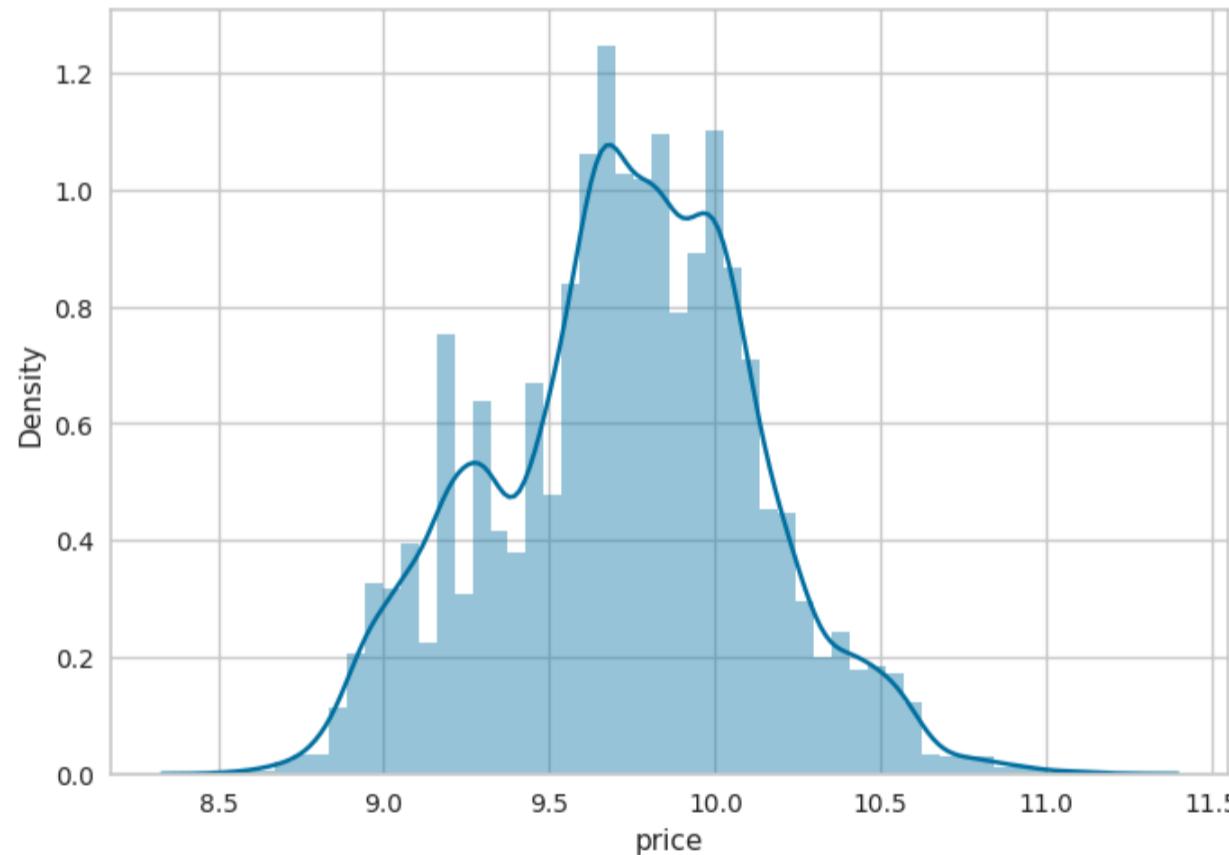
Skewness: 1.268788



- The range of skewness for a fairly symmetrical bell curve distribution is between -0.5 and 0.5;
- moderate skewness is -0.5 to -1.0 and 0.5 to 1.0;
- and highly skewed distribution is < -1.0 and > 1.0.

In our case, we have  $\sim 1.2$ , so it is considered skewed data. Now, we can try to transform our data, so it looks more normally distributed.

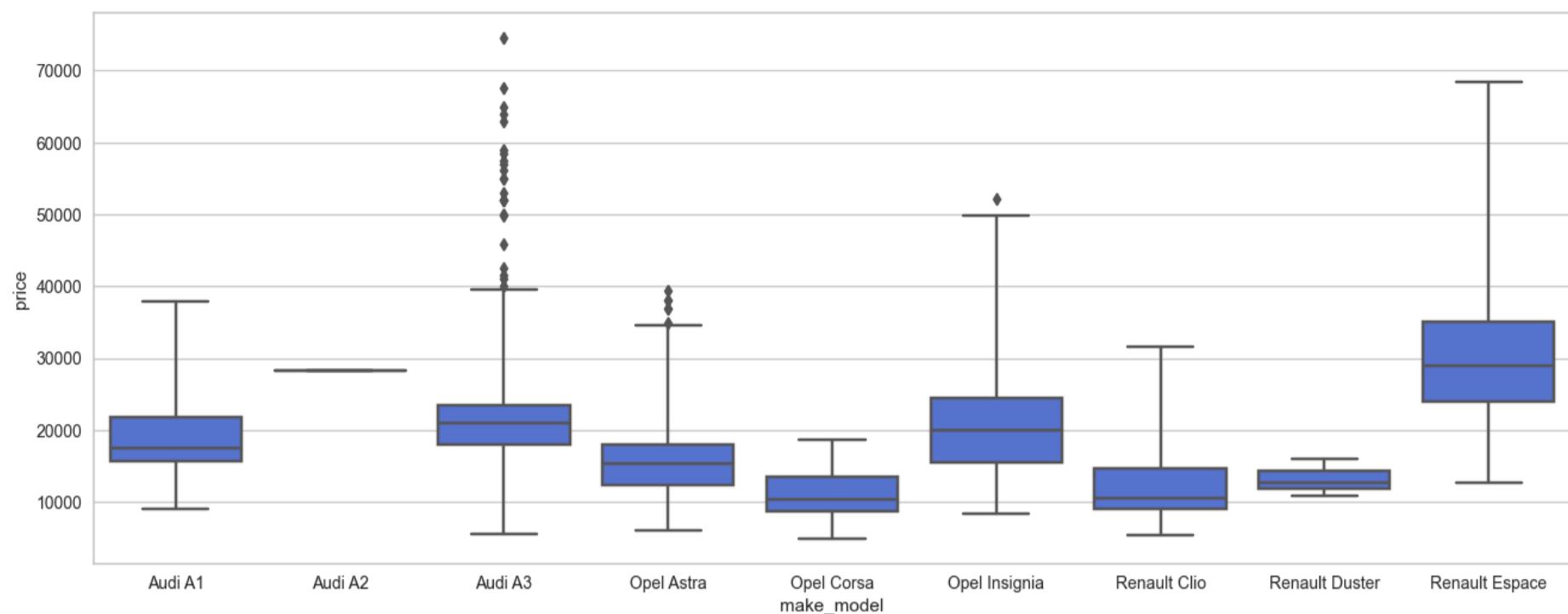
```
In [31]: #After Log Transformation
price_transformed = sns.distplot(np.log(df['price']))
```



## 'Make\_model' Feature

- As we observed in the distributions of the `make_model` feature values above, Audi A2 has only one data point.
  - The fact that the Audi A2 has only one data point indicates that there is not enough data to represent this model.
  - it is insufficient for generalization and should be excluded from the analysis.
  - By removing Audi A2, we ensure the model's stability and accuracy.

```
In [10]: # Make and Models
plt.figure(figsize=(16,6))
sns.boxplot(x="make_model", y="price", data=df, whis=3, color="royalblue")
plt.show()
```



```
In [11]: # Outlier check for the make_model feature
```

```
total_outliers = []

for model in df['make_model'].unique():

    car_prices = df[df['make_model']==model]['price']

    Q1 = car_prices.quantile(0.25)
    Q3 = car_prices.quantile(0.75)
    IQR = Q3-Q1
    lower_lim = Q1-1.5*IQR
    upper_lim = Q3+1.5*IQR

    count_of_outliers = (car_prices[(car_prices < lower_lim) | (car_prices > upper_lim)]).count()

    total_outliers.append(count_of_outliers)

print(f" The count of outlier for {model}<15} : {count_of_outliers<5}, \
      The rate of outliers : {(count_of_outliers/len(df[df['make_model']== model])).round(3)})")
print()
print("Total_outliers : ",sum(total_outliers), "The rate of total outliers :", (sum(total_outliers)/len(df)).round(3))
```

The count of outlier for Audi A1 : 5 , The rate of outliers : 0.002  
The count of outlier for Audi A2 : 0 , The rate of outliers : 0.0  
The count of outlier for Audi A3 : 56 , The rate of outliers : 0.02  
The count of outlier for Opel Astra : 127 , The rate of outliers : 0.055  
The count of outlier for Opel Corsa : 0 , The rate of outliers : 0.0  
The count of outlier for Opel Insignia : 109 , The rate of outliers : 0.045  
The count of outlier for Renault Clio : 37 , The rate of outliers : 0.025  
The count of outlier for Renault Duster : 0 , The rate of outliers : 0.0  
The count of outlier for Renault Espace : 20 , The rate of outliers : 0.023

```
Total_outliers : 354 The rate of total outliers : 0.025
```

- Some models, like Opel Astra, Opel Insignia, and Renault Clio, have a high number of outliers, indicating significant price deviations.
- Models like Audi A1, Opel Corsa, and Renault Duster have more consistent price distributions with fewer outliers.
- The overall outlier rate is 2.6%, showing the dataset's sensitivity to outliers.
- These outliers should be considered in analysis and modeling.

