Price_Prediction_Python

September 7, 2024

[1]:
```python
# This script downloads a dataset from a given URL and saves it as a CSV file

import requests

# URL of the CSV file
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/autos/
 ↪imports-85.data'

# Download the file
response = requests.get(url)

# Save the downloaded file as 'auto.csv'
with open('auto.csv', 'wb') as file:
    file.write(response.content)
```

[2]:
```python
# This script loads and displays the dataset

import pandas as pd
import numpy as np

# Load the dataset from the CSV file
df = pd.read_csv('auto.csv', header=None)
# Display the first few rows of the dataset
df.head()
```

[2]:
```
   0    1            2    3    4     5            6    7      8     9  …  \
0  3    ?  alfa-romero  gas  std   two  convertible  rwd  front  88.6  …
1  3    ?  alfa-romero  gas  std   two  convertible  rwd  front  88.6  …
2  1    ?  alfa-romero  gas  std   two    hatchback  rwd  front  94.5  …
3  2  164         audi  gas  std  four        sedan  fwd  front  99.8  …
4  2  164         audi  gas  std  four        sedan  4wd  front  99.4  …

    16    17    18    19    20   21    22  23  24     25
0  130  mpfi  3.47  2.68   9.0  111  5000  21  27  13495
1  130  mpfi  3.47  2.68   9.0  111  5000  21  27  16500
2  152  mpfi  2.68  3.47   9.0  154  5000  19  26  16500
3  109  mpfi  3.19  3.40  10.0  102  5500  24  30  13950
4  136  mpfi  3.19  3.40   8.0  115  5500  18  22  17450
```

```
[5 rows x 26 columns]
```

[3]:
```
# Display the last few rows of the dataset
df.tail()
```

[3]:
```
      0   1      2       3      4     5      6    7      8      9  …   16  \
200  -1  95  volvo     gas    std  four  sedan  rwd  front  109.1  …  141
201  -1  95  volvo     gas  turbo  four  sedan  rwd  front  109.1  …  141
202  -1  95  volvo     gas    std  four  sedan  rwd  front  109.1  …  173
203  -1  95  volvo  diesel  turbo  four  sedan  rwd  front  109.1  …  145
204  -1  95  volvo     gas  turbo  four  sedan  rwd  front  109.1  …  141

       17    18    19    20   21    22  23  24     25
200  mpfi  3.78  3.15   9.5  114  5400  23  28  16845
201  mpfi  3.78  3.15   8.7  160  5300  19  25  19045
202  mpfi  3.58  2.87   8.8  134  5500  18  23  21485
203   idi  3.01  3.40  23.0  106  4800  26  27  22470
204  mpfi  3.78  3.15   9.5  114  5400  19  25  22625

[5 rows x 26 columns]
```

[4]:
```
# This script adds descriptive headers to the dataset that initially lacks␣
↪headers.
# Create a list of headers based on the information provided
headers = ["symboling", "normalized-losses", "make", "fuel-type", "aspiration",
           "num-of-doors", "body-style", "drive-wheels", "engine-location",
           "wheel-base", "length", "width", "height", "curb-weight",
           "engine-type", "num-of-cylinders", "engine-size", "fuel-system",
           "bore", "stroke", "compression-ratio", "horsepower", "peak-rpm",
           "city-mpg", "highway-mpg", "price"]

# Replace the existing headers (integers) with the descriptive headers we␣
↪created
df.columns = headers
df.head()
```

[4]:
```
   symboling normalized-losses         make fuel-type aspiration num-of-doors  \
0          3                 ?  alfa-romero       gas        std          two
1          3                 ?  alfa-romero       gas        std          two
2          1                 ?  alfa-romero       gas        std          two
3          2               164         audi       gas        std         four
4          2               164         audi       gas        std         four

    body-style drive-wheels engine-location  wheel-base  …  engine-size  \
0  convertible          rwd           front        88.6  …          130
1  convertible          rwd           front        88.6  …          130
```

```
2    hatchback          rwd            front        94.5  …          152
3        sedan          fwd            front        99.8  …          109
4        sedan          4wd            front        99.4  …          136

  fuel-system  bore  stroke compression-ratio horsepower  peak-rpm city-mpg  \
0        mpfi  3.47    2.68               9.0        111      5000       21
1        mpfi  3.47    2.68               9.0        111      5000       21
2        mpfi  2.68    3.47               9.0        154      5000       19
3        mpfi  3.19    3.40              10.0        102      5500       24
4        mpfi  3.19    3.40               8.0        115      5500       18

  highway-mpg  price
0          27  13495
1          27  16500
2          26  16500
3          30  13950
4          22  17450

[5 rows x 26 columns]
```

[5]:
```
# Replace "?" with NaN to handle missing values
df.replace("?", np.nan, inplace=True)
df.head()
```

[5]:
```
  symboling normalized-losses         make fuel-type aspiration num-of-doors  \
0         3               NaN  alfa-romero       gas        std          two
1         3               NaN  alfa-romero       gas        std          two
2         1               NaN  alfa-romero       gas        std          two
3         2               164         audi       gas        std         four
4         2               164         audi       gas        std         four

   body-style drive-wheels engine-location  wheel-base  …  engine-size  \
0 convertible          rwd            front        88.6  …          130
1 convertible          rwd            front        88.6  …          130
2    hatchback          rwd            front        94.5  …          152
3        sedan          fwd            front        99.8  …          109
4        sedan          4wd            front        99.4  …          136

  fuel-system  bore  stroke compression-ratio horsepower  peak-rpm city-mpg  \
0        mpfi  3.47    2.68               9.0        111      5000       21
1        mpfi  3.47    2.68               9.0        111      5000       21
2        mpfi  2.68    3.47               9.0        154      5000       19
3        mpfi  3.19    3.40              10.0        102      5500       24
4        mpfi  3.19    3.40               8.0        115      5500       18

  highway-mpg  price
0          27  13495
```

```
1          27  16500
2          26  16500
3          30  13950
4          22  17450

[5 rows x 26 columns]
```

[6]:
```
# Save the DataFrame to a CSV file
df.to_csv('automobile.csv', index=False)
```

[7]:
```
# Display the data types of each column in the DataFrame
df.dtypes
```

[7]:
```
symboling              int64
normalized-losses     object
make                  object
fuel-type             object
aspiration            object
num-of-doors          object
body-style            object
drive-wheels          object
engine-location       object
wheel-base           float64
length               float64
width                float64
height               float64
curb-weight            int64
engine-type           object
num-of-cylinders      object
engine-size            int64
fuel-system           object
bore                  object
stroke                object
compression-ratio    float64
horsepower            object
peak-rpm              object
city-mpg               int64
highway-mpg            int64
price                 object
dtype: object
```

[8]:
```
# Display summary statistics for the numerical columns in the DataFrame
df.describe()
```

[8]:
```
        symboling  wheel-base      length       width      height  \
count  205.000000  205.000000  205.000000  205.000000  205.000000
mean     0.834146   98.756585  174.049268   65.907805   53.724878
```

```
std       1.245307    6.021776   12.337289    2.145204    2.443522
min      -2.000000   86.600000  141.100000   60.300000   47.800000
25%       0.000000   94.500000  166.300000   64.100000   52.000000
50%       1.000000   97.000000  173.200000   65.500000   54.100000
75%       2.000000  102.400000  183.100000   66.900000   55.500000
max       3.000000  120.900000  208.100000   72.300000   59.800000

       curb-weight  engine-size  compression-ratio    city-mpg  highway-mpg
count   205.000000   205.000000         205.000000  205.000000   205.000000
mean   2555.565854   126.907317          10.142537   25.219512    30.751220
std     520.680204    41.642693           3.972040    6.542142     6.886443
min    1488.000000    61.000000           7.000000   13.000000    16.000000
25%    2145.000000    97.000000           8.600000   19.000000    25.000000
50%    2414.000000   120.000000           9.000000   24.000000    30.000000
75%    2935.000000   141.000000           9.400000   30.000000    34.000000
max    4066.000000   326.000000          23.000000   49.000000    54.000000
```

[9]:

```python
# Display summary statistics for all columns, including non-numerical ones
df.describe(include="all")
```

[9]:

```
         symboling normalized-losses    make fuel-type aspiration  \
count   205.000000               164     205       205        205
unique         NaN                51      22         2          2
top            NaN               161  toyota       gas        std
freq           NaN                11      32       185        168
mean      0.834146               NaN     NaN       NaN        NaN
std       1.245307               NaN     NaN       NaN        NaN
min      -2.000000               NaN     NaN       NaN        NaN
25%       0.000000               NaN     NaN       NaN        NaN
50%       1.000000               NaN     NaN       NaN        NaN
75%       2.000000               NaN     NaN       NaN        NaN
max       3.000000               NaN     NaN       NaN        NaN

       num-of-doors body-style drive-wheels engine-location  wheel-base  …  \
count           203        205          205             205  205.000000  …
unique            2          5            3               2         NaN  …
top            four      sedan          fwd           front         NaN  …
freq            114         96          120             202         NaN  …
mean            NaN        NaN          NaN             NaN   98.756585  …
std             NaN        NaN          NaN             NaN    6.021776  …
min             NaN        NaN          NaN             NaN   86.600000  …
25%             NaN        NaN          NaN             NaN   94.500000  …
50%             NaN        NaN          NaN             NaN   97.000000  …
75%             NaN        NaN          NaN             NaN  102.400000  …
max             NaN        NaN          NaN             NaN  120.900000  …

        engine-size  fuel-system  bore  stroke compression-ratio horsepower  \
```

```
count    205.000000          205   201      201          205.000000        203
unique          NaN            8    38       36                 NaN         59
top             NaN         mpfi  3.62     3.40                 NaN         68
freq            NaN           94    23       20                 NaN         19
mean     126.907317          NaN   NaN      NaN           10.142537        NaN
std       41.642693          NaN   NaN      NaN            3.972040        NaN
min       61.000000          NaN   NaN      NaN            7.000000        NaN
25%       97.000000          NaN   NaN      NaN            8.600000        NaN
50%      120.000000          NaN   NaN      NaN            9.000000        NaN
75%      141.000000          NaN   NaN      NaN            9.400000        NaN
max      326.000000          NaN   NaN      NaN           23.000000        NaN

        peak-rpm    city-mpg highway-mpg price
count        203  205.000000  205.000000   201
unique        23         NaN         NaN   186
top         5500         NaN         NaN  8921
freq          37         NaN         NaN     2
mean         NaN   25.219512   30.751220   NaN
std          NaN    6.542142    6.886443   NaN
min          NaN   13.000000   16.000000   NaN
25%          NaN   19.000000   25.000000   NaN
50%          NaN   24.000000   30.000000   NaN
75%          NaN   30.000000   34.000000   NaN
max          NaN   49.000000   54.000000   NaN

[11 rows x 26 columns]
```

[10]:

```
# Display a concise summary of the DataFrame, including data types and non-null
 ↪counts
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 26 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   symboling          205 non-null    int64
 1   normalized-losses  164 non-null    object
 2   make               205 non-null    object
 3   fuel-type          205 non-null    object
 4   aspiration         205 non-null    object
 5   num-of-doors       203 non-null    object
 6   body-style         205 non-null    object
 7   drive-wheels       205 non-null    object
 8   engine-location    205 non-null    object
 9   wheel-base         205 non-null    float64
 10  length             205 non-null    float64
 11  width              205 non-null    float64
```

```
 12  height             205 non-null    float64
 13  curb-weight        205 non-null    int64
 14  engine-type        205 non-null    object
 15  num-of-cylinders   205 non-null    object
 16  engine-size        205 non-null    int64
 17  fuel-system        205 non-null    object
 18  bore               201 non-null    object
 19  stroke             201 non-null    object
 20  compression-ratio  205 non-null    float64
 21  horsepower         203 non-null    object
 22  peak-rpm           203 non-null    object
 23  city-mpg           205 non-null    int64
 24  highway-mpg        205 non-null    int64
 25  price              201 non-null    object
dtypes: float64(5), int64(5), object(16)
memory usage: 41.8+ KB
```

[11]:
```python
# Create a DataFrame indicating the presence of missing values
# Each cell will be True if the value is missing, and False otherwise
missing_data = df.isnull()

# Display the first few rows of the DataFrame showing missing values
missing_data.head()
```

[11]:
```
   symboling  normalized-losses   make  fuel-type  aspiration  num-of-doors  \
0      False               True  False      False       False         False
1      False               True  False      False       False         False
2      False               True  False      False       False         False
3      False              False  False      False       False         False
4      False              False  False      False       False         False

   body-style  drive-wheels  engine-location  wheel-base  …  engine-size  \
0       False         False            False       False  …        False
1       False         False            False       False  …        False
2       False         False            False       False  …        False
3       False         False            False       False  …        False
4       False         False            False       False  …        False

   fuel-system   bore  stroke  compression-ratio  horsepower  peak-rpm  \
0        False  False   False              False       False     False
1        False  False   False              False       False     False
2        False  False   False              False       False     False
3        False  False   False              False       False     False
4        False  False   False              False       False     False

   city-mpg  highway-mpg  price
0     False        False  False
```

```
1     False        False  False
2     False        False  False
3     False        False  False
4     False        False  False

[5 rows x 26 columns]
```

[12]:
```python
# Iterate over each column in the DataFrame that tracks missing values
for column in missing_data.columns.values.tolist():
    # Print the name of the current column
    print(column)

    # Print the count of True (missing) and False (not missing) values for the␣
↪current column
    print(missing_data[column].value_counts())

    # Print a blank line for better readability between columns
    print("")
```

```
symboling
symboling
False    205
Name: count, dtype: int64

normalized-losses
normalized-losses
False    164
True      41
Name: count, dtype: int64

make
make
False    205
Name: count, dtype: int64

fuel-type
fuel-type
False    205
Name: count, dtype: int64

aspiration
aspiration
False    205
Name: count, dtype: int64

num-of-doors
num-of-doors
False    203
```

```
True       2
Name: count, dtype: int64

body-style
body-style
False    205
Name: count, dtype: int64

drive-wheels
drive-wheels
False    205
Name: count, dtype: int64

engine-location
engine-location
False    205
Name: count, dtype: int64

wheel-base
wheel-base
False    205
Name: count, dtype: int64

length
length
False    205
Name: count, dtype: int64

width
width
False    205
Name: count, dtype: int64

height
height
False    205
Name: count, dtype: int64

curb-weight
curb-weight
False    205
Name: count, dtype: int64

engine-type
engine-type
False    205
Name: count, dtype: int64
```

```
num-of-cylinders
num-of-cylinders
False    205
Name: count, dtype: int64

engine-size
engine-size
False    205
Name: count, dtype: int64

fuel-system
fuel-system
False    205
Name: count, dtype: int64

bore
bore
False    201
True       4
Name: count, dtype: int64

stroke
stroke
False    201
True       4
Name: count, dtype: int64

compression-ratio
compression-ratio
False    205
Name: count, dtype: int64

horsepower
horsepower
False    203
True       2
Name: count, dtype: int64

peak-rpm
peak-rpm
False    203
True       2
Name: count, dtype: int64

city-mpg
city-mpg
False    205
Name: count, dtype: int64
```

```
highway-mpg
highway-mpg
False    205
Name: count, dtype: int64

price
price
False    201
True       4
Name: count, dtype: int64
```

[13]:
```
# Convert the "normalized-losses" column to float type and calculate the mean␣
 ↪value
avg_norm_loss = df["normalized-losses"].astype("float").mean(axis=0)
# Print the average of the "normalized-losses" column
print("Average of normalized-losses:", avg_norm_loss)

# Convert the "bore" column to float type and calculate the mean value
avg_bore = df['bore'].astype('float').mean(axis=0)
# Print the average of the "bore" column
print("Average of bore:", avg_bore)

# Convert the "stroke" column to float type and calculate the mean value
avg_stroke = df["stroke"].astype("float").mean(axis=0)
# Print the average of the "stroke" column
print("Average of stroke:", avg_stroke)

# Convert the "horsepower" column to float type and calculate the mean value
avg_horsepower = df['horsepower'].astype('float').mean(axis=0)
# Print the average of the "horsepower" column
print("Average horsepower:", avg_horsepower)

# Convert the "peak-rpm" column to float type and calculate the mean value
avg_peakrpm = df['peak-rpm'].astype('float').mean(axis=0)
# Print the average of the "peak-rpm" column
print("Average peak rpm:", avg_peakrpm)
```

```
Average of normalized-losses: 122.0
Average of bore: 3.3297512437810943
Average of stroke: 3.255422885572139
Average horsepower: 104.25615763546799
Average peak rpm: 5125.369458128079
```

[14]:
```
# Replace missing values in the "normalized-losses" column with the calculated␣
 ↪average
df["normalized-losses"].replace(np.nan, avg_norm_loss, inplace=True)
```

```python
# Replace missing values in the "bore" column with the calculated average
df["bore"].replace(np.nan, avg_bore, inplace=True)

# Replace missing values in the "stroke" column with the calculated average
df["stroke"].replace(np.nan, avg_stroke, inplace=True)

# Replace missing values in the "horsepower" column with the calculated average
df['horsepower'].replace(np.nan, avg_horsepower, inplace=True)

# Replace missing values in the "peak-rpm" column with the calculated average
df['peak-rpm'].replace(np.nan, avg_peakrpm, inplace=True)
```

[15]:
```python
# Count the occurrences of each value in the "num-of-doors" column
print(df['num-of-doors'].value_counts())

# Identify the most frequent value in the "num-of-doors" column
print(df['num-of-doors'].value_counts().idxmax())

# Replace missing values in the "num-of-doors" column with the most frequent␣
 ↪value ("four")
df["num-of-doors"].replace(np.nan, "four", inplace=True)
```

```
num-of-doors
four    114
two      89
Name: count, dtype: int64
four
```

[16]:
```python
# Drop any row in the DataFrame where the "price" column has a missing value␣
 ↪(NaN)
df.dropna(subset=["price"], axis=0, inplace=True)

# Reset the index of the DataFrame after dropping rows, removing the old index
df.reset_index(drop=True, inplace=True)
```

