Chapter 3

Data Analysis with Python

Note : Will attach all the PDF version of the notebooks at the end of this chapter

Python Packages for Data Science

- A Python library is a collection of functions and methods that allow you to perform lots of actions without writing any code. The libraries usually contain built in modules providing different functionalities which you can use directly.
- Broad Classifications
  - Scientific Computing Libraries
    - **Pandas** offers data structure and tools for effective data manipulation and analysis. It provides facts, access to structured data.
    - **NumPy** library uses arrays for its inputs and outputs. It can be extended to objects for matrices and with minor coding changes, developers can perform fast array processing.
    - **SciPy** includes functions for some advanced math problems as listed on this slide, as well as data visualization.
  - Data Visualization
    - Matplotlib package is the most well known library for data visualization. It is great for making graphs and plots. The graphs are also highly customizable.
    - Seaborn which
      is based on Matplotlib. It's very easy to generate various plots such as heat maps, time series and violin plots.

- Machine Learning
  * Scikit-learn library contains tools statistical modeling, including regression, classification, clustering, and so on. This library is built on NumPy, SciPy and Matplotib.
  * Statsmodels is also a Python module that allows users to explore data, estimate statistical models, and perform statistical tests.

Importing and Exporting Data in Python

```
# Importing the required modules
import pandas as pd
url = "something.csv"
df = pd.read_csv(url, header = None) # read csv ( Also xslx,
    html and so on can be included change read_(type))
df.head(n) # display first n rows
df.tail(n) # display last n rows
path = "something1.csv"
df.to_csv(path) # to save to a csv file
headers = [some list]
df.columns = headers # replaces default headers
df.info # concise summary of dataframe
```

Getting started analysing data in python

```
# Importing the required modules
import pandas as pd
df = read_csv(url)
df.dtypes # gives the data types
df.describe() # returns a statistical summary
df.describe(include = "all") # even for object type arguments
```

Accessing databases with Python

- Databases are powerful tools for data scientists
- Python program communicates with the DBMS using API.

- The application program begins its database access with one or more API calls that connect the program to the DBMS.
- To send the SQL statement to the DBMS, the program builds the statement as a text string in a buffer and then makes an API call to pass the buffer contents to the DBMS.
- The application program makes API calls to check the status of its DBMS request and to handle errors. The application program ends its database access with an API call that disconnects it from the database.
- DB-API is Python's standard API for accessing relational databases. It is a standard that allows you to write a single program that works with multiple kinds of relational databases instead of writing a separate program for each one.
- So, if you learn the DB-API functions, then you can apply that knowledge to use any database with Python. The two main concepts in the Python DB-API are connection objects and query objects.
- You use connection objects to connect to a database and manage your transactions. Cursor objects are used to run queries. You open a cursor object and then run queries. The cursor works similar to a cursor in a text processing system where you scroll down in your result set and get your data into the application.
- Cursors are used to scan through the results of a database. Here are the methods used with connection objects. The cursor() method returns a new cursor object using the connection.
- The commit() method is used to commit any pending transaction to the database.
- The rollback() method causes the database to roll back to the start of any pending transaction.
- The close() method is used to close a database connection. Let's walk through a Python application that uses the DB-API to query a database.
- First, you import your database module by using the connect API from that module. To open a connection to the database, you use the connection function and pass in the parameters that is the database name, username, and password.

- The connect function returns connection object. After this, you create a cursor object on the connection object. The cursor is used to run queries and fetch results.
- After running the queries using the cursor, we also use the cursor to fetch the results of the query. Finally, when the system is done running the queries, it frees all resources by closing the connection.
- Remember that it is always important to close connections to avoid unused connections taking up resources.

```
from dbmodule import connect

#create connection object
connection = connect('databasename','username','pswd')

#create a cursor object
cursor = connection.cursor()

#Run queries
cursor.execute('select * from mytable')
results = cursor.fetchall()

#free resources
cursor.close()
connection.close()
```

Data Pre-Processing

It is the process of converting or mapping data from one raw form into another format to make it ready for further analysis. Data preprocessing is often called data cleaning or data wrangling, and there are likely other terms

```
# Importing the required modules
import pandas as pd
df['column'] # print the column
df['column'] = df['column'] + 1 # adds 1 to each column (Similarly
    a lot of simple arithemtic operations can be done)
```

Dealing with Missing Values in Python

- The first is to check if the person or group that collected the data can go back and find what the actual value should be. Another possibility is just to remove the data where that missing value is found. When you drop data, you could either drop the whole variable or just the single data entry with the missing value. If you don't have a lot of observations with missing data, usually dropping the particular entry is the best. If you're removing data, you want to look to do something that has the least amount of impact.

- Replacing data is better since no data is wasted. However, it is less accurate since we need to replace missing data with a guess of what the data should be.

- One standard for placement technique is to replace missing values by the average value of the entire variable.

```
# Importing the required modules
import pandas as pd
import numpy as np

# Imagine if we wanna remove the price column
df.dropna(subset = ["price"],axis = 1, inplace = True) # axis = 1
    drops the entire column

# Imagine if we wanan remove a certain missing value in rows of a
    specific(Price column will be considered here) column
df.dropna(subset = ["price"],axis = 0, inplace = True) # axis = 0
    drops the entire row

# To replace missing values
mean = df["price"].mean() # calculate the mean of the price column
df["price"].replace(np.nan,mean) # use the replace function that
    is df["column"].replace(missing_value, new_value) / np.nan =
    find all NAN values

# Maximum value for feature scaling is 1 and minimum is 0
```

Binning

- Binning is when you group values together into bins.

- Sometimes, binning can improve accuracy of the predictive models. In addition, sometimes we use data binning to group a set of numerical values into a smaller number of bins to have a better understanding of the data distribution

```
# Importing the required modules
import pandas as pd
bins = np.linspace(min(df["price"]),max(df["price"]),4) # Returns
    the array bins that contains 4 equally spaced numbers over the
    specified interval of the price.

"""
np.linspace() = returns an array refer to numpy documentation
bins = 3 in this example
But in the argument 4 is given as bins + 1 always contains the
    intervals
Example : 1-3 , 3-4, 4-6
3 bins but 4 numbers
"""

names = ["low","Medium","High"] # group names

df["price-binned"] =pd.cut(df["price"],bins,labels = names,
    include_lowest = True) # df["price-bined] changes the column
    price to price-binned
```

Categorical Variables into Quantitative variables

- Statistical models cannot take in objects or strings as input and for model training only take the numbers as inputs.

```
# Importing the required modules
import pandas as pd
df = read_csv(url)

# Convert catgeorical variables to dummy variables 0-1
pd.get_dummies(df["fuel"]) # The get_dummies method automatically
    generates a list of numbers, each one corresponding to a
    particular category of the variable.
```

```
df_dummies = pd.get_dummies(df, prefix='Fuel', prefix_sep='.',
columns=['Fuel']) # This is a more advanced method read more in
    the docs
```

Categorical Variables into Quantitative variables

- Statistical models cannot take in objects or strings as input and for model training only take the numbers as inputs.

```
# Importing the required modules
import pandas as pd
df = read_csv(url)

# Convert catgeorical variables to dummy variables 0-1
pd.get_dummies(df["fuel"]) # The get_dummies method automatically
    generates a list of numbers, each one corresponding to a
    particular category of the variable.

df_dummies = pd.get_dummies(df, prefix='Fuel', prefix_sep='.',
columns=['Fuel']) # This is a more advanced method read more in
    the docs
```

Exploratory Data Analysis

Note : This section required visual graphs and plots so the notebook will cover most of it

```
# Importing the required modules
import pandas as pd
import seaborn as sns

df.value_counts() # summarize the categorical data

"""BOX PLOTS
1 - Box plots are great way to visualize numeric data, since you
    can visualize the various distributions of the data.
2 - Box plots, you can easily spot outliers and also see the
    distribution and skewness of the data. Box plots make it easy
    to compare between groups.
3 - Main features - Median, IQR, Mean, outliers
```

```
"""
sns.boxplot(x = "drive",y = "price", data = df) # using seaborn
    plot a boxplot

""" SCATTER PLOTS
It shows he relationship between two variables.
1 - Typically set the predictor variable on the x-axis or
    horizontal axis - independent
2 - Set the target variable on the y-axis or vertical axis -
    dependent
"""
# Example

y = df["price"]
x = df["engine-size"]
plt.scatter(x,y)

plt.title("Scatterplot of Engine Size vs Price)
plt.xlabel("Engine Size")
plt.ylabel("Price")
```

Groupby

- In Pandas, this can be done using the group by method. The group by method is used on categorical variables, groups the data into subsets according to the different categories of that variable.
- You can group by a single variable or you can group by multiple variables by passing in multiple variable names.

```
# Importing the required modules
import pandas as pd
df = read_csv(url)

df_group = df[["price","size"]]
df_grp = df_group.groupby(["price","size"],as_index =
    False).mean() # groupby these columns and only the average
    price is returned for each size
df_grp

# Pivot table
# One variable displayed along the columns and the other displayed
    along the rows
```

```
df_pivot = df_grp.pivot(index = "drive-wheels", columns
    ="body-size") # returns a rectangular grid which is easier to
    understand

# Heatmap
# plot target over multiple variables

plt.pcolor(df_pivot,cmap = 'RdBu') # pass in the data and also
    cmap specifies the color correlation

plt.colorbar()
plt.show()
```

Correlation

- Correlation is a statistical metric for measuring to what extent different variables are interdependent. In other words, when we look at two variables over time, if one variable changes how does this affect change in the other variable?
- correlation doesnt imply causation.
- Pearson Correlation - Pearson correlation method will give you two values: the correlation coefficient and the P-value
- A value close to 1 implies a large positive correlation, while a value close to negative 1 implies a large negative correlation, and a value close to zero implies no correlation between the variables.
- Next, the P-value will tell us how certain we are about the correlation that we calculated. For the P-value, a value less than.001 gives us a strong certainty about the correlation coefficient that we calculated. A value between.001 and.05 gives us moderate certainty. A value between.05 and.1 will give us a weak certainty. And a P-value larger than.1 will give us no certainty of correlation at all. We can say that there is a strong correlation when the correlation coefficient is close to 1 or negative 1, and the P-value is less than.001.

```
# Importing the required modules
import pandas as pd
import seaborn as sns
```

```
df = read_csv(url)

# Correlation
sns.regplot(x = "size",y = "price" , data = df)
plt.ylim(0,)

# Pearson Correlation
pearson_coef, p_value =
    stats.pearsonr(df['horsepower'],df['price'])
```

Analysis of Variance ANOVA

- The correlation among different categories.
- The ANOVA test returns two values, the F-test score and the p-value. The F-test calculates the ratio of variation between groups mean, over the variation within each of the sample groups. The p-value shows whether the obtained result is statistically significant.
- F-test calculates the ratio of variation between groups means over the variation within each of the sample group means.

```
# Importing the required modules
import pandas as pd
import seaborn as sns
import scipy as stats
df = read_csv(url)

# ANOVA
df_anova = df[["make","price"]]

grouped_anova = df_anova.groupby(["make"]) # return a dataframe
    with grouped columns

results =
    stats.f_oneway(grouped_anova.get_group("honda"["price"],grouped_anova.get_group
# Looks Complicated but it is not it just returns the F value and
    p- value
```

Linear Regression and Multiple Linear Regression

- Linear regression will refer to one independent variable to make a prediction.
- $y' = m_1 x + c$
- Multiple linear regression will refer to multiple independent variables to make a prediction.
- The predictor independent variable x and the target dependent variable y.
- Multiple linear regression is used to explain the relationship between one continuous target y variable and two or more predictor x variables.
- $y' = m_1 x_1 + m_2 x_1 + c$

```
# Importing the required modules
import pandas as pd
import seaborn as sns
from sklearn.linear_model import LinearRegression

lm = LinearRegression() # create the model
x = df[["highway"]]
y = df[["price"]]

lm.fit(x,y) # fit the model

Yhat = lm.predict(x) # get the prediction


# Multiple Linear Model Estimator

z = df[["horsepower","weight"]]]

lm.fit(z,df["price"])

Yhat = lm.predict(x) # returns an array of values

lm.coef_ # returns the slope

lm.intercept_ # returns the intercept
```

Polynomial Regression and Pipelines

- he polynomial regression. We transform our data into a polynomial, then use linear regression to fit the parameter.
- This method is beneficial for describing curvilinear relationships. What is a curvilinear relationship? It's what you get by squaring or setting higher order terms of the predictor variables in the model transforming the data. The model can be quadratic, which means that the predictor variable in the model is squared.
- $y' = b_0 + b_1 x + b_2(x^2)$
- It can go upto any order
-

```
# Importing the required modules
import pandas as pd
import seaborn as sns
import scipy as stats
df = read_csv(url)

# Polynomial Regression

f = np.polyfit(x,y,2)
p = np.polyld(f)

print(p)
```

Measures for In-Sample Evaluation

- These measures are a way to numerically determine how good the model fits on our data. Two important measures that we often use to determine the fit of a model are: Mean Square Error (MSE), and R-squared.

Prediction and Decision Making

- How can we determine if our model is correct? The first thing you should do is make sure your model results make sense. You should always use visualization, numerical measures for evaluation and comparing between different models.

Model Evaulation and Refinement

Note : As this heavily math based and code everything will be explained through the notebook attached below

data-wrangling

June 21, 2020

Data Wrangling

Welcome!

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

What is the purpose of Data Wrangling?

Data Wrangling is the process of converting data from the initial format to a format that may be better for analysis.

What is the fuel consumption (L/100k) rate for the diesel car?

Import data

You can find the "Automobile Data Set" from the following link: https://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.data. We will be using this data set throughout this course.

Import pandas

[57]:
```
import pandas as pd
import matplotlib.pylab as plt
```

Reading the data set from the URL and adding the related headers.

URL of the dataset

This dataset was hosted on IBM Cloud object click HERE for free storage

[58]:
```
filename = "https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/
 ↪CognitiveClass/DA0101EN/auto.csv"
```

Python list headers containing name of headers

[59]:
```
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"]
```

Use the Pandas method read_csv() to load the data from the web address. Set the parameter "names" equal to the Python list "headers".

[60]:
```
df = pd.read_csv(filename, names = headers)
```

Use the method head() to display the first five rows of the dataframe.

[61]:
```
# To see what the data set looks like, we'll use the head() method.
df.head()
```

[61]:
```
   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]
```

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:

dentify missing data

deal with missing data

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 “?”. We replace “?” with NaN (Not a Number), which is Python’s default missing value marker, for reasons of computational speed and convenience. Here we use the function:

to replace A by B

[62]:
```
import numpy as np

# replace "?" to NaN
df.replace("?", np.nan, inplace = True)
df.head(5)
```

[62]:
```
   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]
```

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:

.isnull()

.notnull()

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

[63]:
```
missing_data = df.isnull()
missing_data.head(5)
```

[63]:
```
   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]
```

"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.

[64]:
```
for column in missing_data.columns.values.tolist():
    print(column)
    print (missing_data[column].value_counts())
    print("")
```

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

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

make
False    205
Name: make, dtype: int64

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

aspiration
False    205
Name: aspiration, dtype: int64

num-of-doors
False    203
True       2
Name: num-of-doors, dtype: int64

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

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

engine-location
False    205
```

```
Name: engine-location, dtype: int64

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

length
False    205
Name: length, dtype: int64

width
False    205
Name: width, dtype: int64

height
False    205
Name: height, dtype: int64

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

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

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

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

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

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

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

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

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

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

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

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

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

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

"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

Deal with missing data

How to deal with missing data?

drop data a. drop the whole row b. drop the whole column

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

[65]:
```
avg_norm_loss = df["normalized-losses"].astype("float").mean(axis=0)
print("Average of normalized-losses:", avg_norm_loss)
```

```
Average of normalized-losses: 122.0
```

Replace “NaN” by mean value in “normalized-losses” column

[66]:
```
df["normalized-losses"].replace(np.nan, avg_norm_loss, inplace=True)
```

Calculate the mean value for ‘bore’ column

[67]:
```
avg_bore=df['bore'].astype('float').mean(axis=0)
print("Average of bore:", avg_bore)
```

```
Average of bore: 3.3297512437810943
```

Replace NaN by mean value

[68]:
```
df["bore"].replace(np.nan, avg_bore, inplace=True)
```

Question #1:

According to the example above, replace NaN in "stroke" column by mean.

[69]:
```
avg_stroke = df["stroke"].astype("float").mean(axis = 0)
print("Average of stroke:", avg_stroke)
df["stroke"].replace(np.nan, avg_stroke, inplace = True)
```

```
Average of stroke: 3.255422885572139
```

Double-click here for the solution.

Calculate the mean value for the 'horsepower' column:

[70]:
```
avg_horsepower = df['horsepower'].astype('float').mean(axis=0)
print("Average horsepower:", avg_horsepower)
```

```
Average horsepower: 104.25615763546799
```

Replace "NaN" by mean value:

[71]:
```
df['horsepower'].replace(np.nan, avg_horsepower, inplace=True)
```

Calculate the mean value for 'peak-rpm' column:

[72]:
```
avg_peakrpm=df['peak-rpm'].astype('float').mean(axis=0)
print("Average peak rpm:", avg_peakrpm)
```

```
Average peak rpm: 5125.369458128079
```

Replace NaN by mean value:

[73]:
```
df['peak-rpm'].replace(np.nan, avg_peakrpm, inplace=True)
```

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

[74]:
```
df['num-of-doors'].value_counts()
```

[74]:
```
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:

[75]:
```
df['num-of-doors'].value_counts().idxmax()
```

[75]:
```
'four'
```

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

[76]:
```
#replace the missing 'num-of-doors' values by the most frequent
df["num-of-doors"].replace(np.nan, "four", inplace=True)
```

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

[77]:
```
# simply drop whole row with NaN in "price" column
df.dropna(subset=["price"], axis=0, inplace=True)

# reset index, because we droped two rows
df.reset_index(drop=True, inplace=True)
```

[78]:
```
df.head()
```

[78]:
```
   symboling normalized-losses         make fuel-type aspiration num-of-doors  \
0          3               122  alfa-romero       gas        std          two
1          3               122  alfa-romero       gas        std          two
2          1               122  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]
```

Good! Now, we obtain the dataset with no missing values.

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

[79]:
```
df.dtypes
```

[79]:
```
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.

[80]:
```
df[["bore", "stroke"]] = df[["bore", "stroke"]].astype("float")
df[["normalized-losses"]] = df[["normalized-losses"]].astype("int")
df[["price"]] = df[["price"]].astype("float")
df[["peak-rpm"]] = df[["peak-rpm"]].astype("float")
```

Let us list the columns after the conversion

[81]:
```python
df.dtypes
```

[81]:
```
symboling               int64
normalized-losses       int64
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
```

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.

[82]:
```
df.head()
```

[82]:
```
   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  …

   engine-size  fuel-system  bore  stroke compression-ratio horsepower  \
0          130         mpfi  3.47    2.68               9.0        111
1          130         mpfi  3.47    2.68               9.0        111
2          152         mpfi  2.68    3.47               9.0        154
3          109         mpfi  3.19    3.40              10.0        102
4          136         mpfi  3.19    3.40               8.0        115

   peak-rpm city-mpg  highway-mpg    price
0    5000.0       21           27  13495.0
1    5000.0       21           27  16500.0
2    5000.0       19           26  16500.0
3    5500.0       24           30  13950.0
4    5500.0       18           22  17450.0

[5 rows x 26 columns]
```

[83]:
```
# Convert mpg to L/100km by mathematical operation (235 divided by mpg)
df['city-L/100km'] = 235/df["city-mpg"]

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

[83]:
```
   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  …

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

  highway-mpg    price  city-L/100km
0          27  13495.0     11.190476
1          27  16500.0     11.190476
2          26  16500.0     12.368421
3          30  13950.0      9.791667
4          22  17450.0     13.055556

[5 rows x 27 columns]
```

Question #2:

According to the example above, transform mpg to L/100km in the column of "highway-mpg", and change the name of column to "highway-L/100km".

[84]:
```python
df["highway-mpg"] = 235/df["highway-mpg"]
df.rename(columns={'"highway-mpg"':'highway-L/100km'}, inplace=True)
df.head()
```

[84]:
```
   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  ...

