Importance of data analysis

- data is everywhere
- data analysis/data science helps us answer questions from data
- data analysis plays an important role in:
  - discovering useful infomation
  - answering questions
  - predicting future or the unknown

Example: dataset on used car prices

General process

1 Importing and Exporting Data in Python

2 Exploring the dataset

- understand the datatype, features, ...
- look at distribution of data within the dataset

3 Accessing Databases with Python

4 Data Wrangling/ Data preprocessing/ Data cleaning

- handle missing values
- data formatting
- data normalization (centering & scaling)
- data binning
- convert categorical values into numeric variables

5 Exploratory Data Analysis

- Descriptive Statistics
- Groupby
- Correlation
- Chi-Square

6 Model Development

- Simple and multiple linear regression
- Model Evaluation using visualization
- Polynomial regression and pipelines
- R-squared and MSE for in-sample evaluation
- Prediction and decision making

7 Model Evaluation

- Refinement
- Overfitting, Underfitting and Model Selection
- Ridge Regression
- Grid Search

In [ ]:

Understanding the dataset

- in CSV format
- 26 columns
- The first attribute, symboling, corresponds to the insurance risk level of a car.
- The second attribute, normalized-losses, is the relative average loss payment per insured vehicle year. This value is normalized for all autos within a particular size classification, two door small, station wagons, sports specialty, etc., and represents the average loss per car per year. The values range from 65 to 256.
- ...
- The goal of this project is to predict price in terms of other car features

Type *Markdown* and LaTeX: α^2

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.
- scientific computing libraries:
  - Pandas: offers data structure and tools for effective data manipulation and analysis. It provides facts, access to structured data. The primary instrument of Pandas is the two dimensional table consisting of column and row labels, which are called a data frame. It is designed to provid easy indexing functionality.
  - 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 methods:
  - Matplotlib package: the most well known library for data visualization. It is great for making graphs and plots. The graphs are also highly customizable.
  - Seaborn: high level visualization library It's very easy to generate various plots such as heat maps, time series and violin plot.
- algorithmic libraries: tackles the machine learning tasks from basic to complex.
  - 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.

In [ ]:

Importing and Exporting Data in Python

- A process of loading and reading data into notebook from various sources.
- To read any data using Python's pandas package, there are two important factors to consider, format and file path.
  - Format is the way data is encoded.
    - Some common encodings are: .CSV, .JSON, .XLSX, .HDF and so on.
  - The path tells us where the data is stored.
    - on the computer we are using
    - online on the internet

In [ ]:
```
import pandas as pd

# read the online file by the URL provided above, and assign it to variable "df"
path="https://archive.ics.uci.edu/ml/machine-learning-database/autos/imports-85.data"

# printing the dataframe
df = read_csv(path,header=None)
df
df.head()/df.head(10)
df.tail()/df.tail(10)

# adding headers
headers=["xxx","yyy","zzz"]
df.columms=headers
df.head(5)

# replace the "?" symbol with NaN
df1=df.replace('?',np.NaN)

# drop missing values along the column "price"
df=df1.dropna(subset=["price"], axis=0)

# Find the name of the columns of the dataframe
df.columns

# exporting a pandas dataframe to CSV/ saving the data
path="C:/xxx/.../yyy.csv"
df.to_csv(path)

# exporting to different formats in python
Data Format | Read          | Save
csv         |pd.read_csv()  |df.to_csv()
json        |pd.read_json() |df.to_json()
excel       |pd.read_excel()|df.to_excel()
sql         |pd.read_sql()  |df.to_sql()
```

Data Acquisition

There are various formats for a dataset, .csv, .json, .xlsx etc. The dataset can be stored in different places, on your local machine or sometimes online.
In this section, you will learn how to load a dataset into our Jupyter Notebook.
In our case, the Automobile Dataset is an online source, and it is in CSV (comma separated value) format. Let's use this dataset as an example to practice data reading.

- data source: https://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.data (https://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.data)
- data type: csv

The Pandas Library is a useful tool that enables us to read various datasets into a data frame; our Jupyter notebook platforms have a built-in **Pandas Library** so that all we need to do is import Pandas without installing.

In [1]:
```
# import pandas library
import pandas as pd
import numpy as np
```

Read Data

We use `pandas.read_csv()` function to read the csv file. In the bracket, we put the file path along with a quotation mark, so that pandas will read the file into a data frame from that address. The file path can be either an URL or your local file address.
Because the data does not include headers, we can add an argument `headers = None` inside the `read_csv()` method, so that pandas will not automatically set the first row as a header.
You can also assign the dataset to any variable you create.

In [2]:
```
# Import pandas library
import pandas as pd

# Read the online file by the URL provides above, and assign it to variable "df"
other_path = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperS
df = pd.read_csv(other_path, header=None)
```

In [3]:
```
# show the first 5 rows using dataframe.head() method
print("The first 5 rows of the dataframe")
df.head(5)
```

```
The first 5 rows of the dataframe
```

Out[3]:

	0	1	2	3	4	5	6	7	8	9	...	16	17	18	19	20	21	22	23	24
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21	27
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21	27
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	9.0	154	5000	19	26
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	10.0	102	5500	24	30
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	8.0	115	5500	18	22

5 rows × 26 columns

Add Headers

Take a look at our dataset; pandas automatically set the header by an integer from 0.

To better describe our data we can introduce a header, this information is available at:
https://archive.ics.uci.edu/ml/datasets/Automobile (https://archive.ics.uci.edu/ml/datasets/Automobile)

Thus, we have to add headers manually.

Firstly, we create a list "headers" that include all column names in order. Then, we use `dataframe.columns = headers` to replace the headers by the list we created.

In [4]:
```
# create headers list
headers = ["symboling","normalized-losses","make","fuel-type","aspiration", "num-of-doors","bo
           "drive-wheels","engine-location","wheel-base", "length","width","height","curb-weight
           "num-of-cylinders", "engine-size","fuel-system","bore","stroke","compression-ratio","
           "peak-rpm","city-mpg","highway-mpg","price"]
print("headers\n", headers)
```

```
headers
 ['symboling', 'normalized-losses', 'make', 'fuel-type', 'aspiration', 'num-of-doors', 'body-
style', 'drive-wheels', 'engine-location', 'wheel-base', 'length', 'width', 'height', 'curb-w
eight', 'engine-type', 'num-of-cylinders', 'engine-size', 'fuel-system', 'bore', 'stroke', 'c
ompression-ratio', 'horsepower', 'peak-rpm', 'city-mpg', 'highway-mpg', 'price']
```

In [6]:
```
# We replace headers and recheck our data frame
df.columns = headers
df.head()
```

Out[6]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fu syste
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	m
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	m
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	m
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	m
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	m

5 rows × 26 columns

In [7]:
```
# we need to replace the "?" symbol with NaN so the dropna() can remove the missing values
df1=df.replace('?',np.NaN)
```

In [8]:
```
# we can drop missing values along the column "price" as follows
df=df1.dropna(subset=["price"], axis=0)
df.head(20)
```

Out[8]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size
0	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130
1	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130
2	1	NaN	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136
5	2	NaN	audi	gas	std	two	sedan	fwd	front	99.8	...	136
6	1	158	audi	gas	std	four	sedan	fwd	front	105.8	...	136
7	1	NaN	audi	gas	std	four	wagon	fwd	front	105.8	...	136

In [9]:
```
# Find the name of the columns of the dataframe
df.columns
```

Out[9]:
```
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'],
      dtype='object')
```

Save Dataset

Correspondingly, Pandas enables us to save the dataset to csv by using the `dataframe.to_csv()` method, you can add the file path and name along with quotation marks in the brackets.

For example, if you would save the dataframe **df** as **automobile.csv** to your local machine, you may use the syntax below:

In [10]:
```
df.to_csv("automobile.csv", index=False)
```

Read/Save Other Data Formats

Data Formate	Read	Save
csv	`pd.read_csv()`	`df.to_csv()`
json	`pd.read_json()`	`df.to_json()`
excel	`pd.read_excel()`	`df.to_excel()`
hdf	`pd.read_hdf()`	`df.to_hdf()`
sql	`pd.read_sql()`	`df.to_sql()`
...	...	...

Basic Insight of Dataset

After reading data into Pandas dataframe, it is time for us to explore the dataset.
There are several ways to obtain essential insights of the data to help us better understand our dataset.

Data Types

Data has a variety of types.
The main types stored in Pandas dataframes are **object**, **float**, **int**, **bool** and **datetime64**. In order to better learn about each attribute, it is always good for us to know the data type of each column. In Pandas:

In [11]:
```
df.dtypes
```

Out[11]:
```
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
```

Describe

If we would like to get a statistical summary of each column, such as count, column mean value, column standard deviation, etc. We use the describe method:

In [13]:
```
df.describe()
```

Out[13]:

	symboling	wheel-base	length	width	height	curb-weight	engine-size	compression-ratio	city-r
count	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000
mean	0.840796	98.797015	174.200995	65.889055	53.766667	2555.666667	126.875622	10.164279	25.179
std	1.254802	6.066366	12.322175	2.101471	2.447822	517.296727	41.546834	4.004965	6.423
min	-2.000000	86.600000	141.100000	60.300000	47.800000	1488.000000	61.000000	7.000000	13.000
25%	0.000000	94.500000	166.800000	64.100000	52.000000	2169.000000	98.000000	8.600000	19.000
50%	1.000000	97.000000	173.200000	65.500000	54.100000	2414.000000	120.000000	9.000000	24.000
75%	2.000000	102.400000	183.500000	66.600000	55.500000	2926.000000	141.000000	9.400000	30.000
max	3.000000	120.900000	208.100000	72.000000	59.800000	4066.000000	326.000000	23.000000	49.000

In [14]:
```
# describe all the columns in "df"
df.describe(include = "all")
```

Out[14]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engin siz
count	201.000000	164	201	201	201	199	201	201	201	201.000000	...	201.0000
unique	NaN	51	22	2	2	2	5	3	2	NaN	...	Na
top	NaN	161	toyota	gas	std	four	sedan	fwd	front	NaN	...	Na
freq	NaN	11	32	181	165	113	94	118	198	NaN	...	Na
mean	0.840796	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	98.797015	...	126.8756
std	1.254802	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	6.066366	...	41.5468
min	-2.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	86.600000	...	61.0000
25%	0.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	94.500000	...	98.0000
50%	1.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	97.000000	...	120.0000
75%	2.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	102.400000	...	141.0000
max	3.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	120.900000	...	326.0000

11 rows × 26 columns

In [ ]:
```
# select the columns of a data frame by indicating the name of each column
df[['length', 'compression-ratio'] ].describe()
```

Info

Another method you can use to check your dataset is:

In [15]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 201 entries, 0 to 204
Data columns (total 26 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   symboling          201 non-null    int64  
 1   normalized-losses  164 non-null    object 
 2   make               201 non-null    object 
 3   fuel-type          201 non-null    object 
 4   aspiration         201 non-null    object 
 5   num-of-doors       199 non-null    object 
 6   body-style         201 non-null    object 
 7   drive-wheels       201 non-null    object 
 8   engine-location    201 non-null    object 
 9   wheel-base         201 non-null    float64
 10  length             201 non-null    float64
 11  width              201 non-null    float64
 12  height             201 non-null    float64
 13  curb-weight        201 non-null    int64  
```

In [ ]:

Exploring the dataset

- basic knows using python, pandas and data
- gives an overview of the dataset
- points out potential issues such as the wrong data type of features which may need to be resolved later on.

Data has a variety of types

- The main types stored in Pandas' objects are object, float, Int, and datetime.
- Reasons on checking datatypes:
  - potential info and type mismatch:
    - For example, it should be awkward if the car price column which we should expect to contain continuous numeric numbers, is assigned the data type of object.
  - compatibility with python methods: which python functions can be applied to a specific column.
    - For example, some math functions can only be applied to numerical data. If these functions are applied to non-numerical data an error may result. When the dtype method is applied to the data set, the data type of each column is returned in a series.

In [ ]:
```
# check data types
df.dtypes

# check statistical summary
df.describe()   # count, means, sd, max, min, ...
df.describe(include="all")   # for all columns

# check statistical summary & select the columns of a data frame
df[ ['length', 'compression-ratio'] ].describe()

# provide a concise summary of the DataFrame
df.info()
```

Accessing Databases with Python

SQL API

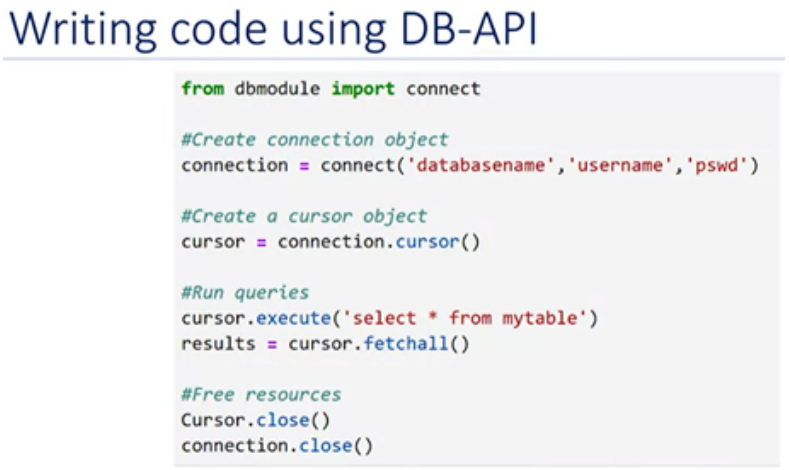
- The Python code connects to the database using API calls.
- An application programming interface is a set of functions that you can call to get access to some type of service.
- The SQL API consists of library function calls as an application programming interface, API, for the DBMS. To pass SQL statements to the DBMS, an application program calls functions in the API, and it calls other functions to retrieve query results and status information from the DBMS.

  - 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

- 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.
- Two main concepts in the Python DB-API: connection objects and query objects
  - Connection objects: connect to a database and manage your transactions
  - Cursor objects are used to run queries: 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.
  - connection methods:
  - 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.



  - Remember that it is always important to close connections to avoid unused connections taking up resources.

In [ ]:

Data Wrangling/ Data Cleaning

Pre-processing Data in Python

- a necessary step in data analysis.
- It is the process of converting or mapping data from one raw form into another format to make it ready for further analysis.

Dealing with Missing Values

- Usually missing value in data set appears as question mark and a zero or just a blank cell. In example, NaN.
- ways:
  - check if the person or group that collected the data can go back and find what the actual value should be
  - remove the data where that missing value is found
    - Drop data: drop the whole variable or just the single data entry with the missing value, usually dropping the particular entry is the best. To have the least amount of impact.
  - replace data: to replace missing values by the average value of the entire variable (standard for placement technique)
  - what if the values cannot be averaged as with categorical variables?
    - one possibility is to try using the mode

Two methods to detect missing data

- .isnull()
- .notnull()

In [ ]:
```
# drop missing vale
df.dropna()     # axis=0 drops the entire row
                # axis=1 drops the entire column
df.dropna( subset=["price"], axis=0, inplace=True )

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

#-------------------------------------------------
# replace missing values like NaNs with actual values
df.replace( missing_value, new_value )
# replace "?" to NaN
df.replace("?", np.nan, inplace = True)

#-------------------------------------------------
# detect missing data
missing_data = df.isnull() #"True" stands for missing value, while "False" stands for not miss

#Count missing values in each column
for column in missing_data.columns.values.tolist():
    print(column)
    print (missing_data[column].value_counts())
    print("")

#-------------------------------------------------
# Calculate the mean vaule for "stroke" column
avg_stroke = df["stroke"].astype("float").mean(axis = 0)
print("Average of stroke:", avg_stroke)
# ANS: Average of stroke: 3.255422885572139

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

#-------------------------------------------------
# To see which values are present in a particular column
df['num-of-doors'].value_counts()
df['num-of-doors'].value_counts().idxmax()
```

In [16]:
```
missing_data = df.isnull()
missing_data.head(5)
```

Out[16]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	
0	False	True	False	False	False	False	False	False	False	False	...	False	False	F
1	False	True	False	False	False	False	False	False	False	False	...	False	False	F
2	False	True	False	False	False	False	False	False	False	False	...	False	False	F
3	False	False	False	False	False	False	False	False	False	False	...	False	False	F
4	False	False	False	False	False	False	False	False	False	False	...	False	False	F

5 rows × 26 columns

In [17]:
```
for column in missing_data.columns.values.tolist():
    print(column)
    print (missing_data[column].value_counts())
    print("")
```

```
horsepower
False    199
True       2
Name: horsepower, dtype: int64

peak-rpm
False    199
True       2
Name: peak-rpm, dtype: int64

city-mpg
False    201
Name: city-mpg, dtype: int64

highway-mpg
False    201
Name: highway-mpg, dtype: int64

price
False    201
```

Deal with missing data

How to deal with missing data?

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

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

Replace by mean:

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

- "peak-rpm": 2 missing data, replace them with mean

Replace by frequency:

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

Drop the whole row:

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

In [21]:
```
# Calculate the average of the column
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
```

In [22]:
```
# Replace "NaN" by mean value in "normalized-losses" column
df["normalized-losses"].replace(np.nan, avg_norm_loss, inplace=True)
```

In [23]:
```
# Calculate the mean value for 'bore' column
avg_bore=df['bore'].astype('float').mean(axis=0)
print("Average of bore:", avg_bore)
```

```
Average of bore: 3.3307106598984775
```

In [24]:
```
# Replace NaN by mean value
df["bore"].replace(np.nan, avg_bore, inplace=True)
```

In [25]:
```
df['num-of-doors'].value_counts()
```

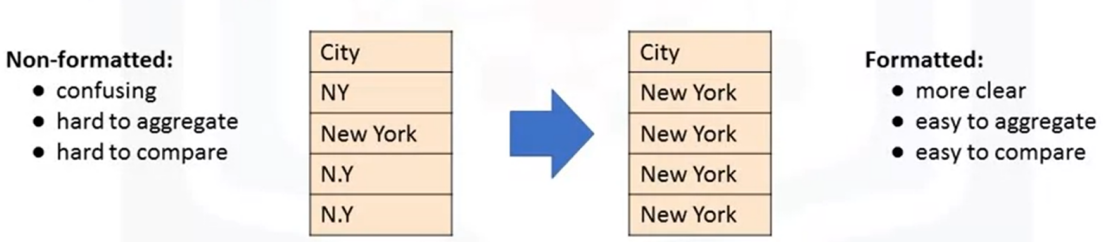
Out[25]:
```
four    113
two      86
Name: num-of-doors, dtype: int64
```

In [26]:
```
df['num-of-doors'].value_counts().idxmax()
```

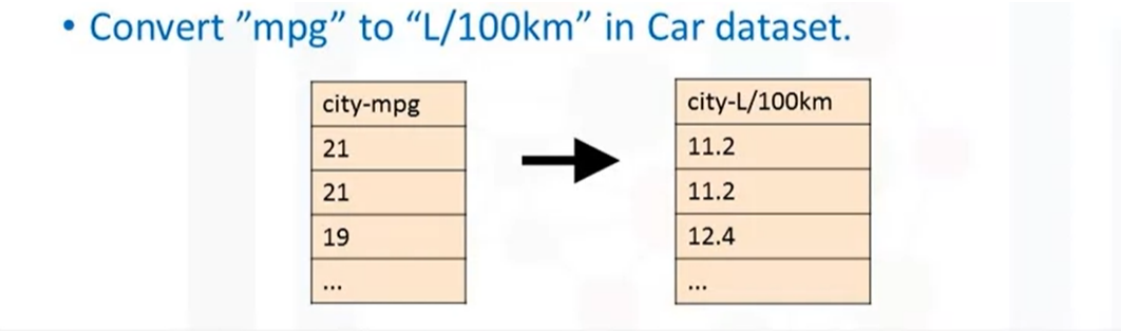
Out[26]:
```
'four'
```

Data Formatting

- Data is usually collected from different places by different people which may be stored in different formats.
- it means bringing data into a common standard of expression that allows users to make meaningful comparisons.
- As a part of dataset cleaning, data formatting ensures the data is consistent and easily understandable.
  - uppercase lowercase, dirty data



- - transform units of data




```
df[”city-mpg”]= 235/df[“city-mpg”]
df.rename(columns={”city_mpg”: “city-L/100km”}, inplace=True)
```

- incorrect data types: wrong data type is assigned to a feature.

In [ ]:
```
# to identify data types
df.dtypes()

# to convert data types
df.astype()
 # e.g. convert column "price" from object to integer
df["price"]=df["price"].astype("int")

# rename the column "city_mpg" to  "city-L/100km"
df.rename(columns={”city_mpg”: “city-L/100km”}, inplace=True)
```

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

In [27]:
```
# list the data types for each column
df.dtypes
```

Out[27]:
```
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
```

In [28]:
```
# Convert data types to proper format
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")
```

```
C:\Users\yeeyee\anaconda3\lib\site-packages\pandas\core\frame.py:3065: SettingWithCopyWarnin
g:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guid
e/indexing.html#returning-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/
user_guide/indexing.html#returning-a-view-versus-a-copy)
  self[k1] = value[k2]
```

In [ ]:
```
```

Data Normalization

- we may want to normalize these variables so that the range of the values is consistent.
- This normalization can make some statistical analyses easier down the road. By making the ranges consistent between variables, normalization enables a fair comparison between the different features, making sure they have the same impact, which is important for computational reasons.
  - Consider a data set containing two features, e.g. age and income. They are in different range of data. Where age ranges from 0-100, while income ranges from 0-20,000 and higher
  - When we do further analysis, like linear regression for example, the attribute income will intrinsically influence the result more due to its larger value. But this doesn't necessarily mean it is more important as a predictor. So, the nature of the data biases the linear regression model to weigh income more heavily than age.
  - To avoid this, we can normalize these two variables into values that range from zero to one.

age	income
20	100000
30	20000
40	500000

age	income
0.2	0.2
0.3	0.04
0.4	1

Not-normalized

- "age" and "income" are in different range.
- hard to compare
- "income" will influence the result more

Normalized

- similar value range.
- similar intrinsic influence on analytical model.

- several approaches for normalization:

① $x_{new} = \frac{x_{old}}{x_{max}}$ Simple Feature scaling

② $x_{new} = \frac{x_{old} - x_{min}}{x_{max} - x_{min}}$ Min-Max

③ $x_{new} = \frac{x_{old} - \mu}{\sigma}$ Z-score

___ values range from 0 to 1,____ values range from 0 to 1,___ range between -3 and +3 (can be higher or lower)

- simple feature scaling method

With Pandas:

length	width	height
168.8	64.1	48.8
168.8	64.1	48.8
180.0	65.5	52.4
...	...	...

→

length	width	height
0.81	64.1	48.8
0.81	64.1	48.8
0.87	65.5	52.4
...	...	...