[17]:
```python
df.head()
```

[17]:
```
   symboling normalized-losses         make fuel-type aspiration num-of-doors  \
0          3             122.0  alfa-romero       gas        std          two
1          3             122.0  alfa-romero       gas        std          two
2          1             122.0  alfa-romero       gas        std          two
3          2               164         audi       gas        std         four
4          2               164         audi       gas        std         four

    body-style drive-wheels engine-location  wheel-base  …  engine-size  \
0  convertible          rwd           front        88.6  …          130
1  convertible          rwd           front        88.6  …          130
```

```
2    hatchback          rwd          front         94.5  ...          152
3        sedan          fwd          front         99.8  ...          109
4        sedan          4wd          front         99.4  ...          136

  fuel-system  bore  stroke compression-ratio horsepower  peak-rpm city-mpg  \
0        mpfi  3.47    2.68               9.0        111      5000       21
1        mpfi  3.47    2.68               9.0        111      5000       21
2        mpfi  2.68    3.47               9.0        154      5000       19
3        mpfi  3.19    3.40              10.0        102      5500       24
4        mpfi  3.19    3.40               8.0        115      5500       18

  highway-mpg  price
0          27  13495
1          27  16500
2          26  16500
3          30  13950
4          22  17450

[5 rows x 26 columns]
```

[18]:
```
# Display a concise summary of the DataFrame, including data types and non-null counts
# Initially, there were 205 entries, but after dropping 4 rows with missing 'price' values, there are now 201 entries.
# The following code shows that each column has 201 non-null entries.
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 201 entries, 0 to 200
Data columns (total 26 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   symboling          201 non-null    int64
 1   normalized-losses  201 non-null    object
 2   make               201 non-null    object
 3   fuel-type          201 non-null    object
 4   aspiration         201 non-null    object
 5   num-of-doors       201 non-null    object
 6   body-style         201 non-null    object
 7   drive-wheels       201 non-null    object
 8   engine-location    201 non-null    object
 9   wheel-base         201 non-null    float64
 10  length             201 non-null    float64
 11  width              201 non-null    float64
 12  height             201 non-null    float64
 13  curb-weight        201 non-null    int64
 14  engine-type        201 non-null    object
 15  num-of-cylinders   201 non-null    object
```

```
 16   engine-size        201 non-null    int64
 17   fuel-system        201 non-null    object
 18   bore               201 non-null    object
 19   stroke             201 non-null    object
 20   compression-ratio  201 non-null    float64
 21   horsepower         201 non-null    object
 22   peak-rpm           201 non-null    object
 23   city-mpg           201 non-null    int64
 24   highway-mpg        201 non-null    int64
 25   price              201 non-null    object
dtypes: float64(5), int64(5), object(16)
memory usage: 41.0+ KB
```

[19]:
```
# Some columns have incorrect data types.
# Numerical columns like 'bore' and 'stroke' should be 'float' or 'int' but are␣
↪currently 'object'.
# We'll convert these to the correct types using the 'astype()' method.

# Convert 'bore' and 'stroke' columns to float type
df[["bore", "stroke"]] = df[["bore", "stroke"]].astype("float")

# Convert 'normalized-losses' column to int type
df[["normalized-losses"]] = df[["normalized-losses"]].astype("int")

# Convert 'price' column to float type
df[["price"]] = df[["price"]].astype("float")

# Convert 'peak-rpm' column to float type
df[["peak-rpm"]] = df[["peak-rpm"]].astype("float")

# Verify the changes by displaying the data types of each column
df.dtypes
```

[19]:
```
symboling              int64
normalized-losses      int32
make                  object
fuel-type             object
aspiration            object
num-of-doors          object
body-style            object
drive-wheels          object
engine-location       object
wheel-base           float64
length               float64
width                float64
height               float64
curb-weight            int64
```

```
engine-type           object
num-of-cylinders      object
engine-size            int64
fuel-system           object
bore                 float64
stroke               float64
compression-ratio    float64
horsepower            object
peak-rpm             float64
city-mpg               int64
highway-mpg            int64
price                float64
dtype: object
```

[20]:
```
# Convert 'city-mpg' to L/100km using the formula: L/100km = 235 / mpg
df['city-L/100km'] = 235 / df["city-mpg"]

# Convert 'highway-mpg' to L/100km using the formula: L/100km = 235 / mpg
df["highway-L/100km"] = 235 / df["highway-mpg"]

# Display the first few rows of the DataFrame to verify the changes
df.head()
```

[20]:
```
   symboling  normalized-losses         make fuel-type aspiration  \
0          3                122  alfa-romero       gas        std
1          3                122  alfa-romero       gas        std
2          1                122  alfa-romero       gas        std
3          2                164         audi       gas        std
4          2                164         audi       gas        std

  num-of-doors   body-style drive-wheels engine-location  wheel-base  …  \
0          two  convertible          rwd           front        88.6  …
1          two  convertible          rwd           front        88.6  …
2          two    hatchback          rwd           front        94.5  …
3         four        sedan          fwd           front        99.8  …
4         four        sedan          4wd           front        99.4  …

   bore  stroke  compression-ratio  horsepower peak-rpm city-mpg  highway-mpg  \
0  3.47    2.68                9.0         111   5000.0       21           27
1  3.47    2.68                9.0         111   5000.0       21           27
2  2.68    3.47                9.0         154   5000.0       19           26
3  3.19    3.40               10.0         102   5500.0       24           30
4  3.19    3.40                8.0         115   5500.0       18           22

     price  city-L/100km  highway-L/100km
0  13495.0     11.190476         8.703704
1  16500.0     11.190476         8.703704
```

```
2  16500.0     12.368421         9.038462
3  13950.0      9.791667         7.833333
4  17450.0     13.055556        10.681818

[5 rows x 28 columns]
```

[21]:
```
# Normalize 'length', 'width', and 'height' columns by scaling values to the␣
↪range 0 to 1
df['length'] = df['length'] / df['length'].max()
df['width'] = df['width'] / df['width'].max()
df['height'] = df['height'] / df['height'].max()

# Display the first few rows of the normalized columns
df[["length", "width", "height"]].head()
```

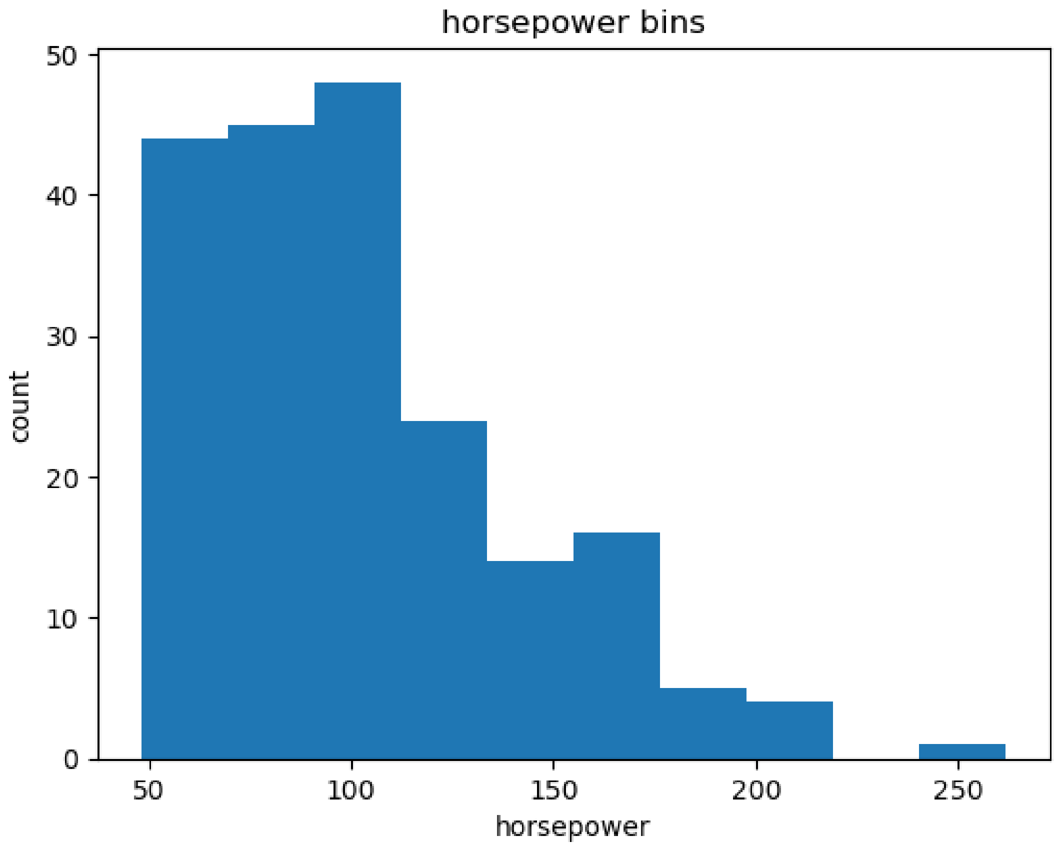
[21]:

	length	width	height
0	0.811148	0.890278	0.816054
1	0.811148	0.890278	0.816054
2	0.822681	0.909722	0.876254
3	0.848630	0.919444	0.908027
4	0.848630	0.922222	0.908027

[22]:
```
# Convert the 'horsepower' column to integer type to prepare for binning
df["horsepower"] = df["horsepower"].astype(int, copy=True)
```

[23]:
```
%matplotlib inline
import matplotlib as plt
from matplotlib import pyplot
plt.pyplot.hist(df["horsepower"])

# set x/y labels and plot title
plt.pyplot.xlabel("horsepower")
plt.pyplot.ylabel("count")
plt.pyplot.title("horsepower bins")
```

[23]:
```
Text(0.5, 1.0, 'horsepower bins')
```



[24]:
```
import numpy as np

# Define bin edges
bins = np.linspace(min(df["horsepower"]), max(df["horsepower"]), 4)
print(bins)

# Define bin labels
group_names = ['Low', 'Medium', 'High']

# Create a new column 'horsepower-binned' with bin labels
df['horsepower-binned'] = pd.cut(df['horsepower'], bins, labels=group_names,␣
    ↪include_lowest=True )
print(df[['horsepower','horsepower-binned']].head())
df["horsepower-binned"].value_counts()
```

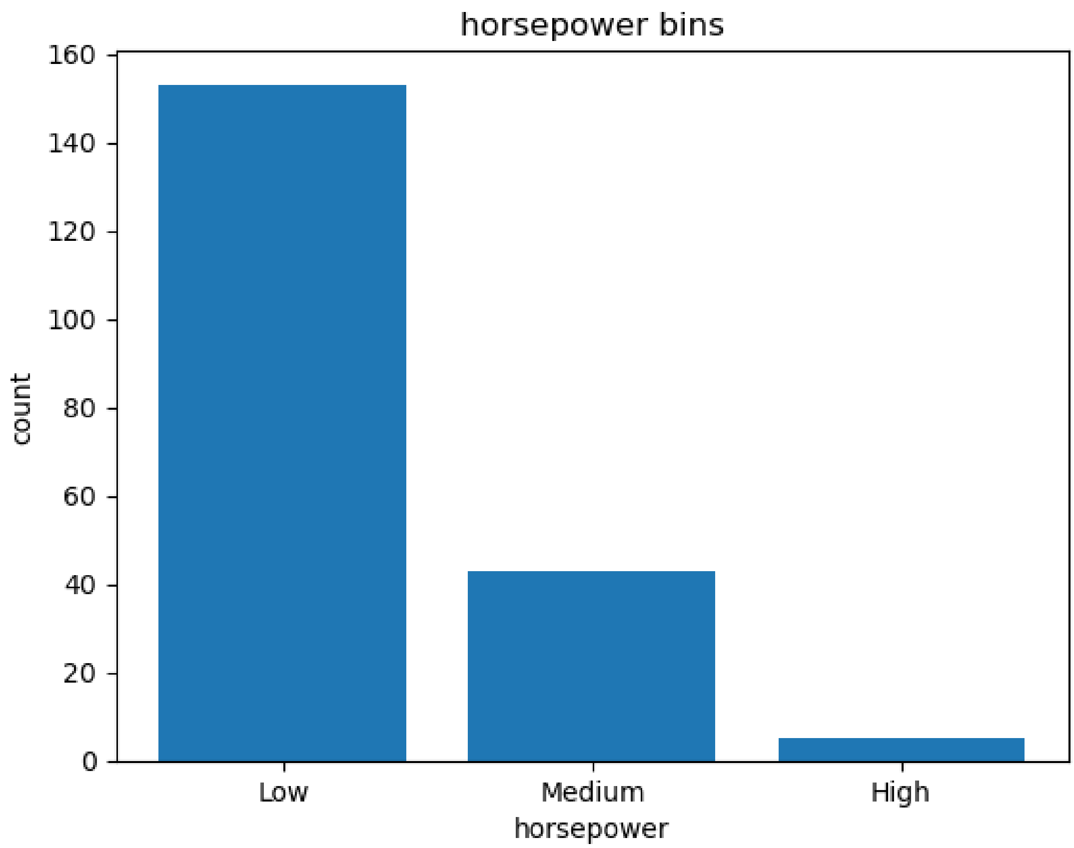
```
[ 48.         119.33333333 190.66666667 262.        ]
   horsepower horsepower-binned
0         111               Low
1         111               Low
2         154            Medium
```

```
3          102                 Low
4          115                 Low
```

[24]:
```
horsepower-binned
Low       153
Medium     43
High        5
Name: count, dtype: int64
```

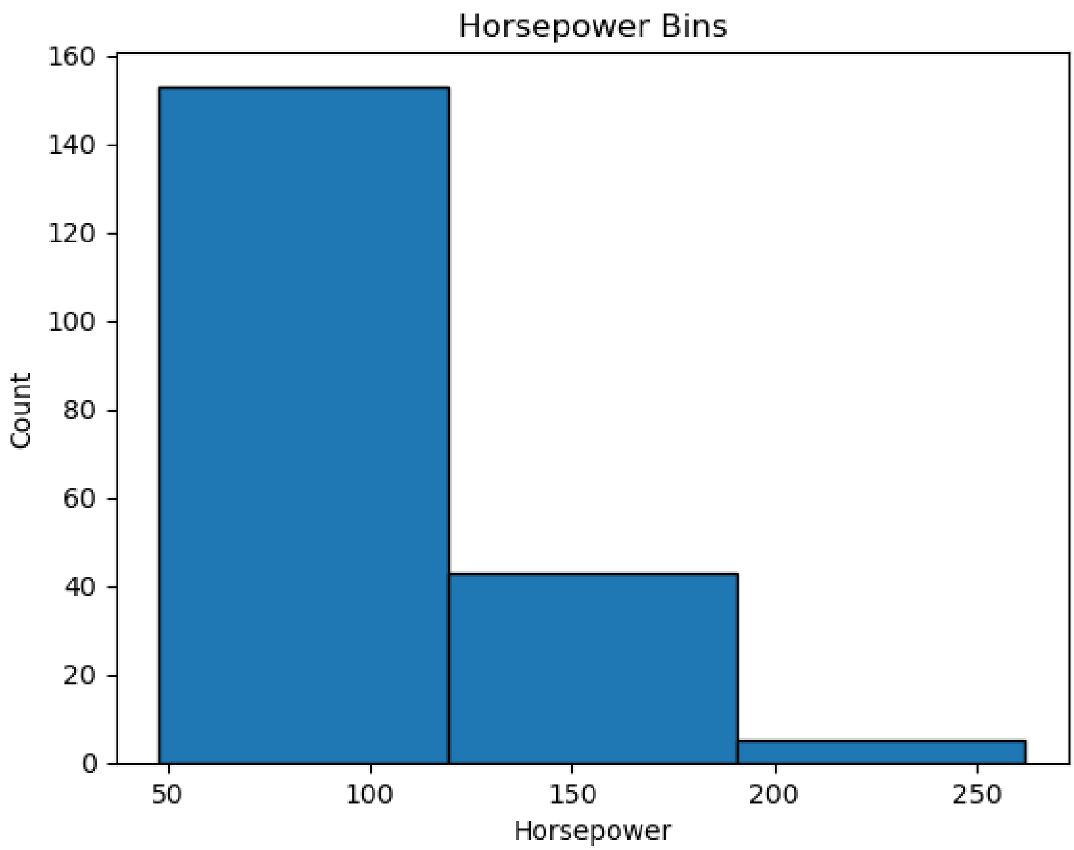
[25]:
```python
# Create a bar chart of the horsepower bins
pyplot.bar(group_names, df["horsepower-binned"].value_counts())

# set x/y labels and plot title
plt.pyplot.xlabel("horsepower")
plt.pyplot.ylabel("count")
plt.pyplot.title("horsepower bins")
```

[25]:
```
Text(0.5, 1.0, 'horsepower bins')
```



[26]:
```python
# Using 3 bins to categorize the 'horsepower' values into three distinct ranges
plt.pyplot.hist(df["horsepower"], bins=3, edgecolor='black')

# Set the label for the x-axis to describe the variable plotted
plt.pyplot.xlabel("Horsepower")

# Set the label for the y-axis to show the count of occurrences for each bin
plt.pyplot.ylabel("Count")

# Set the title of the plot to describe what is being visualized
plt.pyplot.title("Horsepower Bins")
```

[26]:
```
Text(0.5, 1.0, 'Horsepower Bins')
```



[27]:
```python
# Create dummy variables for the "fuel-type" column
dummy_variable_1 = pd.get_dummies(df["fuel-type"])

# Rename the columns to more descriptive names
dummy_variable_1.rename(columns={'gas': 'fuel-type-gas', 'diesel':␣
 ↪'fuel-type-diesel'}, inplace=True)
```

```python
# Display the updated dummy variables for "fuel-type"
print(dummy_variable_1.head())

# Create dummy variables for the "aspiration" column
dummy_variable_2 = pd.get_dummies(df["aspiration"])

# Rename the columns to more descriptive names
dummy_variable_2.rename(columns={'std': 'aspiration-std', 'turbo':
 ↪'aspiration-turbo'}, inplace=True)
# Display the updated dummy variables for "aspiration"
print("")
print(dummy_variable_2.head())
```

```
   fuel-type-diesel  fuel-type-gas
0             False           True
1             False           True
2             False           True
3             False           True
4             False           True

   aspiration-std  aspiration-turbo
0            True             False
1            True             False
2            True             False
3            True             False
4            True             False
```

