In [1]:
```
import warnings
warnings.filterwarnings('ignore')
```

Bonus: The `Cars Price` dataset revisited

🧑‍🍳 During `Machine Learning > 02 - Prepare the Dataset`, we discovered that to run Machine Learning Algorithms properly, you need to feed them with ***cleaned datasets***.

▶ 📝 *Reminders about the Data Preprocessing Workflow* 📝

🚗 We had already worked on a simplified version of the *Cars' Price* dataset.

🎯 The goal of this recap is to build an optimal pipeline to ***predict the price of cars according to their specificities***:

1. We will need a *Preprocessing Pipeline*...
2. ... that we can *chain with a Scikit-Learn Estimator*
3. And go further by:
   - running a *FeaturePermutation*
   - optimizing the hyperparameters with a *GridSearchCV* or a *RandomizedSearchCV*

In [3]:
```
# DATA MANIPULATION
import numpy as np
import pandas as pd
pd.set_option("display.max_columns",None) # Show all columns of a Pandas DataFrame

# DATA VISUALISATION
import matplotlib.pyplot as plt
import seaborn as sns

# STATISTICS
from statsmodels.graphics.gofplots import qqplot
# This function plots your sample distribution against a Normal distribution,
# to check whether your sample is normally distributed or not
```

(1) The dataset

In [4]:
```
cars = pd.read_csv("https://wagon-public-datasets.s3.amazonaws.com/Machine%20Learn
cars.drop(columns = ['car_ID'], inplace = True)
cars.head()
```

Out[4]:

	symboling	CarName	fueltype	aspiration	doornumber	carbody	drivewheel	enginelocat
0	3	alfa-romero giulia	gas	std	two	convertible	rwd	fr
1	3	alfa-romero stelvio	gas	std	two	convertible	rwd	fr
2	1	alfa-romero Quadrifoglio	gas	std	two	hatchback	rwd	fr
3	2	audi 100 ls	gas	std	four	sedan	fwd	fr
4	2	audi 100ls	gas	std	four	sedan	4wd	fr

(1.1) Basic Info

❓ How many cars do we have ❓

In [5]:
```
print(f"There are {cars.shape[0]} cars in the dataset")
```

```
There are 205 cars in the dataset
```

❓ Inspect the types of your columns ❓

In [6]:
```
cars.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 25 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   symboling         205 non-null    int64  
 1   CarName           205 non-null    object 
 2   fueltype          205 non-null    object 
 3   aspiration        205 non-null    object 
 4   doornumber        205 non-null    object 
 5   carbody           205 non-null    object 
 6   drivewheel        205 non-null    object 
 7   enginelocation    205 non-null    object 
 8   wheelbase         205 non-null    float64
 9   carlength         205 non-null    float64
 10  carwidth          205 non-null    float64
 11  carheight         205 non-null    float64
 12  curbweight        205 non-null    int64  
 13  enginetype        205 non-null    object 
 14  cylindernumber    205 non-null    object 
 15  enginesize        205 non-null    int64  
 16  fuelsystem        205 non-null    object 
 17  boreratio         205 non-null    float64
 18  stroke            205 non-null    float64
 19  compressionratio  205 non-null    float64
 20  horsepower        205 non-null    int64  
 21  peakrpm           205 non-null    int64  
 22  citympg           205 non-null    int64  
 23  highwaympg        205 non-null    int64  
 24  price             205 non-null    float64
dtypes: float64(8), int64(7), object(10)
memory usage: 40.2+ KB
```

(1.2) Prerequisites

(1.2.1) Anomalies in the dataset

❓ If you carefully look at the columns with *object*, which columns could/should be converted to numerical columns ❓

👉 Convert them.

▶ *Hint*

In [7]:
```
cars["cylindernumber"] = cars["cylindernumber"].map({"four":4,
                                                   "six":6,
                                                   "five":5,
                                                   "eight":8,
                                                   "two":2,
                                                   "twelve":12,
                                                   "three":3})
```

In [8]:
```
cars["doornumber"].value_counts()
```

Out[8]:
```
four    115
two      90
Name: doornumber, dtype: int64
```

(1.2.2) Removing duplicates

❓ How many duplicated rows do we have in this dataset (if so, get rid of any duplicated row) ❓

In [9]:
```
print(f"number of duplicated rows = {cars.duplicated().sum()}")
print("-"*50)

print(f"cars' shape before removing duplicates = {cars.shape}")
print("-"*50)

cars.drop_duplicates(inplace = True)
print(f"cars' shape after removing duplicates = {cars.shape}")
```

```
number of duplicated rows = 0
--------------------------------------------------
cars' shape before removing duplicates = (205, 25)
--------------------------------------------------
cars' shape after removing duplicates = (205, 25)
```

(1.2.3) Handling Missing Values

❓ How many NaN do we have ❓

In [10]:
```
cars.isna().sum()
```

Out[10]:
```
symboling           0
CarName             0
fueltype            0
aspiration          0
doornumber          0
carbody             0
drivewheel          0
enginelocation      0
wheelbase           0
carlength           0
carwidth            0
carheight           0
curbweight          0
enginetype          0
cylindernumber      0
enginesize          0
fuelsystem          0
boreratio           0
stroke              0
compressionratio    0
horsepower          0
peakrpm             0
citympg             0
highwaympg          0
price               0
dtype: int64
```