```
df["length"] = df["length"]/df["length"].max()
```

- min-max method on the length feature

With Pandas:

length	width	height
168.8	64.1	48.8
168.8	64.1	48.8
180.0	65.5	52.4
...	...	...

→

length	width	height
0.41	64.1	48.8
0.41	64.1	48.8
0.58	65.5	52.4
...	...	...

```
df["length"] = (df["length"]-df["length"].min())/
               (df["length"].max()-df["length"].min())
```

- z-score method on length feature to normalize the values

With Pandas:

length	width	height
168.8	64.1	48.8
168.8	64.1	48.8
180.0	65.5	52.4
...	...	...

→

length	width	height
-0.034	64.1	48.8
-0.034	64.1	48.8
0.039	65.5	52.4
...	...	...

```
df["length"] = (df["length"]-df["length"].mean())/df["length"].std()
```

In [ ]:
```
# normalize the column "height"
df['height'] = df['height']/df['height'].max()

# show the scaled columns
df[["length","width","height"]].head()
```

Binning

Why binning?

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

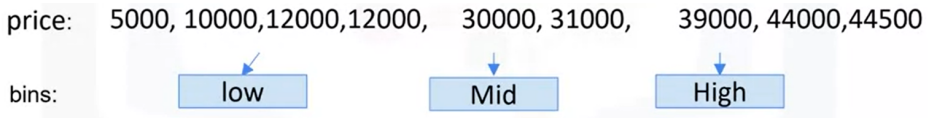
Example:

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

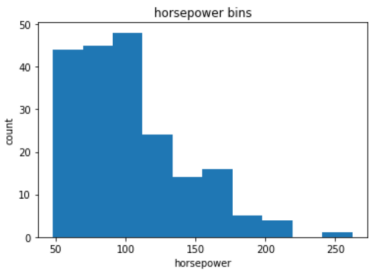
Data Binning

- group values together into bins, e.g. bin “age” into [0 to 5], [6 to 10], [11 to 15] and so on.
- improve accuracy of the predictive models.
- 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.
  - example:



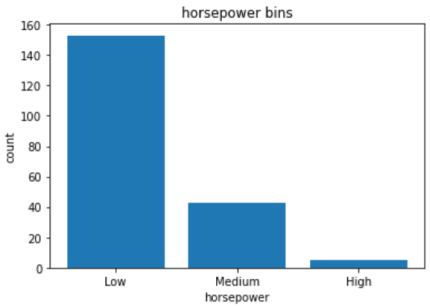
In [ ]:
```
# bin the data
bins=np.linspace( min(df["price"]), max(df["price"]), 4 )
gp_names=["Low", "Medium", "High"]

 # Example
# Convert data to correct format
df["horsepower"]=df["horsepower"].astype(int, copy=True)

# plot the histogram of horspower, to see what the distribution of horsepower looks like
%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")
```

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



In [ ]:
```
# build a bin array
bins = np.linspace(min(df["horsepower"]), max(df["horsepower"]), 4)
bins

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

# apply the function "cut" the determine what each value of "df['horsepower']" belongs to
df['horsepower-binned'] = pd.cut(df['horsepower'], bins, labels=group_names, include_lowest=Tr
df[['horsepower','horsepower-binned']].head(20)
df["horsepower-binned"].value_counts()

# plot the distribution of each bin
%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")
```

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



In [ ]:
```
# Bins visualization
%matplotlib inline
import matplotlib as plt
from matplotlib import pyplot
# 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")
```

Turning categorical variables into quantitative variables

- Problem: Most statistical models cannot take in objects or strings as input and for model training only take the numbers as inputs.
- Solution: categorical -> numeric Add dummy variables for each unique category Assign 0 or 1 in each category
- One-hot encoding

Car	Fuel	...	gas	diesel
A	gas	...	1	0
B	diesel	...	0	1
C	gas	...	1	0
D	gas	...	1	0

In [ ]:
```
# dummy methods
pd.get_dummies()
 # e.g.
pd.get_dummies(df["fuel"])
```

In [ ]:
```
# 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)
```

In [ ]:

Exploratory data analysis (EDA)

- an approach to analyze data
  - summarize main characteristics of the data
  - gain better understanding of the data set
  - uncover relationships between different variables
  - extract important variables for the problem we're trying to solve.
- The main question we are trying to answer is 'what are the characteristics (/features) that have the most impact on the car price?'
- use EDA techniques to answer this question

Descriptive statistics

- describe basic features of a dataset &
- obtain a short summary about the sample and measures of the data

In [12]:
```
import pandas as pd
import numpy as np

path='https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNet
df = pd.read_csv(path)
df.head()

#%%capture
#! pip install seaborn

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

In [ ]:
```
# summarize statistics
df.describe()   # mean, # of data pts, sd, quartiles & extreme values
                # skip all NaN values

df.describe(include=['object'])  # count, unique, top, freq

# summarize the categorical data/ gps
df.value_counts()
  # e.g.
drive_wheels_counts=df["drive_wheels"].value_counts.to_frame()
            |drive_wheels_counts
drive_wheels|
fwd         | 118
rwd         | 75
4wd         | 8

# box plots --- *outliers
 # e.g.
sns.boxplot(x="xxx", y="yyy", data=df)

# scatter plots --- each observation represented as a point
                --- the plot shows relationship between two variables
                --- x-axis: predictor/ independent variables
                --- y-axis: target/ dependent variables
 # e.g.
y=df["yyy"]
x=df["xxx"]
plt.scatter(x,y)

plt.title("zzz")
plt.xlabel("xxx")
plt.ylabel("yyy")

( sns.scatterplot(data=tips, x="total_bill", y="tip") )
```

Basic of grouping data

- help to transform our dataset

groupby()

- 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.

pivot()

- A pivot table has one variable displayed along the columns and the other variable displayed along the rows

heatmap

- takes a rectangular grid of data & assigns a color intensity based on the data value at the grid points.
- It is a great way to plot the target variable over multiple variables and through this get visual clues with the relationship between these variables and the target.

In [4]:
```
# groupby()
 # e.g.
# want to know # of categories of drive wheels
df['drive-wheels'].unique()
# select columns
df_group_one = df[['drive-wheels','body-style','price']]
# grouping results
df_group_one = df_group_one.groupby(['drive-wheels'],as_index=False).mean()
# how about group by both 'drive-wheels' and 'body-style'?
df_gptest = df[['drive-wheels','body-style','price']]
grouped_test1 = df_gptest.groupby(['drive-wheels','body-style'],as_index=False).mean()
grouped_test1
```

Out[4]:

	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

In [5]:
```
# pivot()
 # e.g.
grouped_pivot = grouped_test1.pivot(index='drive-wheels',columns='body-style')
grouped_pivot
```

Out[5]:

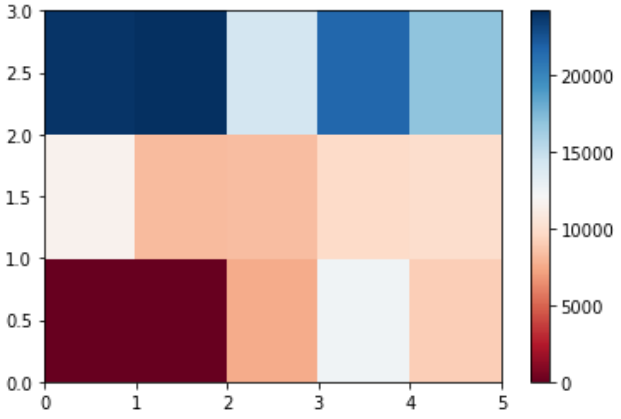
	price				
body-style	**convertible**	**hardtop**	**hatchback**	**sedan**	**wagon**
drive-wheels					
4wd	NaN	NaN	7603.000000	12647.333333	9095.750000
fwd	11595.0	8249.000000	8396.387755	9811.800000	9997.333333
rwd	23949.6	24202.714286	14337.777778	21711.833333	16994.222222

In [9]:
```
grouped_pivot = grouped_pivot.fillna(0) #fill missing values with 0
grouped_pivot
```

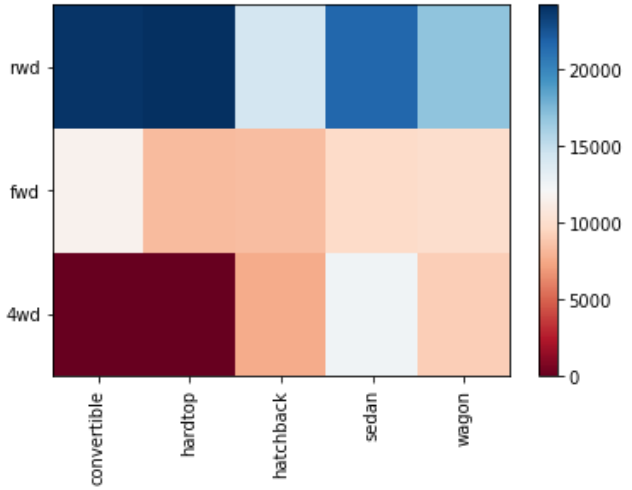
Out[9]:

	price				
body-style	**convertible**	**hardtop**	**hatchback**	**sedan**	**wagon**
drive-wheels					
4wd	0.0	0.000000	7603.000000	12647.333333	9095.750000
fwd	11595.0	8249.000000	8396.387755	9811.800000	9997.333333
rwd	23949.6	24202.714286	14337.777778	21711.833333	16994.222222

In [13]:
```
# heatmap
 # e.g.
plt.pcolor(grouped_pivot, cmap='RdBu')
plt.colorbar()
plt.show()
```



In [14]:
```
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()
```



Correlation between different variables

Correlation

- 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?
- example:
  - smoking is known to be correlated to lung cancer
    - since you have a higher chance of getting lung cancer if you smoke
  - a correlation between umbrella and rain variables
    - where more precipitation means more people use umbrellas.
    - Therefore, we can say that umbrellas and rain are interdependent and by definition they are correlated.
- correlation doesn't imply causation
  - In fact, we can say that umbrella and rain are correlated but we would not have enough information to say whether the umbrella caused the rain or the rain caused the umbrella.

