Basic Data Analysis Tutorial

August 31, 2023

1 Basic Data Analysis Tutorial

[4]:
```
# Install the libraries
# !pip install skillsnetwork
# !pip install matplotlib
# !pip install numpy
# !pip install pandas
# !pip install seaborn
```

[57]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import skillsnetwork
import warnings
```

When engaged in data analysis, having a clear understanding of your objective is essential.

Let's analyze the 'price'!

Import the dataset

[4]:
```
path = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/
↪IBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/auto.csv"
df = pd.read_csv(path, header = None)
```

2 DATA EXPLORATION

[5]:
```
df.head(5)
```

[5]:
```
   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]
```

Visualize the first 5 rows of the dataframe

[6]:
```python
df.describe()
```

[6]:
```
                0           9          10          11          12  \
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

                13          16          20          23          24
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
```

The describe() function provides the summary statistics of the dataset.

(Only contains numerical attributes)

[7]:
```python
df.describe(include = 'all')
```

[7]:
```
                 0    1       2    3    4     5      6    7      8  \
count   205.000000  205     205  205  205   205    205  205    205
unique         NaN   52      22    2    2     3      5    3      2
top            NaN    ?  toyota  gas  std  four  sedan  fwd  front
freq           NaN   41      32  185  168   114     96  120    202
mean      0.834146  NaN     NaN  NaN  NaN   NaN    NaN  NaN    NaN
std       1.245307  NaN     NaN  NaN  NaN   NaN    NaN  NaN    NaN
min      -2.000000  NaN     NaN  NaN  NaN   NaN    NaN  NaN    NaN
25%       0.000000  NaN     NaN  NaN  NaN   NaN    NaN  NaN    NaN
```

```
50%       1.000000  NaN      NaN  NaN  NaN   NaN    NaN  NaN    NaN
75%       2.000000  NaN      NaN  NaN  NaN   NaN    NaN  NaN    NaN
max       3.000000  NaN      NaN  NaN  NaN   NaN    NaN  NaN    NaN

                 9  …         16    17    18    19          20  21    22  \
count   205.000000  …  205.000000  205   205   205  205.000000 205   205
unique         NaN  …         NaN    8    39    37         NaN  60    24
top            NaN  …         NaN mpfi  3.62  3.40         NaN  68  5500
freq           NaN  …         NaN   94    23    20         NaN  19    37
mean     98.756585  …  126.907317  NaN   NaN   NaN   10.142537 NaN   NaN
std       6.021776  …   41.642693  NaN   NaN   NaN    3.972040 NaN   NaN
min      86.600000  …   61.000000  NaN   NaN   NaN    7.000000 NaN   NaN
25%      94.500000  …   97.000000  NaN   NaN   NaN    8.600000 NaN   NaN
50%      97.000000  …  120.000000  NaN   NaN   NaN    9.000000 NaN   NaN
75%     102.400000  …  141.000000  NaN   NaN   NaN    9.400000 NaN   NaN
max     120.900000  …  326.000000  NaN   NaN   NaN   23.000000 NaN   NaN

                23          24   25
count   205.000000  205.000000  205
unique         NaN         NaN  187
top            NaN         NaN    ?
freq           NaN         NaN    4
mean     25.219512   30.751220  NaN
std       6.542142    6.886443  NaN
min      13.000000   16.000000  NaN
25%      19.000000   25.000000  NaN
50%      24.000000   30.000000  NaN
75%      30.000000   34.000000  NaN
max      49.000000   54.000000  NaN

[11 rows x 26 columns]
```

By using the command describe(include='all'), it provides summary statistics for all attributes, including: - unique: indicating how many different unique 'names' or 'elements' are within each categorical attribute. - top: representing the element that appeared most frequently among the categorical attributes. - freq: denoting the frequency or the number of times it appeared.

[8]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 26 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   0       205 non-null    int64
 1   1       205 non-null    object
 2   2       205 non-null    object
 3   3       205 non-null    object
```

```
 4   4       205 non-null    object
 5   5       205 non-null    object
 6   6       205 non-null    object
 7   7       205 non-null    object
 8   8       205 non-null    object
 9   9       205 non-null    float64
 10  10      205 non-null    float64
 11  11      205 non-null    float64
 12  12      205 non-null    float64
 13  13      205 non-null    int64
 14  14      205 non-null    object
 15  15      205 non-null    object
 16  16      205 non-null    int64
 17  17      205 non-null    object
 18  18      205 non-null    object
 19  19      205 non-null    object
 20  20      205 non-null    float64
 21  21      205 non-null    object
 22  22      205 non-null    object
 23  23      205 non-null    int64
 24  24      205 non-null    int64
 25  25      205 non-null    object
dtypes: float64(5), int64(5), object(16)
memory usage: 41.8+ KB
```

The info() function provides us with the data type of each attribute, as well as helps in identifying null values.

[9]:
```
df.dtypes
```

[9]:
```
0       int64
1      object
2      object
3      object
4      object
5      object
6      object
7      object
8      object
9     float64
10    float64
11    float64
12    float64
13      int64
14     object
15     object
16      int64
17     object
```

```
18     object
19     object
20    float64
21     object
22     object
23      int64
24      int64
25     object
dtype: object
```

Alternatively, we can use dtype.

3 We noticed that there are serveral things we need to fix

1.) We noticed that this dataframe does not have a header

2.) We noticed there are missing values in the dataset

3.) We noticed that the some data types do not match

Let's fix them!

[10]:
```
df.replace('?', np.nan, inplace = True)
```

Replacing all '?' to NaN to make it easier

4 1.) Adding Header

[11]:
```
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"]
df.columns = headers
df.head()
```

[11]:
```
   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]
```

5 2.) Dealing With Missing Values

[12]:
```python
missing_data = df.isnull()

# This forloop check the number of missing values for each attribute:

# for column in missing_data.columns.values.tolist():
#     print(column)
#     print(missing_data[column].value_counts())
#     print('')
```

- "normalized-losses": 41 missing data
- "num-of-doors": 2 missing data
- "bore": 4 missing data
- "stroke" : 4 missing data
- "horsepower": 2 missing data
- "peak-rpm": 2 missing data
- "price": 4 missing data

Let's deal with the missing values

There are three most common ways to deal with missing values: * 1.) Drop the missing values * 2.) Replace the missing values * 3.) Leave the missing values

[13]:
```python
# Drop the missing values in the subset price
df.dropna(subset = ['price'], inplace = True)
```