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

  highway-mpg    price  city-L/100km
0    8.703704  13495.0     11.190476
1    8.703704  16500.0     11.190476
2    9.038462  16500.0     12.368421
3    7.833333  13950.0      9.791667
4   10.681818  17450.0     13.055556

[5 rows x 27 columns]
```

Double-click here for the solution.

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)

[85]:
```
# replace (original value) by (original value)/(maximum value)
df['length'] = df['length']/df['length'].max()
df['width'] = df['width']/df['width'].max()
```

Questiont #3:

According to the example above, normalize the column "height".

[86]:
```
df['height'] = df['height']/df['height'].max()
df[["length","width","height"]].head()
```

[86]:

	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

Double-click here for the solution.

Here we can see, we've normalized "length", "width" and "height" in the range of [0,1].

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

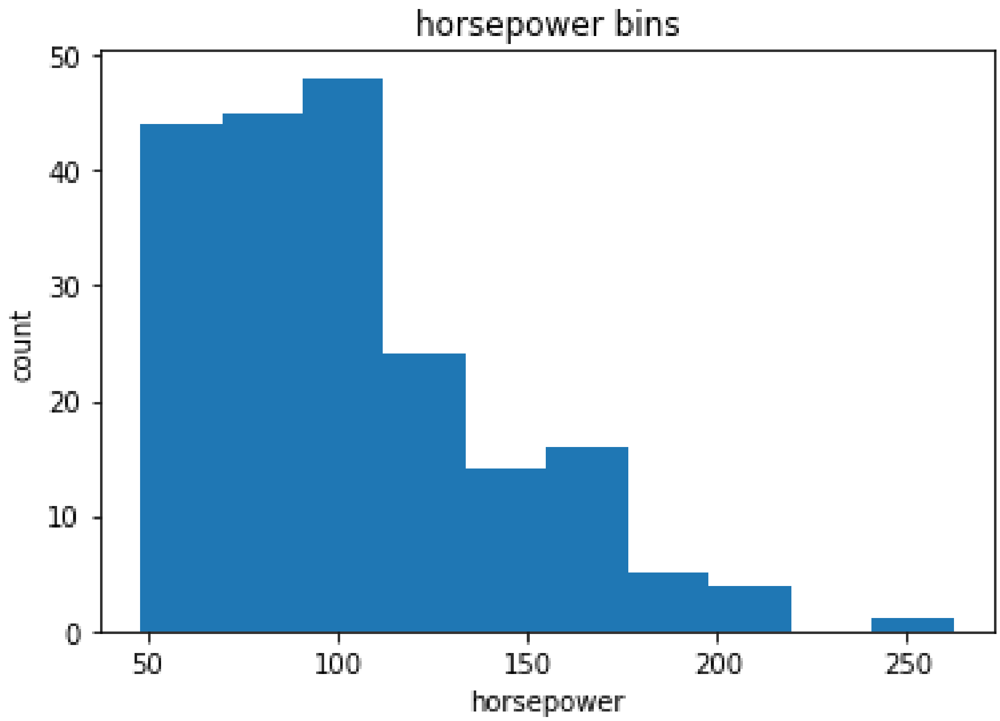
Convert data to correct format

[87]:
```
df["horsepower"]=df["horsepower"].astype(int, copy=True)
```

Lets plot the histogram of horspower, to see what the distribution of horsepower looks like.

[88]:
```
%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")
```

[88]:
```
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.

[89]:
```
bins = np.linspace(min(df["horsepower"]), max(df["horsepower"]), 4)
bins
```

[89]:
```
array([ 48.        , 119.33333333, 190.66666667, 262.        ])
```

We set group names:

[90]:
```
group_names = ['Low', 'Medium', 'High']
```

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

```
[91]: df['horsepower-binned'] = pd.cut(df['horsepower'], bins, labels=group_names,␣
       ↪include_lowest=True )
      df[['horsepower','horsepower-binned']].head(20)
```

```
[91]:     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
```

Lets see the number of vehicles in each bin.

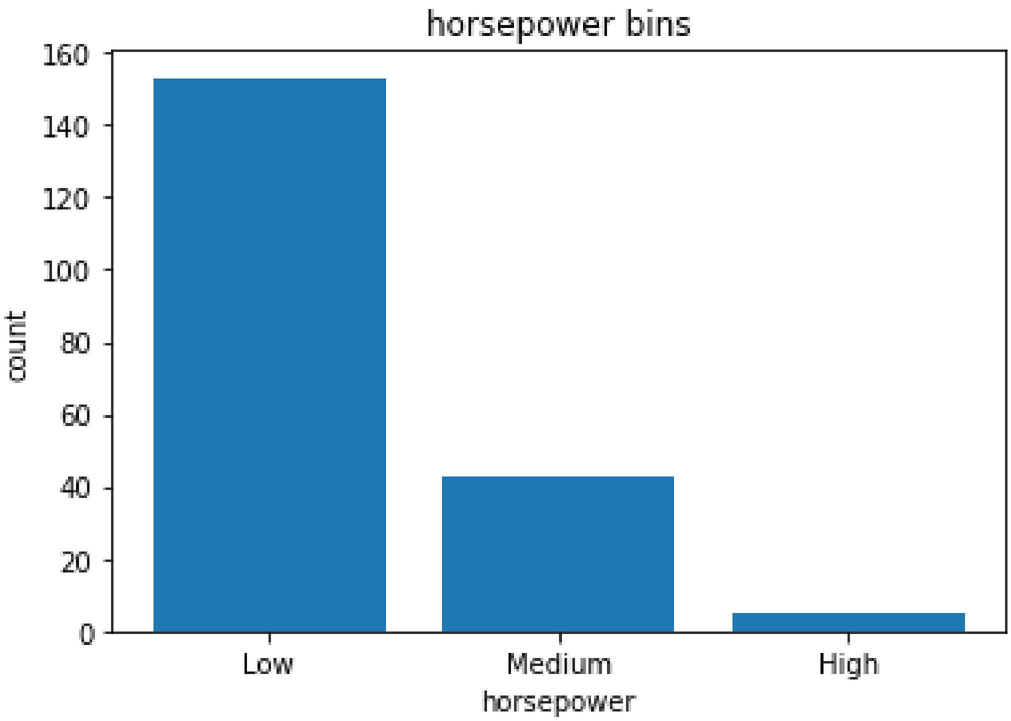
```
[92]: df["horsepower-binned"].value_counts()
```

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

Lets plot the distribution of each bin.

```
[93]: %matplotlib inline
      import matplotlib as plt
      from matplotlib import pyplot
      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")
```

```
[93]: 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").

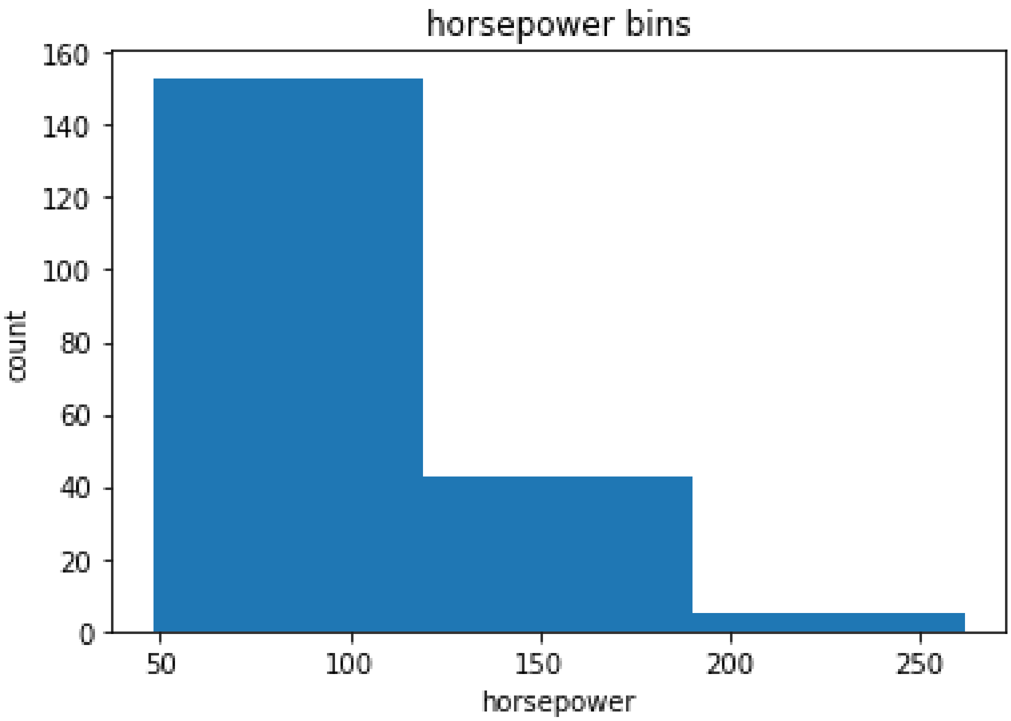
We successfully narrow the intervals from 57 to 3!

Bins visualization

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

[94]:
```
%matplotlib inline
import matplotlib as plt
from matplotlib import pyplot

a = (0,1,2)

# draw historgram of attribute "horsepower" with bins = 3
plt.pyplot.hist(df["horsepower"], bins = 3)

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

[94]:
```
Text(0.5, 1.0, 'horsepower bins')
```



The plot above shows the binning result for attribute "horsepower".

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.

[95]:
```
df.columns
```

[95]:
```
Index(['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', 'city-L/100km', 'horsepower-binned'],
dtype='object')
```

get indicator variables and assign it to data frame "dummy_variable_1"

[96]:
```
dummy_variable_1 = pd.get_dummies(df["fuel-type"])
dummy_variable_1.head()
```

[96]:
```
   diesel  gas
0       0    1
1       0    1
2       0    1
3       0    1
4       0    1
```

change column names for clarity

[97]:
```
dummy_variable_1.rename(columns={'fuel-type-diesel':'gas', 'fuel-type-diesel':
 ↪'diesel'}, inplace=True)
dummy_variable_1.head()
```

[97]:
```
   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.

[98]:
```
# merge data frame "df" and "dummy_variable_1"
df = pd.concat([df, dummy_variable_1], axis=1)

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

[99]:
```
df.head()
```

[99]:
```
   symboling  normalized-losses         make aspiration num-of-doors  \
0          3                122  alfa-romero        std          two
1          3                122  alfa-romero        std          two
2          1                122  alfa-romero        std          two
3          2                164         audi        std         four
4          2                164         audi        std         four

  body-style drive-wheels engine-location  wheel-base    length  …  \
```

```
0  convertible          rwd          front        88.6  0.811148  …
1  convertible          rwd          front        88.6  0.811148  …
2    hatchback          rwd          front        94.5  0.822681  …
3        sedan          fwd          front        99.8  0.848630  …
4        sedan          4wd          front        99.4  0.848630  …

   compression-ratio  horsepower  peak-rpm city-mpg highway-mpg    price  \
0                9.0         111    5000.0       21    8.703704  13495.0
1                9.0         111    5000.0       21    8.703704  16500.0
2                9.0         154    5000.0       19    9.038462  16500.0
3               10.0         102    5500.0       24    7.833333  13950.0
4                8.0         115    5500.0       18   10.681818  17450.0

  city-L/100km  horsepower-binned  diesel  gas
0    11.190476                Low       0    1
1    11.190476                Low       0    1
2    12.368421             Medium       0    1
3     9.791667                Low       0    1
4    13.055556                Low       0    1

[5 rows x 29 columns]
```

The last two columns are now the indicator variable representation of the fuel-type variable. It's all 0s and 1s now.

Question #4:

As above, create indicator variable to the column of "aspiration": "std" to 0, while "turbo" to 1.

[100]:
```
dummy_variable_2 = pd.get_dummies(df['aspiration'])
dummy_variable_2.rename(columns={'std':'aspiration-std', 'turbo':
 ↪'aspiration-turbo'}, inplace=True)
dummy_variable_2.head()
```

[100]:
```
   aspiration-std  aspiration-turbo
0               1                 0
1               1                 0
2               1                 0
3               1                 0
4               1                 0
```

Double-click here for the solution.

Question #5:

Merge the new dataframe to the original dataframe then drop the column 'aspiration'

[101]:
```
df = pd.concat([df, dummy_variable_2], axis=1)
df.drop('aspiration', axis = 1, inplace=True)
```

Double-click here for the solution.

save the new csv

[102]:
```
df.to_csv('clean_df.csv')
```

```
<p><a href="https://cocl.us/corsera_da0101en_notebook_bottom"><img src="https://s3-api.us-geo.c
```

About the Authors:

This notebook was written by Mahdi Noorian PhD, Joseph Santarcangelo, Bahare Talayian, Eric Xiao, Steven Dong, Parizad, Hima Vsudevan and Fiorella Wenver and Yi Yao.

Joseph Santarcangelo is a Data Scientist at IBM, and holds a PhD in Electrical Engineering. His research focused on using Machine Learning, Signal Processing, and Computer Vision to determine how videos impact human cognition. Joseph has been working for IBM since he completed his PhD.

exploratory-data-analysis

June 21, 2020

Exploratory Data Analysis

Welcome!

In this section, we will explore several methods to see if certain characteristics or features can be used to predict car price.

What are the main characteristics which have the most impact on the car price?

1. Import Data from Module 2

Setup

Import libraries

[55]:
```
import pandas as pd
import numpy as np
```

load data and store in dataframe df:

This dataset was hosted on IBM Cloud object click HERE for free storage

[56]:
```
path='https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/
↪CognitiveClass/DA0101EN/automobileEDA.csv'
df = pd.read_csv(path)
df.head()
```

[56]:
```
   symboling  normalized-losses         make aspiration num-of-doors  \
0          3                122  alfa-romero        std          two
1          3                122  alfa-romero        std          two
2          1                122  alfa-romero        std          two
3          2                164         audi        std         four
4          2                164         audi        std         four

    body-style drive-wheels engine-location  wheel-base    length  …  \
0  convertible          rwd           front        88.6  0.811148  …
1  convertible          rwd           front        88.6  0.811148  …
2    hatchback          rwd           front        94.5  0.822681  …
3        sedan          fwd           front        99.8  0.848630  …
4        sedan          4wd           front        99.4  0.848630  …

   compression-ratio  horsepower  peak-rpm city-mpg highway-mpg    price  \
```

```
0               9.0     111.0   5000.0     21           27  13495.0
1               9.0     111.0   5000.0     21           27  16500.0
2               9.0     154.0   5000.0     19           26  16500.0
3              10.0     102.0   5500.0     24           30  13950.0
4               8.0     115.0   5500.0     18           22  17450.0

  city-L/100km  horsepower-binned  diesel  gas
0    11.190476             Medium       0    1
1    11.190476             Medium       0    1
2    12.368421             Medium       0    1
3     9.791667             Medium       0    1
4    13.055556             Medium       0    1

[5 rows x 29 columns]
```

2. Analyzing Individual Feature Patterns using Visualization

To install seaborn we use the pip which is the python package manager.

[57]:
```
%%capture
! pip install seaborn
```

Import visualization packages "Matplotlib" and "Seaborn", don't forget about "%matplotlib inline" to plot in a Jupyter notebook.

[58]:
```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

How to choose the right visualization method?

When visualizing individual variables, it is important to first understand what type of variable you are dealing with. This will help us find the right visualization method for that variable.

[59]:
```
# list the data types for each column
print(df.dtypes)
```

```
symboling              int64
normalized-losses      int64
make                  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             float64
peak-rpm               float64
city-mpg                 int64
highway-mpg              int64
price                  float64
city-L/100km           float64
horsepower-binned       object
diesel                   int64
gas                      int64
dtype: object
```

Question #1:

What is the data type of the column "peak-rpm"?

Double-click here for the solution.

for example, we can calculate the correlation between variables of type "int64" or "float64" using the method "corr":

[60]:
```
df.corr()
```

[60]:

	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	-0.140019	-0.029862	0.493244	0.608971	
stroke	-0.008245	0.055563	0.158502	0.124139	
compression-ratio	-0.182196	-0.114713	0.250313	0.159733	
horsepower	0.075819	0.217299	0.371147	0.579821	
peak-rpm	0.279740	0.239543	-0.360305	-0.285970	
city-mpg	-0.035527	-0.225016	-0.470606	-0.665192	
highway-mpg	0.036233	-0.181877	-0.543304	-0.698142	
price	-0.082391	0.133999	0.584642	0.690628	
city-L/100km	0.066171	0.238567	0.476153	0.657373	
diesel	-0.196735	-0.101546	0.307237	0.211187	
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	-0.140019
normalized-losses	0.086802	-0.373737	0.099404	0.112360	-0.029862
wheel-base	0.814507	0.590742	0.782097	0.572027	0.493244
length	0.857170	0.492063	0.880665	0.685025	0.608971
width	1.000000	0.306002	0.866201	0.729436	0.544885
height	0.306002	1.000000	0.307581	0.074694	0.180449
curb-weight	0.866201	0.307581	1.000000	0.849072	0.644060
engine-size	0.729436	0.074694	0.849072	1.000000	0.572609
bore	0.544885	0.180449	0.644060	0.572609	1.000000
stroke	0.188829	-0.062704	0.167562	0.209523	-0.055390
compression-ratio	0.189867	0.259737	0.156433	0.028889	0.001263
horsepower	0.615077	-0.087027	0.757976	0.822676	0.566936
peak-rpm	-0.245800	-0.309974	-0.279361	-0.256733	-0.267392
city-mpg	-0.633531	-0.049800	-0.749543	-0.650546	-0.582027
highway-mpg	-0.680635	-0.104812	-0.794889	-0.679571	-0.591309
price	0.751265	0.135486	0.834415	0.872335	0.543155
city-L/100km	0.673363	0.003811	0.785353	0.745059	0.554610
diesel	0.244356	0.281578	0.221046	0.070779	0.054458
gas	-0.244356	-0.281578	-0.221046	-0.070779	-0.054458

	stroke	compression-ratio	horsepower	peak-rpm \
symboling	-0.008245	-0.182196	0.075819	0.279740
normalized-losses	0.055563	-0.114713	0.217299	0.239543
wheel-base	0.158502	0.250313	0.371147	-0.360305
length	0.124139	0.159733	0.579821	-0.285970
width	0.188829	0.189867	0.615077	-0.245800
height	-0.062704	0.259737	-0.087027	-0.309974
curb-weight	0.167562	0.156433	0.757976	-0.279361
engine-size	0.209523	0.028889	0.822676	-0.256733
bore	-0.055390	0.001263	0.566936	-0.267392
stroke	1.000000	0.187923	0.098462	-0.065713
compression-ratio	0.187923	1.000000	-0.214514	-0.435780
horsepower	0.098462	-0.214514	1.000000	0.107885
peak-rpm	-0.065713	-0.435780	0.107885	1.000000
city-mpg	-0.034696	0.331425	-0.822214	-0.115413
highway-mpg	-0.035201	0.268465	-0.804575	-0.058598
price	0.082310	0.071107	0.809575	-0.101616
city-L/100km	0.037300	-0.299372	0.889488	0.115830
diesel	0.241303	0.985231	-0.169053	-0.475812
gas	-0.241303	-0.985231	0.169053	0.475812

	city-mpg	highway-mpg	price	city-L/100km	diesel \
symboling	-0.035527	0.036233	-0.082391	0.066171	-0.196735
normalized-losses	-0.225016	-0.181877	0.133999	0.238567	-0.101546
wheel-base	-0.470606	-0.543304	0.584642	0.476153	0.307237

```
length            -0.665192    -0.698142  0.690628      0.657373  0.211187
width             -0.633531    -0.680635  0.751265      0.673363  0.244356
height            -0.049800    -0.104812  0.135486      0.003811  0.281578
curb-weight       -0.749543    -0.794889  0.834415      0.785353  0.221046
engine-size       -0.650546    -0.679571  0.872335      0.745059  0.070779
bore              -0.582027    -0.591309  0.543155      0.554610  0.054458
stroke            -0.034696    -0.035201  0.082310      0.037300  0.241303
compression-ratio  0.331425     0.268465  0.071107     -0.299372  0.985231
horsepower        -0.822214    -0.804575  0.809575      0.889488 -0.169053
peak-rpm          -0.115413    -0.058598 -0.101616      0.115830 -0.475812
city-mpg           1.000000     0.972044 -0.686571     -0.949713  0.265676
highway-mpg        0.972044     1.000000 -0.704692     -0.930028  0.198690
price             -0.686571    -0.704692  1.000000      0.789898  0.110326
city-L/100km      -0.949713    -0.930028  0.789898      1.000000 -0.241282
diesel             0.265676     0.198690  0.110326     -0.241282  1.000000
gas               -0.265676    -0.198690 -0.110326      0.241282 -1.000000