### Remove the AudiA2 model from the df

```
In [12]: models = df['make_model'].value_counts()
models
```

```
Out[12]: make_model
Audi A3      2758
Opel Insignia 2417
Audi A1      2377
Opel Astra    2305
Opel Corsa    1994
Renault Clio   1486
Renault Espace  884
Renault Duster  20
Audi A2        1
Name: count, dtype: int64
```

```
In [13]: # Dropping the only Audi A2 observation to increase model performance
```

```
df = df[df['make_model'] != 'Audi A2']
```

```
In [14]: df[df['make_model']=="Audi A2"]
```

```
Out[14]: make_model  body_type  price  vat  km  type  fuel_type  gears_num  comfort_convenience  entertainment_media  extras  safety_security  age  previous_c
```

# Feature Engineering

## Get Dummies

### Explanation:

**Original Data:** The `comfort_convenience`, `entertainment_media`, `extras`, and `safety_security` columns contain categorical values.

	<code>comfort_convenience</code>	<code>entertainment_media</code>	<code>extras</code>	<code>safety_security</code>
	Air Conditioning, Cruise Control	Bluetooth, CD Player	Alloy Wheels	ABS, Airbags

### After Get Dummies:

- Each category has been converted into a separate column.
- For example, `Air Conditioning` and `Cruise Control` are represented as separate columns with the `cc_` prefix.
- Similarly, other categories are represented with their own prefixes, with the presence of a category indicated by 1 and its absence by 0.

	<code>cc_Air Conditioning</code>	<code>cc_Cruise Control</code>	<code>em_Bluetooth</code>	<code>em_CD Player</code>	<code>ex_Alloy Wheels</code>	<code>ss_ABS</code>	<code>ss_Airbags</code>
	1	1	1	1	1	1	1

This process allows machine learning models to interpret these categorical features as numerical data, thereby improving model performance.

```
In [30]: # Select the categorical features that have more than one value to separate
selected_columns = df.loc[:, ['comfort_convenience', 'entertainment_media', 'extras', 'safety_security']]
selected_columns.head()
```

	<code>comfort_convenience</code>	<code>entertainment_media</code>	<code>extras</code>	<code>safety_security</code>
0	Air conditioning,Armrest,Automatic climate con...	Bluetooth,Hands-free equipment,On-board comput...	Alloy wheels,Catalytic Converter,Voice Control	ABS,Central door lock,Daytime running lights,D...
1	Air conditioning,Automatic climate control,Hil...	Bluetooth,Hands-free equipment,On-board comput...	Alloy wheels,Sport seats,Sport suspension,Voi...	ABS,Central door lock,Central door lock with r...
2	Air conditioning,Cruise control,Electrical sid...	MP3,On-board computer	Alloy wheels,Voice Control	ABS,Central door lock,Daytime running lights,D...
3	Air suspension,Armrest,Auxiliary heating,Elect...	Bluetooth,CD player,Hands-free equipment,MP3,O...	Alloy wheels,Sport seats,Voice Control	ABS,Alarm system,Central door lock with remote...
4	Air conditioning,Armrest,Automatic climate con...	Bluetooth,CD player,Hands-free equipment,MP3,O...	Alloy wheels,Sport package,Sport suspension,Vo...	ABS,Central door lock,Driver-side airbag,Elect...

```
In [15]: # Get Dummies and separate them
df = df.join(df["comfort_convenience"].str.get_dummies(sep = ",").add_prefix("cc_"))
df = df.join(df["entertainment_media"].str.get_dummies(sep = ",").add_prefix("em_"))
df = df.join(df["extras"].str.get_dummies(sep = ",").add_prefix("ex_"))
df = df.join(df["safety_security"].str.get_dummies(sep = ",").add_prefix("ss_"))
```

```
In [16]: # Drop the original features we don't need anymore.
df.drop(["comfort_convenience", "entertainment_media", "extras", "safety_security"], axis=1, inplace=True)
```

```
In [17]: # One-Hot Encoding get_dummies for all df
df = pd.get_dummies(df, drop_first=True)
```

```
In [18]: print(df.shape)
df.head(3)

# New feature values are boolean dtype after get_dummies
```

(14241, 133)

Out[18]:

	<code>price</code>	<code>km</code>	<code>gears_num</code>	<code>age</code>	<code>previous_owners</code>	<code>hp_kw</code>	<code>inspection_new</code>	<code>displacement_cc</code>	<code>weight_kg</code>	<code>fuel_cons_comb</code>	<code>cc_Air conditioning</code>	<code>cc_Air suspension</code>
0	15770	56013.000	7.000	3.000	2.000	66.000	1	1422.000	1220.000	3.800	1	0
1	14500	80000.000	7.000	2.000	1.000	141.000	0	1798.000	1255.000	5.600	1	0
2	14640	83450.000	7.000	3.000	1.000	85.000	0	1598.000	1135.000	3.800	1	0

```
In [19]: # Convert boolean columns(after get_dummy) to integers, changing True to 1 and False to 0.
bool_columns = df.columns[df.dtypes == 'bool']
df[bool_columns] = df[bool_columns].astype(int)
```

```
In [20]: print(df.shape)
df.head(3)

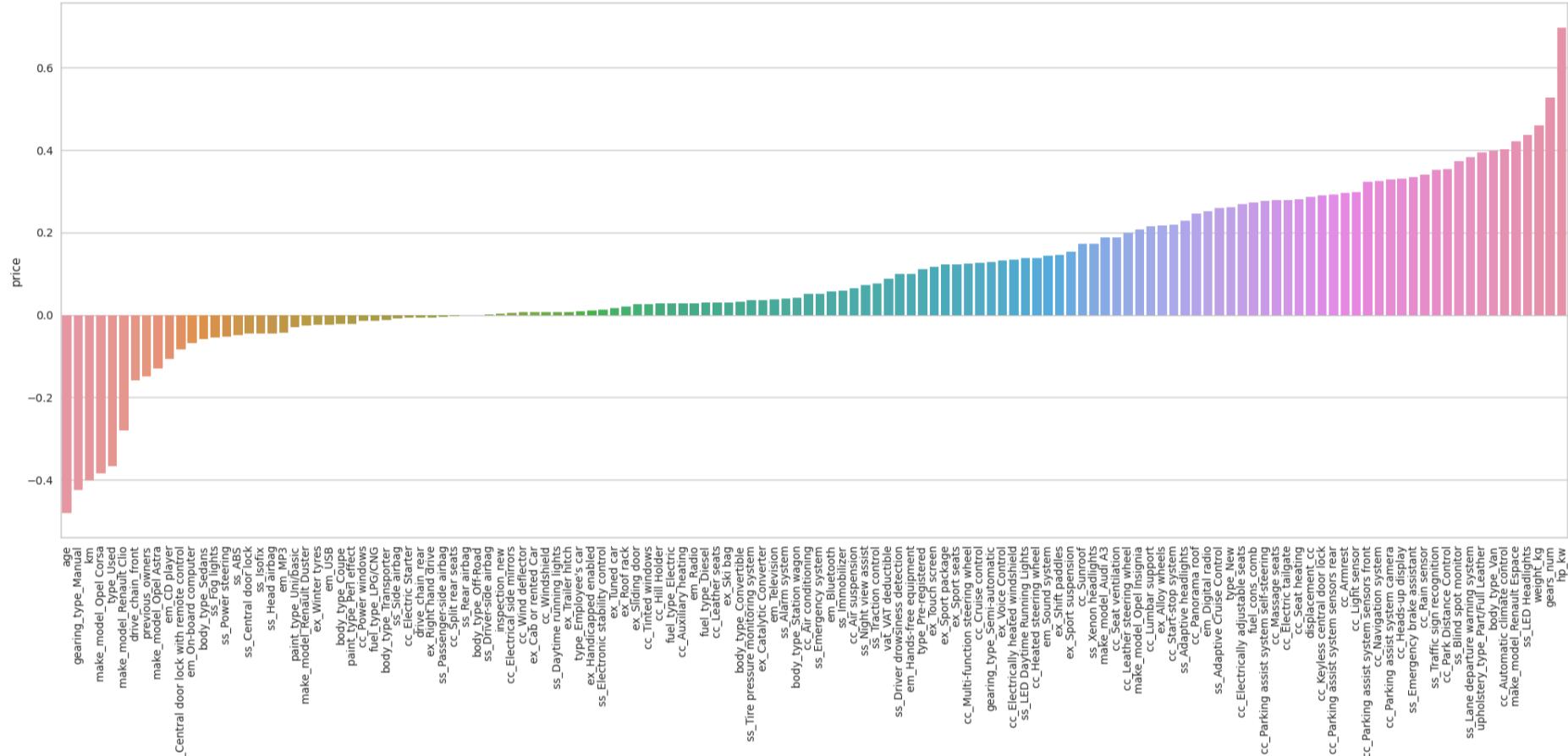
(14241, 133)
```

Out[20]:

	price	km	gears_num	age	previous_owners	hp_kw	inspection_new	displacement_cc	weight_kg	fuel_cons_comb	cc_Air conditioning	cc_Air suspension
0	15770	56013.000	7.000	3.000		2.000	66.000		1	1422.000	1220.000	3.800
1	14500	80000.000	7.000	2.000		1.000	141.000		0	1798.000	1255.000	5.600
2	14640	83450.000	7.000	3.000		1.000	85.000		0	1598.000	1135.000	3.800
...	...	...	...	...	...	...	...	...	...	...	...	...

```
In [ ]: # Check the correlation between target and all features (Including new features after get dummy)
corr_by_price = df.corr()["price"].sort_values()[:-1]
corr_by_price
```

```
In [42]: # Plot correlation between target and all features
plt.figure(figsize = (20,10))
sns.barplot(x = corr_by_price.index, y = corr_by_price)
plt.xticks(rotation=90)
plt.tight_layout();
```



## MACHINE LEARNING MODELLING

Functions to use to evaluate the model's scores at the end.

