Data Wrangling

Welcome!

By the end of this notebook, you will have learned the basics of Data Wrangling!

Table of content

In [ ]:

```
```

Importing necessary libraries

In [1]:

```
import numpy as np
import pandas as pd
```

Importing Data and reading into a Pandas DataFrame

> Python list **columns (cols)** containing name of headers

In [2]:

```
cols = ['symboling', 'normalized-losses', 'make', 'fuel-type', 'aspiration', 'num-o
data = pd.read_csv('../input/imports-85.data.txt', names=cols)
print(data.shape)
data.head()
```

```
(205, 26)
```

Out[2]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee bas
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.

As we can see, several question marks appeared in the dataframe; those are missing values which may hinder our further analysis.
So, how do we identify all those missing values and deal with them?

How to work with missing data?

Steps for working with missing data:

1. dentify missing data
2. deal with missing data
3. correct data format

Identify and handle missing values

Identify missing values

Convert "?" to NaN
In the car dataset, missing data comes with the question mark "?".

In [3]:

```
data = data.replace("?", np.NaN)
data.head()
```

Out[3]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee bas
0	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88.
1	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88.
2	1	NaN	alfa-romero	gas	std	two	hatchback	rwd	front	94.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.

dentify_missing_values

Evaluating for Missing Data

The missing values are converted to Python's default. We use Python's built-in functions to identify these missing values. There are two methods to detect missing data:

1. **.isnull()**
2. **.notnull()**

The output is a boolean value indicating whether the value that is passed into the argument is in fact missing data.

In [4]:

```
data.isnull().any().any()
```

Out[4]:

```
True
```

"True" stands for missing value, while "False" stands for not missing value.

Count missing values in each column

Using a for loop in Python, we can quickly figure out the number of missing values in each column. As mentioned above, "True" represents a missing value, "False" means the value is present in the dataset. In the body of the for loop the method ".value_counts()" counts the number of "True" values.

In [5]:

```
data.isnull().sum()
```

Out[5]:

```
symboling            0
normalized-losses    41
make                 0
fuel-type            0
aspiration           0
num-of-doors         2
body-style           0
drive-wheels         0
engine-location      0
wheel-base           0
length               0
width                0
height               0
curb-weight          0
engine-type          0
num-of-cylinders     0
engine-size          0
fuel-system          0
bore                 4
stroke               4
compression-ratio    0
horsepower           2
peak-rpm             2
city-mpg             0
highway-mpg          0
price                4
dtype: int64
```

Based on the summary above, each column has 205 rows of data, seven columns containing missing data:

1. "normalized-losses": 41 missing data
2. "num-of-doors": 2 missing data
3. "bore": 4 missing data
4. "stroke" : 4 missing data
5. "horsepower": 2 missing data
6. "peak-rpm": 2 missing data
7. "price": 4 missing data

Deal with missing data

How to deal with missing data?

1. drop data
   a. drop the whole row
   b. drop the whole column
2. replace data
   a. replace it by mean
   b. replace it by frequency
   c. replace it based on other functions

Whole columns should be dropped only if most entries in the column are empty. In our dataset, none of the columns are empty enough to drop entirely. We have some freedom in choosing which method to replace data; however, some methods may seem more reasonable than others. We will apply each method to many different columns:

Replace by mean:

- "normalized-losses": 41 missing data, replace them with mean
- "stroke": 4 missing data, replace them with mean
- "bore": 4 missing data, replace them with mean
- "horsepower": 2 missing data, replace them with mean
- "peak-rpm": 2 missing data, replace them with mean

Replace by frequency:

- "num-of-doors": 2 missing data, replace them with "four".
  - Reason: 84% sedans is four doors. Since four doors is most frequent, it is most likely to occur

Drop the whole row:

- "price": 4 missing data, simply delete the whole row
  - Reason: price is what we want to predict. Any data entry without price data cannot be used for prediction; therefore any row now without price data is not useful to us

Calculate the average of the column

In [6]:

```
avg_norm_loss = data['normalized-losses'].astype("float").mean()
avg_norm_loss
```