                        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              -0.054458
stroke            -0.241303
compression-ratio -0.985231
horsepower         0.169053
peak-rpm           0.475812
city-mpg          -0.265676
highway-mpg       -0.198690
price             -0.110326
city-L/100km       0.241282
diesel            -1.000000
gas                1.000000
```

The diagonal elements are always one; we will study correlation more precisely Pearson correlation in-depth at the end of the notebook.

Question #2:

Find the correlation between the following columns: bore, stroke,compression-ratio , and horsepower.

Hint: if you would like to select those columns use the following syntax: df[['bore','stroke' ,'compression-ratio','horsepower']]

[61]:
```python
df[['bore', 'stroke', 'compression-ratio', 'horsepower']].corr()
```

[61]:

	bore	stroke	compression-ratio	horsepower
bore	1.000000	-0.055390	0.001263	0.566936
stroke	-0.055390	1.000000	0.187923	0.098462
compression-ratio	0.001263	0.187923	1.000000	-0.214514
horsepower	0.566936	0.098462	-0.214514	1.000000

Double-click here for the solution.

Continuous numerical variables:

Continuous numerical variables are variables that may contain any value within some range. Continuous numerical variables can have the type "int64" or "float64". A great way to visualize these variables is by using scatterplots with fitted lines.

In order to start understanding the (linear) relationship between an individual variable and the price. We can do this by using "regplot", which plots the scatterplot plus the fitted regression line for the data.

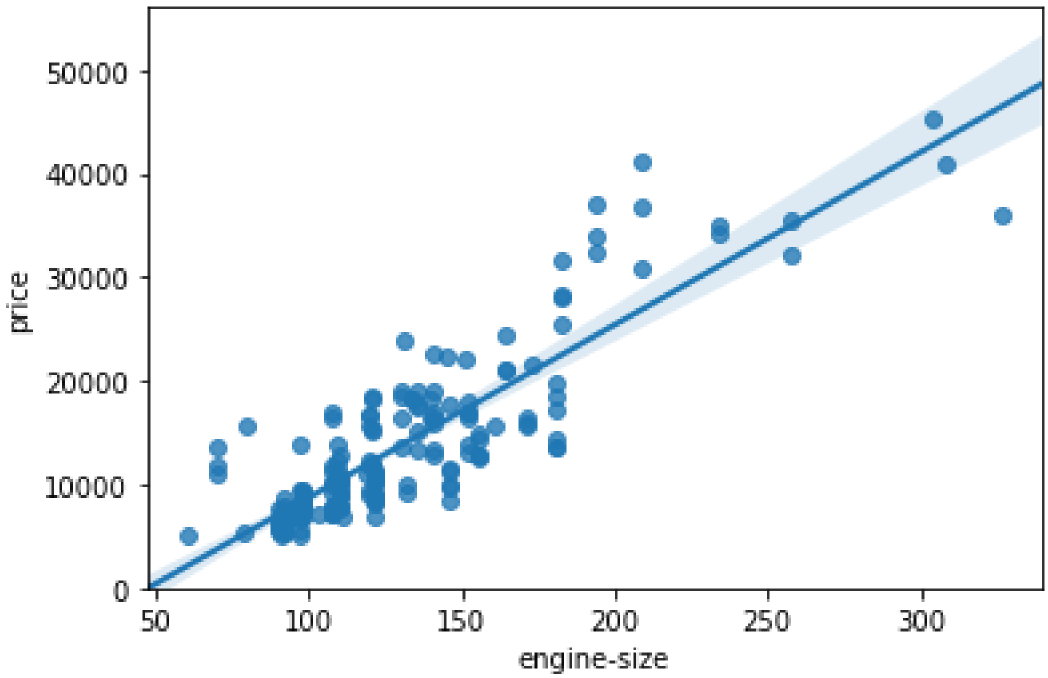
Let's see several examples of different linear relationships:

Positive linear relationship

Let's find the scatterplot of "engine-size" and "price"

[62]:
```python
# Engine size as potential predictor variable of price
sns.regplot(x="engine-size", y="price", data=df)
plt.ylim(0,)
```

[62]: (0, 56053.02835800785)



As the engine-size goes up, the price goes up: this indicates a positive direct correlation between these two variables. Engine size seems like a pretty good predictor of price since the regression line is almost a perfect diagonal line.

We can examine the correlation between 'engine-size' and 'price' and see it's approximately 0.87

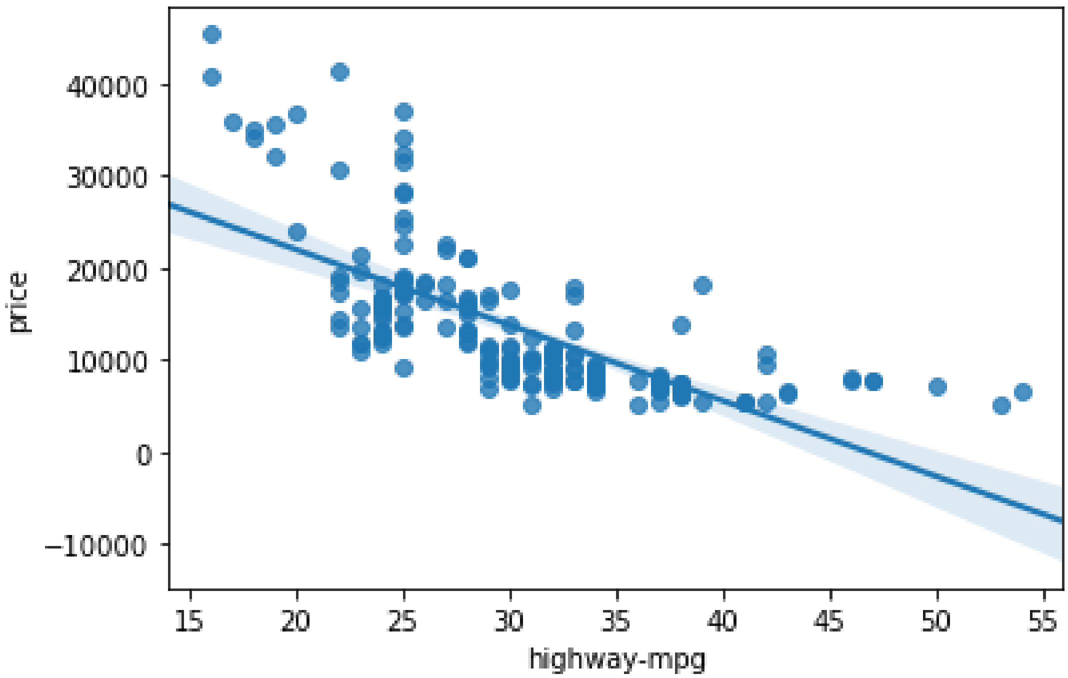
[63]:
```
df[["engine-size", "price"]].corr()
```

[63]:
```
             engine-size     price
engine-size     1.000000  0.872335
price           0.872335  1.000000
```

Highway mpg is a potential predictor variable of price

[64]:
```
sns.regplot(x="highway-mpg", y="price", data=df)
```

[64]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fb5a5b908>
```



As the highway-mpg goes up, the price goes down: this indicates an inverse/negative relationship between these two variables. Highway mpg could potentially be a predictor of price.

We can examine the correlation between 'highway-mpg' and 'price' and see it's approximately -0.704

[65]:
```
df[['highway-mpg', 'price']].corr()
```

[65]:

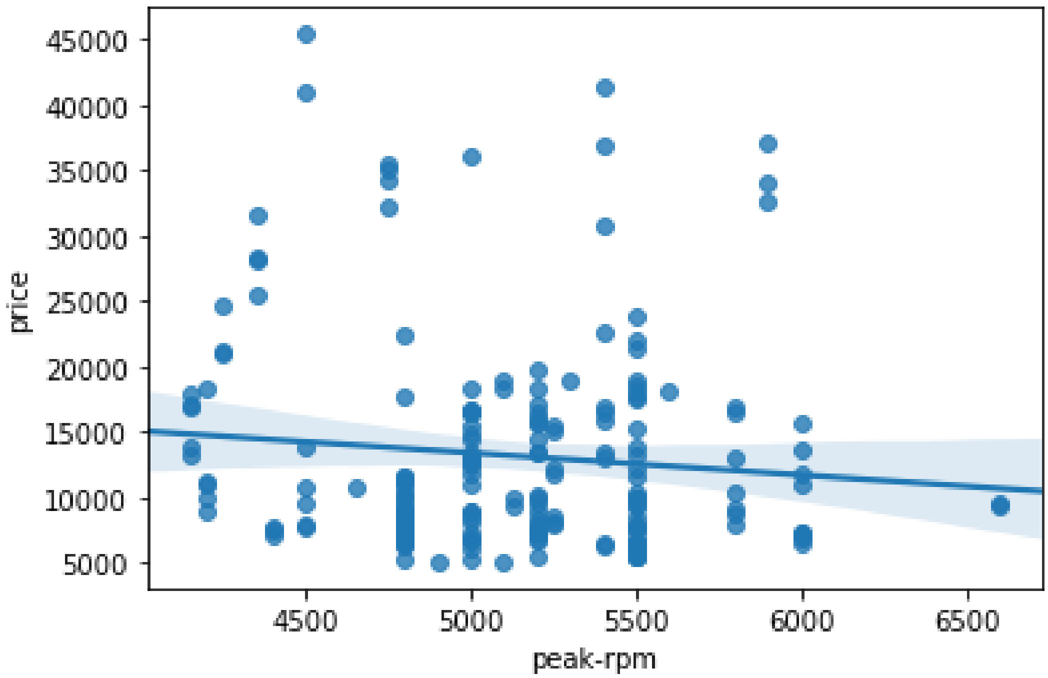
	highway-mpg	price
highway-mpg	1.000000	-0.704692
price	-0.704692	1.000000

Weak Linear Relationship

Let's see if "Peak-rpm" as a predictor variable of "price".

[66]:
```
sns.regplot(x="peak-rpm", y="price", data=df)
```

[66]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fb59cc2e8>
```



Peak rpm does not seem like a good predictor of the price at all since the regression line is close to horizontal. Also, the data points are very scattered and far from the fitted line, showing lots of variability. Therefore it's it is not a reliable variable.

We can examine the correlation between 'peak-rpm' and 'price' and see it's approximately -0.101616

[67]:
```
df[['peak-rpm','price']].corr()
```

[67]:

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

Question 3 a):

Find the correlation between x="stroke", y="price".

Hint: if you would like to select those columns use the following syntax: df[["stroke","price"]]

[68]:
```
df[["stroke","price"]].corr()
```

[68]:

	stroke	price
stroke	1.00000	0.08231
price	0.08231	1.00000

Double-click here for the solution.

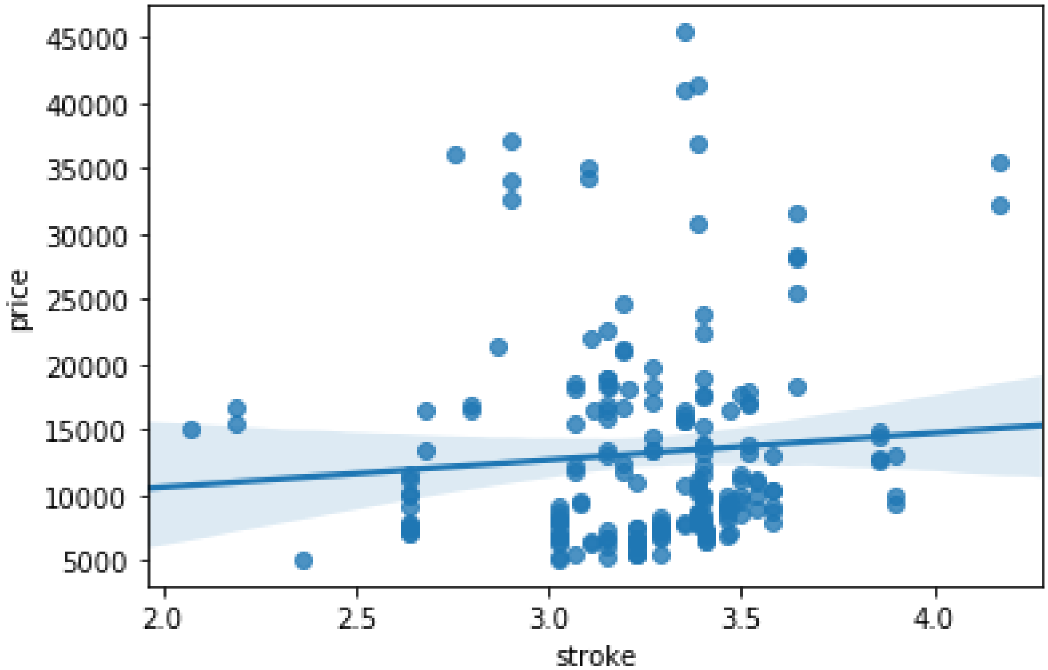
Question 3 b):

Given the correlation results between “price” and “stroke” do you expect a linear relationship?

Verify your results using the function “regplot()”.

[69]:
```
sns.regplot(x="stroke", y="price", data=df)
```

[69]: <matplotlib.axes._subplots.AxesSubplot at 0x7f4fb593cda0>



Double-click here for the solution.

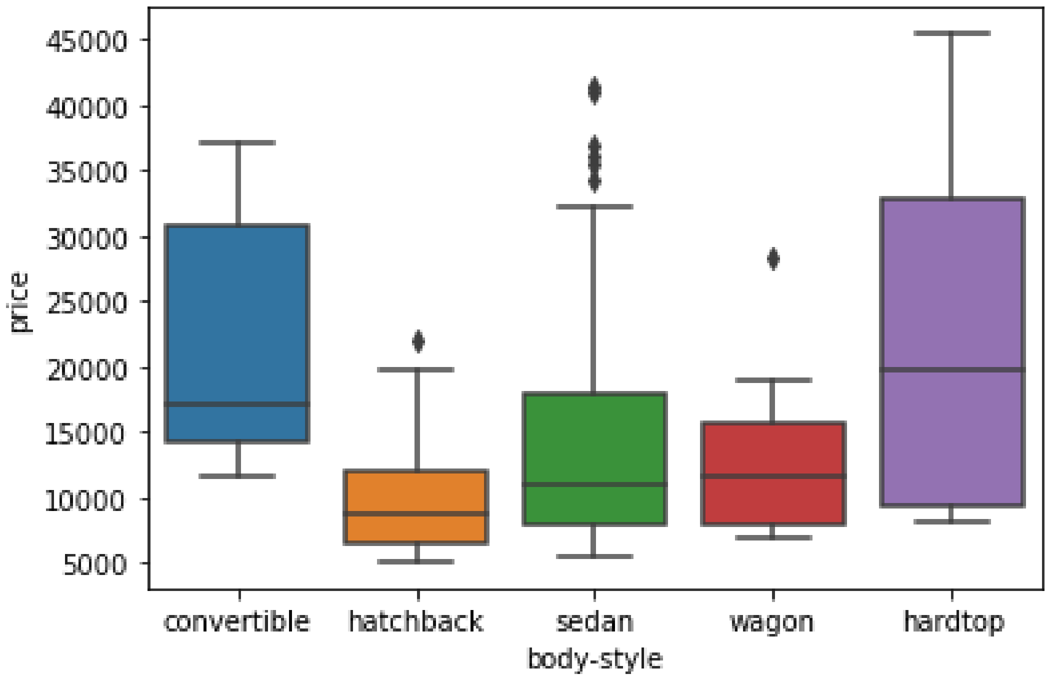
Categorical variables

These are variables that describe a ‘characteristic’ of a data unit, and are selected from a small group of categories. The categorical variables can have the type “object” or “int64”. A good way to visualize categorical variables is by using boxplots.

Let’s look at the relationship between “body-style” and “price”.

[70]:
```
sns.boxplot(x="body-style", y="price", data=df)
```

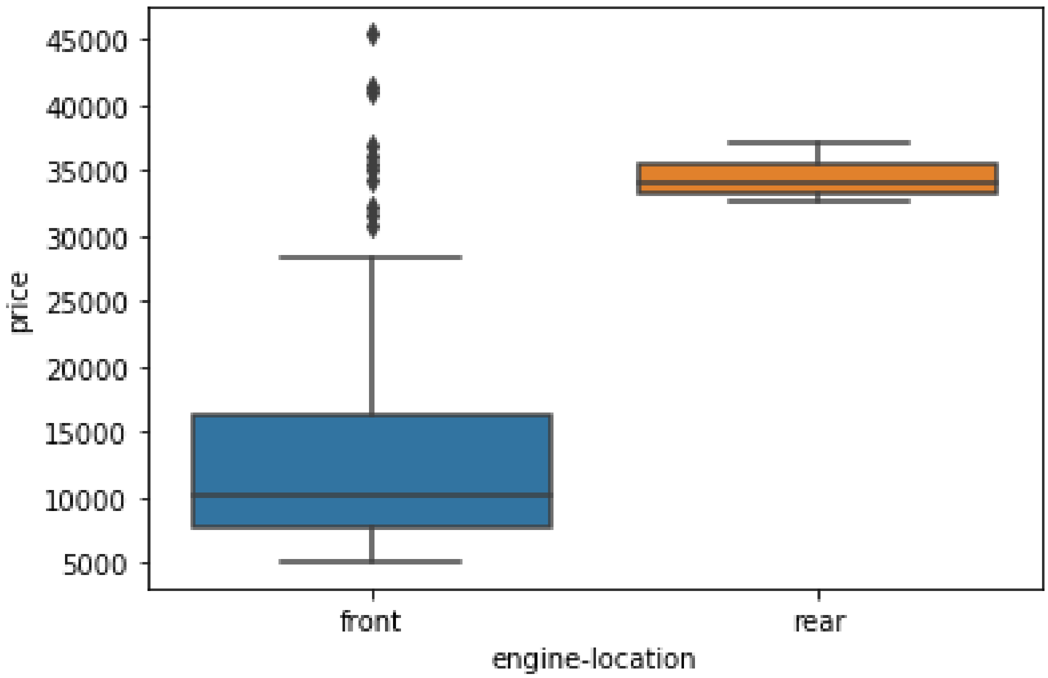
[70]: <matplotlib.axes._subplots.AxesSubplot at 0x7f4fb58b46d8>



We see that the distributions of price between the different body-style categories have a significant overlap, and so body-style would not be a good predictor of price. Let's examine engine "engine-location" and "price":

[71]:
```
sns.boxplot(x="engine-location", y="price", data=df)
```

[71]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4fb5919358>
```

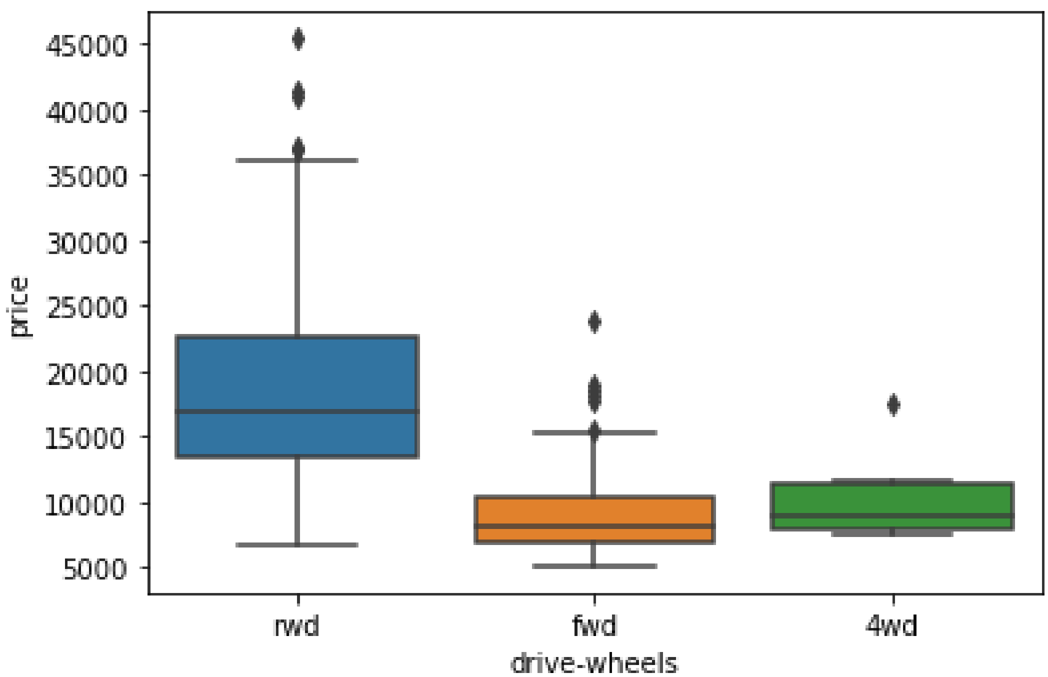


Here we see that the distribution of price between these two engine-location categories, front and rear, are distinct enough to take engine-location as a potential good predictor of price.

Let's examine "drive-wheels" and "price".

[72]:
```
# drive-wheels
sns.boxplot(x="drive-wheels", y="price", data=df)
```

[72]: <matplotlib.axes._subplots.AxesSubplot at 0x7f4f842aeba8>



Here we see that the distribution of price between the different drive-wheels categories differs; as such drive-wheels could potentially be a predictor of price.

3. Descriptive Statistical Analysis

Let's first take a look at the variables by utilizing a description method.

The describe function automatically computes basic statistics for all continuous variables. Any NaN values are automatically skipped in these statistics.

This will show:

the count of that variable

the mean

the standard deviation (std)

the minimum value

the IQR (Interquartile Range: 25%, 50% and 75%)

the maximum value

We can apply the method "describe" as follows:

[73]:
```
df.describe()
```

[73]:
```
        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  197.000000
mean    53.766667  2555.666667   126.875622    3.330692    3.256904
std      2.447822   517.296727    41.546834    0.268072    0.319256
min     47.800000  1488.000000    61.000000    2.540000    2.070000
25%     52.000000  2169.000000    98.000000    3.150000    3.110000
50%     54.100000  2414.000000   120.000000    3.310000    3.290000
75%     55.500000  2926.000000   141.000000    3.580000    3.410000
max     59.800000  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  103.405534  5117.665368   25.179104    30.686567
std             4.004965   37.365700   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      diesel         gas
count    201.000000    201.000000  201.000000  201.000000
mean   13207.129353      9.944145    0.099502    0.900498
std     7947.066342      2.534599    0.300083    0.300083
min     5118.000000      4.795918    0.000000    0.000000
25%     7775.000000      7.833333    0.000000    1.000000
50%    10295.000000      9.791667    0.000000    1.000000
75%    16500.000000     12.368421    0.000000    1.000000
max    45400.000000     18.076923    1.000000    1.000000
```

The default setting of "describe" skips variables of type object. We can apply the method "describe" on the variables of type 'object' as follows:

[74]:
```
df.describe(include=['object'])
```

[74]:
```
          make aspiration num-of-doors body-style drive-wheels  \
count      201        201          201        201          201
unique      22          2            2          5            3
top     toyota        std         four      sedan          fwd
```

```
freq        32        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

       horsepower-binned
count                200
unique                 3
top                  Low
freq                 115
```

Value Counts

Value-counts is a good way of understanding how many units of each characteristic/variable we have. We can apply the "value_counts" method on the column 'drive-wheels'. Don't forget the method "value_counts" only works on Pandas series, not Pandas Dataframes. As a result, we only include one bracket "df['drive-wheels']" not two brackets "df[['drive-wheels']]".