--> Model Performance Evaluation Function: `train_val()`

- This function evaluates a model's performance on both the training and test datasets
- by calculating various metrics such as R<sup>2</sup>, MAE, MSE, and RMSE.

```
In [21]: def train_val(model, X_train, y_train, X_test, y_test, i):
    y_pred = model.predict(X_test)
    y_train_pred = model.predict(X_train)

    scores = {
        "R2": r2_score(y_train, y_train_pred),
        "mae": mean_absolute_error(y_train, y_train_pred),
        "mse": mean_squared_error(y_train, y_train_pred),
        "rmse": np.sqrt(mean_squared_error(y_train, y_train_pred)),

        "R2": r2_score(y_test, y_pred),
        "mae": mean_absolute_error(y_test, y_pred),
        "mse": mean_squared_error(y_test, y_pred),
        "rmse": np.sqrt(mean_squared_error(y_test, y_pred))}
    }

    return pd.DataFrame(scores)
```

--> Adjusted R<sup>2</sup> : `adj_r2()`

- R<sup>2</sup> Score: Indicates how much of the variance in the dependent variable is explained by the model.
- Adjusted R<sup>2</sup> Score: Adjusts the explanatory power of the model based on the number of independent variables, reducing the risk of overfitting.

```
In [22]: def adj_r2(y_test, y_pred, df):
    r2 = r2_score(y_test, y_pred)
    n = df.shape[0]
    p = df.shape[1]-1
    adj_r2 = 1 - (1-r2)*(n-1)/(n-p-1)
    return adj_r2
```

## Split the Data

```
In [23]: # Split Target and Independent Features

X = df.drop(columns="price")
y = df.price
```

```
In [24]: # Train-test Split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
In [25]: X.shape
```

```
Out[25]: (14241, 132)
```

## Linear Regression

- Started with a simple, basic model without any special optimizations, manipulations, scaling or outlier handling on the dataset, which in machine learning and data science is referred to as a "vanilla model."
- By observing the performance of this initial model, we can compare it with more complex and optimized models in the end if we need.

## Model

```
In [26]: # Set and fit the model

linear_model = LinearRegression()

linear_model.fit(X_train, y_train)
```

```
Out[26]: ▾ LinearRegression
          LinearRegression()
```

## Prediction

```
In [27]: # Prediction
y_pred = linear_model.predict(X_test)

# Calculation of the metrics of the test data to compare models at the end.
lm_R2 = r2_score(y_test, y_pred)
lm_mae = mean_absolute_error(y_test, y_pred)
lm_rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print(lm_R2)
print(lm_mae)
print(lm_rmse)
```

```
0.8839352080753037
1725.5088166161436
2501.8602245902453
```

## Evaluating the Linear Model

```
In [28]: # Check the coefficients to see how it affects top-10 feature importance

lm1_coef = pd.DataFrame(linear_model.coef_, index=X.columns, columns=["lm1_Coeff"]).sort_values("lm1_Coeff")
lm1_coef.head(10)
```

	lm1_Coeff
<b>make_model_Renault Duster</b>	-10291.130
<b>make_model_Renault Clio</b>	-5255.546
<b>make_model_Opel Corsa</b>	-5086.767
<b>drive_chain_rear</b>	-3802.868
<b>make_model_Opel Astra</b>	-3786.396
<b>type_Employee's car</b>	-2982.985
<b>type_Used</b>	-2824.140
<b>drive_chain_front</b>	-2384.019
<b>type_Pre-registered</b>	-2191.709
<b>age</b>	-1588.903

In [29]: # Performance metrics

```
lm_score = train_val(linear_model, X_train, y_train, X_test, y_test, 'linear')
lm_score

# The model shows a high R2 score on the training data (0.89) and R2 score on the test data (0.88), indicating potential overfitting.
# The difference between MAE and RMSE indicates outlier effect.
```

Out[29]:

	linear_train	linear_test
R2	0.890	0.884
mae	1718.455	1725.509
mse	6088991.592	6259304.583
rmse	2467.588	2501.860

In [30]:

```
# Normalized RMSE
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print('rmse:', rmse)

#Average error rate;
print('avg_error', rmse/df.price.mean())
```

```
rmse: 2501.8602245902453
avg_error 0.13822233395909522
```

In [31]: # Adjusted R<sup>2</sup>

```
adj_r2(y_test, y_pred, df)
```

```
# Additionally, the Adjusted R2 score (0.882) for the test data confirms that while the model explains a large portion of the variance,
# the inclusion of more features may not significantly improve the model's performance.
```

Out[31]: 0.8828492602064307

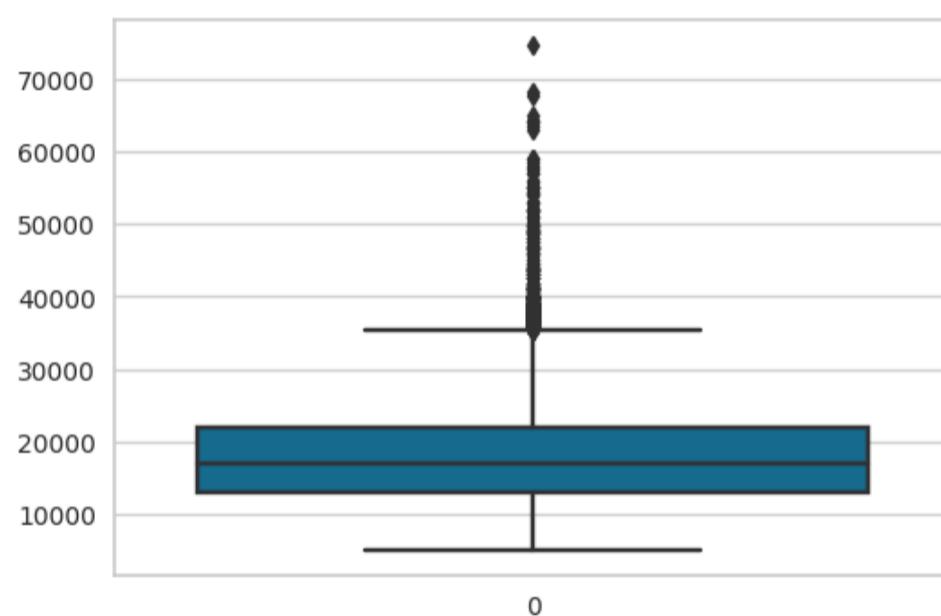
#### NOTE:

- Initial linear regression model was built using the provided dataset without any data manipulation.
- The model explains 88.2% of the variance in the target variable, indicating a strong fit but with some room for improvement.