Out[6]:

```
122.0
```

Replace "NaN" by mean value in "normalized-losses" column

In [7]:

```
data["normalized-losses"].replace(np.NaN, avg_norm_loss, inplace = True)
data["normalized-losses"]
```

Out[7]:

```
0      122
1      122
2      122
3      164
4      164
5      122
6      158
7      122
8      158
9      122
10     192
11     192
12     188
13     188
14     122
15     122
16     122
17     122
18     121
19      98
20      81
21     118
22     118
23     118
24     148
25     148
26     148
27     148
28     110
29     145
      ...
175     65
176     65
177     65
178    197
179    197
180     90
181    122
182    122
183    122
184     94
185     94
186     94
187     94
188     94
189    122
190    256
191    122
192    122
193    122
194    103
195     74
196    103
197     74
```

```
198    103
199     74
200     95
201     95
202     95
203     95
204     95
Name: normalized-losses, Length: 205, dtype: object
```

Calculate the mean value for 'bore' column

In [8]:

```
avg_bore = data["bore"].astype("float").mean()
data["bore"].replace(np.NaN, avg_bore, inplace = True)
data['bore']
```

Out[8]:

```
0      3.47
1      3.47
2      2.68
3      3.19
4      3.19
5      3.19
6      3.19
7      3.19
8      3.13
9      3.13
10     3.50
11     3.50
12     3.31
13     3.31
14     3.31
15     3.62
16     3.62
17     3.62
18     2.91
19     3.03
20     3.03
21     2.97
22     2.97
23     3.03
24     2.97
25     2.97
26     2.97
27     3.03
28     3.34
29     3.60
       ...
175    3.31
176    3.31
177    3.31
178    3.27
179    3.27
180    3.27
181    3.27
182    3.01
183    3.19
184    3.01
185    3.19
186    3.19
187    3.01
188    3.19
189    3.19
190    3.19
191    3.19
192    3.01
193    3.19
194    3.78
195    3.78
196    3.78
```

```
197    3.78
198    3.62
199    3.62
200    3.78
201    3.78
202    3.58
203    3.01
204    3.78
Name: bore, Length: 205, dtype: object
```

In [9]:

```
avg_stroke = data["stroke"].astype("float").mean(axis = 0)
print("Average of stroke:", avg_stroke)

# replace NaN by mean value in "stroke" column
data["stroke"].replace(np.nan, avg_stroke, inplace = True)
```

```
Average of stroke: 3.2554228855721337
```

In [10]:

```
data.isnull().sum()
```

Out[10]:

```
symboling            0
normalized-losses    0
make                 0
fuel-type            0
aspiration           0
num-of-doors         2
body-style           0
drive-wheels         0
engine-location      0
wheel-base           0
length               0
width                0
height               0
curb-weight          0
engine-type          0
num-of-cylinders     0
engine-size          0
fuel-system          0
bore                 0
stroke               0
compression-ratio    0
horsepower           2
peak-rpm             2
city-mpg             0
highway-mpg          0
price                4
dtype: int64
```

In [11]:

```
data["bore"].dtype
```

Out[11]:

```
dtype('O')
```

for loop to achieve the same

In [12]:

```
# for head in cols:
#     if data[head].isnull().any() == True and (data[head].dtype == "int64" or "flo
#         avg = data[head].astype("float").mean(axis = 0)
#         print(avg)
```

In [13]:

```
avg_horsepower = data['horsepower'].astype('float').mean(axis=0)
print("Average horsepower:", avg_horsepower)
data['horsepower'].replace(np.nan, avg_horsepower, inplace=True)
```

```
Average horsepower: 104.25615763546799
```

In [14]:

```
avg_peakrpm= data['peak-rpm'].astype('float').mean(axis=0)
print("Average peak rpm:", avg_peakrpm)
data['peak-rpm'].replace(np.nan, avg_peakrpm, inplace=True)
```

```
Average peak rpm: 5125.369458128079
```

To see which values are present in a particular column, we can use the ".value_counts()" method:

In [15]:

```
data['num-of-doors'].value_counts()
```

Out[15]:

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

We can see that four doors are the most common type. We can also use the ".idxmax()" method to calculate for us the most common type automatically:

In [16]:

```
data['num-of-doors'].value_counts().idxmax()
```

Out[16]:

```
'four'
```

The replacement procedure is very similar to what we have seen previously

In [17]:

```
#replace the missing 'num-of-doors' values by the most frequent
data["num-of-doors"].replace(np.nan, "four", inplace=True)
data.head()
```

Out[17]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel bas
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.

