IBM – Data Analysis With Python

Module 1 : Introduction

Understanding the Data

In this video we'll be looking at the dataset on used car prices. The dataset used in this course is an open dataset, by Jeffrey C. Schlimmer. This dataset is in CSV format, which separates each of the values with commas, making it very easy to import in most tools or applications.

Each line represents a row in the dataset.

In the hands-on lab for this module, you'll be able to download and use the CSV file.

Do you notice anything different about the first row?

Sometimes the first row is a header which contains a column name for each of the 26 columns.

But in this example, it's just another row of data.

So here's the documentation on what each of the 26 columns represent.

1	symboling	-3, -2, -1, 0, 1, 2, 3.	14	curb-weight	continuous from 1488 to 4066.
2	normalized-losses	continuous from 65 to 256.	15	engine-type	dohc, dohcv, l, ohc, ohcf, ohcv, rotor.
3	make	audi, bmw, etc.	16	num-of-cylinders	eight, five, four, six, three, twelve, two.
4	fuel-type	diesel, gas.	17	engine-size	continuous from 61 to 326.
5	aspiration	std, turbo.	18	fuel-system	1bbl, 2bbl, 4bbl, idi, mfi, mpfi, spdi, spfi.
6	num-of-doors	four, two.	19	bore	continuous from 2.54 to 3.94.
7	body-style	hardtop, wagon, etc.	20	stroke	continuous from 2.07 to 4.17.
8	drive-wheels	4wd, fwd, rwd.	21	compression-ratio	continuous from 7 to 23.
9	engine-location	front, rear.	22	horsepower	continuous from 48 to 288.
10	wheel-base	continuous from 86.6 120.9.	23	peak-rpm	continuous from 4150 to 6600.
11	length	continuous from 141.1 to 208.1.	24	city-mpg	continuous from 13 to 49.
12	width	continuous from 60.3 to 72.3.	25	highway-mpg	continuous from 16 to 54.
13	height	continuous from 47.8 to 59.8.	26	price	continuous from 5118 to 45400.

There are a lot of columns, and I'll just go through a few of the column names, but you can also check out the link at the bottom of the slide to go through the descriptions yourself.

The first attribute (column), "symboling", corresponds to the insurance risk level of a car. Cars are initially assigned a risk factor symbol associated with their price. Then, if an automobile is more risky, this symbol is adjusted by moving it up the scale. A value of +3 indicates that the auto is risky, -3 that it is probably pretty safe.

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/speciality, etc...), and represents the average loss per car per year. The values range from 65 to 256.

The other attributes are easy to understand. If you would like to check out more details, refer to the link at the bottom of the slide.

Ok, after we understand the meaning of each feature, we'll notice that the 26th attribute is "price".

This is our target value, or label, in other words.

This means "price" is the value that we want to predict from the dataset, and the predictors should be all the other variables listed, like "symboling", "normalized-losses", "make" and so on.

Thus, the goal of this project is to predict "price" in terms of other car features.

Just a quick note, this dataset is actually from 1985, so the car prices for the models may seem a little low, but just bear in mind that the goal of this exercise is to learn how to analyze the data.

The Problem

In this video, we'll be talking about data analysis and the scenario in which we'll be playing the data analyst or data scientist. But before we begin talking about the problem (used car prices), we should first understand the importance of data analysis.

As you know, data is collected everywhere around us, whether it's collected manually by scientists, or collected digitally every time you click on a website or your mobile device.

But data does not mean information.

Data analysis, and in essence, data science, helps us unlock the information and insights from raw data, to answer our questions.

So data analysis plays an important role by helping us to discover useful information from the data, answer questions, and even predict the future or the unknown.

So let's begin with our scenario.

Let's say we have a friend named Tom. And Tom wants to sell his car.

But the problem is, he doesn't know how much he should sell his car for. Tom wants to sell his car for as much as he can. But he also wants to set the price reasonably so someone would want to purchase it. So the price he sets should represent the value of the car.

How can we help Tom determine the best price for his car?

Let's think like data scientists and clearly define some of his problems:

For example, is there data on the prices of other cars and their characteristics?
What features of cars affect their prices?
Colour?
Brand?
Does horsepower also affect the selling price, or perhaps, something else?

As a data analyst or data scientist, these are some of the questions we can start thinking about.

To answer these questions, we're going to need some data.

In the next videos, we'll be going into how to understand the data, how to import it into Python, and how to begin looking into some basic insights from the data.

Python Packages For Data Science

In order to do data analysis in Python, we should first tell you a little bit about the main packages relevant to analysis in Python.

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.

And there are extensive libraries, offering a broad range of facilities.

We have divided the Python data analysis libraries into three groups:

The first group is called "**scientific computing libraries**."

Pandas offers data structure and tools for effective data manipulation and analysis. It provides fast axis to structured data. The primary instrument of Pandas is a two-dimensional table consisting of column and row labels, which are called a DataFrame. It is designed to provide easy indexing functionality.

The **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. Using data visualization methods is the best way to communicate with others, showing them meaningful results of analysis.
These libraries enable you to create graphs, charts and maps.

"**Data Visualization**"

The **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.

Another high-level visualization library is **Seaborn**. It is based on Matplotlib. It's very easy to generate various plots such as heat maps, time series, and violin
plots.

"**Machine Learning**"

With Machine Learning algorithms, we're able to develop a model using our dataset, and obtain predictions. The algorithmic libraries tackle some machine learning tasks from basic to complex.

Here we introduce two packages:

The **Scikit-learn** library contains tools for statistical modeling, including regression, classification, clustering and so on. This library is built on NumPy, SciPy and Matplotlib.

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

In this video, we'll look at how to read in data using Python's pandas package.

Once we have our data in Python, then we can perform all the subsequent data analysis procedures we need.

Data acquisition is 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.

We can usually tell different encoding schemes by looking at the ending of the file name. Some common encodings are csv, json, xlsx, hdf and so forth.

The (file) path tells us where the data is stored.

Usually it is stored either on the computer we are using, or online on the internet.

In our case, we found a dataset of used cars, which was obtained from the web address shown on the slide.

When Jerry entered the web address in his web browser, he saw something like this.

Each row is one data point. A large number of properties are associated with each data point.

Because the properties are separated from each other by commas, we can guess the data format is csv, which stands for comma separated values.

At this point, these are just numbers and don't mean much to humans, but once we read in this data, we can try to make more sense out of it.

In pandas, the "**read_csv()**" method can read in files with columns separated by commas into a pandas DataFrame.

Reading data in pandas can be done quickly in three lines.

First, import pandas.

Then define a variable with the file path.

And then use the read_csv method to import the data.

However, "read_csv" assumes that the data contains a header.

Our data on used cars has no column headers, so we need to specify "read_csv" to not assign headers by setting header to "none".

After reading the dataset, it is a good idea to look at the dataframe to get a better intuition and to ensure that everything occurred the way you expected.

Since printing the entire dataset may take up too much time and resources, to save time, we can just use **dataframe.head()** to show the first n rows of the data frame.

Similarly, **dataframe.tail(n)** shows the bottom n rows of data frame.

Here, we printed out the first 5 rows of data.

```
import pandas as pd
url = "https://archive.ics.uci.edu/ml/rnachine-learningdatabases/autos/imports-85.data"
df = pd.read_csv(url)
```

*Importing a CSV without a header

```
import pandas as pd
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.data"
df = pd.read_csv(url, header - None)
```

*Printing The dataframe in Python

- df prints the entire dataframe (not recommended for large datasets)
- df. head (n) to show the first n rows of data frame.
- df. tail (n) shows the bottom n rows of data frame.

```
df.head()
```

Header

	0	1	2	3	4	5	6	7	8	9	...	16	17	18	19	20	21	22	23	24	25
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21	27	13495
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21	27	16500
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	9.0	154	5000	19	26	16500
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	10.0	102	5500	24	30	13950
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	8.0	115	5500	18	22	17450

It seems that the dataset was read successfully!

We can see that pandas automatically set the column header as a list of integers, because we set header=None when we read the data.

It is difficult to work with the dataframe without having meaningful column names, however, we can assign column names in pandas.

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

```
df.columns=headers
```

```
df.head(5)
```

In our present case, it turned out the we have the column names in a separate file online.

We first put the Column names in a list called headers.

Then, we set df.columns equals headers to replace the default integer headers by the list.

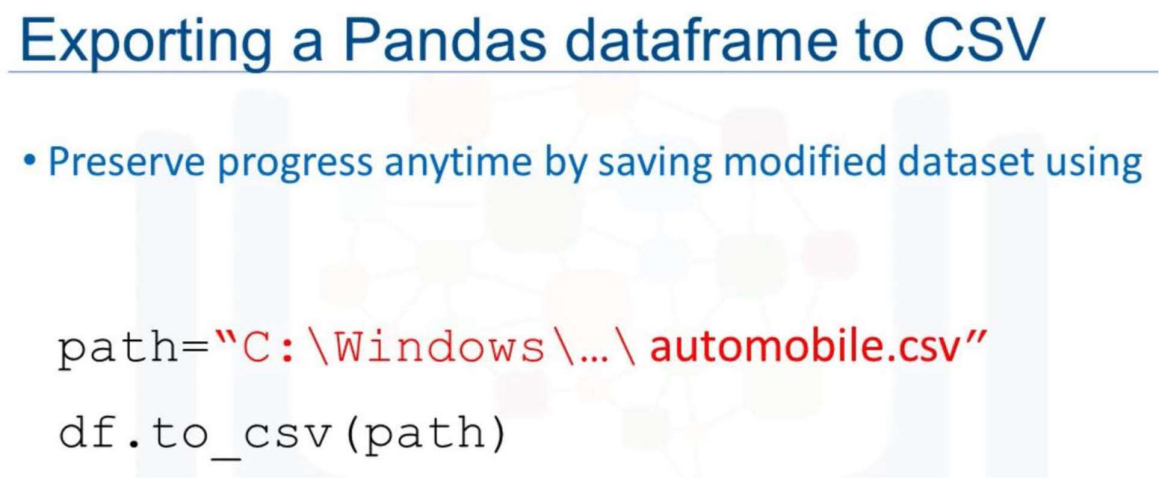
If we use the head() method introduced in the last slide to check the dataset, we see the correct headers inserted at the top of each column.

At some point in time after you've done operations on your dataframe, you may want to export your pandas dataframe to a new CSV file.

You can do this using the method, "to_csv()"

To do this, specify the file path (which includes the filename) that you want to write to.

For example, if you would like to save the dataframe "df" as "automobile.csv" to your own computer, you can use the syntax: df.to_csv ("automobile.csv")



For this course, we will only read and save csv files.

However, pandas also supports importing and exporting of most data filetypes with different dataset formats.

The code syntax for reading and saving other data formats is very similar to read or save csv file.

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()

Each column shows a different method to read and save files into a different format.

Getting Started Analyzing Data In Python

In this video, we introduce some simple pandas methods that all data scientists and analysts should know when using Python pandas and data.

- Understand your data before you begin any analysis
- Should check:
- Data Types
- Data Distribution
- Locate potential issues with the data

Why check data types :

- potential info and type mismatch
- compatibility with python methods

Pandas Type	Native Python Type	Description
object	string	numbers and strings
int64	int	Numeric characters
float64	float	Numeric characters with decimals
datetime64, timedelta[ns]	N/A (but see the datetime module in Python's standard library)	time data.

At this point, we assume that the data has been loaded. It's time for us to explore the dataset.

Pandas has several built in methods that could be used to understand the datatype of features or to look at the distribution of data within the dataset.
Using these methods gives an overview of the dataset. And also point out potential issues, such as the wrong datatype 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.
The datatype names are somewhat different from those in native Python.
This table shows the differences and similarities between them.
Some are very similar, such as the numeric datatypes "int" and "float".

The "object" pandas type functions similar to "string" in Python, save for the change in name, while the "datetime" pandas type, is a very useful type for handling time series data.

There are two reasons to check data types in a dataset.

Pandas automatically assigns types based on the encoding it detects from the original data table. For a number of reasons, this assignment may be incorrect.

For example, it should be awkward if the "car price" column, which we should expect to contain continuous numeric numbers, is assigned the datatype of "object". It would be more natural for it to have the float type.

Jerry may need to manually change the datatype to float.

The second reason is that it allows an experienced data scientist to see 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.

- In pandas, we use `dataframe.dtypes` to check data types

```
df.dtypes
```

```
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
```

When the "dtype" method is applied to the data set, the datatype of each column is returned in a Series.

A good data scientist's intuition tells us that most of the data types make sense.
The make of cars, for example, are names, so this information should be of type object.

The last one on the list could be an issue. As bore is a dimension of an engine, we should expect a numerical data type to be used. Instead, the object type is used.

In later sections, Jerry will have to correct these type mismatches.

Now we would like to check the statistical summary of each column to learn about the distribution of data in each column.

The statistical metrics can tell the data scientist if there are mathematical issues that may exist, such as extreme outliers and large deviations. The data scientist may have to address these issues later.

To get the quick statistics, we use the **describe** method.

It returns the number of terms in the column as "count", average column value as "mean", column standard deviation as "std", the maximum and minimum values, as well as the boundary of each of the quartiles.

By default, the dataframe.describe() function skips rows and columns that do not contain numbers.

It is possible to make the describe method work for object-type columns as well. To enable a summary of all the columns, we could add an argument **include = "all"** inside the describe function bracket.

- Provides full summary statistics

```
df.describe(include="all")
```

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke
count	205.000000	205	205	205	205	205	205	205	205	205.000000	...	205.000000	205	205	205
unique	NaN	52	22	2	2	3	5	3	2	NaN	...	NaN	8	39	37
top	NaN	?	toyota	gas	std	four	sedan	fwd	front	NaN	...	NaN	mpfi	3.62	3.40
freq	NaN	41	32	185	168	114	96	120	202	NaN	...	NaN	94	23	20
mean	0.834146	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	98.756585	...	126.907317	NaN	NaN	NaN
std	1.245307	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	6.021776	...	41.642693	NaN	NaN	NaN
min	-2.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	86.600000	...	61.000000	NaN	NaN	NaN
25%	0.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	94.500000	...	97.000000	NaN	NaN	NaN
50%	1.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	97.000000	...	120.000000	NaN	NaN	NaN
75%	2.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	102.400000	...	141.000000	NaN	NaN	NaN
max	3.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	120.900000	...	326.000000	NaN	NaN	NaN

Now the outcome shows the summary of all the 26 columns, including object-typed attributes.

We see that for object-type columns, a different set of statistics is evaluated, like unique, top and frequency.

"Unique" is the number of distinct objects in the column, "top" is the most frequently occurring object, and "freq" is the number of times the top object appears in the column.

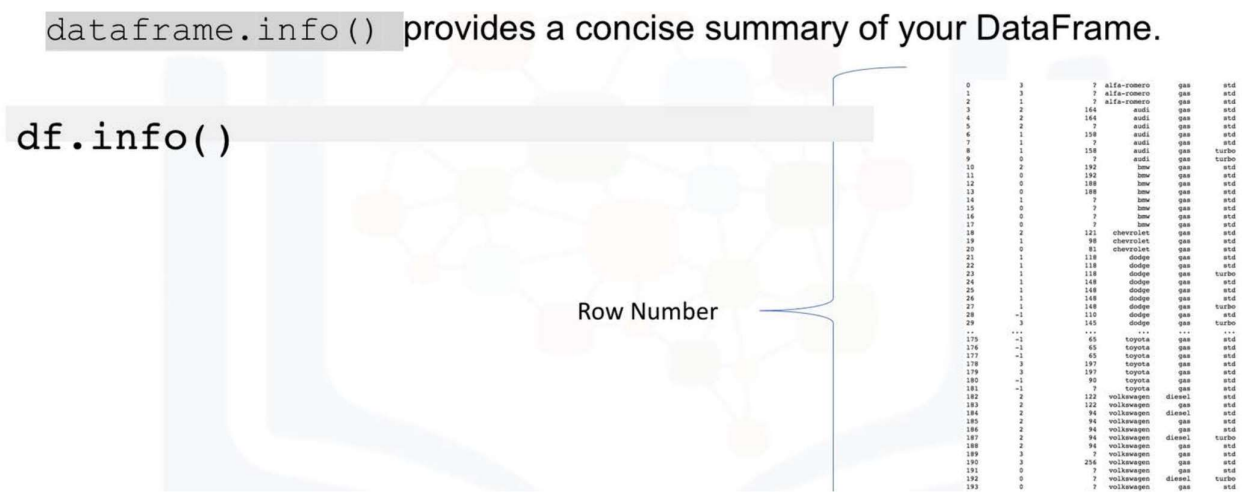
Some values in the table are shown here as "NaN", which stands for "not a number".

This is because that particular statistical metric cannot be calculated for that specific column data type.

Another method you can use to check your dataset is the dataframe.info function.

This function shows the top 30 rows and bottom 30 rows of the dataframe.

df.info() – concise summary of dataframe



Module 2 : Data Wrangling

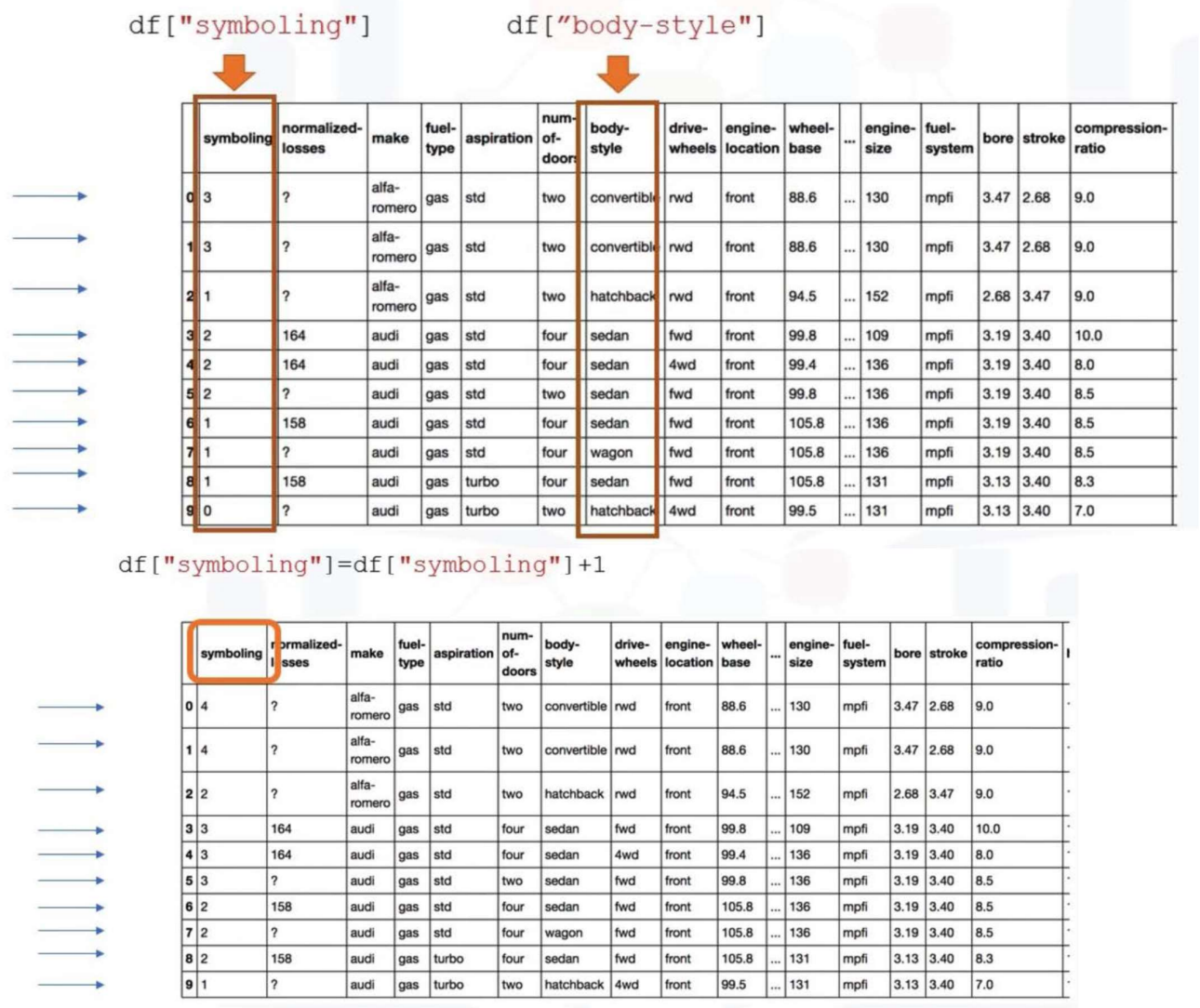
Pre-Processing Data In Python

In this video, we'll be going through some data pre-processing techniques.

If you're unfamiliar with the term, data pre-processing is 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.

Data pre-processing is also often called "data cleaning" or "data wrangling", and there are likely other terms.

Simple Dataframe Operations



	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke	compression-ratio
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	9.0
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	10.0
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	8.0
5	2	?	audi	gas	std	two	sedan	fwd	front	99.8	...	136	mpfi	3.19	3.40	8.5
6	1	158	audi	gas	std	four	sedan	fwd	front	105.8	...	136	mpfi	3.19	3.40	8.5
7	1	?	audi	gas	std	four	wagon	fwd	front	105.8	...	136	mpfi	3.19	3.40	8.5
8	1	158	audi	gas	turbo	four	sedan	fwd	front	105.8	...	131	mpfi	3.13	3.40	8.3
9	0	?	audi	gas	turbo	two	hatchback	4wd	front	99.5	...	131	mpfi	3.13	3.40	7.0

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke	compression-ratio
0	4	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0
1	4	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0
2	2	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	9.0
3	3	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	10.0
4	3	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	8.0
5	3	?	audi	gas	std	two	sedan	fwd	front	99.8	...	136	mpfi	3.19	3.40	8.5
6	2	158	audi	gas	std	four	sedan	fwd	front	105.8	...	136	mpfi	3.19	3.40	8.5
7	2	?	audi	gas	std	four	wagon	fwd	front	105.8	...	136	mpfi	3.19	3.40	8.5
8	2	158	audi	gas	turbo	four	sedan	fwd	front	105.8	...	131	mpfi	3.13	3.40	8.3
9	1	?	audi	gas	turbo	two	hatchback	4wd	front	99.5	...	131	mpfi	3.13	3.40	7.0

Here are the topics that we'll be covering in this module:

First, we'll show you how to identify and handle **missing values**. A "missing value" condition occurs whenever a data entry is left empty.

Then, we'll cover **data formats**. Data from different sources may be in various formats, in different units or in various conventions.

We will introduce some methods in Python pandas that can standardize the values into the same format, or unit, or convention.

After that, we'll cover **data normalization**.

Different columns of numerical data may have very different ranges, and direct comparison is often not meaningful. Normalization is a way to bring all data into a similar range, for more useful comparison. Specifically, we'll focus on the techniques of centering and scaling.

And then, we'll introduce **data binning**.

Binning creates bigger categories from a set of numerical values. It is particularly useful for comparison between groups of data.

And lastly, we'll talk about **categorical variables** and show you how to convert categorical values into numeric variables to make statistical modeling easier.

In Python, we usually perform operations along columns; each row of the column represents a sample, i.e., a different used car in the database.

You access a column by specifying the name of the column.

For example, you can access "symbolling" and "body-style"; each of these columns is a pandas series.

There are many ways to manipulate dataframes in Python.

For example, you can add a value to each entry of a column.
To add 1 to each "symbolling" entry, use this command.
This changes each value of the dataframe column by adding 1 to the current value.

Dealing With Missing Values In Python

Missing Values

What is missing value?

- Missing values occur when no data value is stored for a variable (feature) in an observation.
- Could be represented as"?", "N/A", 0 or just a blank cell.

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location
0	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front

How to deal with missing data ?

Check with the data collection source
Drop the missing values

- drop the variable
- drop the data entry

Replace the missing values

- replace it with an average (of similar datapoints)
- replace it by frequency
- replace it based on other functions

Leave it as missing data

How to drop missing values - dataframes.dropna()

- Use `dataframes.dropna()`:

highway-mpg	price
...	...
20	23875
22	NaN
29	16430
...	...