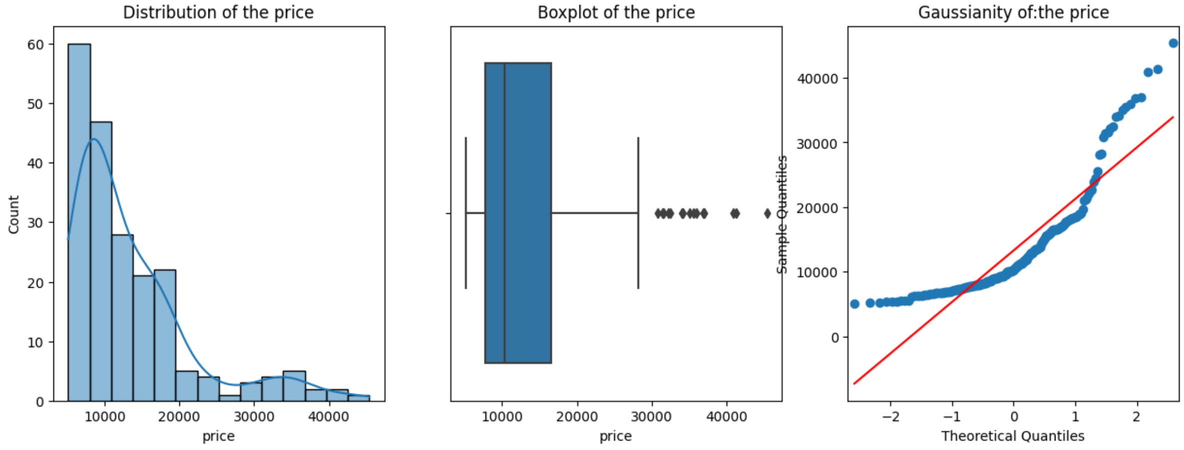
▶ *Answer*

(1.3) Having a glance at your target (`cars' price`)

❓ How does your target look like in terms of *Distribution*, *Outliers*, *Gaussianity* ❓

▶ *Code answer*

In [11]:
```
variable = 'price'
y = cars[f"{variable}"]

fig, ax = plt.subplots(1,3,figsize=(15,5))

ax[0].set_title(f"Distribution of the {variable}")
sns.histplot(data = cars, x = f"{variable}", kde=True, ax = ax[0])

ax[1].set_title(f"Boxplot of the {variable}")
sns.boxplot(data = cars, x = f"{variable}", ax=ax[1])

ax[2].set_title(f"Gaussianity of:the {variable}")
qqplot(cars[f"{variable}"],line='s',ax=ax[2]);
```



In [12]:
```
cars.skew()
```

Out[12]:
```
symboling           0.211072
wheelbase           1.050214
carlength           0.155954
carwidth            0.904003
carheight           0.063123
curbweight          0.681398
cylindernumber      2.817459
enginesize          1.947655
boreratio           0.020156
stroke             -0.689705
compressionratio    2.610862
horsepower          1.405310
peakrpm             0.075159
citympg             0.663704
highwaympg          0.539997
price               1.777678
dtype: float64
```

(2) Preprocessing the features with a Pipeline

🎬 Great, you have an overview of how the cars are distributed.

🔥 It's time to build a *preprocessing pipeline* that we will, in a humble way, call the *preprocessor*.

▶ 🤔 How to deal with the *CarName* to predict the price of a car ?

In [13]:
```
X = cars.drop(columns = ["price", "CarName"])
```

In [14]:
```
# PIPELINE AND COLUMNTRANSFORMER
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.compose import ColumnTransformer, make_column_transformer, make_colum
from sklearn import set_config; set_config(display="diagram")

# IMPUTERS
from sklearn.impute import SimpleImputer

# SCALERS
from sklearn.preprocessing import RobustScaler, StandardScaler, MinMaxScaler

# ENCODER
from sklearn.preprocessing import OneHotEncoder
```

(2.1) Numerical Pipeline

❓ Store the numerical features in a `X_num` variable ❓

In [15]:
```
X_num = X.select_dtypes(exclude = ['object'])
X_num.head()
```

Out[15]:

	symboling	wheelbase	carlength	carwidth	carheight	curbweight	cylindernumber	enginesize
0	3	88.6	168.8	64.1	48.8	2548	4	130
1	3	88.6	168.8	64.1	48.8	2548	4	130
2	1	94.5	171.2	65.5	52.4	2823	6	152
3	2	99.8	176.6	66.2	54.3	2337	4	109
4	2	99.4	176.6	66.4	54.3	2824	5	136

❓ Create a `num_transformer` pipeline to deal with numerical features ❓

▶ 📚 *Reminder about scalers*

MINIMAL SOLUTION WITH ONLY ONE SCALER

In [16]:
```
num_transformer_simplified = make_pipeline(
                    SimpleImputer(strategy = "median"),
                    RobustScaler()
                )

num_transformer_simplified
```

Out[16]:

```
▸ Pipeline
  ▸ SimpleImputer
  ▸ RobustScaler
```

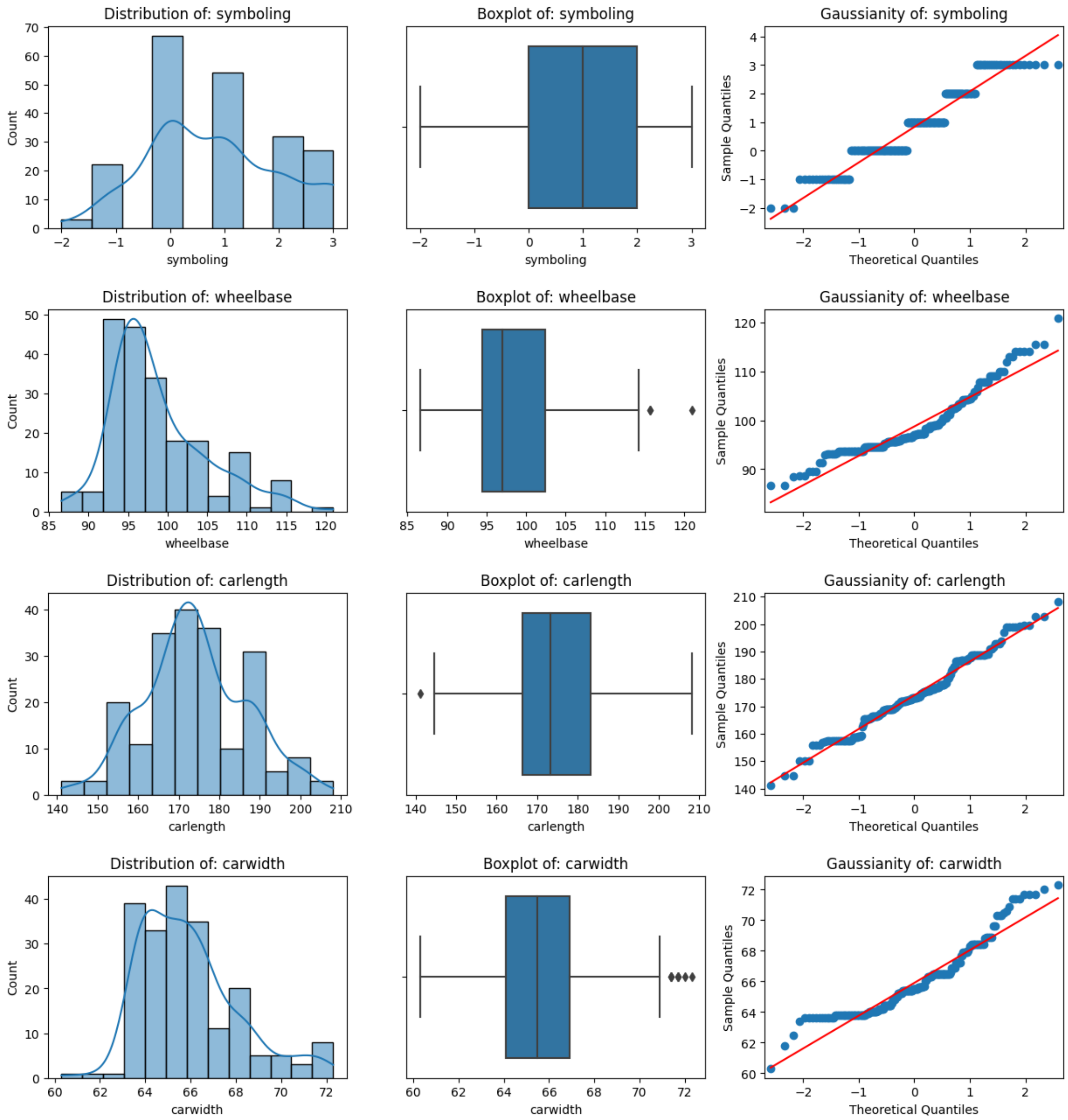
In [17]:
```
pd.DataFrame(num_transformer_simplified.fit_transform(X_num), columns=X_num.column
```