Finally, let's drop all rows that do not have price data:

In [18]:

```
# simply drop whole row with NaN in "price" column
before_rows = data.shape[0]
data.dropna(subset=["price"], axis=0, inplace=True)
after_rows = data.shape[0]
print("Number of dropped rows {}".format(before_rows - after_rows))
# reset index, because we droped two rows
data.reset_index(drop=True, inplace=True)
```

```
Number of dropped rows 4
```

In [19]:

```
data.shape
```

Out[19]:

```
(201, 26)
```

Correct data format

We are almost there!

The last step in data cleaning is checking and making sure that all data is in the correct format (int, float, text or other).

In Pandas, we use

.dtype() to check the data type

.astype() to change the data type

Lets list the data types for each column

In [20]:

```
data.dtypes
```

Out[20]:

```
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
```

As we can see above, some columns are not of the correct data type. Numerical variables should have type 'float' or 'int', and variables with strings such as categories should have type 'object'. For example, 'bore' and 'stroke' variables are numerical values that describe the engines, so we should expect them to be of the type 'float' or 'int'; however, they are shown as type 'object'. We have to convert data types into a proper format for each column using the "astype()" method.

Convert data types to proper format

In [21]:

```
data[["bore", "stroke"]] = data[["bore", "stroke"]].astype("float")
data[["normalized-losses"]] = data[["normalized-losses"]].astype("int")
data[["price"]] = data[["price"]].astype("float")
data[["peak-rpm"]] = data[["peak-rpm"]].astype("float")
data.head()
```

Out[21]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee bas
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.

Wonderful!

Now, we finally obtain the cleaned dataset with no missing values and all data in its proper format.

Data Standardization

Data is usually collected from different agencies with different formats. (Data Standardization is also a term for a particular type of data normalization, where we subtract the mean and divide by the standard deviation)

What is Standardization?

Standardization is the process of transforming data into a common format which allows the researcher to make the meaningful comparison.

Example

Transform mpg to L/100km:

In our dataset, the fuel consumption columns "city-mpg" and "highway-mpg" are represented by mpg (miles per gallon) unit. Assume we are developing an application in a country that accept the fuel consumption with L/100km standard

We will need to apply **data transformation** to transform mpg into L/100km?

The formula for unit conversion is

L/100km = 235 / mpg

We can do many mathematical operations directly in Pandas.

In [22]:

```
data.head()
```

Out[22]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee bas
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.

Transform mpg to L/100km in the column of "highway-mpg", and change the name of column to "highway-L/100km".

In [23]:

```
# transform mpg to L/100km by mathematical operation (235 divided by mpg)
data["highway-mpg"] = 235/data["highway-mpg"]

# rename column name from "highway-mpg" to "highway-L/100km"
data.rename(columns = {'highway-mpg':'highway-L/100km'}, inplace=True)

# check your transformed data
data.head()
```

Out[23]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee bas
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.

Data Normalization

Why normalization?

Normalization is the process of transforming values of several variables into a similar range. Typical normalizations include scaling the variable so the variable average is 0, scaling the variable so the variance is 1, or scaling variable so the variable values range from 0 to 1

Example

To demonstrate normalization, let's say we want to scale the columns "length", "width" and "height"

Target:would like to Normalize those variables so their value ranges from 0 to 1.

Approach: replace original value by (original value)/(maximum value)

In [24]:

```
# replace (original value) by (original value)/(maximum value)
data['length'] = data['length']/data['length'].max()
data['width'] = data['width']/data['width'].max()
data.head()
```

Out[24]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee bas
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.