→

highway-mpg	price
...	...
20	23875
29	16430
...	...

axis=0 drops the entire row
axis=1 drops the entire column

```
df.dropna(subset=["price"], axis=0, inplace = True)
df = df.dropna(subset=["price"], axis=0)
```

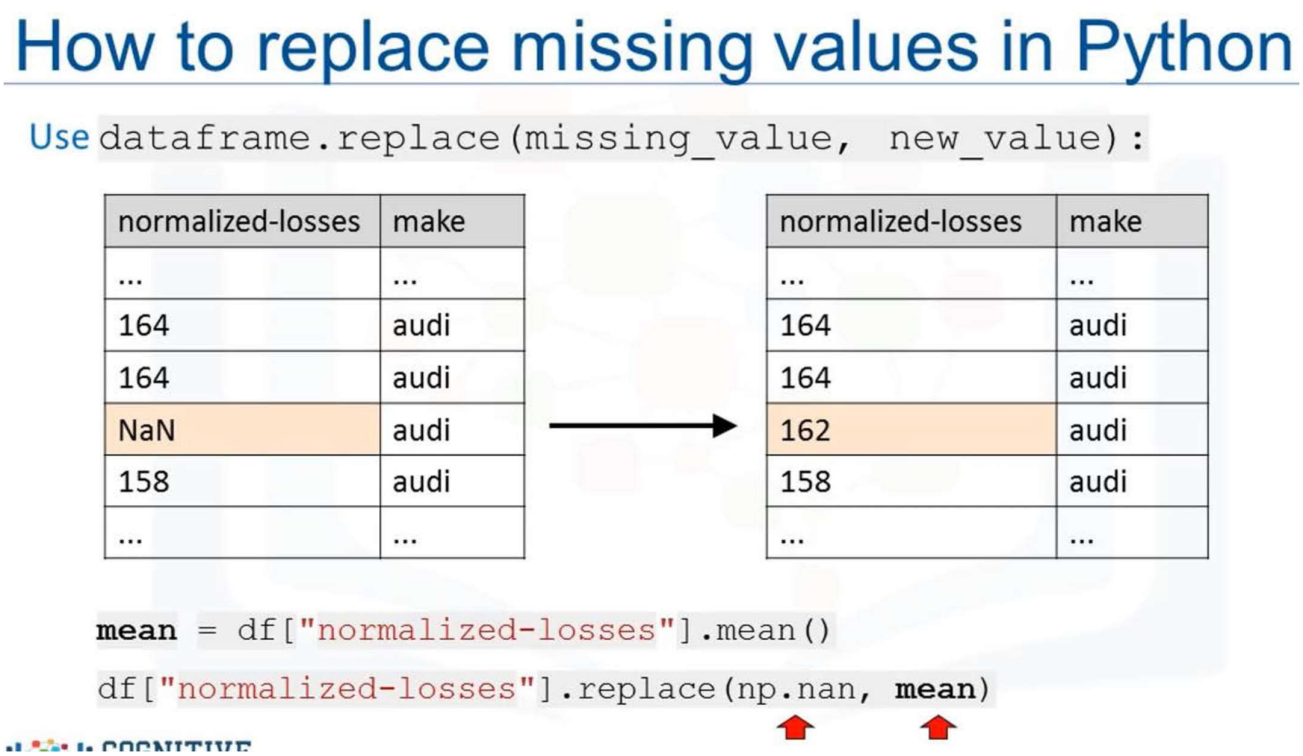
Don't Forget

```
df.dropna(subset=["price"], axis=0)
```

```
df.dropna(subset=["price"], axis=0, inplace = True)
```

http://pandas.pydata.org/

dataframes.replace()



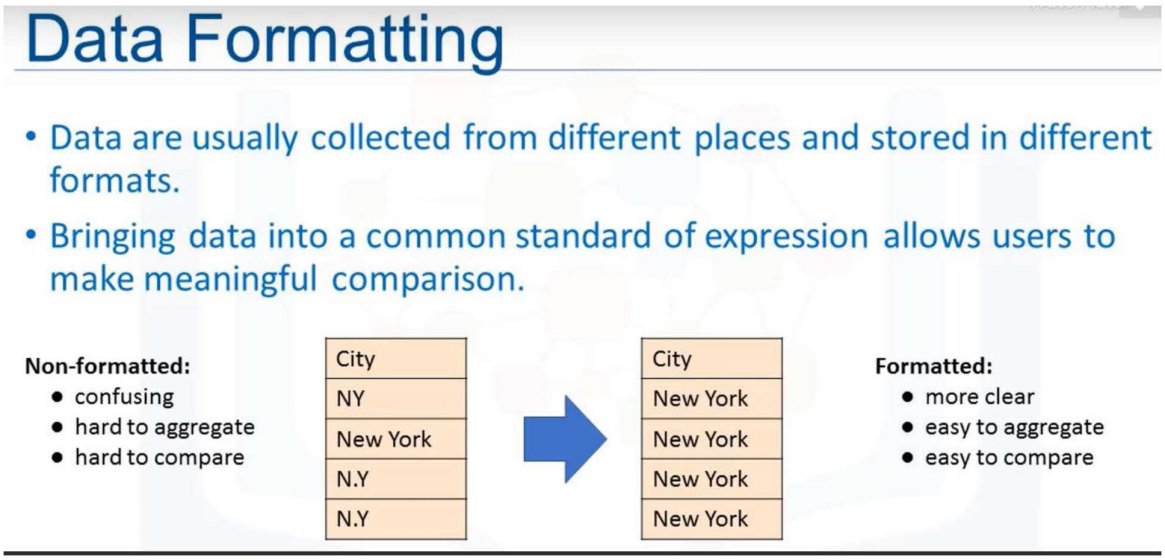
Data Formatting In Python

In this video, we'll look at the problem of data with different formats, units, and conventions, and the pandas methods that help us deal with these issues.

Data is usually collected from different places, by different people, which may be stored in different formats.

Data formatting 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 that data is consistent and easily understandable.



For example, people may use different expressions to represent New York City, such as N.Y., Ny, NY, and New York.

Sometimes, this “uncleaned” data is a good thing to see.

For example, if you’re looking at the different ways people tend to write “New York”, then this is exactly the data that you want.

Or if you’re looking for ways to spot fraud, perhaps writing “N dot Y dot” is more likely to predict an anomaly than if someone wrote out “New York” in full.

But perhaps, more often than not, we just simply want to treat them all as the same entity, or format, to make statistical analyses easier down the road.

Referring to our used car dataset, there’s a feature named “city-mpg” in the dataset, which refers to a car fuel consumption in miles per gallon unit.

However, you may be someone who lives in a country that uses metric units.

So you would want to convert those values to L/100km --the metric version.

To transform mpg to L/100km we need to divide 235 by each value in the city-mpg column.

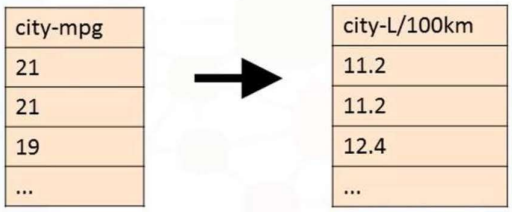
In Python, this can easily be done in one line of code.

You take the column and set it equal to 235 divided by the entire column.
In the second line of code, rename column name from "city-mpg" to "city-L/100km" using the dataframe.rename() method.

```
df["city-mpg"]= 235/df["city-mpg"]
df.rename(columns= {"city_mpg": "city-L/lOOkm"}, inplace=True)
```

Applying calculations to an entire column

- Convert "mpg" to "L/100km" in Car dataset.




```
df["city-mpg"]= 235/df["city-mpg"]

df.rename(columns={"city_mpg": "city-L/100km"}, inplace=True)
```

For a number of reasons, including when you import a dataset into Python, the data type may be incorrectly established.

For example, here we notice that the assigned data type to the price feature is "object" although the expected data type should really be an integer or float type.

Incorrect data types

- Sometimes the wrong data type is assigned to a feature.

```
df["price"].tail(5)

200    16845
201    19045
202    21485
203    22470
204    22625
Name: price, dtype: object
```

It is important for later analysis to explore the feature's data type and convert them to the correct data types; otherwise, the developed models later on may behave strangely, and totally valid data may end up being treated like missing data.

There are many data types in pandas.

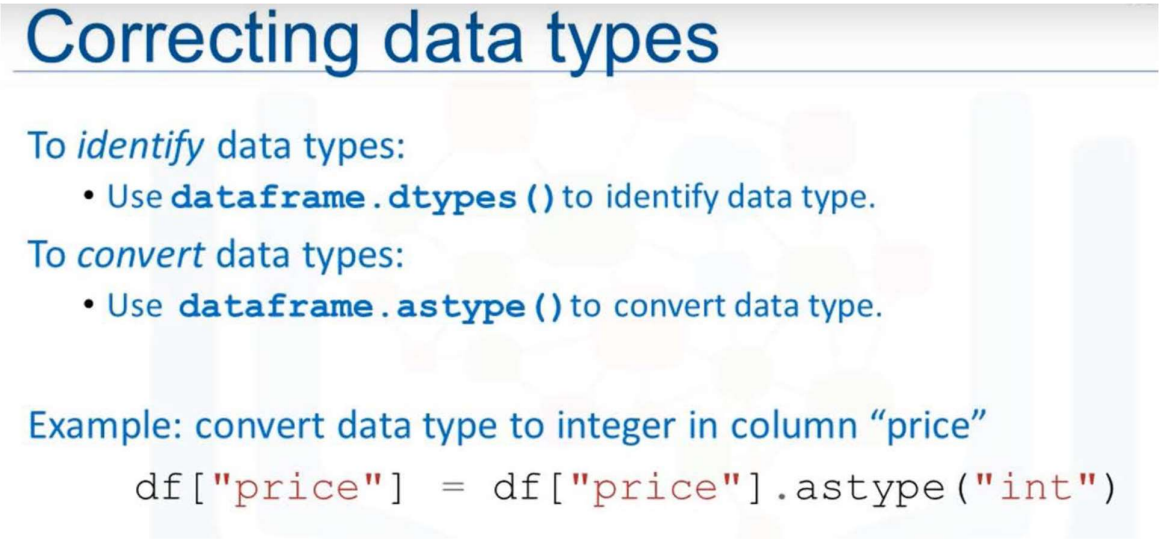
Objects can be letters or words.
Int64 are integers.
And Floats are real numbers.
There are many others that we will not discuss.

To identify a features data type, in Python we can use the dataframe.dtypes() method and check the datatype of each variable in a dataframe.

In the case of wrong datatypes, the method dataframe.astype() can be used to convert a datatype from one format to another.

For example, using astype("int") for the price column, you can convert the object column into an integer type variable.

df ["price"] = df ["price"] . as type ("int")



Data Normalization In Python

In this video, we'll be talking about data normalization, an important technique to understand in data pre-processing.

When we take a look at the used car data set, we notice in the data that the feature "length" ranges from 150 to 250, while feature "width" and "height" ranges from 50 to 100.

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 fairer comparison between the different features.

Making sure they have the same impact, it is also important for computational reasons.

Data Normalization

- Uniform the features value with different range.

length	width	height
168.8	64.1	48.8
168.8	64.1	48.8
171.2	65.5	52.4
176.6	66.2	54.3
176.6	66.4	54.3
177.3	66.3	53.1
192.7	71.4	55.7
192.7	71.4	55.7
192.7	71.4	55.9

scale	[150,250]	[50,100]	[50,100]
impact	large	small	small

Here is another example that will help you understand why normalization is important.

Consider a dataset containing two features: “age” and “income”, where “age” ranges from 0 to 100, while “income” ranges from 0 to 20,000 and higher. “income” is about 1,000 times larger than “age”, and ranges from 20,000 to 500,000. So these two features are in very different ranges.

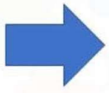
When we do further analysis, like linear regression, for example, the attribute “income” will intrinsically influence the result more, due to its larger value, but this doesn’t necessarily mean it is more ‘important’ as a predictor.

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 0 to 1.

Data Normalization

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

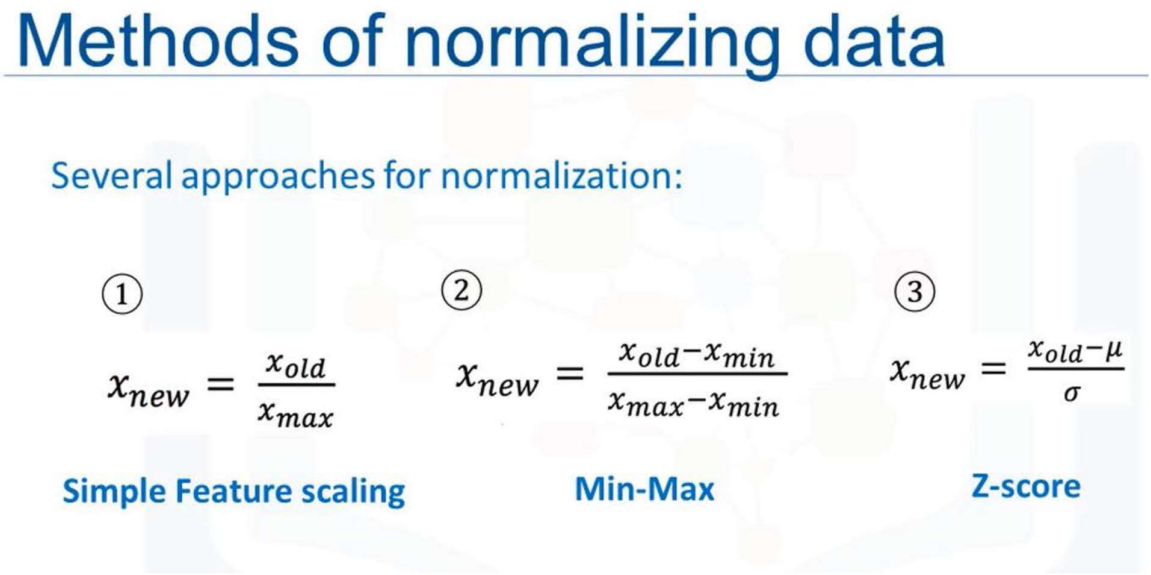
Normalized

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

Compare the two tables at the right.

After normalization, both variables now have a similar influence on the models we will build later.

There are several ways to normalize data.



I will just outline three techniques.

The first method, called "**simple feature scaling**", just divides each value by the maximum value for that feature.

This makes the new values range between 0 and 1.

The second method, called "**Min-Max**", takes each value, X_old, subtracted from the minimum value of that feature, then divides by the range of that feature.

Again, the resulting new values range between 0 and 1.

The third method is called "**z-score**" or "standard score".

In this formula, for each value, you subtract the Mu which is the average of the feature, and then divide by the standard deviation (sigma).

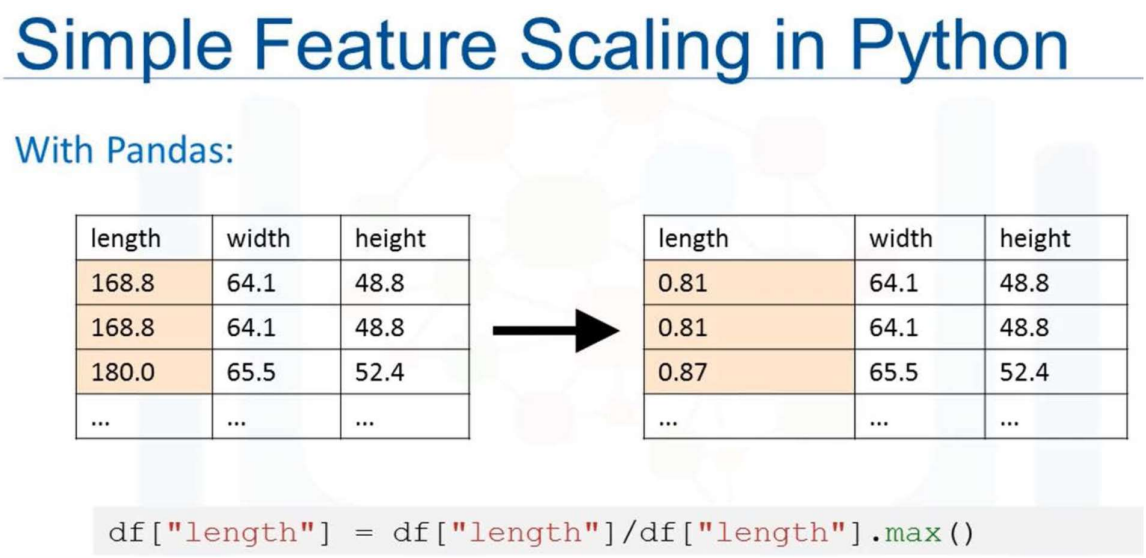
The resulting values hover around 0, and typically range between -3 and +3, but can be higher or lower.

Following our earlier example, we can apply the normalization method on the "length" feature.

First, we use the simple feature scaling method, where we divide it by the maximum value in the feature.

Using the pandas method “max”, this can be done in just one line of code.

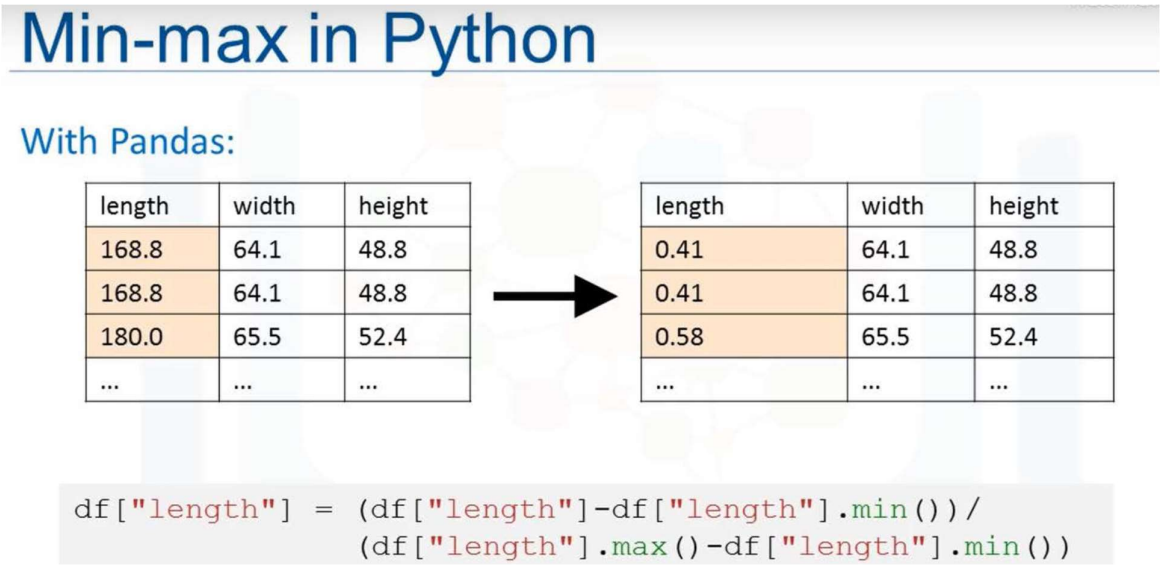
df ["length"] - df ["length"] / df ["length"] . max ()



Here’s the Min-max method on the “length” feature.

We subtract each value by the minimum of that column, then divide it by the range of that column: the max minus the min.

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

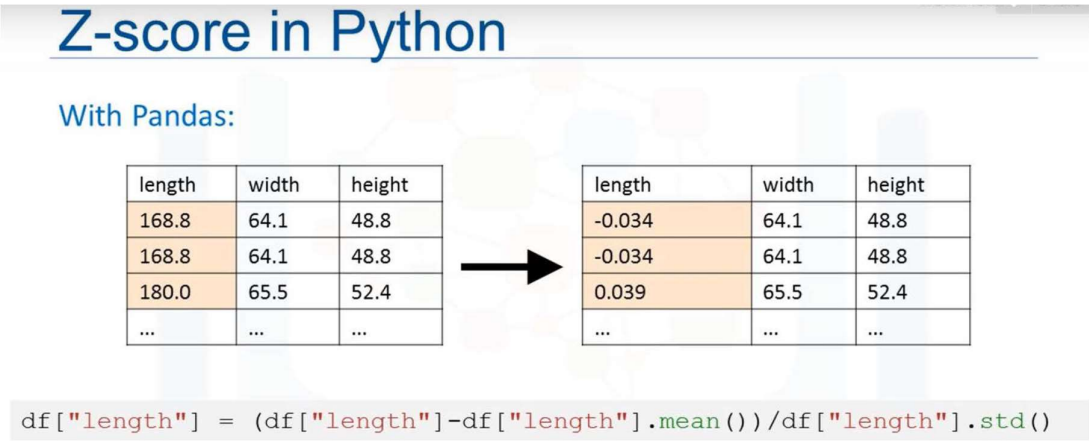


Finally we apply the Z-score method on length feature to normalize the values.

Here, we apply the mean() and std() method on the length feature.

mean() method will return the average value of the feature in the dataset, and std() method will return the standard deviation of the features in the dataset.

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



Binning In Python

In this video, we'll talk about binning as a method of data pre-processing.

Binning is when you group values together into bins. For example, you can bin "age" into [0 to 5], [6 to 10], [11 to 15] and so on.

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.

As example, "price" here is an attribute range from 5,000 to 45,500.

Using binning, we categorize the price into three bins: low price, medium price, and high prices.

Binning

- Binning: Grouping of values into ”bins”
- Converts numeric into categorical variables
- Group a set of numerical values into a set of “bins”
- “price” is a feature range from 5,000 to 45,500
  (in order to have a **better representation** of price)

price: 5000, 10000,12000,12000, 30000, 31000, 39000, 44000,44500

bins: low | Mid | High

In the actual car dataset, ”price" is a numerical variable ranging from 5188 to 45400, it has 201 unique values.

We can categorize them into 3 bins: low, medium, and high-priced cars.

In Python we can easily implement the binning: We would like 3 bins of equal binwidth, so we need 4 numbers as dividers that are equal distance apart.

Binning in Python pandas

price
13495
16500
18920
41315
5151
6295
...

→

price	price-binned
13495	Low
16500	Low
18920	Medium
41315	High
5151	Low
6295	Low
...	...

```
bins = np.linspace(min(df["price"]), max(df["price"]), 4)
group_names = [“Low”, “Medium”, “High”]
df[“price-binned”] = pd.cut(df[“price”], bins, labels=group_names, include_lowest=True )
```

bins = np.linspace(min(df["price"]), max(df["price"]), 4)

group_names = ["Low", "Medium", "High"]

df["price-binned"] = pd.cut(df["price"], bins, labels= group_names, include_lowest=True)

First we use the numpy function “linspace” to return the array “bins” that contains 4 equally spaced numbers over the specified interval of the price.

We create a list “group_names “ that contains the different bin names.

We use the pandas function ”cut” to segment and sort the data values into bins.

You can then use histograms to visualize the distribution of the data after they’ve been divided into bins.

This is the histogram that we plotted based on the binning that we applied in the price feature.

From the plot, it is clear that most cars have a low price, and only very few cars have high price.

Turning Categorical Into Quantitative Variables

Categorical → Numeric

Solution:

- Add dummy variables for each unique category
- Assign 0 or 1 in each category

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

"One-hot encoding"

Dummy variables in Python pandas

- Use pandas.get_dummies() method.
- Convert categorical variables to dummy variables (0 or 1)

fuel
gas
diesel
gas
gas

→

gas	diesel
1	0
0	1
1	0
1	0

```
pd.get_dummies(df['fuel'])
```

pd.get_dummies(df['fuel'])

Module 3 : Exploratory Data Analytics

In this module we're going to cover the basics of Exploratory Data Analysis using Python.

Exploratory Data Analysis, or in short "EDA", is an approach to analyze data in order to:
- summarize main characteristics of the data - gain better understanding of the dataset,
- uncover relationships between different variables, and
- extract important variables for the problem we are trying to solve.

The main question we are trying to answer in this module is:

"What are the characteristics that have the most impact on the car price?"

We will be going through a couple of different useful exploratory data analysis techniques in order to answer this question.

In this module you will learn about:

Descriptive Statistics, which describe basic features of a dataset and obtains a short summary about the sample and measures of the data.

Basic of **Grouping Data using group by**, and how this can help to transform our dataset.

ANOVA, the analysis of variance, a statistical method in which the variation in a set of observations is divided into distinct components.

The **Correlation** between different variables.

And lastly, **Advanced Correlation**, where we'll introduce you to various correlation statistical methods, namely **Pearson Correlation and Correlation Heatmaps**.

Descriptive Statistics

In this video, we'll be talking about Descriptive Statistics.

When you begin to analyze data, it's important to first explore your data before you spend time building complicated models. One easy way to do so is to calculate some descriptive statistics for your data.

Descriptive statistical analysis helps to describe basic features of a dataset and obtains a short summary about the sample and measures of the data. Let's show you a couple different useful methods.

One way in which we can do this is by using the **describe()** function in pandas. Using the describe function and applying it on your dataframe, the "describe" function automatically computes basic statistics for all numerical variables. It shows the mean, the total number of data points, the standard deviation, the quartiles and the extreme values. Any NaN values are automatically skipped in these statistics. This function will give you a clearer idea of the distribution of your different variables.