## Remove Outliers

In [141...]

```
# Checking outliers for target variable 'price' with boxplot
plt.figure(figsize =(6,4))
sns.boxplot(df.price);
```



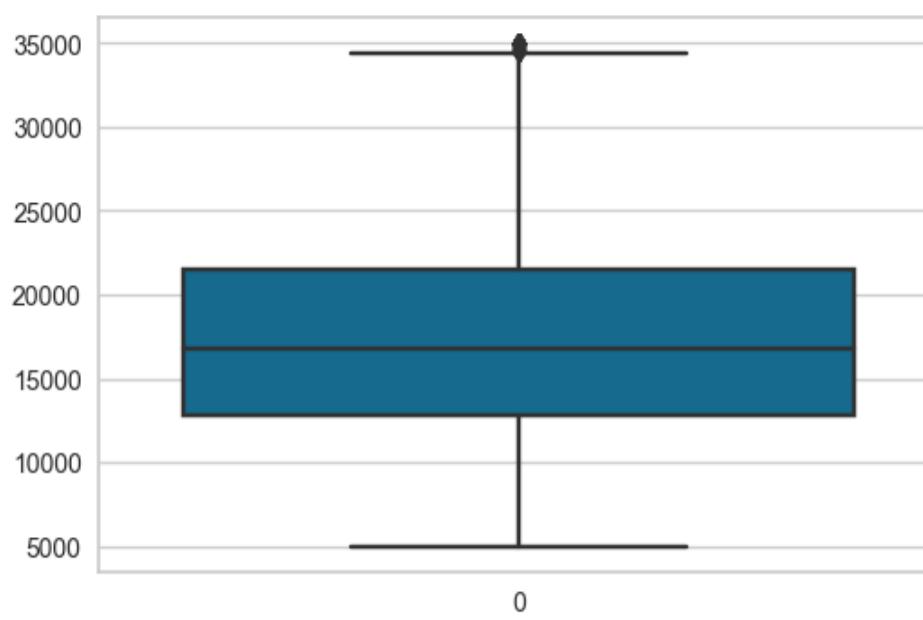
In [32]: len(df[df.price > 35000])

Out[32]: 473

In [33]: # Excluding rows where the price is greater than 35,000 using new df1

```
df1 = df[~(df.price > 35000)]
```

In [34]: plt.figure(figsize = (6,4))
sns.boxplot(df1.price);



## Linear Model Without Outliers

```
In [35]: #Split the data for df1
X = df1.drop(columns = "price")
y = df1.price

#Split train-test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=101)
```

```
In [36]: #Set and fit the model2
linear_model2 = LinearRegression()
linear_model2.fit(X_train, y_train)
```

```
Out[36]: ▾ LinearRegression
LinearRegression()
```

```
In [37]: # Check the coefficients to see how it affects feature importance
lm2_coef = pd.DataFrame(linear_model2.coef_, index = X.columns, columns=["lm2_Coeff"]).sort_values("lm2_Coeff")
lm2_coef.head(10)
```

	lm2_Coeff
make_model_Renault Duster	-9447.389
make_model_Renault Clio	-5424.873
make_model_Opel Corsa	-5245.299
make_model_Opel Astra	-3372.469
drive_chain_rear	-2859.199
type_Employee's car	-2090.024
type_Used	-1975.265
ex_Sliding door	-1618.279
gearing_type_Manual	-1581.537
age	-1445.872

```
In [38]: # Prediction2
y_pred = linear_model2.predict(X_test)

# Calculation of the metrics of the test data to compare models at the end.
lm2_R2 = r2_score(y_test, y_pred)
lm2_mae = mean_absolute_error(y_test, y_pred)
lm2_rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print(lm2_R2)
print(lm2_mae)
print(lm2_rmse)
```

0.8955794125789905  
1415.3813117633913  
1959.6494206641898

## Evaluating the Model

```
In [40]: # Performance Metrics
lm2_score = train_val(linear_model2, X_train, y_train, X_test, y_test, 'linear2_out')
lm2_score
```

Out[40]: linear2\_out\_train linear2\_out\_test

<b>R2</b>	0.898	0.896
<b>mae</b>	1414.209	1415.381
<b>mse</b>	3792751.061	3840225.852
<b>rmse</b>	1947.499	1959.649

In [41]: # Comparing the metric scores with Initial Linear Reg Model

results = pd.concat([lm\_score, lm2\_score], axis=1)  
results

Out[41]: linear\_train linear\_test linear2\_out\_train linear2\_out\_test

<b>R2</b>	0.890	0.884	0.898	0.896
<b>mae</b>	1718.455	1725.509	1414.209	1415.381
<b>mse</b>	6088991.592	6259304.583	3792751.061	3840225.852
<b>rmse</b>	2467.588	2501.860	1947.499	1959.649

In [42]: # Normalized RMSE of Linear model without outliers on target

rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred))

print('rmse:', rmse)

#Average error rate;  
print('avg\_error', rmse/df1.price.mean())rmse: 1959.6494206641898  
avg\_error 0.11316623260947056

- Before removing the outliers, our average prediction error was 13.81%.
- After excluding the outliers, it decreased to 11.31%.
- This represents an improvement of approximately 2.50% in our predictions.

## Scale Data

- In this section, we will rebuild the models using data scaled with MinMaxScaler.
- Scaling the data helps to normalize the feature ranges, which can improve the performance and convergence speed of the models

In [43]: #Split

X= df.drop(columns="price")  
y= df.price

In [44]: # Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

In [45]: #Scale -fit - transform

scaler = MinMaxScaler()  
  
X\_train\_scaled = scaler.fit\_transform(X\_train)  
X\_test\_scaled = scaler.transform(X\_test)

## Regularization

### Ridge Regression

#### GridSearchCV for Ridge Model

In [46]: from sklearn.model\_selection import GridSearchCV

# Set the model  
ridge = Ridge(random\_state=42)# Set the GridSearchCV and params  
alpha\_space = np.linspace(0.01, 100, 100)  
  
param\_grid = {"alpha":alpha\_space}  
  
ridge\_model = GridSearchCV(estimator=ridge,  
 param\_grid=param\_grid,  
 scoring='neg\_root\_mean\_squared\_error',  
 cv=10,  
 n\_jobs = -1)

In [47]: # Fit the Model

ridge\_model.fit(X\_train\_scaled,y\_train)

```
Out[47]: GridSearchCV
  estimator: Ridge
    Ridge
```

```
In [48]: ridge_model.best_params_
```

```
Out[48]: {'alpha': 1.02}
```

```
In [49]: # We obtained the best parameters for the Ridge model with alpha=1.02 after performing GridSearchCV.
```

```
# Set the Ridge model again with best alpha value;
ridge_model_alpha = Ridge(alpha=1.02, random_state=42).fit(X_train_scaled, y_train)

# Check the coefficients to see how it affects feature importance
ridge_coef = pd.DataFrame(ridge_model_alpha.coef_, index=X.columns, columns=["ridge_Coef"]).sort_values("ridge_Coef")
ridge_coef.head(10)
```

```
Out[49]:          ridge_Coef
km           -11817.904
make_model_Renault Duster -9203.373
make_model_Renault Clio   -5251.673
make_model_Opel Corsa    -5205.966
age            -4820.489
make_model_Opel Astra    -3740.368
drive_chain_rear        -2987.850
type_Employee's car     -2968.042
type_Used             -2813.387
drive_chain_front       -2334.704
```

```
In [50]: # Comparing the coefficients of the Ridge and initial Linear regression models to assess feature importance
```

```
pd.concat([lm1_coef, ridge_coef], axis=1).head(10)
```

```
Out[50]:      lm1_Coeff  ridge_Coef
make_model_Renault Duster -10291.130 -9203.373
make_model_Renault Clio   -5255.546 -5251.673
make_model_Opel Corsa    -5086.767 -5205.966
drive_chain_rear         -3802.868 -2987.850
make_model_Opel Astra    -3786.396 -3740.368
type_Employee's car     -2982.985 -2968.042
type_Used              -2824.140 -2813.387
drive_chain_front        -2384.019 -2334.704
type_Pre-registered     -2191.709 -2169.132
age                     -1588.903 -4820.489
```

## - Prediction

```
In [51]: # Prediction
y_pred = ridge_model.predict(X_test_scaled)

# Calculation of the metrics of the test data to compare models at the end.
ridge_R2 = r2_score(y_test, y_pred)
ridge_mae = mean_absolute_error(y_test, y_pred)
ridge_rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print(ridge_R2)
print(ridge_mae)
print(ridge_rmse)
```

0.8841789579307733  
1722.8854868936253  
2499.231741944769

## - Evaluating the Model

```
In [52]: # Model evaluation metrics scores

ridge_scores = train_val(ridge_model, X_train_scaled, y_train, X_test_scaled, y_test, 'ridge')
ridge_scores
```

	ridge_train	ridge_test
R2	0.890	0.884
mae	1716.690	1722.885
mse	6094440.410	6246159.300
rmse	2468.692	2499.232

```
In [53]: # Comparing the metric scores of earlier Models vs Grid Ridge Model
```

```
result = pd.concat([results, ridge_scores], axis=1)
result
```

	linear_train	linear_test	linear2_out_train	linear2_out_test	ridge_train	ridge_test
R2	0.890	0.884	0.898	0.896	0.890	0.884
mae	1718.455	1725.509	1414.209	1415.381	1716.690	1722.885
mse	6088991.592	6259304.583	3792751.061	3840225.852	6094440.410	6246159.300
rmse	2467.588	2501.860	1947.499	1959.649	2468.692	2499.232

## Lasso Regression

### GridSearchCV for Lasso Model

```
In [54]: # Set the model
lasso = Lasso(random_state=42)
```

```
# Set the GridSearchCV and params
alpha_space = np.linspace(0.01, 100, 100)

param_grid = {"alpha":alpha_space}

lasso_model = GridSearchCV(estimator=lasso,
                           param_grid=param_grid,
                           scoring='neg_root_mean_squared_error',
                           cv=10,
                           n_jobs = -1)
```

```
In [55]: # Fit the Model
lasso_model.fit(X_train_scaled,y_train)
```

```
Out[55]:
```

