CAR PRICE PREDICTION USING REGRESSION MODELS

Prediction with Linear Regression + Ridge Regression + Lasso Regression + Elastic-Net

Kubra Uen | Data Scientist | August 2024

Project Introduction

This project seeks to forecast car prices utilizing an assortment of regression algorithms on a dataset sourced from AutoScout24. By performing Exploratory Data Analysis (EDA) and deploying regression models, the aim is to construct precise and reliable prediction models for car prices. This initiative will enhance comprehension of regression techniques and improve the accuracy of price forecasts.

Project Goals

- Comprehend Dataset and Features
- Data Cleaning and Preparation
- Apply Regression Models
- Enhance Model Performance
- Evaluate and Compare Models

Dataset Summary

The dataset sourced from **AutoScout24** includes a range of features for **9 distinct car models**. It consists of **5,000 rows** and **23 columns**, providing comprehensive information such as make, model, price, mileage, and more. This cleaned and pre-processed dataset will be used to apply regression algorithms for predicting car prices.

Inputs

1. **make_model:** Contains the make and model information of the vehicles.
2. **body_type:** Indicates the body type of the vehicles (e.g., sedan, hatchback).
3. **price:** Contains the price information of the vehicles.
4. **vat:** Contains value-added tax (VAT) information (often important in vehicle sales).
5. **km:** Contains the mileage information of the vehicles.
6. **Type:** Indicates the type of the vehicle (e.g., new, used).
7. **Fuel:** Indicates the type of fuel used (e.g., petrol, diesel).
8. **Gears:** Contains the number of gears in the vehicles.
9. **Comfort_Convenience:** Contains the comfort and convenience features of the vehicles.
10. **Entertainment_Media:** Contains entertainment and media features.
11. **Extras:** Contains the extra features of the vehicles.
12. **Safety_Security:** Contains safety features.
13. **age**: The age of the vehicles.
14. **Previous_Owners:** Contains the number of previous owners of the vehicles.
15. **hp_kW:** Contains the horsepower of the vehicles in kilowatts.
16. **Inspection_new:** Contains information about whether the vehicle has a new inspection.
17. **Paint_Type:** Contains the type of paint of the vehicles.
18. **Upholstery_type**: Contains the type of upholstery in the vehicles.
19. **Gearing_Type:** Indicates the type of transmission in the vehicles.
20. **Displacement_cc:** Indicates the engine displacement in cubic centimeters (cc).
21. **Weight_kg:** Indicates the weight of the vehicles in kilograms.
22. **Drive_chain:** Indicates the type of drive train in the vehicles (e.g., front-wheel drive, rear-wheel drive).
23. **cons_comb:** Indicates the combined fuel consumption of the vehicles in liters per 100 kilometers.

Methodology

Regression Algorithms

- Linear Regression
- Ridge Regression
- Lasso Regression
- Elastic-Net

Model Evaluation

- Regression error metrics
- Cross-validation methods
- Hyperparameter tuning

Feature Importance

- Identifying and focusing on important features

Results and Discussion

Model Comparison

- Performance comparison of different algorithms
- Highlighting the model with the highest prediction accuracy

Conclusion

Future Work

- Further analysis on the most successful algorithm to improve predictive capabilities

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split, cross_validate, cross_val_score
from sklearn.pipeline import Pipeline

import warnings
warnings.filterwarnings('ignore')

plt.rcParams["figure.figsize"] = (10,6)   # default: (6.4, 4.8)
pd.set_option('display.max_columns', 500) # None: sınırsız, default:20
pd.set_option('display.max_rows', 500)    # None: sınırsız, default:10
pd.options.display.float_format = '{:.3f}'.format
```

executed in 5.44s, finished 19:48:46 2024-08-12

2 EDA

In [2]:
```
df0 = pd.read_csv("final_scout_not_dummy.csv")
df = df0.copy()
```

executed in 200ms, finished 19:48:46 2024-08-12

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

executed in 25ms, finished 19:48:46 2024-08-12

Out[3]:

	make_model	body_type	price	vat	km	Type	Fuel	Gears	Comfort_Convenience	Entertainment_Media	Extras	Safety_Sec
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	ABS,Ce lock,Day run lights
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,Voic...	ABS,Ce lock,Ce door lock wi
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	ABS,Ce lock,Day run lights
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	ABS,A system,Ce door lock rem
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...	ABS,Ce door lock,D airbag,El

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

executed in 32ms, finished 19:48:46 2024-08-12

```
<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
```

In [5]:
```
duplicate_rows = df.duplicated()

# Counting the number of duplicate rows
duplicate_count = duplicate_rows.sum()
```

executed in 47ms, finished 19:48:46 2024-08-12

In [6]:
```
duplicate_count
```

executed in 5ms, finished 19:48:46 2024-08-12

Out[6]: 1673

In [7]:
```
# Checks for duplicate observations in the dataset and removes 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.")
```

executed in 4ms, finished 19:48:46 2024-08-12

In [8]:
```
duplicate_values(df)
```

executed in 88ms, finished 19:48:46 2024-08-12

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

In [9]:
```
df.describe().T

# We are getting to know the data

# If the standard deviation (std) is greater than or close to the mean,
#it suggests that there might be an outlier problem in our dataset overall.

# If there is a significant gap between the "min" and the first 25th percentile(Q1) and/or between the 75th percentile
#(Q3) and the max, it suggests that there might be a limited number of outlier values in the dataset, even if not overall.
```

executed in 29ms, finished 19:48:46 2024-08-12

Out[9]:

	count	mean	std	min	25%	50%	75%	max
price	14242.000	18100.969	7421.214	4950.000	12950.000	16950.000	21900.000	74600.000
km	14242.000	32582.110	36856.863	0.000	3898.000	21000.000	47000.000	317000.000
Gears	14242.000	5.940	0.703	5.000	5.000	6.000	6.000	8.000
age	14242.000	1.415	1.110	0.000	0.000	1.000	2.000	3.000
Previous_Owners	14242.000	1.041	0.337	0.000	1.000	1.000	1.000	4.000
hp_kW	14242.000	88.713	26.548	40.000	66.000	85.000	103.000	294.000
Inspection_new	14242.000	0.256	0.437	0.000	0.000	0.000	1.000	1.000
Displacement_cc	14242.000	1432.890	277.507	890.000	1229.000	1461.000	1598.000	2967.000
Weight_kg	14242.000	1342.399	201.247	840.000	1165.000	1320.000	1487.000	2471.000
cons_comb	14242.000	4.825	0.862	3.000	4.100	4.800	5.400	9.100

In [10]:
```
df.columns
```

executed in 6ms, finished 19:48:46 2024-08-12

Out[10]:
```
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')
```

2.1 Feature Engineering

In [11]:
```
df.select_dtypes(include ="object").head()
```
executed in 14ms, finished 19:48:46 2024-08-12

Out[11]:

	make_model	body_type	vat	Type	Fuel	Comfort_Convenience	Entertainment_Media	Extras	Safety_Security	Paint_Type	Upholste
0	Audi A1	Sedans	VAT deductible	Used	Diesel	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...	Metallic	
1	Audi A1	Sedans	Price negotiable	Used	Benzine	Air conditioning,Automatic climate control,Hil...	Bluetooth,Hands-free equipment,On-board comput...	Alloy wheels,Sport seats,Sport suspension,Voic...	ABS,Central door lock,Central door lock with r...	Metallic	
2	Audi A1	Sedans	VAT deductible	Used	Diesel	Air conditioning,Cruise control,Electrical sid...	MP3,On-board computer	Alloy wheels,Voice Control	ABS,Central door lock,Daytime running lights,D...	Metallic	
3	Audi A1	Sedans	VAT deductible	Used	Diesel	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...	Metallic	
4	Audi A1	Sedans	VAT deductible	Used	Diesel	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...	Metallic	

2.2 Dummies control ?

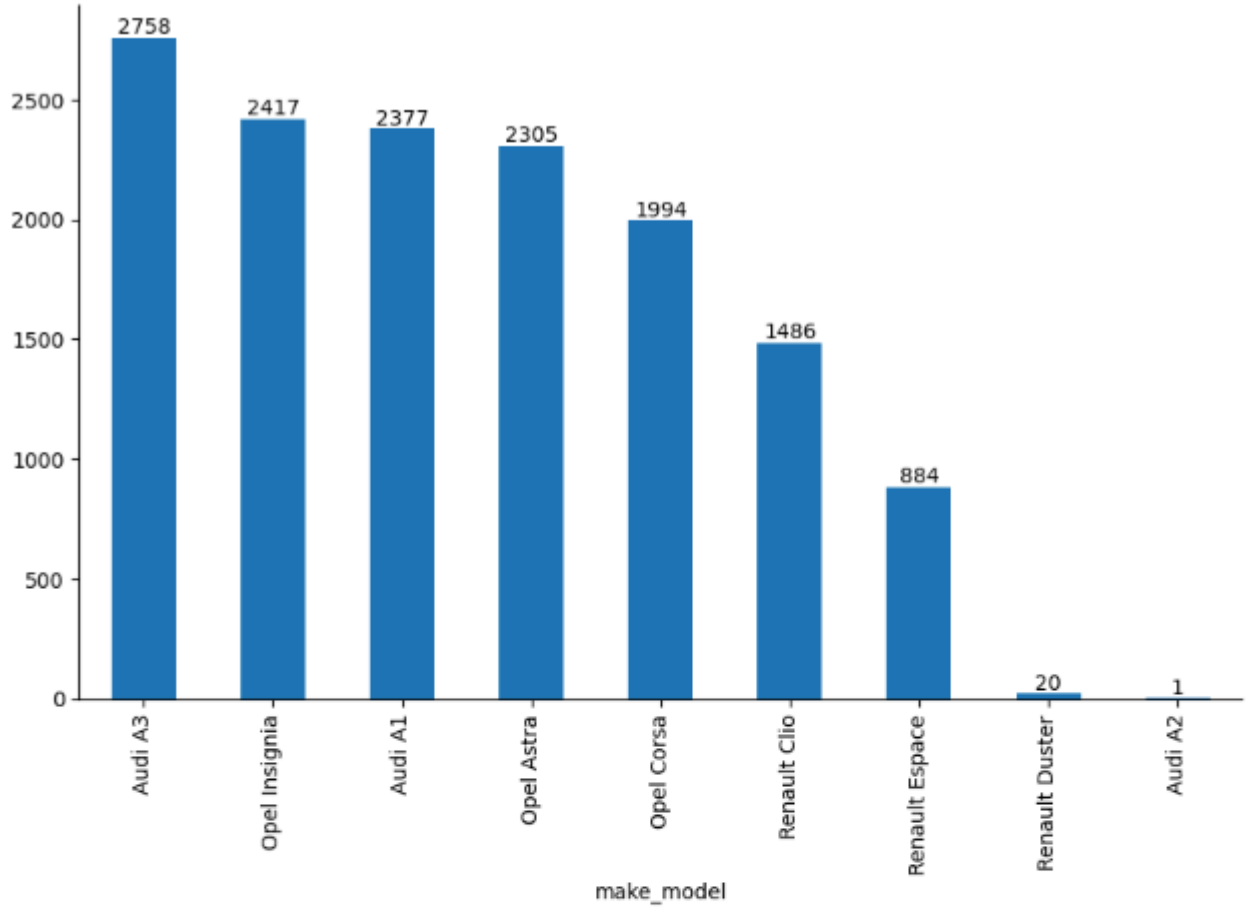
In [12]:
```
for col in df.select_dtypes('object'):
    print(f"{col:<20}:", df[col].nunique())

# A 20-character space is left from the leftmost side to the ":" sign if the length is less than 20,
#and feature names are written in this space.

# The ":" sign is aligned in the same position across all rows.
```
executed in 37ms, finished 19:48:46 2024-08-12

```
make_model          : 9
body_type           : 8
vat                 : 2
Type                : 5
Fuel                : 4
Comfort_Convenience : 6196
Entertainment_Media : 346
Extras              : 659
Safety_Security     : 4442
Paint_Type          : 3
Upholstery_type     : 2
Gearing_Type        : 3
Drive_chain         : 3
```

In [13]:
```
df.make_model.value_counts()
```
executed in 7ms, finished 19:48:46 2024-08-12

```
Opel Corsa        1994
Renault Clio      1486
Renault Espace     884
Renault Duster      20
Audi A2              1
Name: count, dtype: int64
```

In [14]:
```
# This code counts the number of unique values in the "make_model" column of our DataFrame
# and visualizes these counts in a bar chart.


ax = df.make_model.value_counts().plot(kind ="bar")
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
#ax.axis("off")

ax.bar_label(ax.containers[0]);

# for p in ax.patches:
#     ax.annotate(str(p.get_height()), (p.get_x() * 1.03, p.get_height() * 1.03))
```
executed in 375ms, finished 19:48:47 2024-08-12



In [15]:
```
df[df.make_model=="Audi A2"]
```
executed in 18ms, finished 19:48:47 2024-08-12

Out[15]:

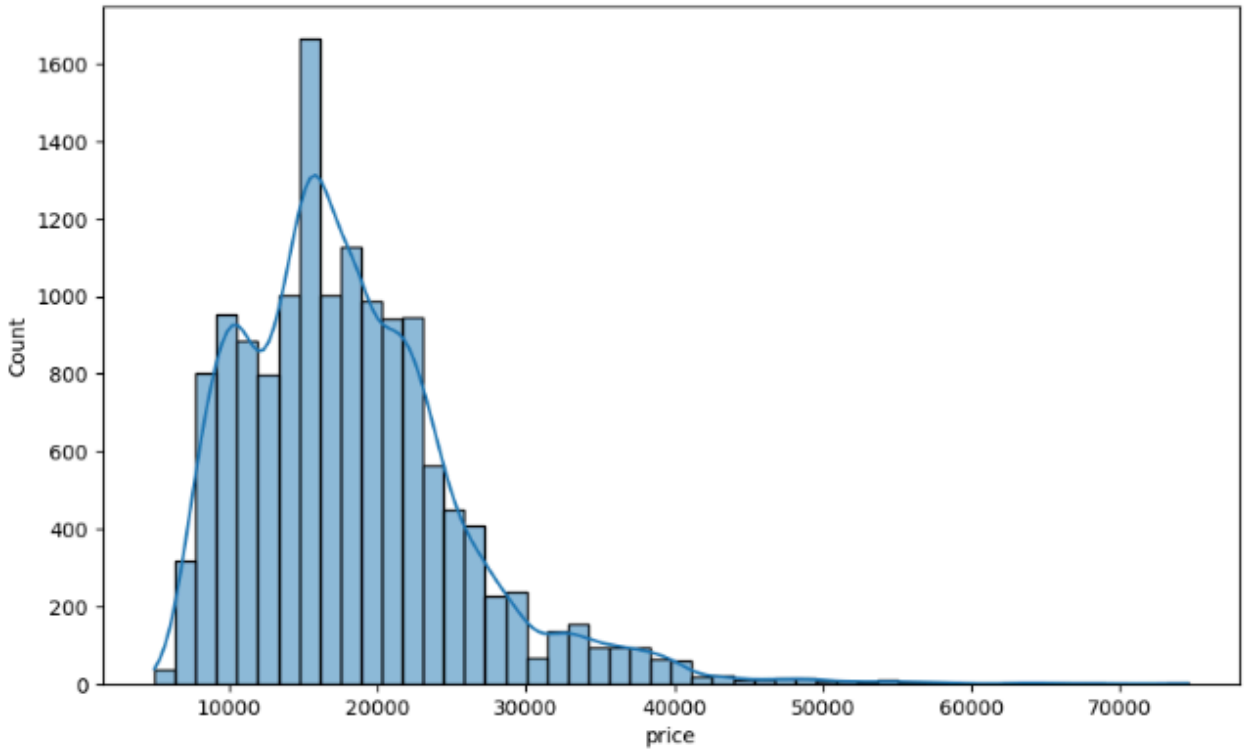
	make_model	body_type	price	vat	km	Type	Fuel	Gears	Comfort_Convenience	Entertainment_Media	Extras	Safety
2614	Audi A2	Off-Road	28200	VAT deductible	26166.000	Employee's car	Diesel	6.000	Air conditioning,Armrest,Automatic climate con...	Bluetooth,CD player,Hands-free equipment,MP3,O...	Alloy wheels	ABS Cont d

In [16]:
```
df.drop(index=[2614], inplace =True)
```
executed in 9ms, finished 19:48:47 2024-08-12

In [17]:
```
df.shape
```
executed in 6ms, finished 19:48:47 2024-08-12

Out[17]: (14241, 23)

In [18]:
```
# Since linear models are very sensitive to outliers, I am trying to detect outliers in the data using a histplot.
# Let's not forget that an outlier in machine learning means that the data belonging to a particular group or groups
# may be insufficient for training. By looking at the visual below, we see that the number of vehicles priced above
#40,000 EURO is very low.


# As the first insight from this visual, we can evaluate that the number of vehicles priced above 40,000 EURO may be
#insufficient for training.