[14]:
```
# Calculate the averages
avg_normalized_losses = df['normalized-losses'].astype('float').mean()
avg_bore = df['bore'].astype('float').mean()
avg_stroke = df['stroke'].astype('float').mean()
avg_peak_rpm = df['peak-rpm'].astype('float').mean()
avg_horsepower = df['horsepower'].astype('float').mean()

# Replace the missing value with the averages calculated above
df['normalized-losses'].replace(np.nan, avg_normalized_losses, inplace =True)
df['bore'].replace(np.nan, avg_bore, inplace= True)
df['stroke'].replace(np.nan, avg_stroke, inplace= True)
df['peak-rpm'].replace(np.nan, avg_peak_rpm, inplace= True)
df['horsepower'].replace(np.nan, avg_horsepower, inplace= True)
```

[15]:
```
# For categorical attributes, we can see how many units of each variable we have
print(df['num-of-doors'].value_counts())
```

```
four    113
two      86
Name: num-of-doors, dtype: int64
```

[16]:
```
# 'four' is the most common, so let's replace the missing values with it
# However, value_counts() only works on pandas series
# As a result, we only include one bracket df['drive-wheels'], not two brackets␣
 ↪df[['drive-wheels']].
df['num-of-doors'].replace(np.nan, 'four', inplace= True)
```

6 3.) Fixing the Data Type

- *'normalized-losses' and 'horsepower' should be of type int*

- *'bore', 'stroke', 'peak-rpm' and 'price' should be of type float*

[17]:
```
# Change to type int
df[['normalized-losses', 'horsepower']] = df[['normalized-losses',␣
 ↪'horsepower']].astype('int')

# Change to type float
df[['bore', 'stroke', 'peak-rpm', 'price']] = df[['bore', 'stroke', 'peak-rpm',␣
 ↪'price']].astype('float')
```

[18]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 201 entries, 0 to 204
Data columns (total 26 columns):
 #   Column             Non-Null Count  Dtype
```

```
---  ------             --------------  -----
 0   symboling          201 non-null    int64
 1   normalized-losses  201 non-null    int64
 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    float64
 19  stroke             201 non-null    float64
 20  compression-ratio  201 non-null    float64
 21  horsepower         201 non-null    int64
 22  peak-rpm           201 non-null    float64
 23  city-mpg           201 non-null    int64
 24  highway-mpg        201 non-null    int64
 25  price              201 non-null    float64
dtypes: float64(9), int64(7), object(10)
memory usage: 42.4+ KB
```

7 Data Transformation

Sometimes dataframe has a different format than what we need

Hence we need to transform data into a common format

[19]:
```
# df.head()
```

[20]:
```
# Let's transform mpg into L/100km!
# L/100km = 235/mpg
df['city-mpg'] = 235/df['city-mpg']
df['highway-mpg'] = 235/df['highway-mpg']

# Don't forget to rename the column name after the transformation!
df.rename(columns={'city-mpg':'city-L/100km', 'highway-mpg' : 'highway-L/
↪100km'}, inplace = True)
```

```
df[['city-L/100km','highway-L/100km']] ## select and display 'city-L/100km' and␣
↪'highway-L/100km'
```

[20]:

```
     city-L/100km  highway-L/100km
0       11.190476         8.703704
1       11.190476         8.703704
2       12.368421         9.038462
3        9.791667         7.833333
4       13.055556        10.681818
..            …                …
200     10.217391         8.392857
201     12.368421         9.400000
202     13.055556        10.217391
203      9.038462         8.703704
204     12.368421         9.400000

[201 rows x 2 columns]
```

8 Data Normalization

It is important to normalize our dataframe

Think about this scenario, we have a data that contain an age attribute and a income attribute

[21]:

```
data1 = {'Age' : [21,38,25,41],
       'Income' : [5000,5500,3000,7700]}
ex1 = pd.DataFrame(data1)
ex1
```

[21]:

	Age	Income
0	21	5000
1	38	5500
2	25	3000
3	41	7700

Income has a bigger effect on the analysis than age because it is larger, we need to make them have equal effects by normalizing them.

There are different ways you can perform normalization such as z-score

However, let's keep it easy and perform value/max_value

[22]:

```
# Let's go back to our dataframe - df and perform this operation on the␣
↪'length', 'width' and 'height'

df['length'] = df['length']/df['length'].max()
df['width'] = df['width']/df['width'].max()
df['height'] = df['height']/df['height'].max()
```

```
df[['length','width','height']].head(6)
```

[22]:

	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
5	0.851994	0.920833	0.887960

9 Data Binning

Why binning?

We perform data binning to transform countinuous numerical variables into discrete categorical 'bins' for better data analysis.

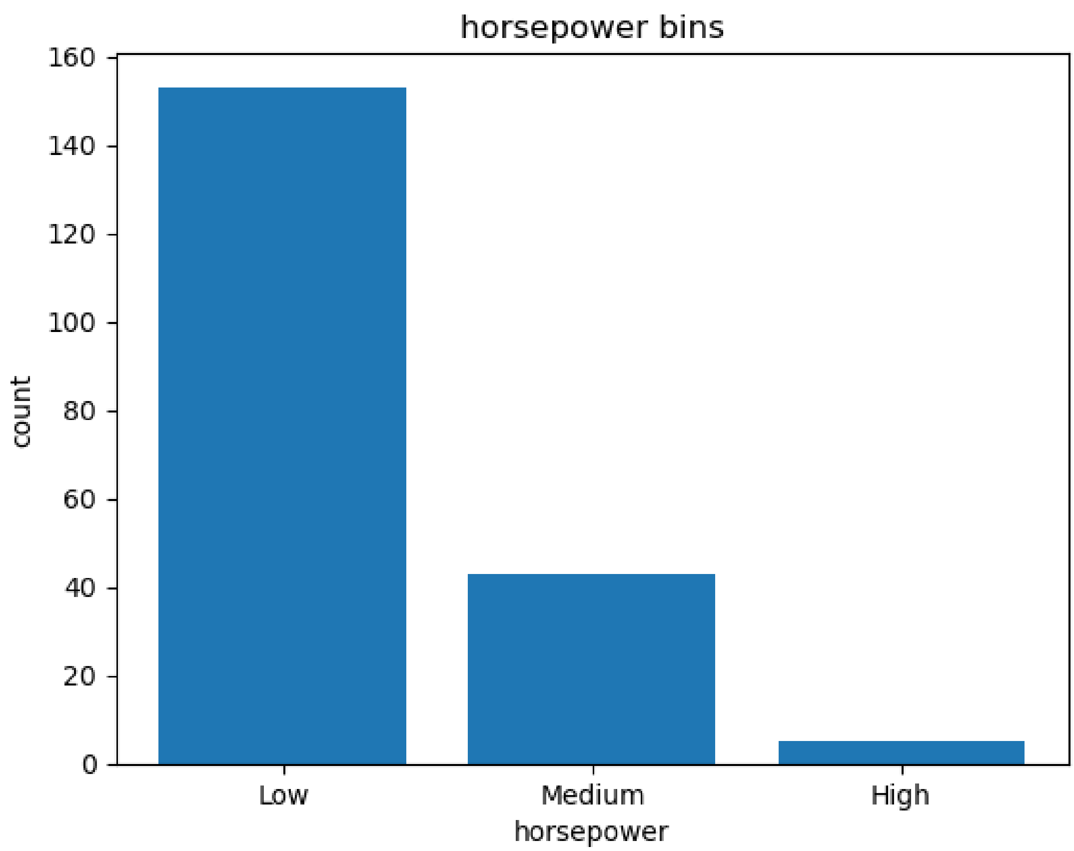
[23]:
```
# First, we need the matplotlib library to help with data visualization
import matplotlib as plt
from matplotlib import pyplot