[28]:
```python
# Merge the new dummy variable dataframes with the original dataframe
df = pd.concat([df, dummy_variable_1, dummy_variable_2], axis=1)

# Display the updated dataframe with new dummy variables
df.head()
```

[28]:
```
   symboling  normalized-losses         make fuel-type aspiration  \
0          3                122  alfa-romero       gas        std
1          3                122  alfa-romero       gas        std
2          1                122  alfa-romero       gas        std
3          2                164         audi       gas        std
4          2                164         audi       gas        std

  num-of-doors   body-style drive-wheels engine-location  wheel-base  …  \
0          two  convertible          rwd           front        88.6  …
1          two  convertible          rwd           front        88.6  …
2          two    hatchback          rwd           front        94.5  …
3         four        sedan          fwd           front        99.8  …
4         four        sedan          4wd           front        99.4  …
```

```
   city-mpg  highway-mpg    price  city-L/100km highway-L/100km  \
0        21           27  13495.0     11.190476        8.703704
1        21           27  16500.0     11.190476        8.703704
2        19           26  16500.0     12.368421        9.038462
3        24           30  13950.0      9.791667        7.833333
4        18           22  17450.0     13.055556       10.681818

  horsepower-binned  fuel-type-diesel fuel-type-gas  aspiration-std  \
0               Low             False          True            True
1               Low             False          True            True
2            Medium             False          True            True
3               Low             False          True            True
4               Low             False          True            True

   aspiration-turbo
0             False
1             False
2             False
3             False
4             False

[5 rows x 33 columns]
```

[29]:
```python
# Save the cleaned and updated DataFrame to a CSV file
df.to_csv('clean_df.csv', index=False)
```

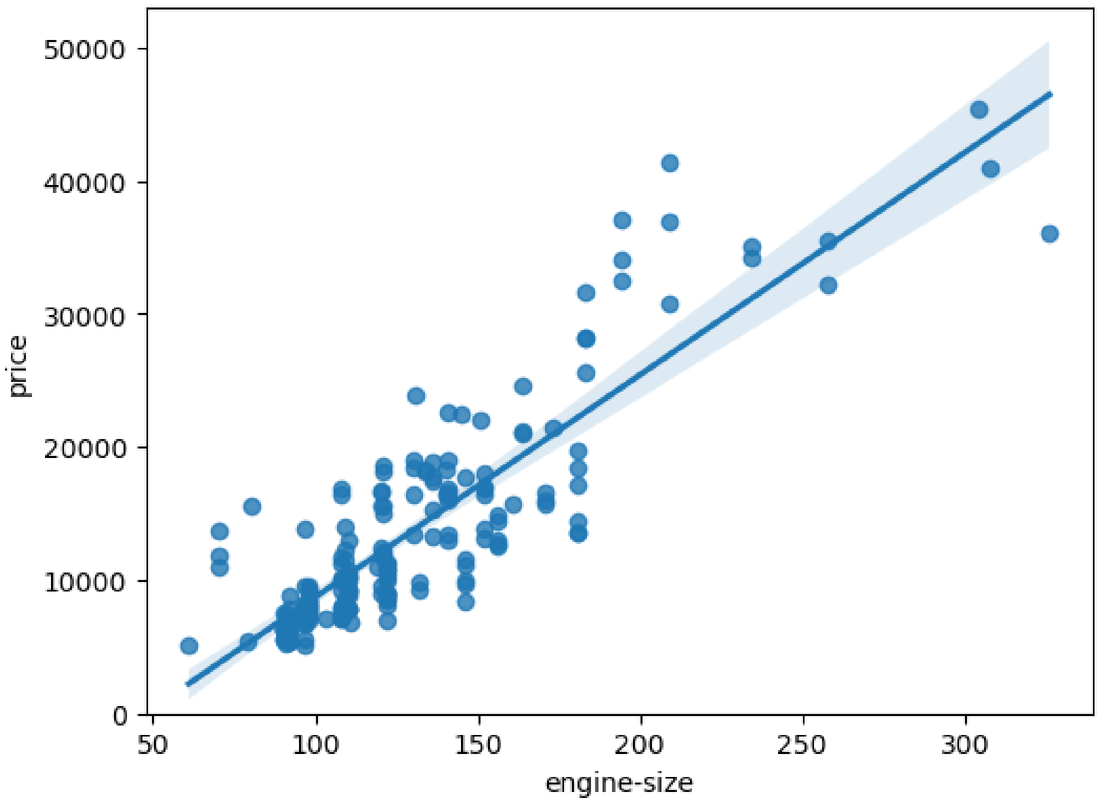
[30]:
```python
df.head()
```

[30]:
```
   symboling  normalized-losses         make fuel-type aspiration  \
0          3                122  alfa-romero       gas        std
1          3                122  alfa-romero       gas        std
2          1                122  alfa-romero       gas        std
3          2                164         audi       gas        std
4          2                164         audi       gas        std

  num-of-doors   body-style drive-wheels engine-location  wheel-base  …  \
0          two  convertible          rwd           front        88.6  …
1          two  convertible          rwd           front        88.6  …
2          two    hatchback          rwd           front        94.5  …
3         four        sedan          fwd           front        99.8  …
4         four        sedan          4wd           front        99.4  …

   city-mpg  highway-mpg    price  city-L/100km highway-L/100km  \
0        21           27  13495.0     11.190476        8.703704
1        21           27  16500.0     11.190476        8.703704
2        19           26  16500.0     12.368421        9.038462
```

```
3        24            30  13950.0      9.791667        7.833333
4        18            22  17450.0     13.055556       10.681818

  horsepower-binned  fuel-type-diesel fuel-type-gas  aspiration-std  \
0               Low             False          True            True
1               Low             False          True            True
2            Medium             False          True            True
3               Low             False          True            True
4               Low             False          True            True

   aspiration-turbo
0             False
1             False
2             False
3             False
4             False

[5 rows x 33 columns]
```

[31]:

```python
import seaborn as sns
import matplotlib.pyplot as plt  # Import matplotlib.pyplot for plotting␣
 ↪functions

# Create a regression plot to visualize the relationship between "engine-size"␣
 ↪and "price"
sns.regplot(x="engine-size", y="price", data=df)

# Set the lower limit of the y-axis to 0 for better visibility
plt.ylim(0,)

# Display the plot
plt.show()
```



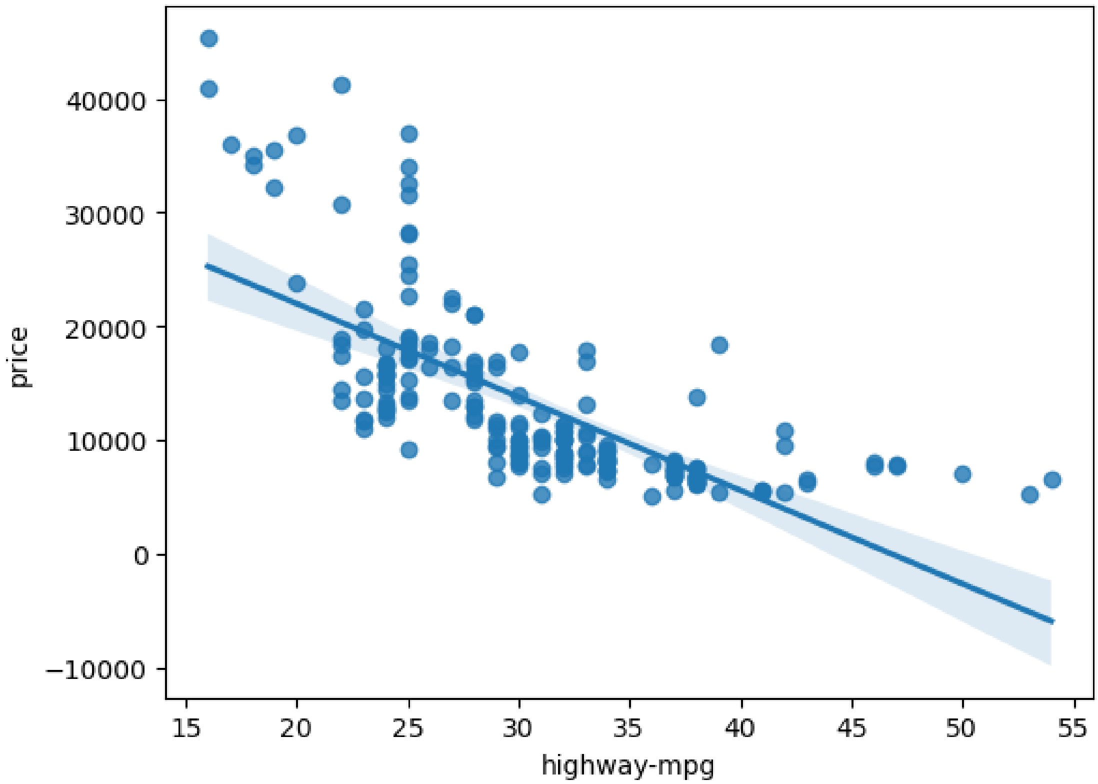
[32]:
```
# Calculate and display the correlation matrix between "engine-size" and "price"
df[["engine-size", "price"]].corr()
```

[32]:

	engine-size	price
engine-size	1.000000	0.872335
price	0.872335	1.000000

[33]:
```
# Plot regression line and scatter plot for "highway-mpg" vs. "price"
sns.regplot(x="highway-mpg", y="price", data=df)
```

[33]:
```
<Axes: xlabel='highway-mpg', ylabel='price'>
```

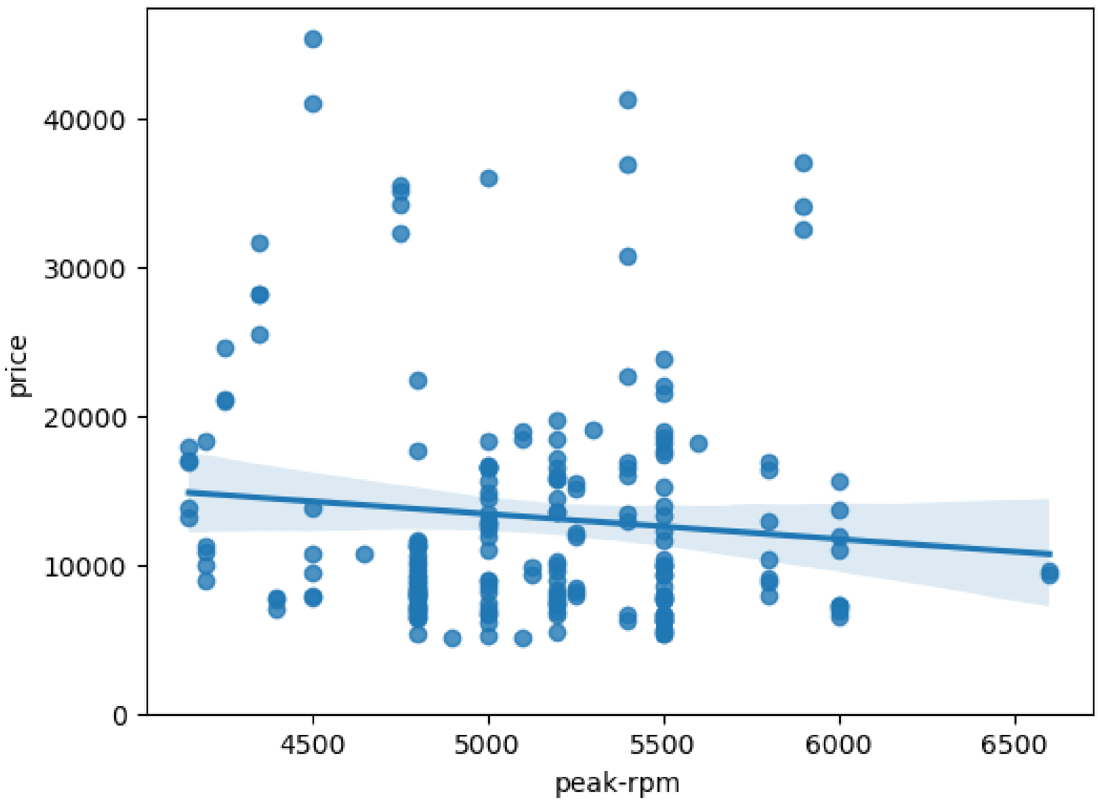


[34]:
```
# Compute and display the correlation matrix between "highway-mpg" and "price"
df[['highway-mpg', 'price']].corr()
```

[34]:

	highway-mpg	price
highway-mpg	1.000000	-0.704692
price	-0.704692	1.000000

[35]:
```
# Create a regression plot to visualize the relationship between "peak-rpm" and␣
↪"price"
sns.regplot(x="peak-rpm", y="price", data=df)

# Set the lower limit of the y-axis to 0 for better visibility
plt.ylim(0,)

# Display the plot
plt.show()
```



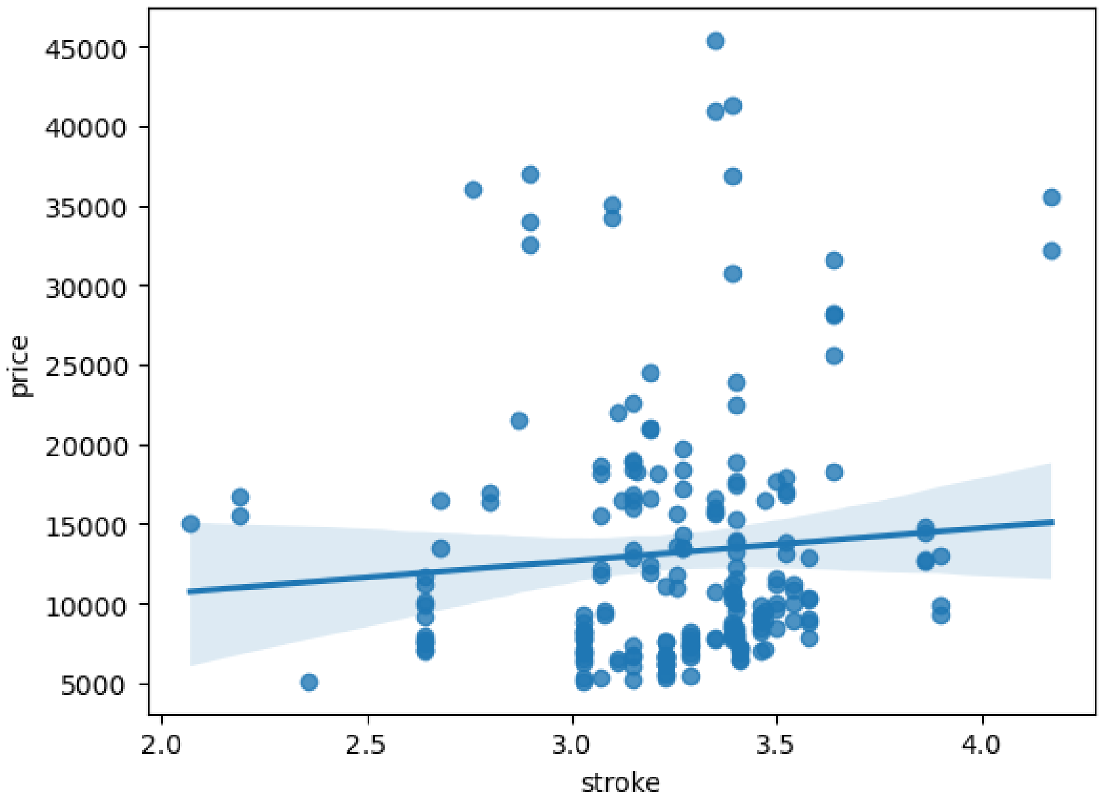
[36]:
```
# Calculate and display the correlation matrix between "peak-rpm" and "price"
df[['peak-rpm', 'price']].corr()
```

[36]:

	peak-rpm	price
peak-rpm	1.000000	-0.101616
price	-0.101616	1.000000

[37]:
```
# Create a regression plot to visualize the relationship between "stroke" and
↪"price"
sns.regplot(x="stroke", y="price", data=df)
```

[37]: <Axes: xlabel='stroke', ylabel='price'>



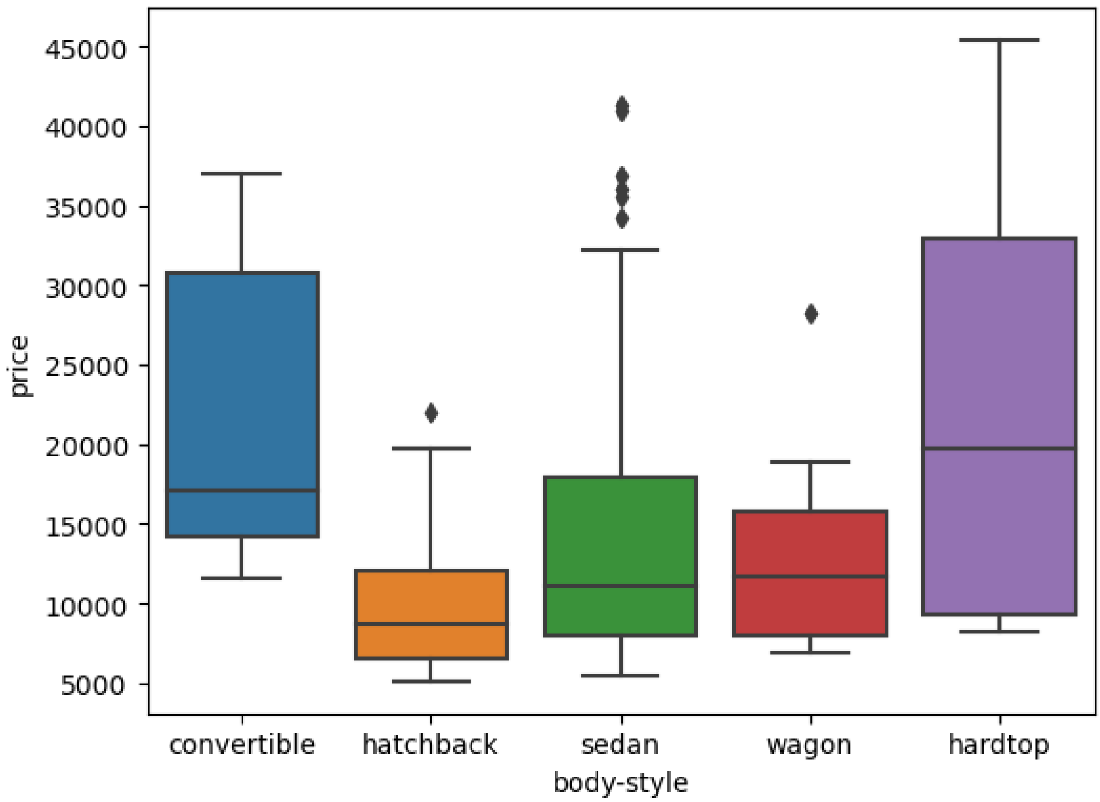
[38]:
```
# Calculate and display the correlation matrix between "stroke" and "price"
df[["stroke", "price"]].corr()
```