[75]:
```
df['drive-wheels'].value_counts()
```

[75]:
```
fwd    118
rwd     75
4wd      8
Name: drive-wheels, dtype: int64
```

We can convert the series to a Dataframe as follows :

[76]:
```
df['drive-wheels'].value_counts().to_frame()
```

[76]:
```
     drive-wheels
fwd           118
rwd            75
4wd             8
```

Let's repeat the above steps but save the results to the dataframe "drive_wheels_counts" and rename the column 'drive-wheels' to 'value_counts'.

[77]:
```
drive_wheels_counts = df['drive-wheels'].value_counts().to_frame()
drive_wheels_counts.rename(columns={'drive-wheels': 'value_counts'},␣
 ↪inplace=True)
drive_wheels_counts
```

[77]:
```
     value_counts
fwd           118
rwd            75
4wd             8
```

Now let's rename the index to 'drive-wheels':

[78]:
```
drive_wheels_counts.index.name = 'drive-wheels'
drive_wheels_counts
```

[78]:
```
              value_counts
drive-wheels
fwd                    118
rwd                     75
4wd                      8
```

We can repeat the above process for the variable 'engine-location'.

[79]:
```
# engine-location as variable
engine_loc_counts = df['engine-location'].value_counts().to_frame()
engine_loc_counts.rename(columns={'engine-location': 'value_counts'},␣
 ↪inplace=True)
engine_loc_counts.index.name = 'engine-location'
engine_loc_counts.head(10)
```

[79]:
```
                 value_counts
engine-location
front                     198
rear                        3
```

Examining the value counts of the engine location would not be a good predictor variable for the price. This is because we only have three cars with a rear engine and 198 with an engine in the front, this result is skewed. Thus, we are not able to draw any conclusions about the engine location.

4. Basics of Grouping

The "groupby" method groups data by different categories. The data is grouped based on one or several variables and analysis is performed on the individual groups.

For example, let's group by the variable "drive-wheels". We see that there are 3 different categories of drive wheels.

[80]:
```
df['drive-wheels'].unique()
```

[80]:
```
array(['rwd', 'fwd', '4wd'], dtype=object)
```

If we want to know, on average, which type of drive wheel is most valuable, we can group "drive-wheels" and then average them.

We can select the columns 'drive-wheels', 'body-style' and 'price', then assign it to the variable "df_group_one".

[81]:
```
df_group_one = df[['drive-wheels','body-style','price']]
```

We can then calculate the average price for each of the different categories of data.

[82]:
```
# grouping results
df_group_one = df_group_one.groupby(['drive-wheels'],as_index=False).mean()
df_group_one
```

[82]:
```
  drive-wheels         price
0          4wd  10241.000000
1          fwd   9244.779661
2          rwd  19757.613333
```

From our data, it seems rear-wheel drive vehicles are, on average, the most expensive, while 4-wheel and front-wheel are approximately the same in price.

You can also group with multiple variables. For example, let's group by both 'drive-wheels' and 'body-style'. This groups the dataframe by the unique combinations 'drive-wheels' and 'body-style'. We can store the results in the variable 'grouped_test1'.

[83]:
```
# grouping results
df_gptest = df[['drive-wheels','body-style','price']]
grouped_test1 = df_gptest.groupby(['drive-wheels','body-style'],as_index=False).
↪mean()
grouped_test1
```

[83]:
```
   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
```

This grouped data is much easier to visualize when it is made into a pivot table. A pivot table is like an Excel spreadsheet, with one variable along the column and another along the row. We can convert the dataframe to a pivot table using the method "pivot" to create a pivot table from the groups.

In this case, we will leave the drive-wheel variable as the rows of the table, and pivot body-style to become the columns of the table:

[84]:
```
grouped_pivot = grouped_test1.pivot(index='drive-wheels',columns='body-style')
grouped_pivot
```

[84]:
```
                    price                                               \
body-style    convertible        hardtop     hatchback         sedan
drive-wheels
4wd                   NaN            NaN   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
```

Often, we won't have data for some of the pivot cells. We can fill these missing cells with the value 0, but any other value could potentially be used as well. It should be mentioned that missing data is quite a complex subject and is an entire course on its own.

[85]:
```
grouped_pivot = grouped_pivot.fillna(0) #fill missing values with 0
grouped_pivot
```

[85]:
```
                    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
```

Question 4:

Use the "groupby" function to find the average "price" of each car based on "body-style" ?

[86]:
```
df_gptest2 = df[['body-style','price']]
grouped_test_bodystyle = df_gptest2.groupby(['body-style'],as_index= False).
 ↪mean()
grouped_test_bodystyle
```

[86]:
```
    body-style         price
0  convertible  21890.500000
1      hardtop  22208.500000
2    hatchback   9957.441176
```

```
3        sedan  14459.755319
4        wagon  12371.960000
```

Double-click here for the solution.

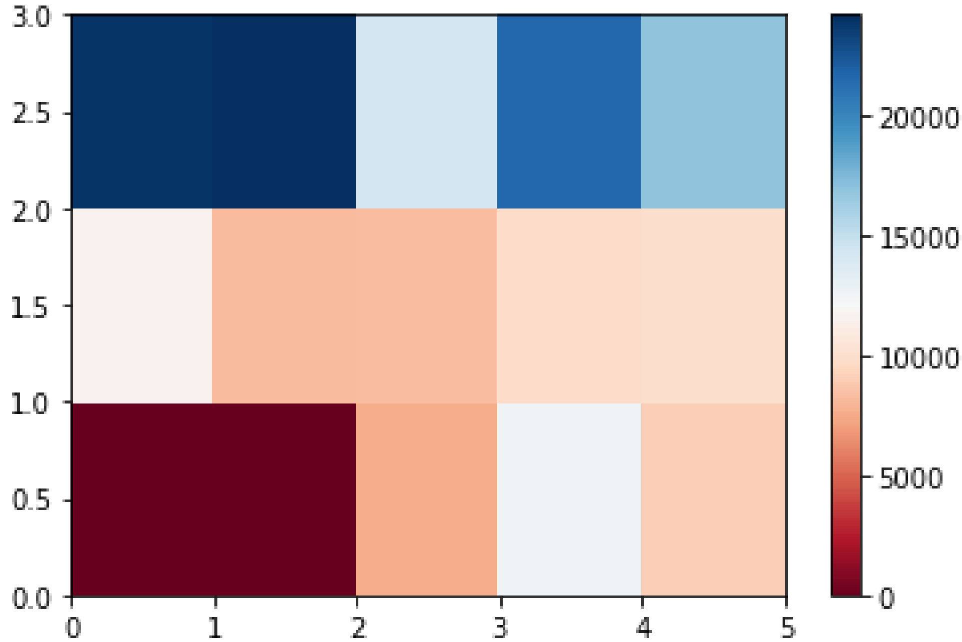
If you did not import "pyplot" let's do it again.

[87]:
```python
import matplotlib.pyplot as plt
%matplotlib inline
```

Variables: Drive Wheels and Body Style vs Price

Let's use a heat map to visualize the relationship between Body Style vs Price.

[88]:
```python
#use the grouped results
plt.pcolor(grouped_pivot, cmap='RdBu')
plt.colorbar()
plt.show()
```

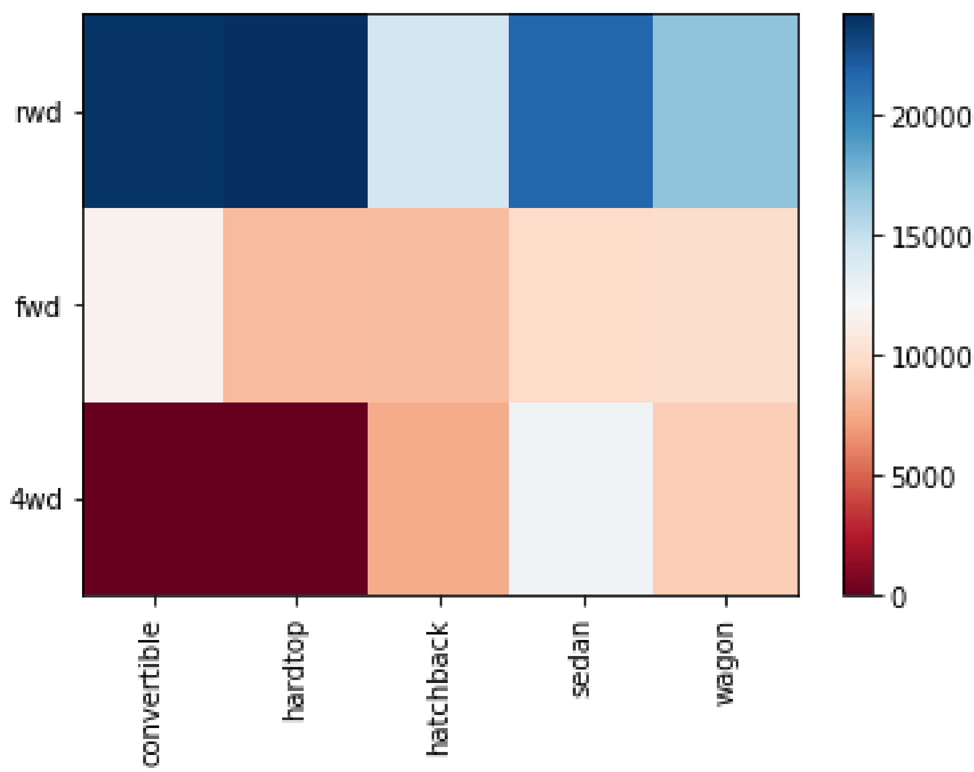


The heatmap plots the target variable (price) proportional to colour with respect to the variables 'drive-wheel' and 'body-style' in the vertical and horizontal axis respectively. This allows us to visualize how the price is related to 'drive-wheel' and 'body-style'.

The default labels convey no useful information to us. Let's change that:

[89]:
```python
fig, ax = plt.subplots()
im = ax.pcolor(grouped_pivot, cmap='RdBu')
```

```
#label names
row_labels = grouped_pivot.columns.levels[1]
col_labels = grouped_pivot.index

#move ticks and labels to the center
ax.set_xticks(np.arange(grouped_pivot.shape[1]) + 0.5, minor=False)
ax.set_yticks(np.arange(grouped_pivot.shape[0]) + 0.5, minor=False)

#insert labels
ax.set_xticklabels(row_labels, minor=False)
ax.set_yticklabels(col_labels, minor=False)

#rotate label if too long
plt.xticks(rotation=90)

fig.colorbar(im)
plt.show()
```



Visualization is very important in data science, and Python visualization packages provide great freedom. We will go more in-depth in a separate Python Visualizations course.

The main question we want to answer in this module, is "What are the main characteristics which have the most impact on the car price?".

To get a better measure of the important characteristics, we look at the correlation of these variables with the car price, in other words: how is the car price dependent on this variable?

5. Correlation and Causation

Correlation: a measure of the extent of interdependence between variables.

Causation: the relationship between cause and effect between two variables.

It is important to know the difference between these two and that correlation does not imply causation. Determining correlation is much simpler the determining causation as causation may require independent experimentation.

Pearson Correlation

The Pearson Correlation measures the linear dependence between two variables X and Y.

The resulting coefficient is a value between -1 and 1 inclusive, where:

1: Total positive linear correlation.

0: No linear correlation, the two variables most likely do not affect each other.

-1: Total negative linear correlation.

Pearson Correlation is the default method of the function "corr". Like before we can calculate the Pearson Correlation of the of the 'int64' or 'float64' variables.

[90]:
```
df.corr()
```

[90]:
```
                   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               -0.140019          -0.029862    0.493244  0.608971
stroke             -0.008245           0.055563    0.158502  0.124139
compression-ratio  -0.182196          -0.114713    0.250313  0.159733
horsepower          0.075819           0.217299    0.371147  0.579821
peak-rpm            0.279740           0.239543   -0.360305 -0.285970
city-mpg           -0.035527          -0.225016   -0.470606 -0.665192
highway-mpg         0.036233          -0.181877   -0.543304 -0.698142
price              -0.082391           0.133999    0.584642  0.690628
city-L/100km        0.066171           0.238567    0.476153  0.657373
diesel             -0.196735          -0.101546    0.307237  0.211187
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 -0.140019
```

```
normalized-losses  0.086802 -0.373737     0.099404     0.112360 -0.029862
wheel-base         0.814507  0.590742     0.782097     0.572027  0.493244
length             0.857170  0.492063     0.880665     0.685025  0.608971
width              1.000000  0.306002     0.866201     0.729436  0.544885
height             0.306002  1.000000     0.307581     0.074694  0.180449
curb-weight        0.866201  0.307581     1.000000     0.849072  0.644060
engine-size        0.729436  0.074694     0.849072     1.000000  0.572609
bore               0.544885  0.180449     0.644060     0.572609  1.000000
stroke             0.188829 -0.062704     0.167562     0.209523 -0.055390
compression-ratio  0.189867  0.259737     0.156433     0.028889  0.001263
horsepower         0.615077 -0.087027     0.757976     0.822676  0.566936
peak-rpm          -0.245800 -0.309974    -0.279361    -0.256733 -0.267392
city-mpg          -0.633531 -0.049800    -0.749543    -0.650546 -0.582027
highway-mpg       -0.680635 -0.104812    -0.794889    -0.679571 -0.591309
price              0.751265  0.135486     0.834415     0.872335  0.543155
city-L/100km       0.673363  0.003811     0.785353     0.745059  0.554610
diesel             0.244356  0.281578     0.221046     0.070779  0.054458
gas               -0.244356 -0.281578    -0.221046    -0.070779 -0.054458

                     stroke  compression-ratio  horsepower  peak-rpm  \
symboling         -0.008245          -0.182196    0.075819  0.279740
normalized-losses  0.055563          -0.114713    0.217299  0.239543
wheel-base         0.158502           0.250313    0.371147 -0.360305
length             0.124139           0.159733    0.579821 -0.285970
width              0.188829           0.189867    0.615077 -0.245800
height            -0.062704           0.259737   -0.087027 -0.309974
curb-weight        0.167562           0.156433    0.757976 -0.279361
engine-size        0.209523           0.028889    0.822676 -0.256733
bore              -0.055390           0.001263    0.566936 -0.267392
stroke             1.000000           0.187923    0.098462 -0.065713
compression-ratio  0.187923           1.000000   -0.214514 -0.435780
horsepower         0.098462          -0.214514    1.000000  0.107885
peak-rpm          -0.065713          -0.435780    0.107885  1.000000
city-mpg          -0.034696           0.331425   -0.822214 -0.115413
highway-mpg       -0.035201           0.268465   -0.804575 -0.058598
price              0.082310           0.071107    0.809575 -0.101616
city-L/100km       0.037300          -0.299372    0.889488  0.115830
diesel             0.241303           0.985231   -0.169053 -0.475812
gas               -0.241303          -0.985231    0.169053  0.475812

                   city-mpg  highway-mpg     price  city-L/100km    diesel  \
symboling         -0.035527     0.036233 -0.082391      0.066171 -0.196735
normalized-losses -0.225016    -0.181877  0.133999      0.238567 -0.101546
wheel-base        -0.470606    -0.543304  0.584642      0.476153  0.307237
length            -0.665192    -0.698142  0.690628      0.657373  0.211187
width             -0.633531    -0.680635  0.751265      0.673363  0.244356
height            -0.049800    -0.104812  0.135486      0.003811  0.281578
```

```
curb-weight       -0.749543    -0.794889  0.834415       0.785353  0.221046
engine-size       -0.650546    -0.679571  0.872335       0.745059  0.070779
bore              -0.582027    -0.591309  0.543155       0.554610  0.054458
stroke            -0.034696    -0.035201  0.082310       0.037300  0.241303
compression-ratio  0.331425     0.268465  0.071107      -0.299372  0.985231
horsepower        -0.822214    -0.804575  0.809575       0.889488 -0.169053
peak-rpm          -0.115413    -0.058598 -0.101616       0.115830 -0.475812
city-mpg           1.000000     0.972044 -0.686571      -0.949713  0.265676
highway-mpg        0.972044     1.000000 -0.704692      -0.930028  0.198690
price             -0.686571    -0.704692  1.000000       0.789898  0.110326
city-L/100km      -0.949713    -0.930028  0.789898       1.000000 -0.241282
diesel             0.265676     0.198690  0.110326      -0.241282  1.000000
gas               -0.265676    -0.198690 -0.110326       0.241282 -1.000000

                        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              -0.054458
stroke            -0.241303
compression-ratio -0.985231
horsepower         0.169053
peak-rpm           0.475812
city-mpg          -0.265676
highway-mpg       -0.198690
price             -0.110326
city-L/100km       0.241282
diesel            -1.000000
gas                1.000000
```

sometimes we would like to know the significant of the correlation estimate.

P-value:

What is this P-value? The P-value is the probability value that the correlation between these two variables is statistically significant. Normally, we choose a significance level of 0.05, which means that we are 95% confident that the correlation between the variables is significant.

By convention, when the

p-value is < 0.001: we say there is strong evidence that the correlation is significant.

the p-value is < 0.05: there is moderate evidence that the correlation is significant.

the p-value is < 0.1: there is weak evidence that the correlation is significant.

the p-value is > 0.1: there is no evidence that the correlation is significant.

We can obtain this information using "stats" module in the "scipy" library.

[91]:
```
from scipy import stats
```

Wheel-base vs Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'wheel-base' and 'price'.

[92]:
```
pearson_coef, p_value = stats.pearsonr(df['wheel-base'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value␣
↪of P =", p_value)
```

```
The Pearson Correlation Coefficient is 0.584641822265508  with a P-value of P =
8.076488270733218e-20
```

Conclusion:

Since the p-value is < 0.001, the correlation between wheel-base and price is statistically significant, although the linear relationship isn't extremely strong (~0.585)

Horsepower vs Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'horsepower' and 'price'.

[93]:
```
pearson_coef, p_value = stats.pearsonr(df['horsepower'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value␣
↪of P = ", p_value)
```

```
The Pearson Correlation Coefficient is 0.8095745670036559  with a P-value of P =
6.369057428260101e-48
```

Conclusion:

Since the p-value is < 0.001, the correlation between horsepower and price is statistically significant, and the linear relationship is quite strong (~0.809, close to 1)

Length vs Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'length' and 'price'.

[94]:
```
pearson_coef, p_value = stats.pearsonr(df['length'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value␣
↪of P = ", p_value)
```

```
The Pearson Correlation Coefficient is 0.6906283804483638  with a P-value of P =
8.016477466159556e-30
```

Conclusion:

Since the p-value is < 0.001, the correlation between length and price is statistically significant, and the linear relationship is moderately strong (~0.691).

Width vs Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'width' and 'price':

[95]:
```
pearson_coef, p_value = stats.pearsonr(df['width'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value␣
↪of P =", p_value )
```

```
The Pearson Correlation Coefficient is 0.7512653440522673  with a P-value of P =
9.200335510481646e-38
```

Conclusion: Since the p-value is < 0.001, the correlation between width and price is statistically significant, and the linear relationship is quite strong (~0.751).

0.0.1 Curb-weight vs Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'curb-weight' and 'price':

[96]:
```
pearson_coef, p_value = stats.pearsonr(df['curb-weight'], df['price'])
print( "The Pearson Correlation Coefficient is", pearson_coef, " with a P-value␣
↪of P = ", p_value)
```

```
The Pearson Correlation Coefficient is 0.8344145257702843  with a P-value of P =
2.189577238894065e-53
```

Conclusion:

Since the p-value is < 0.001, the correlation between curb-weight and price is statistically significant, and the linear relationship is quite strong (~0.834).

Engine-size vs Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'engine-size' and 'price':

[97]:
```
pearson_coef, p_value = stats.pearsonr(df['engine-size'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value␣
↪of P =", p_value)
```

```
The Pearson Correlation Coefficient is 0.8723351674455185  with a P-value of P =
9.265491622198389e-64
```

Conclusion:

Since the p-value is < 0.001, the correlation between engine-size and price is statistically significant, and the linear relationship is very strong (~0.872).

Bore vs Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'bore' and 'price':

[98]:
```
pearson_coef, p_value = stats.pearsonr(df['bore'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value␣
↪of P =  ", p_value )
```

```
The Pearson Correlation Coefficient is 0.5431553832626602  with a P-value of P =
8.049189483935489e-17
```

Conclusion:

Since the p-value is $<$ 0.001, the correlation between bore and price is statistically significant, but the linear relationship is only moderate (~0.521).

We can relate the process for each 'City-mpg' and 'Highway-mpg':

City-mpg vs Price

[99]:
```python
pearson_coef, p_value = stats.pearsonr(df['city-mpg'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value␣
↪of P = ", p_value)
```

```
The Pearson Correlation Coefficient is -0.6865710067844678  with a P-value of P
=  2.321132065567641e-29
```

Conclusion:

Since the p-value is $<$ 0.001, the correlation between city-mpg and price is statistically significant, and the coefficient of ~ -0.687 shows that the relationship is negative and moderately strong.

Highway-mpg vs Price

[100]:
```python
pearson_coef, p_value = stats.pearsonr(df['highway-mpg'], df['price'])
print( "The Pearson Correlation Coefficient is", pearson_coef, " with a P-value␣
↪of P = ", p_value )
```

```
The Pearson Correlation Coefficient is -0.704692265058953  with a P-value of P =
1.7495471144476358e-31
```

Conclusion: Since the p-value is $<$ 0.001, the correlation between highway-mpg and price is statistically significant, and the coefficient of ~ -0.705 shows that the relationship is negative and moderately strong.