# Let's divide horsepower into 3 bins
# linspace(start_value, end_value, numbers_generated)
bins = np.linspace(min(df['horsepower']), max(df['horsepower']),4)

# Assign group names
group_names = ['Low', 'Medium', 'High']

# Apply the cut function to determine what each value of df['horsepower']␣
↪belongs to
df['horsepower-binned'] = pd.cut(df['horsepower'], bins, labels = group_names,␣
↪include_lowest = True)
df[['horsepower','horsepower-binned']].head(10)
```

[23]:

	horsepower	horsepower-binned
0	111	Low
1	111	Low
2	154	Medium
3	102	Low
4	115	Low
5	110	Low
6	110	Low
7	110	Low
8	140	Medium
10	101	Low

[24]:
```python
# Visualize the 'bin'
pyplot.bar(group_names, df["horsepower-binned"].value_counts())
plt.pyplot.xlabel("horsepower")
plt.pyplot.ylabel("count")
plt.pyplot.title("horsepower bins")
```

[24]:
```
Text(0.5, 1.0, 'horsepower bins')
```



Side note: by using the Altair library, visualization can be enhanced

10 Dummy Variables

We assign dummy variables to label categories. It is called 'dummies' becasue the numbers themselves don't have inherent meanings.

[25]:
```python
# Let's assign dummy variables for 'fuel-type' which has two distinct values:
↪'gas' and 'diesel'
dummy_variable = pd.get_dummies(df['fuel-type'])

# Change the column name
```

```
dummy_variable.rename(columns = {'diesel' : 'fuel-type-diesel', 'gas' :␣
↪'fuel-type-gas'}, inplace = True)
```

[26]:
```
dummy_variable.head()
```

[26]:

	fuel-type-diesel	fuel-type-gas
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1

[27]:
```
# Merge the dataframe df and dummy_variable together
df = pd.concat([df,dummy_variable], axis = 1)

# Drop the original 'fuel-type' from the dataframe 'df'
df.drop('fuel-type', axis = 1, inplace = True)
```

11 Data Correlation

Correlation is extremely important when it comes to data analysis, it tells us whether the variables are interdependent.

If a variable have a strong correlation with another, it is an indicator that the variable is a good predictor.

[28]:
```
# To see the correlations, simply use the function corr()
df.corr()
```