# However, we cannot determine if it is indeed insufficient without performing the training.
# We should make a decision by training the model both with and without the observations we consider as outliers, and then com

# Additionally, looking at the histplot for the entire dataset might mislead us regarding outliers.
# For better outlier detection, we should group the data (e.g., Audi A3, Audi A1, Renault Clio, etc.) and detect outliers acc


sns.histplot(df.price, bins=50, kde=True);
```
executed in 384ms, finished 19:48:47 2024-08-12



3 IMPORTANT

Linear models are sensitive to outliers because these models aim to find a line (or a plane or hyperplane in higher-dimensional spaces) that best fits the data points. Outliers can significantly affect the model's goal of achieving this "best fit." Here are some reasons why:

1. **MSE-Based Loss Functions:** Many linear models, like linear regression, use a mean squared error (MSE) based loss function. MSE amplifies large errors by squaring them, so even a single outlier can significantly increase the total error.
2. **Low Robustness:** In linear models, each data point contributes to training the model. However, an outlier can have a much larger impact compared to all other values. This can cause the model to fit excessively to this outlier.
3. **Linear Assumption:** Linear models assume that the relationship between variables is linear. This assumption makes the model particularly sensitive to outliers, as these outliers often disrupt the assumed linear relationship.
4. **Leverage Effect:** Outliers can significantly alter the slope (or gradient) of a linear model, leading to misleading predictions for all other data points.

In many applications, the sensitivity of linear models to outliers can reduce the model's generalization capability and reliability. Therefore, it's generally recommended to detect and appropriately handle outliers before performing linear modeling.

Many statistical and machine learning models are sensitive to outliers. However, this sensitivity varies depending on the model's characteristics and the optimization techniques used.

Models Sensitive to Outliers:

1. **Linear Regression:** As mentioned earlier, linear regression is sensitive to outliers.
2. **Logistic Regression:** Logistic regression is a type of linear regression and can be sensitive to outliers for similar reasons.
3. **K-Nearest Neighbors (KNN):** The KNN algorithm can be affected by outliers when the value of k is small.
4. **Univariate Time Series Models:** Models like ARIMA can be sensitive to outliers.

Models Not Sensitive to Outliers:

1. **Decision Tree & Random Forest:** These models work by splitting data points and are generally resistant to outliers.
2. **Support Vector Machines** (SVM) with RBF Kernel: SVM with RBF kernel can be resistant to outliers.
3. **Ensemble Methods:** Ensemble methods like Boosting and Bagging, which combine multiple models, are generally more resistant to outliers.

Summary

A model's sensitivity to outliers depends on its structural characteristics and the optimization techniques it uses. Linear models are typically sensitive to outliers because they search for linear relationships among data points, and loss functions like MSE amplify large errors. Some models (e.g., decision trees, robust regression) are more resistant to outliers and are less affected by such data. Therefore, when selecting a model, the characteristics of the dataset and the presence of outliers should be considered.

In [19]:
```
df_numeric = df.select_dtypes(include ="number")
df_numeric

# With the select_dtypes(exclude="number") code, we can filter only the object features in the DataFrame (df).
```

executed in 18ms, finished 19:48:47 2024-08-12

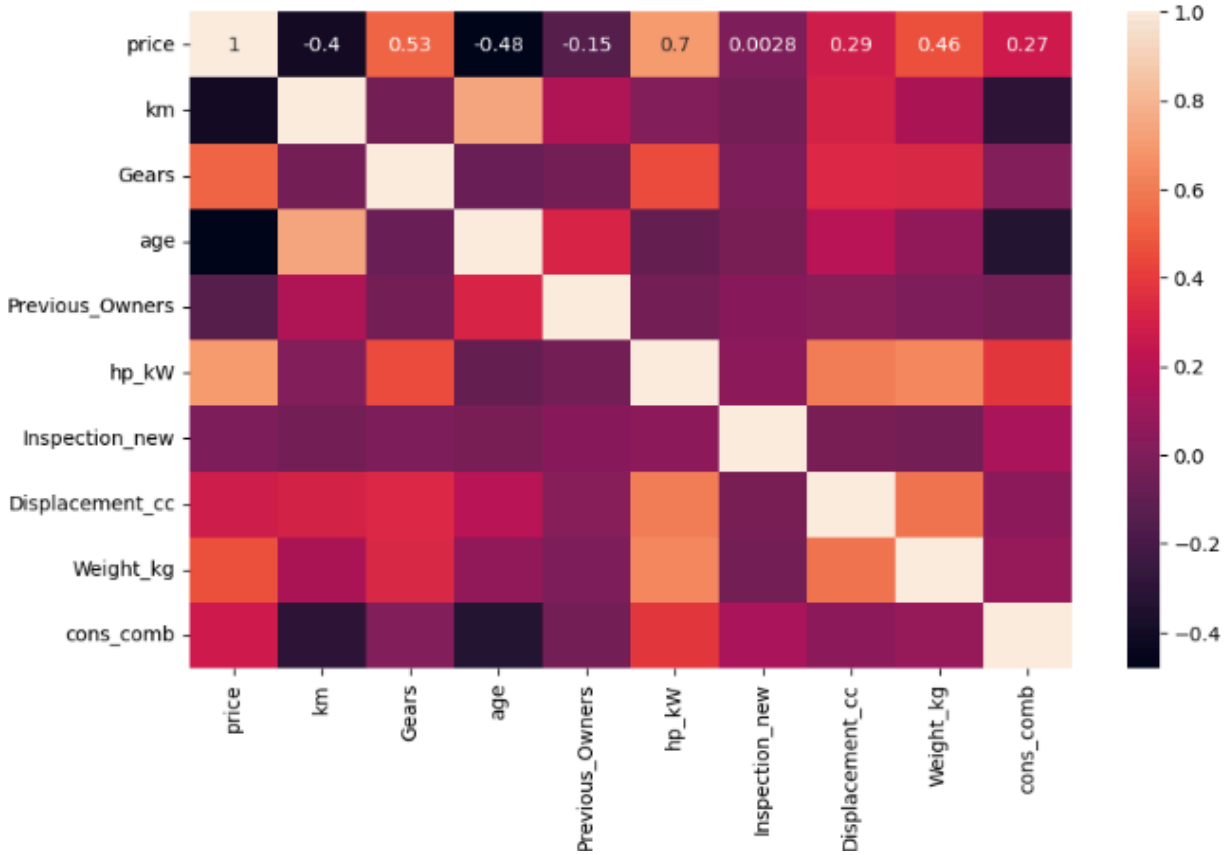
Out[19]:

	price	km	Gears	age	Previous_Owners	hp_kW	Inspection_new	Displacement_cc	Weight_kg	cons_comb
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
3	14500	73000.000	6.000	3.000	1.000	66.000	0	1422.000	1195.000	3.800
4	16790	16200.000	7.000	3.000	1.000	66.000	1	1422.000	1135.000	4.100
...	...	...	...	...	...	...	...	...	...	...
15907	39980	100.000	6.000	0.000	1.000	118.000	0	1598.000	1734.000	4.700
15908	39950	1647.363	6.000	0.000	1.000	147.000	0	1997.000	1758.000	5.300
15909	39950	1000.000	6.000	0.000	1.000	165.000	0	1798.000	1734.000	6.800
15911	39885	9900.000	7.000	0.000	1.000	165.000	0	1798.000	1708.000	7.400
15912	39875	15.000	6.000	0.000	1.000	146.000	1	1997.000	1734.000	5.300

14241 rows × 10 columns

In [20]:
```
sns.heatmap(df_numeric.corr(), annot =True);
```

executed in 329ms, finished 19:48:47 2024-08-12



3.1 Multicollinearity control

Multicollinearity is a problem that arises in gradient descent-based models like linear and logistic regression.

Multicollinearity occurs when there is a high correlation between independent variables. This situation can reduce the importance of other independent variables in the model when one independent variable is strongly related to the others.

Multicollinearity makes it difficult to accurately interpret the model. It is hard to correctly estimate the model coefficients, which can lead to misinterpretation of the effects of some variables. Therefore, the issue of multicollinearity must be resolved to obtain accurate results.

Regularization algorithms address the problem of multicollinearity just as they address the issue of overfitting.

Multicollinearity describes a situation where there is a high degree of linear relationship between independent variables. Multicollinearity can be problematic for some statistical models, while it may not be as critical for others.

Models Sensitive to Multicollinearity:

1. **Linear Regression:** The linear regression model can be significantly affected when there is high correlation (multicollinearity) among independent variables. This can cause instability in the estimates of the regression coefficients and lead to misleading results, as the model struggles to distinguish the effects of the independent variables from one another.
2. **Logistic Regression:** Logistic regression, being a type of linear model, can also be affected by multicollinearity in a similar way, where high correlation among independent variables reduces the reliability of parameter estimates.
3. **Ridge and Lasso Regression:** Ridge and Lasso regressions are regularization techniques that provide resilience against the problem of multicollinearity. However, the effectiveness of these techniques depends on the correct selection of the lambda (regularization parameter) value. The selection of lambda has a significant impact on model performance and is usually determined through cross-validation.

Models Not Sensitive to Multicollinearity:

1. **Decision Trees:** Decision trees and models based on decision trees (e.g., Random Forest) are not affected by relationships among independent variables because these models work based on splitting criteria and evaluate each independent variable separately.
2. **Support Vector Machines (SVM):** SVM, especially when the correct kernel function is selected, is not affected by high correlation among independent variables. The model has the ability to separate data points with an optimal hyperplane, and in this process, the correlation between variables does not pose a significant issue.
3. **Naive Bayes:** The Naive Bayes classifier does not rely on relationships among variables and operates under the assumption of independence, so it is not affected by multicollinearity.

Why Is Multicollinearity a Problem?

Interpretability of Coefficients: In the presence of multicollinearity, the high degree of correlation among independent variables makes it difficult to accurately interpret the effects of these variables on the model.

Instability of Predictions: Small changes in the data can cause large fluctuations in the model coefficients, reducing the reliability of the model's predictions.

Unreliability of Test Statistics: Multicollinearity can make the results of statistical tests misleading, such as producing deceptive p-values and R-squared values.

To address multicollinearity, methods such as variable selection, regularization techniques, or principal component regression can be used.

These methods can help reduce the negative effects of multicollinearity by improving the reliability of predictions and the interpretability of coefficients.

In [21]:
```
df_numeric.corr()[(df_numeric.corr()>= 0.9) & (df_numeric.corr() < 1)].any().any()

# We can check for multicollinearity for correlation values between +0.9 and +1 using this code.
```

executed in 21ms, finished 19:48:47 2024-08-12

Out[21]:
```
False
```

In [22]:
```
df_numeric.corr()[(df_numeric.corr()<= -0.9) & (df_numeric.corr() > -1)].any().any()

# We can check for multicollinearity for correlation values between -0.9 and -1 using this code.
```

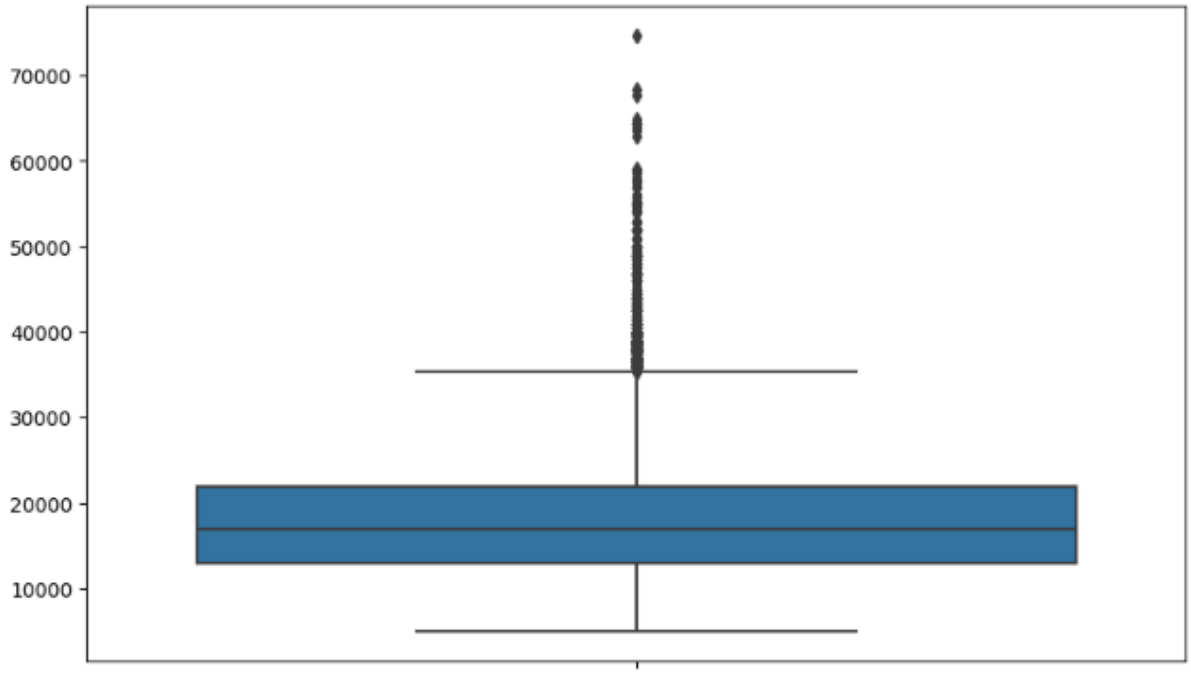
executed in 22ms, finished 19:48:47 2024-08-12

Out[22]:
```
False
```

NOTE: The any() function is used twice. The first any() checks across columns and returns whether there is any True value. The second any() checks these results across rows. In other words, it checks if there is any negative correlation value between -0.9 and -1 in the correlation matrix.

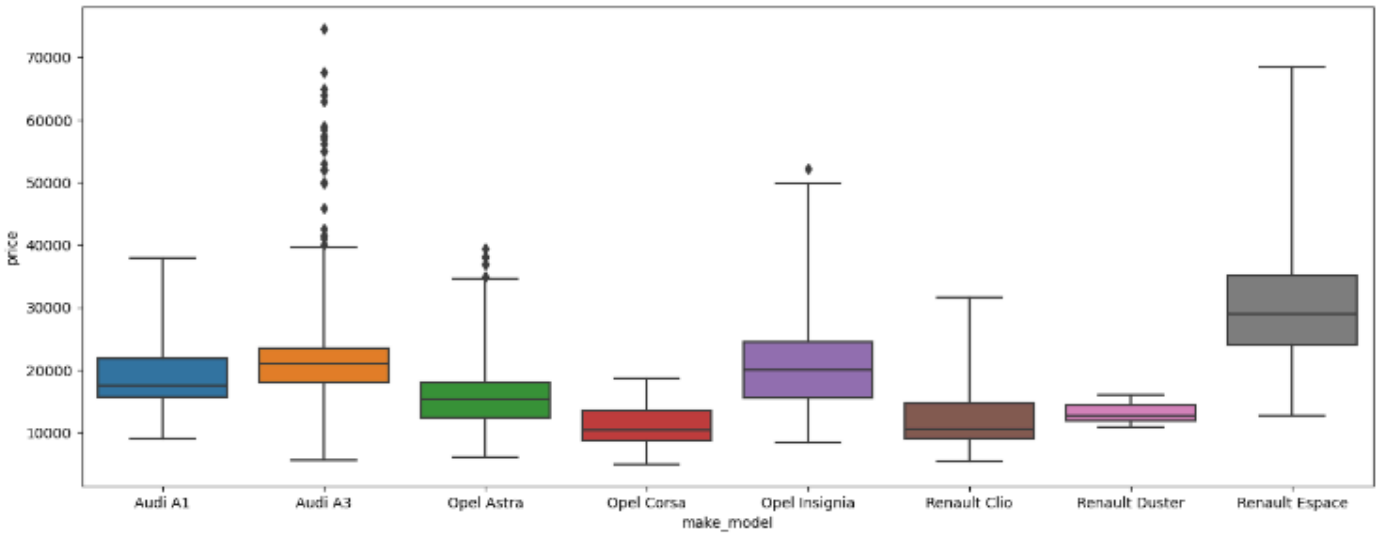
In [23]:
```
sns.boxplot(df.price);

# We can also gain insights about outliers through boxplots. However, these insights can be misleading.
# Therefore, we will make our evaluation not for the entire dataset, but
# based on groups within the data (e.g., Audi A1, Audi A3, Renault Clio, etc.).
```

executed in 123ms, finished 19:48:48 2024-08-12

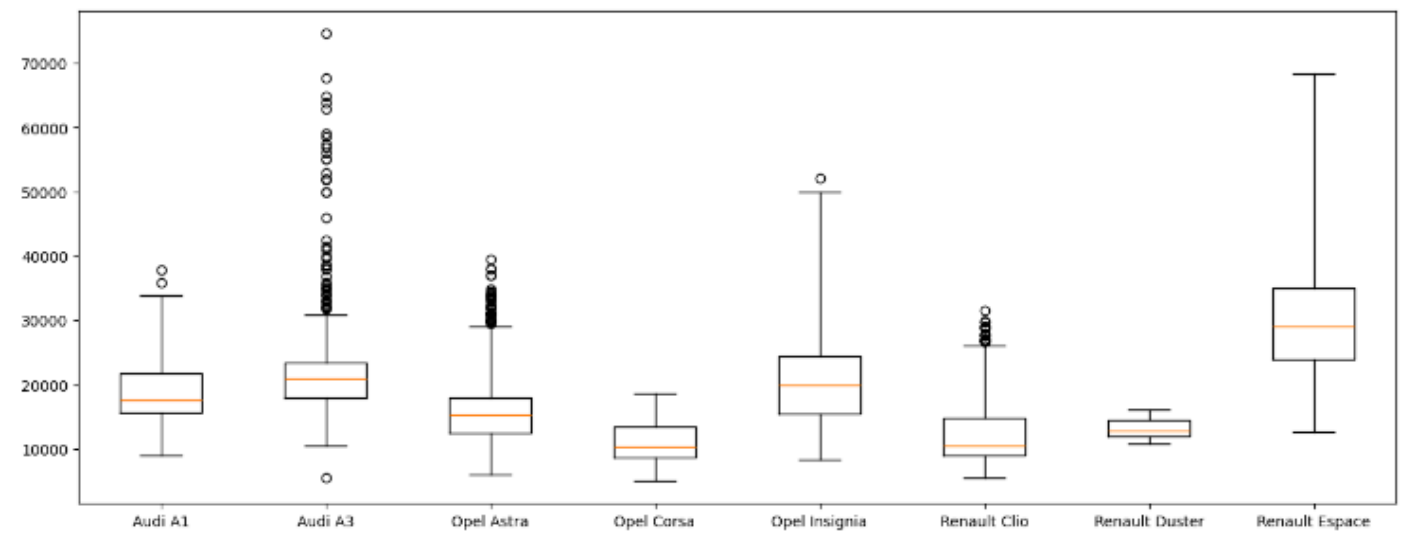


In [24]:
```
plt.figure(figsize=(16,6))
sns.boxplot(x="make_model", y="price", data=df, whis=3)
plt.show()

# When we look at the visuals below, according to the boxplot, we can see observations that could be outliers for Audi A3, Ope
# In your own data, you can determine separate whisker values for each group according to these visuals.

