Car Prices

🎯 The goal of this challenge is to prepare a dataset and apply some feature selection techniques that you have learned so far.

🚘 We are dealing with a dataset about cars and we would like to predict whether a car is expensive or cheap.

In [1]:
```
# Data manipulation
import numpy as np
import pandas as pd
# Data visualisation
import matplotlib.pyplot as plt
import seaborn as sns
# Checking whether a numerical feature has a normal distribution or not
from statsmodels.graphics.gofplots import qqplot
```

In [2]:
```
url = "https://wagon-public-datasets.s3.amazonaws.com/Machine%20Learning%20Dataset
```

❓ Go ahead and load the CSV into a dataframe called `df`.

In [13]:
```
df = pd.read_csv(url)
df.head(2)
```

Out[13]:

	aspiration	enginelocation	carwidth	curbweight	enginetype	cylindernumber	stroke	peakrp
0	std	front	64.1	2548	dohc	four	2.68	500
1	std	front	64.1	2548	dohc	four	2.68	500

ℹ️ The description of the dataset is available here. Make sure to refer to it throughout the exercise.

(1) Duplicates

❓ Remove the duplicates from the dataset if there are any. ❓

Overwite the dataframe `df`

In [14]:
```
df= df.drop_duplicates().reset_index(drop = True)
```

(2) Missing values

❓ Find the missing values and impute them either with `strategy = "most frequent"` (categorical variables) or `strategy = "median"` (numerical variables) ❓

In [15]:
```
round(df.isnull().sum().sort_values(ascending=False)/len(df),3)
```

Out[15]:
```
enginelocation    0.052
carwidth          0.010
aspiration        0.000
curbweight        0.000
enginetype        0.000
cylindernumber    0.000
stroke            0.000
peakrpm           0.000
price             0.000
dtype: float64
```

`carwidth`

▶ 💡 *Hint*

In [16]:
```python
df.carwidth.value_counts(dropna=False)
```

Out[16]:
```
66.5    22
63.8    19
65.4    15
63.6     9
68.4     9
64       9
64.4     9
65.5     8
65.2     7
65.6     6
64.2     6
66.3     6
67.2     6
66.9     5
67.9     5
*        4
68.9     4
71.7     3
70.3     3
65.7     3
63.9     3
64.8     3
65       2
67.7     2
68.3     2
71.4     2
NaN      2
66.6     1
63.4     1
72.3     1
64.1     1
68       1
72       1
70.5     1
66.1     1
70.6     1
69.6     1
61.8     1
66       1
64.6     1
60.3     1
70.9     1
66.4     1
68.8     1
Name: carwidth, dtype: int64
```

In [17]:
```python
import numpy as np
from sklearn.impute import SimpleImputer

df = df.replace("*", np.nan) # Replace occurences of "*" by np.nan

carwidth_imputer = SimpleImputer(strategy="median") # Instanciate median imputer
carwidth_imputer.fit(df[['carwidth']]) # Fit imputer to carwidth column
df['carwidth'] = carwidth_imputer.transform(df[['carwidth']]) # Impute

df.head()
```

Out[17]:

	aspiration	enginelocation	carwidth	curbweight	enginetype	cylindernumber	stroke	peakrpn
0	std	front	64.1	2548	dohc	four	2.68	500
1	std	front	65.5	2823	ohcv	six	3.47	500
2	std	front	65.5	2337	ohc	four	3.40	550
3	std	front	66.4	2824	ohc	five	3.40	550
4	std	front	66.3	2507	ohc	five	3.40	550

enginelocation

▶ 💡 *Hint*

In [18]:
```python
print(df.enginelocation.unique())
print(df.enginelocation.value_counts(dropna=False))
```

```
['front' nan 'rear']
front    179
NaN       10
rear       2
Name: enginelocation, dtype: int64
```

In [20]:
```python
engine_imputer = SimpleImputer(strategy="most_frequent")  # Instantiate most freque
engine_imputer.fit(df[['enginelocation']]) # Fit imputer to enginelocation column
df['enginelocation'] = engine_imputer.transform(df[['enginelocation']]) # Impute

df.head()
```

Out[20]:

	aspiration	enginelocation	carwidth	curbweight	enginetype	cylindernumber	stroke	peakrpn
0	std	front	64.1	2548	dohc	four	2.68	500
1	std	front	65.5	2823	ohcv	six	3.47	500
2	std	front	65.5	2337	ohc	four	3.40	550
3	std	front	66.4	2824	ohc	five	3.40	550
4	std	front	66.3	2507	ohc	five	3.40	550

🧪 **Test your code**

In [21]:
```python
from nbresult import ChallengeResult

result = ChallengeResult('missing_values',
```

```
                                 dataset = df)
result.write()
print(result.check())
```

```
============================= test session starts ==============================
platform linux -- Python 3.10.6, pytest-7.1.3, pluggy-1.0.0 -- /home/joharlewago
n/.pyenv/versions/lewagon/bin/python3
cachedir: .pytest_cache
rootdir: /home/joharlewagon/code/UKVeteran/05-ML/02-Prepare-the-dataset/data-car-p
rices/tests
plugins: anyio-3.6.2, asyncio-0.19.0, typeguard-2.13.3
asyncio: mode=strict
collecting ... collected 2 items

test_missing_values.py::TestMissing_values::test_carwidth PASSED         [ 50%]
test_missing_values.py::TestMissing_values::test_engine_location PASSED  [100%]

============================== 2 passed in 0.49s ===============================


💯 You can commit your code:

git add tests/missing_values.pickle

git commit -m 'Completed missing_values step'

git push origin master
```