```
/tmp/ipykernel_450/2902987488.py:2: FutureWarning: The default value of
numeric_only in DataFrame.corr is deprecated. In a future version, it will
default to False. Select only valid columns or specify the value of numeric_only
to silence this warning.
  df.corr()
```

[28]:

	symboling	normalized-losses	wheel-base	length \
symboling	1.000000	0.466264	-0.535987	-0.365404
normalized-losses	0.466264	1.000000	-0.056661	0.019424
wheel-base	-0.535987	-0.056661	1.000000	0.876024
length	-0.365404	0.019424	0.876024	1.000000
width	-0.242423	0.086802	0.814507	0.857170
height	-0.550160	-0.373737	0.590742	0.492063
curb-weight	-0.233118	0.099404	0.782097	0.880665
engine-size	-0.110581	0.112360	0.572027	0.685025
bore	-0.139896	-0.029800	0.493203	0.608941
stroke	-0.007992	0.055127	0.157964	0.123913
compression-ratio	-0.182196	-0.114713	0.250313	0.159733

horsepower	0.075776	0.217300	0.371297	0.579688
peak-rpm	0.279719	0.239544	-0.360233	-0.286035
city-L/100km	0.066171	0.238567	0.476153	0.657373
highway-L/100km	-0.029807	0.181189	0.577576	0.707108
price	-0.082391	0.133999	0.584642	0.690628
fuel-type-diesel	-0.196735	-0.101546	0.307237	0.211187
fuel-type-gas	0.196735	0.101546	-0.307237	-0.211187

	width	height	curb-weight	engine-size	bore	\
symboling	-0.242423	-0.550160	-0.233118	-0.110581	-0.139896	
normalized-losses	0.086802	-0.373737	0.099404	0.112360	-0.029800	
wheel-base	0.814507	0.590742	0.782097	0.572027	0.493203	
length	0.857170	0.492063	0.880665	0.685025	0.608941	
width	1.000000	0.306002	0.866201	0.729436	0.544879	
height	0.306002	1.000000	0.307581	0.074694	0.180327	
curb-weight	0.866201	0.307581	1.000000	0.849072	0.644041	
engine-size	0.729436	0.074694	0.849072	1.000000	0.572516	
bore	0.544879	0.180327	0.644041	0.572516	1.000000	
stroke	0.188814	-0.060822	0.167412	0.205806	-0.055390	
compression-ratio	0.189867	0.259737	0.156433	0.028889	0.001250	
horsepower	0.614972	-0.086901	0.758001	0.822636	0.566786	
peak-rpm	-0.245852	-0.309913	-0.279350	-0.256753	-0.267338	
city-L/100km	0.673363	0.003811	0.785353	0.745059	0.554726	
highway-L/100km	0.736728	0.084301	0.836921	0.783465	0.559197	
price	0.751265	0.135486	0.834415	0.872335	0.543154	
fuel-type-diesel	0.244356	0.281578	0.221046	0.070779	0.054435	
fuel-type-gas	-0.244356	-0.281578	-0.221046	-0.070779	-0.054435	

	stroke	compression-ratio	horsepower	peak-rpm	\
symboling	-0.007992	-0.182196	0.075776	0.279719	
normalized-losses	0.055127	-0.114713	0.217300	0.239544	
wheel-base	0.157964	0.250313	0.371297	-0.360233	
length	0.123913	0.159733	0.579688	-0.286035	
width	0.188814	0.189867	0.614972	-0.245852	
height	-0.060822	0.259737	-0.086901	-0.309913	
curb-weight	0.167412	0.156433	0.758001	-0.279350	
engine-size	0.205806	0.028889	0.822636	-0.256753	
bore	-0.055390	0.001250	0.566786	-0.267338	
stroke	1.000000	0.187854	0.097598	-0.063720	
compression-ratio	0.187854	1.000000	-0.214392	-0.435721	
horsepower	0.097598	-0.214392	1.000000	0.107882	
peak-rpm	-0.063720	-0.435721	0.107882	1.000000	
city-L/100km	0.036285	-0.299372	0.889454	0.115813	
highway-L/100km	0.047199	-0.223361	0.840695	0.017736	
price	0.082267	0.071107	0.809729	-0.101542	
fuel-type-diesel	0.241033	0.985231	-0.168941	-0.475759	
fuel-type-gas	-0.241033	-0.985231	0.168941	0.475759	

	city-L/100km	highway-L/100km	price	fuel-type-diesel
symboling	0.066171	-0.029807	-0.082391	-0.196735
normalized-losses	0.238567	0.181189	0.133999	-0.101546
wheel-base	0.476153	0.577576	0.584642	0.307237
length	0.657373	0.707108	0.690628	0.211187
width	0.673363	0.736728	0.751265	0.244356
height	0.003811	0.084301	0.135486	0.281578
curb-weight	0.785353	0.836921	0.834415	0.221046
engine-size	0.745059	0.783465	0.872335	0.070779
bore	0.554726	0.559197	0.543154	0.054435
stroke	0.036285	0.047199	0.082267	0.241033
compression-ratio	-0.299372	-0.223361	0.071107	0.985231
horsepower	0.889454	0.840695	0.809729	-0.168941
peak-rpm	0.115813	0.017736	-0.101542	-0.475759
city-L/100km	1.000000	0.958306	0.789898	-0.241282
highway-L/100km	0.958306	1.000000	0.801118	-0.158091
price	0.789898	0.801118	1.000000	0.110326
fuel-type-diesel	-0.241282	-0.158091	0.110326	1.000000
fuel-type-gas	0.241282	0.158091	-0.110326	-1.000000

	fuel-type-gas
symboling	0.196735
normalized-losses	0.101546
wheel-base	-0.307237
length	-0.211187
width	-0.244356
height	-0.281578
curb-weight	-0.221046
engine-size	-0.070779
bore	-0.054435
stroke	-0.241033
compression-ratio	-0.985231
horsepower	0.168941
peak-rpm	0.475759
city-L/100km	0.241282
highway-L/100km	0.158091
price	-0.110326
fuel-type-diesel	-1.000000
fuel-type-gas	1.000000

[29]:
```
# Let's the explore the correlation between 'bore', 'stroke', 'peak-rpm' and
↪'horsepower'
df[['bore','stroke','peak-rpm','horsepower']].corr()
```

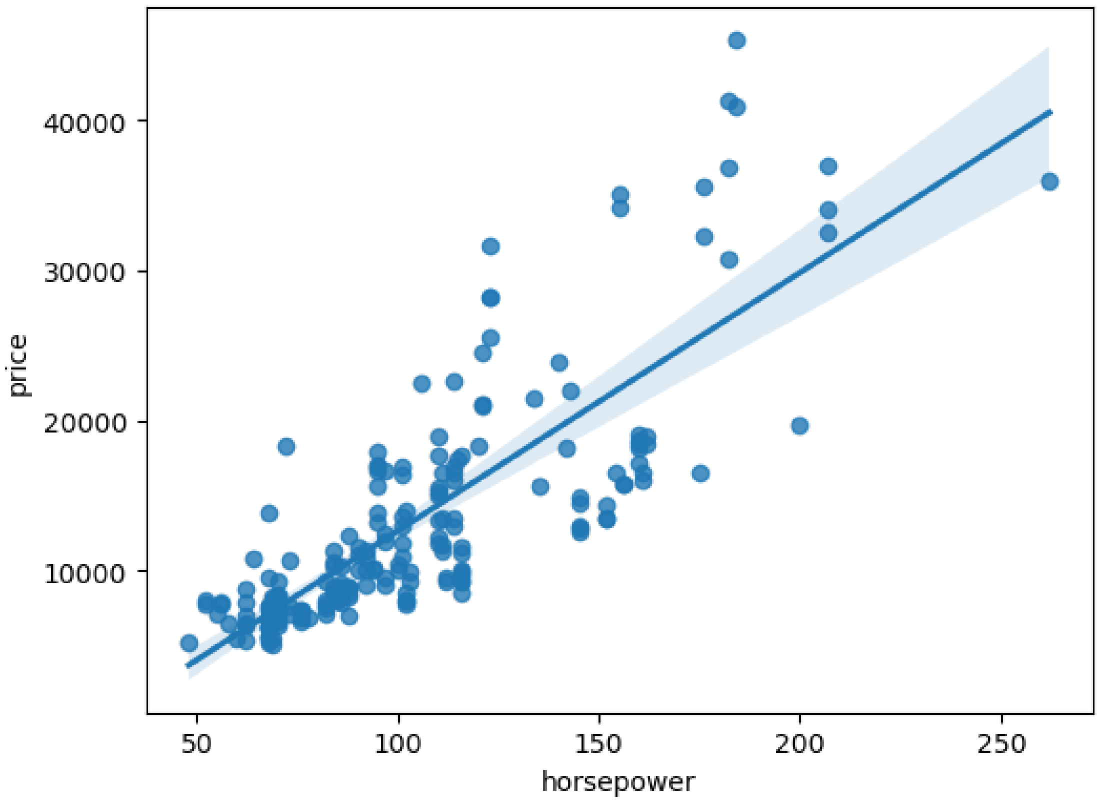
[29]:

	bore	stroke	peak-rpm	horsepower
bore	1.000000	-0.055390	-0.267338	0.566786