# The following steps are used to calculate the IQR:

# The data is sorted from smallest to largest.
# The first and third quartiles, representing 25% and 75% of the data, are calculated.
# The IQR is obtained by subtracting the first quartile from the third quartile.

# Q1 = df.groupby('make_model')['price'].quantile(0.25)
# Q3 = df.groupby('make_model')['price'].quantile(0.75)
# IQR = Q3-Q1
# lower_lim = Q1-1.5*IQR
# upper_lim = Q3+1.5*IQR
```

executed in 249ms, finished 19:48:48 2024-08-12

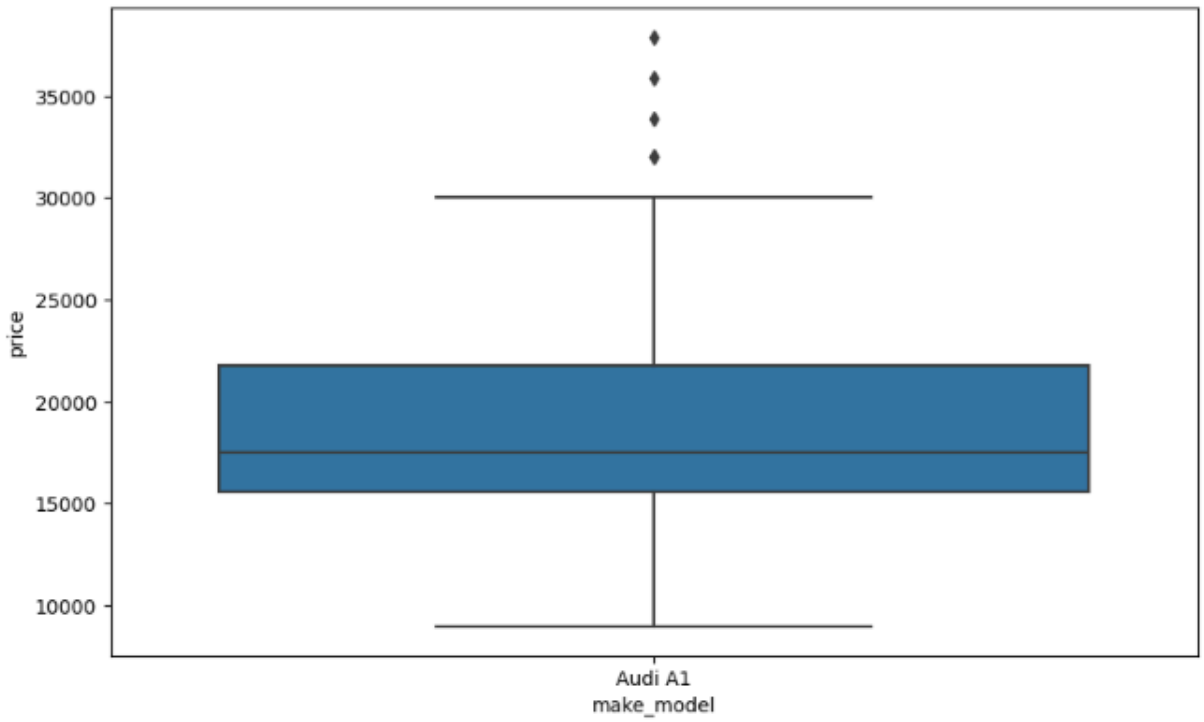


In [25]:
```
# Seaborn's boxplot function uses the same 'whis' value for all categories, so
# we will use matplotlib's boxplot function to plot each category separately.

whisker_values = {
    'Audi A1': 2.0,
    'Audi A3': 1.5,
    'Opel Astra': 2.0,
    'Opel Corsa': 2.5,
    'Opel Insignia': 3.0,
    'Renault Clio': 2.0,
    'Renault Duster': 1.5,
    'Renault Espace': 3.0
}

plt.figure(figsize=(16,6))

for i, make_model in enumerate(whisker_values.keys()):
    model_data = df[df['make_model'] == make_model]['price']
    plt.boxplot(model_data, positions=[i], whis=whisker_values[make_model], widths=0.5)
```



In [26]:
```
sns.boxplot(x="make_model", y="price", data=df[df["make_model"]== "Audi A1"], whis=1.5)
plt.show()
```

executed in 136ms, finished 19:48:48 2024-08-12



In [27]:
```
df[df["make_model"]== "Audi A1"]["price"]

# Prices of Audi A1s. We will try to detect outlier values for each car using this code.
```

executed in 8ms, finished 19:48:48 2024-08-12

Out[27]:
```
0       15770
1       14500
2       14640
3       14500
4       16790
        ...
2609    21800
2610    21789
2611    21750
2612    21760
2613    21750
Name: price, Length: 2377, dtype: int64
```

In [28]:
```
# We are identifying observations that we evaluate as potential outliers for each group based on the 1.5 whisker value.

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))
```

executed in 49ms, finished 19:48:48 2024-08-12

```
 The count of outlier for Audi A1         : 5    ,          The rate of outliers : 0.002
 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
```

3.2 Get dummies

The **get_dummies()** method transforms categorical columns into new columns using binary encoding.

Each category is encoded as a feature, where the feature is coded as 1 if present, and 0 if absent.

In [29]:
```
# This code parses the comma-separated text in each cell of our columns
# and creates a new DataFrame where each element of the text is represented as a separate column.

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_"))
```
executed in 890ms, finished 19:48:49 2024-08-12

In [30]:
```
df.drop(["Comfort_Convenience","Entertainment_Media","Extras","Safety_Security"], axis=1, inplace=True)
```
executed in 17ms, finished 19:48:49 2024-08-12

In [31]:
```
df = pd.get_dummies(df, drop_first =True)

# The drop_first=True parameter removes the column for the first category of each feature.
# This is done to prevent overfitting and multicollinearity.

# For example, if the "Fuel_Type" column has three categories like "Diesel," "Gasoline," and "LPG," only two columns
# (e.g., "Gasoline" and "LPG") will be encoded, and "Diesel" will be used as a reference.
# This way, the "Gasoline" and "LPG" columns are related to the "Diesel" column, helping to prevent any issues with multicolli
# Due to differences in Pandas versions, True/False may be displayed as 0/1.
```
executed in 52ms, finished 19:48:49 2024-08-12

In [32]:
```
# Converts the boolean values in the DataFrame to integers

bool_columns = df.columns[df.dtypes == 'bool']
df[bool_columns] = df[bool_columns].astype(int)
```
executed in 16ms, finished 19:48:49 2024-08-12

In [33]:
```
df.head()
```
executed in 63ms, finished 19:48:49 2024-08-12

Out[33]:

	price	km	Gears	age	Previous_Owners	hp_kW	Inspection_new	Displacement_cc	Weight_kg	cons_comb	cc_Air conditioning	cc_Air suspension	cc_Armrest
0	15770	56013.000	7.000	3.000	2.000	66.000	1	1422.000	1220.000	3.800	1	0	1
1	14500	80000.000	7.000	2.000	1.000	141.000	0	1798.000	1255.000	5.600	1	0	0
2	14640	83450.000	7.000	3.000	1.000	85.000	0	1598.000	1135.000	3.800	1	0	0
3	14500	73000.000	6.000	3.000	1.000	66.000	0	1422.000	1195.000	3.800	0	1	1
4	16790	16200.000	7.000	3.000	1.000	66.000	1	1422.000	1135.000	4.100	1	0	1

In [34]:
```
df.shape
```
executed in 7ms, finished 19:48:49 2024-08-12

Out[34]: (14241, 133)

In [35]:
```
df.isnull().any().any()
```
executed in 8ms, finished 19:48:49 2024-08-12

Out[35]: False

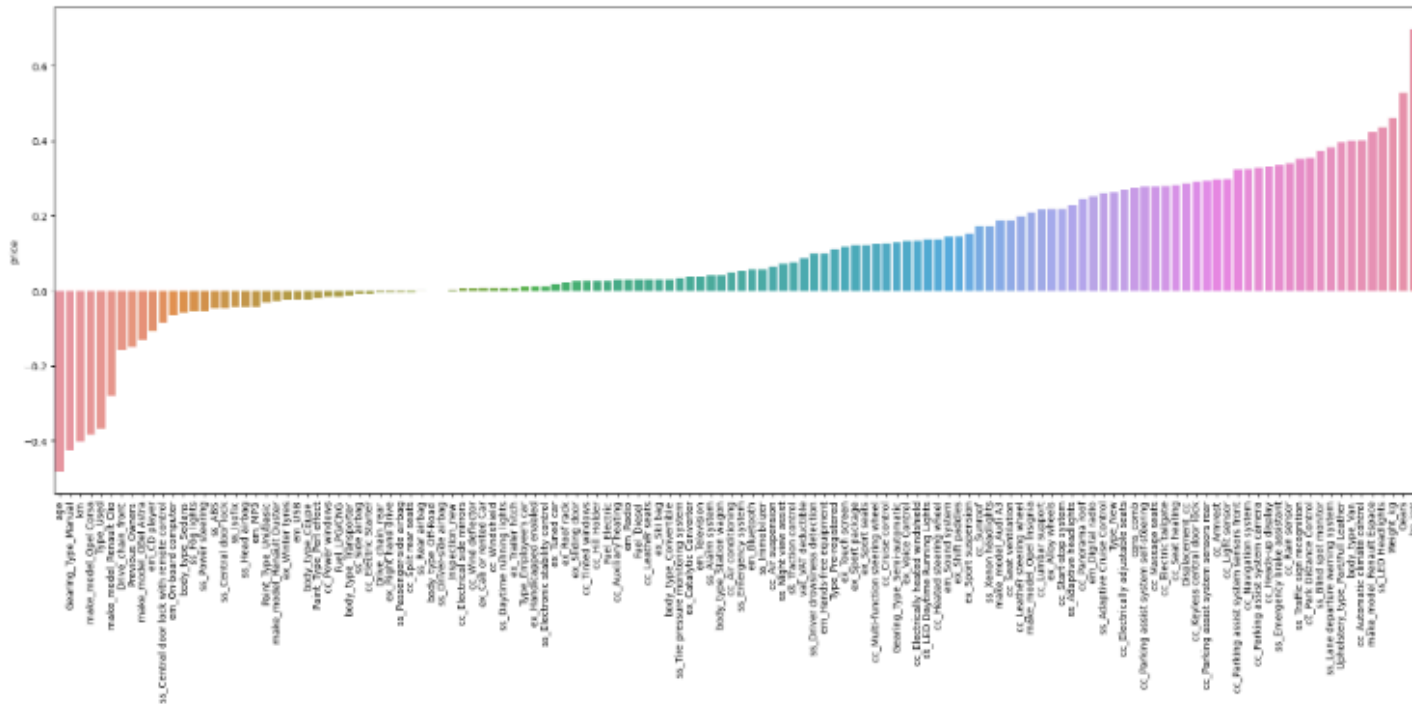
In [36]:
```
corr_by_price = df.corr()["price"].sort_values()[:-1]
corr_by_price

# We are examining the correlations of all features in our dataset with the target.
# We sort the correlations with our target, which is price, from smallest to largest.
# Since we don't want to see the correlation of the target with itself, we slice ([:-1]) to ignore the target using -1.
```
executed in 645ms, finished 19:48:50 2024-08-12

Out[36]:
```
age                                    -0.481
Gearing_Type_Manual                    -0.424
km                                     -0.402
make_model_Opel Corsa                  -0.384
Type_Used                              -0.368
make_model_Renault Clio                -0.281
Drive_chain_front                      -0.159
Previous_Owners                        -0.149
make_model_Opel Astra                  -0.130
em_CD player                           -0.107
ss_Central door lock with remote control   -0.084
em_On-board computer                   -0.067
```

In [37]:

```
plt.figure(figsize = (20,10))
sns.barplot(x = corr_by_price.index, y = corr_by_price)
plt.xticks(rotation=90)
plt.tight_layout();

# We are visualizing the correlations between the features and the target.
```

executed in 1.81s, finished 19:48:52 2024-08-12



4 Train | Test Split

random_state=42

test size = 0.2

Data Preparation and Model Training

1. **Organizing the Dataset:**

Before starting to train the model, the dataset needs to be organized. This may include steps like data cleaning, feature engineering, and handling outliers.

2. **Splitting into Training and Test Sets:**

The dataset is typically split into training and test sets. The training set is used for the model to learn, while the test set is used to evaluate the model's performance.

3. **Separating the Target Variable:**

The target variable (label) you are trying to predict should be separated from the features. The model will attempt to predict this target variable.

Performance Evaluation

1. **Performance Metrics:**

Various performance metrics can be used for regression models, such as Mean Absolute Error (MAE), Mean Squared Error (MSE), and R-squared score.

2. **Defining a Function:**

By defining a function that calculates different performance metrics for the model, you can easily evaluate and compare these metrics.

3. **Using Cross-Validation:**

Cross-validation is used to evaluate the model's performance on different samples of data. The model is trained on different subsets of the training set, and the performance of each subset is evaluated. The average of these scores gives the overall performance of the model.

Summary

The process of data preparation, splitting the dataset into training and test sets, determining the target variable, and training the model are fundamental steps in machine learning projects. Various metrics and cross-validation methods are used to evaluate the model's performance. These approaches help you assess the model's generalization ability more accurately.

In [38]:
```
df.head()
```
executed in 50ms, finished 19:48:52 2024-08-12

Out[38]:

	price	km	Gears	age	Previous_Owners	hp_kW	Inspection_new	Displacement_cc	Weight_kg	cons_comb	cc_Air conditioning	cc_Air suspension	cc_Armrest
0	15770	56013.000	7.000	3.000	2.000	66.000	1	1422.000	1220.000	3.800	1	0	1
1	14500	80000.000	7.000	2.000	1.000	141.000	0	1798.000	1255.000	5.600	1	0	0
2	14640	83450.000	7.000	3.000	1.000	85.000	0	1598.000	1135.000	3.800	1	0	0
3	14500	73000.000	6.000	3.000	1.000	66.000	0	1422.000	1195.000	3.800	0	1	1
4	16790	16200.000	7.000	3.000	1.000	66.000	1	1422.000	1135.000	4.100	1	0	1

In [39]:
```
X= df.drop(columns="price")
y= df.price
```
executed in 14ms, finished 19:48:52 2024-08-12

In [40]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# To train the model and then make predictions, we split the data into train and test sets using the train_test_split function
# By default, test_size=0.25.
# Best practice is to choose values around 0.2, 0.25, or 0.3. The goal here is to provide as much data as possible to the trai
# Especially in small datasets, this ratio can be chosen around 0.1 or 0.15.

# With random_state, we randomly distribute the data into train and test sets,
# ensuring that the same data is sent to train and test sets each time.
```
executed in 36ms, finished 19:48:52 2024-08-12

In [41]:
```
X_train.shape
```
executed in 8ms, finished 19:48:52 2024-08-12

Out[41]: (11392, 132)

In [42]:
```
X_test.shape
```
executed in 7ms, finished 19:48:52 2024-08-12

Out[42]: (2849, 132)

5 Implement Linear Regression

Linear Regression

Predicts the dependent variable based on independent variables.

If there is a single independent variable, it is called simple linear regression; if there are multiple independent variables, it is called multiple linear regression.