(3) Scaling the numerical features

In [22]:
```
# As a reminder, some information about the dataframe
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 191 entries, 0 to 190
Data columns (total 9 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   aspiration      191 non-null    object 
 1   enginelocation  191 non-null    object 
 2   carwidth        191 non-null    float64
 3   curbweight      191 non-null    int64  
 4   enginetype      191 non-null    object 
 5   cylindernumber  191 non-null    object 
 6   stroke          191 non-null    float64
 7   peakrpm         191 non-null    int64  
 8   price           191 non-null    object 
dtypes: float64(2), int64(2), object(5)
memory usage: 13.6+ KB
```

In [23]:
```
# And here are the numerical features of the dataset we need to scale
numerical_features = df.select_dtypes(exclude=['object']).columns
numerical_features
```

Out[23]:
```
Index(['carwidth', 'curbweight', 'stroke', 'peakrpm'], dtype='object')
```

❓ Question: Scaling the numerical features ❓

Investigate the numerical features for outliers and distribution, and apply the solutions below accordingly:

- Robust Scaler
- Standard Scaler

Replace the original columns with the transformed values.

`peakrpm` , `carwidth` , & `stroke`

▶ 💡 *Hint*

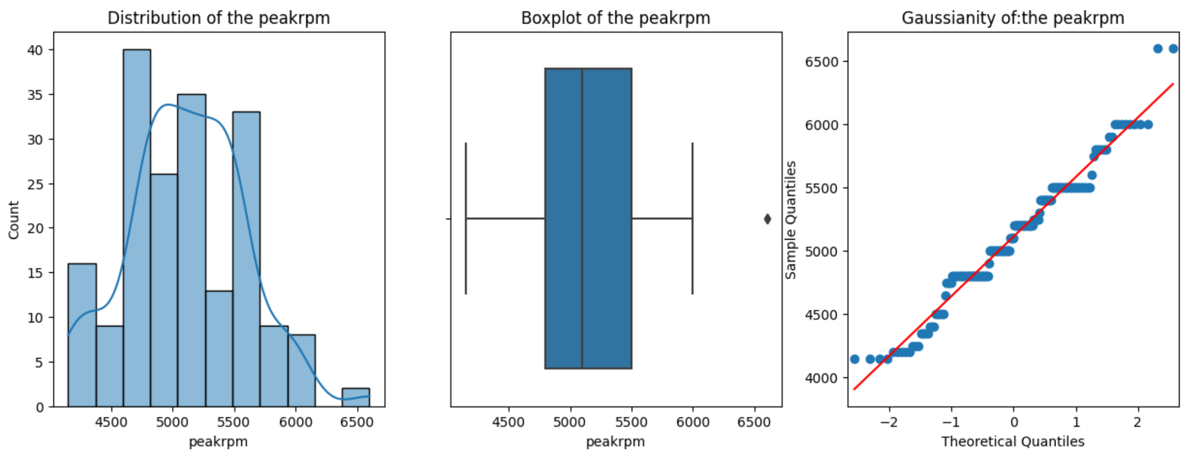
In [25]:
```
variable = 'peakrpm'

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

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

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

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



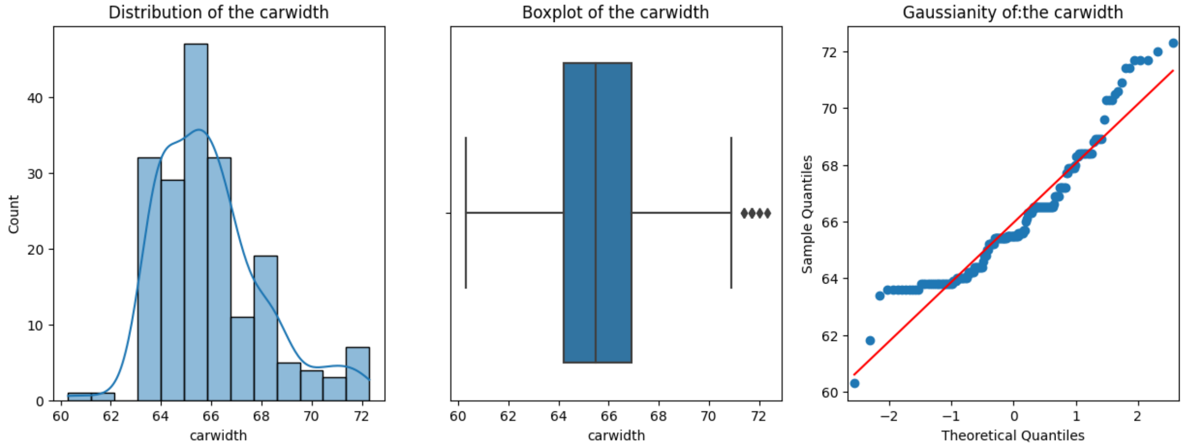
In [26]:
```
variable = 'carwidth'