Descriptive Statistics- Describe()

- Summarize statistics using pandas **describe()** method

```
df.describe()
```

	Unnamed: 0	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore	stroke
count	201.000000	201.000000	164.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000
mean	100.000000	0.840796	122.000000	98.797015	174.200995	65.889055	53.766667	2555.666667	126.875622	3.319154	3.256766
std	58.167861	1.254802	35.442168	6.066366	12.322175	2.101471	2.447822	517.296727	41.546834	0.280130	0.316049
min	0.000000	-2.000000	65.000000	86.600000	141.100000	60.300000	47.800000	1488.000000	61.000000	2.540000	2.070000
25%	50.000000	0.000000	NaN	94.500000	166.800000	64.100000	52.000000	2169.000000	98.000000	3.150000	3.110000
50%	100.000000	1.000000	NaN	97.000000	173.200000	65.500000	54.100000	2414.000000	120.000000	3.310000	3.290000
75%	150.000000	2.000000	NaN	102.400000	183.500000	66.600000	55.500000	2926.000000	141.000000	3.580000	3.410000
max	200.000000	3.000000	256.000000	120.900000	208.100000	72.000000	59.800000	4066.000000	326.000000	3.940000	4.170000

You could have also categorical variables in your dataset. These are variables that can be divided up into different categories, or groups and have discrete values.

For example, in our dataset we have the drive system as a categorical variable, which consists of the categories: forward-wheel drive, rear-wheel drive, and four-wheel drive. One way you can summarize the categorical data is by using the function **value_counts()**.

- summarize the categorical data is by using the **value_counts()** method

```
drive_wheels_counts=df["drive-wheels"].value_counts()

drive_wheels_counts.rename(columns={'drive-wheels':'value_counts' inplace=True)
drive_wheels_counts.index.name= 'drive-wheels'
```

	value_counts
drive-wheels	
fwd	118
rwd	75
4wd	8

COGNITIVE CLASS.ai

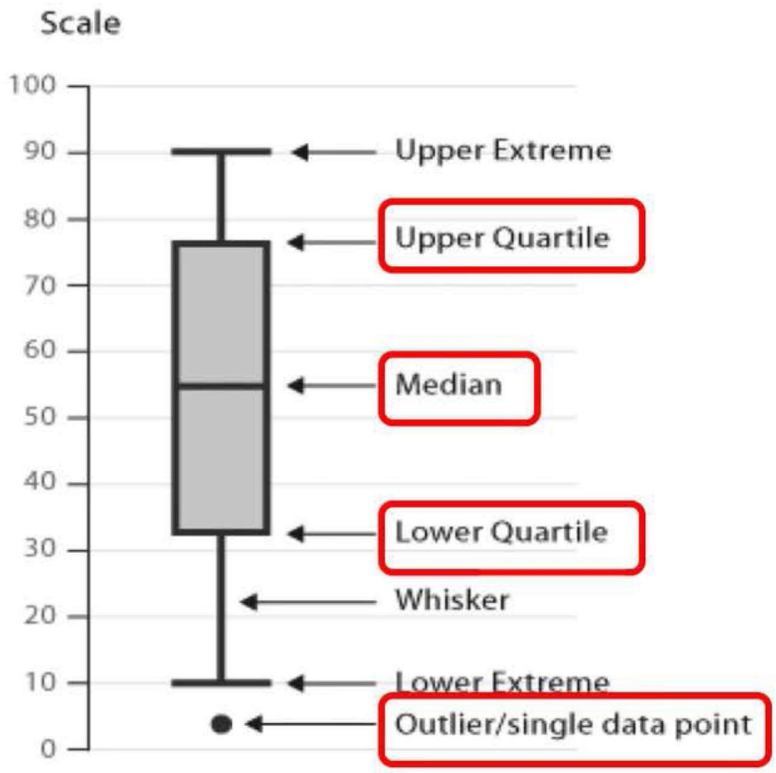
4

We can change the name of the column to make it easier to read. We see that we have 118 cars in the fwd (front wheel drive) category, 75 cars in the rwd (rear wheel drive) category, and 8 cars in the 4wd (four wheel drive) category.

Boxplots are a great way to visualize numeric data, since you can visualize the various distributions of the data. The main features that the boxplot shows are the median of the data, which represents where the middle datapoint is. The Upper Quartile shows where the 75th percentile is, the Lower Quartile shows where the 25th percentile is. The data between the Upper and Lower Quartile represents the Interquartile Range.

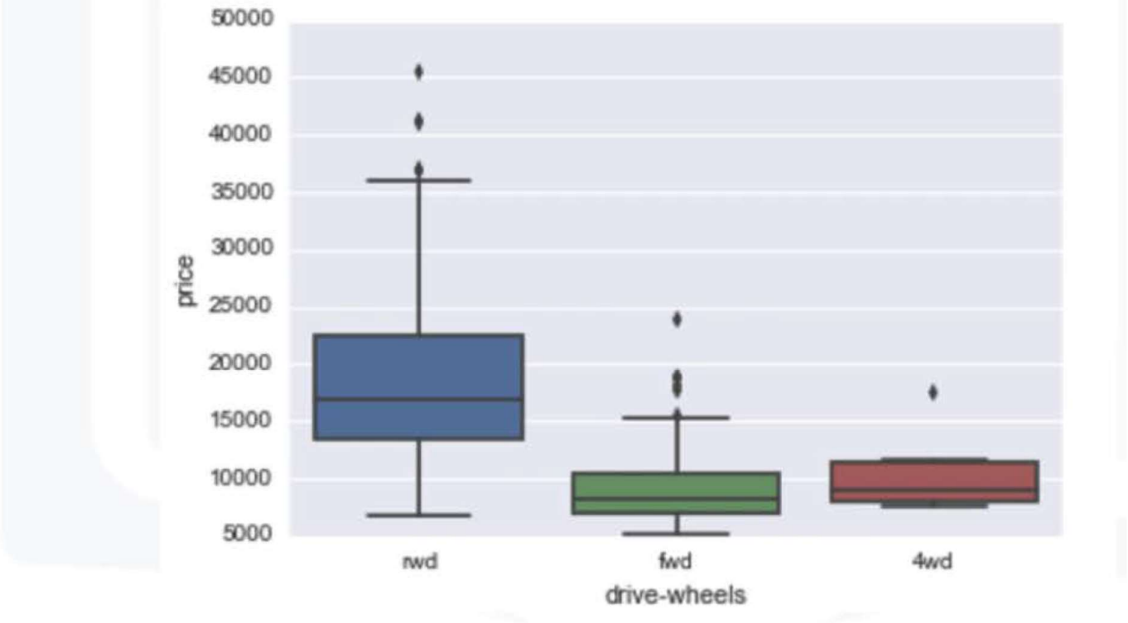
Next, you have the Lower and Upper Extremes. These are calculated as 1.5 times the interquartile range above the 75th percentile, and as 1.5 times the IQR below the 25th percentile.

Finally, boxplots also display outliers as individual dots that occur outside the upper and lower extremes. With boxplots, you can easily spot outliers and also see the distribution and skewness of the data.



Boxplots make it easy to compare between groups. In this example, using Boxplot we can see the distribution of different categories of the “drive-wheels” feature over price feature. We can see that the distribution of price between the rwd (rear wheel drive) and the other categories are distinct, but the price for fwd (front wheel drive) and 4wd (four wheel drive) are almost indistinguishable.

```
sns.boxplot(x= “drive-wheels”, y= “price”,data=df)
```



Often times we tend to see continuous variables in our data. These data points are numbers contained in some range. For example, in our dataset, price and engine size are continuous variables. What if we want to understand the relationship between “engine size” and ”price”? Could engine size possibly predict the price of a car? One good way to visualize this is using a
scatter plot. Each observation in a scatter plot is represented as a point. This plot shows the relationship between two variables:

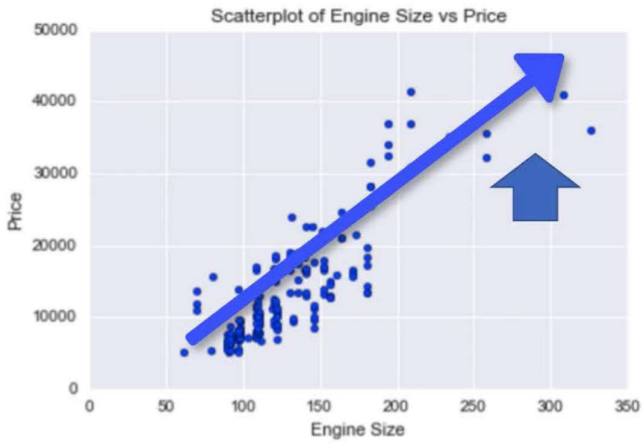
The predictor variable: is the variable that you are using to predict an outcome. In this case, our predictor variable is the engine size.

The target variable: is the variable that you are trying to predict. In this case, our target variable is the price, since this would be the outcome.

In a scatterplot, we typically set the predictor variable on the x-axis, or horizontal axis and we set the target variable on the y-axis or vertical axis.

In this case, we will thus plot the engine size on the x-axis and the price on the y-axis.

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

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



We are using the **Matplotlib function “scatter”** here, taking in x and a y variable.

Something to note is that it’s always important to label your axes and write a general plot title, so that you know what you are looking at.

Now how is the variable Engine Size related to Price? From the scatterplot we see that as the engine size goes up, the price of the car also goes up.

This is giving us an initial indication that there is a positive linear relationship between these two variables.

Group by In Python

In this video, we'll cover the basics of grouping and how this can help to transform our dataset.

Assume you want to know: Is there any relationship between the different types of "drive system" (forward, rear and four-wheel drive) and the "price" of the vehicles? If so, which type of "drive system" adds the most value to a vehicle?

It would be nice if we could group all the data by the different types of drive wheels, and compare the results of these different drive wheels against each other.

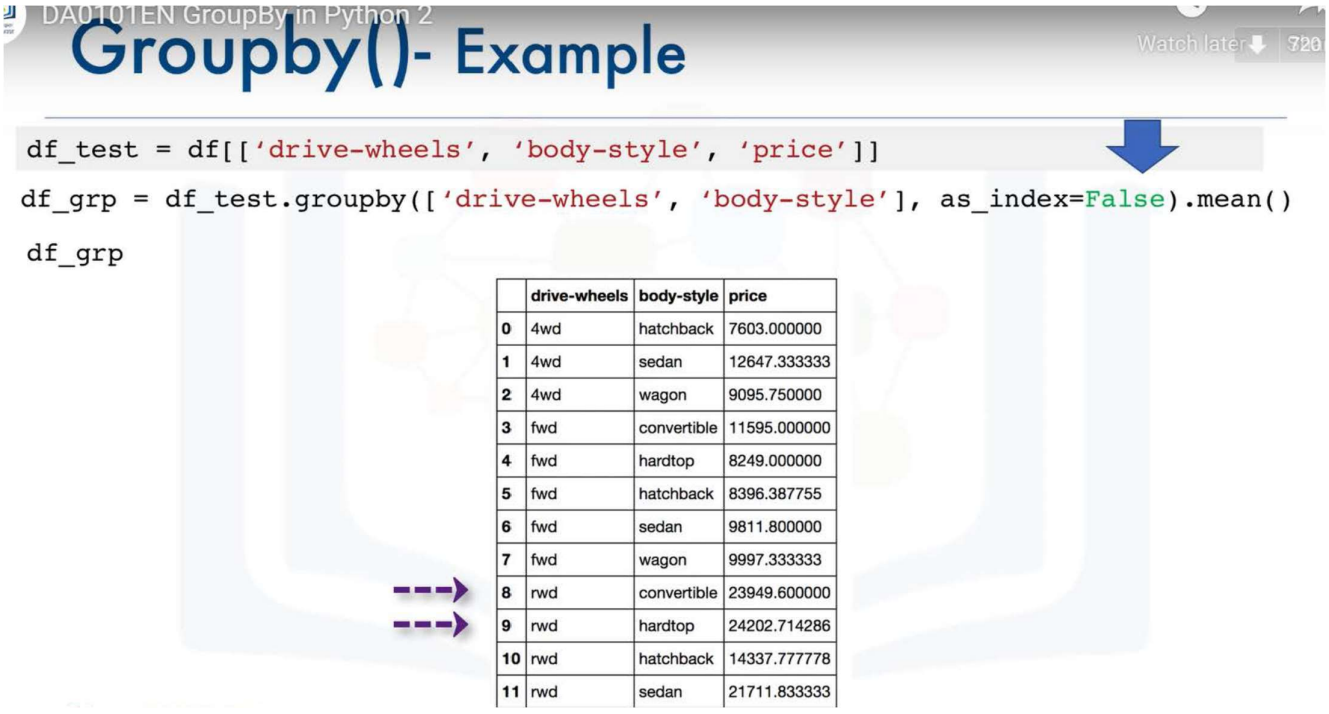
In pandas this can be done using the group by method.
The group by method is used on categorical variables, groups the data into subsets according to the different categories of that variable.

You can group by a single variable or you can group by multiple variables by passing in multiple variable names.

Use Panda dataframe. Groupby() method:

- Can be applied on categorical variables
- Group data into categories
- Single or multiple variables

As an example, let's say we are interested in finding the average price of vehicles and observe how they differ between different types of "body styles" and "drive wheels" variables.



	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

To do this, we first pick out the three data columns we are interested in, which is done in the first line of code.

We then group the reduced data according to ‘drive wheels’ and ‘body style’ in the second line.

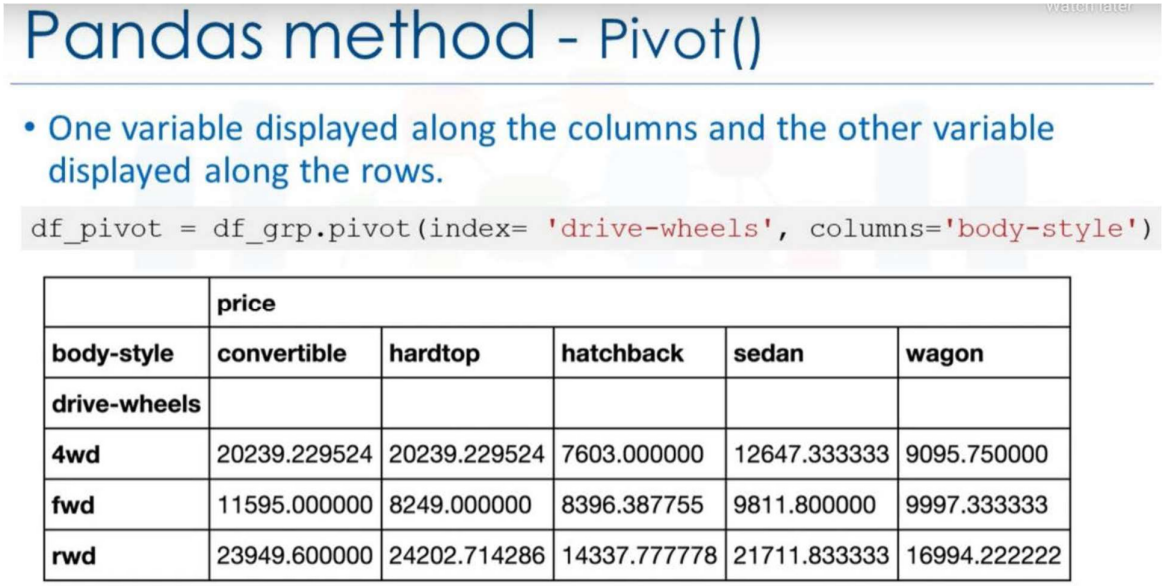
Since we are interested in knowing how the average price differs across the board, we can take the mean of each group and append this bit at the very end of line 2.

The data is now grouped into subcategories and only the average price of each subcategory is shown.

We can see that, according to our data, rear wheel drive convertibles and rear wheel drive hardtops have the highest value, while four wheel drive hatchbacks have the lowest value.

A table of this form isn’t the easiest to read, and also not very easy to visualize.
To make it easier to understand, we can transform this table to a pivot table by using the **pivot method**.

In the previous table, both ‘drive wheels’ and ‘body style’ were listed in columns.
A pivot table has one variable displayed along the columns and the othervariable displayed along the rows



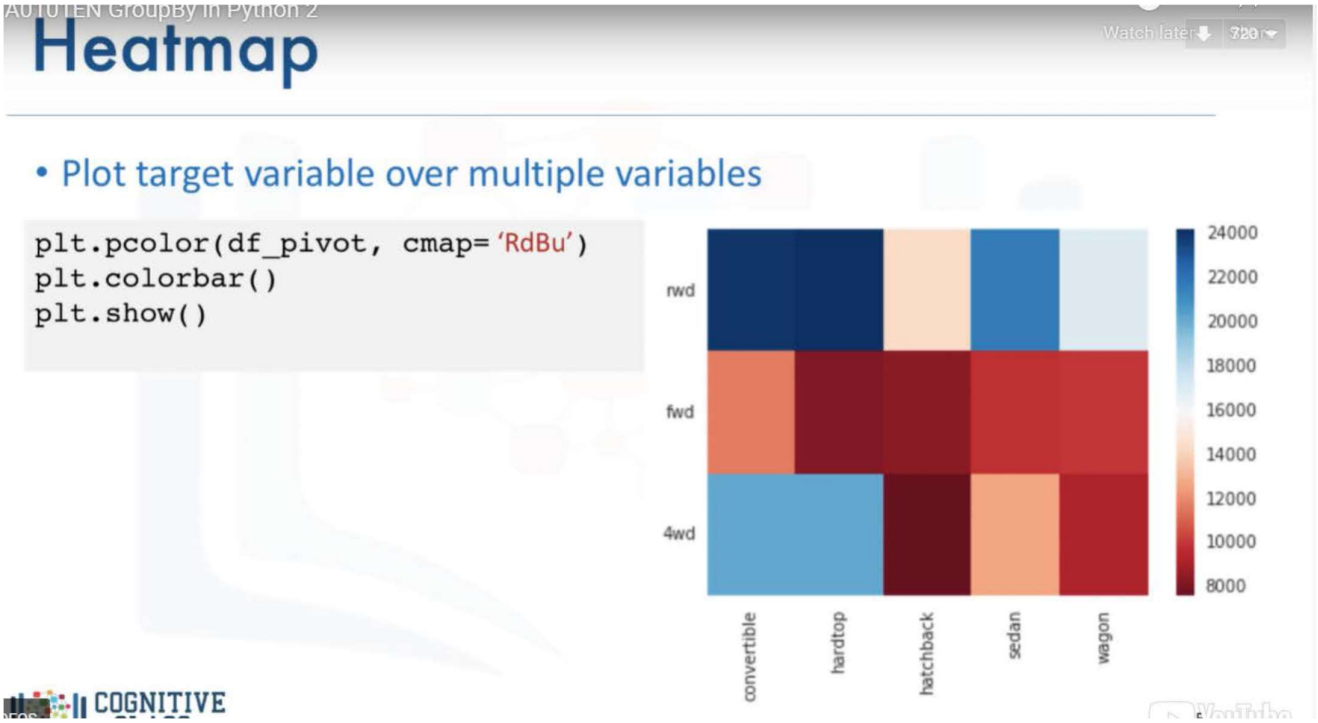
	price				
body-style	**convertible**	**hardtop**	**hatchback**	**sedan**	**wagon**
drive-wheels					
4wd	20239.229524	20239.229524	7603.000000	12647.333333	9095.750000
fwd	11595.000000	8249.000000	8396.387755	9811.800000	9997.333333
rwd	23949.600000	24202.714286	14337.777778	21711.833333	16994.222222

.
Just with one line of code and by using the pandas pivot method, we can pivot the “body style” variable so it is displayed along the columns and the “drive wheels” will be displayed along the rows.

The price data now becomes a rectangular grid, which is easier to visualize.
This is similar to what is usually done in Excel spreadsheets.
Another way to represent the pivot table is using a **heatmap plot**.

Heat map takes a rectangular grid of data and 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 of the relationship between these variables and the target.



In this example, we use pyplot's pcolor method to plot a heat map and convert the previous pivot table into a graphical form.

We specified the Red-blue color scheme.

In the output plot, each type of "body style" is numbered along the x-axis, and each type of "drive wheels" is numbered along the y-axis.

The average prices are plotted with varying colors based on their values, according to the color bar.

We see that the top section of the heat map seems to have higher prices than the bottom section.

Analysis Of Variance (ANOVA)

Assume that we want to analyze a categorical variable and see the correlation among different categories.

For example, consider the car dataset, the question we may ask is, how different categories of the Make feature (as a categorical variable) has impact on the price?

The diagram shows the average price of different vehicle makes. We do see a trend of increasing prices as we move right along the graph.

But which category in the make feature has the most and which one has the least impact on the car price prediction?

To analyze categorical variables such as the "make" variable, we can use a method such as the ANOVA method.

ANOVA is a statistical test that stands for "Analysis of Variance".

ANOVA can be used to find the correlation between different groups of a categorical variable.

According to the car dataset, we can use ANOVA to see if there is any difference in mean price for the different car makes such as Subaru and Honda.

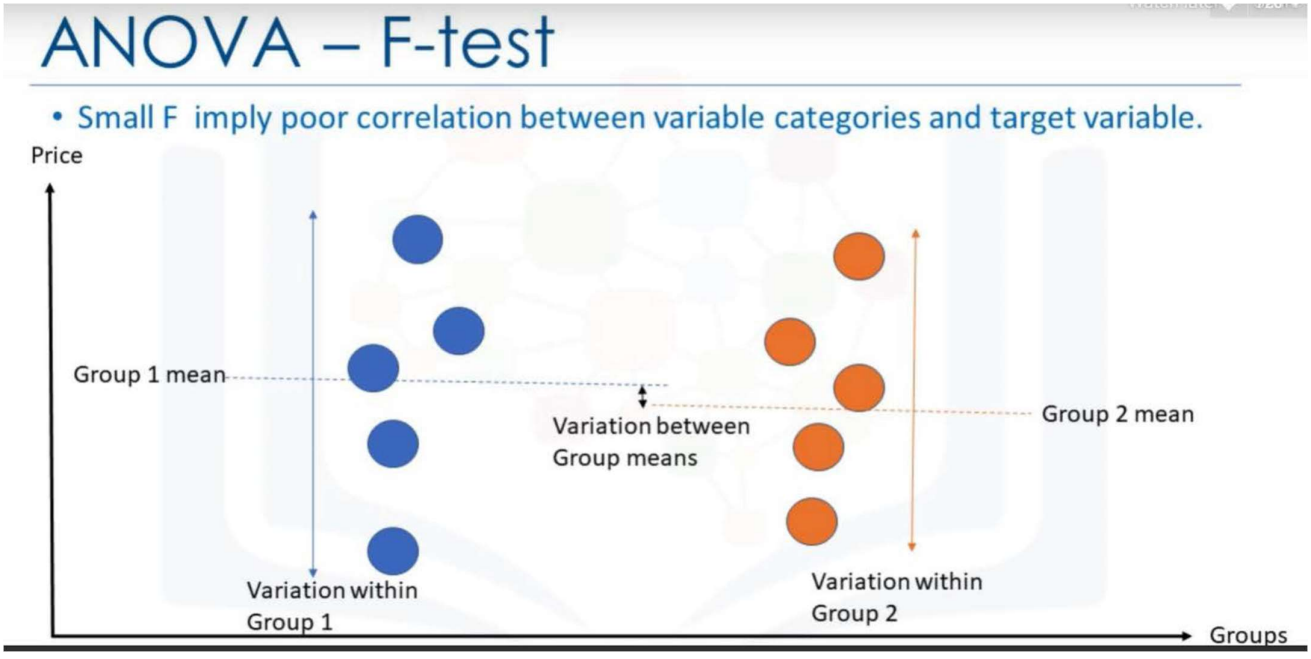
The ANOVA test returns two values: the F-test score and the p-value.

The **F-test** calculates the ratio of variation between the groups's mean over the variation within each of the sample groups.

The **p-value** shows whether the obtained result is statistically significant.