In simple linear regression, if there is a meaningful relationship between the feature and the target, this data is suitable for linear regression.

Basic Assumptions of Linear Regression Analysis

1. **Assumption of a Linear Relationship:** It is assumed that the relationship between the dependent variable and independent variables can be expressed in a linear manner. That is, the relationship between the regression line and the variables should be linear. For example, as X increases, Y should also increase or decrease.
2. **Assumption of Independence:** It is assumed that observations are independent of each other. That is, the result of one observation should not affect the others.
3. **Assumption of Normal Distribution:** Error terms (residuals) should be normally distributed and should not form any pattern. This is necessary to ensure the reliability of the regression model's predictions.
4. **Independence of Independent Variables:** There should be no multicollinearity issue among the independent variables. That is, the independent variables should not be highly correlated with each other.

$\hat{Y} = b0 + b1X$

$\hat{Y}$ = predicted value

b0 = intercept (the point where the line cuts the y-axis)

b1 = slope = coefficient = weight

X = independent variable

Residual = Random error = e = $Y - \hat{Y}$

The important thing is to minimize the error.

The **Best fit** line is drawn in a way that minimizes our errors. When there is a single feature, it is found using the **Ordinary Least Squares** method, and when there are multiple features, it is found using **gradient descent.**

The **Cost - loss** function calculates the mean error by taking the square of the difference between the actual values and the predicted values.

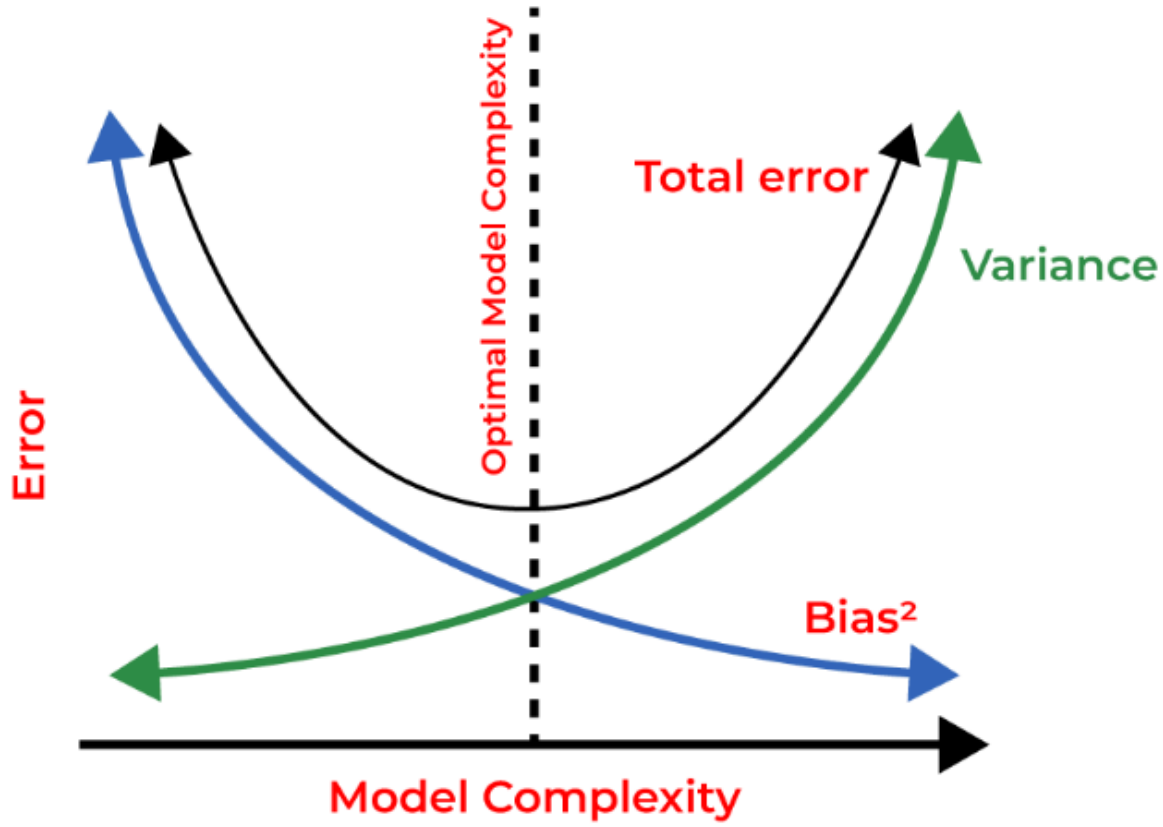
With the **Gradient Descent** optimization algorithm, what you will do is gradually change the w and b values to reduce the cost - loss function and try to bring it to its minimum value.

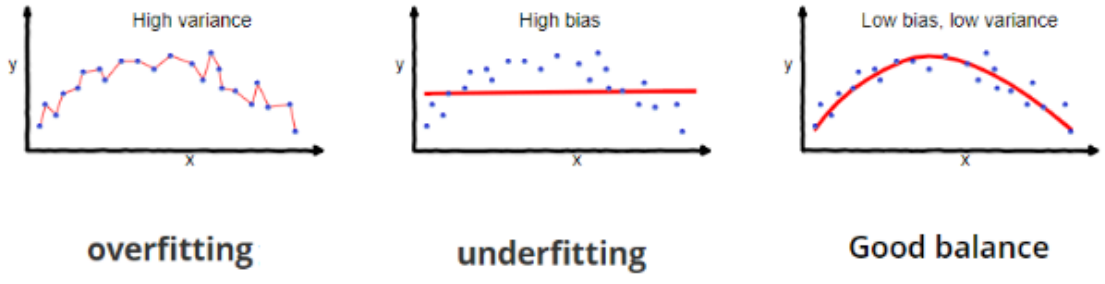
It is an algorithm that iteratively tries to minimize the error.

Bias is the systematic error in a model's predictions. The farther the model's predictions are from the actual values, the higher the bias. High bias can occur because the model is simple and fails to capture the complexity of the data. This results in underfitting.

Variance refers to how much the model's predictions vary across different data points. When trained multiple times on the same dataset, different results are obtained. High variance indicates that the model is overfitting the data and the patterns it has learned cannot be generalized to new data outside the dataset.

Bias-variance trade-off: It is important to strike a balance between bias and variance. Ideally, a model should have both low bias and low variance. This means the model can accurately capture the complexity of the data and make generalizations.





overfitting underfitting Good balance

5.1 Model

In [43]:
```
# We are defining our function to compare the metrics of the train and test sets.

def train_val(model, X_train, y_train, X_test, y_test):

    y_pred = model.predict(X_test)
    y_train_pred = model.predict(X_train)

    scores = {
    "train": {
    "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))},

    "test": {
    "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)
```

executed in 5ms, finished 19:48:52 2024-08-12

In [44]:
```
from sklearn.linear_model import LinearRegression # We are importing the LinearRegression algorithm.
```
executed in 364ms, finished 19:48:52 2024-08-12

In [45]:
```
lm = LinearRegression() # Defining the object


# We always assign our algorithm to a variable. Otherwise, we will encounter an error when making predictions.
```
executed in 4ms, finished 19:48:52 2024-08-12

In [46]:
```
lm.fit(X_train,y_train) # veriyi vererek modeli linear regressiona gore egit

# If we write LinearRegression() instead of lm here, the code will still work. However, if you use LinearRegression()
# directly when making predictions below, you will encounter an error.
# To prevent this error, we always assign model = LinearRegression().
```
executed in 147ms, finished 19:48:52 2024-08-12

Out[46]:
```
LinearRegression
LinearRegression()
```

In [47]:
```
train_val(lm, X_train, y_train, X_test, y_test)

# First, we obtain our metrics from the test set.
# Then, we obtain our metrics from the train set and compare the scores the model achieved on data it has never seen before
# with the scores it achieved on the training data. If the scores are close to each other, it means the model is able to gener
# However, if there are significant differences between the scores, it means the model is not able to generalize.
```
executed in 32ms, finished 19:48:52 2024-08-12

Out[47]:

	train	test
R2	0.890	0.884
mae	1718.455	1725.509
mse	6088991.592	6259304.583
rmse	2467.588	2501.860

5.2 Adjusted R2 Score

R-squared (R2, The Coefficient of Determination) answers the questions of how much of the variation in the target can I explain with the available features, or how much of the information I need to correctly predict the target do I have.

It takes values between 0 and 1. The closer it is to 1, the higher the success rate.

A negative R2 score indicates that the model is misleading the data rather than explaining it, which is a sign of poor model performance. In this case, the data may need to be re-modeled using a different model or a different set of data features.

R-squared = 1 - (SSR / SST)

SSR (Residual Sum of Squares): SSR is the sum of the squares of the differences between the actual values and the predicted values. It is the sum of the squares of y - $\hat{y}$.

This term represents the sum of the squares of the difference between the actual value and the model's prediction for each observation.

SST (Total Sum of Squares): SST is the sum of the squares of the differences between the mean of the actual values and the actual values themselves. It is the sum of the squares of y - y_mean (the mean of the actual values).

This shows the overall variability of the data.

SSR represents how much your model is "wrong." A low SSR value means that your model is making predictions that are close to the actual values.

SST shows the total variability contained in your data. This can be considered the "maximum variability" that your model needs to predict.

The R^2 score (R squared) is a metric that measures how well a regression model performs. However, in some cases, it can be misleading, and it is important to consider the Adjusted R^2 value. Here's why:

Limitations of the R^2 Score

1. **Model Complexity:** The R^2 score tends to increase with each additional feature added to the model. This means that the more features you add to your model, the higher the R^2 value will generally be. This can lead to the false impression that the model is performing better than it actually is.
2. **Number of Features:** R^2 does not take into account the number of features in the model. This means that models with many features might be overfitting to the data, capturing noise rather than the true relationship.

Advantages of Adjusted R^2

1. **Considers the Number of Features:** Adjusted R^2 accounts for the number of features in the model, balancing this effect. When you add more features to your model, the Adjusted R^2 value will only increase if the new feature actually improves the model's performance.
2. **More Reliable Model Evaluation:** Adjusted R^2 is used to more accurately evaluate a model's performance, especially in models with multiple features.

Summary

The R^2 score can increase with the addition of each new feature, which can sometimes be misleading. Adjusted R^2 mitigates this by considering the number of features and generally provides a more reliable performance measure for models with multiple features. Therefore, it is important to look at Adjusted R^2 to more accurately understand the model's true performance

In [48]:
```
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)   # Calculates the value using the Adjusted R2 formula

    return adj_r2
```

executed in 5ms, finished 19:48:52 2024-08-12

In [49]:
```
y_pred = lm.predict(X_test)

# We obtain our predictions (y_pred) from the model to use within the function we defined above.
```

executed in 12ms, finished 19:48:52 2024-08-12

In [50]:
```
adj_r2(y_test, y_pred, df)

# We obtain our score by providing the necessary parameters to the function.
```

executed in 7ms, finished 19:48:52 2024-08-12

Out[50]:
```
0.8828492602064296
```

5.3 Cross Validate

Overfitting Control and Cross Validation

1. **Overfitting Control:**

-Overfitting occurs when a machine learning model fits the training data too closely, resulting in poor performance on new, unseen data. -Overfitting can be checked by comparing the scores on the training set and the validation set. If the scores on the training set are much higher than those on the validation set, this could be a sign of overfitting.

2. **Resetting the Model Before Each Cross Validation:**

The model should be reset before each cross-validation iteration. Otherwise, information from previous iterations may leak into the new iteration (data leakage), leading to misleading results.

3. **Using return_train_score=True:**

In cross-validation, the option return_train_score=True returns the training set scores for each iteration. This is useful for better understanding overfitting by comparing training and validation scores.

Negative Scoring of Metrics

- **Maximized Scores:** Scikit-learn algorithms are designed to maximize scores. However, metrics like MAE (Mean Absolute Error), MSE (Mean Squared Error), and RMSE (Root Mean Squared Error) are actually metrics that should be minimized.

Negative Scoring: Scikit-learn shifts these metrics to the negative axis to align with the algorithm's tendency to maximize. This should be considered when evaluating metrics like MAE, MSE, and RMSE.

Cross_validate and cross_val_score Functions

cross_val_score: Returns a score for a single metric. **cross_validate:** Can return scores for multiple metrics, making it more flexible and broadly applicable. In summary, the use of cross-validation and scoring metrics is critical for evaluating a model's generalization ability and detecting overfitting. These techniques are used to better understand the model's performance on real-world data.

In [51]:
```
model = LinearRegression()
scores = cross_validate(model, X_train, y_train, scoring=['r2',
            'neg_mean_absolute_error','neg_mean_squared_error','neg_root_mean_squared_error'],
             cv =10, return_train_score=True)
```

executed in 1.45s, finished 19:48:54 2024-08-12

In [52]:
```
pd.DataFrame(scores)
```
executed in 16ms, finished 19:48:54 2024-08-12

Out[52]:

	fit_time	score_time	test_r2	train_r2	test_neg_mean_absolute_error	train_neg_mean_absolute_error	test_neg_mean_squared_error	train_neg_mean_squared_e
0	0.109	0.008	0.891	0.890	-1719.520	-1726.579	-5955906.000	-6117627
1	0.108	0.007	0.880	0.891	-1809.082	-1704.708	-7372596.768	-5977292
2	0.161	0.005	0.903	0.888	-1675.348	-1720.954	-5852367.067	-6129630
3	0.101	0.005	0.882	0.890	-1676.788	-1734.738	-5171070.062	-6204713
4	0.103	0.007	0.878	0.891	-1824.524	-1712.041	-6594274.312	-6044960
5	0.117	0.006	0.887	0.890	-1728.856	-1716.903	-5977848.111	-6116207
6	0.131	0.005	0.878	0.891	-1754.029	-1712.446	-6882980.534	-6014497
7	0.132	0.008	0.886	0.890	-1764.317	-1714.978	-6244001.052	-6083169
8	0.147	0.008	0.876	0.891	-1823.994	-1706.043	-7228938.810	-5972788
9	0.097	0.009	0.893	0.889	-1678.446	-1723.104	-6021675.578	-6108254

In [158]:
```
pd.DataFrame(scores).iloc[:, 2:].mean()

# We see that the scores between the train and validation sets are close to the train set.
#There is no overfitting.
```
executed in 78ms, finished 07:51:51 2024-08-13

Out[158]:
```
rmse    1991.057
dtype: float64
```

In [54]:
```
train_val(lm, X_train, y_train, X_test, y_test)

# The train and test set scores we obtain from the train_val function provide insight into whether there is overfitting in our

# However, determining if there is truly overfitting is done by comparing the train and validation scores obtained from CV.

# Additionally, we check if the test scores we provide to the client are truly consistent by comparing them with the validatio
# scores obtained from CV. If the scores are close, we say the scores are consistent; if not, we say the scores are inconsiste
```
executed in 29ms, finished 19:48:54 2024-08-12

Out[54]:

	train	test
R2	0.890	0.884
mae	1718.455	1725.509
mse	6088991.592	6259304.583
rmse	2467.588	2501.860

In [55]:
```
2501/df.price.mean()

# Since we are looking at the errors in terms of averages, we will take the average of the value we want to predict
# and divide the average of the errors (the average of the error metrics) by the average of the value we want to predict.

# The ratio of the two average values will show how much error we are making.
# According to the RMSE score obtained from the test set (hold out set) score we will provide to the client,
# our model is making an average error of 13.9%.
```

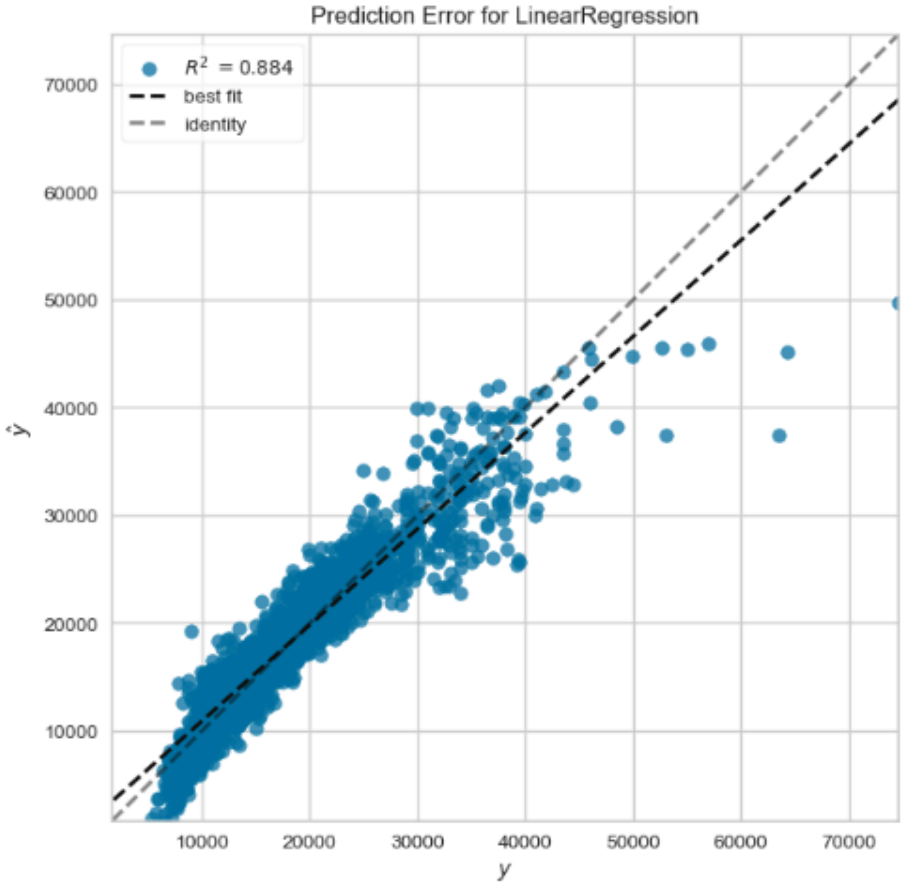
5.4 Prediction Error

In [56]:
```
# sklearn.__version__
# pip install scikit-learn==1.2.1 --user

from yellowbrick.regressor import PredictionError
from yellowbrick.features import RadViz

# A library that has recently become popular in ML visualizations
# allows us to look at the distribution of residuals for both the train and test sets.
```
executed in 696ms, finished 19:48:55 2024-08-12