6. ANOVA

ANOVA: Analysis of Variance

The Analysis of Variance (ANOVA) is a statistical method used to test whether there are significant differences between the means of two or more groups. ANOVA returns two parameters:

F-test score: ANOVA assumes the means of all groups are the same, calculates how much the actual means deviate from the assumption, and reports it as the F-test score. A larger score means there is a larger difference between the means.

P-value: P-value tells how statistically significant is our calculated score value.

If our price variable is strongly correlated with the variable we are analyzing, expect ANOVA to return a sizeable F-test score and a small p-value.

Drive Wheels

Since ANOVA analyzes the difference between different groups of the same variable, the groupby function will come in handy. Because the ANOVA algorithm averages the data automatically, we do not need to take the average before hand.

Let's see if different types 'drive-wheels' impact 'price', we group the data.

Let's see if different types 'drive-wheels' impact 'price', we group the data.

[101]:
```
grouped_test2=df_gptest[['drive-wheels', 'price']].groupby(['drive-wheels'])
grouped_test2.head(2)
```

[101]:
```
    drive-wheels    price
0            rwd  13495.0
1            rwd  16500.0
3            fwd  13950.0
4            4wd  17450.0
5            fwd  15250.0
136          4wd   7603.0
```

[102]:
```
df_gptest
```

[102]:
```
    drive-wheels   body-style    price
0            rwd  convertible  13495.0
1            rwd  convertible  16500.0
2            rwd    hatchback  16500.0
3            fwd        sedan  13950.0
4            4wd        sedan  17450.0
..           …            …        …
196          rwd        sedan  16845.0
197          rwd        sedan  19045.0
198          rwd        sedan  21485.0
199          rwd        sedan  22470.0
200          rwd        sedan  22625.0

[201 rows x 3 columns]
```

We can obtain the values of the method group using the method "get_group".

[103]:
```
grouped_test2.get_group('4wd')['price']
```

[103]:
```
4      17450.0
136     7603.0
140     9233.0
141    11259.0
144     8013.0
145    11694.0
150     7898.0
151     8778.0
Name: price, dtype: float64
```

we can use the function 'f_oneway' in the module 'stats' to obtain the F-test score and P-value.

[104]:
```
# ANOVA
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( "ANOVA results: F=", f_val, ", P =", p_val)
```

```
ANOVA results: F= 67.95406500780399 , P = 3.3945443577151245e-23
```

This is a great result, with a large F test score showing a strong correlation and a P value of almost 0 implying almost certain statistical significance. But does this mean all three tested groups are all this highly correlated?

Separately: fwd and rwd

[105]:
```
f_val, p_val = stats.f_oneway(grouped_test2.get_group('fwd')['price'],␣
↪grouped_test2.get_group('rwd')['price'])

print( "ANOVA results: F=", f_val, ", P =", p_val )
```

```
ANOVA results: F= 130.5533160959111 , P = 2.2355306355677845e-23
```

Let's examine the other groups

4wd and rwd

[106]:
```
f_val, p_val = stats.f_oneway(grouped_test2.get_group('4wd')['price'],␣
↪grouped_test2.get_group('rwd')['price'])

print( "ANOVA results: F=", f_val, ", P =", p_val)
```

```
ANOVA results: F= 8.580681368924756 , P = 0.004411492211225333
```

4wd and fwd

[107]:
```
f_val, p_val = stats.f_oneway(grouped_test2.get_group('4wd')['price'],␣
↪grouped_test2.get_group('fwd')['price'])

print("ANOVA results: F=", f_val, ", P =", p_val)
```

```
ANOVA results: F= 0.665465750252303 , P = 0.41620116697845666
```

Conclusion: Important Variables

We now have a better idea of what our data looks like and which variables are important to take into account when predicting the car price. We have narrowed it down to the following variables:

Continuous numerical variables:

Length

Width

Curb-weight

Engine-size

Horsepower

City-mpg

Highway-mpg

Wheel-base

Bore

Categorical variables:

Drive-wheels

As we now move into building machine learning models to automate our analysis, feeding the model with variables that meaningfully affect our target variable will improve our model's prediction performance.

```
<p><a href="https://cocl.us/corsera_da0101en_notebook_bottom"><img src="https://s3-api.us-geo.
```

About the Authors:

This notebook was written by Mahdi Noorian PhD, Joseph Santarcangelo, Bahare Talayian, Eric Xiao, Steven Dong, Parizad, Hima Vsudevan and Fiorella Wenver and Yi Yao.

Joseph Santarcangelo is a Data Scientist at IBM, and holds a PhD in Electrical Engineering. His research focused on using Machine Learning, Signal Processing, and Computer Vision to determine how videos impact human cognition. Joseph has been working for IBM since he completed his PhD.

model-evaluation-and-refinement

June 21, 2020

Module 5: Model Evaluation and Refinement

We have built models and made predictions of vehicle prices. Now we will determine how accurate these predictions are.

This dataset was hosted on IBM Cloud object click HERE for free storage.

[1]:
```
import pandas as pd
import numpy as np

# Import clean data
path = 'https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/
↪CognitiveClass/DA0101EN/module_5_auto.csv'
df = pd.read_csv(path)
```

[2]:
```
df.to_csv('module_5_auto.csv')
```

First lets only use numeric data

[3]:
```
df=df._get_numeric_data()
df.head()
```

[3]:
```
   Unnamed: 0  Unnamed: 0.1  symboling  normalized-losses  wheel-base  \
0           0             0          3                122        88.6
1           1             1          3                122        88.6
2           2             2          1                122        94.5
3           3             3          2                164        99.8
4           4             4          2                164        99.4

     length     width  height  curb-weight  engine-size  …  stroke  \
0  0.811148  0.890278    48.8         2548          130  …    2.68
1  0.811148  0.890278    48.8         2548          130  …    2.68
2  0.822681  0.909722    52.4         2823          152  …    3.47
3  0.848630  0.919444    54.3         2337          109  …    3.40
4  0.848630  0.922222    54.3         2824          136  …    3.40

   compression-ratio  horsepower  peak-rpm  city-mpg  highway-mpg    price  \
0                9.0       111.0    5000.0        21           27  13495.0
1                9.0       111.0    5000.0        21           27  16500.0
```

```
2                9.0      154.0    5000.0       19           26  16500.0
3               10.0      102.0    5500.0       24           30  13950.0
4                8.0      115.0    5500.0       18           22  17450.0

   city-L/100km  diesel  gas
0     11.190476       0    1
1     11.190476       0    1
2     12.368421       0    1
3      9.791667       0    1
4     13.055556       0    1

[5 rows x 21 columns]
```

Libraries for plotting

[4]:
```
%%capture
! pip install ipywidgets
```

[5]:
```
from IPython.display import display
from IPython.html import widgets
from IPython.display import display
from ipywidgets import interact, interactive, fixed, interact_manual
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/IPython/html.py:14: ShimWarning: The `IPython.html` package has been
deprecated since IPython 4.0. You should import from `notebook` instead.
`IPython.html.widgets` has moved to `ipywidgets`.
  "`IPython.html.widgets` has moved to `ipywidgets`.", ShimWarning)
```

Functions for plotting

[6]:
```
def DistributionPlot(RedFunction, BlueFunction, RedName, BlueName, Title):
    width = 12
    height = 10
    plt.figure(figsize=(width, height))

    ax1 = sns.distplot(RedFunction, hist=False, color="r", label=RedName)
    ax2 = sns.distplot(BlueFunction, hist=False, color="b", label=BlueName,␣
 ↪ax=ax1)

    plt.title(Title)
    plt.xlabel('Price (in dollars)')
    plt.ylabel('Proportion of Cars')

    plt.show()
    plt.close()
```

[7]:
```python
def PollyPlot(xtrain, xtest, y_train, y_test, lr,poly_transform):
    width = 12
    height = 10
    plt.figure(figsize=(width, height))


    #training data
    #testing data
    # lr:  linear regression object
    #poly_transform:  polynomial transformation object

    xmax=max([xtrain.values.max(), xtest.values.max()])

    xmin=min([xtrain.values.min(), xtest.values.min()])

    x=np.arange(xmin, xmax, 0.1)


    plt.plot(xtrain, y_train, 'ro', label='Training Data')
    plt.plot(xtest, y_test, 'go', label='Test Data')
    plt.plot(x, lr.predict(poly_transform.fit_transform(x.reshape(-1, 1))),␣
 ↪label='Predicted Function')
    plt.ylim([-10000, 60000])
    plt.ylabel('Price')
    plt.legend()
```

Part 1: Training and Testing

An important step in testing your model is to split your data into training and testing data. We will place the target data price in a separate dataframe y:

[8]:
```python
y_data = df['price']
```

drop price data in x data

[9]:
```python
x_data=df.drop('price',axis=1)
```

Now we randomly split our data into training and testing data using the function train_test_split.

[10]:
```python
from sklearn.model_selection import train_test_split


x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.
 ↪15, random_state=1)


print("number of test samples :", x_test.shape[0])
print("number of training samples:",x_train.shape[0])
```

```
number of test samples : 31
number of training samples: 170
```

The test_size parameter sets the proportion of data that is split into the testing set. In the above, the testing set is set to 10% of the total dataset.

Question #1):

Use the function "train_test_split" to split up the data set such that 40% of the data samples will be utilized for testing, set the parameter "random_state" equal to zero. The output of the function should be the following: "x_train_1" , "x_test_1", "y_train_1" and "y_test_1".

[11]:
```
x_train1, x_test1, y_train1, y_test1 = train_test_split(x_data, y_data,␣
 ↪test_size=0.4, random_state=0)
print("number of test samples :", x_test1.shape[0])
print("number of training samples:",x_train1.shape[0])
```

```
number of test samples : 81
number of training samples: 120
```

Double-click here for the solution.

Let's import LinearRegression from the module linear_model.

[12]:
```
from sklearn.linear_model import LinearRegression
```

We create a Linear Regression object:

[13]:
```
lre=LinearRegression()
```

we fit the model using the feature horsepower

[14]:
```
lre.fit(x_train[['horsepower']], y_train)
```

[14]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

Let's Calculate the R^2 on the test data:

[15]:
```
lre.score(x_test[['horsepower']], y_test)
```

[15]:
```
0.707688374146705
```

we can see the R^2 is much smaller using the test data.

[16]:
```
lre.score(x_train[['horsepower']], y_train)
```

[16]:
```
0.6449517437659684
```

Question #2):

Find the R^2 on the test data using 90% of the data for training data

[17]:
```
x_train1, x_test1, y_train1, y_test1 = train_test_split(x_data, y_data,␣
↪test_size=0.1, random_state=0)
lre.fit(x_train1[['horsepower']],y_train1)
lre.score(x_test1[['horsepower']],y_test1)
```

[17]:
```
0.7340722810055448
```

Double-click here for the solution.

Sometimes you do not have sufficient testing data; as a result, you may want to perform Cross-validation. Let's go over several methods that you can use for Cross-validation.

Cross-validation Score

Lets import model_selection from the module cross_val_score.

[18]:
```
from sklearn.model_selection import cross_val_score
```

We input the object, the feature in this case ' horsepower', the target data (y_data). The parameter 'cv' determines the number of folds; in this case 4.

[19]:
```
Rcross = cross_val_score(lre, x_data[['horsepower']], y_data, cv=4)
```

The default scoring is R^2; each element in the array has the average R^2 value in the fold:

[20]:
```
Rcross
```

[20]:
```
array([0.7746232 , 0.51716687, 0.74785353, 0.04839605])
```

We can calculate the average and standard deviation of our estimate:

[21]:
```
print("The mean of the folds are", Rcross.mean(), "and the standard deviation␣
↪is" , Rcross.std())
```

```
The mean of the folds are 0.522009915042119 and the standard deviation is
0.291183944475603
```

We can use negative squared error as a score by setting the parameter 'scoring' metric to 'neg_mean_squared_error'.

[22]:
```
-1 * cross_val_score(lre,x_data[['horsepower']],␣
↪y_data,cv=4,scoring='neg_mean_squared_error')
```

[22]:
```
array([20254142.84026702, 43745493.2650517 , 12539630.34014931,
       17561927.72247591])
```

Question #3):

Calculate the average R^2 using two folds, find the average R^2 for the second fold utilizing the horsepower as a feature :

[23]:
```python
Rc=cross_val_score(lre,x_data[['horsepower']], y_data,cv=2)
Rc.mean()
```

[23]: 0.5166761697127429

Double-click here for the solution.

You can also use the function 'cross_val_predict' to predict the output. The function splits up the data into the specified number of folds, using one fold to get a prediction while the rest of the folds are used as test data. First import the function:

[24]:
```python
from sklearn.model_selection import cross_val_predict
```

We input the object, the feature in this case 'horsepower' , the target data y_data. The parameter 'cv' determines the number of folds; in this case 4. We can produce an output:

[25]:
```python
yhat = cross_val_predict(lre,x_data[['horsepower']], y_data,cv=4)
yhat[0:5]
```

[25]:
```
array([14141.63807508, 14141.63807508, 20814.29423473, 12745.03562306,
       14762.35027598])
```

Part 2: Overfitting, Underfitting and Model Selection

It turns out that the test data sometimes referred to as the out of sample data is a much better measure of how well your model performs in the real world. One reason for this is overfitting; let's go over some examples. It turns out these differences are more apparent in Multiple Linear Regression and Polynomial Regression so we will explore overfitting in that context.

Let's create Multiple linear regression objects and train the model using 'horsepower', 'curb-weight', 'engine-size' and 'highway-mpg' as features.

[26]:
```python
lr = LinearRegression()
lr.fit(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],␣
 ↪y_train)
```

[26]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

Prediction using training data:

[27]:
```python
yhat_train = lr.predict(x_train[['horsepower', 'curb-weight', 'engine-size',␣
 ↪'highway-mpg']])
yhat_train[0:5]
```

[27]:
```
array([11927.70699817, 11236.71672034,  6436.91775515, 21890.22064982,
       16667.18254832])
```

Prediction using test data:

[28]:
```
yhat_test = lr.predict(x_test[['horsepower', 'curb-weight', 'engine-size',␣
↪'highway-mpg']])
yhat_test[0:5]
```

[28]:
```
array([11349.16502418,  5914.48335385, 11243.76325987,  6662.03197043,
       15555.76936275])
```

Let's perform some model evaluation using our training and testing data separately. First we import the seaborn and matplotlibb library for plotting.

[29]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

Let's examine the distribution of the predicted values of the training data.

[30]:
```
Title = 'Distribution  Plot of  Predicted Value Using Training Data vs Training␣
↪Data Distribution'
DistributionPlot(y_train, yhat_train, "Actual Values (Train)", "Predicted␣
↪Values (Train)", Title)
```

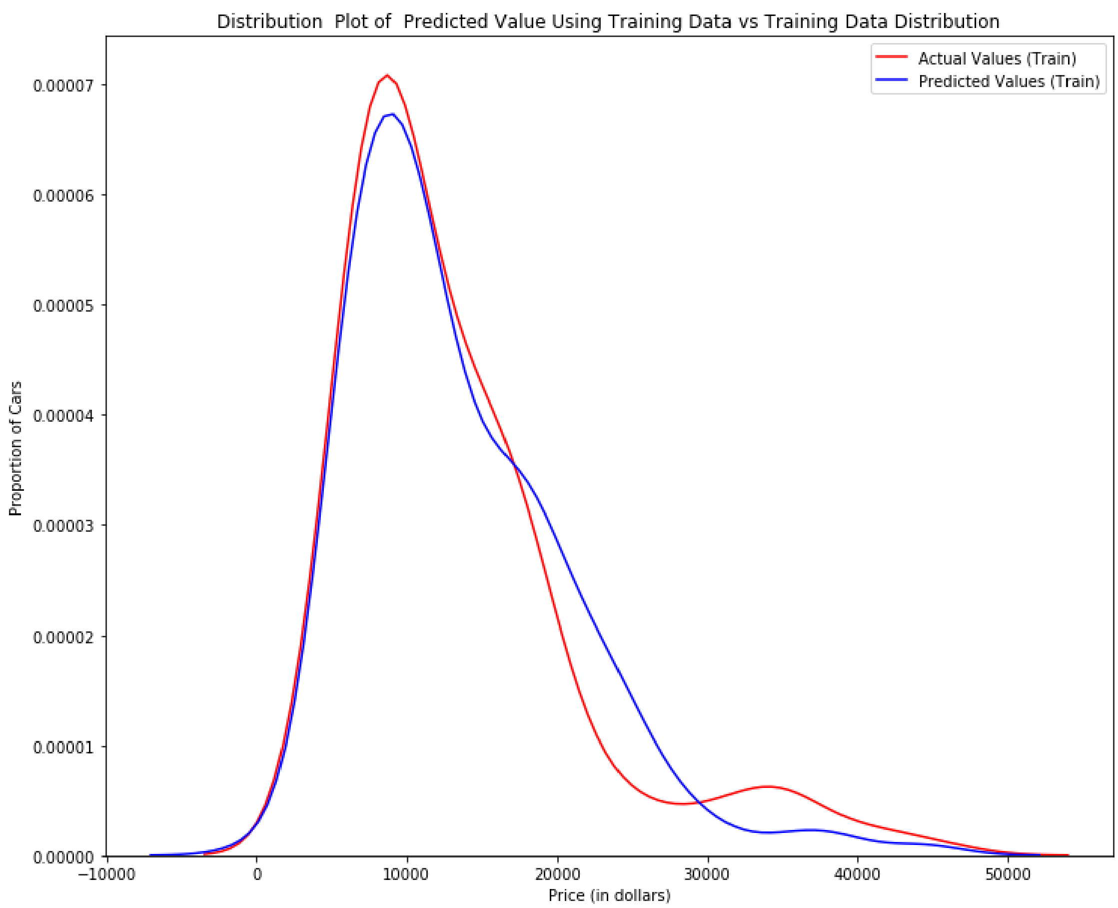
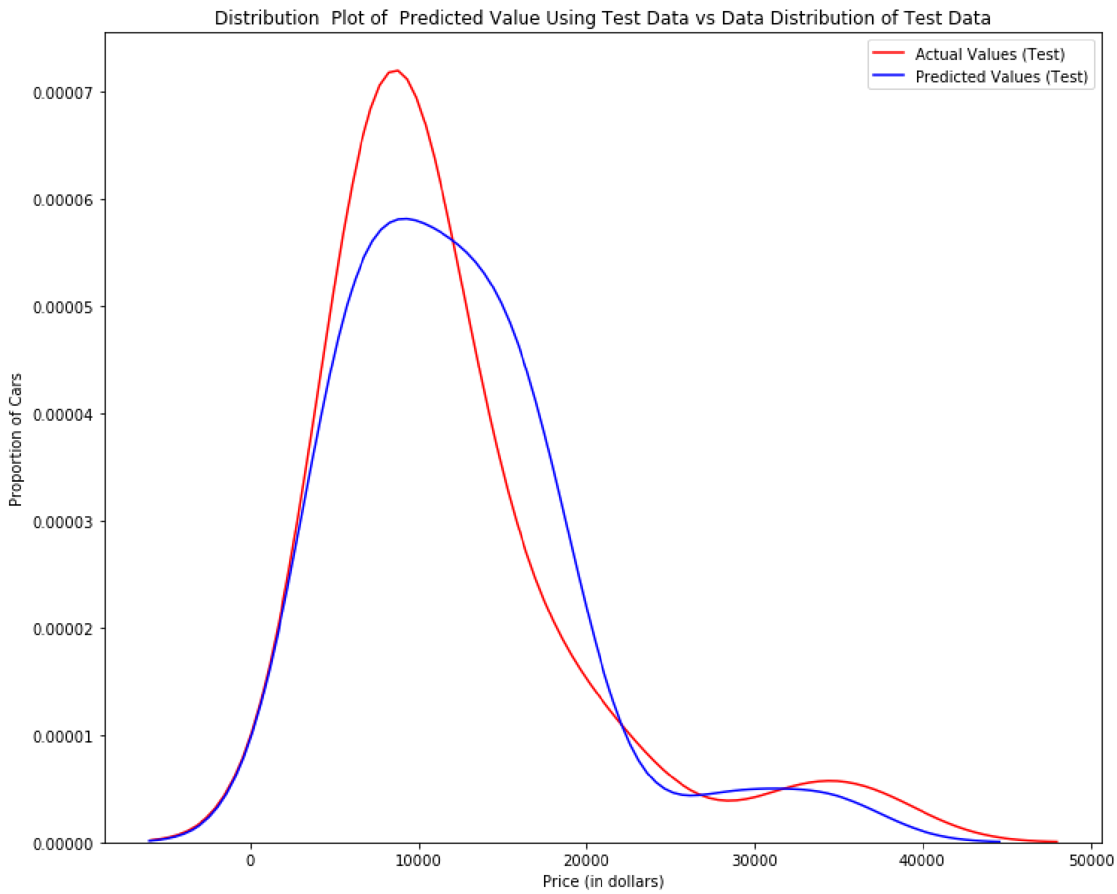


Figure 1: Plot of predicted values using the training data compared to the training data.

So far the model seems to be doing well in learning from the training dataset. But what happens when the model encounters new data from the testing dataset? When the model generates new values from the test data, we see the distribution of the predicted values is much different from the actual target values.

[31]:
```
Title='Distribution  Plot of  Predicted Value Using Test Data vs Data␣
↪Distribution of Test Data'
DistributionPlot(y_test,yhat_test,"Actual Values (Test)","Predicted Values␣
↪(Test)",Title)
```



Figur 2: Plot of predicted value using the test data compared to the test data.