In data science we usually deal more with correlation.

Example:

- look at the correlation between engine size and price.
- visualize these two variables using a scatter plot & an added linear line called a regression line
- The main goal of this plot is to see whether the engine size has any impact on the price

In [ ]:
```
import seaborn as sns
# Positive linear relationship
# 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
sns.regplot(x="engine-size", y="price", data=df)
plt.ylim(0,)

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

Recall

- Correlation: a measure of the extent of interdependence between variables.
- Causation: the relationship between cause and effect between two variables

Correlation - Statistical methods
Pearson correlation

- measure the strength of the correlation between continuous numerical variable
- it gives you two values:
  - the correlation coefficient
    - close to +1: large positive relationship
    - close to -1: large negative relationship
    - close to 0: no relationship
  - the P-value: the probability value that the correlation between these two variables is statistically significant.
    - p-value < 0.001 : strong certainty in the result
    - 0.001 <= p-value < 0.05: moderate certainty in the result
    - 0.05 <= p-value < 0.1 : weak certainty in the result
    - p-value > 0.1 : no certainty in the result
  - Strong Correlation:
    - correlation coefficient close to +1 & p-value < 0.001
    - correlation coefficient close to -1 & p-value < 0.001

Normally, we choose a significance level of 0.05, which means that we are 95% confident that the correlation between the variables is significant.

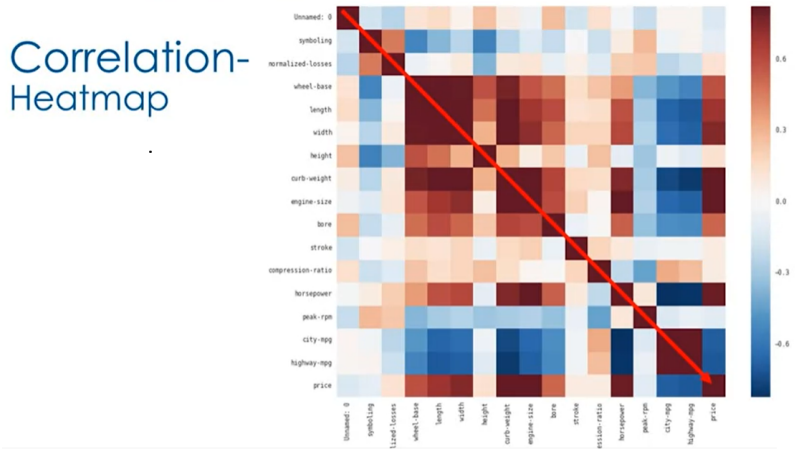
In [19]:
```
from scipy import stats
```

In [ ]:
```
# calculate the Pearson Correlation Coefficient and P-value of 'wheel-base' and 'price'
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_valu
```

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

Conclusion:

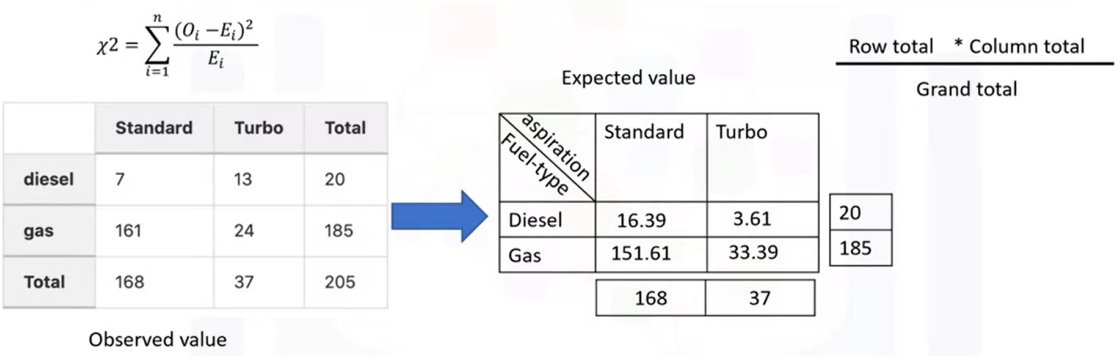
Since the p-value is < 0.001, the correlation between wheel-base and price is statistically significant, although the linear relationship isn't extremely strong (~0.585)



In [ ]:

Association between two categorical variables: Chi-Square

- When dealing with the relationships between two categorical variables, we can't use the same correlation method for continuous variables, i.e. Pearson correlation
- The Chi-square test is intended to test how likely it is that an observed distribution is due to chance.
- It measures how well the observed distribution of data fits with the distribution that is expected if the variables are independent.
- important points
  - The Chi-square tests null hypothesis is that the variables are independent.
  - The test compares the observed data to the values that the model expects if the data was distributed in different categories by chance.
  - Anytime the observed data doesn't fit within the model of the expected values, the probability that the variables are dependent becomes stronger, thus proving the null hypothesis incorrect.
  - The Chi-square does not tell you the type of relationship that exists between both variables only that a relationship exists.
- logic steps:
  - crosstab or contingency table is shows us the counts in each category
  - The summation of the observed value i.e., the counts in each group minus the expected value all squared divided by the Expected value (Expected values are based on the given totals)
  - To calculate the expected value of a standard car with diesel, We take the row total which is twenty multiplied by the column total one hundred and sixty-eight divided by the Grand total of two hundred and five



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.

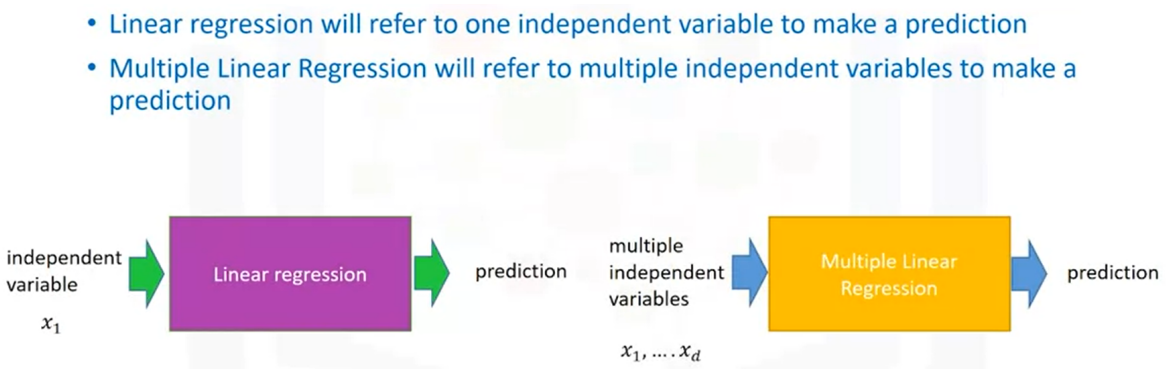
P-value: P-value tells how statistically significant is our calculated score value.

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.

In [ ]:

Model Development

Simple and multiple linear regression



Simple linear regression

- the predictor -- independent variable -- x
- the target -- dependent variable -- y
- y = b_0 + b_1*x
  - b_0: the intercept
  - b_1: the slope
- summarize the process:
  - We have a set of training points.
  - We use these training points to fit or train the model and get parameters. We then use these parameters in the model.
  - We now have a model. We use the hat on the y to denote the model is an estimate.
  - We can use this model to predict values that we haven't seen. But model is not always correct. We can see this by comparing the predicted value to the actual value.

In [ ]:
```
# import linear model from scikit learn
from sklearn.linear_model import LinearRegression

# create a Linear Regression object using the construstor
lm = LinearRegression()

# define x, y variables
X = df[['highway-mpg']]
Y = df['price']

# fit the model, i.e. fine the parameters b_0, b_1
lm.fit(X, Y)
# LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

# obtain a prediction
Yhat = lm.predict(X, Y)
Yhat[0:5]
# array([16236.50464347, 16236.50464347, 17058.23802179, 13771.3045085 , 20345.17153508])

# value of the intercept b_0
lm.intercept_

# value of the slope b_1
lm.coef_

# Train the model using 'engine-size' as the independent variable and 'price' as the dependent
lm1.fit(df[['engine-size']], df[['price']])
lm1
```

multiple linear regression

- the target -- dependent variable -- continuous variable -- y, one
- the predictors -- independent variables -- x, two or more
- y = b_0 + b_1*x_1* + *b_2*x_2 + b_3*x_3 + ... ,

- b_0: the intercept (X=0)
- b_1: the coefficient or parameter of x_1
- b_2: the coefficient or parameter of x_2
- ...

In [ ]:
```
# extract the for 4 predictor variables & store them in Z
Z = df[['highway-mpg', 'price', '...', ',,,']]

# fit the model, i.e. fine the parameters b_0, b_1
Im.fit(Z, df['price'])

# obtain a prediction
Yhat = Im.predict(X)
```

1. Find the intercept (b_0)

```
lm.intercept_
-15678.742628061467
```

2. Find the coefficients (b_1, b_2, b_3, b_4)

```
lm.coef_
array([52.65851272 ,4.69878948,81.95906216 , 33.58258185])
```

The Estimated Linear Model:

- **Price** = -15678.74 + (52.66) * **horsepower** + (4.70) * **curb-weight** + (81.96) * **engine-size** + (33.58) * **highway-mpg**

$$\hat{Y} = b_0 + b_1x_1 + b_2x_2 + b_3x_3 + b_4x_4$$

Regression Plot -- Model Evaluation using Visualization

Regression Plot

- reason of using Regression Plot
  - it gives us a good estimate of:
    - the relationship between two variables
    - the strength of the correlation
    - the direction of the relationship (+/-)
- the plots show us a combination of
  - the scatterplot
  - the fitted linear regression line

Residual plot

- A good way to visualize the variance of the data
- is a graph that shows the residuals on the vertical y-axis and the independent variable on the horizontal x-axis
- represents the error between the actual value
  - The difference between the observed value (y) and the predicted value (Yhat) is called the residual (e)
- can be a curvature

Distribution plot

- counts the predicted value versus the actual value.
- These plots are extremely useful for visualizing models with more than one independent variable or feature

In [ ]:
```
import seaborn as sns

# Regression Plot
sns.regplot(x='highway-mpg', y='price', data=df)
plt.ylim(0,)
##result a plot

 # e.g.
# Regression Plot
width = 12
height = 10
plt.figure(figsize=(width, height))
sns.regplot(x="highway-mpg", y="price", data=df)
plt.ylim(0,)

#-----------------------------------------------------
# Residual plot
sns.residplot(df['highway-mpg'], df['price'])
##result a plot

 # e.g.
width = 12
height = 10
plt.figure(figsize=(width, height))
sns.residplot(df['highway-mpg'], df['price'])
plt.show()

#-----------------------------------------------------
# Distribution plot
axl = sns.distplot(df['price'], hist=False, color="r", label="Actual Value")
sns.distplot(Yhat, hist=False, color="b", label="Fitted Value", ax=axl)

 # e.g.
# Multiple Linear Regression
Y_hat = lm.predict(Z)

plt.figure(figsize=(width, height))


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

What do we do when a linear model is not the best fit for our data?