In [57]:

```
visualizer = RadViz(size=(720, 3000))
model = LinearRegression()
visualizer = PredictionError(model)
visualizer.fit(X_train, y_train)  # Fit the training data to the visualizer
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
visualizer.show();
# With the prediction error visualization, we can see how well our model's predictions are performing.
# We train with the training data
# and get the scores with the test data - then plot the graphs.

# On the y-axis are our predictions, and on the x-axis are the actual values.
# The faint (gray) identity line represents the points where the model's predictions are perfect,
# while the dark-colored line represents the best fit line drawn by our model after training.
# The closer the dark line is to the faint line, the better the model's predictions are.
# On the identity line, the difference between the actual y values and the predicted y values has an average of zero,
# which means that the residuals are 0, and the R2 value equals 1.

# Looking at the visualization, we can see that cars priced at 40,000 EURO and above are pulling our best fit line downward.
# If I remove the cars priced at 40,000 EURO and above, which I see are disrupting my scores, or the outlier-priced cars ident
# and retrain the model on this data, can I achieve better scores?
```

executed in 738ms, finished 19:48:55 2024-08-12

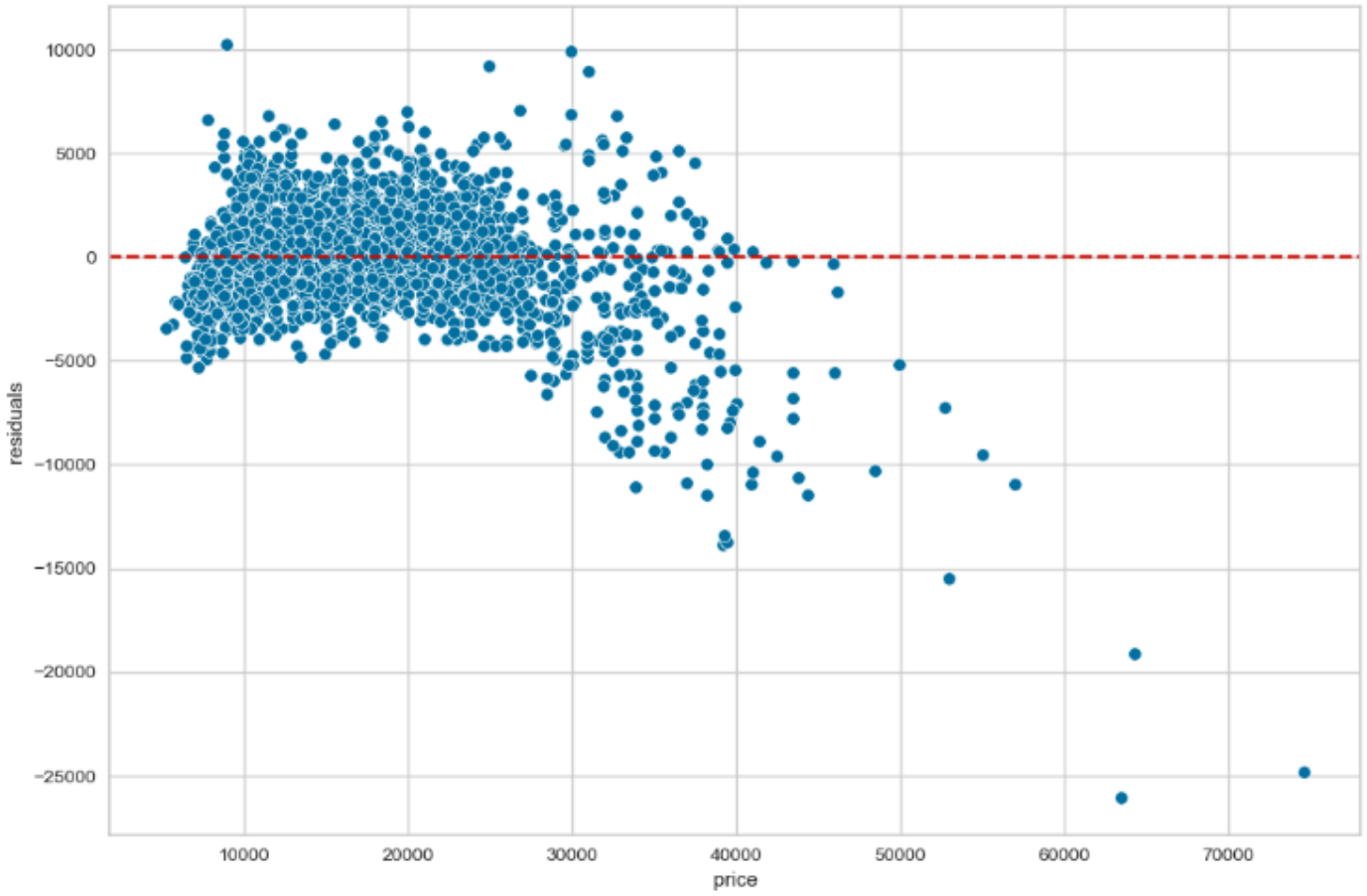


5.5 Residual Plot

In [58]:

```
plt.figure(figsize=(12,8))
residuals = y_test-y_pred

sns.scatterplot(x = y_test, y = -residuals) #-residuals
plt.axhline(y = 0, color ="r", linestyle = "--")
plt.ylabel("residuals")
plt.show()

# When error terms are calculated as negative values, the values on the y-axis of the graph are made positive.
# This more clearly shows the differences in the magnitudes of the error terms.

# Residuals;
# 1- Should be approximately evenly and randomly distributed on both sides of the axis where the error is 0,
# and they should not contain any pattern.
# 2- This distribution should conform to a normal distribution.

# If these conditions are met, we can say that the data is suitable for linear regression.
```

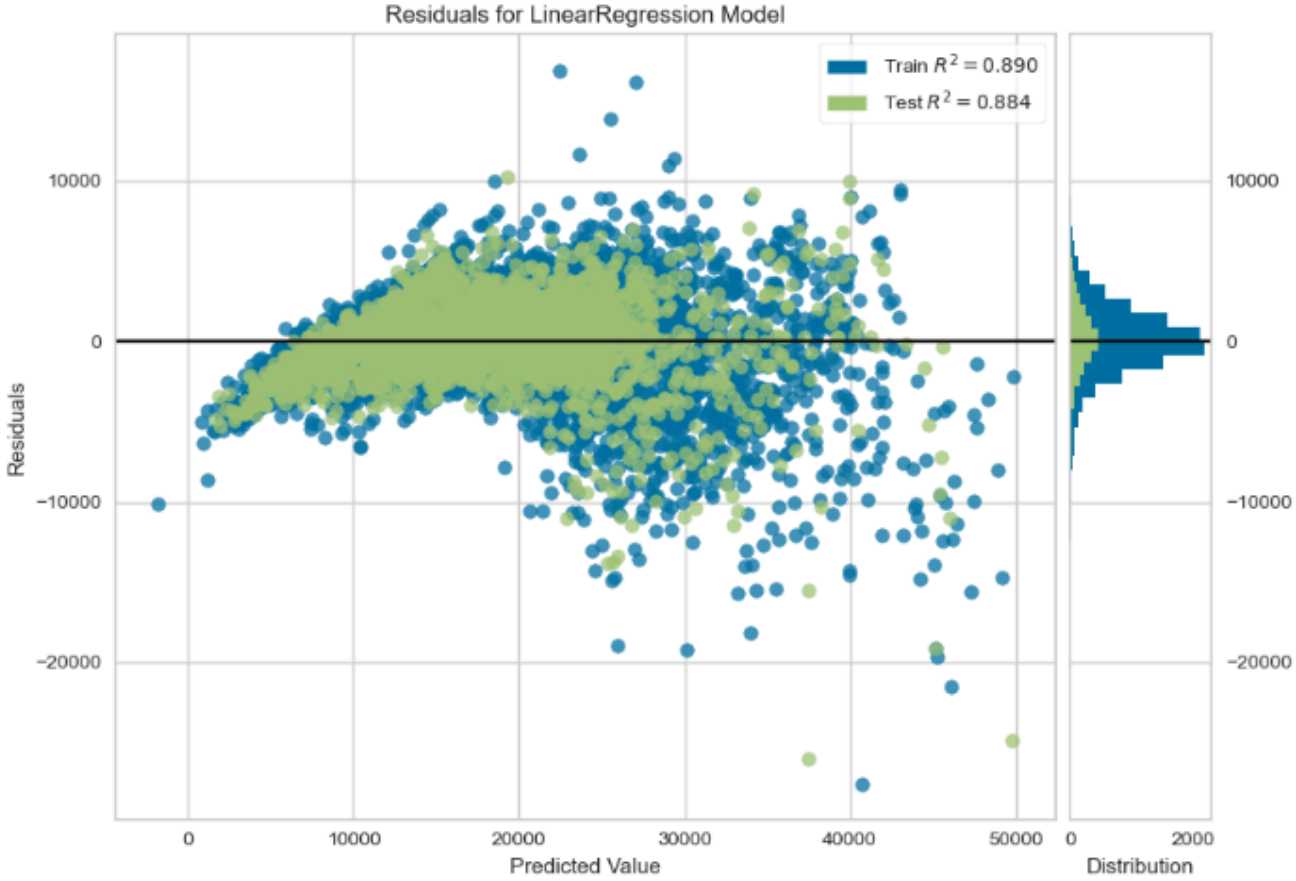
executed in 218ms, finished 19:48:56 2024-08-12



In [59]:

```
from yellowbrick.regressor import ResidualsPlot

visualizer = RadViz(size=(1000, 720))
model = LinearRegression()
visualizer = ResidualsPlot(model)

visualizer.fit(X_train, y_train)  # Fit the training data to the visualizer
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
visualizer.show();
```

executed in 822ms, finished 19:48:56 2024-08-12



5.6 Dropping observations from the dataset that worsen my predictions

In [60]:
```
df1 = df[~(df.price>35000)]
df1.head()
```
executed in 61ms, finished 19:48:57 2024-08-12

Out[60]:

	price	km	Gears	age	Previous_Owners	hp_kW	Inspection_new	Displacement_cc	Weight_kg	cons_comb	cc_Air conditioning	cc_Air suspension	cc_Armrest
0	15770	56013.000	7.000	3.000	2.000	66.000	1	1422.000	1220.000	3.800	1	0	1
1	14500	80000.000	7.000	2.000	1.000	141.000	0	1798.000	1255.000	5.600	1	0	0
2	14640	83450.000	7.000	3.000	1.000	85.000	0	1598.000	1135.000	3.800	1	0	0
3	14500	73000.000	6.000	3.000	1.000	66.000	0	1422.000	1195.000	3.800	0	1	1
4	16790	16200.000	7.000	3.000	1.000	66.000	1	1422.000	1135.000	4.100	1	0	1

In [61]:
```
len(df[df.price>35000])
```
executed in 7ms, finished 19:48:57 2024-08-12

Out[61]: 473

In [62]:
```
df0[df0.price>35000].groupby("make_model").count().iloc[:,0]
```
executed in 12ms, finished 19:48:57 2024-08-12

Out[62]:
```
make_model
Audi A1             2
Audi A3            36
Opel Astra          5
Opel Insignia     216
Renault Espace    236
Name: body_type, dtype: int64
```

In [63]:
```
df0.make_model.value_counts()
```
executed in 8ms, finished 19:48:57 2024-08-12

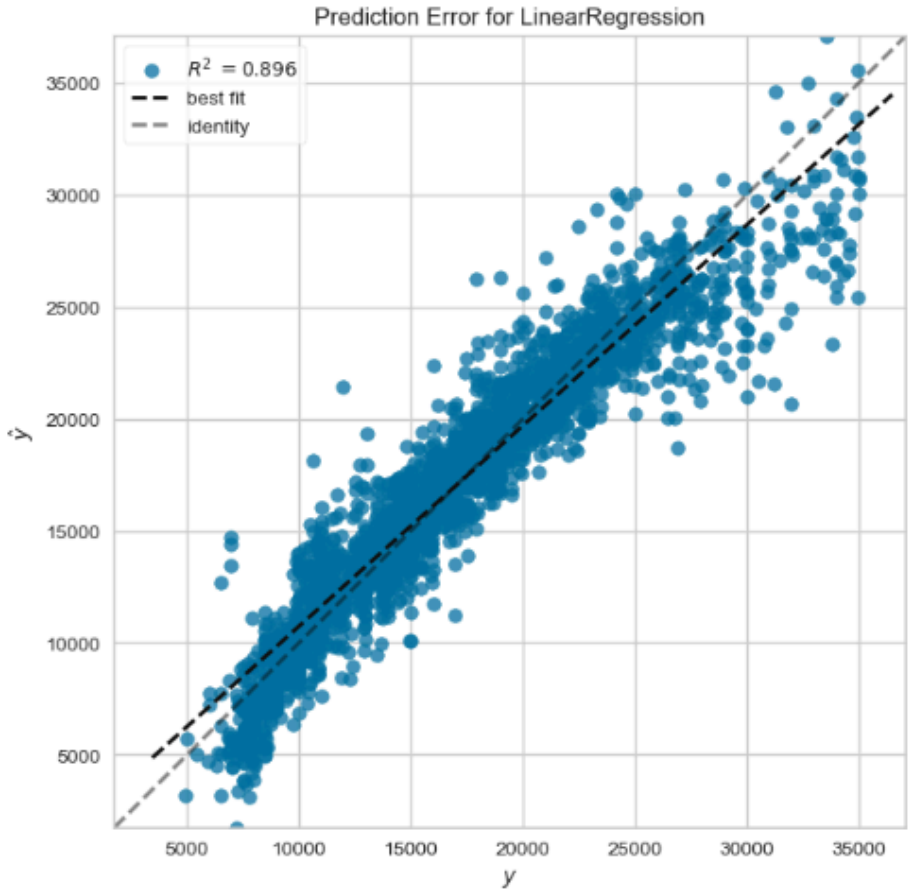
Out[63]:
```
make_model
Audi A3           3097
Audi A1           2614
Opel Insignia     2598
Opel Astra        2525
Opel Corsa        2216
Renault Clio      1839
Renault Espace     991
Renault Duster      34
Audi A2              1
Name: count, dtype: int64
```

In [64]:
```
X = df1.drop(columns = "price")
y = df1.price

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=101)
```
executed in 37ms, finished 19:48:57 2024-08-12

In [65]:
```
lm2 = LinearRegression()
lm2.fit(X_train,y_train)
```
executed in 103ms, finished 19:48:57 2024-08-12

Out[65]:
```
▾ LinearRegression
LinearRegression()
```

In [66]:
```
visualizer = RadViz(size=(720, 3000))
model = LinearRegression()
visualizer = PredictionError(model)
visualizer.fit(X_train, y_train)  # Fit the training data to the visualizer
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
visualizer.show();
```
executed in 461ms, finished 19:48:57 2024-08-12



In [67]:
```
train_val(lm2, X_train, y_train, X_test, y_test)
```
executed in 37ms, finished 19:48:57 2024-08-12

Out[67]:

	train	test
R2	0.898	0.896
mae	1414.209	1415.381
mse	3792751.061	3840225.852
rmse	1947.499	1959.649

```
      train          test
R2    0.890          0.884
mae   1718.455       1725.509
mse   6088991.592    6259304.583
rmse  2467.588       2501.860
```

In [68]:
```
1894/df1.price.mean()

# Our average prediction error decreased from 13.81% to 10.93% after removing outliers.
# There was an improvement of approximately 2.88% in our predictions.
```
executed in 9ms, finished 19:48:57 2024-08-12

Out[68]:
```
0.1093750965362424
```

In [70]:
```
y_pred = lm2.predict(X_test)

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))
```
executed in 12ms, finished 19:48:57 2024-08-12

In [71]:
```
my_dict = { 'Actual': y_test, 'Pred': y_pred, 'Residual': y_test-y_pred }
compare = pd.DataFrame(my_dict)
```
executed in 5ms, finished 19:48:57 2024-08-12