- ▶ GridSearchCV
- ▶ estimator: Lasso
- ▶ Lasso

```
In [167...]: lasso_model.best_params_
```

```
Out[167...]: {'alpha': 1.02}
```

```
In [56]: # We obtained the best parameters for the Lasso model with alpha=1.02 after performing GridSearchCV.
```

```
# Set the Lasso model again with best alpha params;
lasso_model_alpha = Lasso(alpha=1.02, random_state=42).fit(X_train_scaled, y_train)

# Check the coefficients to see how it affects feature importance
lasso_coef = pd.DataFrame(lasso_model_alpha.coef_, index = X.columns, columns=["lasso_Coef"]).sort_values("lasso_Coef")
lasso_coef.head(10)
```

	lasso_Coef
km	-11876.922
make_model_Renault Duster	-9094.844
make_model_Renault Clio	-5185.972
make_model_Opel Corsa	-5114.657
age	-4817.747
make_model_Opel Astra	-3749.536
type_Employee's car	-2890.334
type_Used	-2751.131
drive_chain_front	-2142.016
type_Pre-registered	-2080.564

```
In [57]: pd.concat([ridge_coef, lasso_coef], axis=1).head(10)
```

```
# Comparing the coefficients of the Ridge and Lasso regression models,
# to understand how each regularization technique affects feature importance
```

Out[57]:

	ridge_Coef	lasso_Coef
km	-11817.904	-11876.922
make_model_Renault Duster	-9203.373	-9094.844
make_model_Renault Clio	-5251.673	-5185.972
make_model_Opel Corsa	-5205.966	-5114.657
age	-4820.489	-4817.747
make_model_Opel Astra	-3740.368	-3749.536
drive_chain_rear	-2987.850	-647.723
type_Employee's car	-2968.042	-2890.334
type_Used	-2813.387	-2751.131
drive_chain_front	-2334.704	-2142.016

## - Prediction

In [58]:

```
# Prediction
y_pred = lasso_model.predict(X_test_scaled)

# Calculation of the metrics of the test data to compare models at the end.
lasso_R2 = r2_score(y_test, y_pred)
lasso_mae = mean_absolute_error(y_test, y_pred)
lasso_rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print(lasso_R2)
print(lasso_mae)
print(lasso_rmse)
```

0.8845643139633643  
1721.927898452896  
2495.0705962639568

## - Evaluating the Lasso Model

In [59]:

```
# Model evaluation metrics scores

lasso_scores = train_val(lasso_model, X_train_scaled, y_train, X_test_scaled, y_test, 'lasso')
lasso_scores
```

Out[59]:

	lasso_train	lasso_test
R2	0.890	0.885
mae	1719.602	1721.928
mse	6107178.086	6225377.280
rmse	2471.271	2495.071

In [60]:

```
# Comparing the metric scores of Grid Ridge and Grid Lasso models to evaluate their performance, and
# to assess how each regularization technique affects the model's predictive accuracy and error metrics.

result = pd.concat([result, lasso_scores], axis=1)
result
```

Out[60]:

	linear_train	linear_test	linear2_out_train	linear2_out_test	ridge_train	ridge_test	lasso_train	lasso_test
R2	0.890	0.884	0.898	0.896	0.890	0.884	0.890	0.885
mae	1718.455	1725.509	1414.209	1415.381	1716.690	1722.885	1719.602	1721.928
mse	6088991.592	6259304.583	3792751.061	3840225.852	6094440.410	6246159.300	6107178.086	6225377.280
rmse	2467.588	2501.860	1947.499	1959.649	2468.692	2499.232	2471.271	2495.071

- The comparison of metric scores indicates that both Grid Ridge and Grid Lasso models perform similarly,
- with Lasso slightly outperforming Ridge in test set R<sup>2</sup> and RMSE,
- suggesting marginally better generalization for the Lasso model.

## Elastic-Net Regression

### GridSearchCV for ElasticNet Model

In [61]:

```
# Set the model
elastic = ElasticNet(random_state=42)

# Set the GridSearchCV and params for ElasticNet (alpha and L1_ratio)
param_grid = {
    'alpha': [1.02, 2, 3, 4, 5, 7, 10, 11],
    'l1_ratio': [.5, .7, .9, .95, .99, 1]
}
```

```
elastic_model = GridSearchCV(estimator=elastic,
                             param_grid=param_grid,
                             scoring='neg_root_mean_squared_error',
                             cv=10,
                             n_jobs=-1)
```

alpha:

- This parameter controls the overall regularization strength.
- When alpha = 0, ElasticNet performs no regularization.

= As the alpha value increases, the strength of regularization increases, which can help the model become more resistant to overfitting.

l1\_ratio:

- This parameter controls the mixture between L1 (Lasso) and L2 (Ridge) regularization.
- When l1\_ratio = 1, it means this is entirely L1 (i.e., only Lasso).
- When l1\_ratio = 0, it means this is entirely L2 (i.e., only Ridge).
- When  $0 < l1\_ratio < 1$ , you get a combination of both L1 and L2 regularizations.
- For example, if l1\_ratio = 0.5, it means L1 and L2 regularizations are used equally.

```
In [62]: # Fit the Model
elastic_model.fit(X_train_scaled, y_train)
```

```
Out[62]: GridSearchCV
         estimator: ElasticNet
                  ElasticNet
```

```
In [63]: elastic_model.best_params_
```

```
Out[63]: {'alpha': 1.02, 'l1_ratio': 1}
```

```
In [64]: # Prediction
y_pred = elastic_model.predict(X_test_scaled)

# Calculation of the metrics of the test data to compare models at the end.
em_R2 = r2_score(y_test, y_pred)
em_mae = mean_absolute_error(y_test, y_pred)
em_rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print(em_R2)
print(em_mae)
print(em_rmse)
```

0.8845643139633643  
1721.927898452896  
2495.0705962639568

```
In [65]: # Model evaluation metrics scores
```

```
elastic_scores = train_val(elastic_model, X_train_scaled, y_train, X_test_scaled, y_test, 'elastic')
elastic_scores
```

	elastic_train	elastic_test
R2	0.890	0.885
mae	1719.602	1721.928
mse	6107178.086	6225377.280
rmse	2471.271	2495.071

```
In [66]: # Comparing the metric scores of models to evaluate their performance, and
# to assess how each regularization technique affects the model's predictive accuracy and error metrics.

result = pd.concat([result, elastic_scores], axis=1)
result
```

	linear_train	linear_test	linear2_out_train	linear2_out_test	ridge_train	ridge_test	lasso_train	lasso_test	elastic_train	elastic_test
R2	0.890	0.884	0.898	0.896	0.890	0.884	0.890	0.885	0.890	0.885
mae	1718.455	1725.509	1414.209	1415.381	1716.690	1722.885	1719.602	1721.928	1719.602	1721.928
mse	6088991.592	6259304.583	3792751.061	3840225.852	6094440.410	6246159.300	6107178.086	6225377.280	6107178.086	6225377.280
rmse	2467.588	2501.860	1947.499	1959.649	2468.692	2499.232	2471.271	2495.071	2471.271	2495.071

### Comparing all Models

- The linear2\_out model, where outliers were excluded from the target feature, showed the best performance with the highest R<sup>2</sup> scores and the lowest MAE, MSE, and RMSE, But Considering Overfitting.
- Ridge, Lasso, and Elastic Net regressions with scale had similar performance, not significantly better than the second linear model.
- By combining the benefits of outlier removal and Lasso regularization, could achieve a more accurate and interpretable model.

--> Let's try to see the combined model: Lasso Model without Outliers

## Lasso2 (alpha=1.02) Model without Outliers

```
In [67]: # Excluding rows where the price is greater than 35,000 using df_final
print(len(df[df.price > 35000]))
```

```
df2 = df[~(df.price > 35000)]
```

473

```
In [68]: #Split the data
```

```
X = df2.drop(columns = "price")
y = df2.price
```

```
In [69]: # Split train-test
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=101)
```

```
In [70]: #Scale -fit - transform - X
```

```
scaler = MinMaxScaler()
```

```
X_train_scaled = scaler.fit_transform(X_train)
```

```
X_test_scaled = scaler.transform(X_test)
```

### - Model

```
In [71]: # Set the model
```

```
lasso2 = Lasso(alpha=1.02, random_state=42)
```

```
In [72]: # Fit the Model
```

```
lasso2.fit(X_train_scaled,y_train)
```

```
Out[72]:
```

```
▼ Lasso
Lasso(alpha=1.02, random_state=42)
```

### - Prediction

```
In [73]: # Prediction
```

```
y_pred = lasso2.predict(X_test_scaled)
```

```
# Calculation of the metrics of the test data to compare models at the end.
```

```
lasso2_R2 = r2_score(y_test, y_pred)
```

```
lasso2_mae = mean_absolute_error(y_test, y_pred)
```

```
lasso2_rmse = np.sqrt(mean_squared_error(y_test, y_pred))
```

```
print(lasso2_R2)
```

```
print(lasso2_mae)
```

```
print(lasso2_rmse)
```