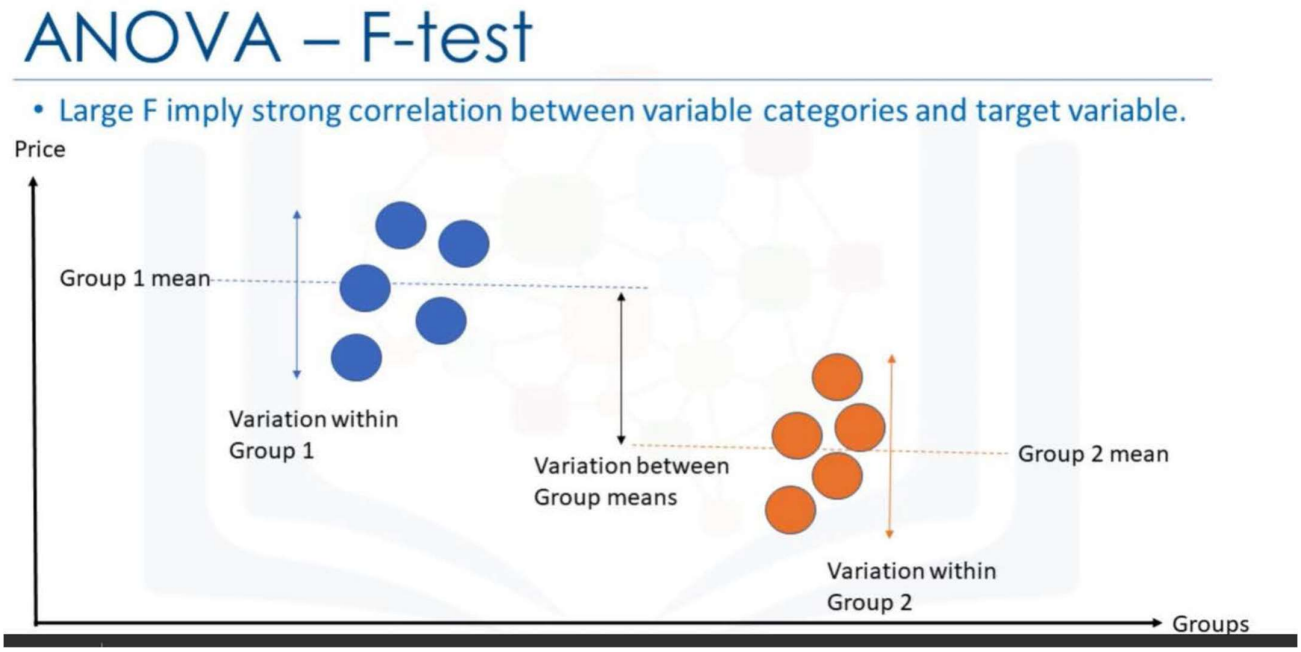
Without going too deep into the details, the F-test calculates the ratio of variation between group means over the variation within each of the sample group means.

This diagram illustrates a case where the F-test score would be small.



Because, as we can see the variation of the prices in each group of data is way larger than the differences between the average values of each group. Looking at this diagram, assume that, group 1 is "Honda" and group 2 is "Subaru"; both are the make feature categories.

Since the F-score is small, the correlation between price as the target variable and the groupings is weak.

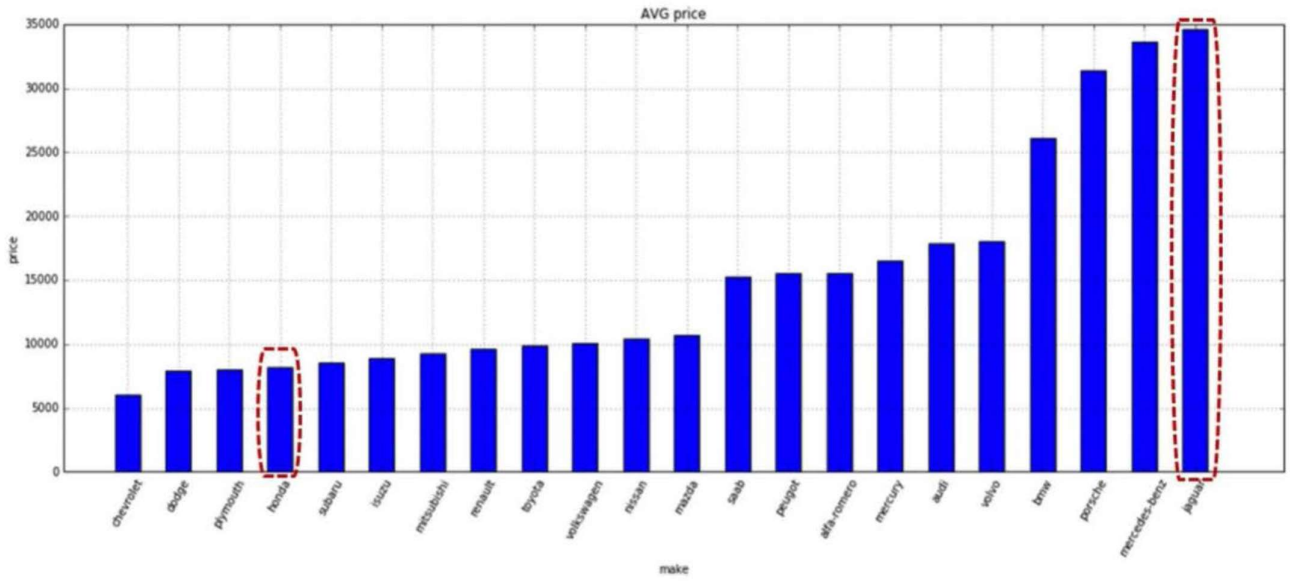


In this second diagram, we see a case where the F-test score would be large.
The variation between the averages of the two groups is comparable to the variations within the two groups.

Assume that group 1 is "Jaguar" and group 2 is "Honda"; both are the Make feature categories.

Since the F-score is large, thus the correlation is strong in this case.

ANOVA



Getting back to our example, the bar chart shows the average price for different categories of the make feature.

As we can see from the bar chart, we expect a small F-score between "Hondas" and "Subarus" because there is a small difference between the average prices.
On the other hand, we can expect a large F-value between Hondas and Jaguars because the differences between the prices is very significant.

However, from this chart we do not know the exact variances, so let's perform an ANOVA test to see if our intuition is correct.

ANOVA

- ANOVA between “Honda” and “Subaru”

```
df_anova=df[["make", "price"]]
grouped_anova=df_anova.groupby(["make"])
```

```
anova_results_1=stats.f_oneway(grouped_anova.get_group("honda")["price"], grouped_anova.get_group("subaru")["price"])
ANOVA results: F=0.19744031275, p=F_onewayResult(statistic=0.1974430127), pvalue=0.660947824)
```

- ANOVA between “Honda” and “Jaguar”

```
anova_results_1=stats.f_oneway(grouped_anova.get_group("honda")["price"], grouped_anova.get_group("jaguar")["price"])
ANOVA results: F=400.92587, p=F_onewayResult(statistic=400.92587), pvalue=1.05861e-11)
```

df _anova= df [["make", "price"]]

grouped -anova=df -anova.groupby(["make»))

anova_results_l=stats. f_oneway (grouped_anova. get_group ("honda") ["price"), grouped_anova. get_group("subaru") ("price"))

ANOVA results: F=0.19744031275, p=F_onewayResult(statistic=0.1974430127), pvalue=0.660947824)

In the first line we extract the make and price data.
Then, we'll group the data by different makes.

The ANOVA test can be performed in Python using the f_oneway method as the built-in function of the **Scipy** package.

We pass in the price data of the two car make groups that we want to compare and it calculates the ANOVA results.
The results confirm what we guessed at first.
The prices between Hondas and Subarus are not significantly different, as the F-test score is less than 1 and p-value is larger than 0.05.

We can do the same for Honda and Jaguar.

anova_results_l=stats. f _oneway (grouped_anova. get_group ("honda") ("price"), grouped_anova. get_group("jaguar") ["price"))
ANOVA results: F=400.92587, p=F_onewayResult(statistic=400.92587), pvalue=l.05861e-11)

The prices between Hondas and Jaguars are significantly different, since the F-score is very large (F = 401) and the p-value is larger than 0.05.

All in all, we can say that there is a strong correlation between a categorical variable and other variables, if the ANOVA test gives us a large F-test value and a small p-value.

Correlation

In this video, we'll talk about the correlation between different variables.

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?

For example, smoking is known to be correlated to lung cancer. Since you have a higher chance of getting lung cancer if you smoke.

In another example, there is a correlation between umbrella and rain variables where more precipitation means more people use umbrellas. Also, if it doesn't rain people would not carry umbrellas. Therefore, we can say that umbrellas and rain are interdependent and by definition they are correlated.

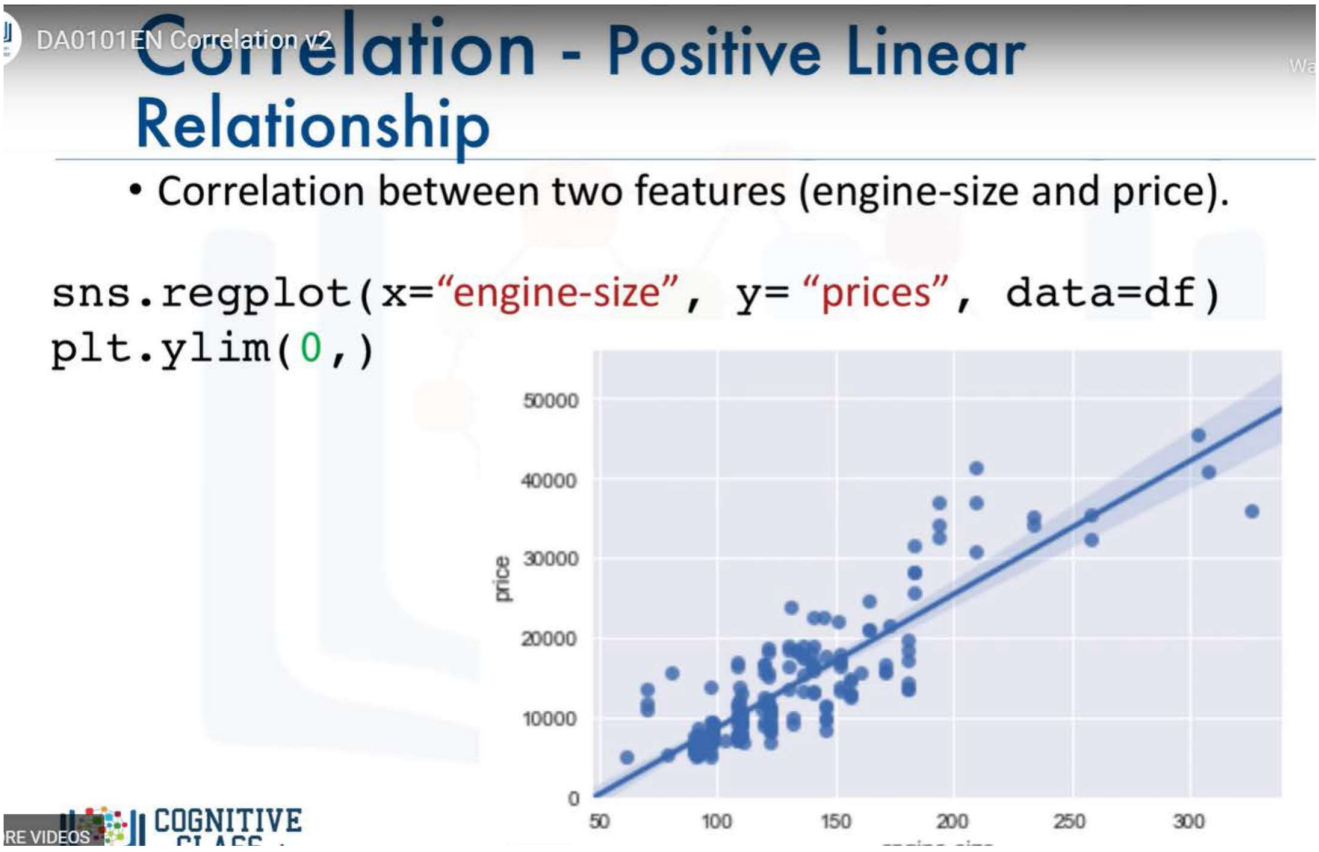
It is important to know that correlation doesn't imply causation.

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.

Let's look at the correlation between engine size and price.

This time we'll visualize these two variables using a scatter plot and an added linear line called a regression line, which indicates the relationship between the two.



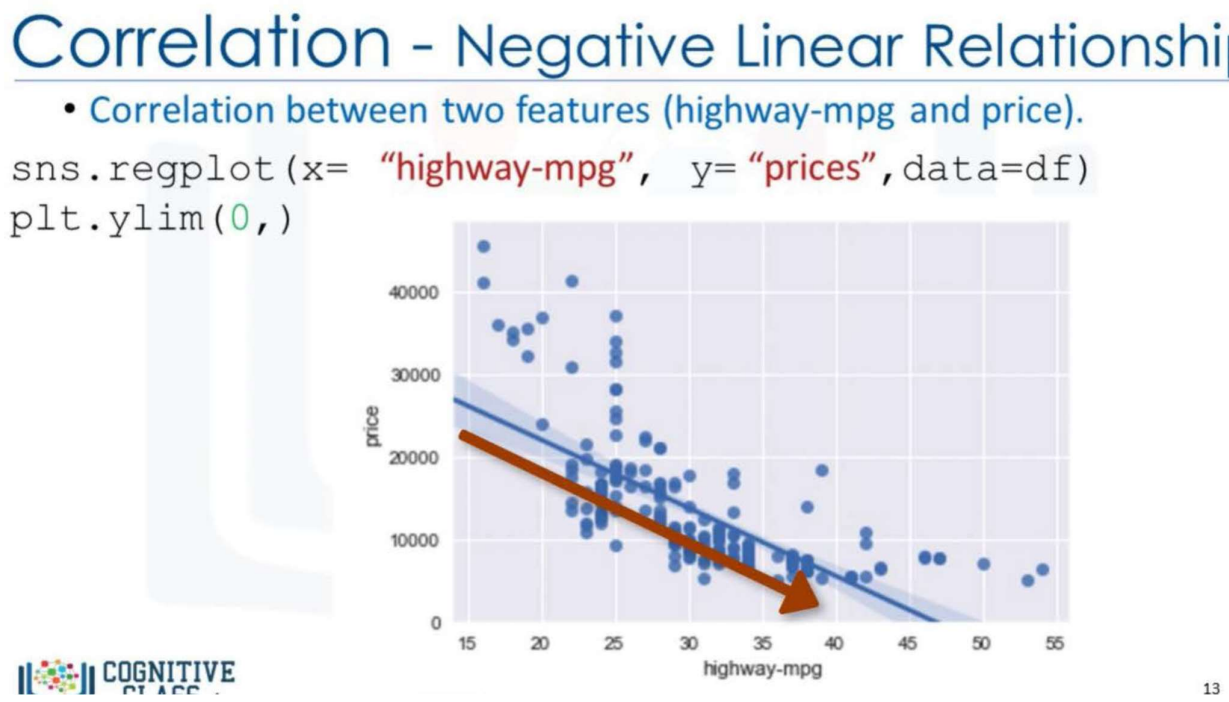
The main goal of this plot is to see whether the engine size has any impact on the price.

In this example, you can see that the straight line through the data points is very steep which shows that there's a positive linear relationship between the two variables.

With increase in values of engine size, values of price go up as well and the slope of the line is positive. So there is a positive correlation between engine size and price.

We can use **seaborn.regplot** to create the scatter plot.

As another example, now let's look at the relationship between highway miles per gallon to see its impact on the car price.



As we can see in this plot, when highway miles per gallon value goes up the value price goes down.

Therefore there is a negative linear relationship between highway miles per gallon and price.

Although this relationship is negative the slope of the line is steep which means that the highway miles per gallon is still a good predictor of price.

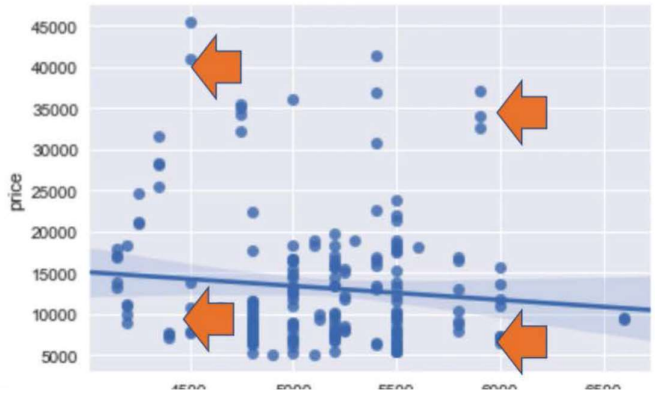
These two variables are said to have a negative correlation.

Finally, we have an example of a weak correlation.

Correlation - Negative Linear Relationship

- Weak correlation between two features (peak-rpm and price).

```
sns.regplot(x="peak-rpm", y="prices", data=df)
plt.ylim(0,)
```



For example, both low peak RPM and high values of peak RPM have low and high prices. Therefore, we cannot use RPM to predict the values.

Correlation Statistics

In this video, we'll introduce you to various correlations statistical methods.

One way to measure the strength of the correlation between continuous numerical variable is by using a method called **Pearson correlation**.

pearson_coef, p_ value = stats. pearsonr (df ('horsepower'], df ('price']

- Pearson correlation: 0.81
- P-value : 9.35 e-48

Pearson correlation method will give you two values: the correlation coefficient and the P-value.

So how do we interpret these values?
For the correlation coefficient, 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.
T
he following plot shows data with different correlation values.
In this example, we want to look at the correlation between the variable's horsepower and car price.

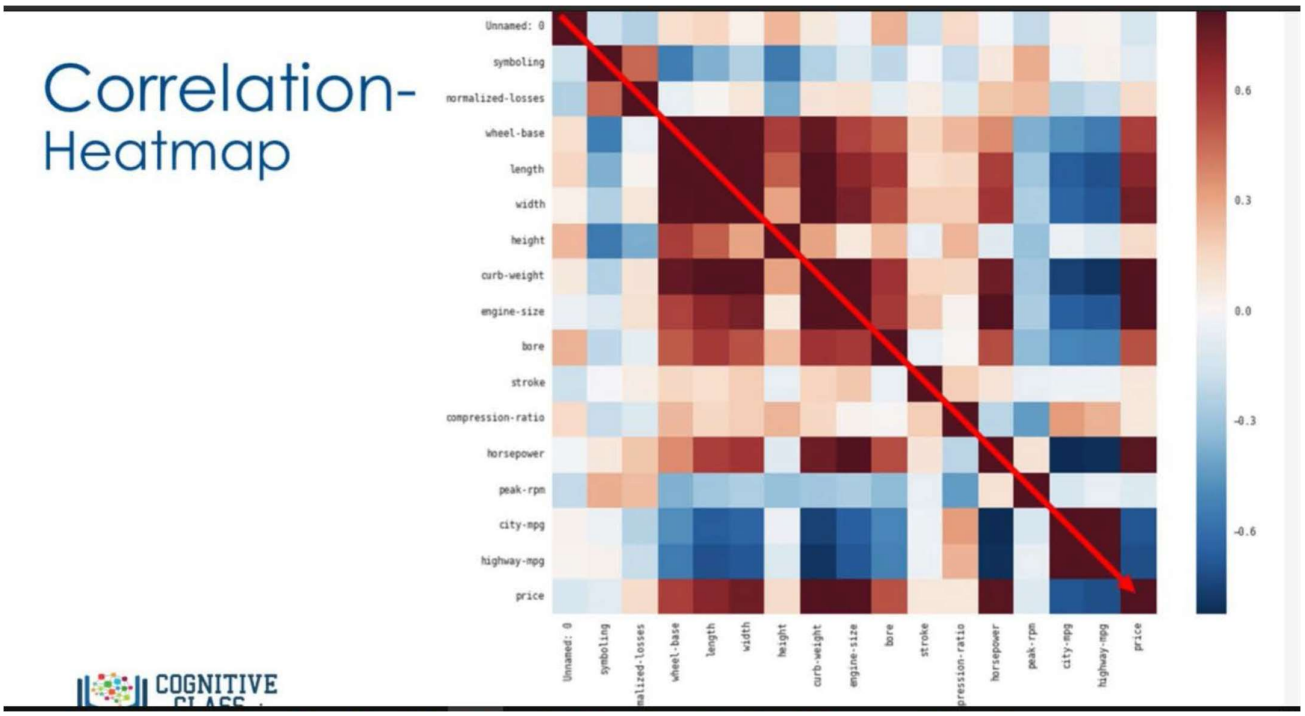
See how easy you can calculate the Pearson correlation using the SI/PI stats package?

We can see that the correlation coefficient is approximately.8, and this is close to 1.

So there is a strong positive correlation.

We can also see that the P-value is very small, much smaller than.001.

And so we can conclude that we are certain about the strong positive correlation.



Taking all variables into account, we can now create a heatmap that indicates the correlation between each of the variables with one another.

The color scheme indicates the Pearson correlation coefficient, indicating the strength of the correlation between two variables.

We can see a diagonal line with a dark red color, indicating that all the values on this diagonal are highly correlated.

This makes sense because when you look closer, the values on the diagonal are the correlation of all variables with themselves, which will be always 1.

This correlation heatmap gives us a good overview of how the different variables are related to one another and, most importantly, how these variables are related to price.

Module 4 – Model Development

Introduction

In this video, we will examine model development by trying to predict the price of a car using our dataset.

In this module you will learn about:

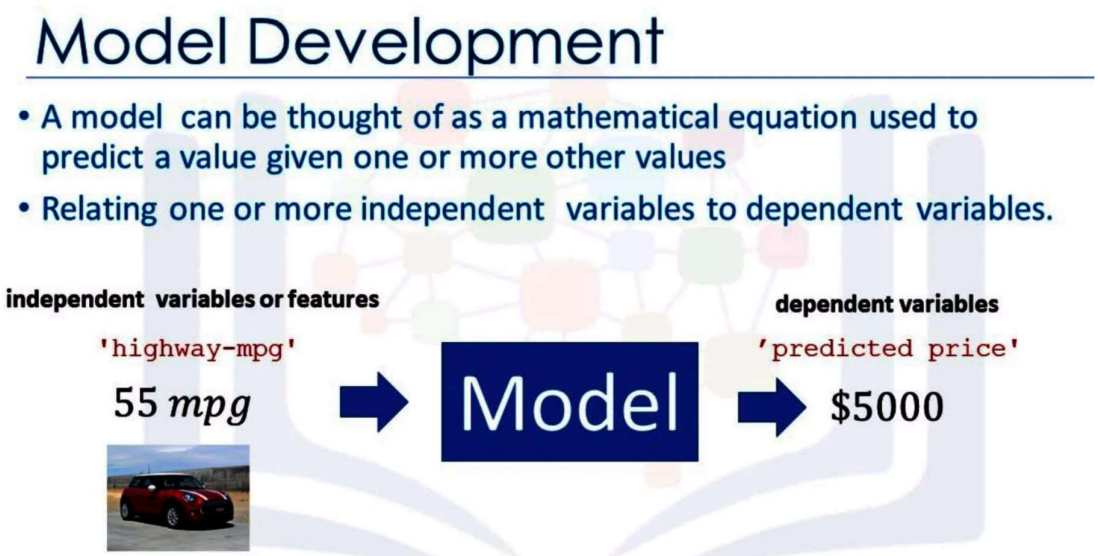
1. Simple and Multiple Linear Regression
2. Model Evaluation using Visualization
3. Polynomial Regression and Pipelines
4. R-squared and MSE for In-Sample Evaluation
5. Prediction and Decision Making

Question : how can you determine a fair value for a used car?

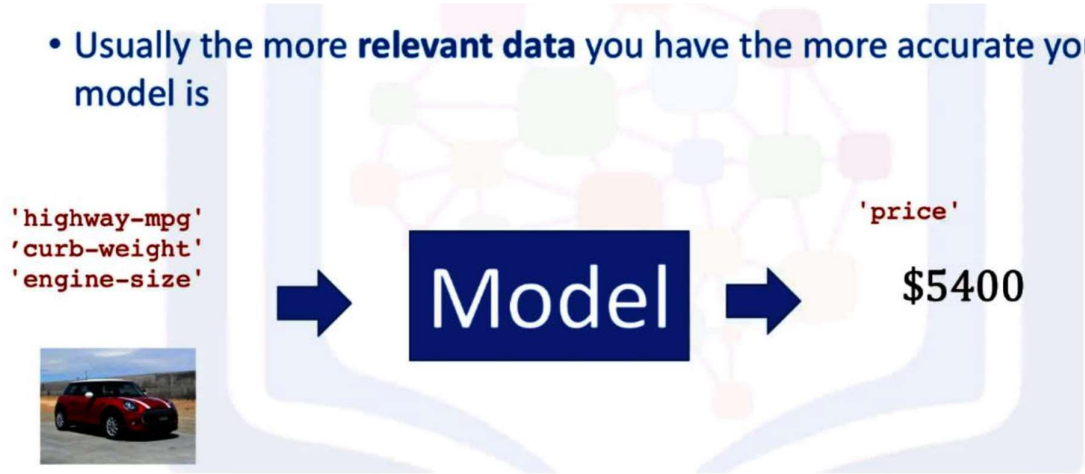
A model or estimator can be thought of as a mathematical equation used to predict a value given one or more other values.

Relating one or more independent variables or features to dependent variables.

For example, you input a car model's highway miles per gallon (MPG) as the independent variable or feature, the output of the model or dependent variable is the price.