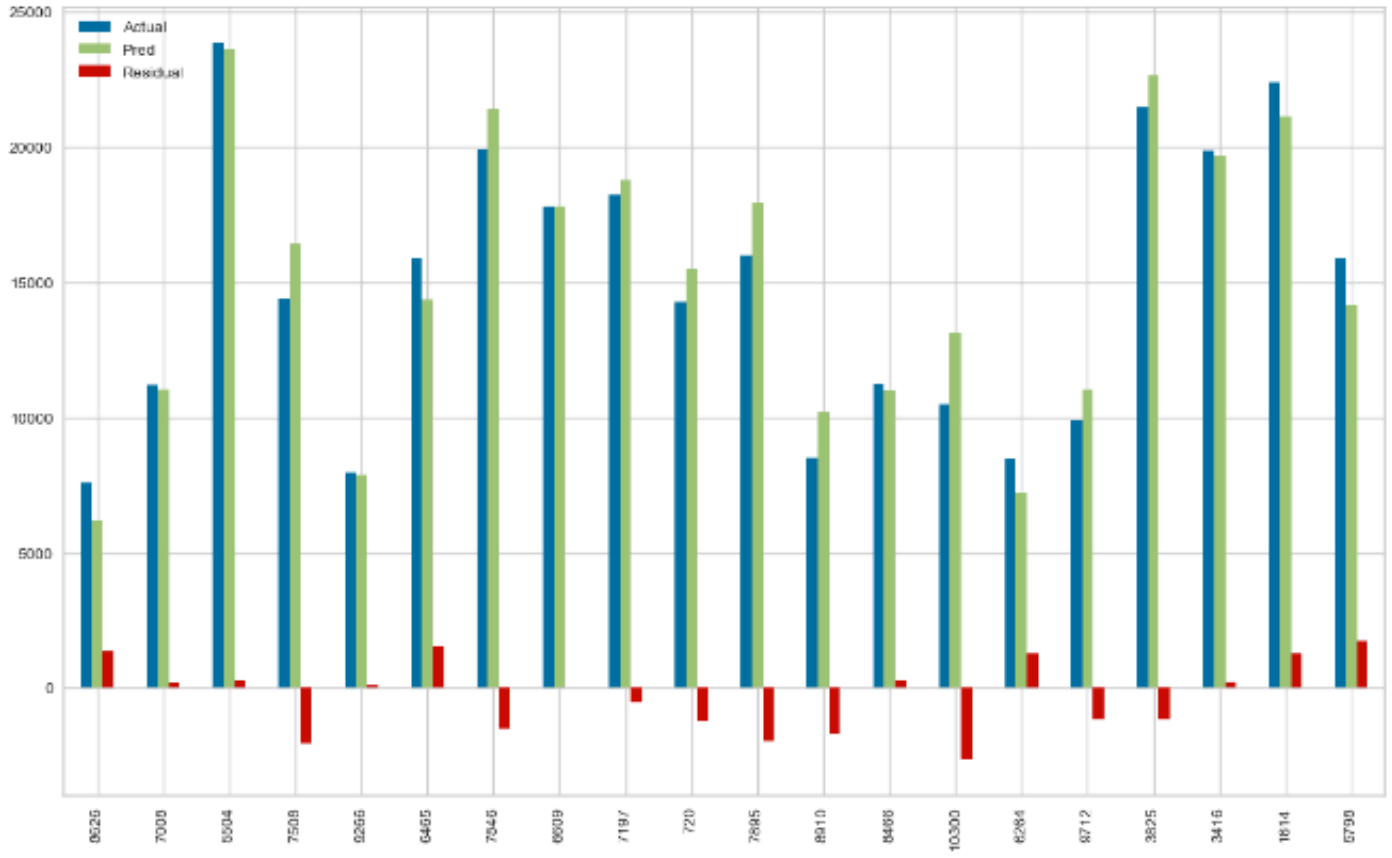
In [72]:
```
comp_sample = compare.sample(20)
comp_sample
```
executed in 11ms, finished 19:48:57 2024-08-12

Out[72]:

	Actual	Pred	Residual
8626	7580	6219.331	1360.669
7008	11200	11031.789	168.211
5504	23880	23615.866	264.134
7508	14400	16450.595	-2050.595
9266	7990	7887.509	102.491
6465	15900	14375.169	1524.831
7846	19940	21436.869	-1496.869
6609	17840	17837.983	2.017
7197	18250	18780.623	-530.623
720	14295	15505.504	-1210.504
7895	15990	17960.594	-1970.594
8910	8500	10207.549	-1707.549
8466	11250	10996.389	253.611
10300	10499	13137.526	-2638.526
6264	8490	7226.341	1263.659
9712	9890	11062.008	-1172.008
3825	21500	22652.324	-1152.324
3416	19900	19660.689	239.311
1614	22400	21133.026	1266.974
5798	15900	14143.564	1756.436

In [73]:
```
comp_sample.plot(kind='bar',figsize=(15,9))
plt.show()
```

executed in 359ms, finished 19:48:58 2024-08-12



- A positive coefficient indicates that as the associated input feature increases, the target variable will also increase.
- A negative coefficient indicates that as the associated input feature increases, the target variable will decrease.
- The larger the absolute value of the coefficient, the greater the impact of the associated input feature on the target variable.
- If the coefficient is close to zero, it means that the associated input feature has little or no effect on the target variable.

In [74]:
```
# We will use the coefficients to see the impact of the features on the prediction and for feature selection.
# The order of the coefficients is the same as the order of the features in the X_train data we provided for model training.

pd.DataFrame(lm2.coef_, index = X.columns, columns=["Coef"]).sort_values("Coef")
```

executed in 12ms, finished 19:48:58 2024-08-12

Out[74]:

	Coef
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
Type_Pre-registered	-1275.244
ex_Electric Starter	-1238.268

- A positive coefficient indicates that as the associated input feature increases, the target variable will also increase.
- A negative coefficient indicates that as the associated input feature increases, the target variable will decrease.
- The larger the absolute value of the coefficient, the greater the impact of the associated input feature on the target variable.
- If the coefficient is close to zero, it means that the associated input feature has little or no effect on the target variable.

5.7 Pipeline

What is a Pipeline?

The concept of a pipeline is used to efficiently and accurately manage data processing and machine learning processes:

A pipeline is a tool that allows you to sequence and execute data preprocessing and modeling steps in a single flow. In the Scikit-learn library, a pipeline combines data transformation processes and an estimator (a prediction model).

Benefits of Using a Pipeline

1. **Code Readability and Cleanliness:** It gathers all processing steps within a single structure, making the code easier to read and maintain.
2. **Preventing Data Leakage:** Especially when data preprocessing steps are performed before model training, it prevents information leakage about future data. The pipeline structure performs these operations separately for each training subset, thereby reducing the risk of data leakage.
3. **Ease of Hyperparameter Tuning:** During cross-validation and hyperparameter tuning processes, you can set and manage the parameters of all steps within the pipeline from a single place.

Steps for Creating and Using a Pipeline

1. **Defining the Processing Steps:** First, you define the transformation steps you will perform on your data (e.g., scaling, categorical data transformation).
2. **Creating the Pipeline:** You create a pipeline object that includes the selected processing steps and the prediction model.
3. **Training the Pipeline:** By fitting the pipeline on the training dataset, you train all transformation steps and the model.

Summary

Using a pipeline makes data science and machine learning workflows more efficient and reliable. It ensures that each step is applied in the correct order and only on the training data, preventing data leakage and improving the model's ability to generalize. Additionally, it simplifies the model selection and parameter tuning process, reducing errors.

In [75]:
```
operations = [("scaler", MinMaxScaler()), ("linear", LinearRegression())]
```

executed in 4ms, finished 19:48:58 2024-08-12

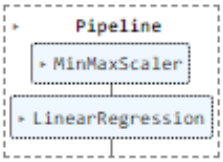
In [76]:
```
pipe_model = Pipeline(steps=operations)
```

executed in 5ms, finished 19:48:58 2024-08-12

In [77]:
```
pipe_model.fit(X_train, y_train)

# We use the fit method to train the data transformation steps and the model.
# This first scales our data using MinMaxScaler and then trains the LinearRegression model on the scaled data.
```

executed in 126ms, finished 19:48:58 2024-08-12

Out[77]:



In [78]:
```
train_val(pipe_model, X_train, y_train, X_test, y_test)
```

executed in 55ms, finished 19:48:58 2024-08-12

Out[78]:

	train	test
R2	0.898	0.896
mae	1414.209	1415.381
mse	3792751.061	3840225.852
rmse	1947.499	1959.649

6 Implement Ridge Regression

Why and When Should I Use Ridge-Lasso?

Ridge and Lasso regressions are regularized versions of linear regression. This regularization makes the model more resistant to overfitting and can sometimes improve the overall performance of the model.

1. **Preventing Overfitting:**

Ridge (L2 Regularization): Adds the sum of the squared coefficients as a penalty to shrink the coefficients. This reduces the complexity of the model, thereby preventing overfitting. **Lasso (L1 Regularization):** Adds the sum of the absolute values of the coefficients as a penalty to shrink the coefficients towards zero. This can lead to some coefficients being reduced to exactly zero, performing feature selection and reducing the complexity of the model.

2. **Feature Selection:**

Lasso regression can reduce some coefficients to zero, effectively removing insignificant features from the model. This process is known as feature selection and is very useful in high-dimensional datasets.

Multicollinearity:

If there is high correlation (multicollinearity) among features in a dataset, it can destabilize the coefficients in linear regression. Ridge and Lasso can stabilize the coefficients in such cases.

Model Interpretability:

Lasso can improve the interpretability of the model by including only the most important features. Ridge also makes the impact of features more understandable by shrinking the coefficients. In conclusion, Ridge and Lasso regressions help us overcome some of the limitations of linear regression and build more generalizable, more stable models.

The choice of method depends on the dataset, the problem definition, and how flexible you want the model to be.

Approaches like Elastic Net are available to find a balance between these two regularization techniques.

L2 Regularization (Ridge Regularization)

Purpose: Prevent model complexity and overfitting.

Method: Reduces the weights by penalizing the sum of the squared coefficients.

Result: The coefficients approach zero but do not become zero. All variables in the model are retained.

Usage: Preferred when we want every feature to be included in the model.

Penalty: The sum of the squared coefficients, scaled by λ (lambda - the regularization parameter), is added to the model's loss function.

L1 Regularization (Lasso Regularization)

Purpose: Prevent overfitting and perform feature selection.

Method: Reduces some weights to exactly zero by penalizing the sum of the absolute values of the coefficients.

Result: The coefficients of insignificant features are set to zero, thus automatically performing feature selection.

Usage: Preferred when we believe that only a few features are impactful in the model.

Penalty: The sum of the absolute values of the coefficients, scaled by λ, is added to the model's loss function.

Both regularization methods use the λ parameter; the larger the value, the stronger the regularization, and the smaller the model weights.

However, while L2 retains all features in the model, L1 selects only the most important features and discards the rest.

For this reason, L1 regularization is considered a useful tool for feature selection, whereas L2 is more focused on reducing overfitting.

- Import the modul
- Do not forget to scale the data or use Normalize parameter as True
- Fit the model
- Predict the test set
- Evaluate model performance (use performance metrics for regression)
- Tune alpha hiperparameter by using cross validation and determine the optimal alpha value.
- Fit the model and predict again with the new alpha value.

6.1 Scaling

In [79]:
```
# In the ridge regression algorithm, the data we train on must be scaled.
#This process ensures that all features are equally weighted by the model, addressing the multicollinearity issue and allowing

# Why do we use MinMax scaling??
# Because we have features that were scaled between 0 and 1 using get_dummies.

scaler = MinMaxScaler()
scaler.fit(X_train)

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

executed in 53ms, finished 19:48:58 2024-08-12

Hedef değişkeninin scale edilmesi linear regresyon için gerekli değildir, çünkü hedef değişken katsayıların hesaplanmasında kullanılmaz.

6.2 Model

In [80]:
```
from sklearn.linear_model import Ridge
```

executed in 4ms, finished 19:48:58 2024-08-12

In [81]:
```
ridge_model = Ridge(alpha=1, random_state=42)
```

executed in 5ms, finished 19:48:58 2024-08-12

In [82]:
```
ridge_model.fit(X_train_scaled, y_train)
```

executed in 55ms, finished 19:48:58 2024-08-12

Out[82]:

Ridge

```
Ridge(alpha=1, random_state=42)
```

In [83]:
```
train_val(ridge_model, X_train_scaled, y_train, X_test_scaled, y_test)
```

executed in 19ms, finished 19:48:58 2024-08-12

Out[83]:

	train	test
R2	0.898	0.896
mae	1413.716	1412.912
mse	3794938.109	3829231.926
rmse	1948.060	1956.842

6.3 Finding best alpha for Ridge

Purpose of GridSearchCV

Hyperparameter Optimization: GridSearchCV is used to find the best hyperparameter combinations to maximize the performance of a machine learning model.

Comprehensive Search: When there are multiple hyperparameters, GridSearchCV tries all possible hyperparameter combinations to find the one that gives the best result.

Hyperparameter vs. Parameter

Hyperparameter: These are parameters that are set before the model is trained and control the learning process and structure of the model. For example, the depth of a tree in a tree-based model or the regularization value in a linear model.

Parameter: These are the values learned by the model during the training process and derived from the dataset. For example, coefficients and intercepts in a linear regression model.

How GridSearchCV Works

1. **Defining the Parameter Grid:** A list of various hyperparameters and the values to be tested for these parameters (parameter grid) is created by the user.
2. **Search and Evaluation:** GridSearchCV trains the model by trying each combination in this parameter grid and evaluates the model's performance using cross-validation for each combination.
3. **Finding the Best Combination:** Among all combinations, the hyperparameter combination that maximizes the model's performance is selected.
4. **Training the Final Model:** The final model is trained with the best hyperparameters selected. Therefore, as a result of GridSearchCV, the model is trained with these best hyperparameters.

Summary

GridSearchCV is a hyperparameter optimization tool used to maximize the performance of a machine learning model. By systematically trying all possible hyperparameter combinations, it determines the combination that gives the best result and trains the final model with this combination. This process is critical to ensuring the model makes more effective and accurate predictions.

In [84]:
```
from sklearn.model_selection import GridSearchCV
```
executed in 4ms, finished 19:48:58 2024-08-12

In [85]:
```
alpha_space = np.linspace(0.01, 100, 100)
alpha_space

# The GridSearchCV (GS) algorithm first requires a search space for the relevant hyperparameter (HP).
# To do this, we define the search space to provide to GS.
# (We provide 100 different values evenly spaced between 0.01 and 100.)
```
executed in 9ms, finished 19:48:58 2024-08-12

Out[85]:
```
array([1.000e-02, 1.020e+00, 2.030e+00, 3.040e+00, 4.050e+00, 5.060e+00,
       6.070e+00, 7.080e+00, 8.090e+00, 9.100e+00, 1.011e+01, 1.112e+01,
       1.213e+01, 1.314e+01, 1.415e+01, 1.516e+01, 1.617e+01, 1.718e+01,
       1.819e+01, 1.920e+01, 2.021e+01, 2.122e+01, 2.223e+01, 2.324e+01,
       2.425e+01, 2.526e+01, 2.627e+01, 2.728e+01, 2.829e+01, 2.930e+01,
       3.031e+01, 3.132e+01, 3.233e+01, 3.334e+01, 3.435e+01, 3.536e+01,
       3.637e+01, 3.738e+01, 3.839e+01, 3.940e+01, 4.041e+01, 4.142e+01,
       4.243e+01, 4.344e+01, 4.445e+01, 4.546e+01, 4.647e+01, 4.748e+01,
       4.849e+01, 4.950e+01, 5.051e+01, 5.152e+01, 5.253e+01, 5.354e+01,
       5.455e+01, 5.556e+01, 5.657e+01, 5.758e+01, 5.859e+01, 5.960e+01,
       6.061e+01, 6.162e+01, 6.263e+01, 6.364e+01, 6.465e+01, 6.566e+01,
       6.667e+01, 6.768e+01, 6.869e+01, 6.970e+01, 7.071e+01, 7.172e+01,
       7.273e+01, 7.374e+01, 7.475e+01, 7.576e+01, 7.677e+01, 7.778e+01,
       7.879e+01, 7.980e+01, 8.081e+01, 8.182e+01, 8.283e+01, 8.384e+01,
       8.485e+01, 8.586e+01, 8.687e+01, 8.788e+01, 8.889e+01, 8.990e+01,
       9.091e+01, 9.192e+01, 9.293e+01, 9.394e+01, 9.495e+01, 9.596e+01,
       9.697e+01, 9.798e+01, 9.899e+01, 1.000e+02])
```

These notations are a form of numerical expression known as scientific notation. Scientific notation is used to represent very large or very small numbers and indicates how a number can be expressed in terms of powers of 10.

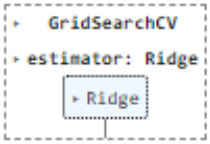
In [86]:
```
ridge_model = Ridge(random_state=42)

# GridSearchCV also requires the machine learning algorithm that we will use.
```
executed in 5ms, finished 19:48:58 2024-08-12

In [87]:
```
param_grid = {"alpha":alpha_space}

# We write the hyperparameter spaces in a dictionary using the names of the hyperparameters (HP) in the ML algorithm.
# We define the space as a list or array for each HP.
# Note that the alpha_space variable defined for alpha here is a 100-element array.
```
executed in 5ms, finished 19:48:58 2024-08-12

In [88]:
```
ridge_grid_model = GridSearchCV(estimator=ridge_model,
                                param_grid=param_grid,
                                scoring='neg_root_mean_squared_error',
                                cv=10,
                                n_jobs = -1)

# For the ridge ML model, we specify the metric we want to improve the scores for by writing it next to scoring.
# We can specify only one metric. The default value is R2_score.