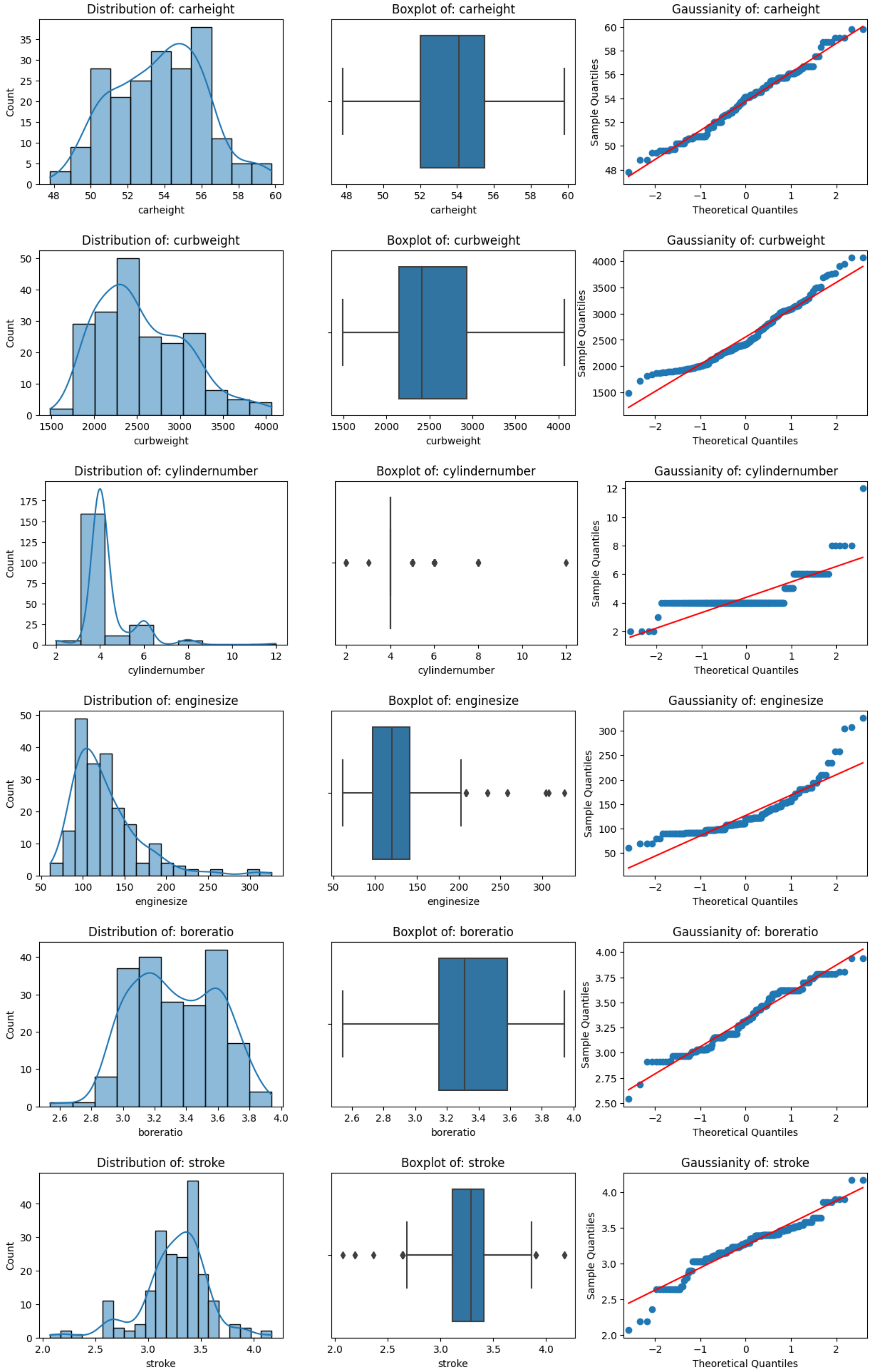
Out[17]:

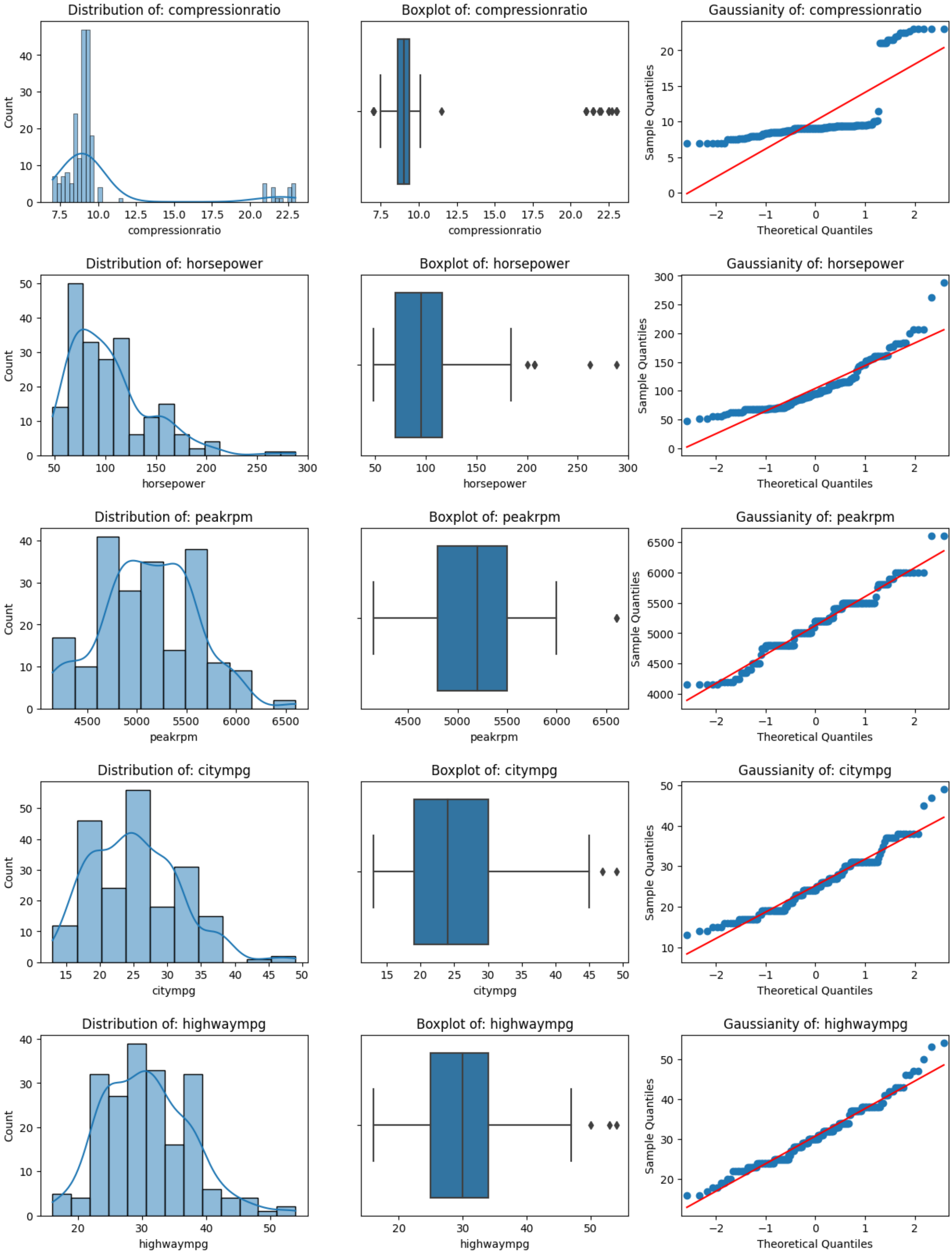
	symboling	wheelbase	carlength	carwidth	carheight	curbweight	cylindernumber	enginesiz
0	1.0	-1.063291	-0.261905	-0.500000	-1.514286	0.169620	0.0	0.22727
1	1.0	-1.063291	-0.261905	-0.500000	-1.514286	0.169620	0.0	0.22727
2	0.0	-0.316456	-0.119048	0.000000	-0.485714	0.517722	2.0	0.72727
3	0.5	0.354430	0.202381	0.250000	0.057143	-0.097468	0.0	-0.25000
4	0.5	0.303797	0.202381	0.321429	0.057143	0.518987	1.0	0.36363

ADVANCED SOLUTION WITH THREE DIFFERENT SCALERS

In [18]:
```python
for numerical_feature in X_num.columns:

    # Creating three subplots per numerical_feature
    fig, ax =plt.subplots(1,3,figsize=(15,3))

    # Histogram to get an overview of the distribution of each numerical_feature
    ax[0].set_title(f"Distribution of: {numerical_feature}")
    sns.histplot(data = X_num, x = numerical_feature, kde=True, ax = ax[0])

    # Boxplot to detect outliers
    ax[1].set_title(f"Boxplot of: {numerical_feature}")
    sns.boxplot(data = X_num, x = numerical_feature, ax=ax[1])

    # Analyzing whether a feature is normally distributed or not
    ax[2].set_title(f"Gaussianity of: {numerical_feature}")
    qqplot(X_num[numerical_feature],line='s',ax=ax[2]);
```




Distribution of: carheight
Count
carheight
Boxplot of: carheight
carheight
Gaussianity of: carheight
Sample Quantiles
Theoretical Quantiles
Distribution of: curbweight
Count
curbweight
Boxplot of: curbweight
curbweight
Gaussianity of: curbweight
Sample Quantiles
Theoretical Quantiles
Distribution of: cylindernumber
Count
cylindernumber
Boxplot of: cylindernumber
cylindernumber
Gaussianity of: cylindernumber
Sample Quantiles
Theoretical Quantiles
Distribution of: enginesize
Count
enginesize
Boxplot of: enginesize
enginesize
Gaussianity of: enginesize
Sample Quantiles
Theoretical Quantiles
Distribution of: boreratio
Count
boreratio
Boxplot of: boreratio
boreratio
Gaussianity of: boreratio
Sample Quantiles
Theoretical Quantiles
Distribution of: stroke
Count
stroke
Boxplot of: stroke
stroke
Gaussianity of: stroke
Sample Quantiles
Theoretical Quantiles