```
stroke      -0.055390  1.000000 -0.063720    0.097598
peak-rpm    -0.267338 -0.063720  1.000000    0.107882
horsepower   0.566786  0.097598  0.107882    1.000000
```

[30]:
```
# To see the correlation between 'horsepower' and 'price' visually

sns.regplot(x= 'horsepower', y = 'price', data= df)
```

[30]:
```
<Axes: xlabel='horsepower', ylabel='price'>
```



[31]:
```
df[['horsepower','price']].corr()
```

[31]:

	horsepower	price
horsepower	1.000000	0.809729
price	0.809729	1.000000

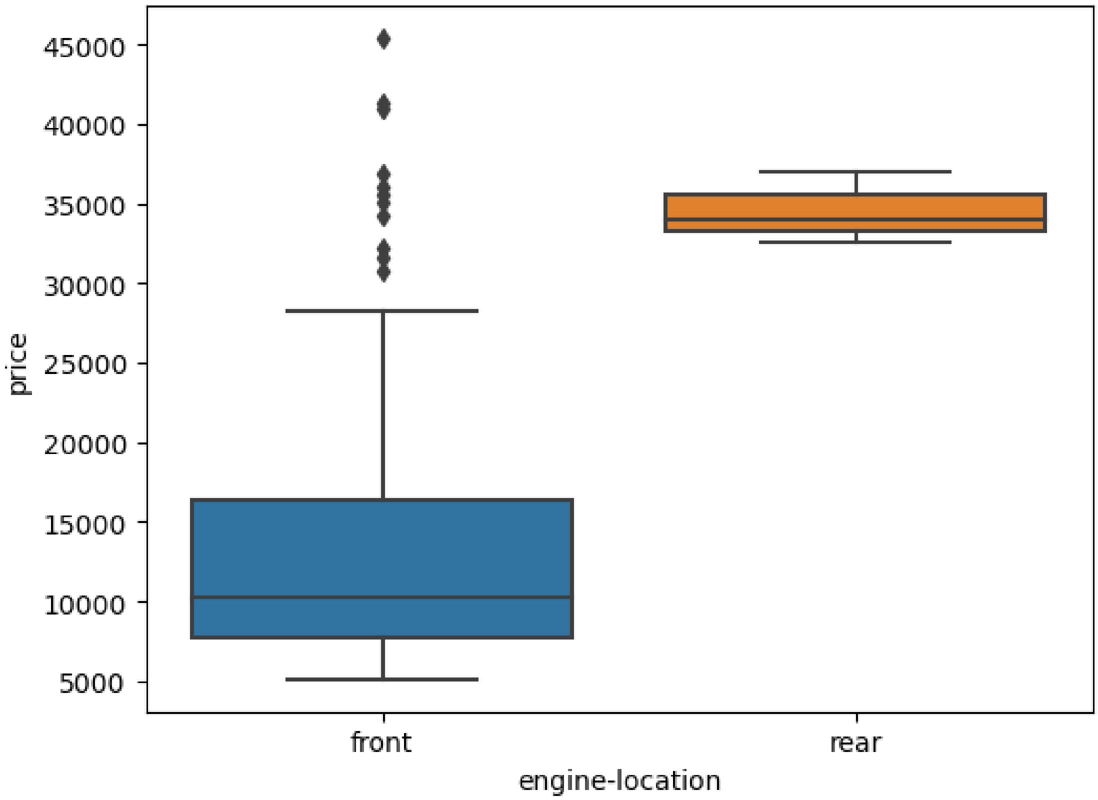
Above, we can see there is a positive correlation relationship betweenn the variable 'horsepower' and 'price'.

Also, with a correlation of 0.809729. It seems like a pretty good predictor.

Did you notice we are only dealing with continuous numerical variables? What about categorical variables?

[32]:
```
#To determine whether a categorical variable is a good indicator visually, we␣
↪can use boxplot.
sns.boxplot(x= 'engine-location', y= 'price', data= df)
```

[32]:
```
<Axes: xlabel='engine-location', ylabel='price'>
```



The distinct differences between the engine-location shows it might be a good indicator of price.

If they share significant overlap, it might not be a good indicator. However, this might be too general and deeper analysis is usually required.

[33]:
```
# Let's revisit the value_counts() function again for categorical variables.
# It counts the number of units and covert to a dataframe and change the column␣
↪name
drive_wheels_counts = df['drive-wheels'].value_counts().to_frame()
drive_wheels_counts.rename(columns={'drive-wheels': 'value_counts'},␣
↪inplace=True)
drive_wheels_counts
```

[33]:

	value_counts
fwd	118
rwd	75

```
4wd              8
```

[34]:
```
# Renme the index to 'drive-wheels'
drive_wheels_counts.index.name = 'drive-wheels'
drive_wheels_counts
```

[34]:
```
              value_counts
drive-wheels
fwd                    118
rwd                     75
4wd                      8
```

12 Data Grouping

We use the 'groupby' method to group the data by different categories.

Let's do an example and group by the variable 'drive-wheels'!

[35]:
```
# Select the coumns 'drive-wheels', 'body-style' and 'price'.
df_group_one = df[['drive-wheels','body-style','price']]

# Group and calculate the average price for each of the different categories
df_group_one = df_group_one.groupby(['drive-wheels','body-style'], as_index =␣
 ↪False).mean()
df_group_one
```

[35]:
```
   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
```

Pivot

We can make it into a pivot table so it is easier to visualize.

[36]:
```
grouped_pivot = df_group_one.pivot(index= 'drive-wheels', columns= 'body-style')
grouped_pivot
```

```
# Often we will encounter missing values, so let's fill it with 0
grouped_pivot.replace(np.nan, 0, inplace = True)
# OR
# grouped_pivot = grouped_pivot.fillna(0)
grouped_pivot
```

[36]:

```
                   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
```

13 Model - Linear Regression Model

The linear regression model help us understand the relationship between two variables, the dependent(Y) and the independent(X) variable.

Linear Function

$$Yhat = a + bX$$

- where 'a' is the intercept and 'b' is the slope

[38]:

```
from sklearn.linear_model import LinearRegression
```

Let's perform the linear regression model for engine size and price.

[39]:

```
lm = LinearRegression()
X = df[['engine-size']] #dataframe
Y = df['price'] #series

#fit the linear model
lm.fit(X,Y)
```

[39]:

```
LinearRegression()
```

[40]:

```
# Calculate the intercept
lm.intercept_
```

[40]:

```
-7963.338906281046
```

[41]:
```
# Calculate the slope
lm.coef_
```

[41]:
```
array([166.86001569])
```

[42]:
```
# <b>Therefore, we have:</b>

# $$
# Yhat = -7963.34 + 166.86*X
# $$
```

14 Model - Multiple Linear Regression Model

What if we want to examine the relationship between the dependent variable with more than one predictor variables?

$$Yhat = a + b\_1X\_1 + b\_2X\_2 + b\_3X\_3 + b\_4X\_4$$

[44]:
```
# Let's explore the relationships of 'Horsepower','Curb-weight','Engine-size'
# and 'Highway-L/100km' with 'price'
lm1 = LinearRegression()
Z = df[['horsepower','curb-weight','engine-size','highway-L/100km']]