Polynomial regression and pipelines

Polynomial regression

- a special case of the general linear regression model
- useful for describing curvilinear relationships
  - by squaring or setting higher-order terms of the predictor variables
    - quadratic -- 2nd order: Y^ = b_0 + b_1*x_1* + *b_2*(x_1)^2
    - cubic -- 3rd order: Y^ = b_0 + b_1*x_1* + *b_2*(x_1)^2 + b_3*(x_1)^3
    - higher order: Y^ = b_0 + b_1*x_1* + *b_2*(x_1)^2 + b_3*(x_1)^3* + *b_4*(x_1)^4 + ...
- with more than one dimension
  - e.g. multi-dimensional polynomial linear regression:
  - Y^ = b_0 + b_1*x_1* + *b_2x_2* + *b_3x_1*x_2* + *b_4*(x_1)^2 + b_5*(x_2)^2 + ...

In [ ]:

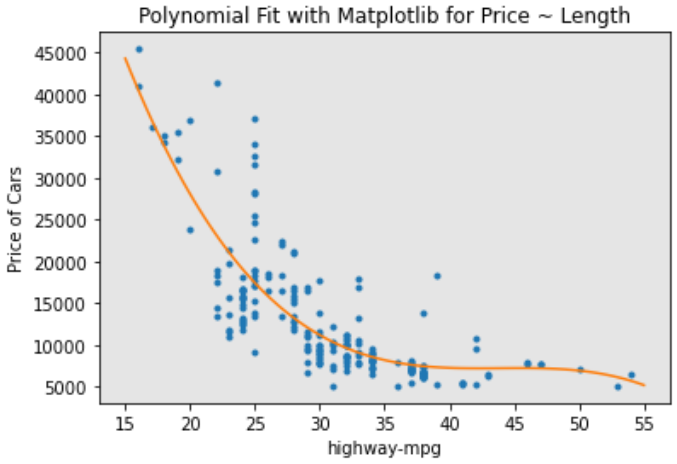
```
# calculate poly. of 3rd order
f=np.polyfit(x,y,3)
p=np.polyld(f)

# print out the model
print(p)

# multi-dimensional polynomial linear regression
from sklearn.preprocessing import PolynomialFeatures
pr=PolynomialFeatures( degree=2, include_bias=False ) # transform np arrays to polynomial feat
                                                      # with the fit underscore transform meth
pr.fit_transform([[1,2]])
```

- For example we can Normalize the each feature simultaneously

```
from sklearn.preprocessing import StandardScaler

SCALE=StandardScaler()

SCALE.fit(x_data[['horsepower', 'highway-mpg']])

x_scale=SCALE.transform(x_data[['horsepower', 'highway-mpg']])
```

In [21]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

path = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsN
df = pd.read_csv(path)
df.head()

from sklearn.linear_model import LinearRegression
# import the visualization package: seaborn
import seaborn as sns
%matplotlib inline

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()

x = df['highway-mpg']
y = df['price']

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

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

np.polyfit(x, y, 3)
```

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

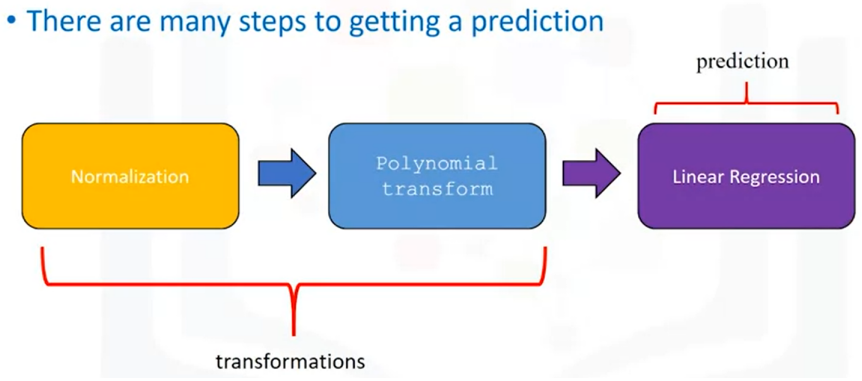


Out[21]:
```
array([-1.55663829e+00,  2.04754306e+02, -8.96543312e+03,  1.37923594e+05])
```

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

pr=PolynomialFeatures(degree=2)
pr

Z_pr=pr.fit_transform(Z)

Z.shape  # The original data is of 201 samples and 4 features

Z_pr.shape  # after the transformation, there 201 samples and 15 features
```

Pipelines -- pipeline library

- used for simply a code
- Then,
  - There are many steps to getting a prediction.
  - For example, normalization, polynomial transform, and linear regression.
  - It sequentially perform a series of transformations.
  - The last step carries out a prediction.



In [ ]:
```
# import all the modules we need
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler

# import the library pipeline
from sklearn.pipeline import Pipeline

# We create a list of tuples
Input=[ ('scale', StandardScaler()), ('Polynomial', PolynomialFeatures(degree=2), ..., ('mode'
# the first element in the tuple contains the name of the estimator model
# the second element contains model constructor

# We input the list in the pipeline constructor
pipe=Pipeline(Input)   # We now have a pipeline object.

# We can train the pipeline by applying the train method to the pipeline object.
# We can also produce a prediction as well.
Pipe.fit( df[['xxx', 'yyy', 'zzz']],y )
yhat=Pipe.predict( df[['xxx', 'yyy', 'zzz']],y )
# The method normalizes the data, performs a polynomial transform, then outputs a prediction.
```

In [24]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

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

Input=[('scale',StandardScaler()), ('polynomial', PolynomialFeatures(include_bias=False)), ('m

pipe=Pipeline(Input)
pipe

pipe.fit(Z,y)

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

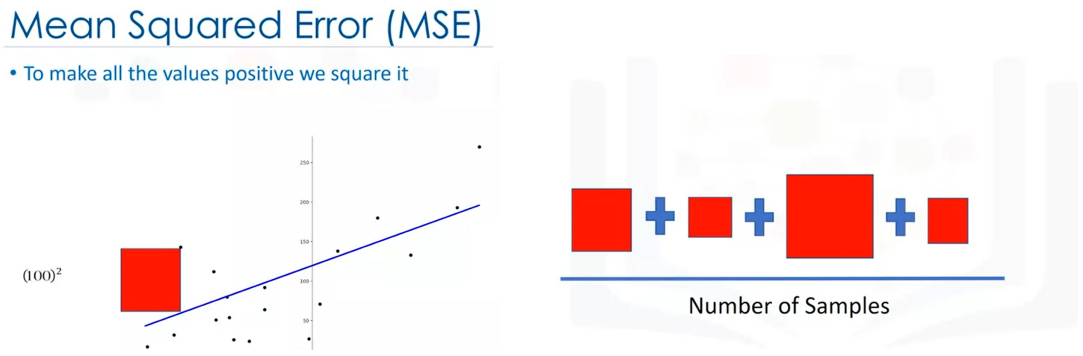
Out[24]:
```
array([13102.74784201, 13102.74784201, 18225.54572197, 10390.29636555])
```

R-squared and Mean Suared Error for in-sample evaluation

Measures for in-sample evaluation is a way to numerically determine how good the model fits on dataset

Two important measures to determine the fit of a model

- Mean Suared Error (MSE)
  - To measure the MSE, we find the difference between the actual value y and the predicted value yhat then square it.

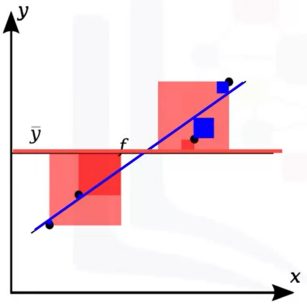


- R-squared
  - is also called the coefficient of determination.
  - It's a measure to determine how close the data is to the fitted regression line.
  - So how close is our actual data to our estimated model? Think about it as comparing a regression model to a simple model, i.e., the mean of the data points.

$$R^2 = \left(1 - \frac{\text{MSE of regression line}}{\text{MSE of the average of the data}}\right)$$

  - For the most part, it takes values between 0 and 1
  - Interpret R-square (x 100) as the percentage of the variation in the response variable y that is explained by the variation in explanatory variable(s) x

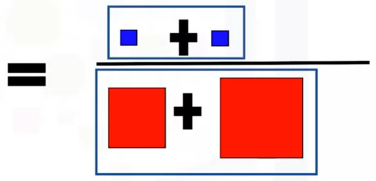
Coefficient of Determination (R^2)

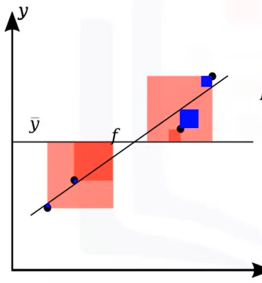


- The blue line represents the regression line
- The blue squares represents the MSE of the regression line
- The red line represents the average value of the data points
- The red squares represent the MSE of the red line
- We see the area of the blue squares is much smaller than the area of the red squares

- In this case ratio of the areas of MSE is close to zero

$$\frac{\text{MSE of regression line}}{\text{MSE of } \bar{y}} = 0$$



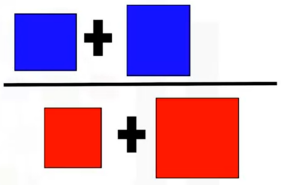


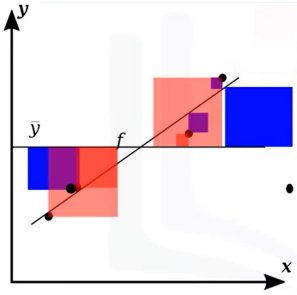
$$R^2 = \left(1 - \frac{\text{MSE of regression line}}{\text{MSE of } \bar{y}}\right)$$

$$= (1 - 0)$$

$$= 1$$

$$\frac{\text{MSE of regression line}}{\text{MSE of } \bar{y}} = 1$$





$$R^2 = \left(1 - \frac{\text{MSE of regression line}}{\text{MSE of } \bar{y}}\right)$$

$$= (1 - 1)$$

$$= 0$$

- 1: the line is good fit for the data

In [ ]:
```
# measure MSE
from sklearn.metrics import mean_squared_error
mean_squared_error( df['price'], Y_predict_simple_fit )
# 3163502.944639888

# find the R-squared value in Python by using the score method, in the linear regression objec
X = df[['xxx']]
Y = df['yyy']

Im.fit(X, Y)

Im.score(X, y)     # R^2 value is usually between 0 and 1.
                   # If your R^2 is negative, it can be due to over fitting

# calculate the R^2
from sklearn.metrics import r2_score
r_squared = r2_score(y, p(x))
print('The R-square value is: ', r_squared)
# The R-square value is:  0.674194666390652

 # e.g.
X1 = df[['sqft_living']]
Y1 = df['price']
lm = LinearRegression()
lm.fit(X1,Y1)
lm.score(X1, Y1)  # 0.4928532179037931 = R^2
```

Prediction and Decision Making

How can we determine if our model is correct?