0.8962714979555529

1408.9561317098285

1953.144480067058

### - Evaluating the Lasso Model

```
In [74]: # Model evaluation metrics scores
```

```
lasso2_scores = train_val(lasso2, X_train_scaled, y_train, X_test_scaled, y_test, 'lasso2')
```

```
Out[74]:
```

	lasso2_train	lasso2_test
R2	0.898	0.896
mae	1414.186	1408.956
mse	3806891.515	3814773.360
rmse	1951.126	1953.144

```
In [75]: # Comparing the metric scores of Grid Ridge and Grid Lasso models to evaluate their performance, and
# to assess how each regularization technique affects the model's predictive accuracy and error metrics.
```

```
result = pd.concat([result, lasso2_scores], axis=1)
result
```

```
Out[75]:
```

	linear_train	linear_test	linear2_out_train	linear2_out_test	ridge_train	ridge_test	lasso_train	lasso_test	elastic_train	elastic_test	lasso2_train
R2	0.890	0.884	0.898	0.896	0.890	0.884	0.890	0.885	0.890	0.885	0.8
mae	1718.455	1725.509	1414.209	1415.381	1716.690	1722.885	1719.602	1721.928	1719.602	1721.928	1414.1
mse	6088991.592	6259304.583	3792751.061	3840225.852	6094440.410	6246159.300	6107178.086	6225377.280	6107178.086	6225377.280	3806891.5
rmse	2467.588	2501.860	1947.499	1959.649	2468.692	2499.232	2471.271	2495.071	2471.271	2495.071	1951.1

### Comparing Models:

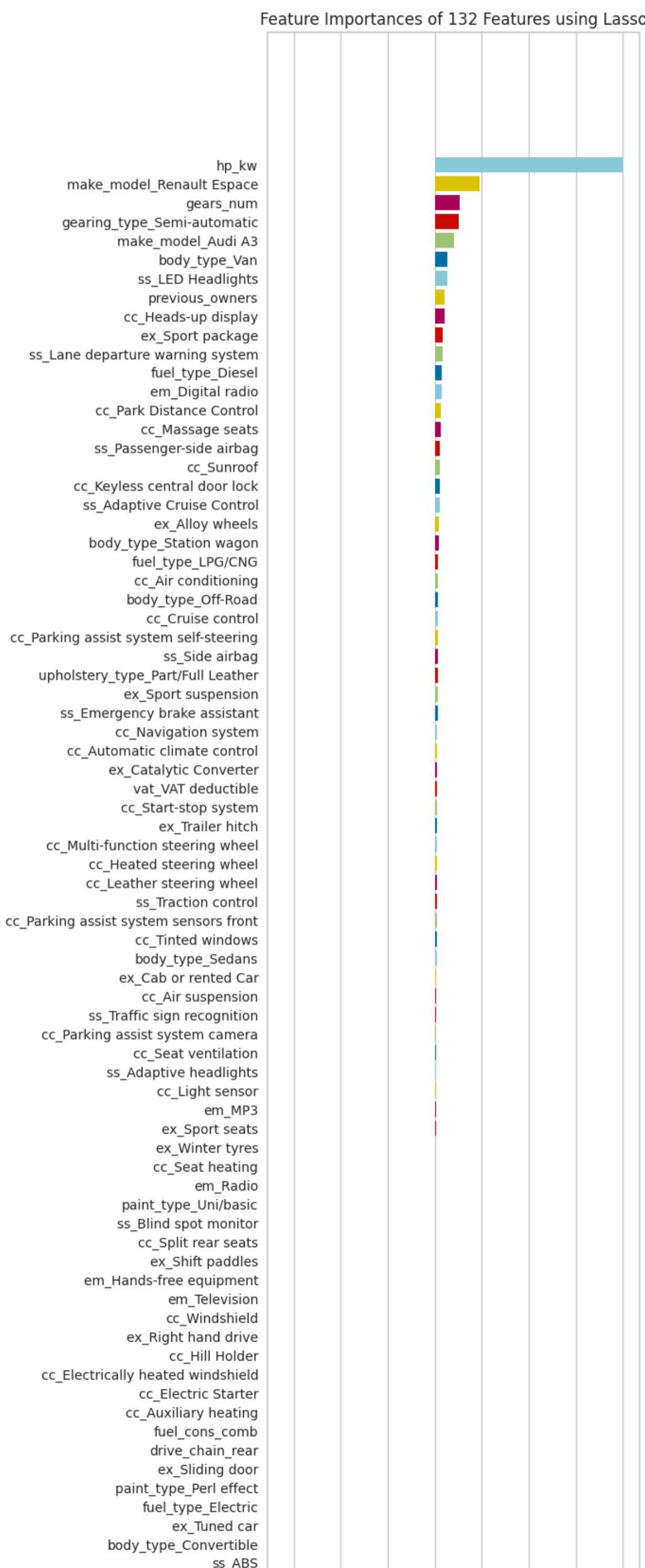
- The Ridge model, while scaled, does not outperform the Linear2 and Final Lasso models, indicating that the additional complexity does not provide a significant benefit in this case.

Recommendation for the Final Model:

- The Lasso2 model is recommended as it combines the benefits of regularization through Lasso, data scaling, and outlier removal, leading to robust performance and minimal overfitting.

## Feature Importance

```
In [103...]  
from yellowbrick.model_selection import FeatureImportances  
from yellowbrick.features import RadViz  
  
viz = FeatureImportances(Lasso(alpha=1.02), labels=X_train.columns) # Lasso Model is selected here  
visualizer = RadViz(size=(720, 3000))  
viz.fit(X_train_scaled, y_train)  
viz.show()  
  
# We will use Lasso for the Final model
```





```
Out[103]: <Axes: title={'center': 'Feature Importances of 132 Features using Lasso'}, xlabel='relative coefficient magnitude'>
```

## Lasso3 Model with Selected Features

```
In [76]: # Create new df with the selected features
df_new = df0[['make_model', 'hp_kw', 'km', 'age', 'Gearing_Type', 'price']]

# Selecting the top 5 features with the most impact on prediction.
# You might wonder why the make_model feature was chosen. When examining the visualization above,
# it was observed that among the features with the most impact on prediction, the make_model feature includes unique categorical observations
```

```
In [189]: df_new.head()
```

Out[189...]

	make_model	hp_kW	km	age	Gearing_Type	price
0	Audi A1	66.000	56013.000	3.000	Automatic	15770
1	Audi A1	141.000	80000.000	2.000	Automatic	14500
2	Audi A1	85.000	83450.000	3.000	Automatic	14640
3	Audi A1	66.000	73000.000	3.000	Automatic	14500
4	Audi A1	66.000	16200.000	3.000	Automatic	16790

In [77]: df\_new['make\_model'].value\_counts()

```
Out[77]: make_model
Audi A3      3097
Audi A1      2614
Opel Insignia 2598
Opel Astra    2525
Opel Corsa    2216
Renault Clio   1839
Renault Espace  991
Renault Duster  34
Audi A2        1
Name: count, dtype: int64
```

In [78]: df\_new = df\_new[df\_new['make\_model'] != 'Audi A2'] # Excluding data has only 1 value

In [79]: df\_new = df\_new[df\_new['price'] &lt; 35000] # Excluding the Outliers from target feature

In [80]: df\_new = pd.get\_dummies(df\_new) # Get Dummies for the final dataset

```
print(len(df_new))
df_new.head(3)
```

# After get dummy, features have boolean dtype.

15419

	hp_kW	km	age	price	make_model_Audi_A1	make_model_Audi_A3	make_model_Opel_Astra	make_model_Opel_Corsa	make_model_Opel_Insignia	make_model_Renault_Clio
0	66.000	56013.000	3.000	15770	True	False	False	False	False	False
1	141.000	80000.000	2.000	14500	True	False	False	False	False	False
2	85.000	83450.000	3.000	14640	True	False	False	False	False	False

## Split and Scale for df\_new

In [81]: X = df\_new.drop(columns=["price"])
y = df\_new['price']

In [82]: X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,
y,
test\_size=0.2,
random\_state=42)

In [83]: scaler = MinMaxScaler()
scaler.fit(X\_train)

X\_train\_scaled = scaler.transform(X\_train)
X\_test\_scaled = scaler.transform(X\_test)

## Model

In [84]: lasso\_model = Lasso(random\_state=42)

param\_grid = {'alpha': alpha\_space}

lasso3 = GridSearchCV(estimator=lasso\_model,
param\_grid=param\_grid,
scoring='neg\_root\_mean\_squared\_error',
cv=10,
n\_jobs=-1)

In [85]: lasso3.fit(X\_train\_scaled, y\_train)

Out[85]:

```
GridSearchCV
  estimator: Lasso
    Lasso
```

In [86]: lasso3.best\_params\_

Out[86]: {'alpha': 0.01}

In [87]: lasso3.best\_score\_ # best score of the test data

Out[87]: -2239.588352950935

```
In [88]: #Prediction
y_pred = lasso3.predict(X_test_scaled)

lasso3_R2 = r2_score(y_test, y_pred)
lasso3_mae = mean_absolute_error(y_test, y_pred)
lasso3_rmse = np.sqrt(mean_squared_error(y_test, y_pred))
```

```
In [89]: # Model evaluation metrics scores

lasso3_scores = train_val(lasso3, X_train_scaled, y_train, X_test_scaled, y_test, 'lasso3')
lasso3_scores
```

	lasso3_train	lasso3_test
R2	0.867	0.877
mae	1611.742	1553.998
mse	5007030.903	4547724.302
rmse	2237.640	2132.539

```
In [90]: result = pd.concat([result, lasso3_scores], axis=1)
result
```

	linear_train	linear_test	linear2_out_train	linear2_out_test	ridge_train	ridge_test	lasso_train	lasso_test	elastic_train	elastic_test	lasso2_train
R2	0.890	0.884	0.898	0.896	0.890	0.884	0.890	0.885	0.890	0.885	0.8
mae	1718.455	1725.509	1414.209	1415.381	1716.690	1722.885	1719.602	1721.928	1719.602	1721.928	1414.1
mse	6088991.592	6259304.583	3792751.061	3840225.852	6094440.410	6246159.300	6107178.086	6225377.280	6107178.086	6225377.280	3806891.5
rmse	2467.588	2501.860	1947.499	1959.649	2468.692	2499.232	2471.271	2495.071	2471.271	2495.071	1951.1

- The Lasso3 model, which included only the selected features `["make_model", "hp_kw", "km", "age", "Gearing_Type", "price"]`, demonstrated slightly lower performance with R<sup>2</sup> scores of 0.867 (train) and 0.877 (test), but it still maintained a reasonable prediction accuracy.