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

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

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

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



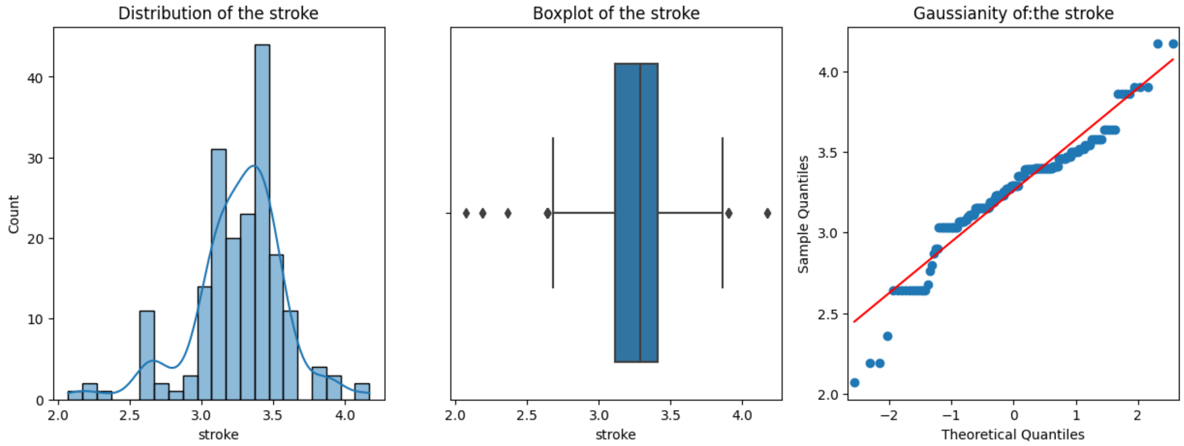
In [27]:
```
variable = 'stroke'

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

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

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

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



In [28]:
```
from sklearn.preprocessing import RobustScaler

rb_scaler = RobustScaler()
df['peakrpm'],df['carwidth'],df['stroke'] = rb_scaler.fit_transform(df[['peakrpm',
df.head()
```

Out[28]:

	aspiration	enginelocation	carwidth	curbweight	enginetype	cylindernumber	stroke	pea
0	std	front	-0.518519	2548	dohc	four	-2.033333	-0.14
1	std	front	0.000000	2823	ohcv	six	0.600000	-0.14
2	std	front	0.000000	2337	ohc	four	0.366667	0.57
3	std	front	0.333333	2824	ohc	five	0.366667	0.57
4	std	front	0.296296	2507	ohc	five	0.366667	0.57

`curbweight`

▶ 💡 *Hint*

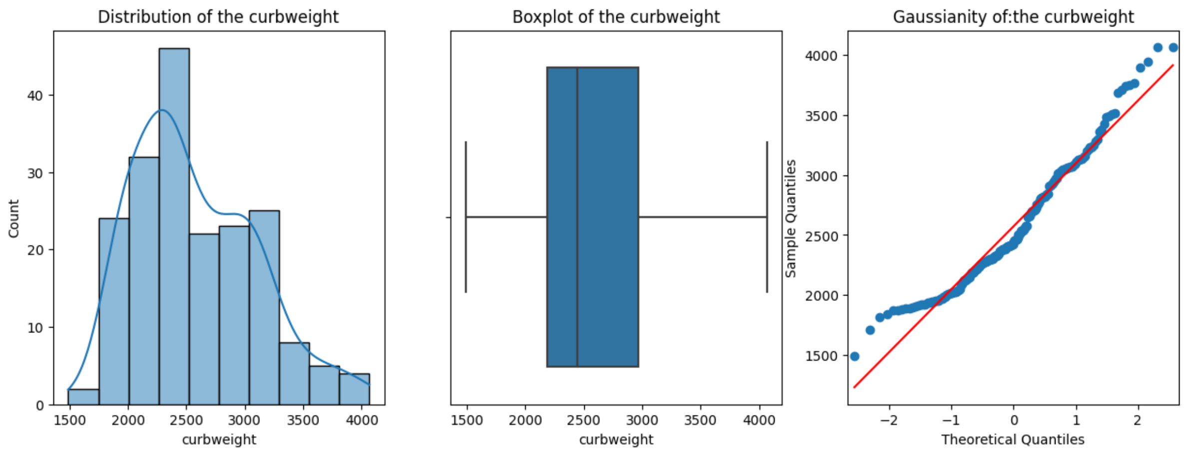
In [29]:
```
variable = 'curbweight'

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

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

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

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



In [30]:
```
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()
df['curbweight'] = std_scaler.fit_transform(df[['curbweight']])
df.head()
```

Out[30]:

	aspiration	enginelocation	carwidth	curbweight	enginetype	cylindernumber	stroke	pea
0	std	front	-0.518519	-0.048068	dohc	four	-2.033333	-0.14
1	std	front	0.000000	0.476395	ohcv	six	0.600000	-0.14
2	std	front	0.000000	-0.450474	ohc	four	0.366667	0.57
3	std	front	0.333333	0.478302	ohc	five	0.366667	0.57
4	std	front	0.296296	-0.126260	ohc	five	0.366667	0.57

🧪 **Test your code**

In [ ]:
```
from nbresult import ChallengeResult

result = ChallengeResult('scaling',
                         dataset = df
)

result.write()
print(result.check())
```

(4) Encoding the categorical features

❓ Question: encoding the categorical variables ❓

👇 Investigate the features that require encoding, and apply the following techniques accordingly:

- One-hot encoding
- Manual ordinal encoding

In the Dataframe, replace the original features with their encoded version(s).