In [19]:
```
features_robust = ["cylindernumber", "enginesize", "stroke", "compressionratio", "|
features_standard = ["symboling", "wheelbase", "carlength", "carwidth", "carheight
                     "curbweight", "boreratio", "highwaympg"]
features_minmax = ["peakrpm", "citympg"]
features_already_scaled = []
```

In [20]:
```
# Checking what we didn't forget any numerical feature
X_num.shape[-1] == len(features_robust + features_standard + features_minmax + fea
```

Out[20]:
```
True
```

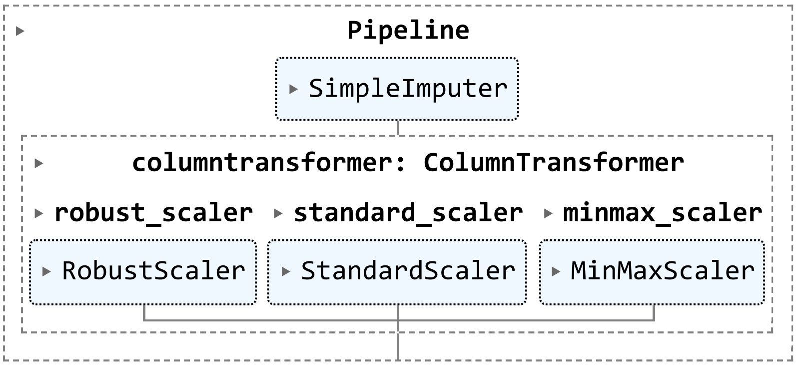
⚖️ Let's use a ColumnTransformer that we will simply name `scalers` to use three different scalers on the numerical features to be scaled:

In [21]:
```
scalers = ColumnTransformer(
                                [
                                    ("robust_scaler", RobustScaler(), features_robu
                                    ("standard_scaler", StandardScaler(), features_
                                    ("minmax_scaler", MinMaxScaler(), features_mini
                                ])
```

🛠️ Now, let's chain a *SimpleImputer* with this `scalers` to create a Pipeline called `num_transformer`

In [22]:
```
num_transformer = make_pipeline(
                    SimpleImputer(strategy = "median"),
                    scalers
                )

num_transformer
```

Out[22]:



👉 Try to apply this `num_transformer` on `X_num`

In [23]:
```
# X_num_scaled = pd.DataFrame(num_transformer.fit_transform(X_num))
# X_num_scaled.head()
```

🤔 Why doesn't it work ? `scalers` is a ColumnTransformer which requires the columns' names... Unfortunately, after fitting an Imputer, they disappeared!

We have to build an `inputer_with_name`

In [24]:
```
# OPTION 1) We 'pipe' a new class `ColumnNameExtractor` to preserve the columns' no
from sklearn.base import BaseEstimator, TransformerMixin


class  ColumnNameExtractor(BaseEstimator, TransformerMixin):
    def __init__(self, columns):
        self.columns = columns

    def fit(self, *_):
        return self

    def transform(self, X, *_):
        return pd.DataFrame(X, columns = self.columns)

    def fit_transform(self, X, *_):
        return pd.DataFrame(X, columns = self.columns)

imputer_with_name = make_pipeline(
    SimpleImputer(strategy="median"),
    ColumnNameExtractor(features_robust + features_standard + features_minmax +
                        features_already_scaled),
```

```
)

pd.DataFrame(imputer_with_name.fit_transform(X_num)).head()
```

Out[24]:

	cylindernumber	enginesize	stroke	compressionratio	horsepower	symboling	wheelbase	carl
0	3.0	88.6	168.8	64.1	48.8	2548.0	4.0	
1	3.0	88.6	168.8	64.1	48.8	2548.0	4.0	
2	1.0	94.5	171.2	65.5	52.4	2823.0	6.0	
3	2.0	99.8	176.6	66.2	54.3	2337.0	4.0	
4	2.0	99.4	176.6	66.4	54.3	2824.0	5.0	

In [25]:
```
# OPTION 2) CUSTOMIZING THE SIMPLE IMPUTER CLASS -  SimpleImputer does not have ge

class CustomSimpleImputer(SimpleImputer):
    def fit(self, X, *args, **kwargs):
        self.columns = X.columns
        return super().fit(X, *args, **kwargs)

    def transform(self, *args, **kwargs):
        return pd.DataFrame(super().transform(*args, **kwargs), columns=self.colum

    def fit_transform(self, *args, **kwargs):
        return pd.DataFrame(super().fit_transform(*args, **kwargs), columns=self.c

imputer_with_name = CustomSimpleImputer(strategy='median')

pd.DataFrame(imputer_with_name.fit_transform(X_num)).head()
```

Out[25]:

	symboling	wheelbase	carlength	carwidth	carheight	curbweight	cylindernumber	enginesize
0	3.0	88.6	168.8	64.1	48.8	2548.0	4.0	130.0
1	3.0	88.6	168.8	64.1	48.8	2548.0	4.0	130.0
2	1.0	94.5	171.2	65.5	52.4	2823.0	6.0	152.0
3	2.0	99.8	176.6	66.2	54.3	2337.0	4.0	109.0
4	2.0	99.4	176.6	66.4	54.3	2824.0	5.0	136.0

In [26]:
```
# OPTION 3) we override existing class

from sklearn.impute import SimpleImputer
SimpleImputer.get_feature_names_out = (lambda self, names=None:
                                       self.feature_names_in_)
imputer_with_name = SimpleImputer(strategy='median')
pd.DataFrame(imputer_with_name.fit_transform(X_num), columns = imputer_with_name.g
```

Out[26]:

	symboling	wheelbase	carlength	carwidth	carheight	curbweight	cylindernumber	enginesi
0	3.0	88.6	168.8	64.1	48.8	2548.0	4.0	13
1	3.0	88.6	168.8	64.1	48.8	2548.0	4.0	13
2	1.0	94.5	171.2	65.5	52.4	2823.0	6.0	15
3	2.0	99.8	176.6	66.2	54.3	2337.0	4.0	10
4	2.0	99.4	176.6	66.4	54.3	2824.0	5.0	13
...	...	...	...	...	...	...	...	
200	-1.0	109.1	188.8	68.9	55.5	2952.0	4.0	14
201	-1.0	109.1	188.8	68.8	55.5	3049.0	4.0	14
202	-1.0	109.1	188.8	68.9	55.5	3012.0	6.0	17
203	-1.0	109.1	188.8	68.9	55.5	3217.0	6.0	14
204	-1.0	109.1	188.8	68.9	55.5	3062.0	4.0	14