- do the predicted values make sense
- visualization
- numerical measures for evaluation
- comparing models

Do the predicted values make sense

- First we train the model

```
lm.fit(df['highway-mpg'],df['prices'])
```

- Let's predict the price of a car with 30 highway-mpg.

```
lm.predict(np.array(30.0).reshape(-1,1))
```

- Result: $ 13771.30

```
lm.coef_
-821.73337832
```

  - **Price** = 38423.31 - 821.73 * **highway-mpg**

- First we import numpy

```
import numpy as np
```

- We use the numpy function arrange to generate a sequence from 1 to 100

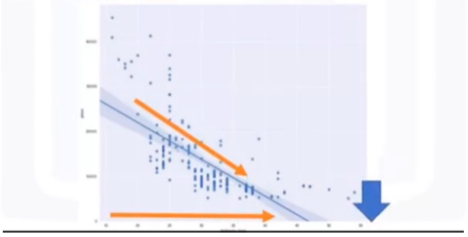
```
new_input=np.arange(1,101,1).reshape(-1,1)
```

1	2	...	99	100

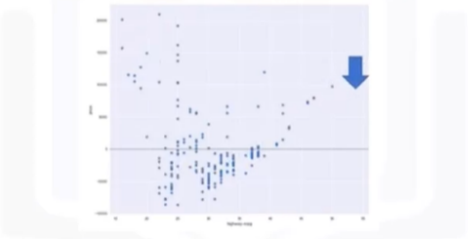
- We can predict new values

```
yhat=lm.predict(new_input)
```

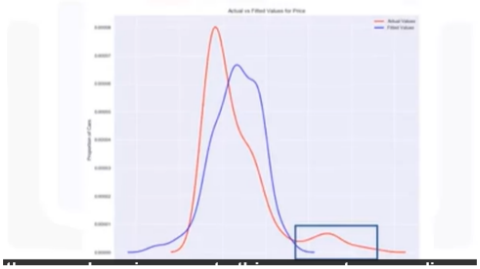
• Simply visualizing your data with a regression



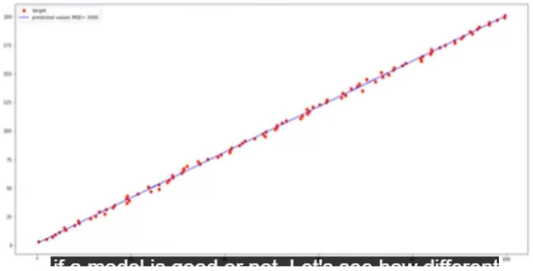
Residual Plot

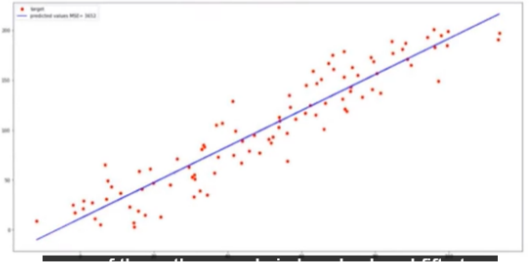


Visualization



Numerical measures for Evaluation





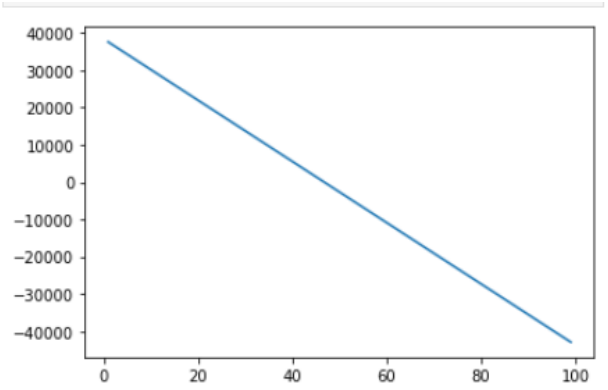
Comparing MLR and SLR

Does a lower Mean Square Error imply better fit?

- Not necessarily

1. Mean Square Error for a Multiple Linear Regression Model will be smaller than the Mean Square Error for a Simple Linear Regression model, since the errors of the data will decrease when more variables are included in the model
2. Polynomial regression will also have a smaller Mean Square Error than the linear regular regression
3. In the next section we will look at more accurate ways to evaluate the model

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

%matplotlib inline

# Create a new input
new_input=np.arange(1, 100, 1).reshape(-1, 1)

# Fit the model
lm.fit(X, Y)
lm   # LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
     #   normalize=False)

# Produce a prediction
yhat=lm.predict(new_input)
yhat[0:5]
# array([37601.57247984, 36779.83910151, 35958.10572319, 35136.37234487,
       34314.63896655])

# plot the data
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?
- What is a good MSE?

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

1 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

- MSE: The 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).
- MLR >> SLR

2 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.
- Polynomial Fit >> SLR

3 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.
- => we conclude that the MLR model is the best model to be able to predict price from our dataset

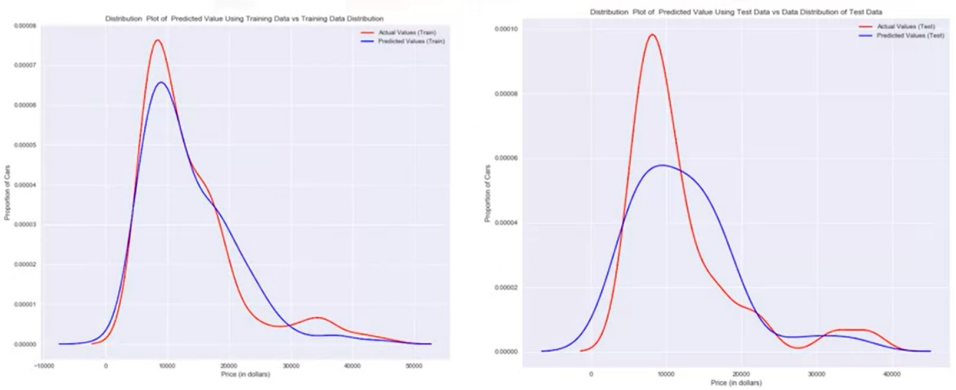
Model Evaluation and Refinement

- Recall: In-sample evaluation -- how well our model fits the data already given to train it.
  - Problems? It does not give us an estimate of how well the train model can predict new data.
  - The solution:
    - To split our data up, use the in-sample data or training data to train the model.
    - The rest of the data, called Test Data, is used as out-of-sample data. This data is then used to approximate, how the model performs in the real world.
- Separating data into training and testing sets (important part of model evaluation)
  - training: 70%
  - testing: 30%
  - => use training set to build a model and discover predictive relationships.
  - => use a testing set to evaluate model performance.
  - ==> When we have completed testing our model, we should use all the data to train the model.

In [ ]:

In [ ]:
```
# scikit-learn package for splitting datasets
# split data into random train and test subsets
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=0.3, random_stat
# - x_data: features or independent variables
# - y_data: dataset target, e.g. df['price'], above is to try to predict the price
# => output is array
# {x_train, y_train} is the subsets of training
# {x_test, y_test} is the subsets of testing
# The random state is a random seed for random data set splitting.
```

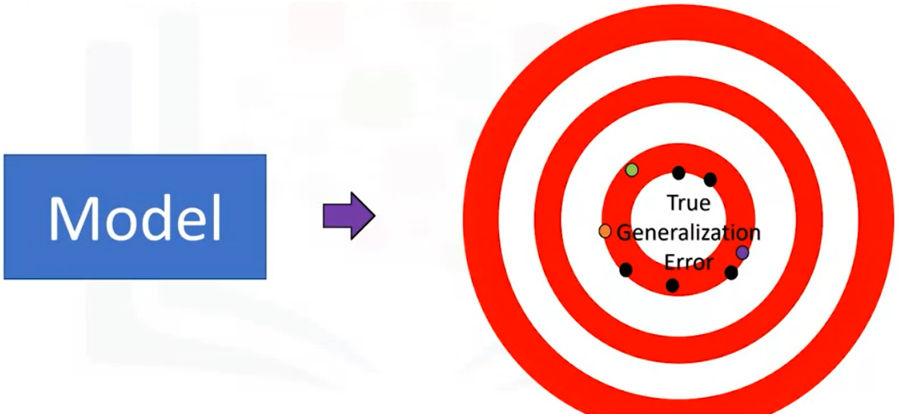
- Generalization Performence
  - Generalization error is a measure of how well our data does at predicting previously unseen data.
  - The error we obtain using our testing data is an approximation of this error.



  - This figure shows the distribution of the actual values in red compared to the predicted values from a linear regression in blue. We see the distributions are somewhat similar.

- If we generate the same plot using the test data, we see the distributions are relatively different. The difference is due to a generalization error and represents what we see in the real world.

- [con't]
  - Using a lot of data for training ->
    - gives us an accurate means of determining how well our model will perform in the real world.
    - BUT the precision of the performance will be low.
    - an example: 90 percent of the data for training and 10 percent for testing



- [con't]
  - Using fewer data points for training & more to test the model ->
    - the accuracy of the generalization performance will be less, but the model will have good precision.
    - figure:
    - All our error estimates are relatively close together, but they are further away from the true generalization performance.

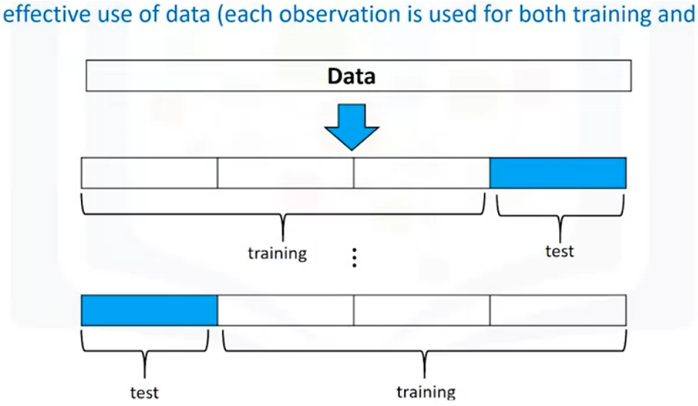
Problem causes!!!

To overcome this --> use cross-validation

cross-validation

- most common out of sample evaluation metrics
- concept:
  - the dataset is split into K equal groups.
  - Each group is referred to as a fold.
  - eg below:

- More effective use of data (each observation is used for both training and testing)



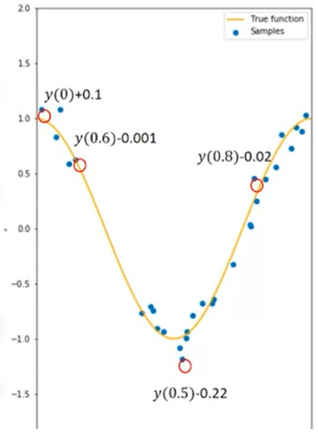
  - Use three folds for training, then use one fold for testing.
  - This is repeated until each partition is used for both training and testing.
  - At the end, we use the average results as the estimate of out-of-sample error.
- The evaluation metric depends on the model, for example, the r squared.