In [32]:
```
print(f"The unique values of `aspiration` are {df.aspiration.unique()}") # Check u

print(f"The unique values of `enginelocation` are {df.enginelocation.unique()}") #
```

```
The unique values of `aspiration` are ['std' 'turbo']
The unique values of `enginelocation` are ['front' 'rear']
```

`aspiration` & `enginelocation`

▶ 💡 *Hint*

In [33]:
```
from sklearn.preprocessing import OneHotEncoder

binary_encoder = OneHotEncoder(sparse=False, drop='if_binary')
df['aspiration'], df['enginelocation'] = binary_encoder.fit_transform(df[['aspirat

df.head()
```

Out[33]:

	aspiration	enginelocation	carwidth	curbweight	enginetype	cylindernumber	stroke	pea
0	0.0	0.0	-0.518519	-0.048068	dohc	four	-2.033333	-0.14
1	0.0	0.0	0.000000	0.476395	ohcv	six	0.600000	-0.14
2	0.0	0.0	0.000000	-0.450474	ohc	four	0.366667	0.57
3	0.0	0.0	0.333333	0.478302	ohc	five	0.366667	0.57
4	0.0	0.0	0.296296	-0.126260	ohc	five	0.366667	0.57

`enginetype`

▶ 💡 *Hint*

In [34]:
```
df.enginetype.unique()
```

Out[34]:
```
array(['dohc', 'ohcv', 'ohc', 'l', 'rotor', 'ohcf', 'dohcv'], dtype=object)
```

In [35]:
```
df.shape
```

Out[35]:
```
(191, 9)
```

In [36]:
```
from sklearn.preprocessing import OneHotEncoder

# Instantiate a OneHotEncoder for the categorical feature EngineType
ohe = OneHotEncoder(sparse=False)

# Fitting it
ohe.fit(df[['enginetype']])

# Showing the categories detected by the encoder
display(ohe.categories_)

# Since Sklearn 1.1, we can retrieve the names of the generated columns
display(ohe.get_feature_names_out())

# Let's encode EngineType
enginetype_encoded = ohe.transform(df[['enginetype']])

# Now we store the encoded values in the dataframe
df[ohe.get_feature_names_out()] = enginetype_encoded

# We can get rid of the original column EngineType now
df.drop(columns='enginetype', inplace = True)

# And show df
df
```

```
[array(['dohc', 'dohcv', 'l', 'ohc', 'ohcf', 'ohcv', 'rotor'], dtype=object)]
array(['enginetype_dohc', 'enginetype_dohcv', 'enginetype_l',
       'enginetype_ohc', 'enginetype_ohcf', 'enginetype_ohcv',
       'enginetype_rotor'], dtype=object)
```

Out[36]:

	aspiration	enginelocation	carwidth	curbweight	cylindernumber	stroke	peakrpm	
0	0.0	0.0	-0.518519	-0.048068	four	-2.033333	-0.142857	expe
1	0.0	0.0	0.000000	0.476395	six	0.600000	-0.142857	expe
2	0.0	0.0	0.000000	-0.450474	four	0.366667	0.571429	expe
3	0.0	0.0	0.333333	0.478302	five	0.366667	0.571429	expe
4	0.0	0.0	0.296296	-0.126260	five	0.366667	0.571429	expe
...	...	...	...	...	...	...	...	
186	0.0	0.0	1.259259	0.722416	four	-0.466667	0.428571	expe
187	1.0	0.0	1.222222	0.907408	four	-0.466667	0.285714	expe
188	0.0	0.0	1.259259	0.836844	six	-1.400000	0.571429	expe
189	1.0	0.0	1.259259	1.227807	six	0.366667	-0.428571	expe
190	1.0	0.0	1.259259	0.932201	four	-0.466667	0.428571	expe

191 rows × 15 columns

`cylindernumber`

▶ 💡 Hint

In [37]:
```
df.cylindernumber.unique()
```

Out[37]:
```
array(['four', 'six', 'five', 'three', 'twelve', 'two', 'eight'],
      dtype=object)
```

In [38]:
```
df['cylindernumber'] = df['cylindernumber'].map({'four': 4,
                                                  'six': 6,
                                                  "five":5,
                                                  'three': 3,
                                                  'twelve':12,
                                                  'two':2,
                                                  'eight':8})
df.head()
```

Out[38]:

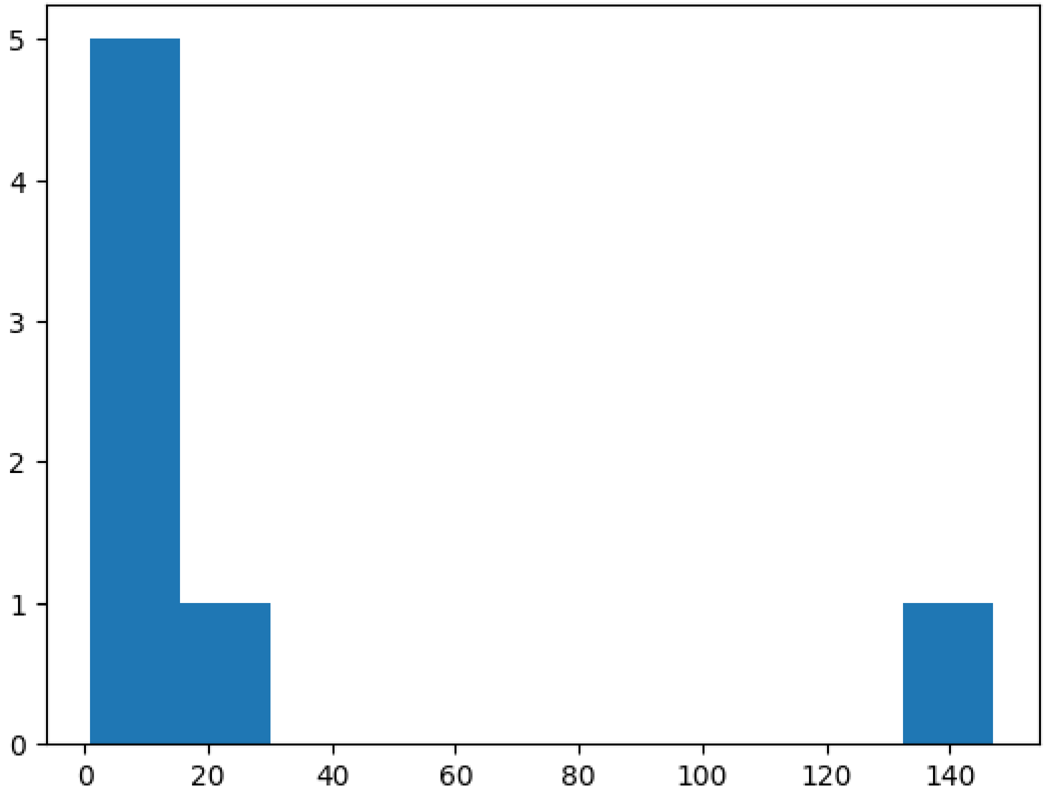
	aspiration	enginelocation	carwidth	curbweight	cylindernumber	stroke	peakrpm	pr
0	0.0	0.0	-0.518519	-0.048068	4	-2.033333	-0.142857	expens
1	0.0	0.0	0.000000	0.476395	6	0.600000	-0.142857	expens
2	0.0	0.0	0.000000	-0.450474	4	0.366667	0.571429	expens
3	0.0	0.0	0.333333	0.478302	5	0.366667	0.571429	expens
4	0.0	0.0	0.296296	-0.126260	5	0.366667	0.571429	expens

❓ Now that you've made `cylindernumber` into a numeric feature between 2 and 12, you need to scale it ❓

▶ 💡 Hint

In [40]:
```
df['cylindernumber'].value_counts()
import matplotlib.pyplot as plt
plt.hist(df['cylindernumber'].value_counts())
```

Out[40]:
```
(array([5., 1., 0., 0., 0., 0., 0., 0., 0., 1.]),
 array([  1. ,  15.6,  30.2,  44.8,  59.4,  74. ,  88.6, 103.2, 117.8,
        132.4, 147. ]),
 <BarContainer object of 10 artists>)
```



In [42]:
```
from sklearn.preprocessing import MinMaxScaler, RobustScaler

mm_scaler = MinMaxScaler()
mm = pd.DataFrame(mm_scaler.fit_transform(df[['cylindernumber']]))

rb_scaler = RobustScaler()
rb = pd.DataFrame(rb_scaler.fit_transform(df[['cylindernumber']]))
```

In [43]:
```
mm.value_counts()
```

Out[43]:
```
0.2    147
0.4     23
0.3     11
0.6      5
0.0      3
0.1      1
1.0      1
dtype: int64
```

In [44]:
```
rb.value_counts()
```

Out[44]:
```
 0.0    147
 2.0     23
 1.0     11
 4.0      5
-2.0      3
-1.0      1
 8.0      1
dtype: int64
```

In [45]:
```
from sklearn.preprocessing import RobustScaler

rb_scaler = RobustScaler()

df['cylindernumber'] = rb_scaler.fit_transform(df[['cylindernumber']])

df.head()
```

Out[45]:

	aspiration	enginelocation	carwidth	curbweight	cylindernumber	stroke	peakrpm	pr
0	0.0	0.0	-0.518519	-0.048068	0.0	-2.033333	-0.142857	expens
1	0.0	0.0	0.000000	0.476395	2.0	0.600000	-0.142857	expens
2	0.0	0.0	0.000000	-0.450474	0.0	0.366667	0.571429	expens
3	0.0	0.0	0.333333	0.478302	1.0	0.366667	0.571429	expens
4	0.0	0.0	0.296296	-0.126260	1.0	0.366667	0.571429	expens

▶ *Here is a screenshot of how your dataframe shoud look like after scaling and encoding*

`price`

👇 Encode the target `price`.