In [25]:

```
# replace height with normalized values
data["height"] = data["height"]/data["height"].max()
```

In [26]:

```
data.head()
```

Out[26]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee bas
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.

Binning

Why binning?

Binning is a process of transforming continuous numerical variables into discrete categorical 'bins', for grouped analysis.

Example:

In our dataset, "horsepower" is a real valued variable ranging from 48 to 288, it has 57 unique values. What if we only care about the price difference between cars with high horsepower, medium horsepower, and little horsepower (3 types)? Can we rearrange them into three ‘bins' to simplify analysis?

We will use the Pandas method 'cut' to segment the 'horsepower' column into 3 bins

Example of Binning Data In Pandas

Convert data to correct format

In [27]:

```
data["horsepower"] = data["horsepower"].astype(int, copy = True)
```

In [28]:

```
data.head()
```

Out[28]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee bas
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.

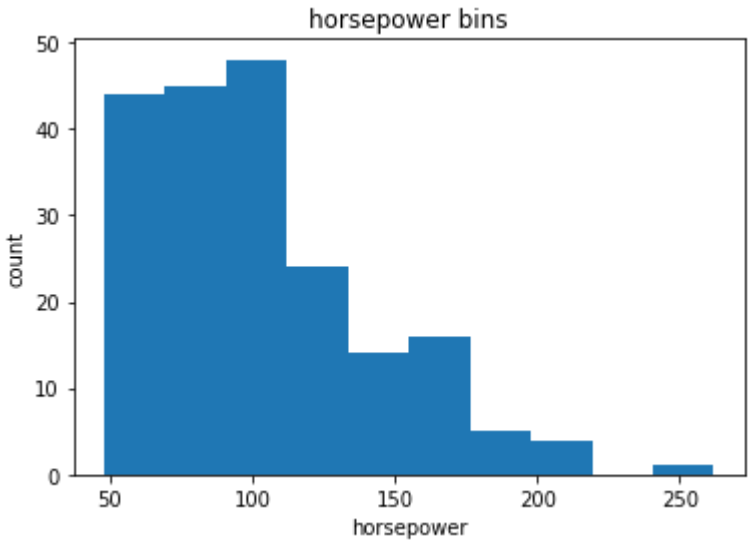
In [29]:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.hist(data["horsepower"])

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

Out[29]:

```
Text(0.5, 1.0, 'horsepower bins')
```



We would like 3 bins of equal size bandwidth so we use numpy's `linspace(start_value, end_value, numbers_generated` function.

Since we want to include the minimum value of horsepower we want to set start_value=min(df["horsepower"]).

Since we want to include the maximum value of horsepower we want to set end_value=max(df["horsepower"]).

Since we are building 3 bins of equal length, there should be 4 dividers, so numbers_generated=4.

We build a bin array, with a minimum value to a maximum value, with bandwidth calculated above. The bins will be values used to determine when one bin ends and another begins.

In [30]:

```
bins = np.linspace(min(data["horsepower"]), max(data["horsepower"]), 4)
bins
```

Out[30]:

```
array([ 48.        , 119.33333333, 190.66666667, 262.        ])
```

In [31]:

```
group_names = ['Low', 'Medium', 'High']
```

We apply the function "cut" the determine what each value of "df['horsepower']" belongs to.

In [32]:

```
data['horsepower-binned'] = pd.cut(data['horsepower'], bins, labels=group_names, in
data[['horsepower','horsepower-binned']].head(20)
```

Out[32]:

	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
9	101	Low
10	101	Low
11	121	Medium
12	121	Medium
13	121	Medium
14	182	Medium
15	182	Medium
16	182	Medium
17	48	Low
18	70	Low
19	70	Low

In [33]:

```
data["horsepower-binned"].value_counts()
```

Out[33]:

```
Low       153
Medium     43
High        5
Name: horsepower-binned, dtype: int64
```