# Fit the multiple linear model
lm1.fit(Z,Y)
```

[44]:
```
LinearRegression()
```

[45]:
```
lm1.intercept_
```

[45]:
```
-14382.161315163685
```

[46]:
```
lm1.coef_
```

[46]:
```
array([ 36.76149419,   3.50153554,  85.32658561, 498.91963877])
```

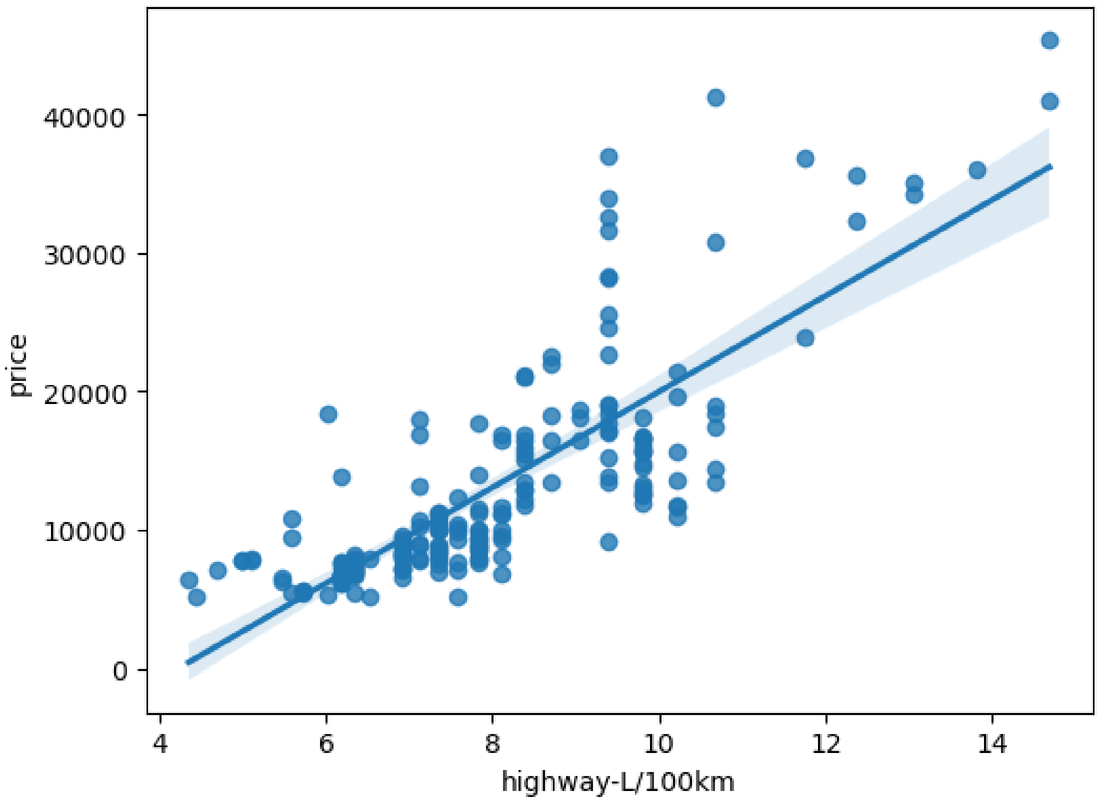
Hence,

Price = -14382.161315163685 + 36.76149419 x horsepower + 3.50153554 x curb-weight + 85.32658561 x engine-size + 498.91963877 x highway-L/100km

15 Model - Visualization (Simple Linear Regression Model)

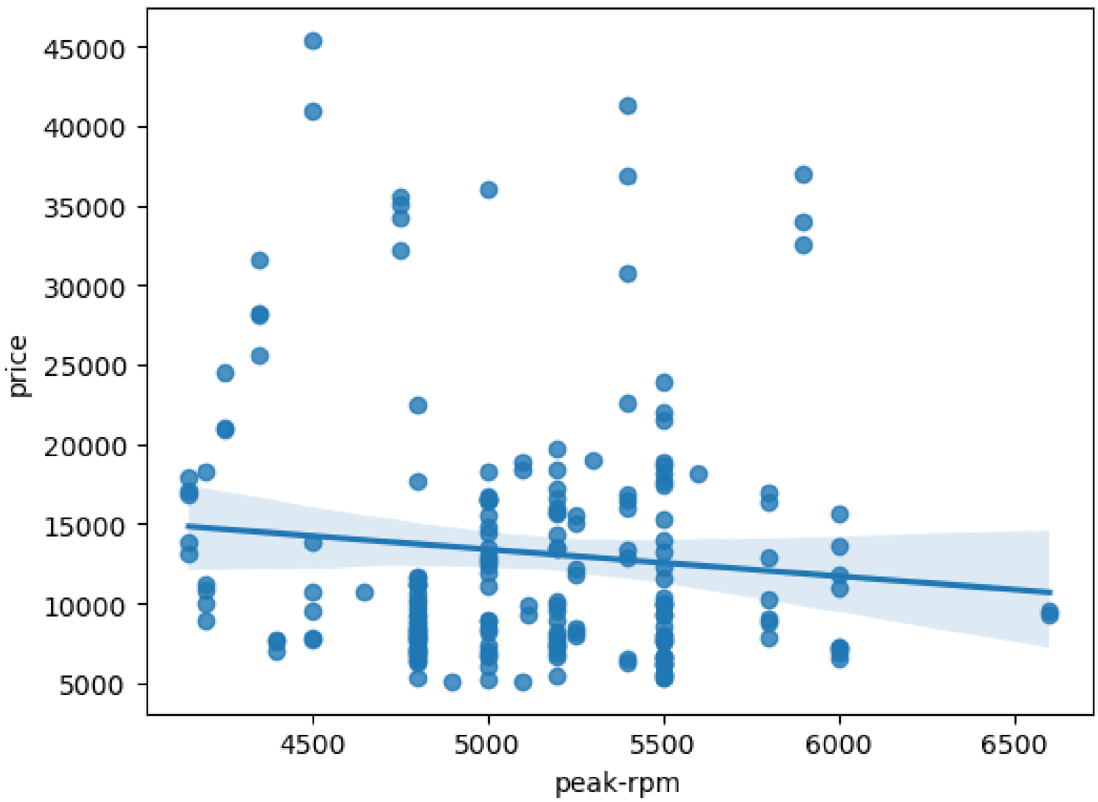
[47]:
```
# Fitting the simple linear regression
# Let's visualize the relationship between 'highway-L/100km' and 'price'
sns.regplot(x = 'highway-L/100km', y = 'price', data = df)
```

[47]: <Axes: xlabel='highway-L/100km', ylabel='price'>



[48]:
```
# This time, visualize the relationship between 'peak-rpm' and 'price'
sns.regplot(x = 'peak-rpm', y = 'price', data = df)
```