▶ 💡 Hint

In [46]:
```
from sklearn.preprocessing import LabelEncoder

df['price'] = LabelEncoder().fit_transform(df['price'])
df.head()
```

Out[46]:

	aspiration	enginelocation	carwidth	curbweight	cylindernumber	stroke	peakrpm	price
0	0.0	0.0	-0.518519	-0.048068	0.0	-2.033333	-0.142857	1
1	0.0	0.0	0.000000	0.476395	2.0	0.600000	-0.142857	1
2	0.0	0.0	0.000000	-0.450474	0.0	0.366667	0.571429	1
3	0.0	0.0	0.333333	0.478302	1.0	0.366667	0.571429	1
4	0.0	0.0	0.296296	-0.126260	1.0	0.366667	0.571429	1

🧪 **Test your code**

In [47]:
```
from nbresult import ChallengeResult

result = ChallengeResult('encoding',
                         dataset = df)
result.write()
print(result.check())
```

```
============================= test session starts ==============================
platform linux -- Python 3.10.6, pytest-7.1.3, pluggy-1.0.0 -- /home/joharlewago
n/.pyenv/versions/lewagon/bin/python3
cachedir: .pytest_cache
rootdir: /home/joharlewagon/code/UKVeteran/05-ML/02-Prepare-the-dataset/data-car-p
rices/tests
plugins: anyio-3.6.2, asyncio-0.19.0, typeguard-2.13.3
asyncio: mode=strict
collecting ... collected 4 items

test_encoding.py::TestEncoding::test_aspiration PASSED                   [ 25%]
test_encoding.py::TestEncoding::test_enginelocation PASSED               [ 50%]
test_encoding.py::TestEncoding::test_enginetype PASSED                   [ 75%]
test_encoding.py::TestEncoding::test_price PASSED                        [100%]

============================== 4 passed in 1.55s ===============================
```

💯 You can commit your code:

```
git add tests/encoding.pickle

git commit -m 'Completed encoding step'

git push origin master
```

(5) Base Modelling

👏 The dataset has been preprocessed and is now ready to be fitted to a model.

❓ Question: a first attempt to evaluate a classification model ❓

Cross-validate a `LogisticRegression` on this preprocessed dataset and save its score under a variable named `base_model_score`.

In [48]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

X = df.drop(columns=['price'])
y = df['price']

model = LogisticRegression()

scores = cross_val_score(model, X, y, cv=10)
base_model_score = scores.mean()

base_model_score
```

Out[48]:
```
0.8797368421052632
```

🧪 Test your code

In [49]:
```
from nbresult import ChallengeResult

result = ChallengeResult('base_model',
                         score = base_model_score
)
```

```
result.write()
print(result.check())
```

```
============================= test session starts ==============================
platform linux -- Python 3.10.6, pytest-7.1.3, pluggy-1.0.0 -- /home/joharlewago
n/.pyenv/versions/lewagon/bin/python3
cachedir: .pytest_cache
rootdir: /home/joharlewagon/code/UKVeteran/05-ML/02-Prepare-the-dataset/data-car-p
rices/tests
plugins: anyio-3.6.2, asyncio-0.19.0, typeguard-2.13.3
asyncio: mode=strict
collecting ... collected 1 item

test_base_model.py::TestBase_model::test_base_model_score PASSED         [100%]

============================== 1 passed in 0.35s ===============================


💯 You can commit your code:

git add tests/base_model.pickle

git commit -m 'Completed base_model step'

git push origin master
```

(6) Feature Selection (with *Permutation Importance*)

👩🏻‍🏫 A powerful way to detect whether a feature is relevant or not to predict a target is to:

1. Run a model and score it
2. Shuffle this feature, re-run the model and score it
   - If the performance significantly dropped, the feature is important and you shoudn't have dropped it
   - If the performance didn't decrease a lot, the feature may be discarded.

❓ Questions ❓

1. Perform a feature permutation to detect which features bring the least amount of information to the model.
2. Remove the weak features from your dataset until you notice model performance dropping substantially
3. Using your new set of strong features, cross-validate a new model, and save its score under variable name `strong_model_score`.

In [50]:
```
import numpy as np
from sklearn.model_selection import cross_validate
from sklearn.inspection import permutation_importance

# Evaluate your model without feature permutation
model = LogisticRegression()
cv_results = cross_validate(model, X, y, cv = 5)
score = cv_results["test_score"].mean()
print(f"Before any feature permutation, the cross-validated accuracy is equal to {

## Question 1 - Permutation importance
model = LogisticRegression().fit(X,y) # Fit the model
```

```
permutation_score = permutation_importance(model, X, y, n_repeats=100) # Perform Pe
importance_df = pd.DataFrame(np.vstack((X.columns,
                                        permutation_score.importances_mean)).T, # U
                             columns = ['feature','feature_importance'])

print("After feature permutation, here are the decreases in terms of scores:")
importance_df = importance_df.sort_values(by="feature_importance", ascending = Fals
importance_df
```

```
Before any feature permutation, the cross-validated accuracy is equal to 0.84
After feature permutation, here are the decreases in terms of scores:
```

Out[50]:

	feature	feature_importance
3	curbweight	0.289267
2	carwidth	0.106073
5	stroke	0.029895
11	enginetype_ohcf	0.018429
6	peakrpm	0.015497
10	enginetype_ohc	0.015445
13	enginetype_rotor	0.011885
0	aspiration	0.008272
4	cylindernumber	0.008168
7	enginetype_dohc	0.004974
12	enginetype_ohcv	0.000419
1	enginelocation	0.000314
8	enginetype_dohcv	0.0
9	enginetype_l	0.0

In [51]:

```
## Question 2 - remove weak features

# I want to get rid of features which caused less than this  in terms of performanc
threshold = 0.05

# Decompose this one-liner piece of code step by step if you don't understand it at
weak_features = importance_df[importance_df.feature_importance <= threshold]["featu
weak_features
```

Out[51]:

```
array(['stroke', 'enginetype_ohcf', 'peakrpm', 'enginetype_ohc',
       'enginetype_rotor', 'aspiration', 'cylindernumber',
       'enginetype_dohc', 'enginetype_ohcv', 'enginelocation',
       'enginetype_dohcv', 'enginetype_l'], dtype=object)
```

In [52]:

```
## Question 3 - Cross validating the model with strong features only
X_strong_features = df.drop(columns=list(weak_features) + ["price"])

print(f"Our strong features are {list(X_strong_features.columns)}")

model = LogisticRegression()

scores = cross_val_score(model, X_strong_features, y, cv = 5)
strong_model_score = scores.mean()

print(f"Before removing weak features, the cross-validated accuracy was equal to {r
```

```
print(f"The LogisticRegression fitted with the strong features only has a score of

#### NOTE - The score may even be better because
### some features were bringing nothing else than noise to the model
```

```
Our strong features are ['carwidth', 'curbweight']
Before removing weak features, the cross-validated accuracy was equal to 0.84
The LogisticRegression fitted with the strong features only has a score of 0.91
```

🧪 **Test your code**

In [53]:
```
from nbresult import ChallengeResult

result = ChallengeResult('strong_model',
                         score = strong_model_score
)

result.write()
print(result.check())
```

```
============================= test session starts ==============================
platform linux -- Python 3.10.6, pytest-7.1.3, pluggy-1.0.0 -- /home/joharlewago
n/.pyenv/versions/lewagon/bin/python3
cachedir: .pytest_cache
rootdir: /home/joharlewagon/code/UKVeteran/05-ML/02-Prepare-the-dataset/data-car-p
rices/tests
plugins: anyio-3.6.2, asyncio-0.19.0, typeguard-2.13.3
asyncio: mode=strict
collecting ... collected 1 item

test_strong_model.py::TestStrong_model::test_strong_model_score PASSED   [100%]

============================== 1 passed in 0.49s ===============================


💯 You can commit your code:

git add tests/strong_model.pickle

git commit -m 'Completed strong_model step'

git push origin master
```

Bonus - Stratifying your data ⚖️

💡 As we split our data into training and testing, we need to be mindful of the proportion of categorical variables in our dataset - whether it's the classes of our target `y` or a categorical feature in `X`.

Let's have a look at an example 👇

❓ Split your original `X` and `y` into training and testing data, using sklearn's `train_test_split`; use `random_state=1` and `test_size=0.3` to have comparable results.

In [54]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_st
```

❓ Check the proportion of `price` class `1` cars in your training dataset and testing dataset.

> *If you check the proportion of them in the raw `df`, it should be very close to 50/50*

In [55]:
```
print('Training data share of class 1 cars:', y_train.mean())
print('Testing data share of class 1 cars:', y_test.mean())
```

```
Training data share of class 1 cars: 0.5037593984962406
Testing data share of class 1 cars: 0.5172413793103449
```

It should still be pretty close to 50/50 ☝️

But what if we change the random state?

❓ Loop through random states 1 through 10, each time calculating the share of `price` class `1` cars in the training and testing data. ❓

In [56]:
```
for i in range(1, 11):

    print("-"*50)
    print("##### Random state set =", i)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, randon
    print('Training data share of class 1 cars:', round(y_train.mean(), 3))
    print('Testing data share of class 1 cars:', round(y_test.mean(), 3))
```

```
--------------------------------------------------
##### Random state set = 1
Training data share of class 1 cars: 0.504
Testing data share of class 1 cars: 0.517
--------------------------------------------------
##### Random state set = 2
Training data share of class 1 cars: 0.481
Testing data share of class 1 cars: 0.569
--------------------------------------------------
##### Random state set = 3
Training data share of class 1 cars: 0.504
Testing data share of class 1 cars: 0.517
--------------------------------------------------
##### Random state set = 4
Training data share of class 1 cars: 0.534
Testing data share of class 1 cars: 0.448
--------------------------------------------------
##### Random state set = 5
Training data share of class 1 cars: 0.534
Testing data share of class 1 cars: 0.448
--------------------------------------------------
##### Random state set = 6
Training data share of class 1 cars: 0.496
Testing data share of class 1 cars: 0.534
--------------------------------------------------
##### Random state set = 7
Training data share of class 1 cars: 0.534
Testing data share of class 1 cars: 0.448
--------------------------------------------------
##### Random state set = 8
Training data share of class 1 cars: 0.489
Testing data share of class 1 cars: 0.552
--------------------------------------------------
##### Random state set = 9
Training data share of class 1 cars: 0.579
Testing data share of class 1 cars: 0.345
--------------------------------------------------
##### Random state set = 10
Training data share of class 1 cars: 0.489
Testing data share of class 1 cars: 0.552
```

You will observe that the proportion changes every time, sometimes even quite drastically 😱! This can affect model performance!

❓ Compare the test score of a logistic regression when trained using `train_test_split(random_state=1)` *vs.* `random_state=9` ❓

Remember to fit on training data and score on testing data.