205 rows × 15 columns

Now we can pipe Imputer and Column Transformer

In [27]:
```
num_transformer = make_pipeline(
                    CustomSimpleImputer(strategy='median'),
                    ColumnTransformer(
                                [
                                    ("robust_scaler", RobustScaler(), features_robu
                                    ("standard_scaler", StandardScaler(), features_
                                    ("minmax_scaler", MinMaxScaler(), features_minm
                                ])
                )

num_transformer.fit(X_num)
num_transformer.transform(X_num)
```

Out[27]:
```
array([[ 0.        ,  0.22727273, -2.03333333, ..., -0.54605874,
         0.34693878,  0.22222222],
       [ 0.        ,  0.22727273, -2.03333333, ..., -0.54605874,
         0.34693878,  0.22222222],
       [ 2.        ,  0.72727273,  0.6       , ..., -0.69162706,
         0.34693878,  0.16666667],
       ...,
       [ 2.        ,  1.20454545, -1.4       , ..., -1.12833203,
         0.55102041,  0.13888889],
       [ 2.        ,  0.56818182,  0.36666667, ..., -0.54605874,
         0.26530612,  0.36111111],
       [ 0.        ,  0.47727273, -0.46666667, ..., -0.83719538,
         0.51020408,  0.16666667]])
```

🤯 The Column Transformer lost column names again: Let's keep column names also with a CustomColumnTransformer

In [28]:
```
# --------------------------------------- #
#   CUSTOMIZED COLUMN TRANSFORMER        #
# --------------------------------------- #
# Nice class to keep the columns'names before fitting a model
```

```
class CustomColumnTransformer(ColumnTransformer):

    def fit(self, *args, **kwargs):
        return super().fit(*args, **kwargs)

    def transform(self, X, *args, **kwargs):
        return pd.DataFrame(super().transform(X, *args, **kwargs), columns=self.ge

    def fit_transform(self, X, *args, **kwargs):
        return pd.DataFrame(super().fit_transform(X, *args, **kwargs), columns=sel
```

In [29]:

```
num_transformer = make_pipeline(
                    CustomSimpleImputer(strategy = "median"),
                    CustomColumnTransformer(
                                [
                                    ("robust_scaler", RobustScaler(), features_robu
                                    ("standard_scaler", StandardScaler(), features_
                                    ("minmax_scaler", MinMaxScaler(), features_mini
                                ])
                )

num_transformer.fit(X_num)
num_transformer.transform(X_num)
num_transformer.fit_transform(X_num)
```

Out[29]:

	robust_scaler__cylindernumber	robust_scaler__enginesize	robust_scaler__stroke	robust_scaler__
0	0.0	0.227273	-2.033333	
1	0.0	0.227273	-2.033333	
2	2.0	0.727273	0.600000	
3	0.0	-0.250000	0.366667	
4	1.0	0.363636	0.366667	
...	...	...	...	
200	0.0	0.477273	-0.466667	
201	0.0	0.477273	-0.466667	
202	2.0	1.204545	-1.400000	
203	2.0	0.568182	0.366667	
204	0.0	0.477273	-0.466667	

205 rows × 15 columns

(2.2) Categorical Pipeline

❓ Store the categorical features in a variable called `cars_cat` ❓

In [30]:

```
X_cat = X.select_dtypes(include=['object'])
X_cat.head()
```

Out[30]:

	fueltype	aspiration	doornumber	carbody	drivewheel	enginelocation	enginetype	fuelsyst
0	gas	std	two	convertible	rwd	front	dohc	m
1	gas	std	two	convertible	rwd	front	dohc	m
2	gas	std	two	hatchback	rwd	front	ohcv	m
3	gas	std	four	sedan	fwd	front	ohc	m
4	gas	std	four	sedan	4wd	front	ohc	m

In [31]:
```
cat_features = list(X_cat.columns)
cat_features
```

Out[31]:
```
['fueltype',
 'aspiration',
 'doornumber',
 'carbody',
 'drivewheel',
 'enginelocation',
 'enginetype',
 'fuelsystem']
```

❓ Check how many columns you would end up with, if you decide to One Hot Encode them all. Is it a reasonable number ❓

In [32]:
```
unique_occurences = {cat_feature:
                                 len(X_cat[cat_feature].value_counts())
                                 for cat_feature in X_cat.columns}

unique_occurences = pd.DataFrame.from_dict(unique_occurences,
                                           orient = "index",
                                           columns = ["unique_occurenc

unique_occurences = unique_occurences.sort_values(by = "unique_occurences",
                                                  ascending = False)
```

In [33]:
```
print(unique_occurences)
```

```
                unique_occurences
fuelsystem                      8
enginetype                      7
carbody                         5
drivewheel                      3
fueltype                        2
aspiration                      2
doornumber                      2
enginelocation                  2
```

In [34]:
```
multiple_cat = list(unique_occurences[unique_occurences.unique_occurences > 2].ind
multiple_cat
```

Out[34]:
```
['fuelsystem', 'enginetype', 'carbody', 'drivewheel']
```

In [35]:
```
binary_cat = list(unique_occurences[unique_occurences.unique_occurences <= 2].inde
binary_cat
```

Out[35]:
```
['fueltype', 'aspiration', 'doornumber', 'enginelocation']
```

In [36]:

```
columns_generated_by_multiple_ohe = unique_occurences.loc[multiple_cat].sum()[0]
columns_generated_by_binary_ohe = len(binary_cat)
columns_ohe = columns_generated_by_multiple_ohe + columns_generated_by_binary_ohe



print(f"If we are to One-Hot-Encode all the categorical columns of this cars' datas
```

```
If we are to One-Hot-Encode all the categorical columns of this cars' dataset, we
will generate 23 + 4 = 27 columns
```

❓ Create a `cat_transformer` pipeline to deal with categorical features ❓

In [37]:

```
cat_transformer = make_pipeline(
                    SimpleImputer(strategy = "most_frequent"),
                    OneHotEncoder(sparse = False, handle_unknown = "ignore", drop=
                )

cat_transformer
```