Comparing Figure 1 and Figure 2; it is evident the distribution of the test data in Figure 1 is much better at fitting the data. This difference in Figure 2 is apparent where the ranges are from 5000 to 15 000. This is where the distribution shape is exceptionally different. Let's see if polynomial regression also exhibits a drop in the prediction accuracy when analysing the test dataset.

[32]:
```
from sklearn.preprocessing import PolynomialFeatures
```

Overfitting

Overfitting occurs when the model fits the noise, not the underlying process. Therefore when testing your model using the test-set, your model does not perform as well as it is modelling noise, not the underlying process that generated the relationship. Let's create a degree 5 polynomial model.

Let's use 55 percent of the data for testing and the rest for training:

[33]:
```
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.
↪45, random_state=0)
```

We will perform a degree 5 polynomial transformation on the feature 'horse power'.

[35]:
```
pr = PolynomialFeatures(degree=5)
x_train_pr = pr.fit_transform(x_train[['horsepower']])
x_test_pr = pr.fit_transform(x_test[['horsepower']])
pr
```

[35]:
```
PolynomialFeatures(degree=5, include_bias=True, interaction_only=False)
```

Now let's create a linear regression model "poly" and train it.

[36]:
```
poly = LinearRegression()
poly.fit(x_train_pr, y_train)
```

[36]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

We can see the output of our model using the method "predict." then assign the values to "yhat".

[37]:
```
yhat = poly.predict(x_test_pr)
yhat[0:5]
```

[37]:
```
array([ 6728.65561887,  7307.98782321, 12213.78770965, 18893.24804015,
       19995.95195136])
```

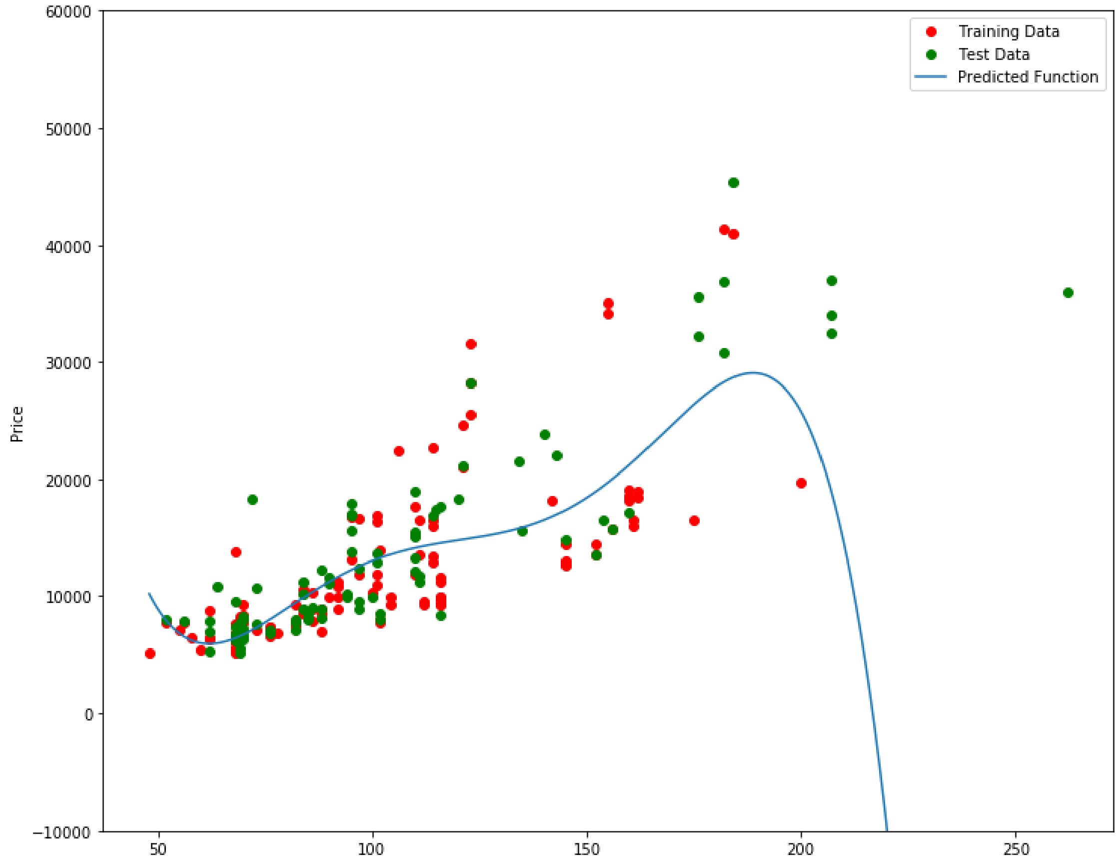
Let's take the first five predicted values and compare it to the actual targets.

[38]:
```
print("Predicted values:", yhat[0:4])
print("True values:", y_test[0:4].values)
```

```
Predicted values: [ 6728.65561887  7307.98782321 12213.78770965 18893.24804015]
True values: [ 6295. 10698. 13860. 13499.]
```

We will use the function "PollyPlot" that we defined at the beginning of the lab to display the training data, testing data, and the predicted function.

[39]:
```
PollyPlot(x_train[['horsepower']], x_test[['horsepower']], y_train, y_test,␣
↪poly,pr)
```



Figur 4 A polynomial regression model, red dots represent training data, green dots represent test data, and the blue line represents the model prediction.

We see that the estimated function appears to track the data but around 200 horsepower, the function begins to diverge from the data points.

R^2 of the training data:

[40]:
```
poly.score(x_train_pr, y_train)
```

[40]: 0.556771690212023

R^2 of the test data:

[41]:
```
poly.score(x_test_pr, y_test)
```

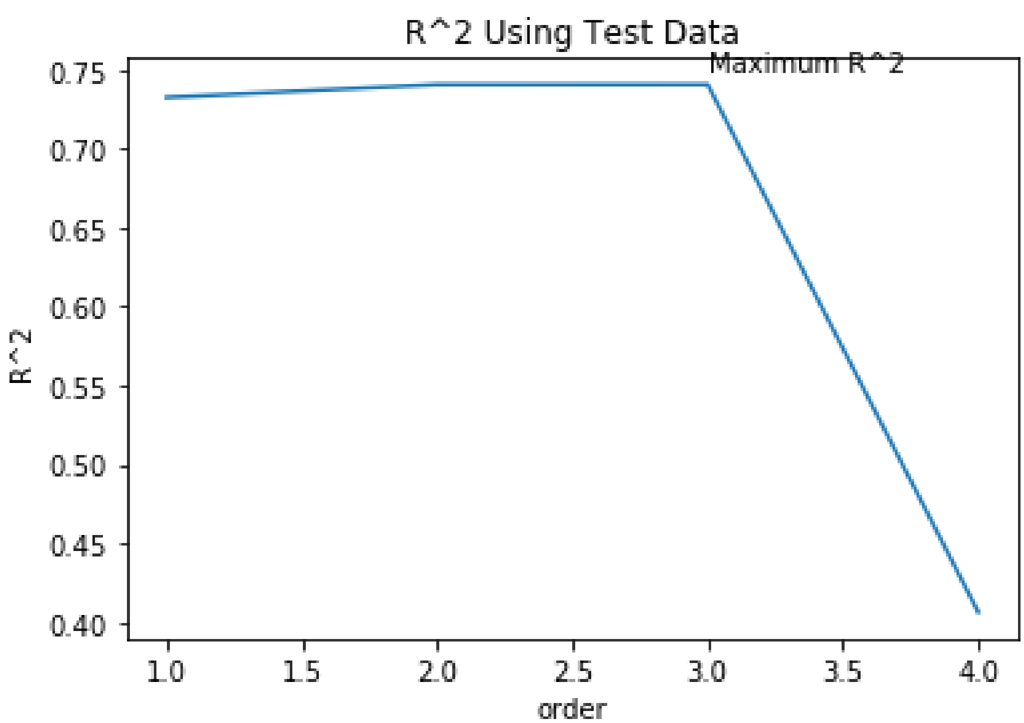
[41]: -29.871340302044153

We see the R^2 for the training data is 0.5567 while the R^2 on the test data was -29.87. The lower the R^2, the worse the model, a Negative R^2 is a sign of overfitting.

Let's see how the R^2 changes on the test data for different order polynomials and plot the results:

[42]:
```
Rsqu_test = []

order = [1, 2, 3, 4]
for n in order:
    pr = PolynomialFeatures(degree=n)

    x_train_pr = pr.fit_transform(x_train[['horsepower']])

    x_test_pr = pr.fit_transform(x_test[['horsepower']])

    lr.fit(x_train_pr, y_train)

    Rsqu_test.append(lr.score(x_test_pr, y_test))

plt.plot(order, Rsqu_test)
plt.xlabel('order')
plt.ylabel('R^2')
plt.title('R^2 Using Test Data')
plt.text(3, 0.75, 'Maximum R^2 ')
```

[42]:
```
Text(3, 0.75, 'Maximum R^2 ')
```



We see the R^2 gradually increases until an order three polynomial is used. Then the R^2 dramatically decreases at four.

The following function will be used in the next section; please run the cell.

[43]:
```
def f(order, test_data):
    x_train, x_test, y_train, y_test = train_test_split(x_data, y_data,␣
↪test_size=test_data, random_state=0)
    pr = PolynomialFeatures(degree=order)
    x_train_pr = pr.fit_transform(x_train[['horsepower']])
    x_test_pr = pr.fit_transform(x_test[['horsepower']])
    poly = LinearRegression()
    poly.fit(x_train_pr,y_train)
    PollyPlot(x_train[['horsepower']], x_test[['horsepower']], y_train,y_test,␣
↪poly, pr)
```

The following interface allows you to experiment with different polynomial orders and different amounts of data.

[44]:
```
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, d
```

[44]:
```
<function __main__.f(order, test_data)>
```

Question #4a):

We can perform polynomial transformations with more than one feature. Create a "PolynomialFeatures" object "pr1" of degree two?

Double-click here for the solution.

Question #4b):

Transform the training and testing samples for the features 'horsepower', 'curb-weight', 'engine-size' and 'highway-mpg'. Hint: use the method "fit_transform" ?

Double-click here for the solution.

Question #4c):

How many dimensions does the new feature have? Hint: use the attribute "shape"

Double-click here for the solution.

Question #4d):

Create a linear regression model "poly1" and train the object using the method "fit" using the polynomial features?

Double-click here for the solution.

Question #4e):

Use the method "predict" to predict an output on the polynomial features, then use the function "DistributionPlot" to display the distribution of the predicted output vs the test data?

Double-click here for the solution.

Question #4f):

Use the distribution plot to determine the two regions were the predicted prices are less accurate than the actual prices.

Double-click here for the solution.

In this section, we will review Ridge Regression we will see how the parameter Alfa changes the model. Just a note here our test data will be used as validation data.

Let's perform a degree two polynomial transformation on our data.

[45]:
```
pr=PolynomialFeatures(degree=2)
x_train_pr=pr.fit_transform(x_train[['horsepower', 'curb-weight',␣
↪'engine-size', 'highway-mpg','normalized-losses','symboling']])
x_test_pr=pr.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size',␣
↪'highway-mpg','normalized-losses','symboling']])
```

Let's import Ridge from the module linear models.

[46]:
```
from sklearn.linear_model import Ridge
```

Let's create a Ridge regression object, setting the regularization parameter to 0.1

[47]:
```
RigeModel=Ridge(alpha=0.1)
```

Like regular regression, you can fit the model using the method fit.

[49]:
```
RigeModel.fit(x_train_pr, y_train)
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/sklearn/linear_model/ridge.py:125: LinAlgWarning: Ill-conditioned
matrix (rcond=1.02972e-16): result may not be accurate.
  overwrite_a=True).T
```

[49]:
```
Ridge(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

Similarly, you can obtain a prediction:

[50]:
```
yhat = RigeModel.predict(x_test_pr)
```

Let's compare the first five predicted samples to our test set

[51]:
```
print('predicted:', yhat[0:4])
print('test set :', y_test[0:4].values)
```

```
predicted: [ 6567.83081933  9597.97151399 20836.22326843 19347.69543463]
test set : [ 6295. 10698. 13860. 13499.]
```

We select the value of Alfa that minimizes the test error, for example, we can use a for loop.

[52]:
```
Rsqu_test = []
Rsqu_train = []
dummy1 = []
ALFA = 10 * np.array(range(0,1000))
for alfa in ALFA:
    RigeModel = Ridge(alpha=alfa)
    RigeModel.fit(x_train_pr, y_train)
    Rsqu_test.append(RigeModel.score(x_test_pr, y_test))
    Rsqu_train.append(RigeModel.score(x_train_pr, y_train))
```

We can plot out the value of R^2 for different Alphas

[53]:
```
width = 12
height = 10
plt.figure(figsize=(width, height))

plt.plot(ALFA,Rsqu_test, label='validation data  ')
plt.plot(ALFA,Rsqu_train, 'r', label='training Data ')
plt.xlabel('alpha')
plt.ylabel('R^2')
plt.legend()
```

[53]:
```
<matplotlib.legend.Legend at 0x7f0978484630>
```

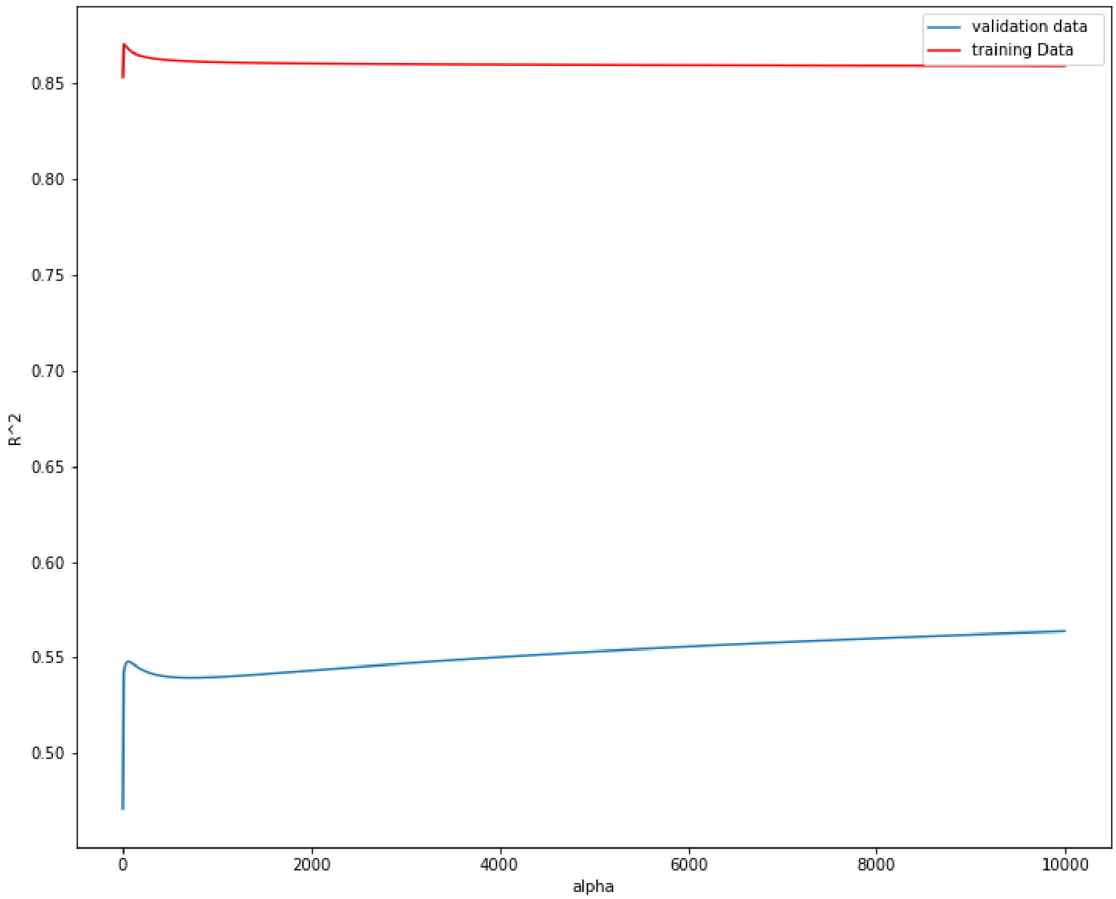


Figure 6:The blue line represents the R^2 of the test data, and the red line represents the R^2 of the training data. The x-axis represents the different values of Alfa

The red line in figure 6 represents the R^2 of the test data, as Alpha increases the R^2 decreases; therefore as Alfa increases the model performs worse on the test data. The blue line represents the R^2 on the validation data, as the value for Alfa increases the R^2 decreases.

Question #5):

Perform Ridge regression and calculate the R^2 using the polynomial features, use the training data to train the model and test data to test the model. The parameter alpha should be set to 10.

[54]:
```
RigeModel = Ridge(alpha=0)
RigeModel.fit(x_train_pr, y_train)
RigeModel.score(x_test_pr, y_test)
```

[54]: 0.47098333063511094

Double-click here for the solution.

Part 4: Grid Search

The term Alfa is a hyperparameter, sklearn has the class GridSearchCV to make the process of finding the best hyperparameter simpler.

Let's import GridSearchCV from the module model_selection.

[55]:
```
from sklearn.model_selection import GridSearchCV
```

We create a dictionary of parameter values:

[56]:
```
parameters1= [{'alpha': [0.001,0.1,1, 10, 100, 1000, 10000, 100000, 100000]}]
parameters1
```

[56]:
```
[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 100000]}]
```

Create a ridge regions object:

[57]:
```
RR=Ridge()
RR
```

[57]:
```
Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

Create a ridge grid search object

[58]:
```
Grid1 = GridSearchCV(RR, parameters1,cv=4)
```

Fit the model

[59]:
```
Grid1.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],␣
 ↪y_data)
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/sklearn/model_selection/_search.py:841: DeprecationWarning: The default
of the `iid` parameter will change from True to False in version 0.22 and will
be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
```

[59]:
```
GridSearchCV(cv=4, error_score='raise-deprecating',
       estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True,
max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid=[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000,
100000]}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

The object finds the best parameter values on the validation data. We can obtain the estimator with the best parameters and assign it to the variable BestRR as follows:

[60]:
```python
BestRR=Grid1.best_estimator_
BestRR
```

[60]:
```
Ridge(alpha=10000, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

We now test our model on the test data

[61]:
```python
BestRR.score(x_test[['horsepower', 'curb-weight', 'engine-size',␣
 ↪'highway-mpg']], y_test)
```

[61]:
```
0.8411649831036152
```

Question #6):

Perform a grid search for the alpha parameter and the normalization parameter, then find the best values of the parameters

[62]:
```python
parameters2= [{'alpha': [0.001,0.1,1, 10, 100,␣
 ↪1000,10000,100000,100000],'normalize':[True,False]} ]
Grid2 = GridSearchCV(Ridge(), parameters2,cv=4)
Grid2.fit(x_data[['horsepower', 'curb-weight', 'engine-size',␣
 ↪'highway-mpg']],y_data)
Grid2.best_estimator_
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/sklearn/model_selection/_search.py:841: DeprecationWarning: The default
of the `iid` parameter will change from True to False in version 0.22 and will
be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
```

[62]:
```
Ridge(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=True, random_state=None, solver='auto', tol=0.001)
```

Double-click here for the solution.

```
<p><a href="https://cocl.us/corsera_da0101en_notebook_bottom"><img src="https://s3-api.us-geo.
```

About the Authors:

This notebook was written by Mahdi Noorian PhD, Joseph Santarcangelo, Bahare Talayian, Eric Xiao, Steven Dong, Parizad, Hima Vsudevan and Fiorella Wenver and Yi Yao.

Joseph Santarcangelo is a Data Scientist at IBM, and holds a PhD in Electrical Engineering. His research focused on using Machine Learning, Signal Processing, and Computer Vision to determine how videos impact human cognition. Joseph has been working for IBM since he completed his PhD.

model-development

June 21, 2020

Module 4: Model Development

In this section, we will develop several models that will predict the price of the car using the variables or features. This is just an estimate but should give us an objective idea of how much the car should cost.

Some questions we want to ask in this module

do I know if the dealer is offering fair value for my trade-in?

do I know if I put a fair value on my car?

Data Analytics, we often use Model Development to help us predict future observations from the data we have.

A Model will help us understand the exact relationship between different variables and how these variables are used to predict the result.

Setup

Import libraries

[3]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

load data and store in dataframe df:

This dataset was hosted on IBM Cloud object click HERE for free storage.

[4]:
```
# path of data
path = 'https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/
↪CognitiveClass/DA0101EN/automobileEDA.csv'
df = pd.read_csv(path)
df.head()
```

[4]:
```
   symboling  normalized-losses         make aspiration num-of-doors  \
0          3                122  alfa-romero        std          two
1          3                122  alfa-romero        std          two
2          1                122  alfa-romero        std          two
3          2                164         audi        std         four
4          2                164         audi        std         four
```

```
   body-style drive-wheels engine-location  wheel-base    length  …  \
0  convertible          rwd           front        88.6  0.811148  …
1  convertible          rwd           front        88.6  0.811148  …
2    hatchback          rwd           front        94.5  0.822681  …
3        sedan          fwd           front        99.8  0.848630  …
4        sedan          4wd           front        99.4  0.848630  …

   compression-ratio  horsepower  peak-rpm city-mpg highway-mpg    price  \
0                9.0       111.0    5000.0       21          27  13495.0
1                9.0       111.0    5000.0       21          27  16500.0
2                9.0       154.0    5000.0       19          26  16500.0
3               10.0       102.0    5500.0       24          30  13950.0
4                8.0       115.0    5500.0       18          22  17450.0

  city-L/100km  horsepower-binned  diesel  gas
0    11.190476             Medium       0    1
1    11.190476             Medium       0    1
2    12.368421             Medium       0    1
3     9.791667             Medium       0    1
4    13.055556             Medium       0    1

[5 rows x 29 columns]
```

1. Linear Regression and Multiple Linear Regression

Linear Regression

One example of a Data Model that we will be using is

Simple Linear Regression.