In [57]:
```
model_1 = LogisticRegression()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_st

model_1.fit(X_train, y_train)

model_1.score(X_test, y_test)
```

Out[57]: 0.9310344827586207

In [58]:
```
model_9 = LogisticRegression()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_st
```

```python
model_9.fit(X_train, y_train)

model_9.score(X_test, y_test)
```

Out[58]:
```
0.7931034482758621
```

👀 You should see a much lower score with `random_state=9` because the proportion of class `1` cars in that test set is 34.5%, quite far from the 57.9% in the training set or even the 50% in the original dataset.

This is substantial, as this accidental imbalance in our dataset can not only make model performance worse, but also distort the "reality" during training or scoring 😳

So how do we fix this issue? How do we keep the same distribution of classes across the train set and the test set? 🔧

🎁 Luckily, this is taken care of by `cross_validate` in sklearn, when the estimator (a.k.a the model) is a classifier and the target is a class. Check out the documentation of the `cv` parameter in 📚 **sklearn.model_selection.cross_validate**.

The answer is to use the following:

> 📚 **Stratification**

Stratification of the target

💡 We can also use the ***strafification*** technique in a `train_test_split`.

❓ Run through the same 1 to 10 random state loop again, but this time also ***pass `stratify=y` into the holdout method***. ❓

In [59]:
```python
for i in range(1, 11):
    print("-"*50)
    print("##### Random state set =", i)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, randor
    print('Training data share of class 1 cars:', round(y_train.mean(), 3))
    print('Testing data share of class 1 cars:', round(y_test.mean(), 3))
```

```
--------------------------------------------------
##### Random state set = 1
Training data share of class 1 cars: 0.511
Testing data share of class 1 cars: 0.5
--------------------------------------------------
##### Random state set = 2
Training data share of class 1 cars: 0.511
Testing data share of class 1 cars: 0.5
--------------------------------------------------
##### Random state set = 3
Training data share of class 1 cars: 0.511
Testing data share of class 1 cars: 0.5
--------------------------------------------------
##### Random state set = 4
Training data share of class 1 cars: 0.511
Testing data share of class 1 cars: 0.5
--------------------------------------------------
##### Random state set = 5
Training data share of class 1 cars: 0.511
Testing data share of class 1 cars: 0.5
--------------------------------------------------
##### Random state set = 6
Training data share of class 1 cars: 0.511
Testing data share of class 1 cars: 0.5
--------------------------------------------------
##### Random state set = 7
Training data share of class 1 cars: 0.511
Testing data share of class 1 cars: 0.5
--------------------------------------------------
##### Random state set = 8
Training data share of class 1 cars: 0.511
Testing data share of class 1 cars: 0.5
--------------------------------------------------
##### Random state set = 9
Training data share of class 1 cars: 0.511
Testing data share of class 1 cars: 0.5
--------------------------------------------------
##### Random state set = 10
Training data share of class 1 cars: 0.511
Testing data share of class 1 cars: 0.5
```

👀 Even if the random state is changing, the proportion of classes inside the training and testing data is kept the same as in the original `y`. This is what *stratification* is.

Using `train_test_split` with the `stratify` parameter, we can also preserve proportions of a feature across training and testing data. This can be extremely important, for example:

- preserving proportion of male and female customers in predicting churn 🙋‍♂️ 🙋‍♀️
- preserving the proportion big and small houses in predicting their prices 🏠 🏰
- preserving distribution of 1-5 review scores (multiclass!) in recommending the next product 🛍️
- etc...

For instance, in our dataset, to holdout the same share of `aspiration` feature in both training and testing data, we could simply write `train_test_split(X, y, test_size=0.3, stratify=X.aspiration)`

As we saw, `cross_validate` **can automatically stratify the target, but not the features...** 🤔 We need a bit of extra work for that.

We need `StratifiedKFold` 🔬

Stratification - generalized

📚 **StratifiedKFold** allows us to split the data into `K` splits, while stratifying on certain columns (features or target).

This way, we can do a manual cross-validation while keeping proportions on the categorical features of interest - let's try it with the binary `aspiration` feature:

In [60]:
```
from sklearn.model_selection import StratifiedKFold

# initializing a stratified k-fold that will split the data into 5 folds
skf = StratifiedKFold(n_splits=5)
scores = []

# .split() method creates an iterator; 'X.aspiration' is the feature that we strat
for train_indices, test_indices in skf.split(X, X.aspiration):

    # 'train_indices' and 'test_indices' are lists of indices that produce proport
    X_train, X_test = X.iloc[train_indices], X.iloc[test_indices]
    y_train, y_test = y.iloc[train_indices], y.iloc[test_indices]

    # initialize and fit a model
    model = LogisticRegression()
    model.fit(X_train, y_train)

    # append a score to get an average of 5 folds in the end
    scores.append(model.score(X_test, y_test))

np.array(scores).mean()
```

Out[60]:
```
0.8585695006747638
```

📖 Some sklearn reads on **stratification**:

- Visualization of how different holdout methods in sklearn work
- Overall cross-validation and stratification understanding

🏁 Congratulations! You have prepared a whole dataset, ran feature selection and even learned about stratification 💪

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

🚀 ... and move on to the next challenge!