Usually the more relevant data you have the more accurate your model is.



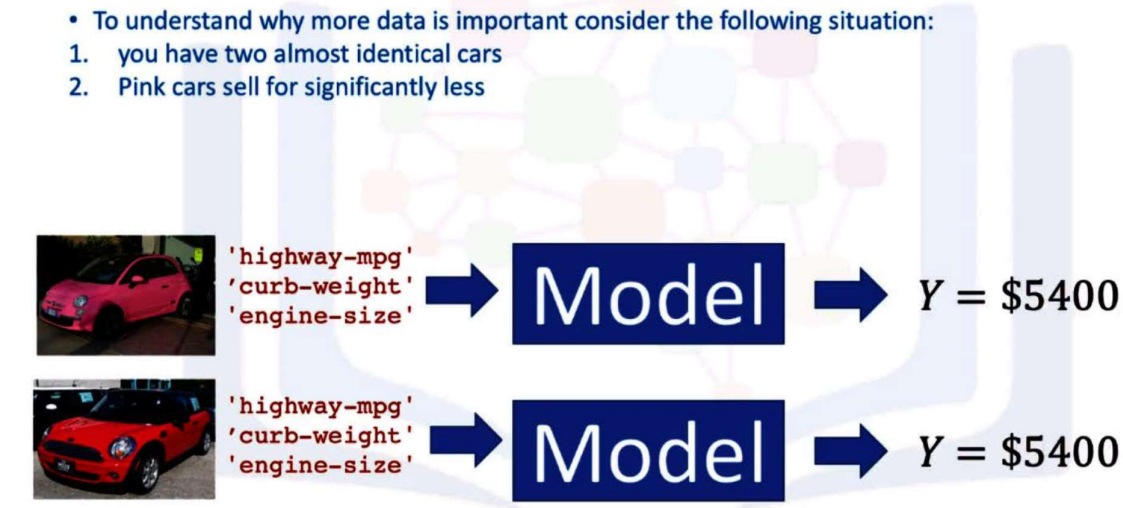
For example, you input multiple independent variables or features to your model.

Therefore, your model may predict a more accurate price for the car.

To understand why more data is important, consider the following situation:

- You have two almost identical cars - Pink cars sell for significantly less
- You want to use your model to determine the price of two cars, one pink, one red.

If your model's independent variables or features do not include color, your model will predict the same price for cars that may sell for much less.



In addition to getting more data, you can try different types of models.

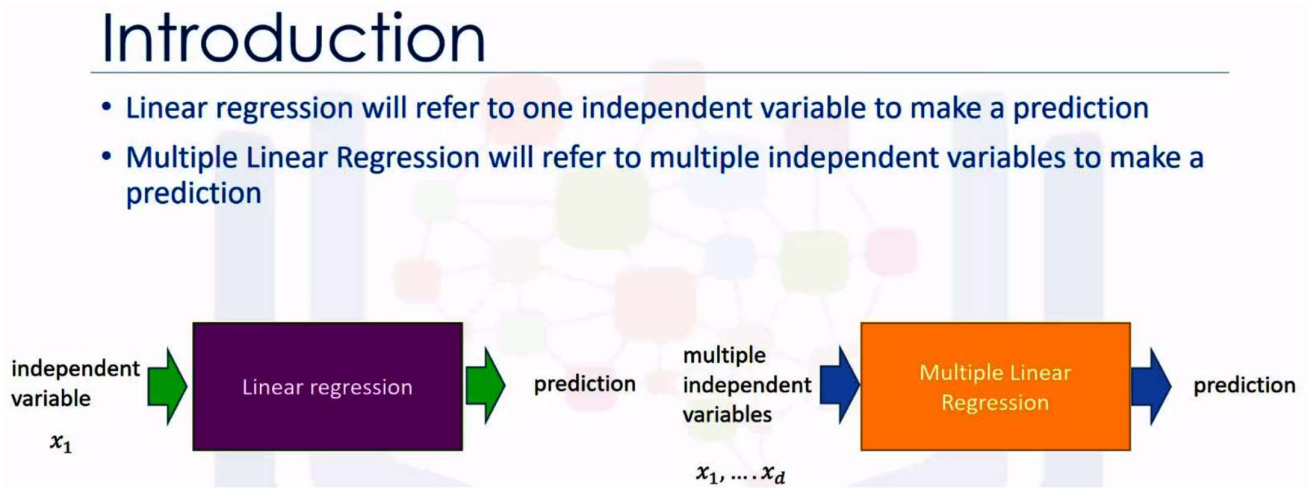
In this course you will learn about:

1. Simple Linear Regression
2. Multiple Linear Regression
3. Polynomial Regression

Linear And Multiple Linear Regression

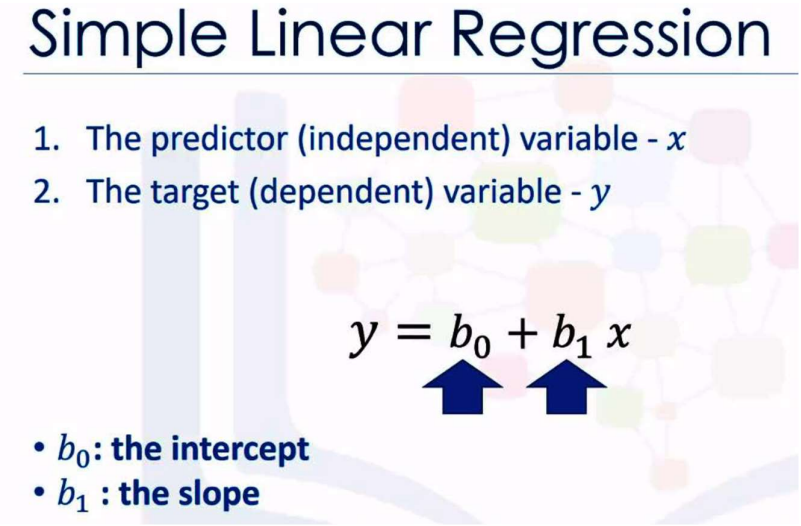
In this video, we'll be talking about simple linear regression and multiple linear regression.

- Linear Regression will refer to one independent variable to make a prediction.
- Multiple Linear Regression will refer to multiple independent variables to make a prediction.



Simple Linear Regression (or SLR) is: A method to help us understand the relationship between two variables: The predictor (independent) variable x, and the target (dependent) variable y.

We would like to come up with a linear relationship between the variables shown here.

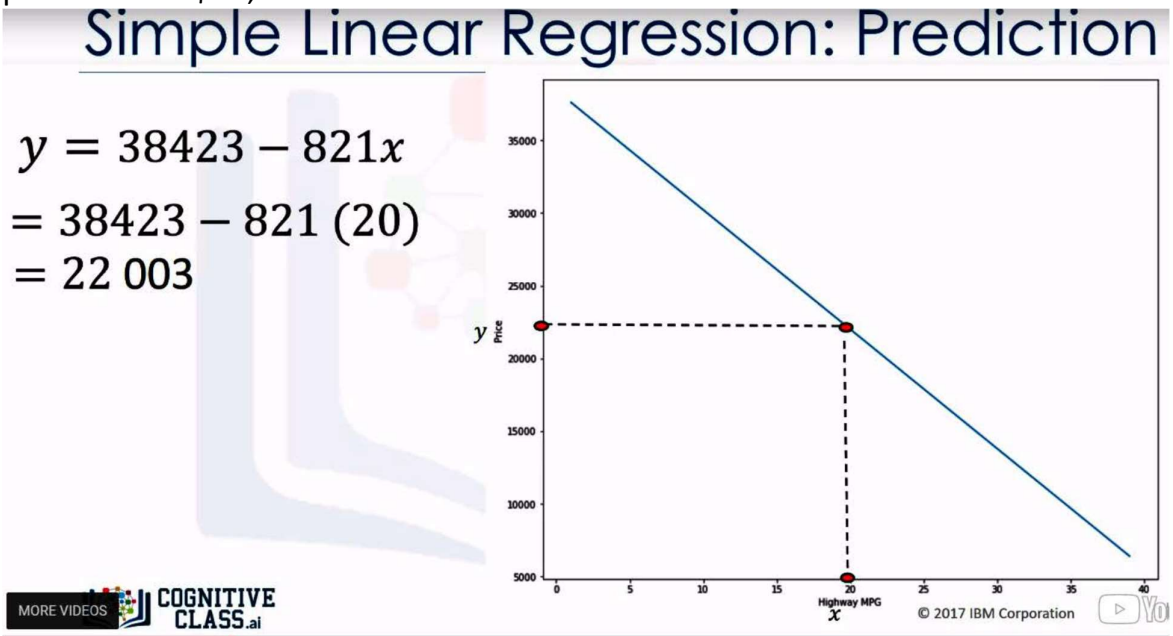


- The parameter b zero is the intercept
- The parameter b one is the slope

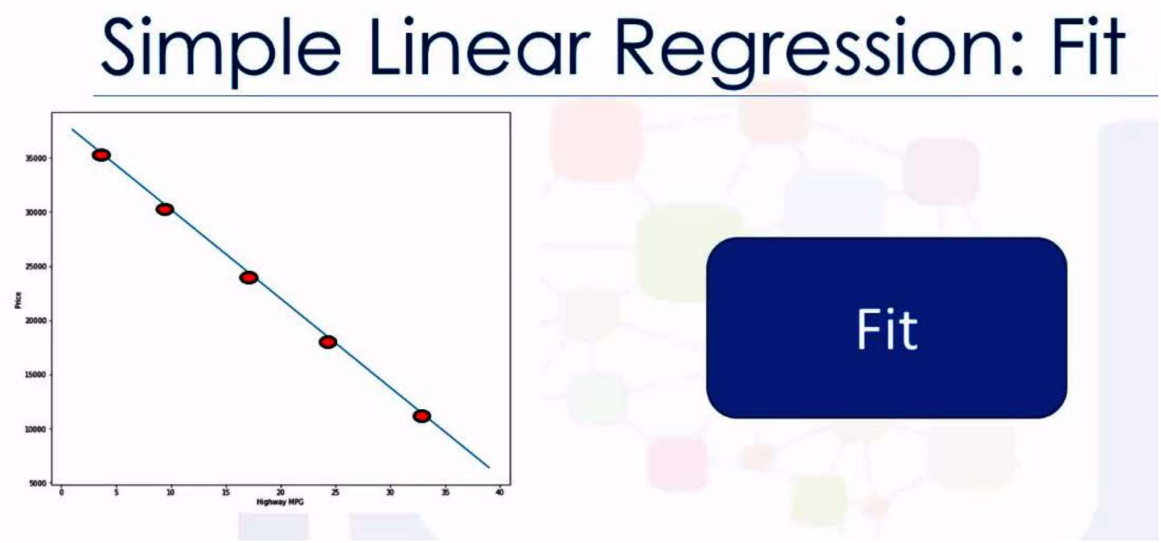
When we fit or train the model, we will come up with these parameters. This step requires lots of math, so we will not focus on this part.

Let's clarify the prediction step. It's hard to figure out how much a car costs, but the Highway Miles per Gallon is in the owner's manual. If we assume, there is a linear relationship between these variables, we can use this relationship to formulate a model to determine the price of the car.

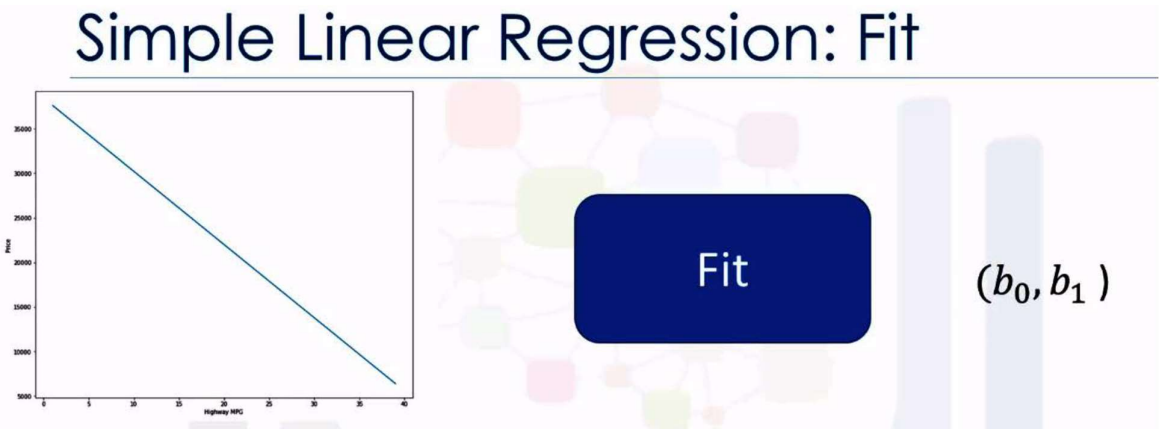
If the Highway Miles per Gallon is 20, we can input his value into the model to obtain a prediction of $22,000.



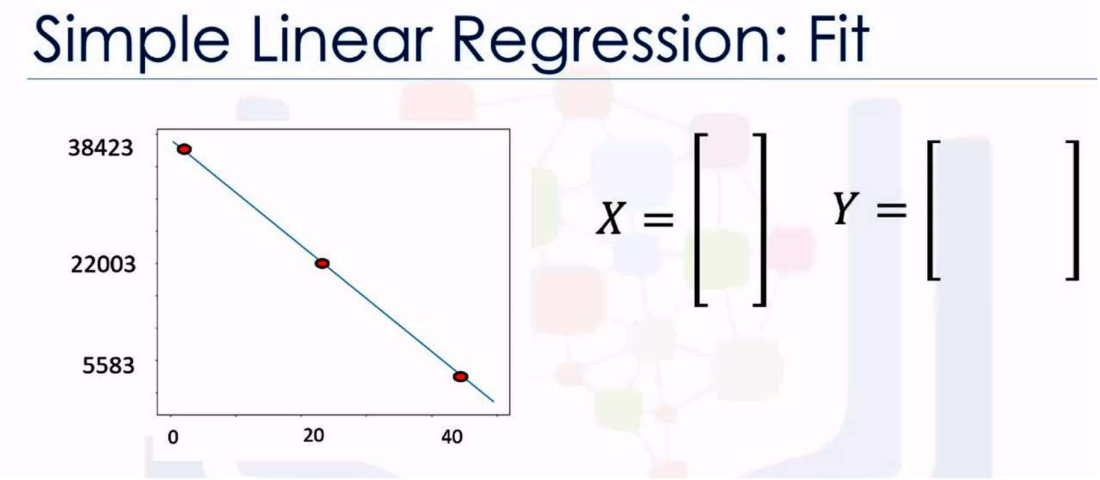
In order to determine the line, we take data points from our data set marked in red here.

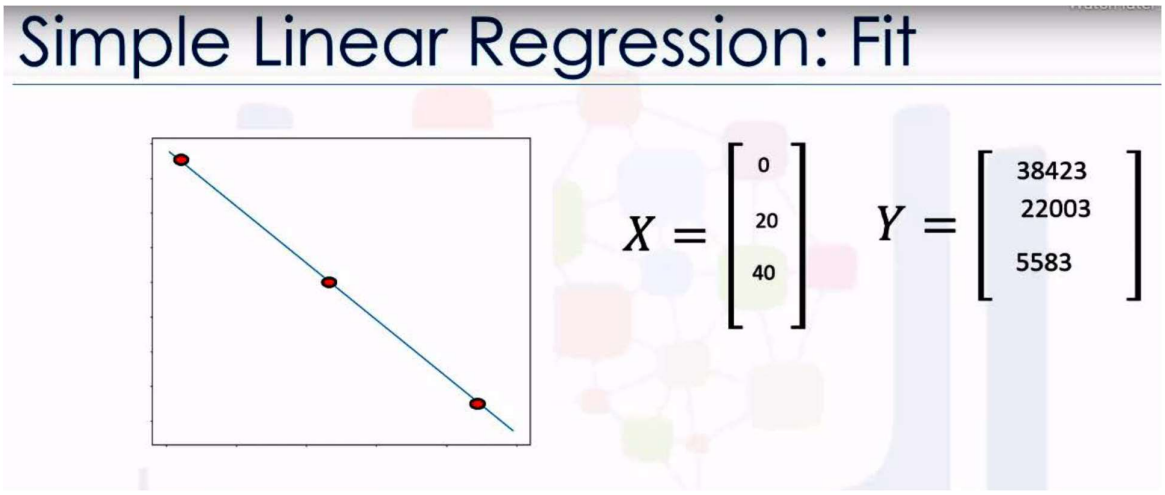


We then use these training points to fit our model; the results of the training points are the parameters.



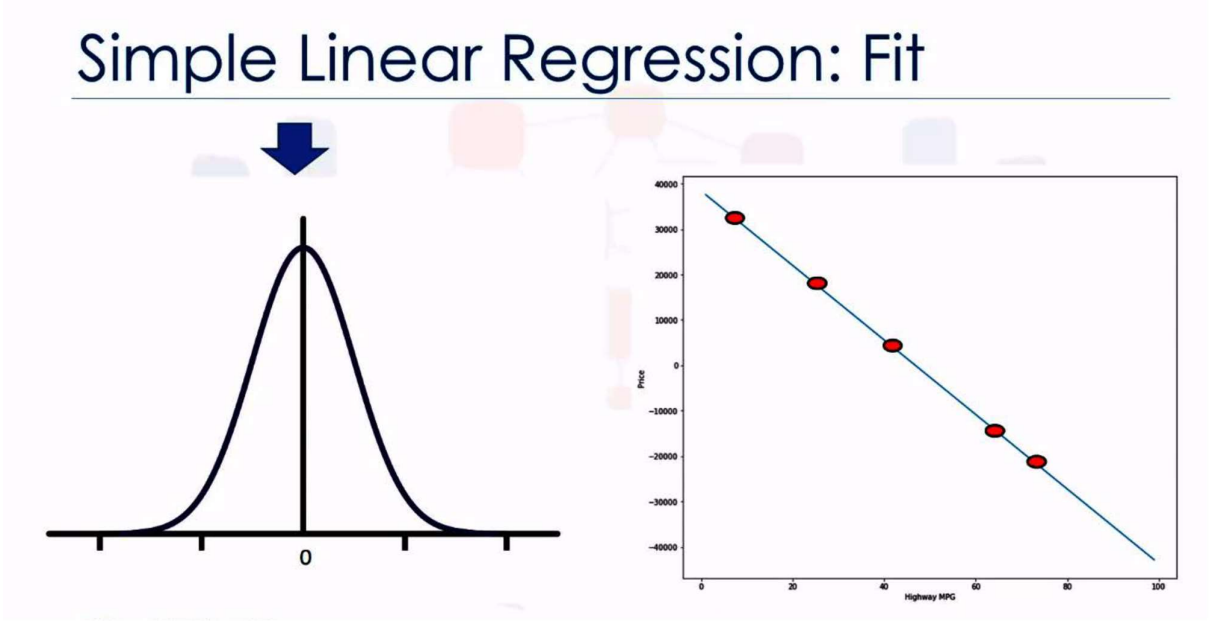
We usually store the data points in two dataframe or numpy arrays. The value we would like to predict is called the target that we store in the array y, we store the dependent variable in the dataframe or array X.





Each sample corresponds to a different row in each dataframe or array. In many cases, many factors influence how much people pay for a car, for example, make or how old the car is.

In this model, this uncertainty is taken into account by assuming a small random value is added to the point on the line; this is called noise.



The figure on the left shows the distribution of the noise. The vertical axis shows the value added and the horizontal axis illustrates the probability that the value will be added. Usually, a small positive value is added, or a small negative value. Sometimes large values are added, but for the most part, the values added are near zero.

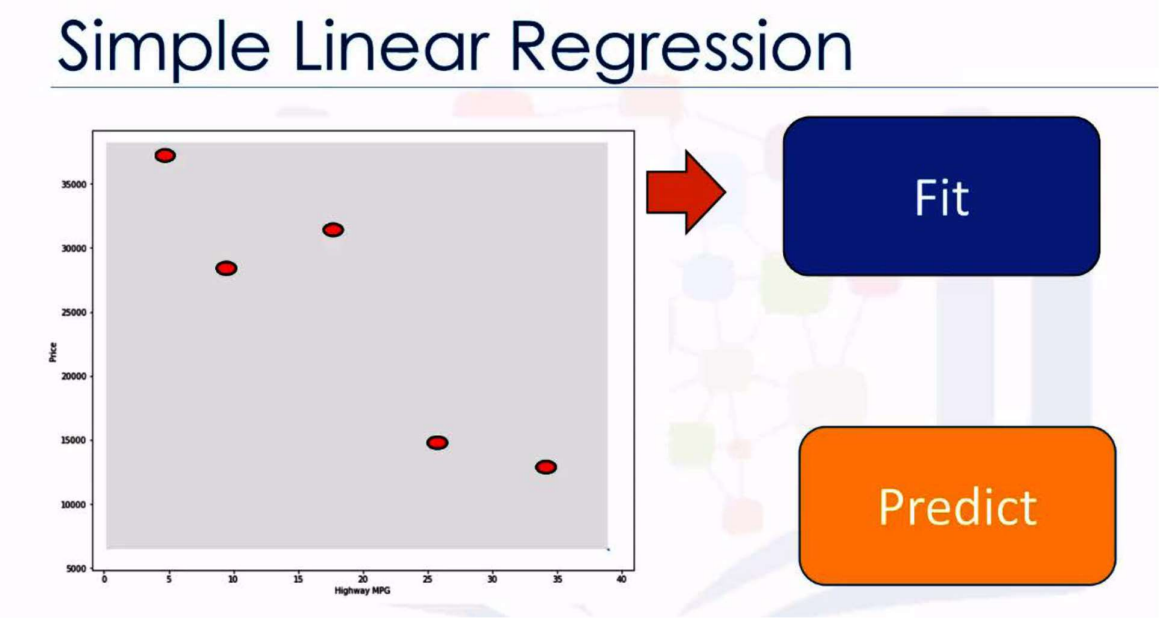
We can summarize the process like this:

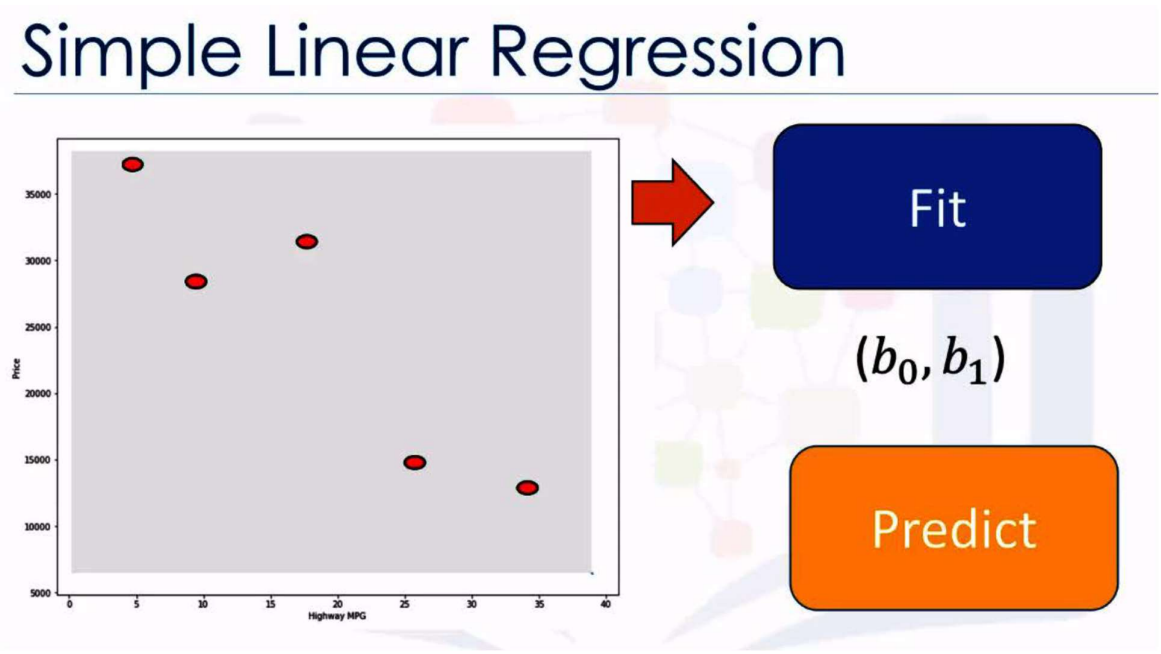
1. We have a set of training points - We use these training points to fit or train the model and get parameters - We then use these parameters in the model
2. We now have a model; we use the hat on the y to denote the model is an estimate
3. We can use this model to predict values that we haven't seen.

For example, we have no car with 20 Highway Miles per Gallon, we can use our model to make a prediction for the price of this car. But don't forget our model is not always correct.