Simple Linear Regression is a method to help us understand the relationship between two variables:

The predictor/independent variable (X)

The response/dependent variable (that we want to predict)(Y)

The result of Linear Regression is a linear function that predicts the response (dependent) variable as a function of the predictor (independent) variable.

$$Y : Response\ Variable X : Predictor\ Variables$$

Linear function:

$$Yhat = a + bX$$

a refers to the intercept of the regression line0, in other words: the value of Y when X is 0

b refers to the slope of the regression line, in other words: the value with which Y changes when X increases by 1 unit

Lets load the modules for linear regression

[8]:
```
from sklearn.linear_model import LinearRegression
```

Create the linear regression object

[7]:
```
lm = LinearRegression()
lm
```

[7]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

How could Highway-mpg help us predict car price?

For this example, we want to look at how highway-mpg can help us predict car price. Using simple linear regression, we will create a linear function with "highway-mpg" as the predictor variable and the "price" as the response variable.

[9]:
```
X = df[['highway-mpg']]
Y = df['price']
```

Fit the linear model using highway-mpg.

[10]:
```
lm.fit(X,Y)
```

[10]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

We can output a prediction

[11]:
```
Yhat=lm.predict(X)
Yhat[0:5]
```

[11]:
```
array([16236.50464347, 16236.50464347, 17058.23802179, 13771.3045085 ,
       20345.17153508])
```

What is the value of the intercept (a)?

[12]:
```
lm.intercept_
```

[12]:
```
38423.3058581574
```

What is the value of the Slope (b)?

[13]:
```
lm.coef_
```

[13]:
```
array([-821.73337832])
```

What is the final estimated linear model we get?

As we saw above, we should get a final linear model with the structure:

$$Yhat = a + bX$$

Plugging in the actual values we get:

price = 38423.31 - 821.73 x highway-mpg

Question #1 a):

Create a linear regression object?

[14]:
```
lm1 = LinearRegression()
lm1
```

[14]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

Double-click here for the solution.

Question #1 b):

Train the model using 'engine-size' as the independent variable and 'price' as the dependent variable?

[22]:
```
x = df[["engine-size"]]
y = df[["price"]]
lm1.fit(x,y)
```

[22]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

Double-click here for the solution.

Question #1 c):

Find the slope and intercept of the model?

Slope

[27]:
```
yhat = lm1.predict(x)
lm1.coef_
```

[27]:
```
array([[166.86001569]])
```

Intercept

[28]:
```
# Write your code below and press Shift+Enter to execute
lm1.intercept_
```

[28]:
```
array([-7963.33890628])
```

Double-click here for the solution.

Question #1 d):

What is the equation of the predicted line. You can use x and yhat or 'engine-size' or 'price'?

1 You can type you answer here

Double-click here for the solution.

Multiple Linear Regression

What if we want to predict car price using more than one variable?

If we want to use more variables in our model to predict car price, we can use Multiple Linear Regression. Multiple Linear Regression is very similar to Simple Linear Regression, but this method is used to explain the relationship between one continuous response (dependent) variable and two or more predictor (independent) variables. Most of the real-world regression models involve multiple predictors. We will illustrate the structure by using four predictor variables, but these results can generalize to any integer:

$$Y: Response Variable X_1 : Predictor Variable 1 X_2 : Predictor Variable 2 X_3 : Predictor Variable 3 X_4 : Predicto$$

$$a : intercept b_1 : coefficients of Variable 1 b_2 : coefficients of Variable 2 b_3 : coefficients of Variable 3 b_4 : coef$$

The equation is given by

$$Yhat = a + b_1 X_1 + b_2 X_2 + b_3 X_3 + b_4 X_4$$

From the previous section we know that other good predictors of price could be:

Horsepower

Curb-weight

Engine-size

Highway-mpg

Let's develop a model using these variables as the predictor variables.

[29]:
```
Z = df[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']]
```

Fit the linear model using the four above-mentioned variables.

[30]:
```
lm.fit(Z, df['price'])
```

[30]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

What is the value of the intercept(a)?

[31]:
```
lm.intercept_
```

[31]: -15806.62462632922

What are the values of the coefficients (b1, b2, b3, b4)?

[33]:
```python
lm.coef_
```

[33]: array([53.49574423, 4.70770099, 81.53026382, 36.05748882])

What is the final estimated linear model that we get?

As we saw above, we should get a final linear function with the structure:

$$Yhat = a + b_1 X_1 + b_2 X_2 + b_3 X_3 + b_4 X_4$$

What is the linear function we get in this example?

Price = -15678.742628061467 + 52.65851272 x horsepower + 4.69878948 x curb-weight + 81.95906216 x engine-size + 33.58258185 x highway-mpg

Question #2 a):

Create and train a Multiple Linear Regression model "lm2" where the response variable is price, and the predictor variable is 'normalized-losses' and 'highway-mpg'.

[34]:
```python
lm2 = LinearRegression()
lm2.fit(df[['normalized-losses' , 'highway-mpg']],df['price'])
```

[34]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

Double-click here for the solution.

Question #2 b):

Find the coefficient of the model?

[36]:
```python
lm2.coef_
```

[36]: array([1.49789586, -820.45434016])

Double-click here for the solution.

2) Model Evaluation using Visualization

Now that we've developed some models, how do we evaluate our models and how do we choose the best one? One way to do this is by using visualization.

import the visualization package: seaborn

[37]:
```python
# import the visualization package: seaborn
import seaborn as sns
%matplotlib inline
```

Regression Plot

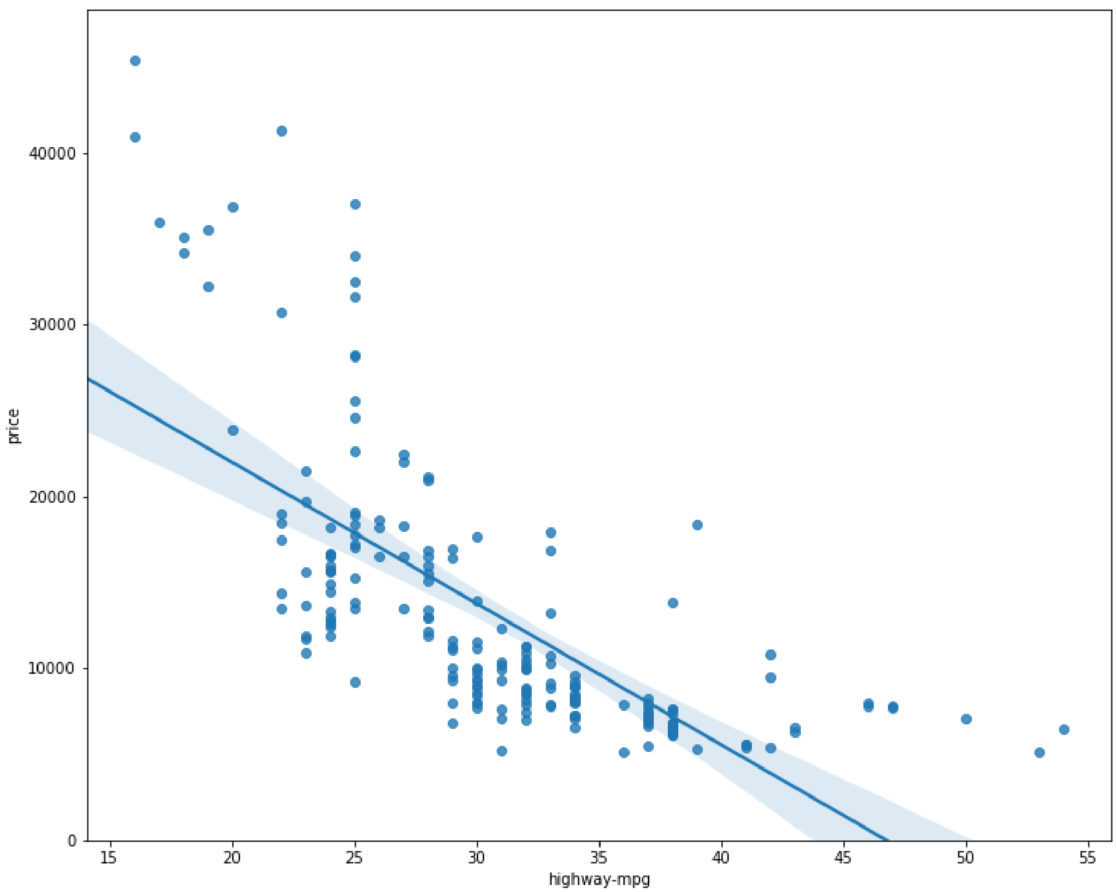
When it comes to simple linear regression, an excellent way to visualize the fit of our model is by using regression plots.

This plot will show a combination of a scattered data points (a scatter plot), as well as the fitted linear regression line going through the data. This will give us a reasonable estimate of the relationship between the two variables, the strength of the correlation, as well as the direction (positive or negative correlation).

Let's visualize Horsepower as potential predictor variable of price:

[38]:
```
width = 12
height = 10
plt.figure(figsize=(width, height))
sns.regplot(x="highway-mpg", y="price", data=df)
plt.ylim(0,)
```

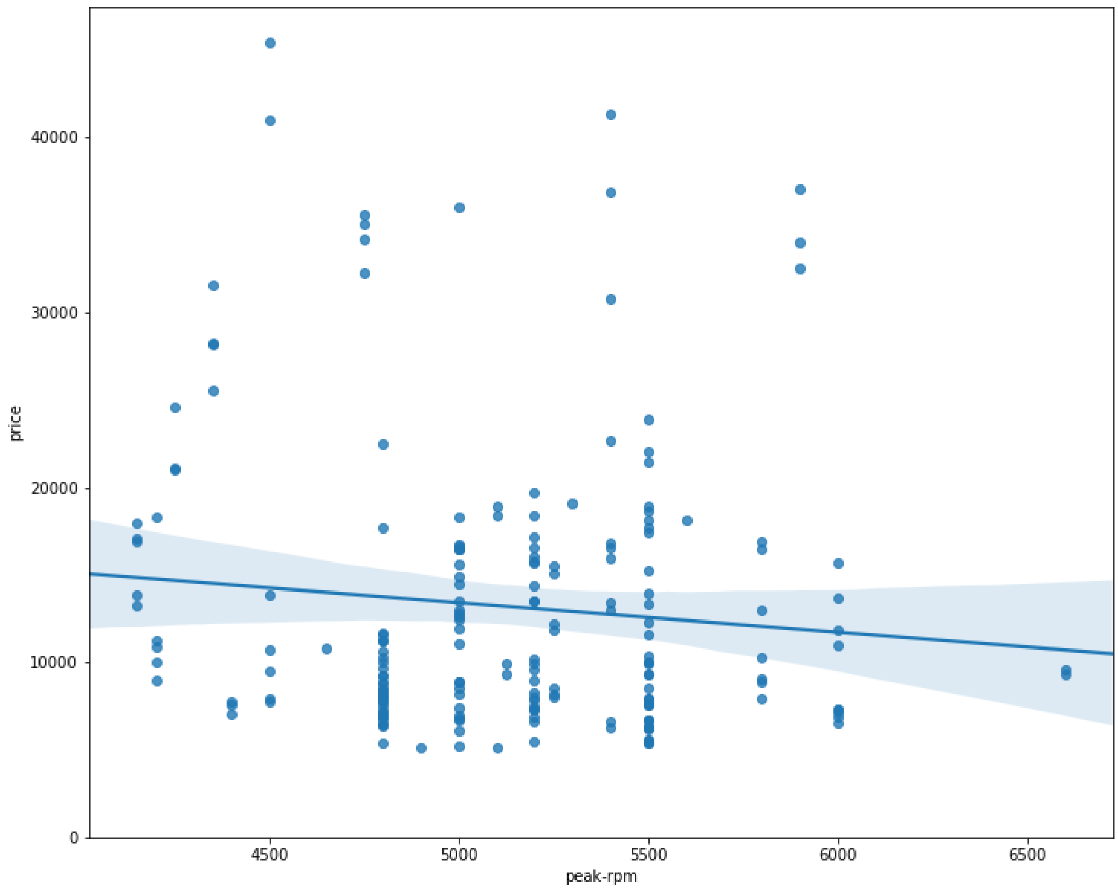
[38]:
```
(0, 48290.29917233836)
```



We can see from this plot that price is negatively correlated to highway-mpg, since the regression slope is negative. One thing to keep in mind when looking at a regression plot is to pay attention to how scattered the data points are around the regression line. This will give you a good indication of the variance of the data, and whether a linear model would be the best fit or not. If the data is too far off from the line, this linear model might not be the best model for this data. Let's compare this plot to the regression plot of "peak-rpm".

[39]:
```
plt.figure(figsize=(width, height))
sns.regplot(x="peak-rpm", y="price", data=df)
plt.ylim(0,)
```

[39]: (0, 47422.919330307624)



Comparing the regression plot of "peak-rpm" and "highway-mpg" we see that the points for "highway-mpg" are much closer to the generated line and on the average decrease. The points for "peak-rpm" have more spread around the predicted line, and it is much harder to determine if the points are decreasing or increasing as the "highway-mpg" increases.

Question #3:

Given the regression plots above is "peak-rpm" or "highway-mpg" more strongly correlated with "price". Use the method ".corr()" to verify your answer.

[40]:
```
df[["peak-rpm","highway-mpg","price"]].corr()
```

[40]:

	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

Double-click here for the solution.

Residual Plot

A good way to visualize the variance of the data is to use a residual plot.

What is a residual?

The difference between the observed value (y) and the predicted value (Yhat) is called the residual (e). When we look at a regression plot, the residual is the distance from the data point to the fitted regression line.

So what is a residual plot?

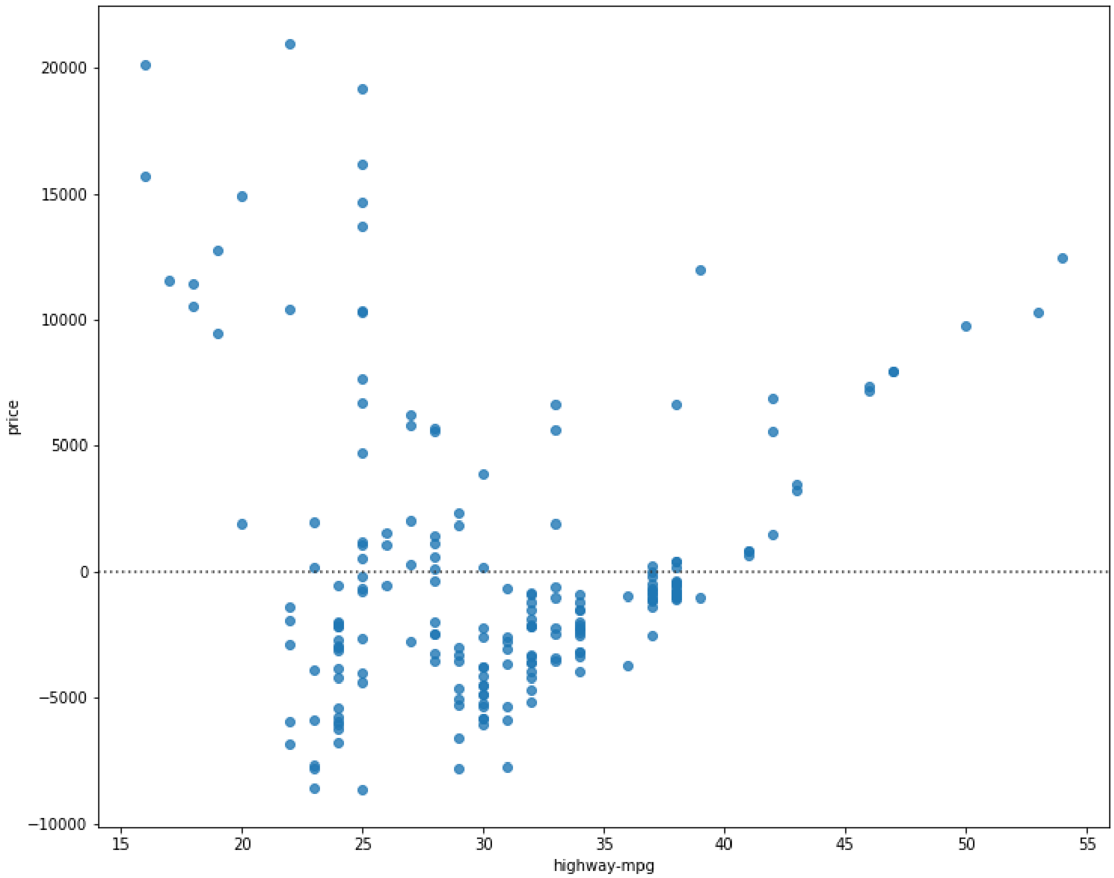
A residual plot is a graph that shows the residuals on the vertical y-axis and the independent variable on the horizontal x-axis.

What do we pay attention to when looking at a residual plot?

We look at the spread of the residuals:

- If the points in a residual plot are randomly spread out around the x-axis, then a linear model is appropriate for the data. Why is that? Randomly spread out residuals means that the variance is constant, and thus the linear model is a good fit for this data.

[41]:
```
width = 12
height = 10
plt.figure(figsize=(width, height))
sns.residplot(df['highway-mpg'], df['price'])
plt.show()
```



What is this plot telling us?

We can see from this residual plot that the residuals are not randomly spread around the x-axis, which leads us to believe that maybe a non-linear model is more appropriate for this data.

Multiple Linear Regression

How do we visualize a model for Multiple Linear Regression? This gets a bit more complicated because you can't visualize it with regression or residual plot.

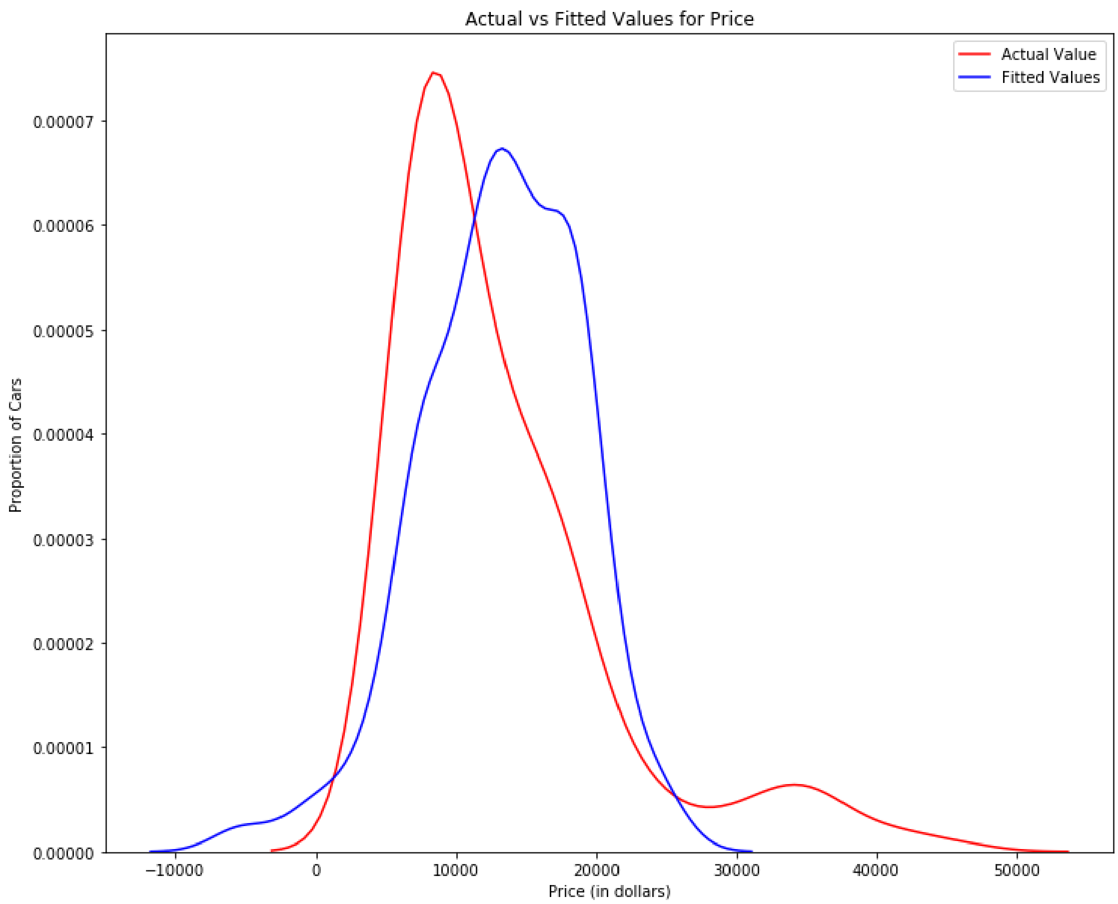
One way to look at the fit of the model is by looking at the distribution plot: We can look at the distribution of the fitted values that result from the model and compare it to the distribution of the actual values.