In [ ]:
```
# cross-validation
# cross_val_score function, which performs multiple out-of-sample evaluations.
from sklearn.model_selection import cross_val_score
scores = cross_val_score (lr, x_data, y_data, cv=3)
                          |                    |
# the type of model we are using    We can manage the number of partitions
# to do the cross-validation        with the cv parameter.e.g.3 for the data set
#                                   is split into three equal partitions.

# The function returns an array of scores, one for each partition that was chosen as the testi
# We can average the result together to estimate out of sample r squared using the mean functi
np.mean(scores)


# if we want to know the actual predicted values supplied by our model before the r squared va
from sklearn.model_selection import cross_val_predict
yhat = cross_val_predict (lr2e, x_data, y_data, cv=3)   # output is prediction
# These predictions are stored in an array

# use negative squared error as a score by setting the parameter 'scoring' metric to 'neg_mean
-1 * cross_val_score(lre,x_data[['horsepower']], y_data,cv=4,scoring='neg_mean_squared_error')
```

- Overfitting, Underfitting and Model Selection
  - recall: polynomial regression
- Model Selection
  - discuss how to pick the best polynomial order and problems that arise when selecting the wrong order polynomial.

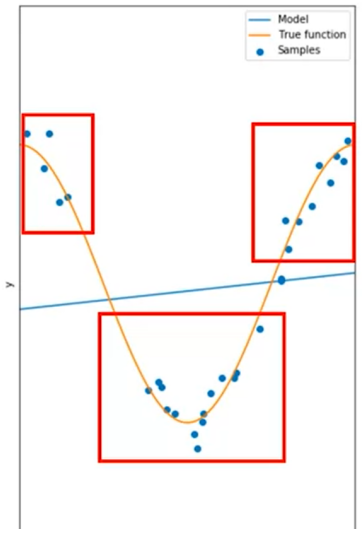
$y(x)$+noise

Consider the following function, we assume the training points come from a polynomial function plus some noise. The goal of Model Selection is to determine the order of the polynomial to provide the best estimate of the function y(x).



- Then,

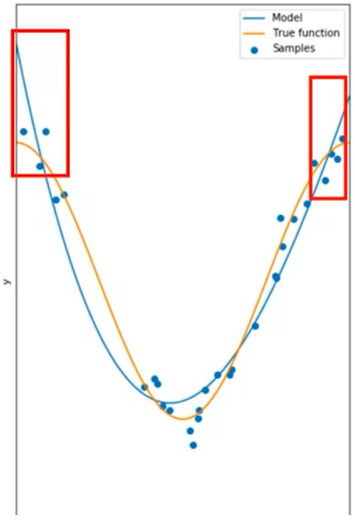
$$y = b_0 + b_1 x$$



- If we try and fit the function with a linear function, the line is not complex enough to fit the data.
- As a result, there are many errors.
- This is called underfitting, where the model is too simple to fit the data.

- then,

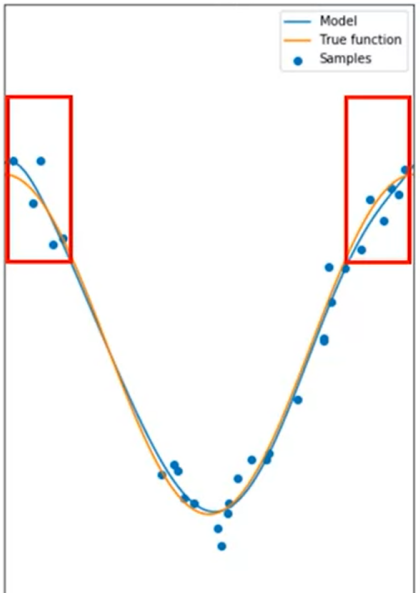
$$y = b_0 + b_1 x + b_2 x^2$$



- If we increase the order of the polynomial, the model fits better, but the model is still not flexible enough and exhibits underfitting.

- then,

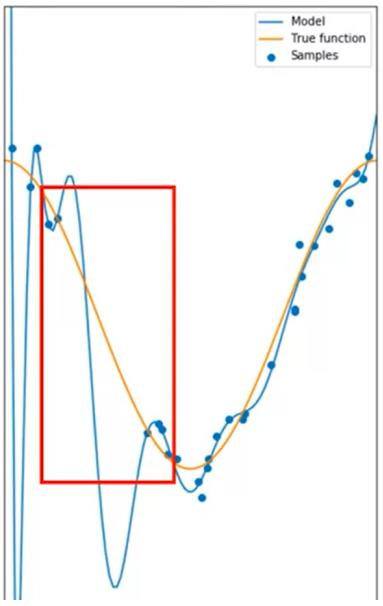
$$\hat{y} = b_0 + b_1 x + b_2 x^2 + b_3 x^3 + b_4 x^4 + b_5 x^5 + b_6 x^6 + b_7 x^7 + b_8 x^8$$



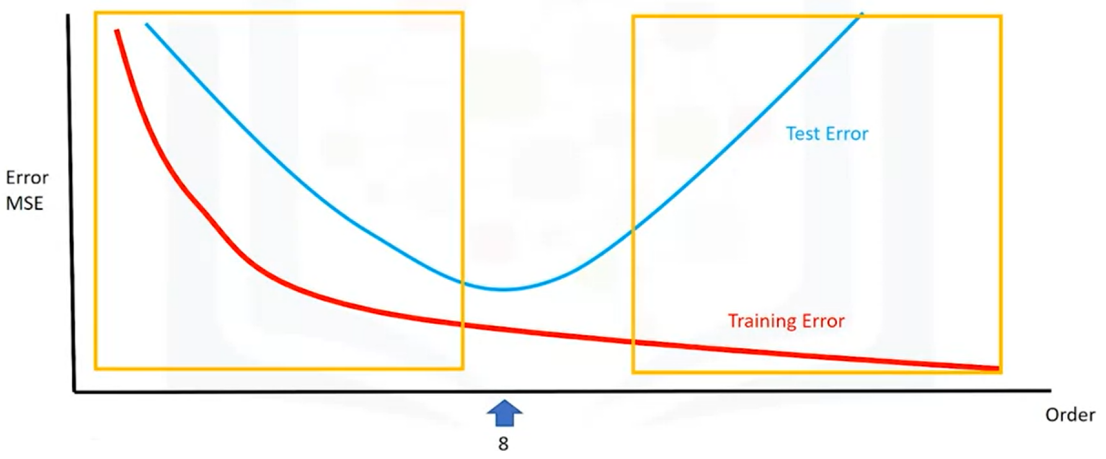
- This is an example of the 8th order polynomial used to fit the data.
- We see the model does well at fitting the data and estimating the function even at the inflection points.

- then,

$$\hat{y} = b_0 + b_1 x + b_2 x^2 + b_3 x^3 + b_4 x^4 + b_5 x^5 + b_6 x^6 + b_7 x^7 + b_8 x^8 + ..$$
$$+ b_9 x^9 + b_{10} x^{10} + b_{11} x^{11} + b_{12} x^{12} + b_{13} x^{13} + b_{14} x^{14} + b_{15} x^{15} + b_{16} x^{16}$$



  - Increasing it to a 16th order polynomial, the model does extremely well at tracking the training point but performs poorly at estimating the function. This is especially apparent where there is little training data.
  - The estimated function oscillates not tracking the function.
  - This is called overfitting, where the model is too flexible and fits the noise rather than the function.

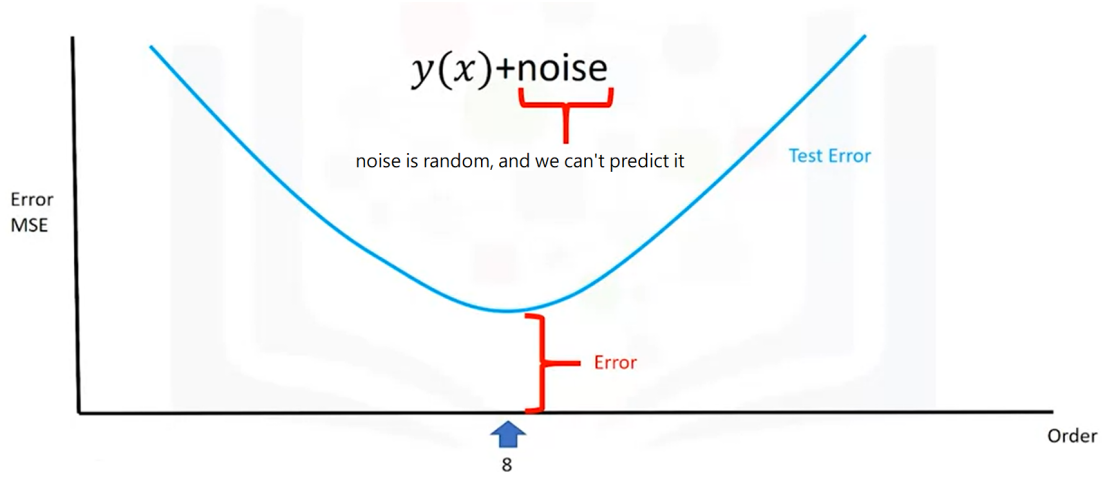


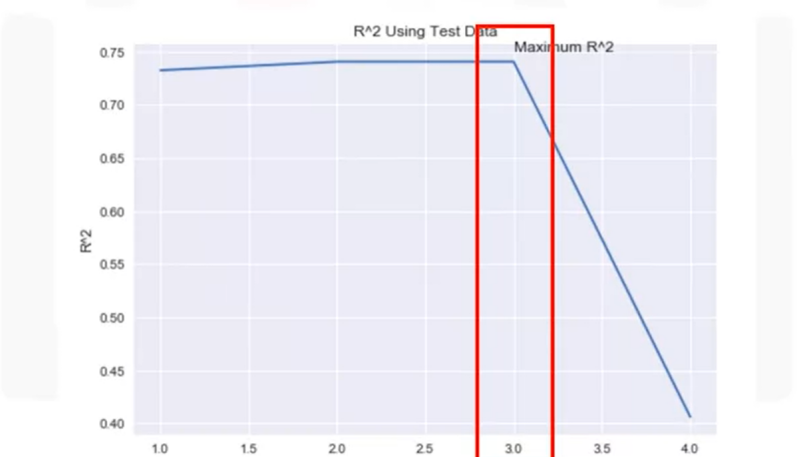
- The horizontal axis represents the order of the polynomial. - The vertical axis is the mean square error. The training error decreases with the order of the polynomial.
- => The test error is a better means of estimating the error of a polynomial.
- i.e., The error decreases 'till the best order of the polynomial is determined.

Then the error begins to increase.

We select the order that minimizes the test error. In this case, it was eight.

- Anything on the left would be considered underfitting.
- Anything on the right is overfitting.