[38]:

	stroke	price
stroke	1.000000	0.082269
price	0.082269	1.000000

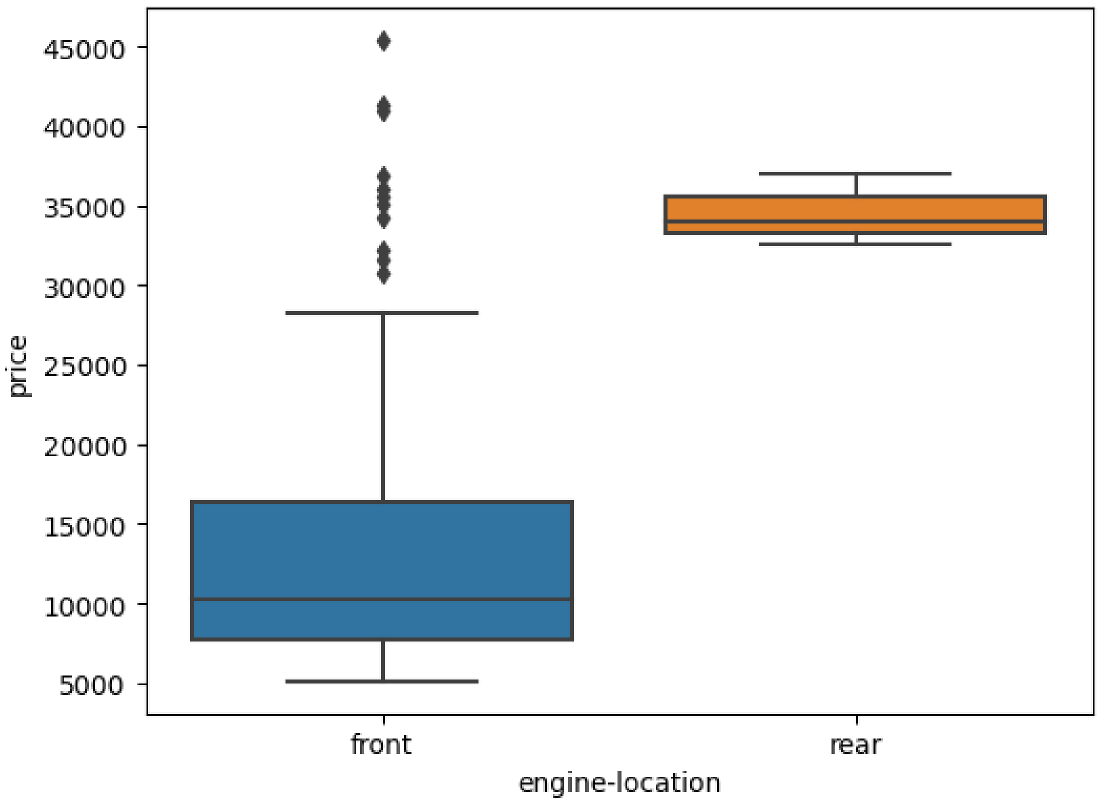
[39]:
```
# Create a boxplot to visualize the distribution of "price" across different␣
↪"body-style" categories
sns.boxplot(x="body-style", y="price", data=df)
```

[39]: <Axes: xlabel='body-style', ylabel='price'>



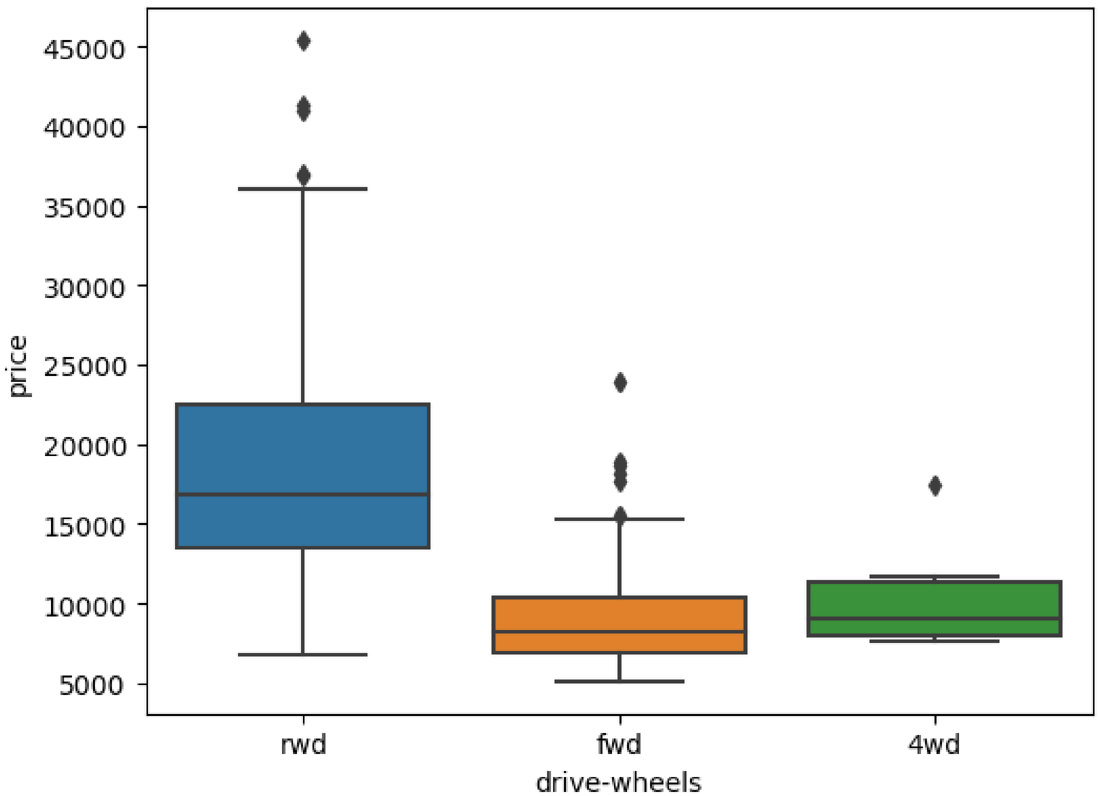
[40]:
```
# Create a boxplot to visualize the distribution of "price" across different␣
↪"engine-location" categories
sns.boxplot(x="engine-location", y="price", data=df)
```

[40]: <Axes: xlabel='engine-location', ylabel='price'>



[41]:
```
# Create a boxplot to visualize the distribution of "price" across different␣
↪"drive-wheels" categories
sns.boxplot(x="drive-wheels", y="price", data=df)
```

[41]: <Axes: xlabel='drive-wheels', ylabel='price'>



[42]:
```
# Generate descriptive statistics for the dataset after data wrangling
df.describe()
```

[42]:
```
        symboling  normalized-losses  wheel-base      length       width  \
count  201.000000          201.00000  201.000000  201.000000  201.000000
mean     0.840796          122.00000   98.797015    0.837102    0.915126
std      1.254802           31.99625    6.066366    0.059213    0.029187
min     -2.000000           65.00000   86.600000    0.678039    0.837500
25%      0.000000          101.00000   94.500000    0.801538    0.890278
50%      1.000000          122.00000   97.000000    0.832292    0.909722
75%      2.000000          137.00000  102.400000    0.881788    0.925000
max      3.000000          256.00000  120.900000    1.000000    1.000000

           height  curb-weight  engine-size        bore      stroke  \
count  201.000000   201.000000   201.000000  201.000000  201.000000
mean     0.899108  2555.666667   126.875622    3.330692    3.256874
std      0.040933   517.296727    41.546834    0.268072    0.316048
min      0.799331  1488.000000    61.000000    2.540000    2.070000
25%      0.869565  2169.000000    98.000000    3.150000    3.110000
50%      0.904682  2414.000000   120.000000    3.310000    3.290000
75%      0.928094  2926.000000   141.000000    3.580000    3.410000
```

```
max      1.000000  4066.000000   326.000000    3.940000    4.170000

       compression-ratio  horsepower     peak-rpm    city-mpg  highway-mpg  \
count         201.000000  201.000000   201.000000  201.000000   201.000000
mean           10.164279  103.402985  5117.665368   25.179104    30.686567
std             4.004965   37.365650   478.113805    6.423220     6.815150
min             7.000000   48.000000  4150.000000   13.000000    16.000000
25%             8.600000   70.000000  4800.000000   19.000000    25.000000
50%             9.000000   95.000000  5125.369458   24.000000    30.000000
75%             9.400000  116.000000  5500.000000   30.000000    34.000000
max            23.000000  262.000000  6600.000000   49.000000    54.000000

              price  city-L/100km  highway-L/100km
count    201.000000    201.000000       201.000000
mean   13207.129353      9.944145         8.044957
std     7947.066342      2.534599         1.840739
min     5118.000000      4.795918         4.351852
25%     7775.000000      7.833333         6.911765
50%    10295.000000      9.791667         7.833333
75%    16500.000000     12.368421         9.400000
max    45400.000000     18.076923        14.687500
```

[43]:
```
# Generate descriptive statistics for categorical variables in the dataset
df.describe(include=['object'])
```

[43]:
```
          make fuel-type aspiration num-of-doors body-style drive-wheels  \
count      201       201        201          201        201          201
unique      22         2          2            2          5            3
top     toyota       gas        std         four      sedan          fwd
freq        32       181        165          115         94          118

       engine-location engine-type num-of-cylinders fuel-system
count              201         201              201         201
unique               2           6                7           8
top              front         ohc             four        mpfi
freq               198         145              157          92
```

[44]:
```
# Count the frequency of each unique value in the 'drive-wheels' column and␣
↪convert the result to a DataFrame
drive_wheels_counts = df['drive-wheels'].value_counts().to_frame()

# Rename the column 'drive-wheels' to 'value_counts' for clarity
drive_wheels_counts.rename(columns={'drive-wheels': 'value_counts'},␣
↪inplace=True)

# Set the index name to 'drive-wheels' for better readability in the output
drive_wheels_counts.index.name = 'drive-wheels'
```

```
# Display the DataFrame with the count of each 'drive-wheels' category
drive_wheels_counts
```

[44]:

```
              count
drive-wheels
fwd             118
rwd              75
4wd               8
```

[45]:

```
# Count the frequency of each unique value in the 'engine-location' column and␣
↪convert the result to a DataFrame
engine_loc_counts = df['engine-location'].value_counts().to_frame()

# Rename the column 'engine-location' to 'value_counts' for clarity
engine_loc_counts.rename(columns={'engine-location': 'value_counts'},␣
↪inplace=True)

# Set the index name to 'engine-location' for better readability in the output
engine_loc_counts.index.name = 'engine-location'

# Display the DataFrame with the count of each 'engine-location' category
engine_loc_counts.head()
```

[45]:

```
                 count
engine-location
front              198
rear                 3
```

[46]:

```
# Select relevant columns for analysis
df_group_one = df[['drive-wheels', 'price']]

# Group by 'drive-wheels' and calculate the mean 'price' for each group
df_group_one = df_group_one.groupby(['drive-wheels'], as_index=False).mean()

# Display the resulting DataFrame
df_group_one
```

[46]:

```
  drive-wheels         price
0          4wd  10241.000000
1          fwd   9244.779661
2          rwd  19757.613333
```

[47]:

```
# Select relevant columns for analysis: 'body-style' and 'price'
df_gptest2 = df[['body-style', 'price']]
```

```
# Group the data by 'body-style' and calculate the mean 'price' for each␣
↪'body-style'
# The parameter 'as_index=False' ensures that 'body-style' remains a column in␣
↪the result DataFrame
grouped_test_bodystyle = df_gptest2.groupby(['body-style'], as_index=False).
↪mean()

# Display the resulting DataFrame showing the mean price for each body style
grouped_test_bodystyle
```

[47]:

	body-style	price
0	convertible	21890.500000
1	hardtop	22208.500000
2	hatchback	9957.441176
3	sedan	14459.755319
4	wagon	12371.960000

[48]:

```
# Select relevant columns for analysis: 'drive-wheels', 'body-style', and␣
↪'price'
df_gptest = df[['drive-wheels', 'body-style', 'price']]

# Group the data by 'drive-wheels' and 'body-style', and calculate the mean␣
↪'price' for each combination
# The parameter 'as_index=False' ensures that 'drive-wheels' and 'body-style'␣
↪remain columns in the result DataFrame
grouped_test1 = df_gptest.groupby(['drive-wheels', 'body-style'],␣
↪as_index=False).mean()

# Display the resulting DataFrame showing the mean price for each combination␣
↪of 'drive-wheels' and 'body-style'
grouped_test1
```

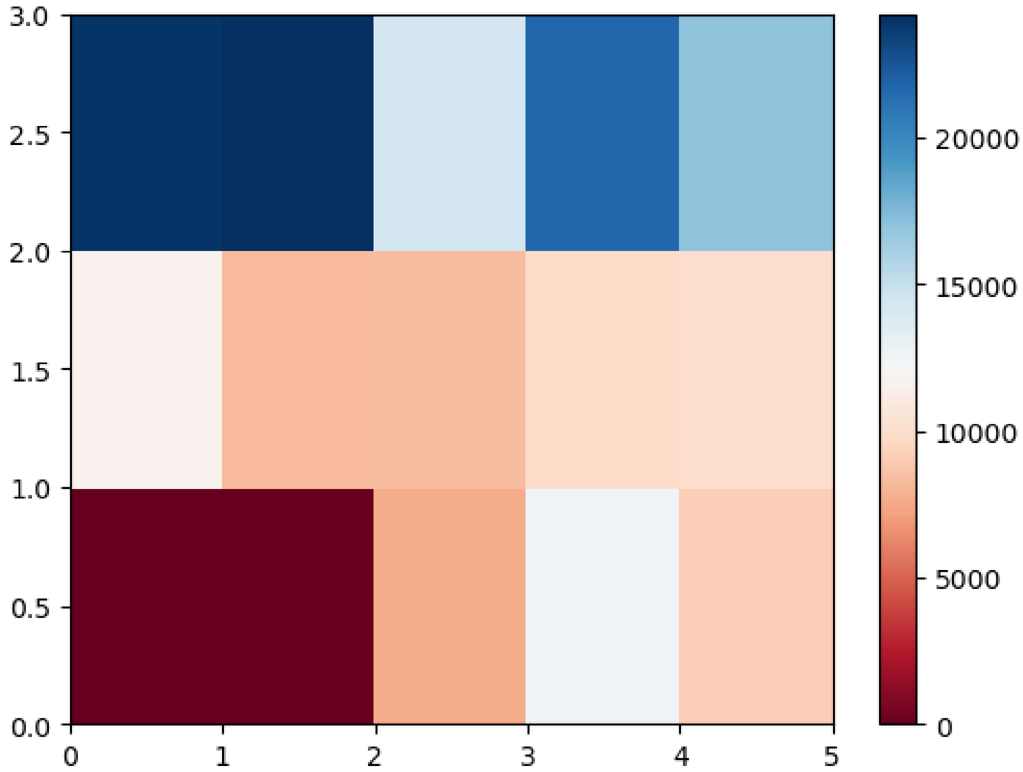
[48]:

	drive-wheels	body-style	price
0	4wd	hatchback	7603.000000
1	4wd	sedan	12647.333333
2	4wd	wagon	9095.750000
3	fwd	convertible	11595.000000
4	fwd	hardtop	8249.000000
5	fwd	hatchback	8396.387755
6	fwd	sedan	9811.800000
7	fwd	wagon	9997.333333
8	rwd	convertible	23949.600000
9	rwd	hardtop	24202.714286
10	rwd	hatchback	14337.777778
11	rwd	sedan	21711.833333
12	rwd	wagon	16994.222222

[49]:
```
# Pivot the grouped DataFrame to reformat it, with 'drive-wheels' as rows and␣
↪'body-style' as columns
# This transformation makes it easier to analyze and compare mean prices across␣
↪different body styles and drive wheels
grouped_pivot = grouped_test1.pivot(index='drive-wheels', columns='body-style')

# Fill missing values in the pivoted DataFrame with 0
# This step is important to handle any gaps in the data where certain␣
↪combinations of 'drive-wheels' and 'body-style' do not exist
grouped_pivot = grouped_pivot.fillna(0)

# Display the pivoted DataFrame with missing values filled
grouped_pivot
```