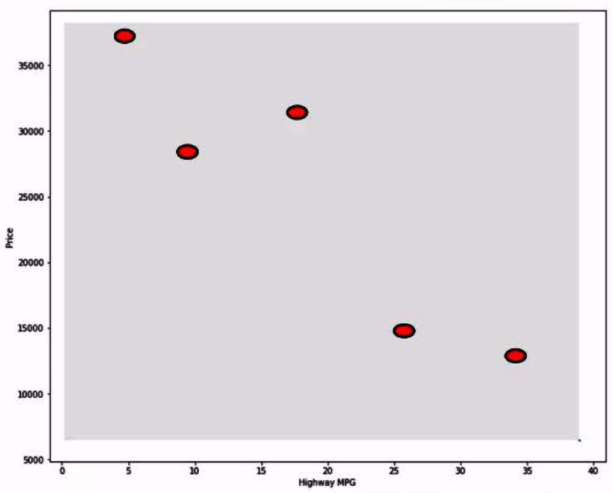
We can see this by comparing the predicted value to the actual value.

We have a sample for 10 Highway Miles per Gallon, but the predicted value does not match the actual value. If the linear assumption is correct this error is due to the noise but there can be other reasons.





Simple Linear Regression



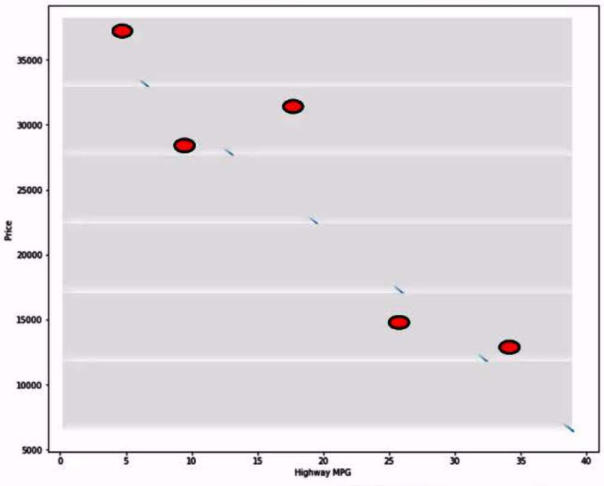
Fit

Predict

$$\hat{y} = b_0 + b_1 x$$

COGNITIVE CLASS.ai

Simple Linear Regression



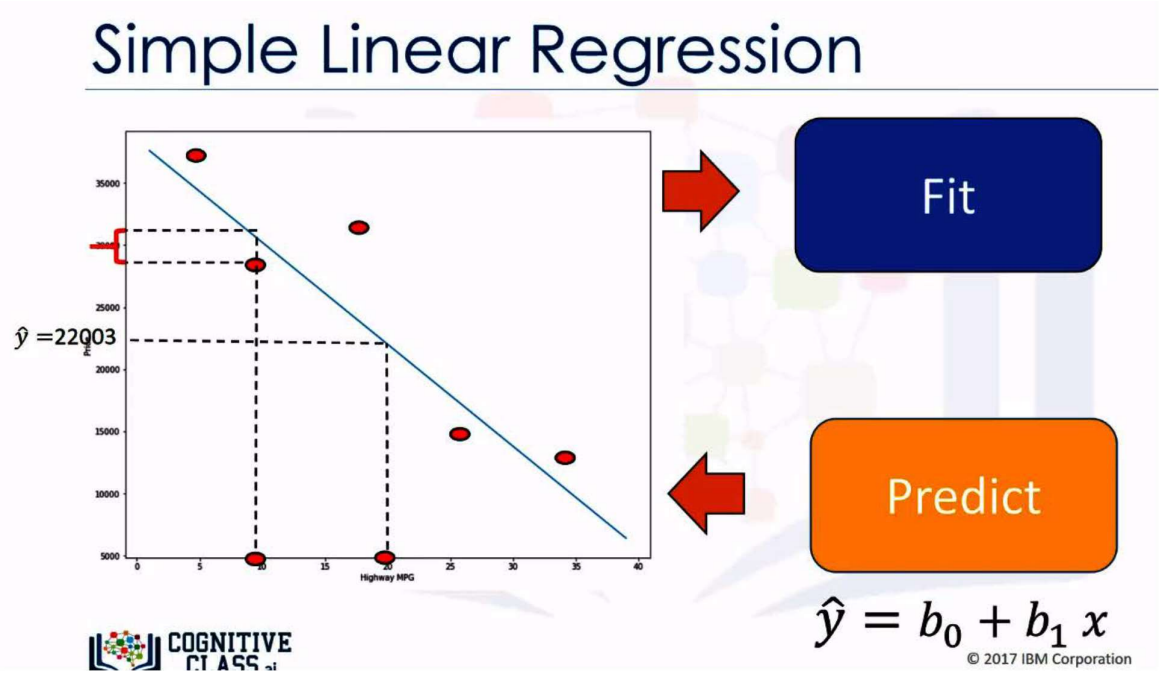
Fit

Predict

$$\hat{y} = b_0 + b_1 x$$

COGNITIVE CLASS.ai

© 2017 IBM Corporation

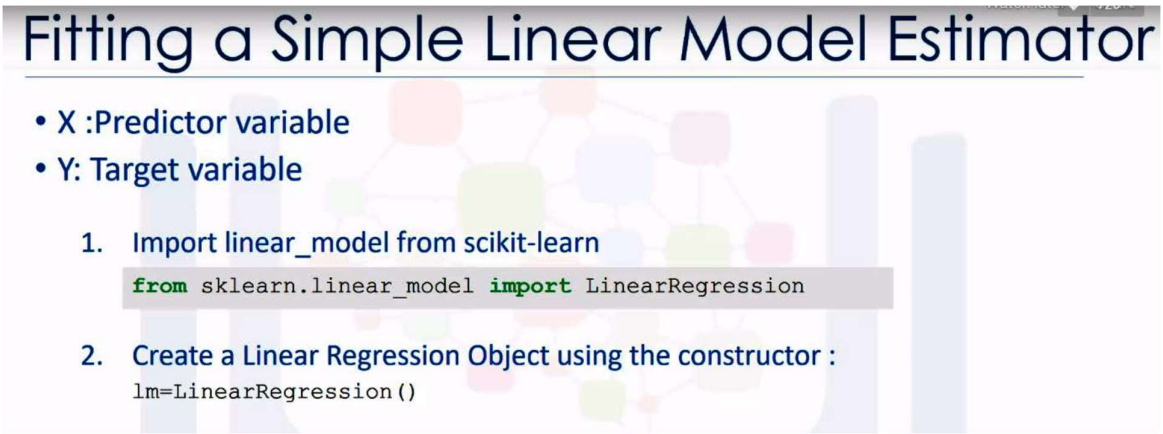


To fit the model in Python, first we import linear model from **scikit-learn**; then Create a Linear Regression Object using the constructor.

- X :Predictor variable
- Y: Target variable

1. Import linear _model from sci kit-learn
   from sklearn.linear_model import LinearRegression

2. Create a Linear Regression Object using the constructor :
   lm=LinearRegression()



We define the predictor variable and target variable. Then use the method fit to fit the model and find the parameters b0 and b1. The input are the features and the targets.

We can obtain a prediction using the method predict.

- We define the predictor variable and target variable

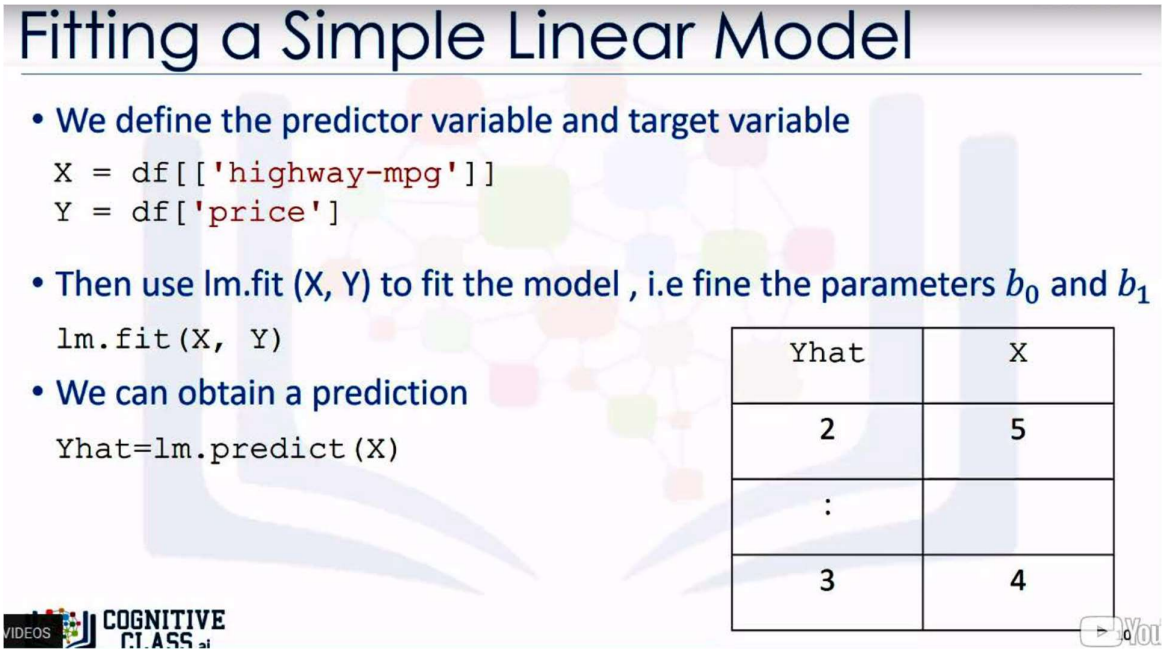
  X = df[['highway-mpg']]

  Y = df['price']
- Then use Im.fit (X, Y) to fit the model , i.e fine the parameters b0 and b1

  lm.fit(X, Y)

We can obtain a prediction

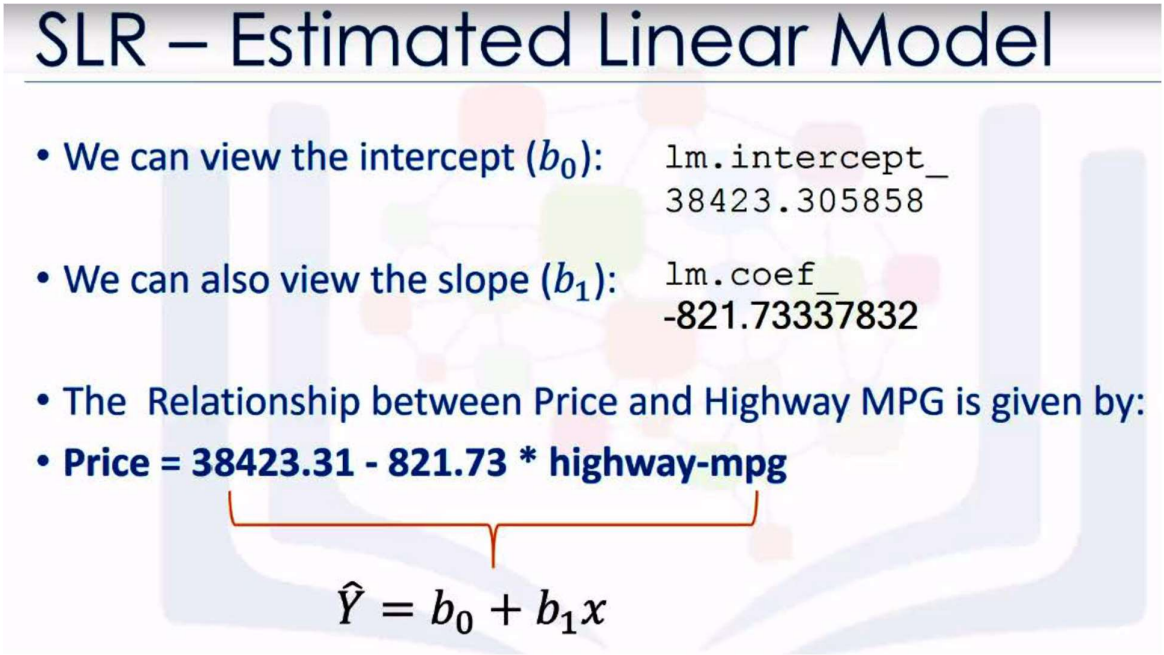
Yhat=lm.predict(X)



The output is an array, the array has the same number of samples as the input X.

The intercept (b0) is an attribute of the object “lm”.
The slope (b1) is also an attribute of the object “lm”.

The Relationship between Price and Highway MPG is given by this equation in bold:

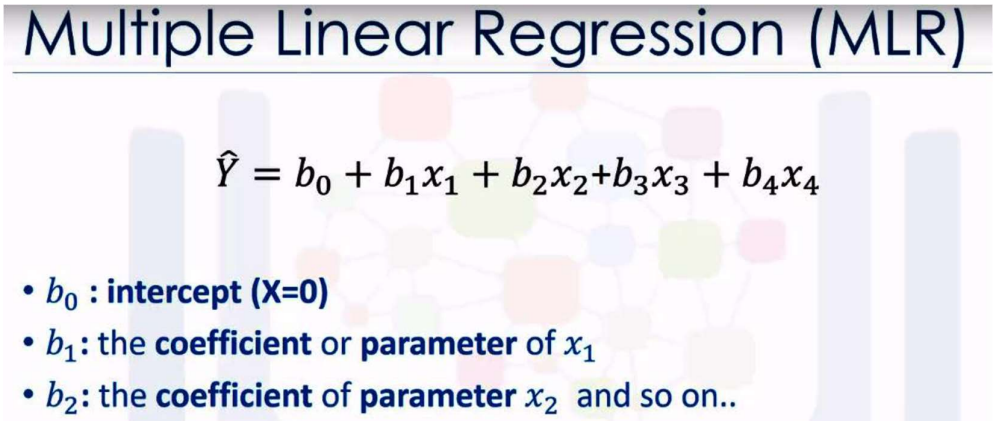


"Price = 38,423.31 minus 821.73 times highway mpg" like the equation we discussed before.

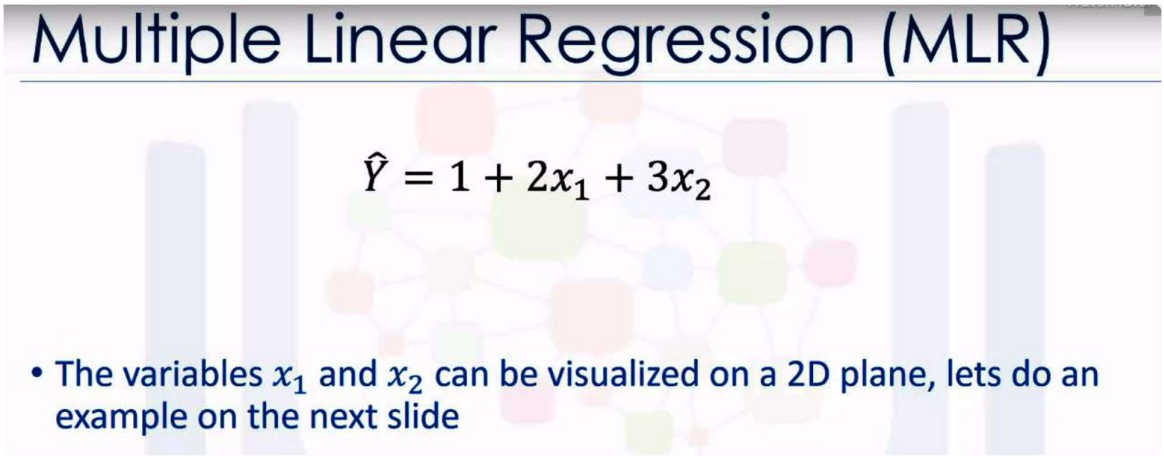
Multiple Linear Regression is used to explain the relationship between

- One continuous target (Y) variable, and
- Two or more predictor (X) variables.

If we have for example 4 predictor variables, then:
- B0: intercept (X=0)
- B1: the coefficient or parameter of $X1$:
- B2: the coefficient of parameter $X2$: and so on

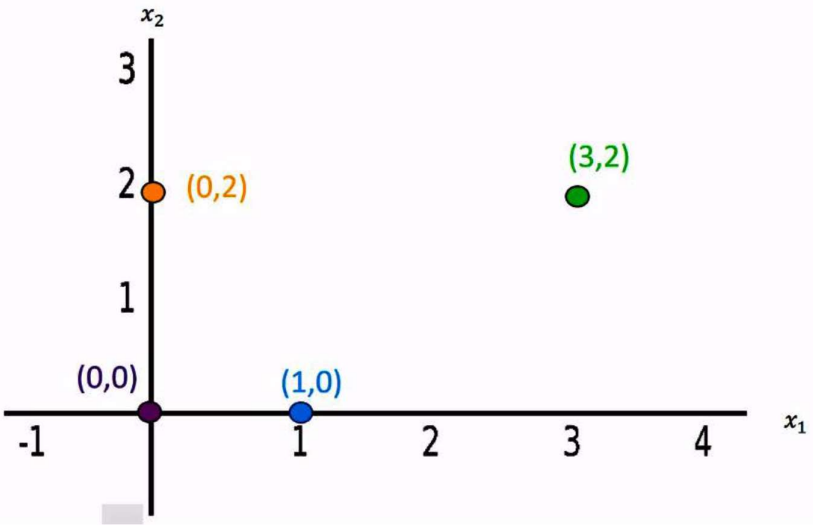


If there are only two variables then we can visualize the values. Consider the following function.



The variables $X1$and $X2$ can be visualized on a 2D plane; let's do an example on the next slide.

n	x_1	x_2
1	0	0
2	0	2
3	1	0
4	3	2

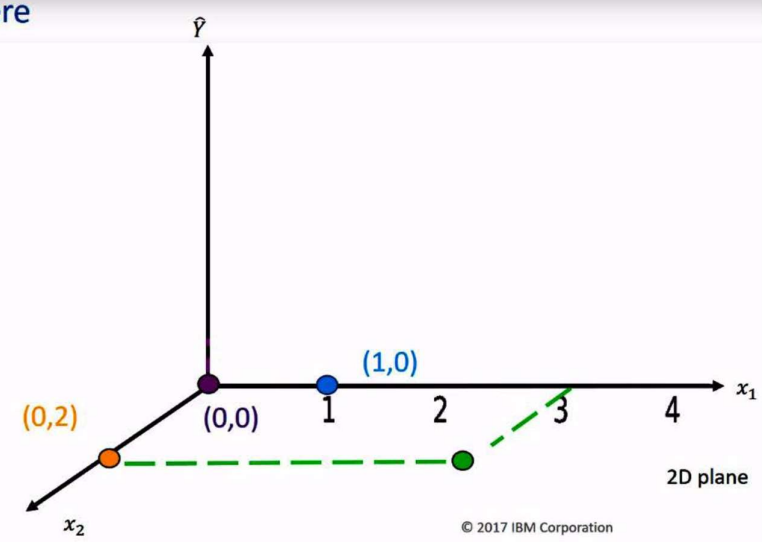


The table contains different values of the predictor variables $X1$ and $X2$. The position of each point is placed on the 2D plane, color coded accordingly.

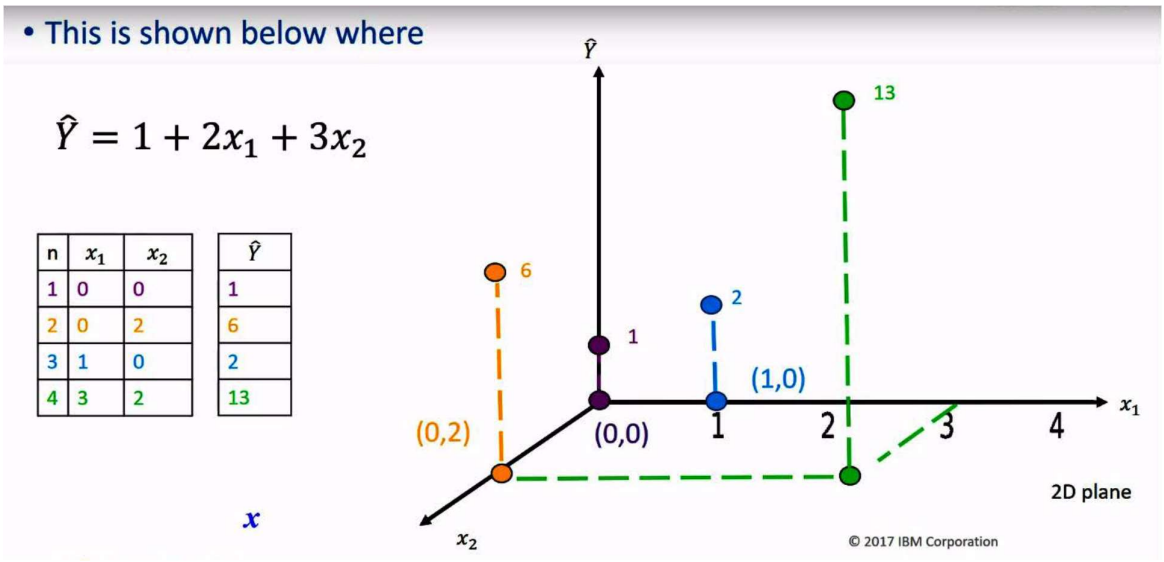
- This is shown below where

$$\hat{Y} = 1 + 2x_1 + 3x_2$$

n	x_1	x_2	$\hat{Y}$
1	0	0	1
2	0	2	6
3	1	0	2
4	3	2	13

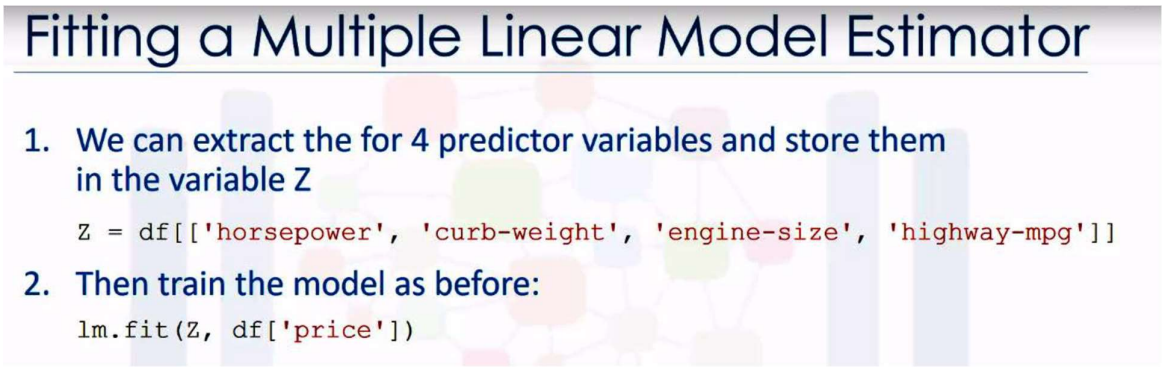


Each value of the predictor variables $X1$ and $X2$ will be mapped to a new value Y (y hat) the new values of Y (y hat) are mapped in the vertical direction, with height proportional to the value that yhat takes.



We can fit the Multiple linear regression as follows:

- We can extract the 4 predictor variables and store them in the variable Z.
- Then train the model as before using the method train, with the features or depended variables and the targets : (colon)



We can also obtain a prediction using the method predict. In this case, the input is an array or dataframe with 4 columns, the number of rows correspond to the number of samples.

The output is an array with the same number of elements as number of samples.

The intercept is an attribute of the object. And the coefficients are also attributes.

It is helpful to visualize the equation, replacing the dependent variable names with actual names.