Out[37]:

```
▸ Pipeline
  ▸ SimpleImputer
  ▸ OneHotEncoder
```

👉 Try to _fittransform_ this `cat_transformer` on `X_cat`

In [38]:

```
X_cat_encoded = pd.DataFrame(cat_transformer.fit_transform(X_cat))
X_cat_encoded
```

Out[38]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	2
0	1.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
1	1.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
2	1.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0
3	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0
4	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
200	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0
201	1.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0
202	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0
203	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0
204	1.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0

205 rows × 27 columns

In [39]:

```
# cutstom OHE that passes column names
from sklearn.preprocessing import OneHotEncoder
```

```
class CustomOHE(OneHotEncoder):
    def fit(self, *args, **kwargs):
        return super().fit(*args, **kwargs)

    def transform(self,*args, **kwargs):
        return pd.DataFrame(super().transform(*args, **kwargs), columns=self.get_f

    def fit_transform(self,*args, **kwargs):
        return pd.DataFrame(super().fit_transform(*args, **kwargs), columns=self.g

cat_transformer = make_pipeline(
                    CustomSimpleImputer(strategy = "most_frequent"),
                    CustomOHE(sparse = False, handle_unknown = "ignore", drop='if_
                )
X_cat_encoded = pd.DataFrame(cat_transformer.fit_transform(X_cat))
X_cat_encoded.shape
```

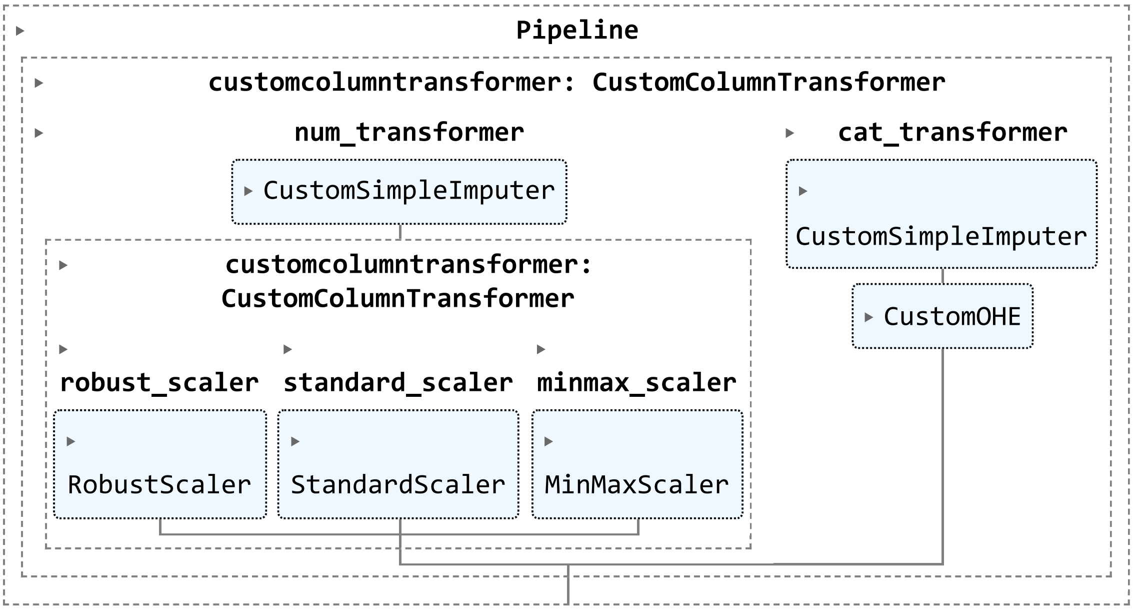
Out[39]: (205, 27)

(2.3) Full Preprocessor

❓ Create the `preprocessor` which combines the `num_transformer` and the `cat_transformer` ❓

In [40]:
```
preprocessor = make_pipeline(
                CustomColumnTransformer([
                    ("num_transformer", num_transformer, make_column_selector(dtyp
                    ("cat_transformer", cat_transformer, make_column_selector(dtyp
                    ])
                )

preprocessor
```

Out[40]:



❓ Try to _fit*transform* the full `preprocessor` on `X` to make sure your full pipeline works properly ❓

In [41]:
```
fully_preprocessed_dataset = pd.DataFrame(preprocessor.fit_transform(X))
fully_preprocessed_dataset
```

Out[41]:

	num_transformer__robust_scaler__cylindernumber	num_transformer__robust_scaler__enginesize
0	0.0	0.227273
1	0.0	0.227273
2	2.0	0.727273
3	0.0	-0.250000
4	1.0	0.363636
...	...	...
200	0.0	0.477273
201	0.0	0.477273
202	2.0	1.204545
203	2.0	0.568182
204	0.0	0.477273