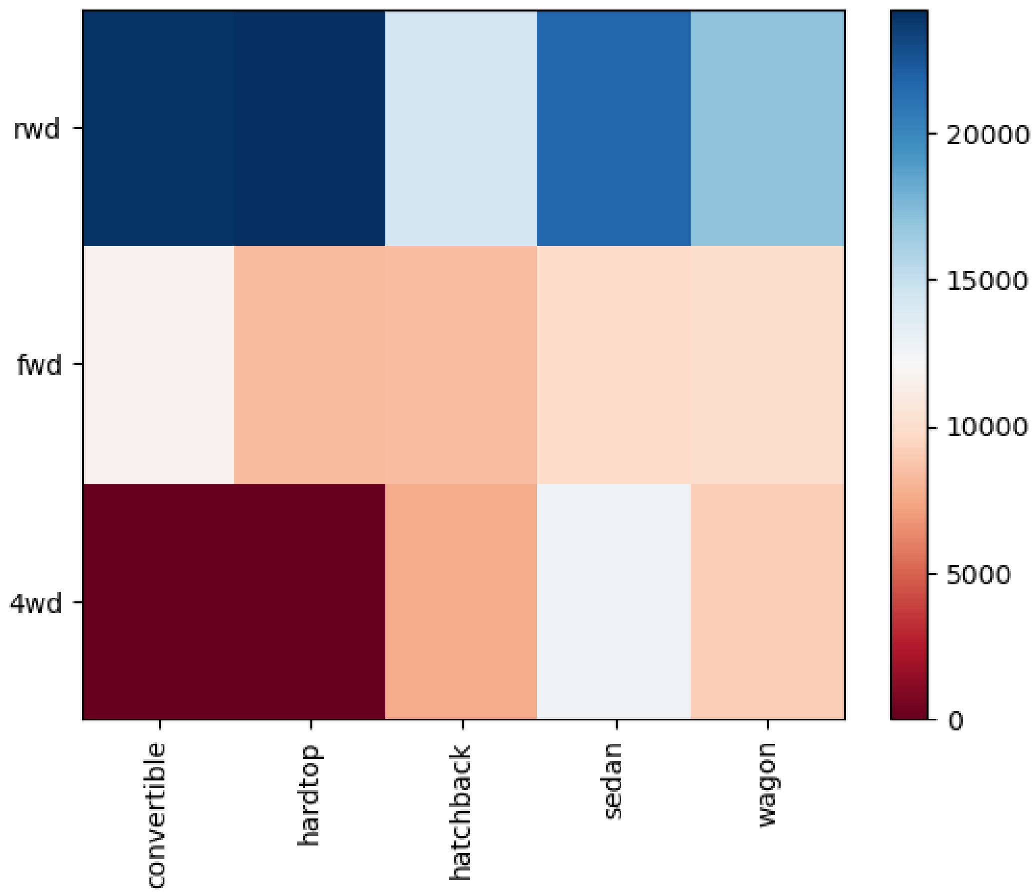
[49]:
```
                    price                                               \
body-style    convertible       hardtop     hatchback         sedan
drive-wheels
4wd                   0.0      0.000000   7603.000000  12647.333333
fwd               11595.0   8249.000000   8396.387755   9811.800000
rwd               23949.6  24202.714286  14337.777778  21711.833333


body-style            wagon
drive-wheels
4wd             9095.750000
fwd             9997.333333
rwd            16994.222222
```

[50]:
```
# Use the grouped results to create a heatmap-like plot
# 'pcolor' is used to display the values in the pivoted DataFrame as a␣
↪color-coded grid
plt.pcolor(grouped_pivot, cmap='RdBu')

# Add a color bar to the plot for reference, showing the mapping between color␣
↪and value
plt.colorbar()

# Display the plot
plt.show()
```



[51]:
```
# Create a figure and axis object for the plot
fig, ax = plt.subplots()

# Generate a pseudocolor plot (heatmap) using the grouped pivot data
im = ax.pcolor(grouped_pivot, cmap='RdBu')

# Define the labels for the columns and rows
row_labels = grouped_pivot.columns.levels[1]  # Get the body-style labels␣
↪(columns)
col_labels = grouped_pivot.index  # Get the drive-wheels labels (rows)

# Position the ticks and labels in the center of each cell
ax.set_xticks(np.arange(grouped_pivot.shape[1]) + 0.5, minor=False)  # X-axis␣
↪(body-style)
ax.set_yticks(np.arange(grouped_pivot.shape[0]) + 0.5, minor=False)  # Y-axis␣
↪(drive-wheels)

# Set the labels for the X and Y ticks
ax.set_xticklabels(row_labels, minor=False)  # Apply body-style labels to X-axis
ax.set_yticklabels(col_labels, minor=False)  # Apply drive-wheels labels to␣
↪Y-axis
```

```
# Rotate the X-axis labels if they are too long for better readability
plt.xticks(rotation=90)

# Add a color bar to provide a reference for the color mapping
fig.colorbar(im)

# Display the plot
plt.show()
```



[52]:
```
from scipy import stats

# Calculate Pearson correlation coefficient and P-value for 'wheel-base' and␣
↪'price'
pearson_coef, p_value = stats.pearsonr(df['wheel-base'], df['price'])
print("The Pearson Correlation Coefficient for Wheel-Base vs. Price is",␣
↪pearson_coef, " with a P-value of P =", p_value)
```

```
# Calculate Pearson correlation coefficient and P-value for 'horsepower' and␣
↪'price'
pearson_coef, p_value = stats.pearsonr(df['horsepower'], df['price'])
print("The Pearson Correlation Coefficient for Horsepower vs. Price is",␣
↪pearson_coef, " with a P-value of P =", p_value)

# Calculate Pearson correlation coefficient and P-value for 'length' and 'price'
pearson_coef, p_value = stats.pearsonr(df['length'], df['price'])
print("The Pearson Correlation Coefficient for Length vs. Price is",␣
↪pearson_coef, " with a P-value of P = ", p_value)

# Calculate Pearson correlation coefficient and P-value for 'width' and 'price'
pearson_coef, p_value = stats.pearsonr(df['width'], df['price'])
print("The Pearson Correlation Coefficient for Width vs. Price is",␣
↪pearson_coef, " with a P-value of P =", p_value)

# Calculate Pearson correlation coefficient and P-value for 'curb-weight' and␣
↪'price'
pearson_coef, p_value = stats.pearsonr(df['curb-weight'], df['price'])
print("The Pearson Correlation Coefficient for Curb-Weight vs. Price is",␣
↪pearson_coef, " with a P-value of P =", p_value)

# Calculate Pearson correlation coefficient and P-value for 'engine-size' and␣
↪'price'
pearson_coef, p_value = stats.pearsonr(df['engine-size'], df['price'])
print("The Pearson Correlation Coefficient for Engine-Size vs. Price is",␣
↪pearson_coef, " with a P-value of P =", p_value)

# Calculate Pearson correlation coefficient and P-value for 'bore' and 'price'
pearson_coef, p_value = stats.pearsonr(df['bore'], df['price'])
print("The Pearson Correlation Coefficient for Bore vs. Price is",␣
↪pearson_coef, " with a P-value of P =  ", p_value)

# Calculate Pearson correlation coefficient and P-value for 'city-mpg' and␣
↪'price'
pearson_coef, p_value = stats.pearsonr(df['city-mpg'], df['price'])
print("The Pearson Correlation Coefficient for City-mpg vs. Price is",␣
↪pearson_coef, " with a P-value of P = ", p_value)

# Calculate Pearson correlation coefficient and P-value for 'highway-mpg' and␣
↪'price'
pearson_coef, p_value = stats.pearsonr(df['highway-mpg'], df['price'])
print("The Pearson Correlation Coefficient for Highway-mpg vs. Price is",␣
↪pearson_coef, " with a P-value of P = ", p_value)
```

```
The Pearson Correlation Coefficient for Wheel-Base vs. Price is
0.5846418222655083  with a P-value of P = 8.076488270732552e-20
```

```
The Pearson Correlation Coefficient for Horsepower vs. Price is
0.8096068016571052  with a P-value of P = 6.273536270651023e-48
The Pearson Correlation Coefficient for Length vs. Price is 0.6906283804483644
with a P-value of P =  8.016477466158383e-30
The Pearson Correlation Coefficient for Width vs. Price is 0.7512653440522674
with a P-value of P = 9.20033551048144e-38
The Pearson Correlation Coefficient for Curb-Weight vs. Price is
0.8344145257702849  with a P-value of P = 2.189577238893391e-53
The Pearson Correlation Coefficient for Engine-Size vs. Price is
0.8723351674455185  with a P-value of P = 9.265491622198793e-64
The Pearson Correlation Coefficient for Bore vs. Price is 0.5431553832626606
with a P-value of P =   8.049189483935034e-17
The Pearson Correlation Coefficient for City-mpg vs. Price is
-0.6865710067844681  with a P-value of P =  2.3211320655673725e-29
The Pearson Correlation Coefficient for Highway-mpg vs. Price is
-0.7046922650589533  with a P-value of P =  1.7495471144474792e-31
```

[53]:
```python
# Group the dataset by 'drive-wheels' and select the 'price' column for analysis
grouped_test2 = df_gptest[['drive-wheels', 'price']].groupby(['drive-wheels'])

# Display the first few rows of the grouped data for verification
grouped_test2.head()

# Access the 'price' values for the '4wd' drive-wheels group
grouped_test2.get_group('4wd')['price']

# Perform ANOVA to compare the means of 'price' across different 'drive-wheels'␣
↪groups
# We use the 'f_oneway' function from scipy.stats to calculate the F-test score␣
↪and p-value
f_val, p_val = stats.f_oneway(
    grouped_test2.get_group('fwd')['price'],
    grouped_test2.get_group('rwd')['price'],
    grouped_test2.get_group('4wd')['price']
)

# Print the results of the ANOVA test
print("ANOVA results for 'drive-wheels' and 'price': F=", f_val, ", P =", p_val)

# Group the dataset by 'drive-wheels' and select the 'price' column for analysis
grouped_test2 = df_gptest[['drive-wheels', 'price']].groupby(['drive-wheels'])

# Display the first few rows of the grouped data for verification
grouped_test2.head()

# Access the 'price' values for the '4wd' drive-wheels group
grouped_test2.get_group('4wd')['price']
```

```python
# Perform ANOVA to compare the means of 'price' across 'fwd' and 'rwd'␣
↪drive-wheels groups
# We use the 'f_oneway' function from scipy.stats to calculate the F-test score␣
↪and p-value
f_val, p_val = stats.f_oneway(
    grouped_test2.get_group('fwd')['price'],
    grouped_test2.get_group('rwd')['price']
)

# Print the results of the ANOVA test comparing 'fwd' and 'rwd' drive-wheels␣
↪groups
print("ANOVA results for 'fwd' vs 'rwd' drive-wheels and 'price': F =", f_val,␣
↪", P =", p_val)
# F-test score: Indicates the ratio of variance between 'fwd' and 'rwd' groups␣
↪to variance within these groups.
# P-value: Shows the probability that the observed differences between 'fwd'␣
↪and 'rwd' groups are due to chance.

# Perform ANOVA to compare the means of 'price' across '4wd' and 'rwd'␣
↪drive-wheels groups
f_val, p_val = stats.f_oneway(
    grouped_test2.get_group('4wd')['price'],
    grouped_test2.get_group('rwd')['price']
)

# Print the results of the ANOVA test comparing '4wd' and 'rwd' drive-wheels␣
↪groups
print("ANOVA results for '4wd' vs 'rwd' drive-wheels and 'price': F =", f_val,␣
↪", P =", p_val)
# F-test score: Indicates the ratio of variance between '4wd' and 'rwd' groups␣
↪to variance within these groups.
# P-value: Shows the probability that the observed differences between '4wd'␣
↪and 'rwd' groups are due to chance.

# Perform ANOVA to compare the means of 'price' across '4wd' and 'fwd'␣
↪drive-wheels groups
f_val, p_val = stats.f_oneway(
    grouped_test2.get_group('4wd')['price'],
    grouped_test2.get_group('fwd')['price']
)

# Print the results of the ANOVA test comparing '4wd' and 'fwd' drive-wheels␣
↪groups
print("ANOVA results for '4wd' vs 'fwd' drive-wheels and 'price': F =", f_val,␣
↪", P =", p_val)
```

```
# F-test score: Indicates the ratio of variance between '4wd' and 'fwd' groups␣
↪to variance within these groups.
# P-value: Shows the probability that the observed differences between '4wd'␣
↪and 'fwd' groups are due to chance.
```

```
ANOVA results for 'drive-wheels' and 'price': F= 67.95406500780399 , P =
3.3945443577151245e-23
ANOVA results for 'fwd' vs 'rwd' drive-wheels and 'price': F = 130.5533160959111
, P = 2.2355306355677845e-23
ANOVA results for '4wd' vs 'rwd' drive-wheels and 'price': F = 8.580681368924756
, P = 0.004411492211225333
ANOVA results for '4wd' vs 'fwd' drive-wheels and 'price': F = 0.665465750252303
, P = 0.41620116697845666
```

[54]:
```
# Import the LinearRegression class from sklearn
from sklearn.linear_model import LinearRegression

# Create an instance of the LinearRegression model
lm = LinearRegression()

# Display the created instance
lm
```

[54]:
```
LinearRegression()
```

[55]:
```
# Select 'highway-mpg' as the predictor variable (independent variable) and␣
↪assign it to X
X = df[['highway-mpg']]

# Select 'price' as the response variable (dependent variable) and assign it to␣
↪Y
Y = df['price']

# Fit the linear regression model using the selected predictor (X) and response␣
↪(Y)
lm.fit(X, Y)

# Use the fitted model to predict the price (Yhat) based on the values in X
Yhat = lm.predict(X)

# Display the first 5 predicted values
print("The first predicted values are:", Yhat[0:5])

# Get the intercept (a) of the linear regression line
intercept = lm.intercept_
print("The intercept is: a =", intercept)
```

```
# Get the coefficient (b) of the linear regression line, indicating the slope
coef = lm.coef_[0]
print("The coefficient is: b =", coef)

# Display the linear equation based on the calculated intercept and coefficient
print(f'The linear equation is: Y = {intercept:.2f} + {coef:.2f}X')
```

```
The first predicted values are: [16236.50464347 16236.50464347 17058.23802179
13771.3045085
 20345.17153508]
The intercept is: a = 38423.305858157386
The coefficient is: b = -821.733378321925
The linear equation is: Y = 38423.31 + -821.73X
```

[56]:

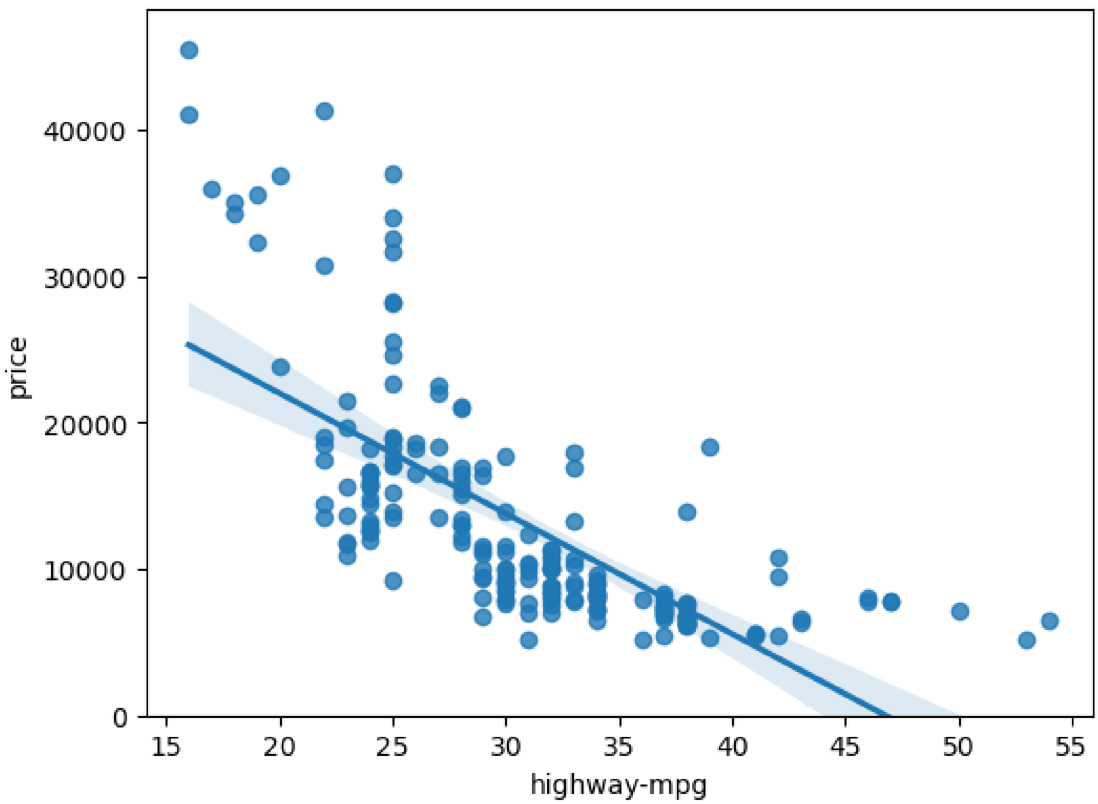
```
# Define the predictor variables (independent variables) for the Multiple␣
↪Linear Regression model
# We are using 'highway-mpg', 'engine-size', 'horsepower', and 'curb-weight' as␣
↪predictors
Z = df[['highway-mpg', 'engine-size', 'horsepower', 'curb-weight']]

# Fit the Multiple Linear Regression model using the predictor variables (Z)␣
↪and the response variable ('price')
lm.fit(Z, df['price'])

# Predict the values of 'price' using the fitted model
Yhat = lm.predict(Z)

# Display the first 5 predicted values
print("The first predicted values are: ", Yhat[:5])

# Display the coefficients for each predictor variable
print("The coefficients for each predictor are: ", lm.coef_)

# Display the intercept of the Multiple Linear Regression model
print("The intercept is: a =", lm.intercept_)

# Display the linear equation of the Multiple Linear Regression model
# Note: The coefficients and intercept will be used to construct the equation
# For example, the equation format is: Y = a + b1*X1 + b2*X2 + ... + bn*Xn
coefficients = lm.coef_
intercept = lm.intercept_
equation = f"Y = {intercept:.2f} + " + " + ".join([f"{coef:.2f}*{feature}" for␣
↪coef, feature in zip(coefficients, Z.columns)])
print("The linear equation is: ", equation)
```

```
The first predicted values are:  [13699.07700462 13699.07700462 19052.71346719
10620.61524404
 15520.90025344]
```

```
The coefficients for each predictor are:  [36.1593925  81.51280006 53.53022809
4.70805253]
The intercept is: a = -15811.86376772925
The linear equation is:  Y = -15811.86 + 36.16*highway-mpg + 81.51*engine-size +
53.53*horsepower + 4.71*curb-weight
```

[57]:
```
# Generate a regression plot with 'highway-mpg' on the x-axis and 'price' on
 ↪the y-axis using the seaborn library
sns.regplot(x="highway-mpg", y="price", data=df)

# Set the lower limit for the y-axis to 0, ensuring the plot starts from 0 on
 ↪the y-axis
plt.ylim(0,)
```