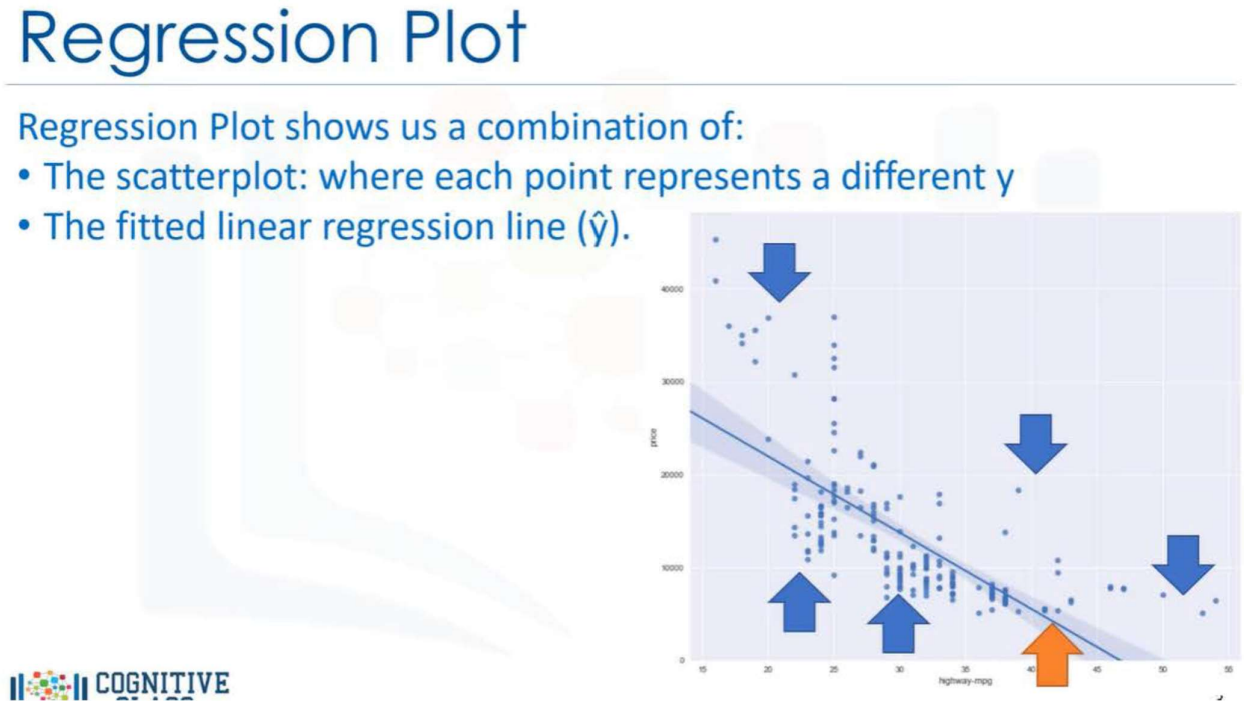
This is identical to the form we discussed earlier.

Model Evaluation Using Visualization

In this video, we'll look at model evaluation using visualization.

Regression plots are a good estimate of:

- The relationship between two variables,
- The strength of the correlation, and
- The direction of the relationship (positive or negative).
- The horizontal axis is the independent variable.



The vertical axis is the dependent variable.

Each point represents a different target point.
The fitted line represents the predicted value.

There are several ways to plot a regression plot; a simple way is to use "regplot" from the seaborne library.

First import seaborn.
Then use the "regplot" function.
The parameter x is the name of the column that contains the dependent variable or feature.
The parameter y contains the name of the column that contains the name of the dependent variable
or target.
The parameter data is the name of the dataframe.
The result is given by the plot.

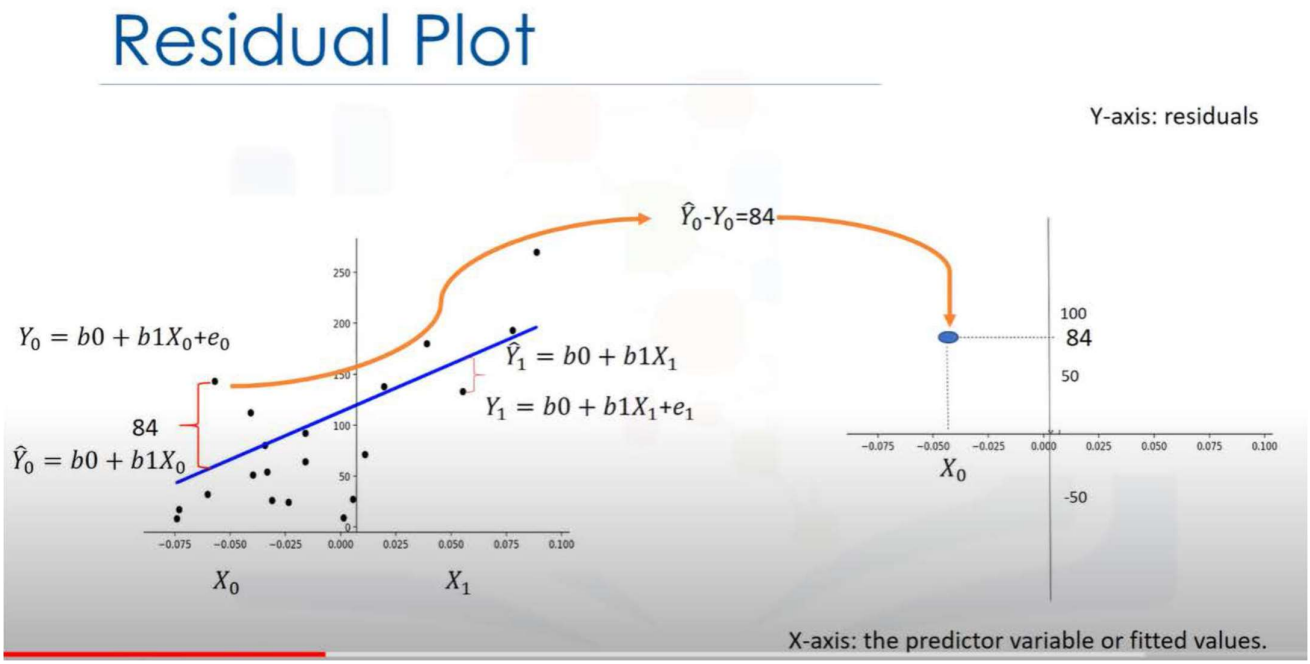
import seaborn as sns
sns.regplot(x="highway-mpg", y="price", data=df) plt.ylim(O,)

The residual plot represents the error between the actual values.

Examining the predicted value and actual value we see a difference.

We obtain that value by subtracting the predicted value and the actual target value.

We then plot that value on the vertical axis, with the dependent variable as the horizontal axis.

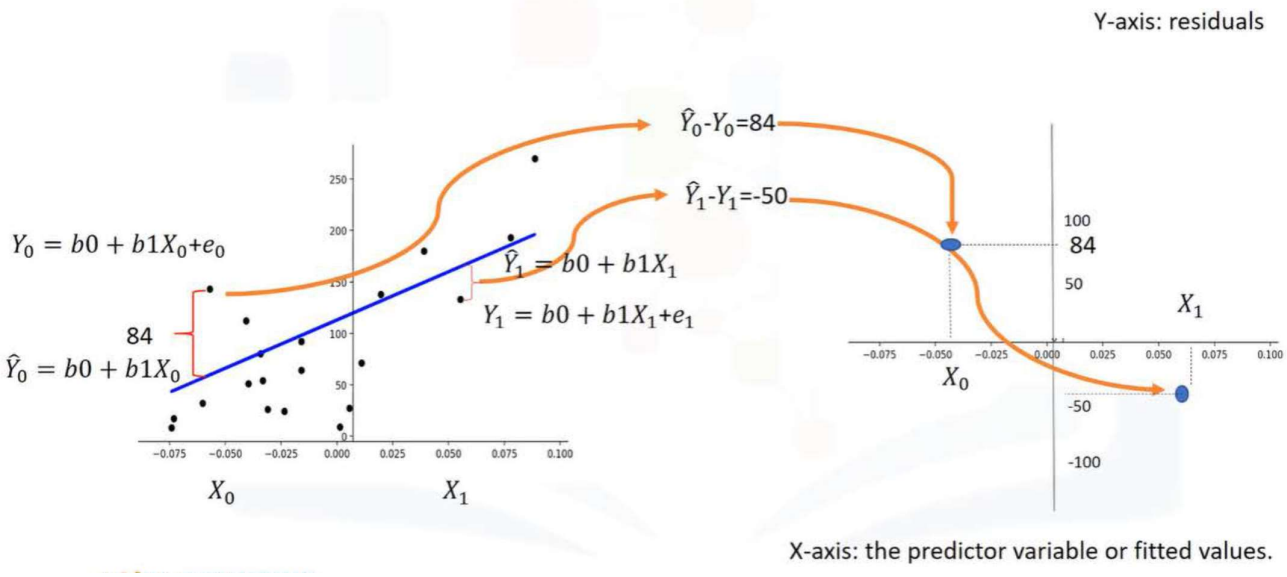


Similarly, for the second sample, we repeat the process.

Subtracting the target value from the predicted value; then plotting the value accordingly.

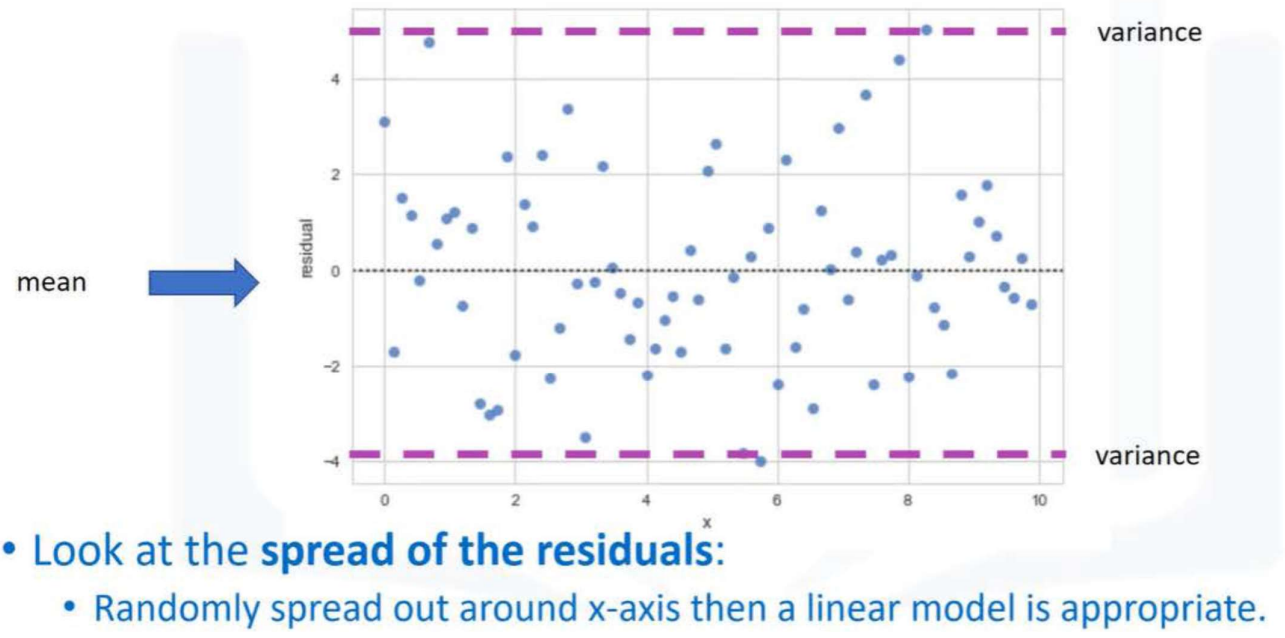
Looking at the plot gives us some insight into our data.

Residual Plot



We expect to see the results to have zero mean.

Residual Plot

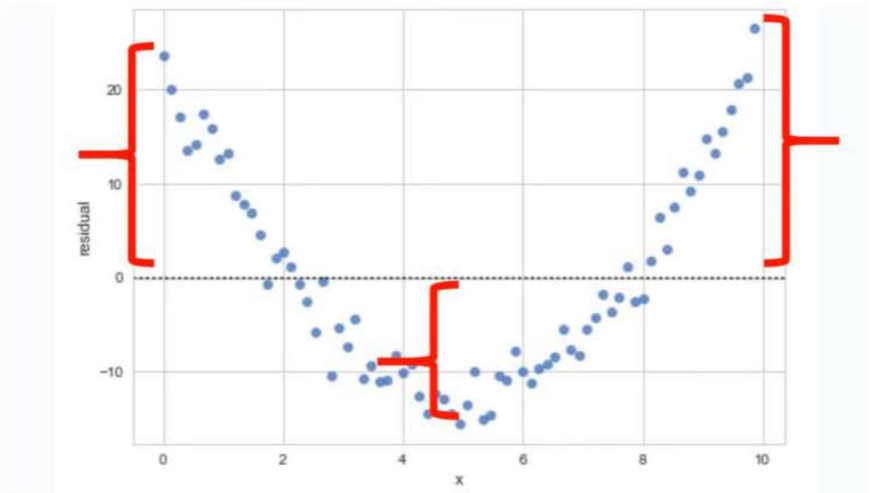


- Look at the **spread of the residuals**:
  - Randomly spread out around x-axis then a linear model is appropriate.

Distributed evenly around the x axis with similar variance; there is no curvature.

This type of residual plot suggests a linear plot is appropriate.
In this residual plot there is curvature, the values of the error change with x.

Residual Plot



- Not randomly spread out around the x-axis
- Nonlinear model may be more appropriate

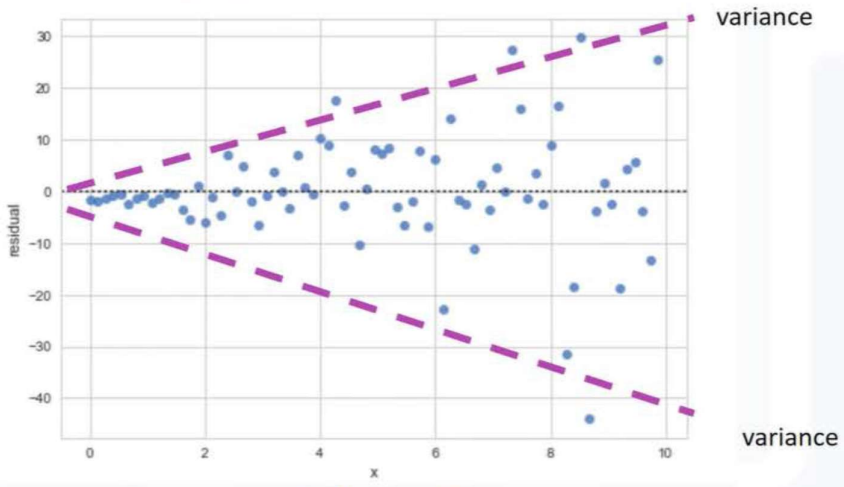
For example, in the region, all the residual errors are positive.
In this area, the residuals are negative.
In the final location, the error is large.

The residuals are not randomly separated; this suggests the linear assumption is incorrect.

This plot suggests a non-linear function, we will deal with this in the next section.

In this plot, we see that variance of the residuals increases with x, therefore our model is incorrect.

Residual Plot



- Not randomly spread out around the x-axis
- Variance appears to change with x axis

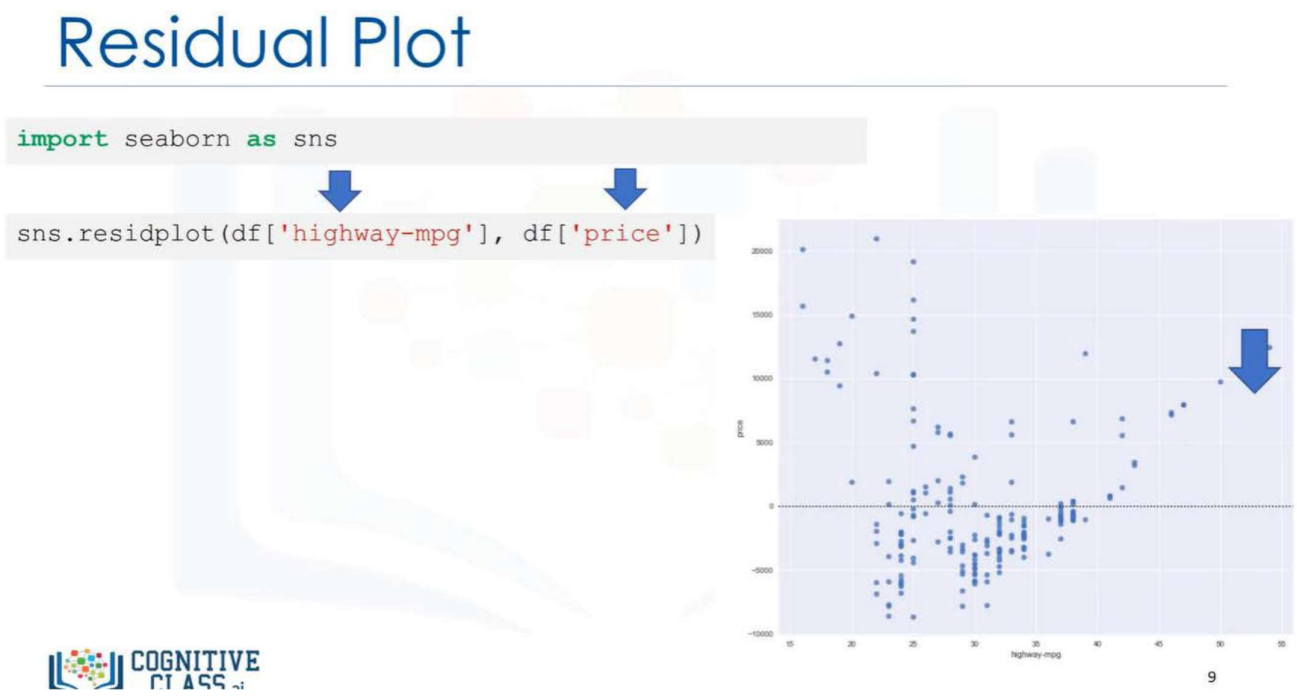
We can use seabborn to create a Residual Plot.

First import seaborn.
We use the “residplot” function.

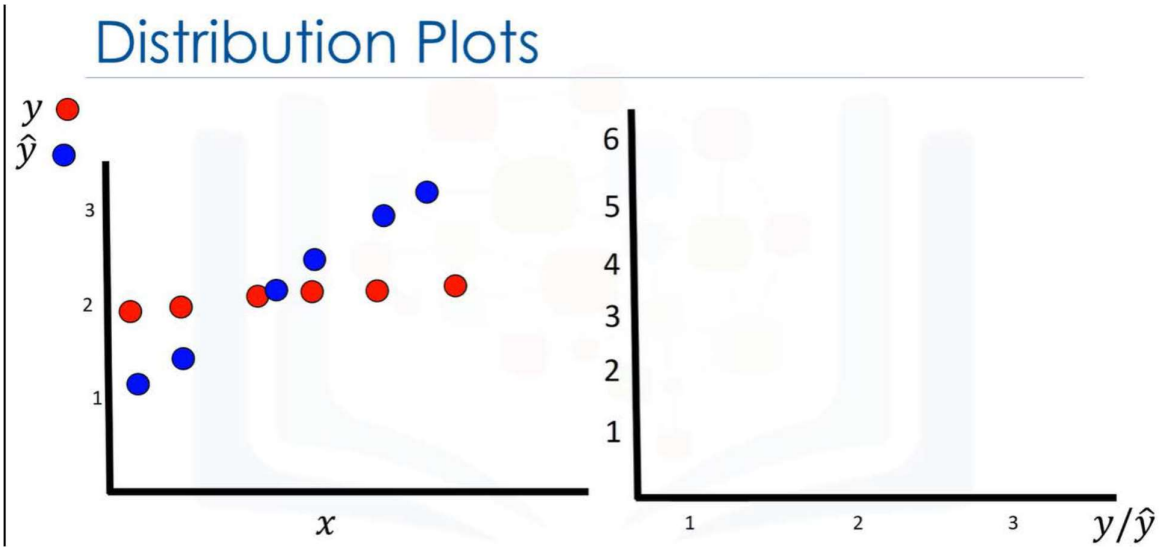
The first parameter is a series of dependent variable or feature.
The second parameter is a Series of dependent variable or target.

We see in this case the Residuals have a curvature.

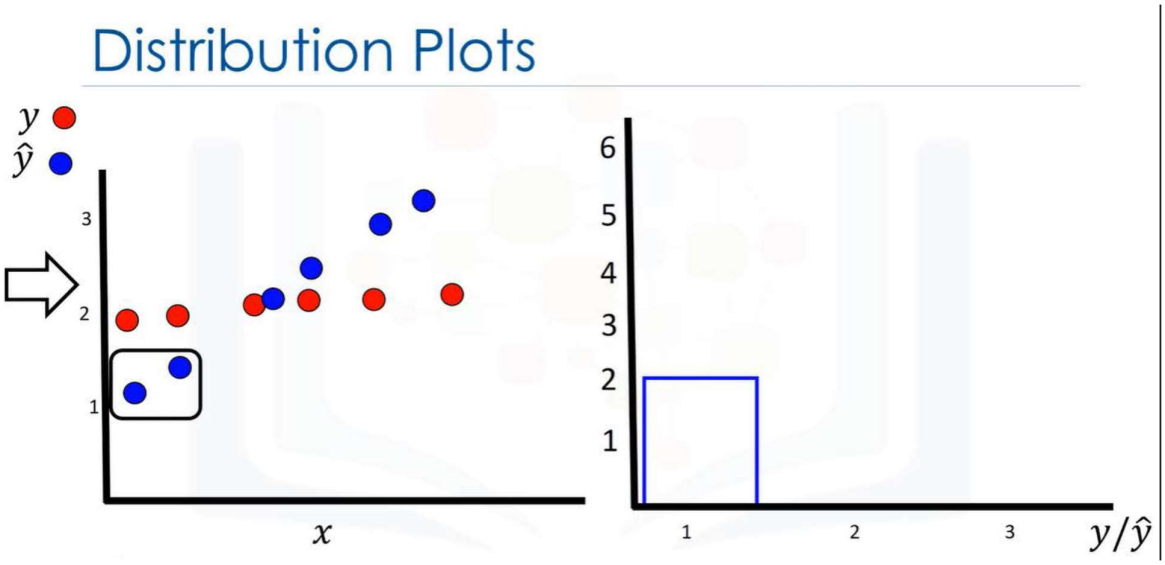
```
import seaborn as sns
sns.residplot(df['highway-mpg'], df['price'])
```



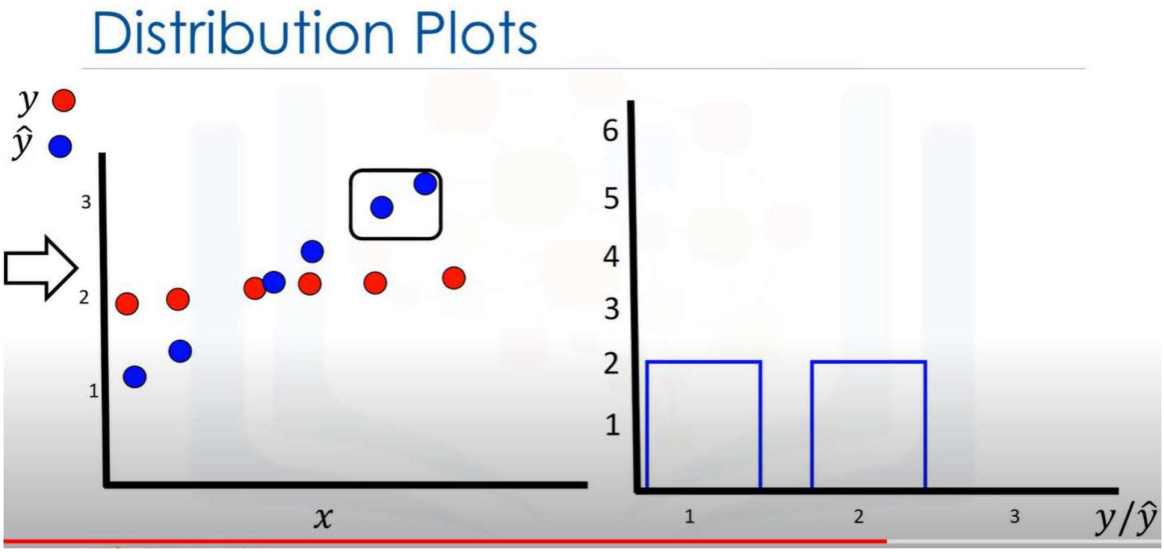
A distribution plot counts the predicted value versus the actual value.
These plots are extremely useful for visualizing models with more than one independent variable or feature.



Let's look at a simplified example: - We examine the vertical axis.
- We then count and plot the number of predicted points that are approximately equal to one.