Let's plot the distribution of each bin

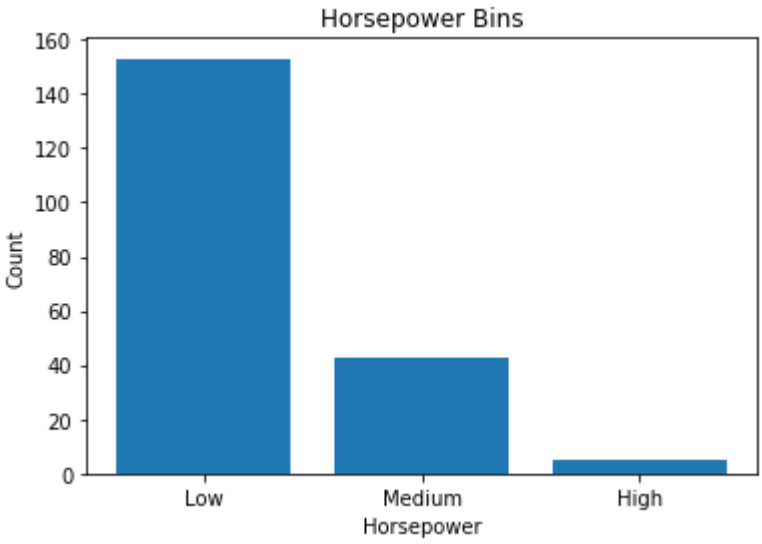
In [34]:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.bar(group_names, data["horsepower-binned"].value_counts())

# set x/y labels and plot title
plt.xlabel('Horsepower')
plt.ylabel("Count")
plt.title("Horsepower Bins")
```

Out[34]:

```
Text(0.5, 1.0, 'Horsepower Bins')
```



Check the dataframe above carefully, you will find the last column provides the bins for "horsepower" with 3 categories ("Low","Medium" and "High").

We successfully narrow the intervals from 57 to 3!

Bins visualization

Normally, a histogram is used to visualize the distribution of bins we created above.

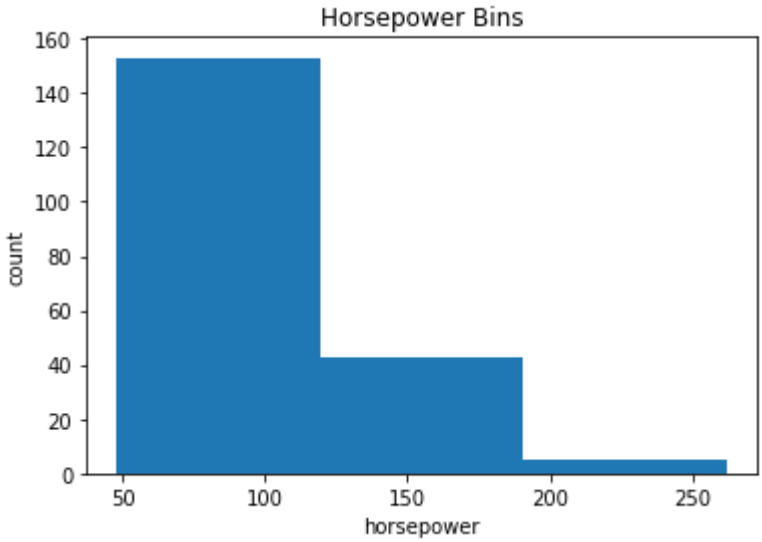
In [35]:

```
%matplotlib inline
import matplotlib.pyplot as plt

a = (0, 1, 2)

# draw histogram of attribute
plt.hist(data["horsepower"], bins = 3)

# set x / y labels and plot title
plt.xlabel("horsepower")
plt.ylabel("count")
plt.title("Horsepower Bins")
plt.show()
```



Indicator variable (or dummy variable)

What is an indicator variable?

An indicator variable (or dummy variable) is a numerical variable used to label categories. They are called 'dummies' because the numbers themselves don't have inherent meaning.

Why we use indicator variables?

So we can use categorical variables for regression analysis in the later modules.

Example

We see the column "fuel-type" has two unique values, "gas" or "diesel". Regression doesn't understand words, only numbers. To use this attribute in regression analysis, we convert "fuel-type" into indicator variables.

We will use the panda's method 'get_dummies' to assign numerical values to different categories of fuel type.

In [36]:

```
data.columns
```

Out[36]:

```
Index(['symboling', 'normalized-losses', 'make', 'fuel-type', 'aspirat
ion',
       'num-of-doors', 'body-style', 'drive-wheels', 'engine-locatio
n',
       'wheel-base', 'length', 'width', 'height', 'curb-weight', 'engi
ne-type',
       'num-of-cylinders', 'engine-size', 'fuel-system', 'bore', 'stro
ke',
       'compression-ratio', 'horsepower', 'peak-rpm', 'city-mpg',
       'highway-L/100km', 'price', 'horsepower-binned'],
      dtype='object')
```

In [37]:

```
dummy_variable_1 = pd.get_dummies(data["fuel-type"])
dummy_variable_1.head()
```

Out[37]:

	diesel	gas
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1

In [38]:

```
dummy_variable_1.rename(columns={'fuel-type-diesel':'gas', 'fuel-type-diesel':'dies
dummy_variable_1.head()
```

Out[38]:

	diesel	gas
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1

We now have the value 0 to represent "gas" and 1 to represent "diesel" in the column "fuel-type". We will now insert this column back into our original dataset.

In [39]:

```
# merge data frame "df" and "dummy_variable_1"
data = pd.concat([data, dummy_variable_1], axis=1)

# drop original column "fuel-type" from "df"
data.drop("fuel-type", axis = 1, inplace=True)
```

In [40]:

```
# get indicator variables of aspiration and assign it to data frame "dummy_variable_
dummy_variable_2 = pd.get_dummies(data['aspiration'])

# change column names for clarity
dummy_variable_2.rename(columns={'std':'aspiration-std', 'turbo': 'aspiration-turbo

# show first 5 instances of data frame "dummy_variable_1"
dummy_variable_2.head()
```

Out[40]:

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

In [41]:

```
#merge the new dataframe to the original datafram
data = pd.concat([data, dummy_variable_2], axis=1)

# drop original column "aspiration" from "df"
data.drop('aspiration', axis = 1, inplace=True)
```

In [42]:

```
data.head(10)
```

Out[42]:

	symboling	normalized-losses	make	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	length	v
0	3	122	alfa-romero	two	convertible	rwd	front	88.6	0.811148	0.89
1	3	122	alfa-romero	two	convertible	rwd	front	88.6	0.811148	0.89
2	1	122	alfa-romero	two	hatchback	rwd	front	94.5	0.822681	0.90
3	2	164	audi	four	sedan	fwd	front	99.8	0.848630	0.91
4	2	164	audi	four	sedan	4wd	front	99.4	0.848630	0.92
5	2	122	audi	two	sedan	fwd	front	99.8	0.851994	0.92
6	1	158	audi	four	sedan	fwd	front	105.8	0.925997	0.99
7	1	122	audi	four	wagon	fwd	front	105.8	0.925997	0.99
8	1	158	audi	four	sedan	fwd	front	105.8	0.925997	0.99
9	2	192	bmw	two	sedan	rwd	front	101.2	0.849592	0.90

In [43]:

```
data.describe()
```

Out[43]:

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight
count	201.000000	201.00000	201.000000	201.000000	201.000000	201.000000	201.000000
mean	0.840796	122.00000	98.797015	0.837102	0.915126	0.899108	2555.666667
std	1.254802	31.99625	6.066366	0.059213	0.029187	0.040933	517.296727
min	-2.000000	65.00000	86.600000	0.678039	0.837500	0.799331	1488.000000
25%	0.000000	101.00000	94.500000	0.801538	0.890278	0.869565	2169.000000
50%	1.000000	122.00000	97.000000	0.832292	0.909722	0.904682	2414.000000
75%	2.000000	137.00000	102.400000	0.881788	0.925000	0.928094	2926.000000
max	3.000000	256.00000	120.900000	1.000000	1.000000	1.000000	4066.000000

In [44]:

```
# Convert to CSV file
data.to_csv('wrangled_data.csv')
```