- Overall, the lasso2 model, where outlier excluded with Lasso regularization on the original df, provides the best balance of accuracy and robustness, indicating that removing outliers and focusing on relevant features significantly improves model performance.

## Compare Models Performance

```
In [203...]: scores = {
    "linear_m": {
        "r2_score": lm_R2,
        "mae": lm_mae,
        "rmse": lm_rmse
    },
    "linear2_m": {
        "r2_score": lm2_R2,
        "mae": lm2_mae,
        "rmse": lm2_rmse
    },
    "ridge_m": {
        "r2_score": ridge_R2,
        "mae": ridge_mae,
        "rmse": ridge_rmse
    },
    "lasso_m": {
        "r2_score": lasso_R2,
        "mae": lasso_mae,
        "rmse": lasso_rmse
    },
    "lasso2_m": {
        "r2_score": lasso2_R2,
        "mae": lasso2_mae,
        "rmse": lasso2_rmse
    },
    "lasso3_m": {
        "r2_score": lasso3_R2,
        "mae": lasso3_mae,
        "rmse": lasso3_rmse
    }
}
scores = pd.DataFrame(scores).T
scores
```

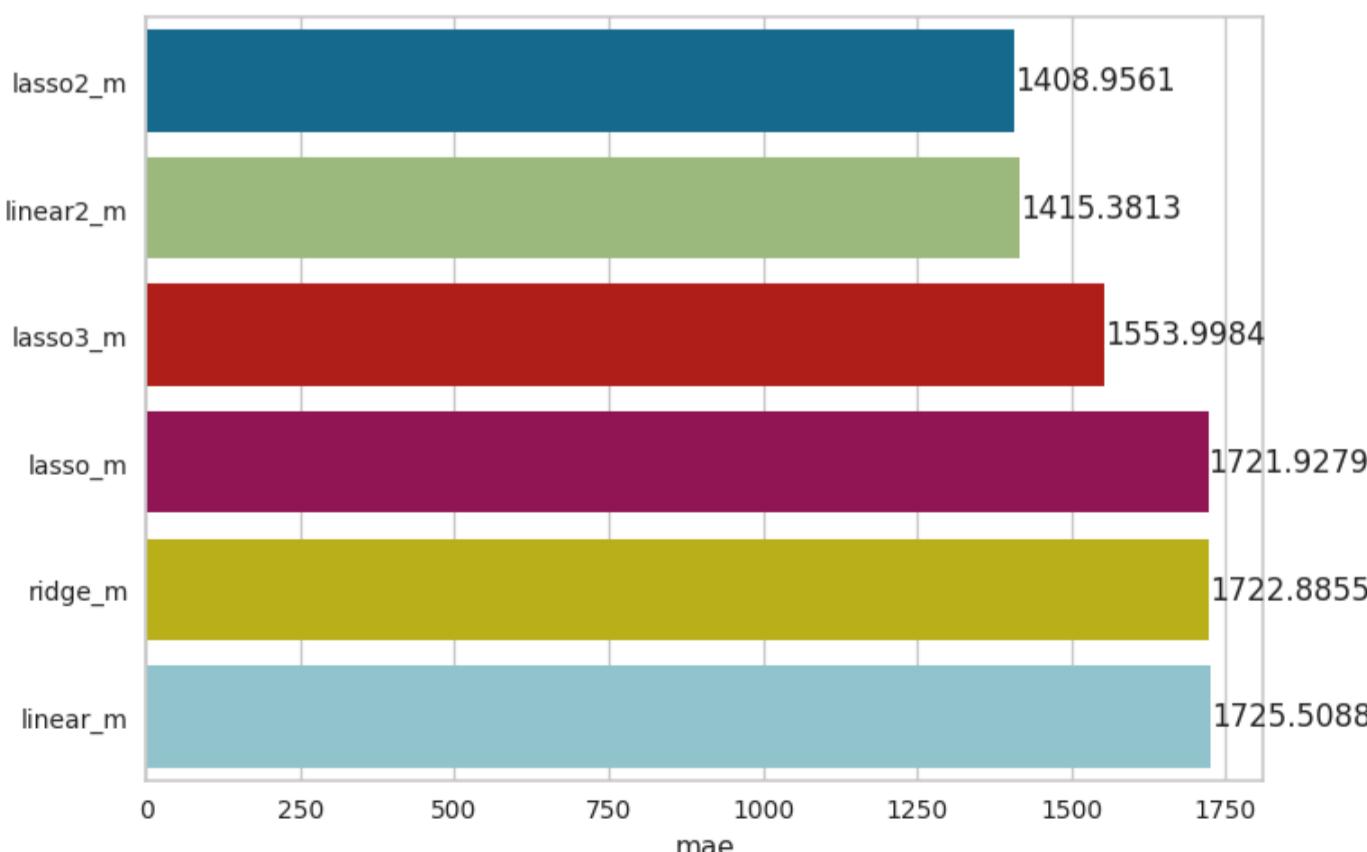
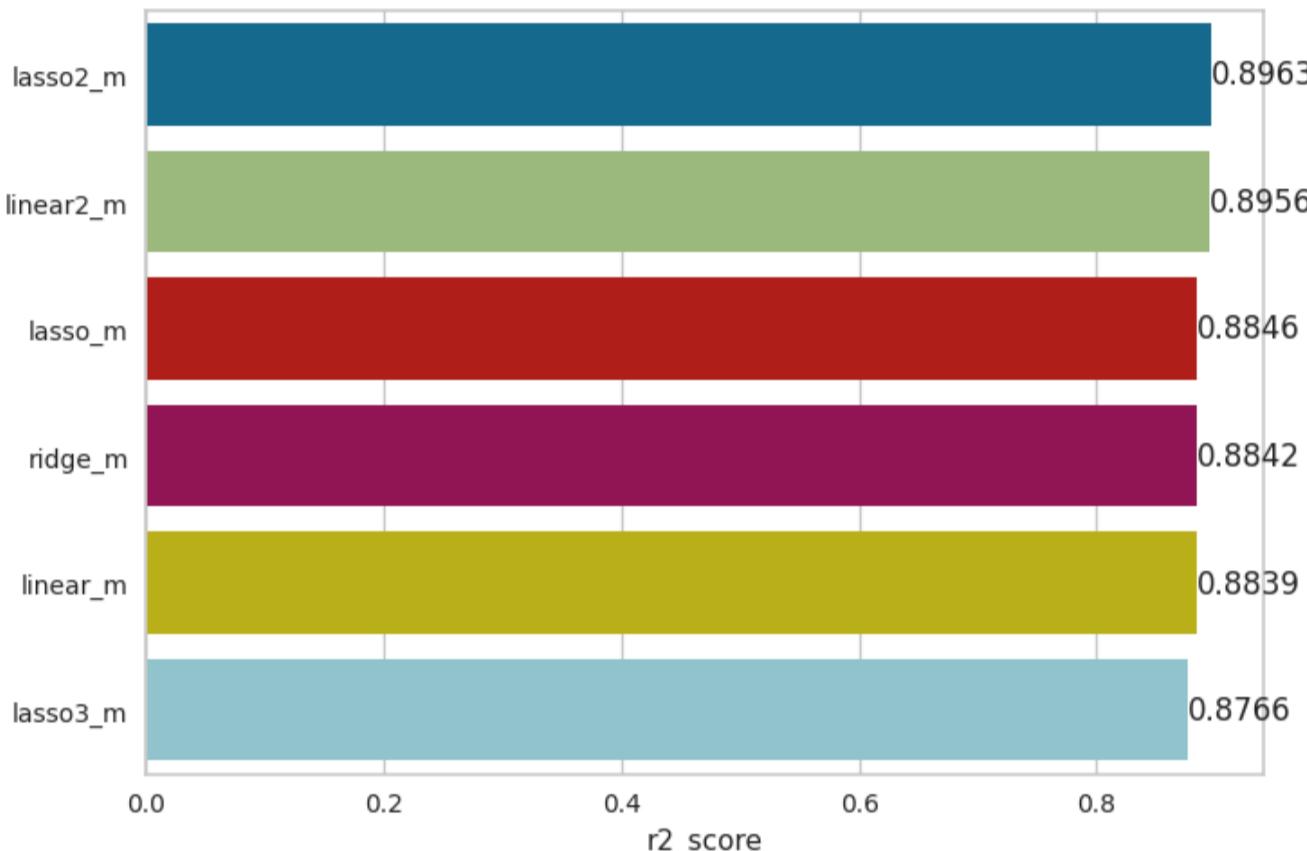
Out[203...]