[48]: <Axes: xlabel='peak-rpm', ylabel='price'>



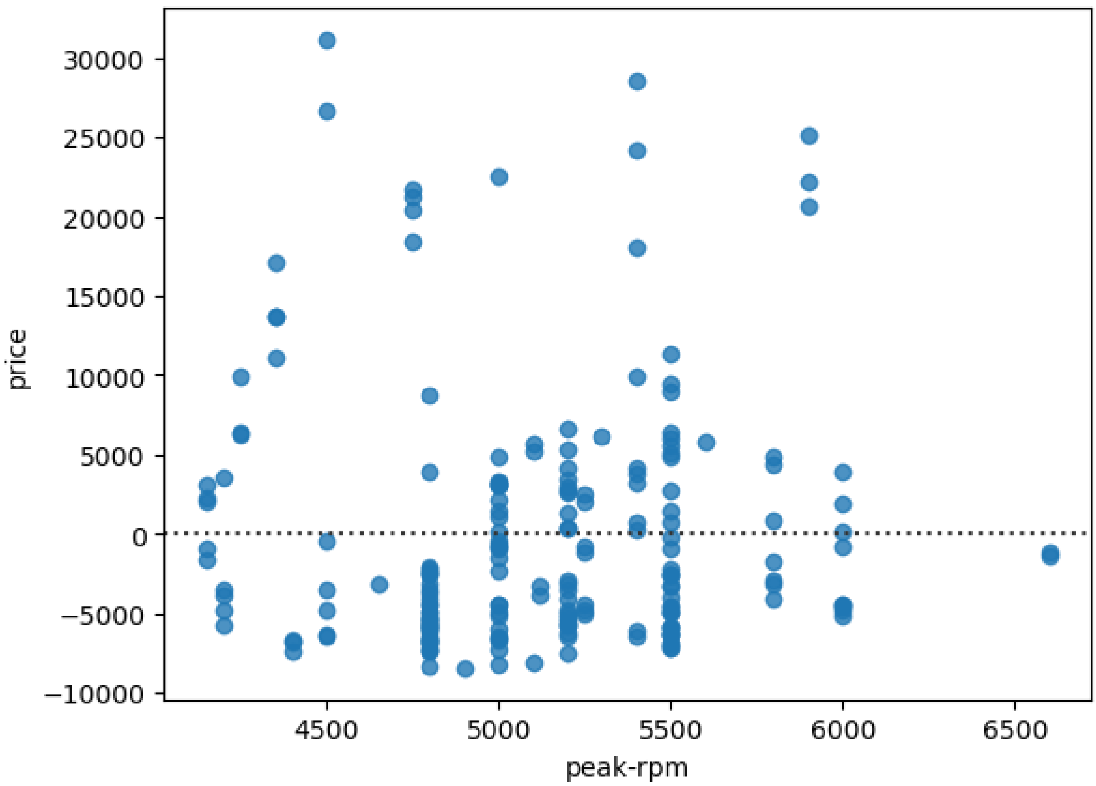
16 Model - Residual Plot

The residual plot tells us whether the constant variance assumption is violate or not.

If the 'assumptions' are met, then the linear model is appropriate.

[49]:
```
# Let's explore the residual plot of 'peak-rpm' and 'price'
sns.residplot(x= df['peak-rpm'],y = df['price'])
```

[49]: `<Axes: xlabel='peak-rpm', ylabel='price'>`



The residual plot above shows that the residuals are not randomly distributed hence it might not be an appropriate linear model.

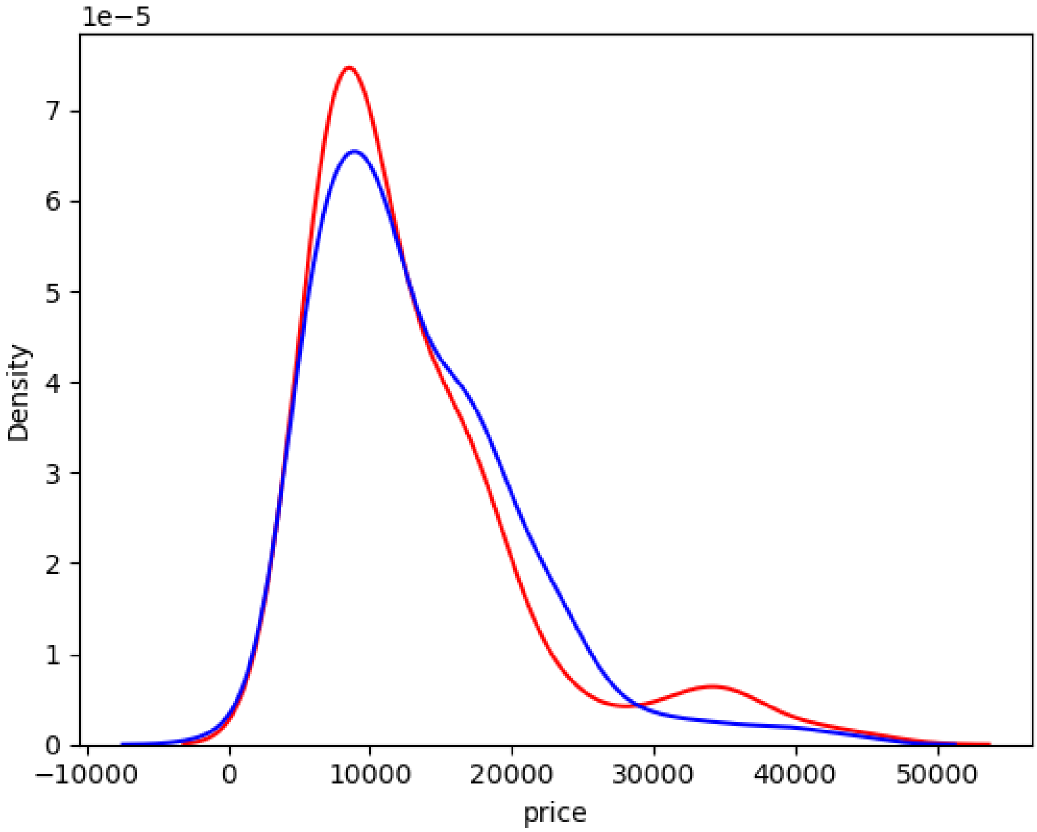
17 Model - Visualization (Multiple Linear Regression Model)

How do we visualize multiple linear regression model?

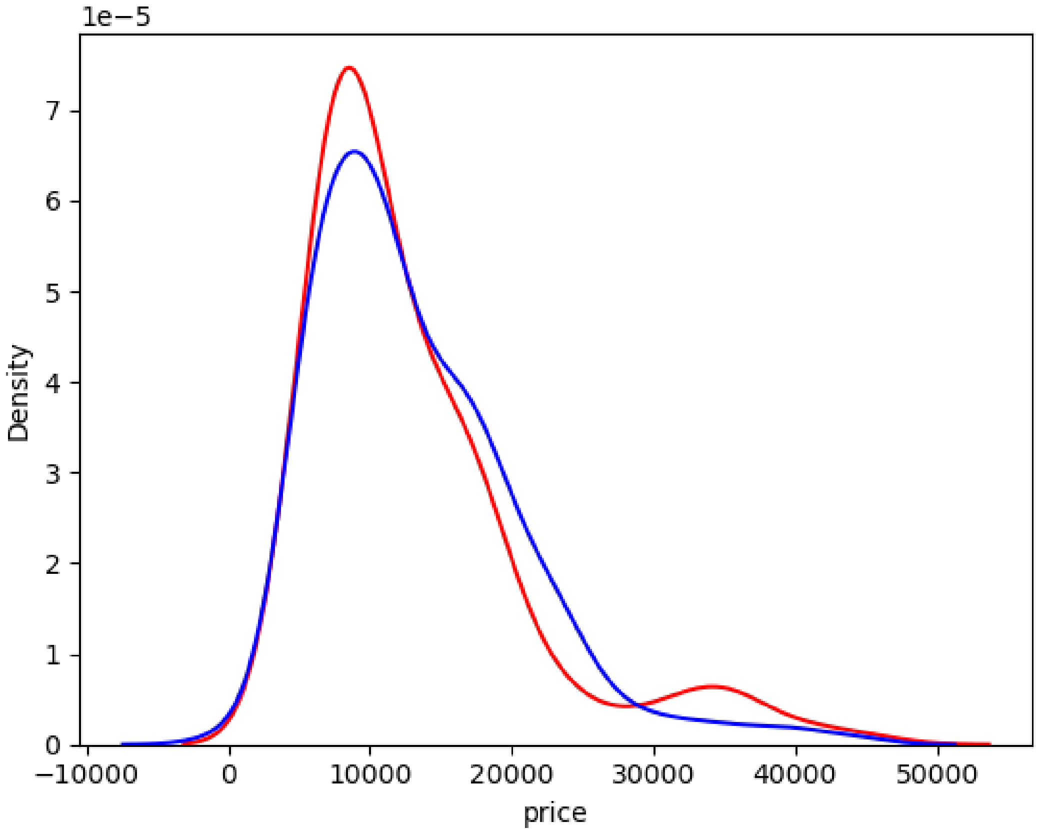
One way is by looking at the fit of the model on distribution plot

[50]:
```python
# Let's predict the Y_hat from our multiple linear regression model lm1
Y_hat = lm1.predict(Z)