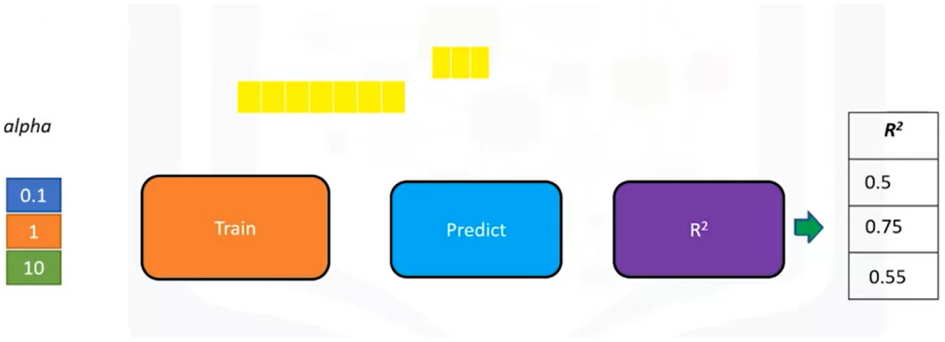
- The following is a plot of the R-squared value.
- The horizontal axis represents the order polynomial models.
- The closer the R-squared is to one, the more accurate the model is.
- Here, we see the R-squared is optimal when the order of the polynomial is three.
- The R-squared drastically decreases when the order is increased to four, validating our initial assumption.

In [ ]:
```
# First, we create an empty list to store the values.
Rsqu_test=[]

# We create a list containing different polynomial orders.
order=[1,2,3,4]

# We then iterate through the list using a loop.
# We create a polynomial feature object with the order of the polynomial as a parameter. We tr
# We fit the regression model using the transform data.
# We then calculate the R-squared using the test data and store it in the array.

for n in order:
    pr=PolynomialFeatures(degree=n)
    x_train_pr=pr.fit_transform(x_train[['xxx']])
    x_test_pr=pr.fit_transform(x_test[['xxx']])
    lr.fit(x_train_pr, y_train)
    Rsqu_test.append(lr.score(x_test_pr, y_test))
```

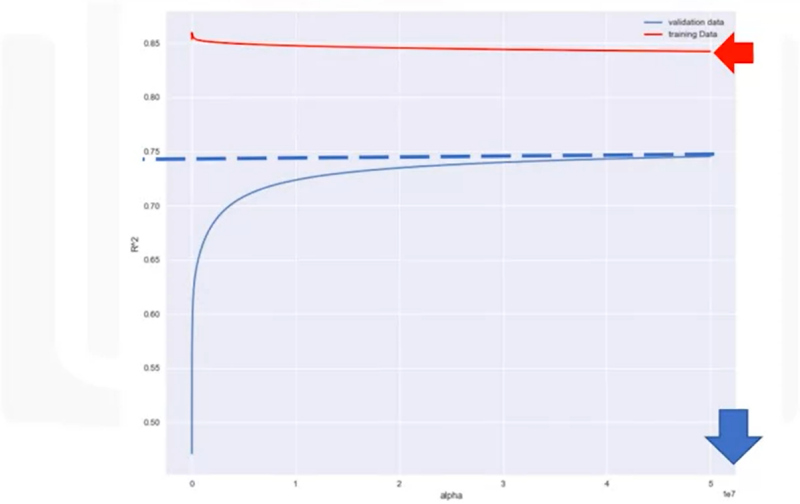
- Ridge regression
  - a regression that is employed in a Multiple regression model when Multicollinearity occurs.
  - Multicollinearity is when there is a strong relationship among the independent variables.
- It is very common with polynomial regression.
- BELOW shows how Ridge regression is used to regularize and reduce the standard errors to avoid over-fitting a regression model

Apply Ridge Regression to linear regression models

In [ ]:
```
# To make a prediction using ridge regression, import ridge from sklearn.linear_models.
from sklearn.linear_model import Ridge

# Create a ridge object using the constructor.
RidgeModel=Ridge(alpha=0.1)   # The parameter alpha is one of the arguments of the constructor

# We train the model using the fit method.
RidgeModel.fit(X,y)

# we use the predict method.
Yhat=RidgeModel.predict(X)


# select the value of Alpha that minimizes the test error by using a for loop
Rsqu_test = []
Rsqu_train = []
dummy1 = []
Alpha = 10 * np.array(range(0,1000))
for alpha in Alpha:
    RigeModel = Ridge(alpha=alpha)
    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))

width = 12
height = 10
plt.figure(figsize=(width, height))

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

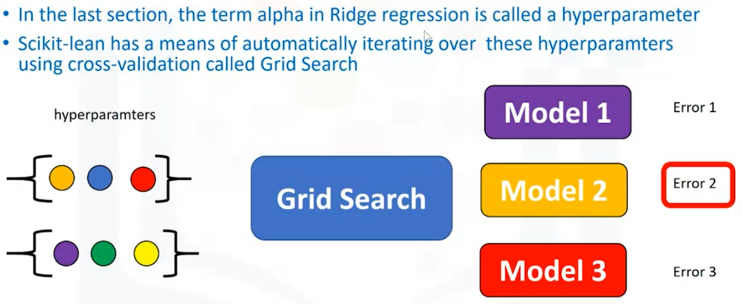


- In order to determine the parameter alpha, we use some data for training. We use a second set called validation data, which is similar to test data, but it is used to select parameters like alpha.
- We start with a small value of alpha.
- We train the model,
- make a prediction using the validation data,
- then calculate the R-squared and store the values.
- We repeat the process for a different alpha value, and finally we can make a prediction.
- We select the value of alpha that maximizes the R-squared. - - *Note that we can use other metrics to select the value of alpha, like mean squared error.



- The overfitting problem is even worse if we have lots of features.
- The following plot shows the different values of R-squared on the vertical axis. The horizontal axis represents different values for alpha.
- We use several features from our used car data set and a second order polynomial function.
- The training data is in red and validation data is in blue.
- We see as the value for alpha increases, the value of R-squared increases and converges at approximately 0.75.
- In this case, we select the maximum value of alpha because running the experiment for higher values of alpha have little impact. Conversely, as alpha increases, the R-squared on the test data decreases. This is because the term alpha prevents overfitting. This may improve the results in the unseen data, but the model has worse performance on the test data.

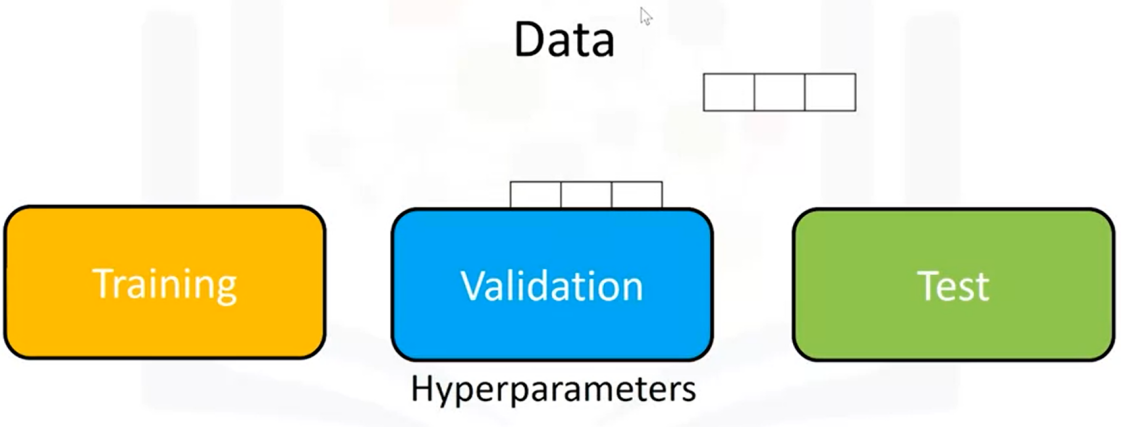
- Grid Search



Grid Search takes the model or objects you would like to train and different values of the hyperparameters.
It then calculates the mean square error or R-squared for various hyperparameter values, allowing you to choose the best values.

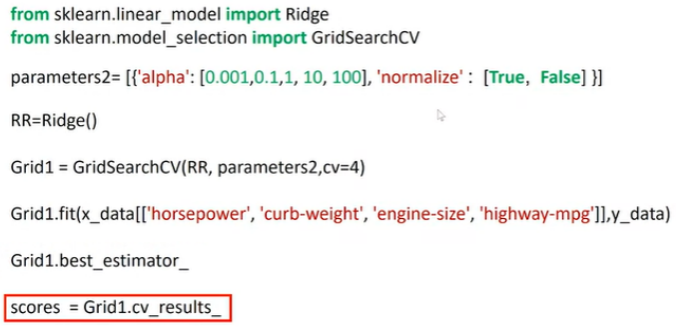
  - Let the small circles represent different hyperparameters.
  - We start off with one value for hyperparameters and train the model. We use different hyperparameters to train the model.
  - We continue the process until we have exhausted the different free parameter values.
  - Each model produces an error. We select the hyperparameter that minimizes the error.

-



- To select the hyperparameter

- we split our dataset into three parts,
  - the training set: We train the model for different hyperparameters. We use the R-squared or mean square error for each model.
  - validation set: We select the hyperparameter that minimizes the mean squared error or maximizes the R-squared on the validation set
  - test set: We finally test our model performance using the test data.



```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

parameters2= [{'alpha': [0.001,0.1,1, 10, 100], 'normalize' : [True, False] }]

RR=Ridge()

Grid1 = GridSearchCV(RR, parameters2,cv=4)

Grid1.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],y_data)

Grid1.best_estimator_

scores = Grid1.cv_results_
```

In [ ]:

```
# concept: R-squared the number of folds, the model or object, and the free parameter values.
#          Some of the outputs include the different scores for different free parameter value
#  the R-squared along with a free parameter values that have the best score

from sklearn.model_selection import GridSearchCV

# create a dictionary of parameter values
parameters1= [{'alpha': [0.001,0.1,1, 10, 100, 1000, 10000, 100000, 100000]}]
parameters1

# Create a ridge regions object
RR=Ridge()
RR

# Create a ridge grid search object
Grid1 = GridSearchCV(RR, parameters1,cv=4)

# Fit the model
Grid1.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y_data)

# obtain the estimator with the best parameters and assign it to the variable BestRR
# since we find the best parameter values on the validation data
BestRR=Grid1.best_estimator_
BestRR

# test our model on the test data
BestRR.score(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y_test)
# 0.6373308904159379
```

Tune hyper-parameters of an estimator using Grid search: Grid search is a time-efficient tuning technique that exhaustively computes the optimum values of hyperparameters performed on specific parameter values of estimators.

Course Creators

Joseph has a Ph.D. 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.

Mahdi Noorian Ph.D., is a Postdoctoral Fellow at the Laboratory for Systems, Software and Semantics (LS3) of the Ryerson University. He holds a Ph.D degree in Computer Science from University of New Brunswick. As a Data Scientist, he is interested in application of machine learning, data mining, optimization, and semantic data analysis for big data to solve the real-world problems.