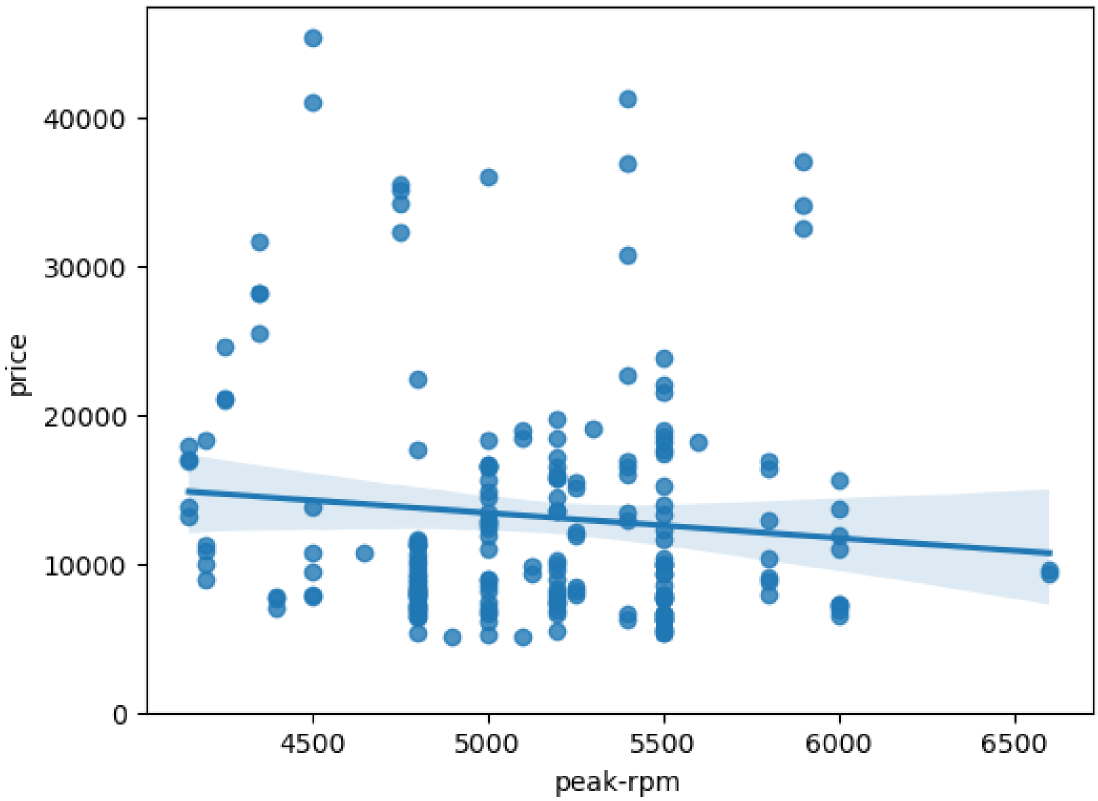
[57]:
```
(0.0, 48170.09339993266)
```



[58]:
```
# Create a regression plot using Seaborn to visualize the relationship between
 ↪'peak-rpm' and 'price'
sns.regplot(x="peak-rpm", y="price", data=df)

# Set the y-axis limit to start from 0 to ensure the plot starts at the origin
```

```
plt.ylim(0,)
```

[58]: (0.0, 47414.1)



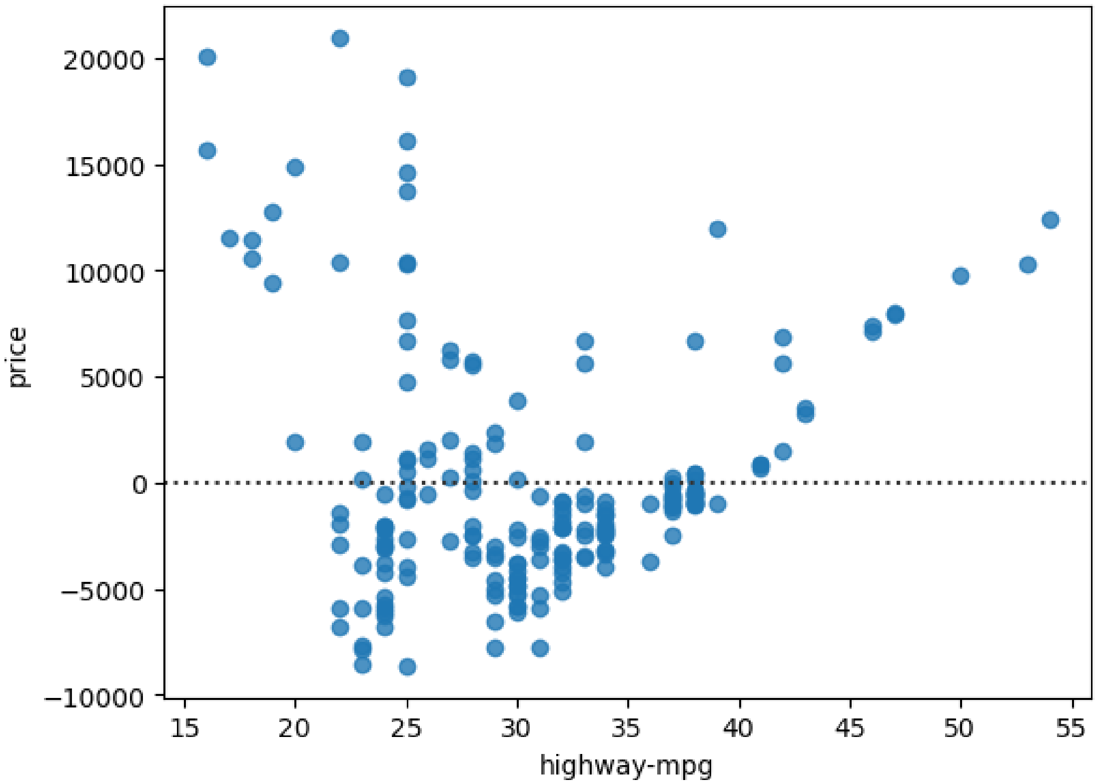
[59]:
```
# Calculate the correlation matrix for the selected columns: 'peak-rpm',␣
↪'highway-mpg', and 'price'
# This will show the correlation coefficients between each pair of variables in␣
↪the dataframe.
df[["peak-rpm", "highway-mpg", "price"]].corr()
```

[59]:

	peak-rpm	highway-mpg	price
peak-rpm	1.000000	-0.058598	-0.101616
highway-mpg	-0.058598	1.000000	-0.704692
price	-0.101616	-0.704692	1.000000

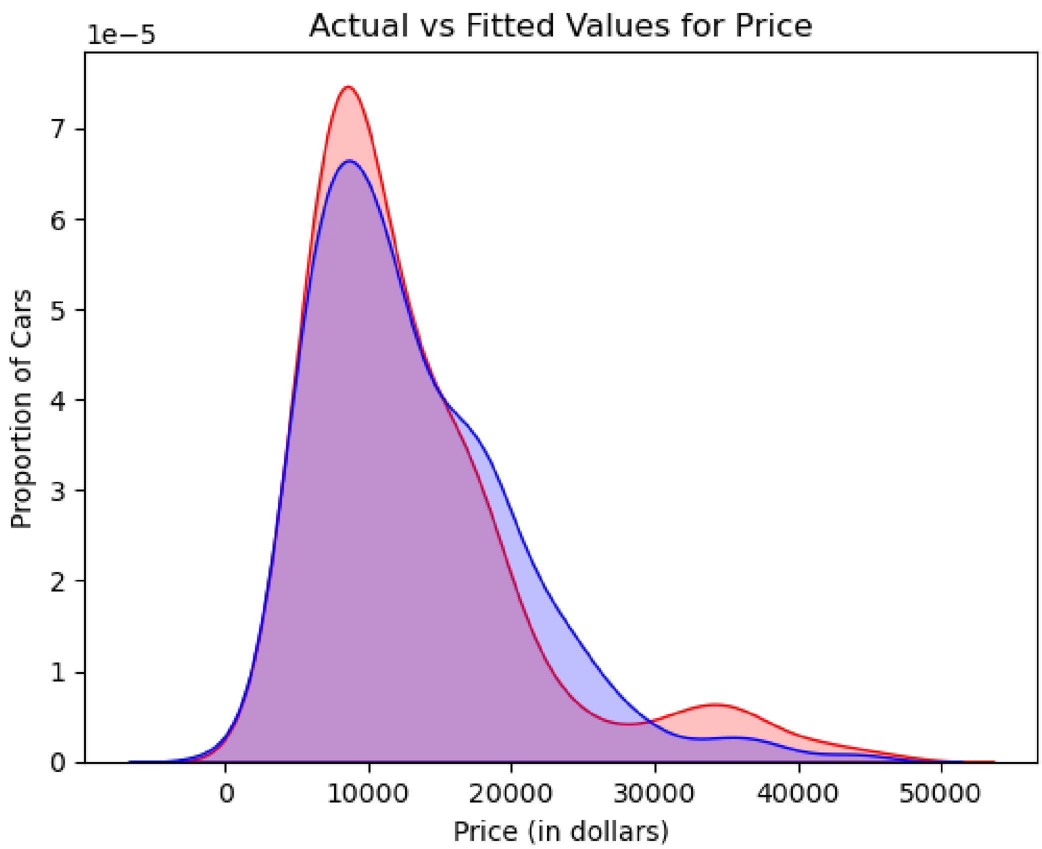
[60]:
```
# Generate a residual plot to visualize the residuals of the regression model
# Residuals are the differences between observed and predicted values, plotted␣
↪against 'highway-mpg'
# This plot helps evaluate the model's fit and identify patterns or issues.

sns.residplot(x=df['highway-mpg'], y=df['price'])
```

```
plt.show()
```



[61]:
```
# Predicts the car prices using the linear model and stores the results in␣
↪Y_hat.
Y_hat = lm.predict(Z)

# Creates a kernel density estimate plot comparing the actual car prices (in␣
↪red) with the fitted values predicted by the model (in blue).
ax1 = sns.kdeplot(df['price'], color="r", label="Actual Value", fill=True)
sns.kdeplot(Y_hat, color="b", label="Fitted Values", ax=ax1, fill=True)

# Sets the title and labels for the plot, displaying the comparison of actual␣
↪vs fitted prices.
plt.title('Actual vs Fitted Values for Price')
plt.xlabel('Price (in dollars)')
plt.ylabel('Proportion of Cars')

# Displays the plot.
plt.show()

# Closes the plot to free up resources.
```

```
plt.close()
```



[62]:
```
def PlotPolly(model, independent_variable, dependent_variable, Name):
    # Generate a range of new values for the independent variable to evaluate␣
 ↪the polynomial model.
    x_new = np.linspace(15, 55, 100)
    # Compute the corresponding predicted values using the polynomial model.
    y_new = model(x_new)

    # Plot the original data points as dots.
    plt.plot(independent_variable, dependent_variable, '.', label='Actual Data')
    # Plot the polynomial fit as a line.
    plt.plot(x_new, y_new, '-', label='Polynomial Fit')
    # Set the title of the plot.
    plt.title('Polynomial Fit with Matplotlib for Price ~ Length')
    # Get the current axes of the plot.
    ax = plt.gca()
    # Set the background color of the plot.
    ax.set_facecolor((0.898, 0.898, 0.898))
```

```
    # Get the current figure.
    fig = plt.gcf()
    # Label the x-axis with the name of the independent variable.
    plt.xlabel(Name)
    # Label the y-axis with 'Price of Cars'.
    plt.ylabel('Price of Cars')
    # Display the plot.
    plt.show()
    # Close the plot to free up resources.
    plt.close()
```

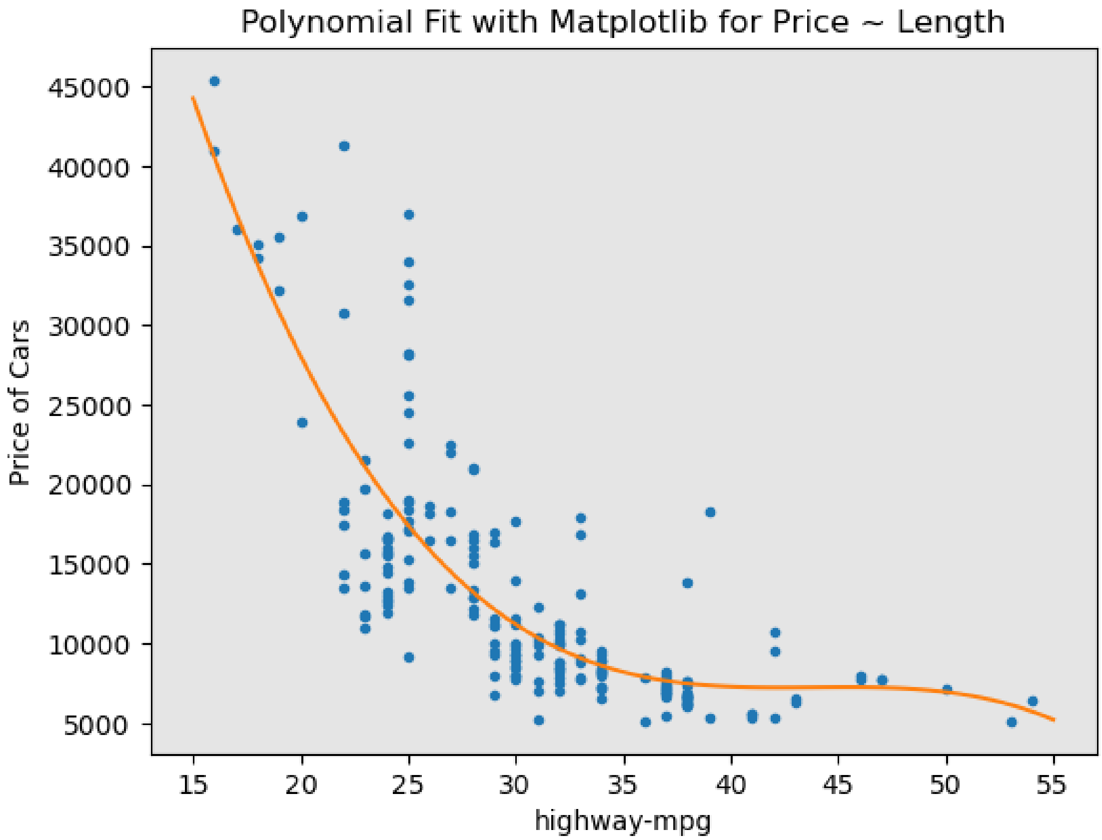
[63]:

```
# Extract the 'highway-mpg' and 'price' columns from the DataFrame into␣
↪separate variables.
x = df['highway-mpg']  # Independent variable: highway-mpg
y = df['price']        # Dependent variable: price

# Fit a 3rd-order polynomial (cubic) to the data.
f = np.polyfit(x, y, 3)  # np.polyfit fits a polynomial of the specified order␣
↪(3rd order) to the data.

# Create a polynomial function from the coefficients obtained.
p = np.poly1d(f)  # np.poly1d generates a polynomial function from the␣
↪coefficients.

# Extract coefficients from the polynomial function
b3, b2, b1, a = f

# Print the polynomial equation with labeled coefficients.
print(f"Polynomial equation: ")
print("")
print(f"Price = {a:.2f} + {b1:.2f} * (highway-mpg) + {b2:.2f} * (highway-mpg)^2␣
↪+ {b3:.2f} * (highway-mpg)^3  ")
```

```
Polynomial equation:

Price = 137923.59 + -8965.43 * (highway-mpg) + 204.75 * (highway-mpg)^2 + -1.56
* (highway-mpg)^3
```

[64]:

```
PlotPolly(p, x, y, 'highway-mpg')
```



[65]:

```
# Import the mean_squared_error function from sklearn.metrics to calculate the␣
↪Mean Squared Error (MSE).
from sklearn.metrics import mean_squared_error

# Fit the linear regression model to the data (X as independent variable and Y␣
↪as dependent variable).
lm.fit(X, Y)

# Calculate and print the R-squared value of the model, which indicates the␣
↪proportion of variance explained by the model.
print('The R-square is: ', lm.score(X, Y))

# Predict the dependent variable values (Yhat) using the fitted model.
Yhat = lm.predict(X)

# Print the first four predicted values to observe the model's output.
print('The output of the first four predicted value is: ', Yhat[0:4])

# Calculate the Mean Squared Error (MSE) between the actual values␣
↪(df['price']) and the predicted values (Yhat).
```

```
mse = mean_squared_error(df['price'], Yhat)

# Print the Mean Squared Error, which provides an indication of the average␣
↪squared difference between actual and predicted values.
print('The mean square error of price and predicted value is: ', mse)
```

```
The R-square is:  0.4965911884339175
The output of the first four predicted value is:  [16236.50464347 16236.50464347
17058.23802179 13771.3045085 ]
The mean square error of price and predicted value is:  31635042.944639895
```

[66]:
```
# Fit the model for Multiple Linear Regression
lm.fit(Z, df['price'])
# The model is being fitted using Multiple Linear Regression, where Z␣
↪represents the independent variables and df['price'] is the dependent␣
↪variable. This trains the regression model to understand the relationship␣
↪between multiple predictors and car prices.

# Find the R^2
print('The R-square is: ', lm.score(Z, df['price']))
# The R-squared value of the Multiple Linear Regression model is computed and␣
↪printed. It measures how well the model explains the variability in car␣
↪prices based on the independent variables in Z. A higher R-squared indicates␣
↪a better fit.

# Predict the prices using the fitted Multiple Linear Regression model
Y_predict_multifit = lm.predict(Z)
# The fitted Multiple Linear Regression model is used to predict car prices␣
↪based on the independent variables in Z. The predicted values are stored in␣
↪Y_predict_multifit.

# Compute and print the Mean Squared Error (MSE) between the actual prices and␣
↪the predicted values
print('The mean square error of price and predicted value using multifit is: ',␣
↪\
      mean_squared_error(df['price'], Y_predict_multifit))
```

```
The R-square is:  0.8093732522175299
The mean square error of price and predicted value using multifit is:
11979300.349818885
```

[67]:
```
from sklearn.metrics import r2_score, mean_squared_error

# Calculate the R-squared value for the polynomial fit
r_squared = r2_score(df['price'], p(x))

# Print the R-squared value, which provides an indication of the goodness of␣
↪fit for the polynomial model.
```

```
print('The R-square value is: ', r_squared)


# Calculate the Mean Squared Error (MSE) for the polynomial fit
mse = mean_squared_error(df['price'], p(x))

print('The Mean Squared Error is: ', mse)
```

```
The R-square value is:  0.674194666390652
The Mean Squared Error is:  20474146.426361207
```