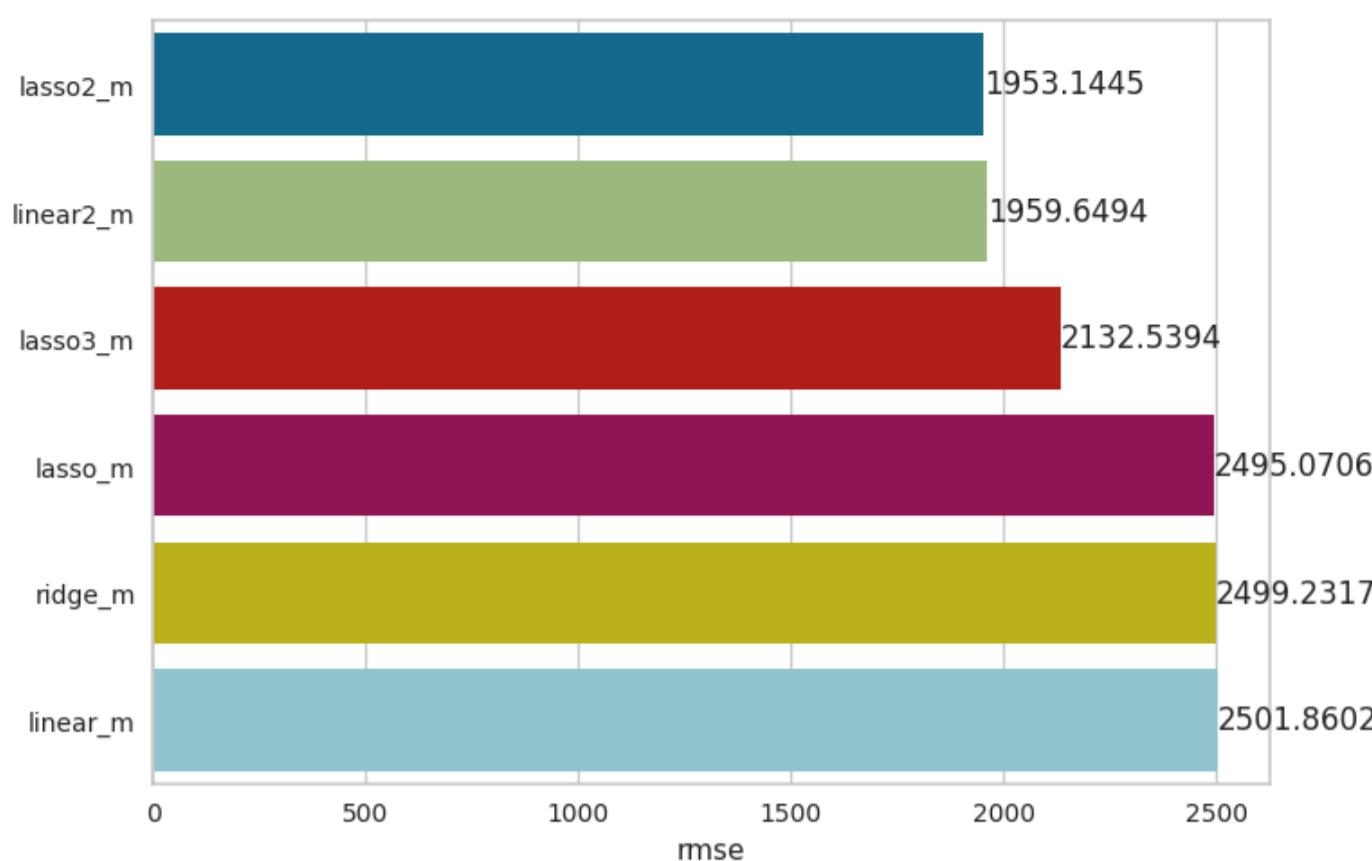
	r2_score	mae	rmse
linear_m	0.884	1725.509	2501.860
linear2_m	0.896	1415.381	1959.649
ridge_m	0.884	1722.885	2499.232
lasso_m	0.885	1721.928	2495.071
lasso2_m	0.896	1408.956	1953.144
lasso3_m	0.877	1553.998	2132.539

In [204...]

```
#metrics = scores.columns

for i, j in enumerate(scores):
    plt.figure(i)
    if j == "r2_score":
        ascending = False
    else:
        ascending = True
    compare = scores.sort_values(by=j, ascending=ascending)
    ax = sns.barplot(x = compare[j] , y= compare.index)
    for p in ax.patches:
        width = p.get_width() # get bar length
        ax.text(width, # set the text at 1 unit right of the bar
                p.get_y() + p.get_height() / 2, # get Y coordinate + X coordinate / 2
                '{:.4f}'.format(width), # set variable to display, 2 decimals
                ha = 'left', # horizontal alignment
                va = 'center')
```





#### Conclusion:

- The lasso2\_m model, which excludes outliers and applies Lasso regularization, consistently performs the best across all metrics ( $R^2$ , MAE, MSE), making it the most effective model.
  - The final\_m model, which includes feature selection, also shows good performance, particularly in MAE and MSE, but does not outperform lasso2\_m.
  - Regularization techniques like Lasso and Ridge improve model performance compared to the simple linear model, but removing outliers significantly enhances the results.
- Based on these metrics, the lasso2\_m model is recommended for its superior accuracy and lower error rates.**

## Final Model

- As selected, the Lasso2 model will be used for the final model where we removed the outliers and applied Lasso regularization alpha=0.01

## Model

```
In [91]: # Excluding rows where the price is greater than 35,000 using df_final
print(len(df[df.price > 35000]))

df2 = df[~(df.price > 35000)]
```

473

```
In [92]: df2.shape
```

```
Out[92]: (13768, 133)
```

```
In [93]: #Split the data

X = df2.drop(columns = "price")
y = df2.price
```

```
In [94]: # Set the model and scaler with pipeline

operations = [('scaler', MinMaxScaler()), ('lasso', Lasso(alpha=1.02))]
final_model = Pipeline(steps=operations)
final_model.fit(X,y)
```

```
Out[94]: Pipeline
         |
         +--> MinMaxScaler
         |
         +--> Lasso
```

```
In [95]: import pickle

# To export the model
pickle.dump(final_model, open('AutoScout24', 'wb'))
```

```
In [96]: final_model = pickle.load(open('AutoScout24', 'rb'))
```

## Predict New Observations

### Creating new samples for prediction

```
In [97]: # Get the columns of the original dataset
columns = df2.drop(columns=['price']).columns

# Generate random values (using normal distribution)
n_samples = 5
n_features = len(columns)
random_data = np.random.normal(loc=0, scale=1, size=(n_samples, n_features))

# Convert the new random data sample to a DataFrame
new_sample = pd.DataFrame(random_data, columns=columns)

print(new_sample.shape)
new_sample.head()
```

(5, 132)

Out[97]:

	km	gears_num	age	previous_owners	hp_kw	inspection_new	displacement_cc	weight_kg	fuel_cons_comb	cc_Air conditioning	cc_Air suspension	cc_Armrest	
0	0.296	1.050	-1.339		-0.306	-0.132	1.263	-2.719	-0.722	-1.255	-0.375	-0.020	-0.828
1	0.197	1.646	-1.348		0.242	0.041	1.475	1.072	0.381	0.538	0.064	1.973	1.406
2	-0.651	-0.798	0.123		-1.315	0.029	-0.603	-1.799	-0.411	-1.713	1.751	-1.649	1.237
3	0.470	1.441	0.368		1.032	-0.175	0.694	-0.526	1.558	1.243	-0.327	-2.118	0.798
4	0.788	-0.072	1.740		-2.168	-0.597	2.094	1.436	-0.923	-0.838	0.012	0.157	1.753

◀ ▶

In [98]: # Scale the new sample data using the pre-fitted MinMaxScaler

new\_sample\_scaled = final\_model.named\_steps['scaler'].transform(new\_sample)

In [99]: # Prediction

```
y_pred = final_model.predict(new_sample_scaled)

y_pred_df = pd.DataFrame(y_pred, columns=['Predicted Prices'])
y_pred_df
```

Out[99]: Predicted Prices

0	10720.644
1	-1529.746
2	5049.185
3	19688.395
4	27830.784

Thank you...

Duygu Jones | Data Scientist | 2024

Follow me: [duygujones.com](http://duygujones.com) | [Linkedin](#) | [GitHub](#) | [Kaggle](#) | [Medium](#) | [Tableau](#)