# GridSearchCV will take scores from different regions of the data for each value in the hyperparameter space (according to th
# in a way that maximizes the metric we specified, and then it calculates the average of these scores.
# It returns the hyperparameter value(s) with the highest average score.

# The default cv count is 5.

# Since GridSearchCV also performs Cross Validation, we set return_train_score=True to obtain both validation and train set sc
# (It only returns the metric score specified in scoring.)
```
executed in 5ms, finished 19:48:58 2024-08-12

In [89]:
```
ridge_grid_model.fit(X_train_scaled,y_train)

# Since we set cv=10, it performs training on 10 different regions of the data for each alpha value.
```
executed in 23.0s, finished 19:49:21 2024-08-12

Out[89]:



In [92]:
```
pd.DataFrame(ridge_grid_model.cv_results_)
```
executed in 116ms, finished 19:49:21 2024-08-12

Out[92]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_alpha	params	split0_test_score	split1_test_score	split2_test_score
0	0.132	0.020	0.004	0.002	0.010	{'alpha': 0.01}	-1978.044	-1937.280	-1904.261
1	0.106	0.016	0.004	0.002	1.020	{'alpha': 1.02}	-1972.226	-1938.831	-1906.079
2	0.122	0.018	0.005	0.008	2.030	{'alpha': 2.03}	-1969.822	-1941.251	-1908.777
3	0.163	0.042	0.004	0.002	3.040	{'alpha': 3.04}	-1969.348	-1944.208	-1911.925
4	0.153	0.024	0.004	0.003	4.050	{'alpha': 4.05}	-1970.051	-1947.502	-1915.303
5	0.144	0.038	0.012	0.023	5.060	{'alpha': 5.06}	-1971.508	-1951.005	-1918.792
6	0.199	0.050	0.005	0.001	6.070	{'alpha': 6.07}	-1973.465	-1954.635	-1922.322
7	0.114	0.011	0.004	0.002	7.080	{'alpha': 7.08}	-1975.759	-1958.335	-1925.853
8	0.118	0.020	0.005	0.007	8.090	{'alpha': 8.09}	-1978.283	-1962.068	-1929.359
9	0.149	0.023	0.003	0.001	9.100	{'alpha': 9.1}	-1980.964	-1965.807	-1932.826

In [93]:
```
ridge_grid_model.best_index_

# Returns the index of the DataFrame where the best score was obtained.
```
executed in 7ms, finished 19:49:21 2024-08-12

Out[93]: 1

In [94]:
```
train_val(ridge_grid_model, X_train_scaled, y_train, X_test_scaled, y_test)
```
executed in 22ms, finished 19:49:21 2024-08-12

Out[94]:

	train	test
R2	0.898	0.896
mae	1413.712	1412.875
mse	3795015.419	3829086.131
rmse	1948.080	1956.805

In [95]:
```
y_pred = ridge_grid_model.predict(X_test_scaled)
rm_R2 = r2_score(y_test, y_pred)
rm_mae = mean_absolute_error(y_test, y_pred)
rm_rmse = np.sqrt(mean_squared_error(y_test, y_pred))
```
executed in 11ms, finished 19:49:21 2024-08-12

In [96]:
```
ridge = Ridge(alpha=1.02, random_state=42).fit(X_train_scaled, y_train)

pd.DataFrame(ridge.coef_, index = X.columns, columns=["Coef"]).sort_values("Coef")

# 'GridSearchCV' object has no attribute 'coef_'
# Since grid models do not have the coef_ attribute, we manually rebuild and train the model using the best hyperparameters
# found after GridSearch, and then we obtain the coefficients.
```
executed in 37ms, finished 19:49:21 2024-08-12

Out[96]:

	Coef
km	-11101.861
make_model_Renault Duster	-8384.639
make_model_Renault Clio	-5409.447
make_model_Opel Corsa	-5295.138
age	-4382.308
make_model_Opel Astra	-3334.959
Displacement_cc	-2271.755
Type_Employee's car	-2076.174
Type_Used	-1967.430
Drive_chain_rear	-1817.244
Weight_kg	-1681.670
Gearing_Type_Manual	-1604.260

7 Implement Lasso Regression

7.1 Model

In [97]:
```
from sklearn.linear_model import Lasso
```
executed in 5ms, finished 19:49:21 2024-08-12

In [98]:
```
lasso_model = Lasso(random_state=42, alpha=1)

# The Lasso model also requires the data to be provided in a scaled form. This process ensures that all features are equally
# addressing the multicollinearity issue and allowing us to perform feature selection.
```
executed in 5ms, finished 19:49:21 2024-08-12

In [99]:
```
lasso_model.fit(X_train_scaled, y_train)
```
executed in 177ms, finished 19:49:21 2024-08-12

Out[99]:
```
Lasso
Lasso(alpha=1, random_state=42)
```

In [100]:
```
train_val(lasso_model, X_train_scaled, y_train, X_test_scaled, y_test)
```
executed in 21ms, finished 19:49:21 2024-08-12

Out[100]:

	train	test
R2	0.898	0.896
mae	1414.170	1409.010
mse	3806483.613	3814782.540
rmse	1951.021	1953.147

7.2 Finding best alpha for Lasso

In [101]:
```
lasso_model = Lasso(random_state=42)

param_grid = {'alpha':alpha_space}

lasso_grid_model = GridSearchCV(estimator=lasso_model,
                                param_grid=param_grid,
                                scoring='neg_root_mean_squared_error',
                                cv=10,
                                n_jobs = -1)
```
executed in 7ms, finished 19:49:21 2024-08-12

In [102]:
```
lasso_grid_model.fit(X_train_scaled,y_train)
```
executed in 1m 1.01s, finished 19:50:22 2024-08-12

Out[102]:
```
GridSearchCV
estimator: Lasso
Lasso
```

In [103]:
```
lasso_grid_model.best_params_
```
executed in 7ms, finished 19:50:23 2024-08-12

Out[103]:
```
{'alpha': 1.02}
```

In [104]:
```
train_val(lasso_grid_model, X_train_scaled, y_train, X_test_scaled, y_test)
```
executed in 271ms, finished 19:50:23 2024-08-12

Out[104]:

	train	test
R2	0.898	0.896
mae	1414.186	1408.958
mse	3806891.515	3814773.360
rmse	1951.126	1953.144

In [105]:
```
y_pred = lasso_grid_model.predict(X_test_scaled)
lasm_R2 = r2_score(y_test, y_pred)
lasm_mae = mean_absolute_error(y_test, y_pred)
lasm_rmse = np.sqrt(mean_squared_error(y_test, y_pred))
```

executed in 8ms, finished 19:50:23 2024-08-12

In [106]:
```
lasso = Lasso(alpha=1.02, random_state=42).fit(X_train_scaled, y_train)
pd.DataFrame(lasso.coef_, index = X.columns, columns=["Coef"]).sort_values("Coef")
```

executed in 661ms, finished 19:50:23 2024-08-12

ss_Emergency system	-584.726
ss_Rear airbag	-550.434
body_type_Transporter	-496.124
ss_Isofix	-346.314
Type_New	-322.568
cc_Parking assist system sensors rear	-310.770
ss_Immobilizer	-270.665
cc_Panorama roof	-266.736
em_USB	-260.430
ss_Central door lock	-258.191
ss_Central door lock with remote control	-236.094
ss_Power steering	-226.247
em_Bluetooth	-224.626

8 Implement Elastic-Net

It is a combination of Ridge Regression and Lasso techniques.

It has two separate regularization parameters to determine the strength of the regularization terms, allowing a balance between Ridge Regression and Lasso.

When l1_ratio=1 is selected as a hyperparameter, it behaves like Lasso, and when l1_ratio=0 is selected, it behaves like Ridge. Generally, Lasso is more inclined.

When to Use Elastic-Net:

Elastic-Net is particularly useful when many features have high correlations with each other and when you also want to eliminate some features entirely.

If you need feature selection while also managing high correlations among features, Elastic-Net meets this need for balance.

Advantages of Elastic-Net:

1. **Feature Selection and Regularization:** Elastic-Net combines the feature selection capability of Lasso with the regularization applied by Ridge to the model coefficients. This is useful when you want to perform both feature selection and regularization of the coefficients.
2. **Multicollinearity:** In datasets with highly correlated features, Lasso may randomly select one feature. Elastic-Net can mitigate this issue by selecting such feature groups together.
3. **High-Dimensional Data:** When the number of features exceeds the number of observations, Elastic-Net provides the benefits of Lasso while maintaining the stability of Ridge.

- Import the modul
- Do not forget to scale the data or use Normalize parameter as True(If needed)
- Fit the model
- Predict the test set
- Evaluate model performance (use performance metrics for regression)
- Tune alpha hyperparameter by using GridSearchCV and determine the optimal alpha value.
- Fit the model and predict again with the new alpha value.
- Compare different evaluation metrics

8.1 Model

In [107]:
```
from sklearn.linear_model import ElasticNet
```

executed in 6ms, finished 19:50:23 2024-08-12

In [108]:
```
elastic_model = ElasticNet(random_state=42)
elastic_model.fit(X_train_scaled,y_train)
```

executed in 186ms, finished 19:50:24 2024-08-12

Out[108]:
```
ElasticNet(random_state=42)
```

In [109]:
```
train_val(elastic_model, X_train_scaled, y_train, X_test_scaled, y_test)
```
executed in 33ms, finished 19:50:24 2024-08-12

Out[109]:

	train	test
R2	0.584	0.592
mae	3099.732	3053.445
mse	15472875.539	15016889.532
rmse	3933.558	3875.161

8.2 Finding best alpha and l1_ratio for ElasticNet

In [110]:
```
elastic_model = ElasticNet(random_state=42)
```
executed in 5ms, finished 19:50:24 2024-08-12

In [111]:
```
param_grid = {'alpha':[1.02, 2,  3, 4, 5, 7, 10, 11],
              'l1_ratio':[.5, .7, .9, .95, .99, 1]}

elastic_grid_model = GridSearchCV(estimator=elastic_model,

                                  param_grid=param_grid,
                          scoring='neg_root_mean_squared_error',
                          cv=10,
                          n_jobs = -1)
```
executed in 11ms, finished 19:50:24 2024-08-12

alpha:

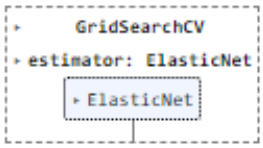
- This parameter controls the overall strength of the regularization.
- When alpha = 0, ElasticNet applies no regularization.
- As the alpha value increases, the strength of the regularization increases. This can help the model become -more resistant to overfitting.

l1_ratio:

- This parameter controls the mix between L1 (Lasso) and L2 (Ridge) regularization.
- When l1_ratio = 1, it means purely L1 (i.e., only Lasso).
- When l1_ratio = 0, it means purely 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 that L1 and L2 regularizations are used equally.

In [112]:
```
elastic_grid_model.fit(X_train_scaled,y_train)
```
executed in 26.9s, finished 19:50:51 2024-08-12

Out[112]:



In [113]:
```
elastic_grid_model.best_params_
```
executed in 9ms, finished 19:50:51 2024-08-12

Out[113]:
```
{'alpha': 1.02, 'l1_ratio': 1}
```

In [114]:
```
train_val(elastic_grid_model, X_train_scaled, y_train, X_test_scaled, y_test)
```
executed in 30ms, finished 19:50:51 2024-08-12

Out[114]:

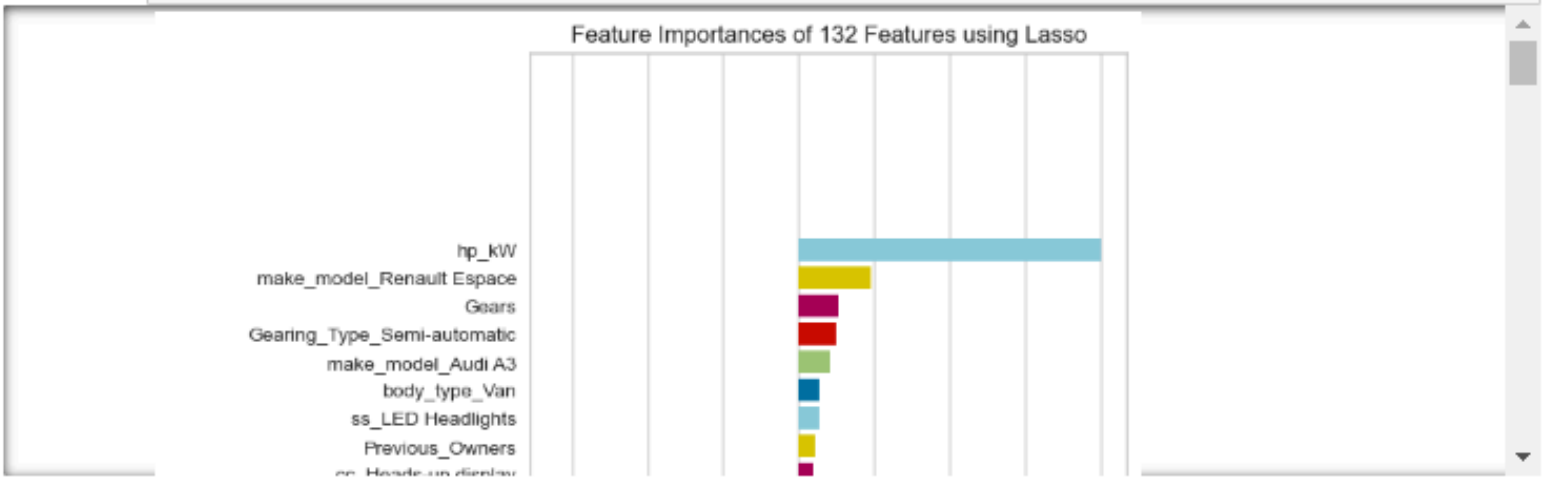
	train	test
R2	0.898	0.896
mae	1414.186	1408.956
mse	3806891.515	3814773.360
rmse	1951.126	1953.144

In [115]:
```
y_pred = elastic_grid_model.predict(X_test_scaled)
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))
```
executed in 13ms, finished 19:50:51 2024-08-12

9 Feature Importance

In [116]:
```
from yellowbrick.model_selection import FeatureImportances
from yellowbrick.features import RadViz

viz = FeatureImportances(Lasso(alpha=1.02), labels=X_train.columns)
visualizer = RadViz(size=(720, 3000))
viz.fit(X_train_scaled, y_train)
viz.show()
```

executed in 2.76s, finished 19:50:53 2024-08-12



In [117]:
```
df_new = df0[["make_model", "hp_kW", "km", "age", "Gearing_Type", "price"]]

# We select the 5 features that have the most impact on the predictions.
# You might wonder why the make_model feature was selected. Upon examining the visual above,
# we noticed that among the features with the most impact on the predictions, the make_model feature includes unique categoric
# (e.g., Audi A3, Audi A1, Renault Espace, etc.), which is why we selected the make_model feature as well.
```

executed in 10ms, finished 19:50:53 2024-08-12

In [118]:
```
df_new.head()
```

executed in 17ms, finished 19:50:54 2024-08-12

Out[118]:

	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 [119]:
```
df_new[df_new["make_model"] == "Audi A2"]
```

executed in 15ms, finished 19:50:54 2024-08-12

Out[119]:

	make_model	hp_kW	km	age	Gearing_Type	price
2614	Audi A2	85.000	26166.000	1.000	Manual	26200

In [120]:
```
df_new.drop(index=[2614], inplace=True)
```

executed in 11ms, finished 19:50:54 2024-08-12

In [121]:
```
df_new = df_new[~(df_new.price > 35000)]
```

executed in 7ms, finished 19:50:54 2024-08-12

In [122]:
```
df_new = pd.get_dummies(df_new)
df_new.head()
```

executed in 49ms, finished 19:50:54 2024-08-12

Out[122]:

	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	mak
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	
3	66.000	73000.000	3.000	14500	True	False	False	False	False	False	
4	66.000	16200.000	3.000	16790	True	False	False	False	False	False	

In [123]:
```
len(df_new)
```
executed in 19ms, finished 19:50:54 2024-08-12

Out[123]: 15419

In [124]:
```
X = df_new.drop(columns=["price"])
y = df_new.price
```
executed in 12ms, finished 19:50:54 2024-08-12

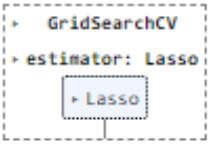
In [125]:
```
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42)
```
executed in 17ms, finished 19:50:54 2024-08-12

In [126]:
```
scaler = MinMaxScaler()
scaler.fit(X_train)