First lets make a prediction

[42]:
```
Y_hat = lm.predict(Z)
```

[43]:
```
plt.figure(figsize=(width, height))


ax1 = sns.distplot(df['price'], hist=False, color="r", label="Actual Value")
```

```
sns.distplot(Yhat, hist=False, color="b", label="Fitted Values" , ax=ax1)


plt.title('Actual vs Fitted Values for Price')
plt.xlabel('Price (in dollars)')
plt.ylabel('Proportion of Cars')

plt.show()
plt.close()
```



We can see that the fitted values are reasonably close to the actual values, since the two distributions overlap a bit. However, there is definitely some room for improvement.

Part 3: Polynomial Regression and Pipelines

Polynomial regression is a particular case of the general linear regression model or multiple linear regression models.

We get non-linear relationships by squaring or setting higher-order terms of the predictor variables.

There are different orders of polynomial regression:

Quadratic - 2nd order

$$Yhat = a + b_1 X^2 + b_2 X^2$$

Cubic - 3rd order

$$Yhat = a + b_1 X^2 + b_2 X^2 + b_3 X^3$$

Higher order:

$$Y = a + b_1 X^2 + b_2 X^2 + b_3 X^3 ....$$

We saw earlier that a linear model did not provide the best fit while using highway-mpg as the predictor variable. Let's see if we can try fitting a polynomial model to the data instead.

We will use the following function to plot the data:

[44]:
```
def PlotPolly(model, independent_variable, dependent_variabble, Name):
    x_new = np.linspace(15, 55, 100)
    y_new = model(x_new)

    plt.plot(independent_variable, dependent_variabble, '.', x_new, y_new, '-')
    plt.title('Polynomial Fit with Matplotlib for Price ~ Length')
    ax = plt.gca()
    ax.set_facecolor((0.898, 0.898, 0.898))
    fig = plt.gcf()
    plt.xlabel(Name)
    plt.ylabel('Price of Cars')

    plt.show()
    plt.close()
```

lets get the variables

[45]:
```
x = df['highway-mpg']
y = df['price']
```

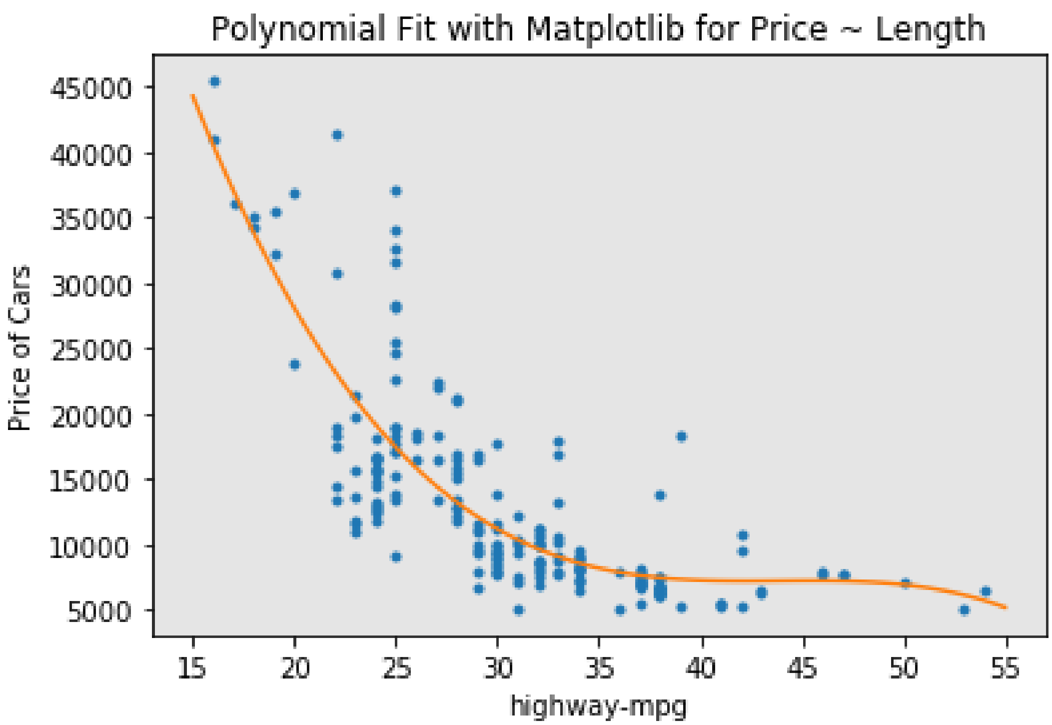
Let's fit the polynomial using the function polyfit, then use the function poly1d to display the polynomial function.

[46]:
```
# Here we use a polynomial of the 3rd order (cubic)
f = np.polyfit(x, y, 3)
p = np.poly1d(f)
print(p)
```

```
        3         2
-1.557 x + 204.8 x - 8965 x + 1.379e+05
```

Let's plot the function

[47]:
```
PlotPolly(p, x, y, 'highway-mpg')
```



[48]:
```
np.polyfit(x, y, 3)
```

[48]:
```
array([-1.55663829e+00,  2.04754306e+02, -8.96543312e+03,  1.37923594e+05])
```

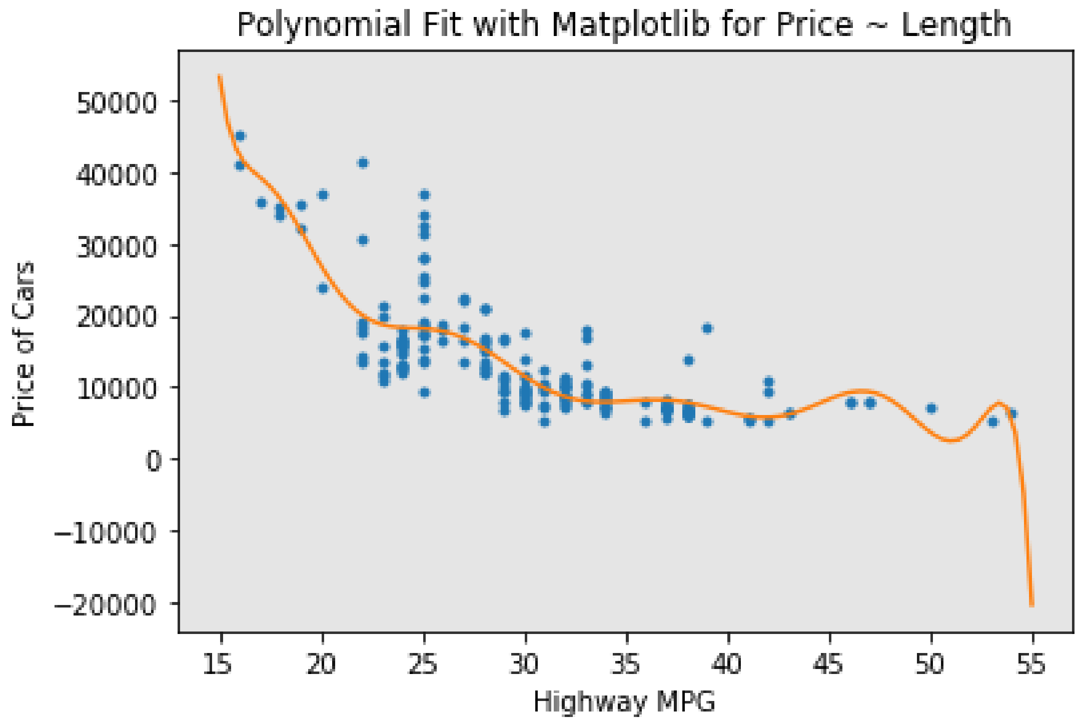
We can already see from plotting that this polynomial model performs better than the linear model. This is because the generated polynomial function "hits" more of the data points.

Question #4:

Create 11 order polynomial model with the variables x and y from above?

[49]:
```
f1 = np.polyfit(x, y, 11)
p1 = np.poly1d(f1)
print(p)
PlotPolly(p1,x,y, 'Highway MPG')
```

```
        3         2
-1.557 x + 204.8 x - 8965 x + 1.379e+05
```



Double-click here for the solution.

The analytical expression for Multivariate Polynomial function gets complicated. For example, the expression for a second-order (degree=2)polynomial with two variables is given by:

$$Yhat = a + b_1 X_1 + b_2 X_2 + b_3 X_1 X_2 + b_4 X_1^2 + b_5 X_2^2$$

We can perform a polynomial transform on multiple features. First, we import the module:

[50]:
```
from sklearn.preprocessing import PolynomialFeatures
```

We create a PolynomialFeatures object of degree 2:

[51]:
```
pr=PolynomialFeatures(degree=2)
pr
```

[51]:
```
PolynomialFeatures(degree=2, include_bias=True, interaction_only=False)
```

[52]:
```
Z_pr=pr.fit_transform(Z)
```

The original data is of 201 samples and 4 features

[53]:
```
Z.shape
```

[53]: (201, 4)

after the transformation, there 201 samples and 15 features

[54]:
```
Z_pr.shape
```

[54]: (201, 15)

Pipeline

Data Pipelines simplify the steps of processing the data. We use the module Pipeline to create a pipeline. We also use StandardScaler as a step in our pipeline.

[55]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

We create the pipeline, by creating a list of tuples including the name of the model or estimator and its corresponding constructor.

[56]:
```
Input=[('scale',StandardScaler()), ('polynomial',␣
 ↪PolynomialFeatures(include_bias=False)), ('model',LinearRegression())]
```

we input the list as an argument to the pipeline constructor

[57]:
```
pipe=Pipeline(Input)
pipe
```

[57]:
```
Pipeline(memory=None,
     steps=[('scale', StandardScaler(copy=True, with_mean=True, with_std=True)),
('polynomial', PolynomialFeatures(degree=2, include_bias=False,
interaction_only=False)), ('model', LinearRegression(copy_X=True,
fit_intercept=True, n_jobs=None,
         normalize=False))])
```

We can normalize the data, perform a transform and fit the model simultaneously.

[58]:
```
pipe.fit(Z,y)
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/sklearn/preprocessing/data.py:625: DataConversionWarning: Data with
input dtype int64, float64 were all converted to float64 by StandardScaler.
  return self.partial_fit(X, y)
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/sklearn/base.py:465: DataConversionWarning: Data with input dtype
int64, float64 were all converted to float64 by StandardScaler.
  return self.fit(X, y, **fit_params).transform(X)
```

[58]:
```
Pipeline(memory=None,
     steps=[('scale', StandardScaler(copy=True, with_mean=True, with_std=True)),
```

```
('polynomial', PolynomialFeatures(degree=2, include_bias=False,
interaction_only=False)), ('model', LinearRegression(copy_X=True,
fit_intercept=True, n_jobs=None,
         normalize=False))])
```

Similarly, we can normalize the data, perform a transform and produce a prediction simultaneously

[59]:
```
ypipe=pipe.predict(Z)
ypipe[0:4]
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/sklearn/pipeline.py:331: DataConversionWarning: Data with input dtype
int64, float64 were all converted to float64 by StandardScaler.
  Xt = transform.transform(Xt)
```

[59]:
```
array([13102.74784201, 13102.74784201, 18225.54572197, 10390.29636555])
```

Question #5:

Create a pipeline that Standardizes the data, then perform prediction using a linear regression model using the features Z and targets y

[60]:
```
Input=[('scale',StandardScaler()),('model',LinearRegression())]

pipe=Pipeline(Input)

pipe.fit(Z,y)

ypipe=pipe.predict(Z)
ypipe[0:10]
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/sklearn/preprocessing/data.py:625: DataConversionWarning: Data with
input dtype int64, float64 were all converted to float64 by StandardScaler.
  return self.partial_fit(X, y)
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/sklearn/base.py:465: DataConversionWarning: Data with input dtype
int64, float64 were all converted to float64 by StandardScaler.
  return self.fit(X, y, **fit_params).transform(X)
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/sklearn/pipeline.py:331: DataConversionWarning: Data with input dtype
int64, float64 were all converted to float64 by StandardScaler.
  Xt = transform.transform(Xt)
```

[60]:
```
array([13699.11161184, 13699.11161184, 19051.65470233, 10620.36193015,
       15521.31420211, 13869.66673213, 15456.16196732, 15974.00907672,
       17612.35917161, 10722.32509097])
```

Double-click here for the solution.

Part 4: Measures for In-Sample Evaluation

When evaluating our models, not only do we want to visualize the results, but we also want a quantitative measure to determine how accurate the model is.

Two very important measures that are often used in Statistics to determine the accuracy of a model are:

R^2 / R-squared

Mean Squared Error (MSE)

R-squared

R squared, also known as the coefficient of determination, is a measure to indicate how close the data is to the fitted regression line.

The value of the R-squared is the percentage of variation of the response variable (y) that is explained by a linear model.

Mean Squared Error (MSE)

The Mean Squared Error measures the average of the squares of errors, that is, the difference between actual value (y) and the estimated value (ŷ).

Model 1: Simple Linear Regression

Let's calculate the R^2

[61]:
```
#highway_mpg_fit
lm.fit(X, Y)
# Find the R^2
print('The R-square is: ', lm.score(X, Y))
```

```
The R-square is:  0.4965911884339176
```

We can say that ~ 49.659% of the variation of the price is explained by this simple linear model "horsepower_fit".

Let's calculate the MSE

We can predict the output i.e., "yhat" using the predict method, where X is the input variable:

[62]:
```
Yhat=lm.predict(X)
print('The output of the first four predicted value is: ', Yhat[0:4])
```

```
The output of the first four predicted value is:  [16236.50464347 16236.50464347
17058.23802179 13771.3045085 ]
```

lets import the function mean_squared_error from the module metrics

[63]:
```
from sklearn.metrics import mean_squared_error
```

we compare the predicted results with the actual results

[64]:
```
mse = mean_squared_error(df['price'], Yhat)
print('The mean square error of price and predicted value is: ', mse)
```

```
The mean square error of price and predicted value is:  31635042.944639888
```

Model 2: Multiple Linear Regression

Let's calculate the R^2

[65]:
```
# fit the model
lm.fit(Z, df['price'])
# Find the R^2
print('The R-square is: ', lm.score(Z, df['price']))
```

```
The R-square is:  0.8093562806577457
```

We can say that ~ 80.896 % of the variation of price is explained by this multiple linear regression "multi_fit".

Let's calculate the MSE

we produce a prediction

[66]:
```
Y_predict_multifit = lm.predict(Z)
```

we compare the predicted results with the actual results

[67]:
```
print('The mean square error of price and predicted value using multifit is: ',␣
↪\
      mean_squared_error(df['price'], Y_predict_multifit))
```

```
The mean square error of price and predicted value using multifit is:
11980366.87072649
```

Model 3: Polynomial Fit

Let's calculate the R^2

let's import the function r2_score from the module metrics as we are using a different function

[68]:
```
from sklearn.metrics import r2_score
```

We apply the function to get the value of r^2

[69]:
```
r_squared = r2_score(y, p(x))
print('The R-square value is: ', r_squared)
```

```
The R-square value is:  0.674194666390652
```

We can say that ~ 67.419 % of the variation of price is explained by this polynomial fit

MSE

We can also calculate the MSE:

[70]:
```python
mean_squared_error(df['price'], p(x))
```

[70]: 20474146.426361218

Part 5: Prediction and Decision Making

Prediction

In the previous section, we trained the model using the method fit. Now we will use the method predict to produce a prediction. Lets import pyplot for plotting; we will also be using some functions from numpy.

[71]:
```python
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline
```

Create a new input

[72]:
```python
new_input=np.arange(1, 100, 1).reshape(-1, 1)
```

Fit the model

[73]:
```python
lm.fit(X, Y)
lm
```

[73]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

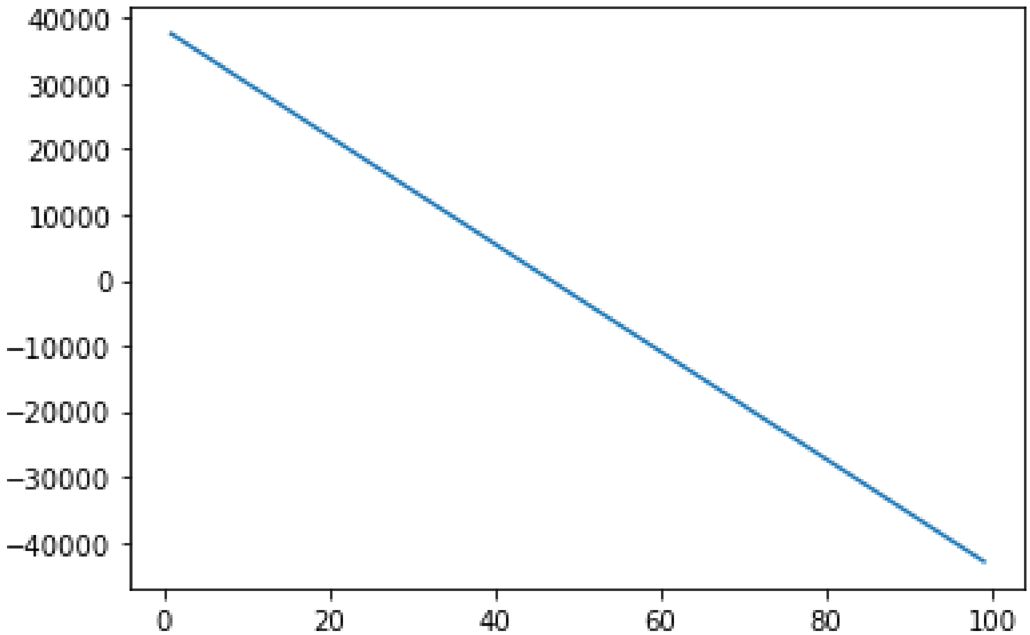
Produce a prediction

[74]:
```python
yhat=lm.predict(new_input)
yhat[0:5]
```

[74]:
```
array([37601.57247984, 36779.83910151, 35958.10572319, 35136.37234487,
       34314.63896655])
```

we can plot the data

[75]:
```python
plt.plot(new_input, yhat)
plt.show()
```



Decision Making: Determining a Good Model Fit

Now that we have visualized the different models, and generated the R-squared and MSE values for the fits, how do we determine a good model fit?

What is a good R-squared value?

When comparing models, the model with the higher R-squared value is a better fit for the data.

What is a good MSE?

When comparing models, the model with the smallest MSE value is a better fit for the data.

Let's take a look at the values for the different models.

Simple Linear Regression: Using Highway-mpg as a Predictor Variable of Price.

R-squared: 0.49659118843391759

MSE: 3.16 x10^7

Multiple Linear Regression: Using Horsepower, Curb-weight, Engine-size, and Highway-mpg as Predictor Variables of Price.

R-squared: 0.80896354913783497

MSE: 1.2 x10^7

Polynomial Fit: Using Highway-mpg as a Predictor Variable of Price.

R-squared: 0.6741946663906514

MSE: 2.05 x 10^7

Simple Linear Regression model (SLR) vs Multiple Linear Regression model (MLR)

Usually, the more variables you have, the better your model is at predicting, but this is not always true. Sometimes you may not have enough data, you may run into numerical problems, or many of the variables may not be useful and or even act as noise. As a result, you should always check the MSE and R^2.

So to be able to compare the results of the MLR vs SLR models, we look at a combination of both the R-squared and MSE to make the best conclusion about the fit of the model.

MSEThe MSE of SLR is 3.16x10^7 while MLR has an MSE of 1.2 x10^7. The MSE of MLR is much smaller.

R-squared: In this case, we can also see that there is a big difference between the R-squared of the SLR and the R-squared of the MLR. The R-squared for the SLR (~0.497) is very small compared to the R-squared for the MLR (~0.809).

This R-squared in combination with the MSE show that MLR seems like the better model fit in this case, compared to SLR.

Simple Linear Model (SLR) vs Polynomial Fit

MSE: We can see that Polynomial Fit brought down the MSE, since this MSE is smaller than the one from the SLR.

R-squared: The R-squared for the Polyfit is larger than the R-squared for the SLR, so the Polynomial Fit also brought up the R-squared quite a bit.

Since the Polynomial Fit resulted in a lower MSE and a higher R-squared, we can conclude that this was a better fit model than the simple linear regression for predicting Price with Highway-mpg as a predictor variable.

Multiple Linear Regression (MLR) vs Polynomial Fit

MSE: The MSE for the MLR is smaller than the MSE for the Polynomial Fit.

R-squared: The R-squared for the MLR is also much larger than for the Polynomial Fit.

Conclusion:

Comparing these three models, we conclude that the MLR model is the best model to be able to predict price from our dataset. This result makes sense, since we have 27 variables in total, and we know that more than one of those variables are potential predictors of the final car price.

Thank you for completing this notebook

```
<p><a href="https://cocl.us/corsera_da0101en_notebook_bottom"><img src="https://s3-api.us-geo.c
```

About the Authors:

This notebook was written by Mahdi Noorian PhD, Joseph Santarcangelo, Bahare Talayian, Eric Xiao, Steven Dong, Parizad, Hima Vsudevan and Fiorella Wenver and Yi Yao.

Joseph Santarcangelo is a Data Scientist at IBM, and holds a PhD in Electrical Engineering. His research focused on using Machine Learning, Signal Processing, and Computer Vision to determine how videos impact human cognition. Joseph has been working for IBM since he completed his PhD.