[68]:
```
def DistributionPlot(RedFunction, BlueFunction, RedName, BlueName, Title):

    # Plot the KDE for the red function with its label
    ax1 = sns.kdeplot(RedFunction, color="r", label=RedName)

    # Plot the KDE for the blue function with its label
    ax2 = sns.kdeplot(BlueFunction, color="b", label=BlueName, ax=ax1)

    # Add title and labels to the plot
    plt.title(Title)
    plt.xlabel('Price (in dollars)')
    plt.ylabel('Proportion of Cars')

    # Add a legend to the plot
    plt.legend()

    # Display the plot
    plt.show()
    plt.close()
```

[69]:
```
def PollyPlot(xtrain, xtest, y_train, y_test, lr, poly_transform):

    # xtrain, xtest: Training and testing data for the independent variable
    # y_train, y_test: Training and testing data for the dependent variable
    # lr: Linear regression object that has been trained
    # poly_transform: Polynomial transformation object used for transforming␣
 ↪the input data

    # Determine the range of x values by finding the minimum and maximum of␣
 ↪both training and testing sets
    xmax = max([xtrain.values.max(), xtest.values.max()])
    xmin = min([xtrain.values.min(), xtest.values.min()])

    # Create a range of values from xmin to xmax with a step size of 0.1
    x = np.arange(xmin, xmax, 0.1)
```

```python
    # Plot the training data as red dots
    plt.plot(xtrain, y_train, 'ro', label='Training Data')

    # Plot the testing data as green dots
    plt.plot(xtest, y_test, 'go', label='Test Data')

    # Plot the predicted polynomial function line using the polynomial␣
↪transformation on the x values
    plt.plot(x, lr.predict(poly_transform.fit_transform(x.reshape(-1, 1))),␣
↪label='Predicted Function')

    # Set the limits for the y-axis to range from -10000 to 60000
    plt.ylim([-10000, 60000])

    # Set the label for the y-axis
    plt.ylabel('Price')

    # Display the legend to identify the different data series
    plt.legend()
```

[70]:

```python
# Extract only the numeric columns from the DataFrame and assign them to a new␣
↪DataFrame 'df1'
df1 = df._get_numeric_data()

# Display the first few rows of the 'df1' DataFrame to inspect the numeric data
df1.head()
```

[70]:

```
   symboling  normalized-losses  wheel-base    length     width    height  \
0          3                122        88.6  0.811148  0.890278  0.816054
1          3                122        88.6  0.811148  0.890278  0.816054
2          1                122        94.5  0.822681  0.909722  0.876254
3          2                164        99.8  0.848630  0.919444  0.908027
4          2                164        99.4  0.848630  0.922222  0.908027

   curb-weight  engine-size  bore  stroke  …  peak-rpm  city-mpg  \
0         2548          130  3.47    2.68  …    5000.0        21
1         2548          130  3.47    2.68  …    5000.0        21
2         2823          152  2.68    3.47  …    5000.0        19
3         2337          109  3.19    3.40  …    5500.0        24
4         2824          136  3.19    3.40  …    5500.0        18

   highway-mpg    price  city-L/100km  highway-L/100km  fuel-type-diesel  \
0           27  13495.0     11.190476         8.703704             False
1           27  16500.0     11.190476         8.703704             False
2           26  16500.0     12.368421         9.038462             False
3           30  13950.0      9.791667         7.833333             False
4           22  17450.0     13.055556        10.681818             False
```

```
   fuel-type-gas  aspiration-std  aspiration-turbo
0           True            True             False
1           True            True             False
2           True            True             False
3           True            True             False
4           True            True             False

[5 rows x 22 columns]
```

[71]:

```python
from sklearn.model_selection import train_test_split

# The target variable is 'price' from the dataset.
y_data = df1['price']

# All other numeric columns except 'price' are used as input features.
x_data = df1.drop('price', axis=1)

# Split the dataset into training and testing sets.
# 10% of the data is used for testing, and the random_state is set for␣
↪reproducibility.
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.
↪10, random_state=1)

# Print the number of samples in the test set.
print("Number of test samples:", x_test.shape[0])

# Print the number of samples in the training set.
print("Number of training samples:", x_train.shape[0])

# Create a LinearRegression object.
lre = LinearRegression()

# Fit the model using 'horsepower' as the predictor and the training data.
lre.fit(x_train[['horsepower']], y_train)

# Calculate and print the R-squared value for the test set.
r_squared_test = lre.score(x_test[['horsepower']], y_test)
print("R-squared value for the test set with 10% split:", r_squared_test)

# Calculate and print the R-squared value for the training set.
r_squared_train = lre.score(x_train[['horsepower']], y_train)
print("R-squared value for the training set with 10% split:", r_squared_train)
```

```
Number of test samples: 21
Number of training samples: 180
R-squared value for the test set with 10% split: 0.3635480624962414
R-squared value for the training set with 10% split: 0.662028747521533
```

[72]:
```
# Split the data into training and test sets with 40% of the data reserved for␣
↪testing.
x_train1, x_test1, y_train1, y_test1 = train_test_split(x_data, y_data,␣
↪test_size=0.4, random_state=0)

# Print the number of samples in the training and test sets.
print("Number of test samples:", x_test1.shape[0])
print("Number of training samples:", x_train1.shape[0])

# Create a LinearRegression object.
lre = LinearRegression()

# Fit the model using 'horsepower' as the predictor with the new training data.
lre.fit(x_train1[['horsepower']], y_train1)

# Evaluate the model by calculating the R-squared value for the test set.
test_r_squared = lre.score(x_test1[['horsepower']], y_test1)
print("R-squared value for the test set with 40% split:", test_r_squared)

# Evaluate the model by calculating the R-squared value for the training set.
train_r_squared = lre.score(x_train1[['horsepower']], y_train1)
print("R-squared value for the training set with 40% split:", train_r_squared)
```

```
Number of test samples: 81
Number of training samples: 120
R-squared value for the test set with 40% split: 0.7139737368233015
R-squared value for the training set with 40% split: 0.5754853866574969
```

[73]:
```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_val_predict

# Perform cross-validation to evaluate the model using the 'horsepower' feature.
# This calculates the R-squared scores for each fold in the cross-validation.
Rcross = cross_val_score(lre, x_data[['horsepower']], y_data, cv=4)
print("The mean of the folds are", Rcross.mean(), "and the standard deviation␣
↪is" , Rcross.std())

# Calculate the negative Mean Squared Error (MSE) for each fold in the␣
↪cross-validation.
# The negative sign is used because cross_val_score treats higher scores as␣
↪better,
# and MSE should be minimized.
neg_mse = -1 * cross_val_score(lre, x_data[['horsepower']], y_data, cv=4,␣
↪scoring='neg_mean_squared_error')
print("The mean of the negative MSEs is", neg_mse.mean(), "and the standard␣
↪deviation is", neg_mse.std())
```

```
# Perform cross-validation to compute R-squared scores using 2-fold␣
↪cross-validation.
Rc = cross_val_score(lre, x_data[['horsepower']], y_data, cv=2)
print("The mean R-squared score with 2-fold cross-validation is", Rc.mean())

# Predict the target values using cross-validation and return the predictions.
# This helps to evaluate how well the model performs across different data␣
↪folds.
yhat = cross_val_predict(lre, x_data[['horsepower']], y_data, cv=4)
print("The first five predicted values are:", yhat[0:5])
```

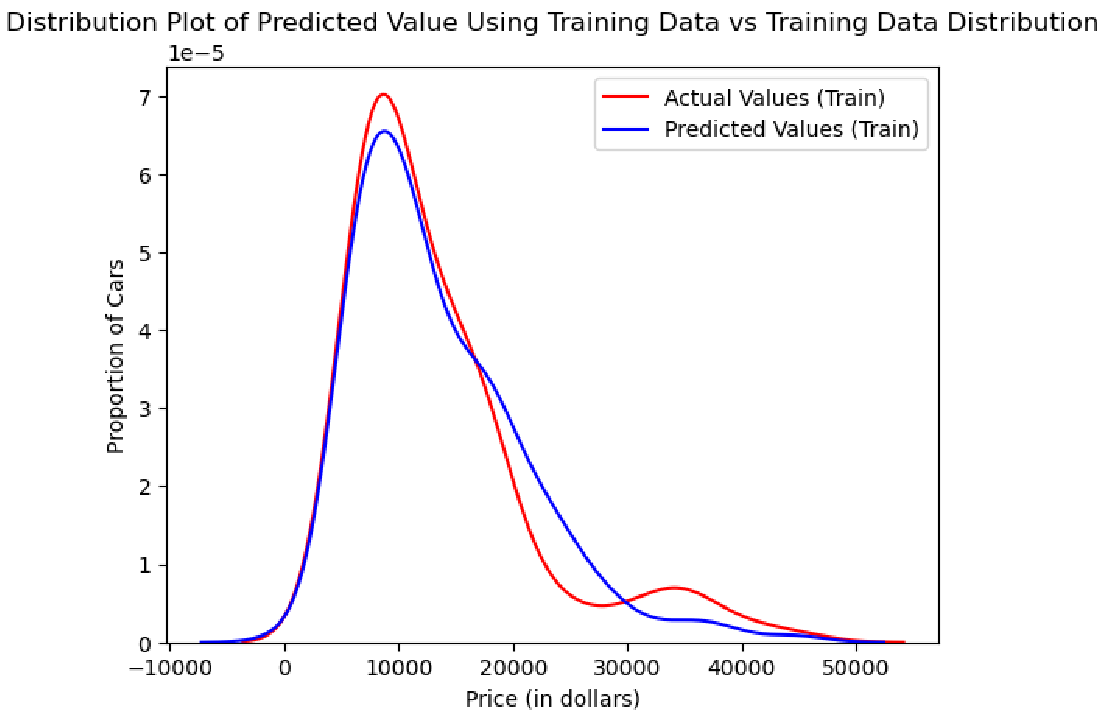
```
The mean of the folds are 0.5220592359225413 and the standard deviation is
0.29130480666118463
The mean of the negative MSEs is 23521246.47057339 and the standard deviation is
12000467.588458164
The mean R-squared score with 2-fold cross-validation is 0.516835099979672
The first five predicted values are: [14142.23793549 14142.23793549
20815.3029844  12745.549902
 14762.9881726 ]
```

[74]:

```
# Create a LinearRegression object.
lr = LinearRegression()

# Fit the model using 'horsepower', 'curb-weight', 'engine-size', and␣
↪'highway-mpg' as predictors.
lr.fit(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],␣
↪y_train)

# Predict the target values using the training data.
yhat_train = lr.predict(x_train[['horsepower', 'curb-weight', 'engine-size',␣
↪'highway-mpg']])

# Print the first five predicted values for the training set.
print("First five predicted values for the training set:", yhat_train[0:5])

# Predict the target values using the test data.
yhat_test = lr.predict(x_test[['horsepower', 'curb-weight', 'engine-size',␣
↪'highway-mpg']])

# Print the first five predicted values for the test set.
print("First five predicted values for the test set:", yhat_test[0:5])
```

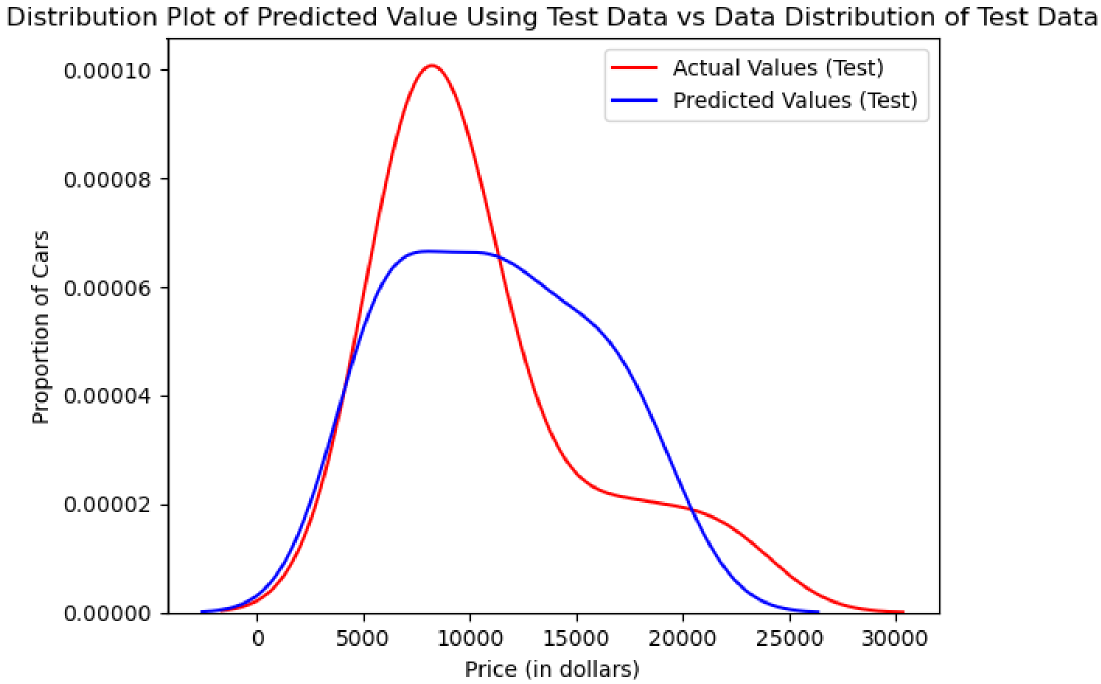
```
First five predicted values for the training set: [ 7426.34910902 28324.42490838
14212.74872339  4052.80810192
 34499.8541269 ]
First five predicted values for the test set: [11349.68099115  5884.25292475
11208.31007475  6641.03017109
 15565.98722248]
```

[75]:
```
# Set the title for the distribution plot.
Title = 'Distribution Plot of Predicted Value Using Training Data vs Training␣
↪Data Distribution'

# Generate the distribution plot.
# 'y_train' holds the actual training values.
# 'yhat_train' holds the predicted values from the training data.
# The labels "Actual Values (Train)" and "Predicted Values (Train)" will appear␣
↪in the plot legend.
# 'Title' sets the title of the plot.
DistributionPlot(y_train, yhat_train, "Actual Values (Train)", "Predicted␣
↪Values (Train)", Title)
```



[76]:
```
# Set the title for the distribution plot.
Title = 'Distribution Plot of Predicted Value Using Test Data vs Data␣
↪Distribution of Test Data'

# Generate the distribution plot.
# 'y_test' contains the actual test values.
# 'yhat_test' contains the predicted values from the test data.
# The labels "Actual Values (Test)" and "Predicted Values (Test)" will appear␣
↪in the plot legend.
# 'Title' sets the title of the plot.
```

```
DistributionPlot(y_test, yhat_test, "Actual Values (Test)", "Predicted Values␣
↪(Test)", Title)
```

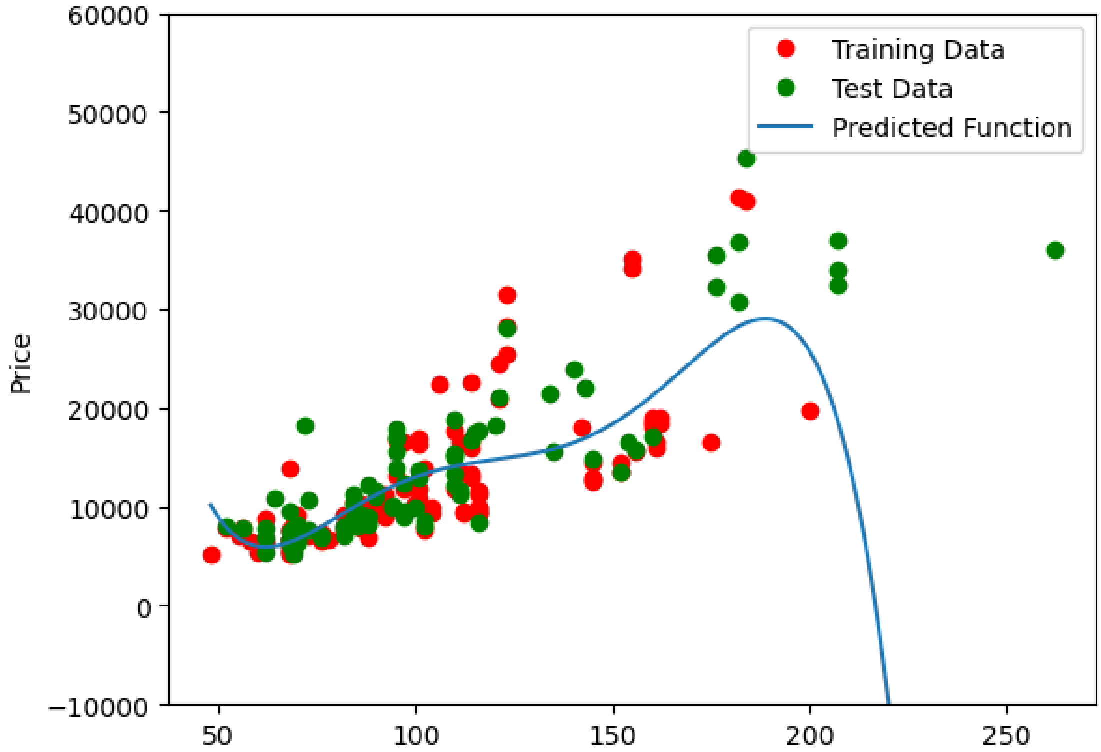


[77]:
```
from sklearn.preprocessing import PolynomialFeatures

# Split the data into training and test sets, with 55% of the data used for␣
↪training and 45% for testing.
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.
↪45, random_state=0)

# Create a polynomial feature transformer with a degree of 5.
pr = PolynomialFeatures(degree=5)

# Transform the 'horsepower' feature in both training and test sets to␣
↪polynomial features.
x_train_pr = pr.fit_transform(x_train[['horsepower']])
x_test_pr = pr.fit_transform(x_test[['horsepower']])

# Initialize and train a linear regression model using the polynomial features.
poly = LinearRegression()
poly.fit(x_train_pr, y_train)

# Predict the target values for the test set using the trained model.
yhat = poly.predict(x_test_pr)
```

```
# Print the first five predicted values.
print("Predicted values:", yhat[0:5])

# Print the first four predicted values and compare them with the actual target␣
↪values.
print("Predicted values (first 4):", yhat[0:4])
print("True values (first 4):", y_test[0:4].values)
```

```
Predicted values: [ 6727.50134739  7306.62980155 12213.64942948 18895.18190498
 19997.01014697]
Predicted values (first 4): [ 6727.50134739  7306.62980155 12213.64942948
18895.18190498]
True values (first 4): [ 6295. 10698. 13860. 13499.]
```