X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```
executed in 36ms, finished 19:50:54 2024-08-12

In [127]:
```
lasso_model = Lasso(random_state=42)

param_grid = {'alpha': alpha_space}

lasso_final_model = GridSearchCV(estimator=lasso_model,
                                 param_grid=param_grid,
                                 scoring='neg_root_mean_squared_error',
                                 cv=10,
                                 n_jobs=-1)
```
executed in 6ms, finished 19:50:54 2024-08-12

In [128]:
```
lasso_final_model.fit(X_train_scaled, y_train)
```
executed in 10.3s, finished 19:51:04 2024-08-12

Out[128]:



In [129]:
```
lasso_final_model.best_params_
```
executed in 8ms, finished 19:51:04 2024-08-12

Out[129]: {'alpha': 0.01}

In [130]:
```
lasso_final_model.best_score_
```
executed in 8ms, finished 19:51:04 2024-08-12

Out[130]: -2239.588352950935

In [131]:
```
train_val(lasso_final_model, X_train_scaled, y_train, X_test_scaled, y_test)
```
executed in 21ms, finished 19:51:04 2024-08-12

Out[131]:

	train	test
R2	0.867	0.877
mae	1611.742	1553.998
mse	5007030.903	4547724.302
rmse	2237.640	2132.539

In [132]:
```
2132 / df_new.price.mean()
```
executed in 8ms, finished 19:51:04 2024-08-12

Out[132]: 0.1233209499596912

In [133]:
```
y_pred = lasso_final_model.predict(X_test_scaled)
fm_R2 = r2_score(y_test, y_pred)
fm_mae = mean_absolute_error(y_test, y_pred)
fm_rmse = np.sqrt(mean_squared_error(y_test, y_pred))
```
executed in 10ms, finished 19:51:04 2024-08-12

10 Compare Models Performance

In [134]:
```
# We assign the metrics obtained from all models to the scores variable. Then, we take the transpose of the DataFrame
# so that the model names appear in the index and the metrics appear as features.


scores = {
    "linear_m": {
        "r2_score": lm_R2,
        "mae": lm_mae,
        "rmse": lm_rmse
    },
    "ridge_m": {
        "r2_score": rm_R2,
        "mae": rm_mae,
        "rmse": rm_rmse
    },
    "lasso_m": {
        "r2_score": lasm_R2,
        "mae": lasm_mae,
        "rmse": lasm_rmse
    },
    "elastic_m": {
        "r2_score": em_R2,
        "mae": em_mae,
        "rmse": em_rmse
    },
    "final_m": {
        "r2_score": fm_R2,
        "mae": fm_mae,
        "rmse": fm_rmse
    }
}
scores = pd.DataFrame(scores).T
scores
```

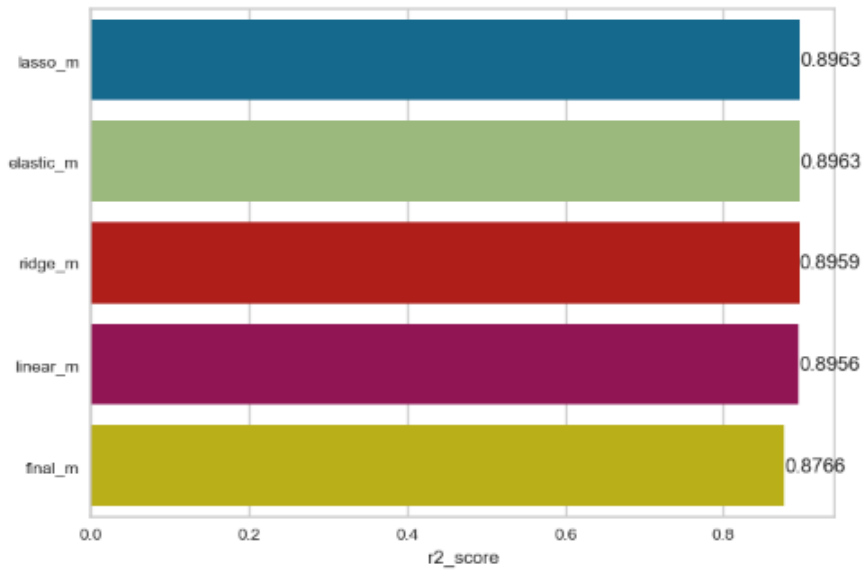
executed in 20ms, finished 19:51:04 2024-08-12

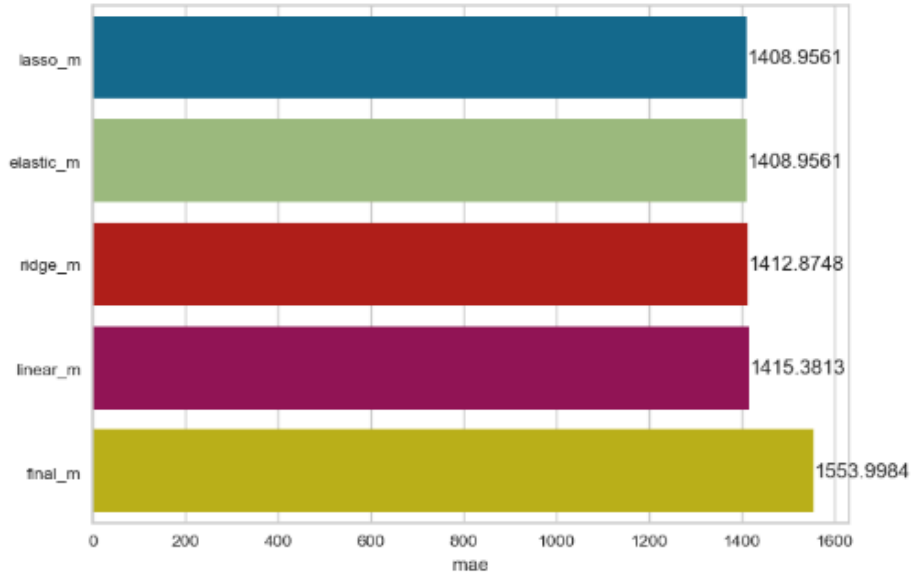
Out[134]:

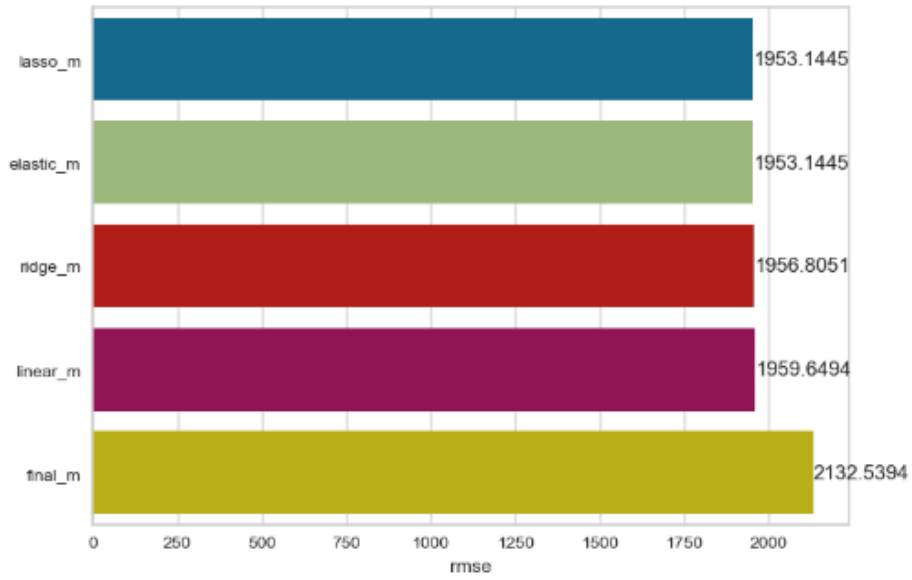
	r2_score	mae	rmse
linear_m	0.896	1415.381	1959.649
ridge_m	0.896	1412.875	1956.805
lasso_m	0.896	1408.956	1953.144
elastic_m	0.896	1408.956	1953.144
final_m	0.877	1553.998	2132.539

In [135]:
```
#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()
            ax.text(width,
                    p.get_y() + p.get_height() / 2,
                    '{:.4f}'.format(width),
                    ha = 'left',
                    va = 'center')
```

executed in 643ms, finished 19:51:05 2024-08-12







11 Prediction

11.1 Prediction with new observation

In [136]:
```
final_scaler = MinMaxScaler()
final_scaler.fit(X)
X_scaled = final_scaler.transform(X)
```
executed in 21ms, finished 19:51:05 2024-08-12

In [137]:
```
lasso_model = Lasso()

param_grid = {'alpha': alpha_space}

final_model = GridSearchCV(estimator=lasso_model,
                           param_grid=param_grid,
                           scoring='neg_root_mean_squared_error',
                           cv=10,
                           n_jobs=-1)
```
executed in 5ms, finished 19:51:05 2024-08-12

In [138]:
```
final_model.fit(X_scaled, y)
```
executed in 14.4s, finished 19:51:19 2024-08-12

Out[138]:

GridSearchCV
estimator: Lasso
Lasso

In [140]:
```
my_dict = {
    "hp_kW": 66,
    "age": 2,
    "km": 17000,
    "make_model": 'Audi A3',
    "Gearing_Type": "Automatic"
}
```
executed in 5ms, finished 19:51:19 2024-08-12

In [141]:
```
my_dict = pd.DataFrame([my_dict])
my_dict
```
executed in 17ms, finished 19:51:19 2024-08-12

Out[141]:

	hp_kW	age	km	make_model	Gearing_Type
0	66	2	17000	Audi A3	Automatic

In [142]:
```
my_dict = pd.get_dummies(my_dict)
my_dict
```
executed in 18ms, finished 19:51:19 2024-08-12

Out[142]:

	hp_kW	age	km	make_model_Audi A3	Gearing_Type_Automatic
0	66	2	17000	True	True

In [143]:
```
X.head(1)
```
executed in 23ms, finished 19:51:19 2024-08-12

Out[143]:

	hp_kW	km	age	make_model_Audi A1	make_model_Audi A3	make_model_Opel Astra	make_model_Opel Corsa	make_model_Opel Insignia	make_model_Renault Clio	make_model_
0	66.000	56013.000	3.000	True	False	False	False	False	False	

In [144]:
```
my_dict = my_dict.reindex(columns=X.columns, fill_value=0)
my_dict
```
executed in 20ms, finished 19:51:19 2024-08-12

Out[144]:

	hp_kW	km	age	make_model_Audi A1	make_model_Audi A3	make_model_Opel Astra	make_model_Opel Corsa	make_model_Opel Insignia	make_model_Renault Clio	make_model_Rena Dus
0	66	17000	2	0	True	0	0	0	0	

In [145]:
```
my_dict = final_scaler.transform(my_dict)
my_dict
```
executed in 13ms, finished 19:51:19 2024-08-12

Out[145]:
```
array([[0.13065327, 0.05362776, 0.66666667, 0.        , 1.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 1.        , 0.        , 0.        ]])
```

In [146]:
```
final_model.predict(my_dict)
```
executed in 10ms, finished 19:51:19 2024-08-12

Out[146]:
```
array([19559.29001107])
```

11.2 Prediction with random samples

In [147]:
```
final_scaler = MinMaxScaler()
final_scaler.fit(X)
X_scaled = final_scaler.transform(X)
```
executed in 27ms, finished 19:51:19 2024-08-12

In [148]:
```
lasso_model = Lasso()

param_grid = {'alpha': alpha_space}

final_model = GridSearchCV(estimator=lasso_model,
                           param_grid=param_grid,
                           scoring='neg_root_mean_squared_error',
                           cv=10,
                           n_jobs=-1)
```
executed in 6ms, finished 19:51:19 2024-08-12

In [149]:
```
final_model.fit(X_scaled, y)
```
executed in 13.2s, finished 19:51:33 2024-08-12

Out[149]:
```
GridSearchCV
estimator: Lasso
Lasso
```

In [150]:
```
final_model.best_estimator_
```
executed in 8ms, finished 19:51:33 2024-08-12

Out[150]:
```
Lasso
Lasso(alpha=0.01)
```

Lasso(alpha=0.01) ?

alpha controls the strength of regularization in Lasso regression. Regularization is used to reduce the model's tendency to overfit.

- If the alpha value is close to 0 (e.g., 0.01), Lasso regularization is reduced, and the model behaves more like linear regression.
- If the alpha value is large, Lasso regularization increases, and more feature coefficients are set to zero, which simplifies the model.

In summary, this means the model provides some protection against overfitting while still allowing the use of most features.

In [151]:
```
random_samples = df_new.sample(n=20)
```
executed in 7ms, finished 19:51:33 2024-08-12

In [152]:
```
X_random = random_samples.drop(columns=["price"])
```
executed in 6ms, finished 19:51:33 2024-08-12

In [153]:
```
X_random = final_scaler.transform(X_random)
```
executed in 8ms, finished 19:51:33 2024-08-12

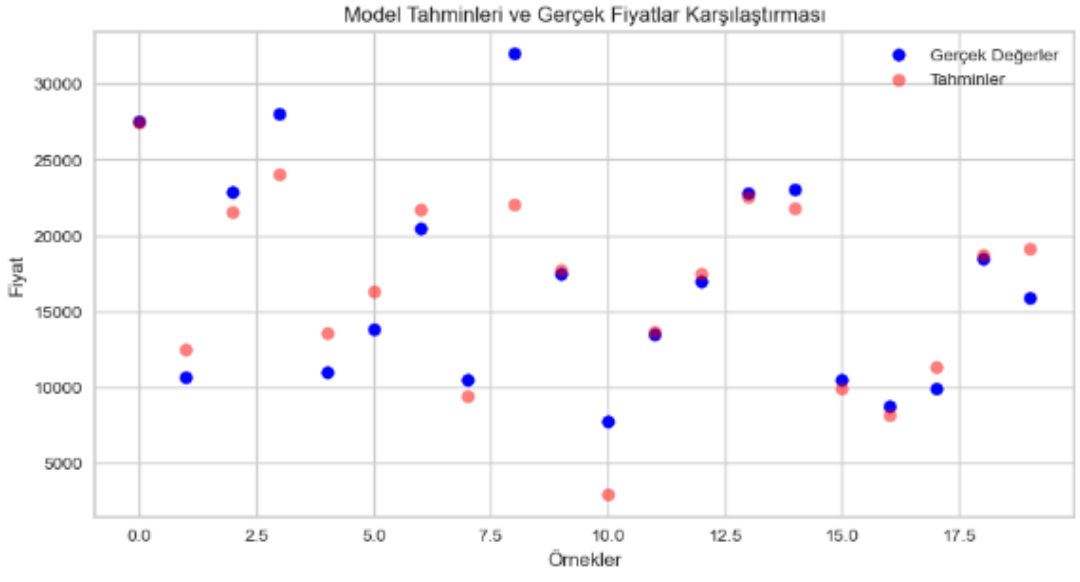
In [154]:
```
predictions = final_model.predict(X_random)
predictions
```
executed in 8ms, finished 19:51:33 2024-08-12

Out[154]:
```
array([27450.93383433, 12480.24434617, 21540.25932273, 23995.6845107 ,
       13598.66680997, 16306.436065  , 21726.69085182,  9433.90107874,
       22029.62259096, 17719.6286586 ,  2908.62294448, 13617.15530054,
       17472.57811012, 22556.56925812, 21804.06871374,  9915.53935413,
        8183.85530134, 11324.46365686, 18734.90680204, 19114.19630132])
```

In [155]:
```
true_labels = random_samples["price"].values
true_labels
```
executed in 7ms, finished 19:51:33 2024-08-12

Out[155]:
```
array([27488, 10689, 22900, 27980, 10990, 13800, 20500, 10480, 31990,
       17495,  7750, 13480, 16980, 22750, 23021, 10480,  8780,  9900,
       18500, 15880], dtype=int64)
```

In [156]:
```
plt.figure(figsize=(10, 5))
plt.scatter(range(len(true_labels)), true_labels, color='blue', label='Gerçek Değerler')
plt.scatter(range(len(predictions)), predictions, color='red', label='Tahminler', alpha=0.5)
plt.title('Model Tahminleri ve Gerçek Fiyatlar Karşılaştırması')
plt.xlabel('Örnekler')
plt.ylabel('Fiyat')
plt.legend()
plt.show()
```
executed in 262ms, finished 19:51:33 2024-08-12



In [157]:

```
plt.figure(figsize=(10, 6))
plt.scatter(true_labels, predictions, color='darkorange', label='Tahminler')
plt.plot([min(true_labels), max(true_labels)], [min(true_labels), max(true_labels)], 'k--', lw=2, label='İdeal')
plt.xlabel('Gerçek Değerler')
plt.ylabel('Tahmin Edilen Değerler')
plt.title('Gerçek ve Tahmin Edilen Fiyat Karşılaştırması')
plt.legend()
plt.show()
```

executed in 202ms, finished 19:51:33 2024-08-12