205 rows × 42 columns

(3) Full pipeline with a Regression Model

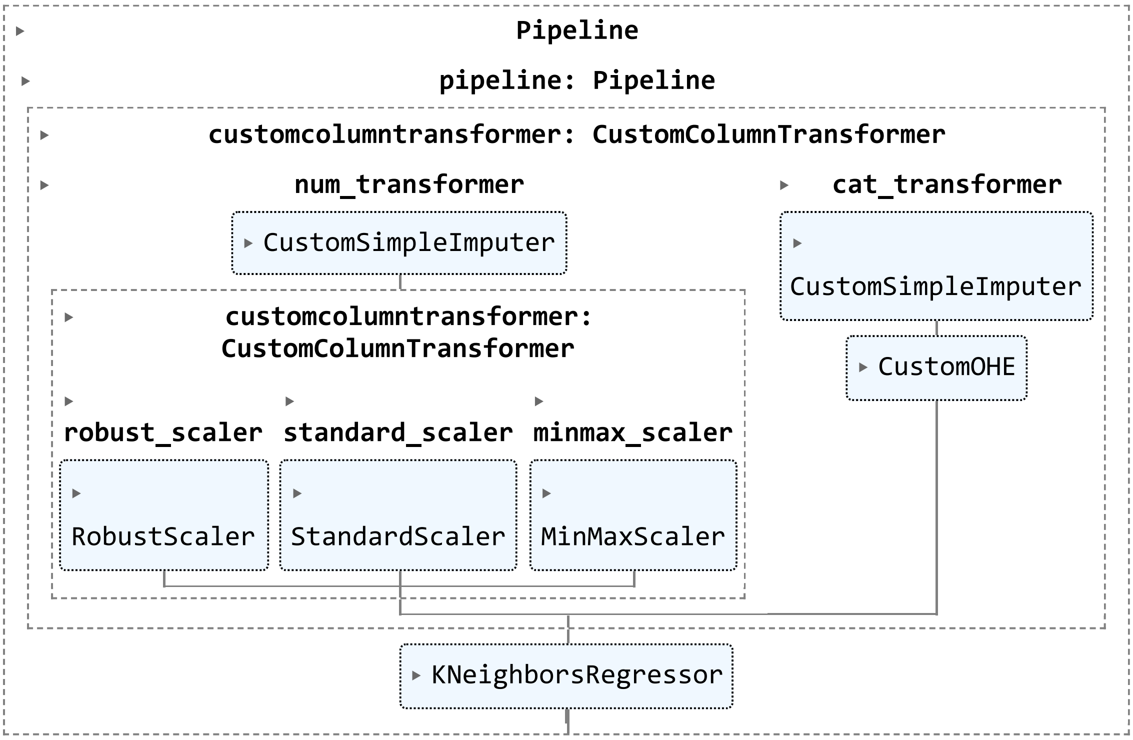
🚀 We can now try different regression model pipelined with the preprocessor 🚀

❓ Create a function that will create a Pipeline with the `preprocessor` and a regression model ❓

In [42]:
```
def cars_regression_models(regression_model):
    piped_regressor = make_pipeline(preprocessor, regression_model)
    return piped_regressor
```

In [43]:
```
# Here is an example of a pipelined regressor

from sklearn.neighbors import KNeighborsRegressor
cars_regression_models(KNeighborsRegressor())
```

Out[43]:



❓ Testing different pipelined regression models ❓

🤓 Do not forget to refer to [Scikit-Learn - Choosing the right estimator](#).

In [44]:

```
# LINEAR MODELS
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet, SGDRe

# NEIGHBORS
from sklearn.neighbors import KNeighborsRegressor

# SVM
from sklearn.svm import SVR

# TREES AND ENSEMBLE METHODS
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, AdaBoostRegressor, GradientBoo
```

In [45]:

```
models = [LinearRegression(),
          Ridge(),
          Lasso(),
          ElasticNet(),
          SGDRegressor(),
          KNeighborsRegressor(),
          SVR(kernel = "linear"),
          SVR(kernel = "poly", degree = 2),
          SVR(kernel = "poly", degree = 3),
          SVR(kernel = "rbf"),
          DecisionTreeRegressor(),
          RandomForestRegressor(),
          AdaBoostRegressor(),
          GradientBoostingRegressor()
]
```

In [46]:

```
models_names = ["linear_regression",
                "ridge",
                "lasso",
                "elastic_net",
                "sgd_regressor",
```

```
                "kneighbors_regressor",
                "SVR_linear",
                "SVR_poly_two",
                "SVR_poly_three",
                "SVR_rbf",
                "decision_tree_regressor",
                "random_forest_regressor",
                "ada_boost_regressor",
                "gradient_boosting_regressor"
]
```

❓ Evaluating the pipelined models: which pipelined regressor performed the best ❓

In [47]:
```
from sklearn.model_selection import train_test_split
```

In [48]:
```
%%time

X_train, X_test, y_train, y_test = train_test_split(X, y)
different_test_scores = []

for model_name, model in zip(models_names, models):

    temp_piped_regressor = cars_regression_models(model)
    temp_piped_regressor.fit(X_train, y_train)
    different_test_scores.append(temp_piped_regressor.score(X_test, y_test))


comparing_regression_models_cars = pd.DataFrame(list(zip(models_names, different_te
                                                columns =['model_name', 'test_score

round(comparing_regression_models_cars.sort_values(by = "test_score", ascending = 
```

```
CPU times: user 1.86 s, sys: 177 ms, total: 2.04 s
Wall time: 2.11 s
```

Out[48]:

	model_name	test_score
11	random_forest_regressor	0.85
2	lasso	0.84
12	ada_boost_regressor	0.84
0	linear_regression	0.83
1	ridge	0.83
13	gradient_boosting_regressor	0.81
4	sgd_regressor	0.80
3	elastic_net	0.77
5	kneighbors_regressor	0.69
10	decision_tree_regressor	0.63
6	SVR_linear	0.16
8	SVR_poly_three	-0.01
9	SVR_rbf	-0.02
7	SVR_poly_two	-0.02

🧑‍🏫 You could even cross-validate all these pipelined models... !

In [49]:
```
from sklearn.model_selection import cross_val_score
```

In [50]:
```
%%time

different_test_scores_cv = []

for model_name, model in zip(models_names, models):

    temp_piped_regressor = cars_regression_models(model)
    different_test_scores_cv.append(cross_val_score(temp_piped_regressor, X, y).mea

comparing_regression_models_cars_cv = pd.DataFrame(list(zip(models_names, differen
                                                   columns = ['model_name', 'cross_va

round(comparing_regression_models_cars_cv.sort_values(by = "cross_val_score", ascen
```

```
CPU times: user 10.2 s, sys: 13.6 ms, total: 10.2 s
Wall time: 9.89 s
```

Out[50]:

	model_name	cross_val_score
11	random_forest_regressor	0.85
2	lasso	0.84
12	ada_boost_regressor	0.84
0	linear_regression	0.83
1	ridge	0.83
13	gradient_boosting_regressor	0.81
4	sgd_regressor	0.80
3	elastic_net	0.77
5	kneighbors_regressor	0.69
10	decision_tree_regressor	0.63
6	SVR_linear	0.16
8	SVR_poly_three	-0.01
9	SVR_rbf	-0.02
7	SVR_poly_two	-0.02

🏁 Congratulations!

💾 Don't forget to git add/commit/push your notebook...

🚀 You are now a master at `Pipeline` and `ColumnTransformer` !