# Create the distribution plot,
# The first argument is the variable of the distribution
# The second argument, setting histogram = False
# The third argument, setting the color
# Finally, overlapped the two plots

plot1 = sns.distplot(df['price'], hist= False, color= 'r', label = 'Actual␣
↪Value')
plot2 = sns.distplot(Y_hat, hist= False, color= 'b', ax = plot1, label =␣
↪'Fitted Value')
```

```
/tmp/ipykernel_450/2046564249.py:10: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `kdeplot` (an axes-level function for kernel density
plots).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  plot1 = sns.distplot(df['price'], hist= False, color= 'r', label = 'Actual
Value')
/tmp/ipykernel_450/2046564249.py:11: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `kdeplot` (an axes-level function for kernel density
plots).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  plot2 = sns.distplot(Y_hat, hist= False, color= 'b', ax = plot1, label =
'Fitted Value')
```



[51]:
```
# Alternatively, we can use the code below since sns update will deprecate
 ↪distplot()
plot1 = sns.kdeplot(df['price'], label = 'Actual Value', color = 'r')
plot2 = sns.kdeplot(Y_hat, label = 'Fitted Value', ax = plot1, color = 'b')
```



18 Model - Polynomial Regression Model

Sometimes when fitting mutiple regression model,the relationsip between the independent variables and the dependent variable does not appear linear.

We can fix achieve better fit by using polynomial regression model!

Quadratic - 2nd Order

$$Yhat = a + b_1X + b_2X^2$$

Cubic - 3rd Order

$$Yhat = a + b_1X + b_2X^2 + b_3X^3$$

Higher-Order:

$$Y = a + b_1X + b_2X^2 + b_3X^3....$$

[53]:
```python
# Let's visualize that by defining a new function 'ploy'

# 1. defines a function named ploy that takes in four arguments
def ploy(model, independent_variable, dependent_variable, Name):
    # Creating 100 evenly spaced values between 15 and 55
    x_new = np.linspace(15, 55, 100)
    # Calculates the corresponding y-values by applying the provided model
    y_new = model(x_new)

    # Creating the plot,
    # '.', dots for actual data points from indepdent_variable and dependent␣
 ↪variable
    # '-', line connecting the new x-value and their corresponding y-value
    plt.plot(independent_variable, dependent_variable, '.', x_new, y_new, '-')

    # Adding title and changing the background color
    plt.title('Polynomial Fit with Matplotlib for Price ~ Length')
    ax = plt.gca()
    ax.set_facecolor((0.898, 0.898, 0.898))

    # Customizing X and Y labels
    fig = plt.gcf()
    plt.xlabel(Name)
    plt.ylabel('Price of Cars')

    # Displaying and closing the plot
    plt.show()
    plt.close()

# We can use this function to visualize the distribution plot
```

[63]:
```python
# Get the variables and then use a polynomial of the 3rd order
x = df['peak-rpm']
y = df['price']
# Fit the polynomial using the function polyfit and poly1d to display the␣
 ↪function
f = np.polyfit(x,y,3)
p = np.poly1d(f)
print(f)
print(p)
```

```
[-3.66921457e-06  6.03980389e-02 -3.29227169e+02  6.07145229e+05]
            3          2
-3.669e-06 x + 0.0604 x - 329.2 x + 6.071e+05
```

Reference: https://www.coursera.org/learn/data-analysis-with-python