[78]:
```
# Visualize the polynomial regression results using the PollyPlot function.
# This function displays the training data, test data, and the predicted␣
↪polynomial function.
# x_train['horsepower'] and x_test['horsepower']: Input features for training␣
↪and test sets.
# y_train and y_test: Actual target values for training and test sets.
# poly: The trained polynomial regression model.
# pr: The polynomial feature transformer.

PollyPlot(x_train['horsepower'], x_test['horsepower'], y_train, y_test, poly,␣
↪pr)
```



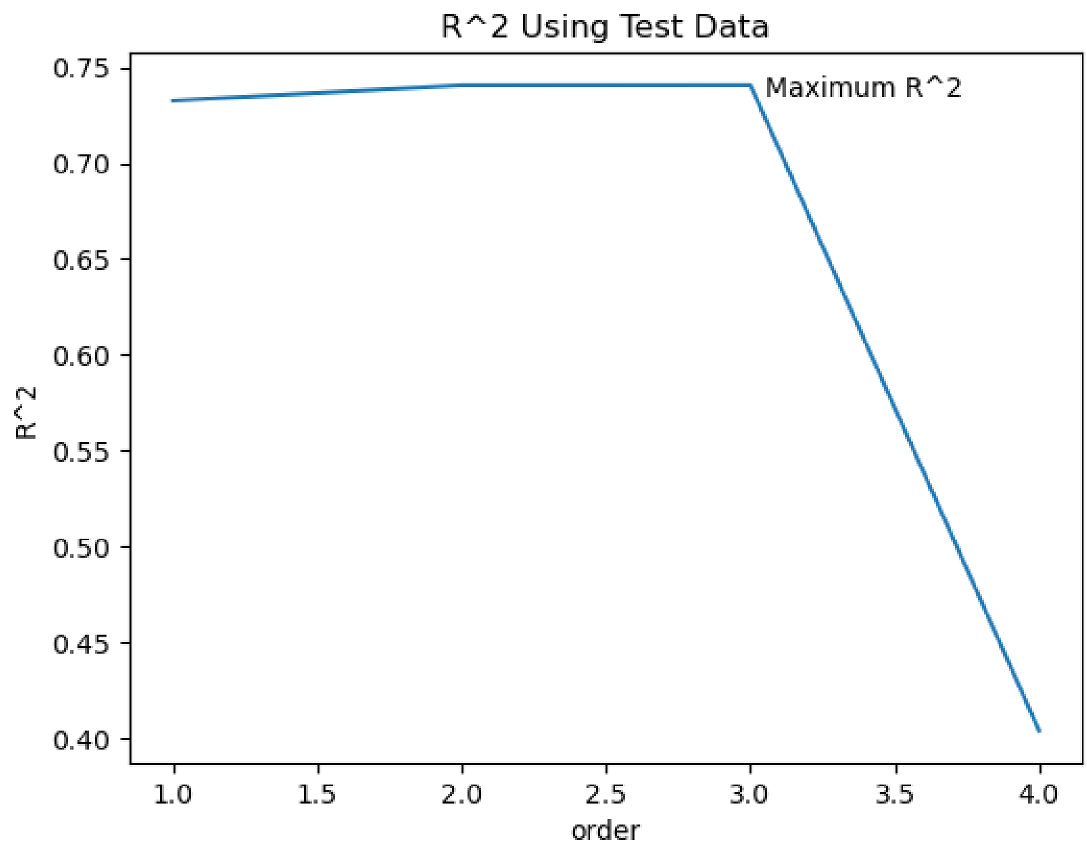
[79]:
```
# Calculate R² score for the training data
train_r2 = poly.score(x_train_pr, y_train)
print("R² of the training data:", train_r2)

# Calculate R² score for the test data
test_r2 = poly.score(x_test_pr, y_test)
print("")
print("R² of the test data:", test_r2)
```

```
R² of the training data: 0.5568527852117562

R² of the test data: -29.815108072386607
```

[80]:
```
# Initialize an empty list to store R² scores for different polynomial degrees
Rsqu_test = []

# Define the range of polynomial orders to test
order = [1, 2, 3, 4]

# Loop through each polynomial order
for n in order:
    # Create a PolynomialFeatures object for the current degree
    pr = PolynomialFeatures(degree=n)
```

```python
    # Transform the training and testing data using the polynomial features
    x_train_pr = pr.fit_transform(x_train[['horsepower']])
    x_test_pr = pr.fit_transform(x_test[['horsepower']])

    # Create and train the Linear Regression model
    lr = LinearRegression()
    lr.fit(x_train_pr, y_train)

    # Calculate the R² score on the test data and append it to the list
    Rsqu_test.append(lr.score(x_test_pr, y_test))

# Plot R² scores against polynomial orders
plt.plot(order, Rsqu_test)

# Label the x-axis as 'order'
plt.xlabel('order')

# Label the y-axis as 'R^2'
plt.ylabel('R^2')

# Set the title of the plot
plt.title('R^2 Using Test Data')

# Annotate the plot with a text label indicating the maximum R² value
plt.text(3.05, 0.735, 'Maximum R^2 ')
```

[80]:
```
Text(3.05, 0.735, 'Maximum R^2 ')
```



[81]:
```
def f(order, test_data):
    # Split the data into training and testing sets
    x_train, x_test, y_train, y_test = train_test_split(x_data, y_data,␣
 ↪test_size=test_data, random_state=0)

    # Create a PolynomialFeatures object with the specified degree
    pr = PolynomialFeatures(degree=order)

    # Transform the training and testing features using polynomial features
    x_train_pr = pr.fit_transform(x_train[['horsepower']])
    x_test_pr = pr.fit_transform(x_test[['horsepower']])

    # Initialize and fit a LinearRegression model using the transformed␣
 ↪training features
    poly = LinearRegression()
    poly.fit(x_train_pr, y_train)

    # Plot the results using the PollyPlot function, showing training data,␣
 ↪testing data, and the model's predictions
```

```
    PollyPlot(x_train['horsepower'], x_test['horsepower'], y_train, y_test,␣
↪poly, pr)
```

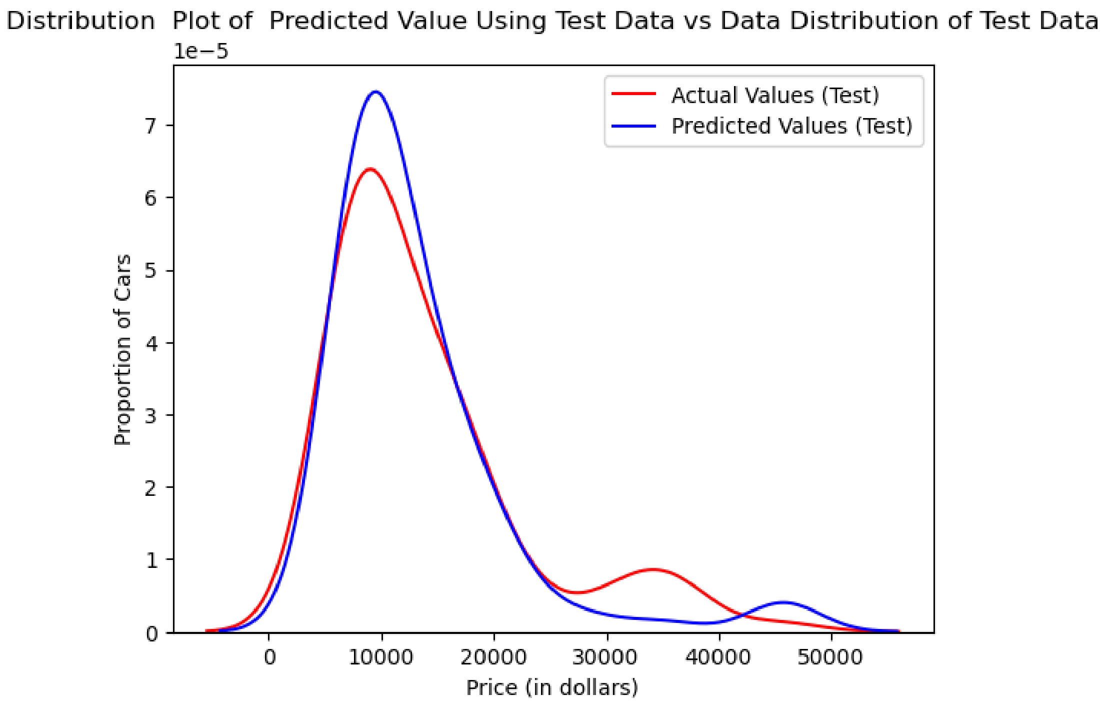
[82]:
```
from ipywidgets import interact

# Create interactive widgets to adjust the parameters of the function "f".
interact(f, order=(0, 6, 1), test_data=(0.05, 0.95, 0.05))
```

```
interactive(children=(IntSlider(value=3, description='order', max=6),␣
↪FloatSlider(value=0.45, description='tes…
```

[82]:
```
<function __main__.f(order, test_data)>
```

[83]:
```
# Create a PolynomialFeatures object with a degree of 2 for polynomial␣
↪transformation
pr1 = PolynomialFeatures(degree=2)

# Transform the training features 'horsepower', 'curb-weight', 'engine-size',␣
↪and 'highway-mpg' into polynomial features
x_train_pr1 = pr1.fit_transform(x_train[['horsepower', 'curb-weight',␣
↪'engine-size', 'highway-mpg']])

# Transform the testing features 'horsepower', 'curb-weight', 'engine-size',␣
↪and 'highway-mpg' into polynomial features
x_test_pr1 = pr1.fit_transform(x_test[['horsepower', 'curb-weight',␣
↪'engine-size', 'highway-mpg']])

# Output the shape of the transformed training data to see the number of␣
↪features after the polynomial transformation
x_train_pr1.shape

# Fit a LinearRegression model using the transformed polynomial features from␣
↪the training data
poly1 = LinearRegression().fit(x_train_pr1, y_train)
```

[84]:
```
yhat_test1=poly1.predict(x_test_pr1)

Title='Distribution  Plot of  Predicted Value Using Test Data vs Data␣
↪Distribution of Test Data'

DistributionPlot(y_test, yhat_test1, "Actual Values (Test)", "Predicted Values␣
↪(Test)", Title)
```



[85]:
```
from sklearn.linear_model import Ridge

# Create a PolynomialFeatures object with a degree of 2 for polynomial␣
↪transformation
pr = PolynomialFeatures(degree=2)

# Transform the training features into polynomial features
x_train_pr = pr.fit_transform(x_train[['horsepower', 'curb-weight',␣
↪'engine-size', 'highway-mpg', 'normalized-losses', 'symboling']])

# Transform the testing features into polynomial features
x_test_pr = pr.fit_transform(x_test[['horsepower', 'curb-weight',␣
↪'engine-size', 'highway-mpg', 'normalized-losses', 'symboling']])

# Initialize a Ridge regression model with an alpha value of 1 (controls␣
↪regularization strength)
RigeModel = Ridge(alpha=1)

# Fit the Ridge regression model using the transformed polynomial features from␣
↪the training data
RigeModel.fit(x_train_pr, y_train)

# Predict the target values using the trained Ridge model on the test data
```

```
yhat = RigeModel.predict(x_test_pr)

# Print the first four predicted values and the corresponding actual test␣
↪values for comparison
print('predicted:', yhat[0:4])
print('test set :', y_test[0:4].values)
```

```
predicted: [ 6572.19586866  9634.40697747 20948.17104272 19403.38016094]
test set : [ 6295. 10698. 13860. 13499.]
```

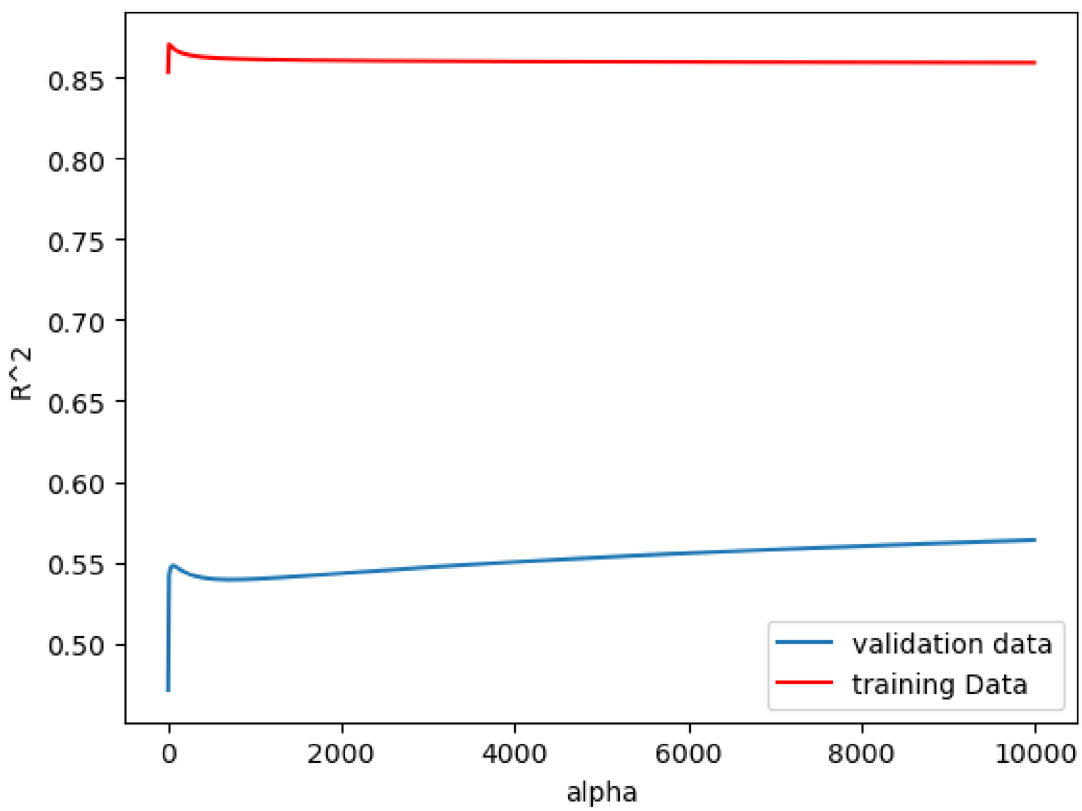
[86]:
```
from tqdm import tqdm

# Initialize empty lists to store R^2 scores for the test and training sets
Rsqu_test = []
Rsqu_train = []

# Create an empty list 'dummy1' (potentially for later use, though it's not␣
↪used here)
dummy1 = []

# Generate an array of alpha values, scaled by a factor of 10, ranging from 0␣
↪to 9990
Alpha = 10 * np.array(range(0,1000))

# Initialize a progress bar to visualize the loop's progress over the Alpha␣
↪values
pbar = tqdm(Alpha)

# Iterate over each alpha value in the Alpha array
for alpha in pbar:
    # Initialize a Ridge regression model with the current alpha value
    RigeModel = Ridge(alpha=alpha)

    # Fit the Ridge regression model using the training polynomial features
    RigeModel.fit(x_train_pr, y_train)

    # Calculate the R^2 scores for the test and training data
    test_score, train_score = RigeModel.score(x_test_pr, y_test), RigeModel.
↪score(x_train_pr, y_train)

    # Update the progress bar with the current test and training R^2 scores
    pbar.set_postfix({"Test Score": test_score, "Train Score": train_score})

    # Append the current R^2 scores to their respective lists
    Rsqu_test.append(test_score)
    Rsqu_train.append(train_score)
```

```
100%|      | 1000/1000 [00:06<00:00, 162.43it/s, Test Score=0.564, Train
```

```
Score=0.859]
```

[87]:
```
# Plot the R^2 scores for the test (validation) data against the alpha values
plt.plot(Alpha, Rsqu_test, label='validation data')

# Plot the R^2 scores for the training data against the alpha values, using a␣
↪red line
plt.plot(Alpha, Rsqu_train, 'r', label='training Data')

# Label the x-axis with 'alpha' to indicate that it represents the␣
↪regularization strength
plt.xlabel('alpha')

# Label the y-axis with 'R^2' to indicate that the y-axis represents the R^2␣
↪score
plt.ylabel('R^2')

# Display a legend to differentiate between the lines representing the test and␣
↪training data
plt.legend()
```

[87]:
```
<matplotlib.legend.Legend at 0x25f890ee650>
```



[88]:
```python
from sklearn.model_selection import GridSearchCV

# Define the parameter grid for alpha values to search
parameters1 = [{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000]}]

# Initialize a Ridge regression model
RR = Ridge()

# Perform GridSearchCV to find the best alpha using 4-fold cross-validation
Grid1 = GridSearchCV(RR, parameters1, cv=4)

# Fit GridSearchCV with the training data to find the best model
Grid1.fit(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],␣
↪y_train)

# Retrieve the best model with the optimal alpha value
BestRR = Grid1.best_estimator_

# Evaluate the best model's performance on the test data
score = BestRR.score(x_test[['horsepower', 'curb-weight', 'engine-size',␣
↪'highway-mpg']], y_test)

# Print the score and explanation
print(f'The best model, optimized using Grid Search, achieved a R^2 score of␣
↪{score:.4f} on the test data.')
print('This R^2 score indicates how well the model predicts the car prices on␣
↪unseen test data.')
print('A higher R^2 score signifies a better fit of the model to the test data,␣
↪with 1 being a perfect fit.')
```

```
The best model, optimized using Grid Search, achieved a R^2 score of 0.7723 on
the test data.
This R^2 score indicates how well the model predicts the car prices on unseen
test data.
A higher R^2 score signifies a better fit of the model to the test data, with 1
being a perfect fit.
```