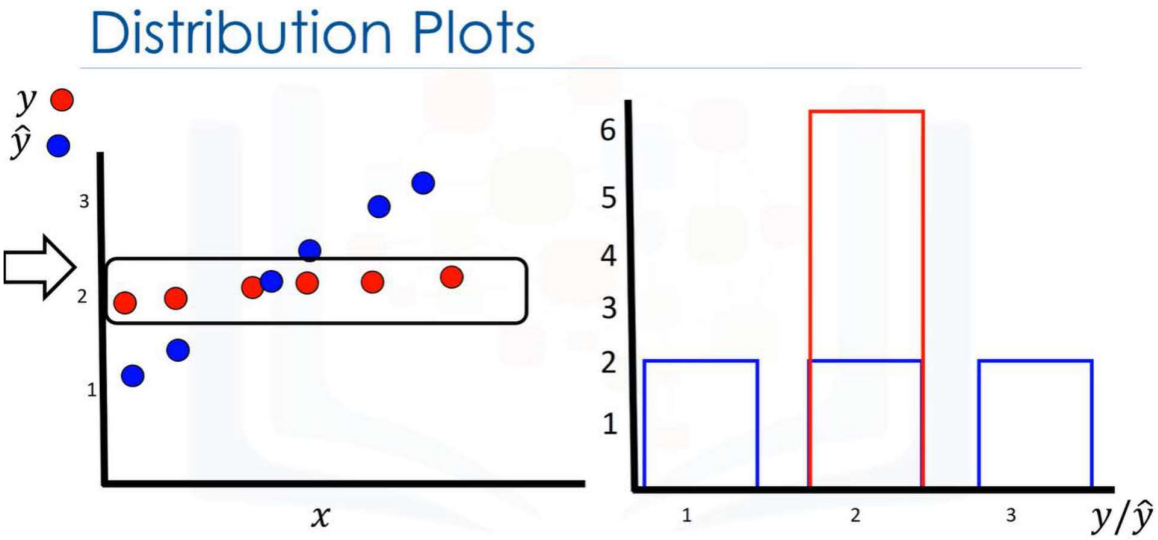


- We then count and plot the number of predicted points that are approximately equal to two.

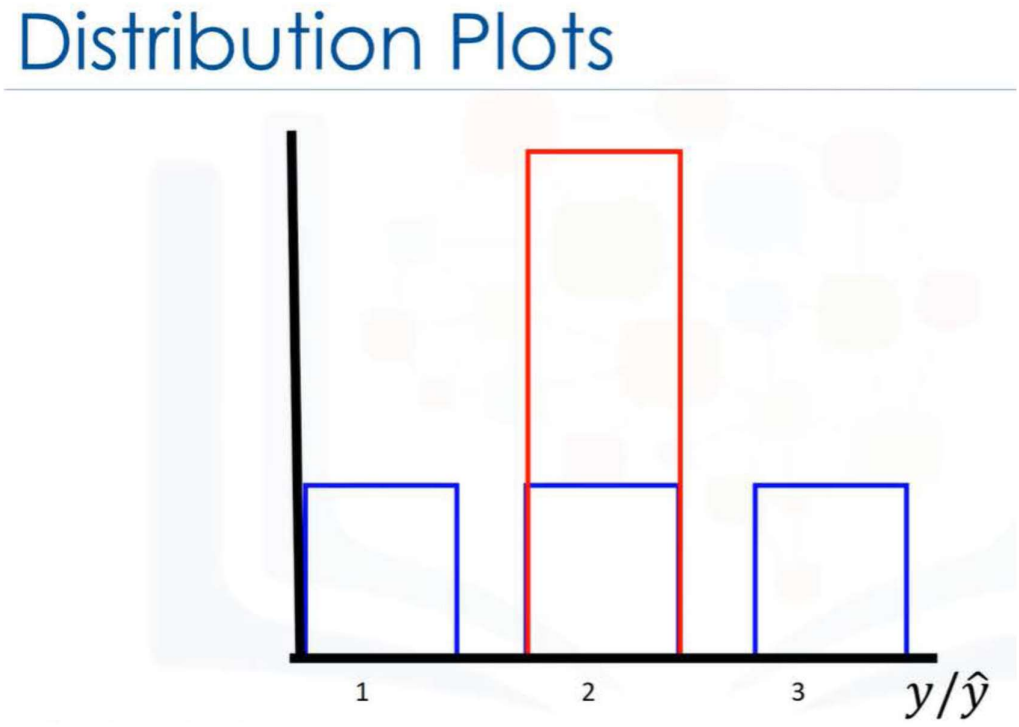


- We repeat the process for predicted points that are approximately equal to three.

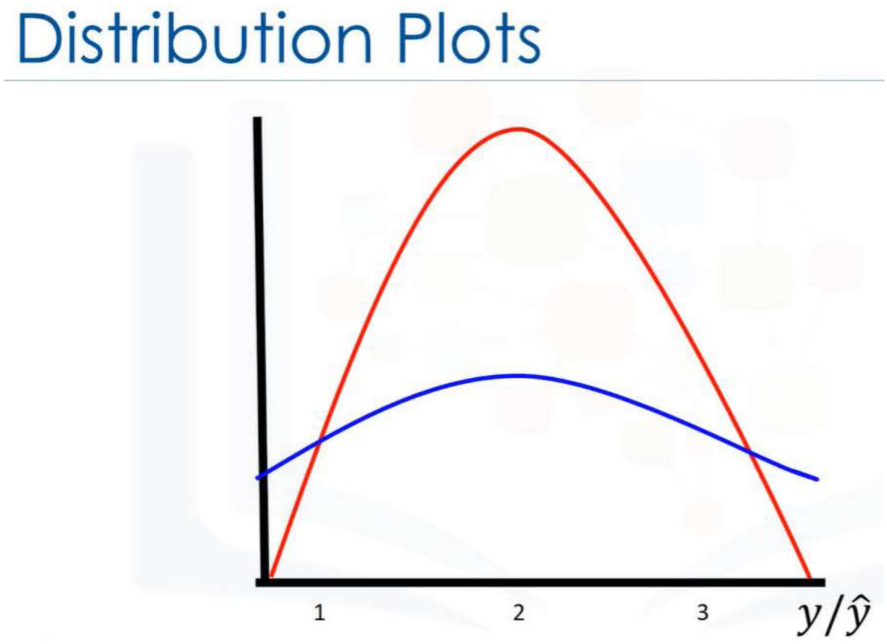
Then we repeat the process for the target values.
In this case, all the target values are approximately equal to two.



The values of the targets and predicted values are continuous.



A histogram is for discrete values.
Therefore pandas will convert them to a distribution.
The vertical access is scaled to make the area under the distribution equal to one.

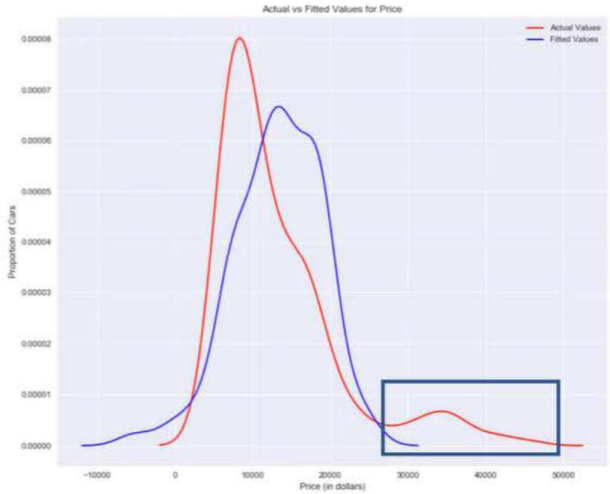


This is an example of using a distribution plot.

Distribution Plots

Compare the distribution plots:

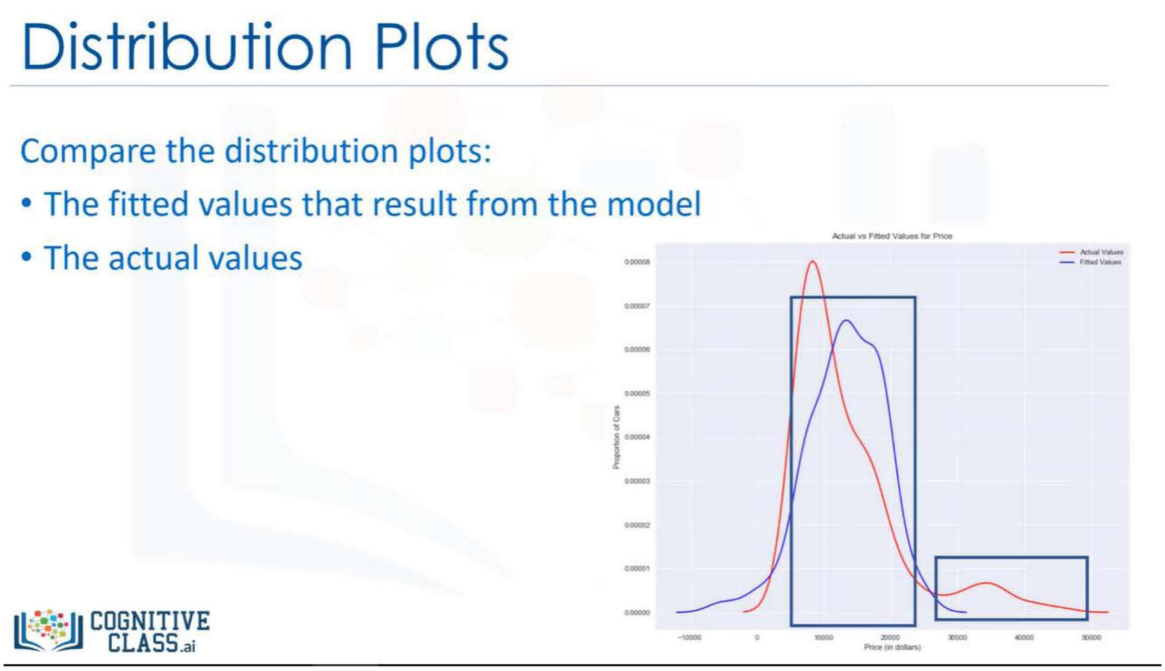
- The fitted values that result from the model
- The actual values



COGNITIVE CLASS.ai

The dependent variable or feature is price.
The fitted values that result from the model are in blue.
The actual values are in red.
We see the predicted values for prices in the range from $40 000 to $50 000 are inaccurate.

The prices in the region form $10 000 to $20 000 are much closer to the target value.



In this example, we use multiple features or independent variables.

Comparing it to the plot on the last slide, we see predicted values are much closer to the target values.
Here is the code to create a Distribution Plot.

```
import seaborn as sns

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

The actual values are used as a parameter.

We want a distribution instead of a histogram so we want the hist parameter set to false.

The color is red.

The label is also included.

The predicted values are included for the second plot; the rest of the parameters are set accordingly.

Polynomial Regression and Pipelines v2

In this video we will cover Polynomial Regression and Pipelines.

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

Let's look into another type of regression model: **the polynomial regression**.

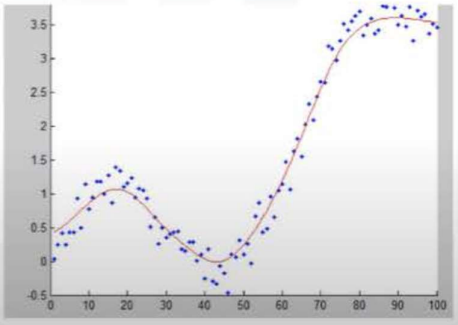
We Transform our data into a polynomial, then use linear regression to fit the parameter.

Then we will discuss pipelines. **Pipelines** are a way to simplify your code.

Polynomial Regressions

- A special case of the general linear regression model
- Useful for describing curvilinear relationships.

Curvilinear relationships:
By squaring or setting higher-order terms of the predictor variables.



Polynomial regression is a special case of the general linear regression.

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.

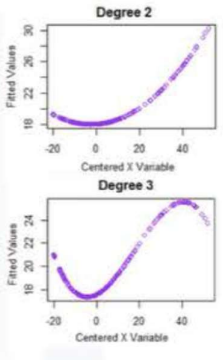
Polynomial Regression

- Quadratic – 2nd order

$$\hat{Y} = b_0 + b_1 x_1 + b_2 (x_1)^2$$

- Cubic – 3rd order

$$\hat{Y} = b_0 + b_1 x_1 + b_2 (x_1)^2 + b_3 (x_1)^3$$



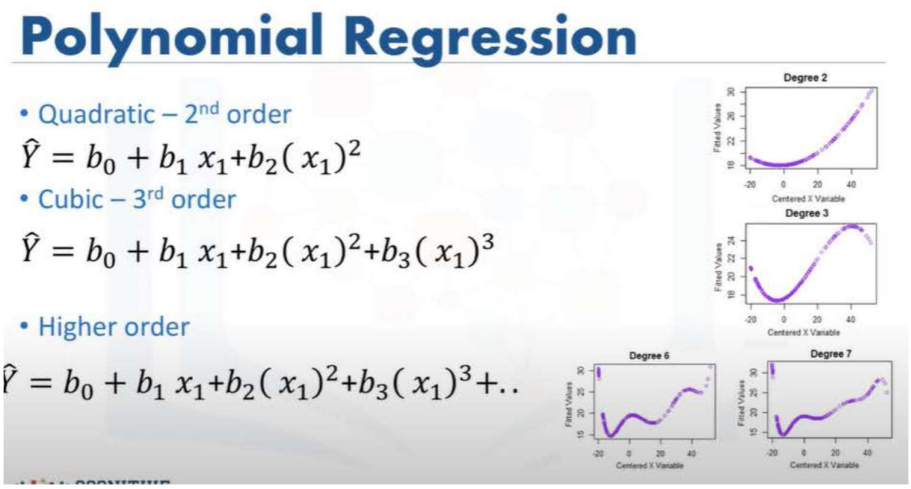
We use a bracket to indicate it is an exponent.

This is a second order Polynomial regression with a figure representing the function.

The model can be cubic, which means that the predictor variable is cubed.
This is a third order Polynomial regression.

We see by examining the figure that the function has more variation.

There also exists higher order polynomial regressions, when a good fit hasn't been achieved by second or third order.

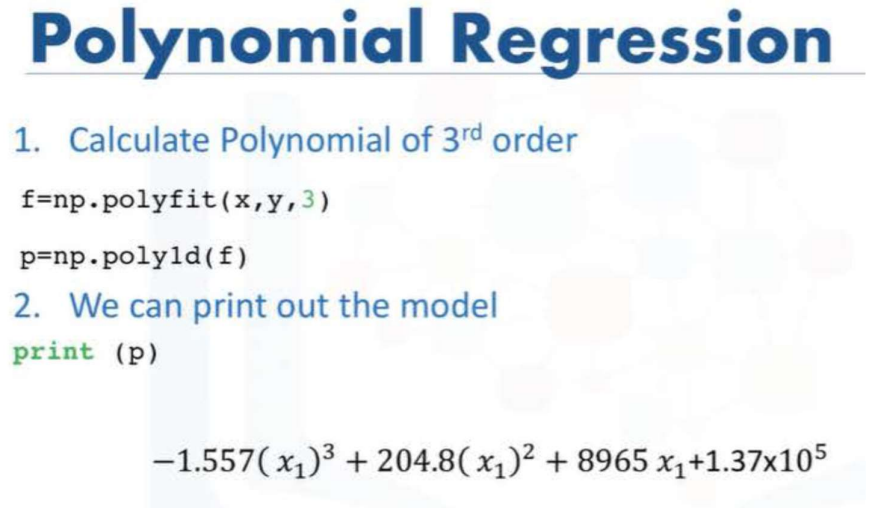


We can see in figures how much the graphs change when we change the order of the polynomial regression.

The degree of the regression makes a big difference and can result in a better fit if you pick the right value.

In all cases, the relationship between the variable and the parameter is always linear.

Let's look at an example from our data where we generate a polynomial regression model.



1. Calculate Polynomial of 3rd order

```
f=np.polyfit(x,y,3)
p=np.poly1d(f)
```

2. We can print out the model

```
print (p)
```

In Python, we do this by using the polyfit() function.

In this example, we develop a third order polynomial regression model base.
We can print out the model.

Symbolic form for the model is given by the following expression (negative 1.557 x1 cubed plus 204.8 x1 squared, plus 8,965 x1 plus 1.37 times 10 to the power of 5).
We can also have multi-dimensional polynomial linear regression.

Polynomial Regression with More the One Dimension

- We can also have multi dimensional polynomial linear regression

$$\hat{Y} = b0 + b1\,X_1 + b2\,X_2 + b3\,X_1\,X_2 + b4(X_1)^2 + b5(X_2)^2 + ..$$

The expression can get complicated; here are just some of the terms for a two-dimensional second order polynomial.

Numpy's "polyfit" function cannot perform this type of regression.

We use the "preprocessing" library in sci-kit-learn, to create a polynomial feature object.

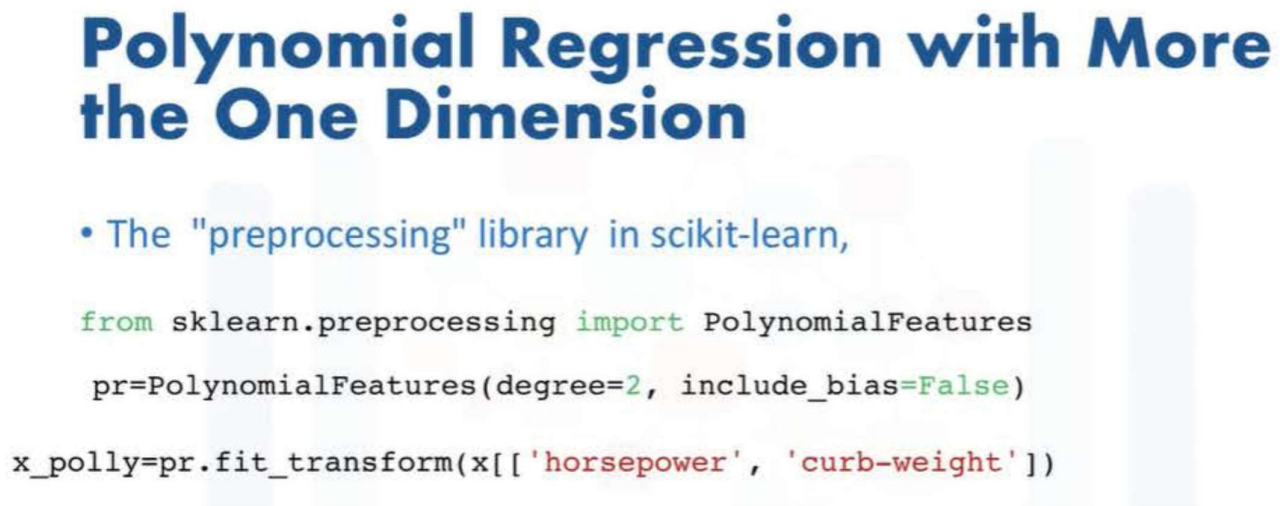
Polynomial Regression with More the One Dimension

- The "preprocessing" library in scikit-learn,

```
from sklearn.preprocessing import PolynomialFeatures
pr=PolynomialFeatures(degree=2, include_bias=False)
```

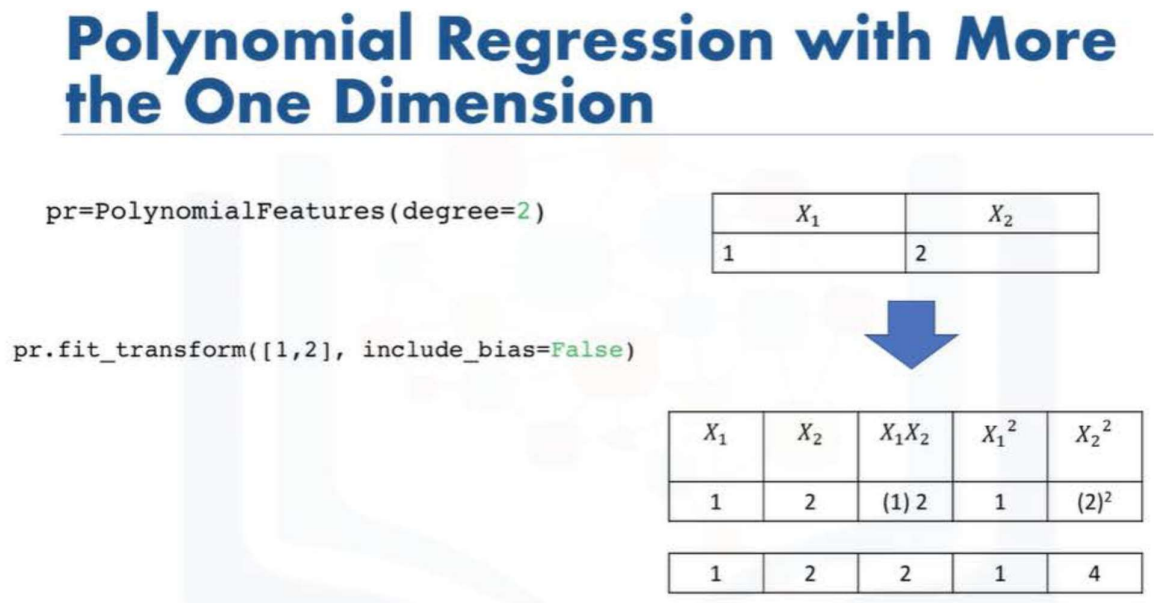
The constructor takes the degree of the polynomial as a parameter.

Then we transform the features into a polynomial feature with the “fit_transform” method.



from sklearn.preprocessing import PolynomialFeatures
pr=Polynomia1Features(degree=2, include_bias=False)
x_polly=pr.fit_transform(x[['horsepower', 'curb-weight'])

Let's do a more intuitive example. Consider the features shown here.

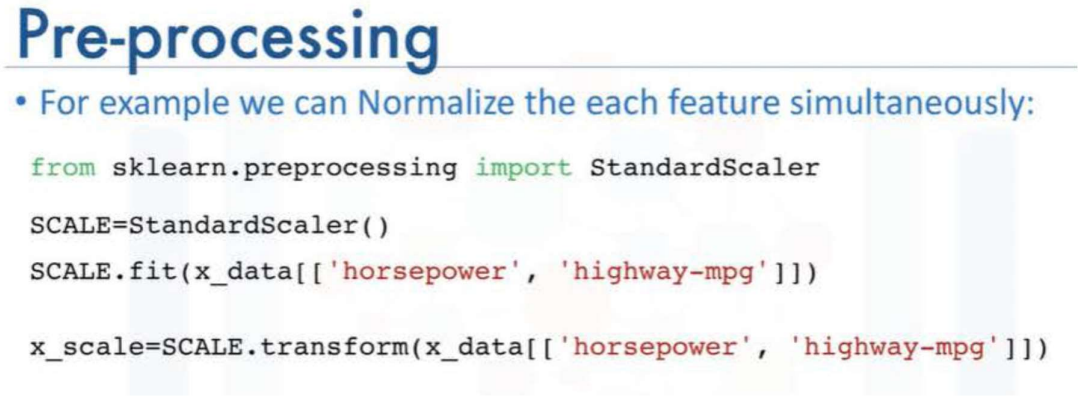


Applying the method, we transform the data We now have a new set of features that are a transformed version of our original features.

As the dimension of the data gets larger we may want to normalize multiple features in scikit-learn, instead, we can use the preprocessing module to simplify many tasks.

For example, we can Standardize each feature simultaneously.

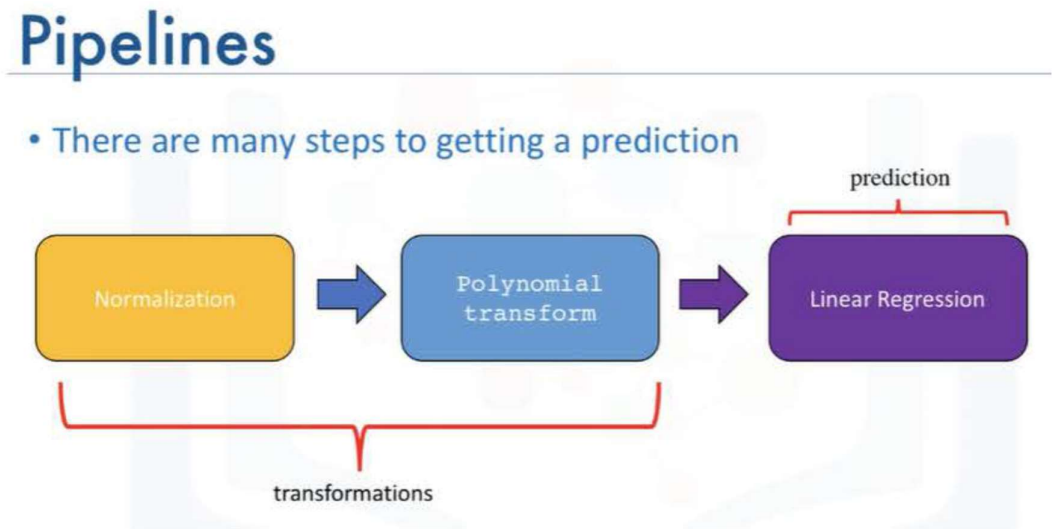
We import “StandardScaler” We train the object, fit the scale object;
then transform the data into a new dataframe on array “x_scale”.
There are more normalization methods available in the preprocessing library, as well as other transformations.



PipeLines

We can simplify our code by using a pipeline library.

There are many steps to getting a prediction, for example, Normalization, Polynomial transform, and Linear regression.



We simplify the process using a pipeline.

Pipelines sequentially perform a series of transformation.

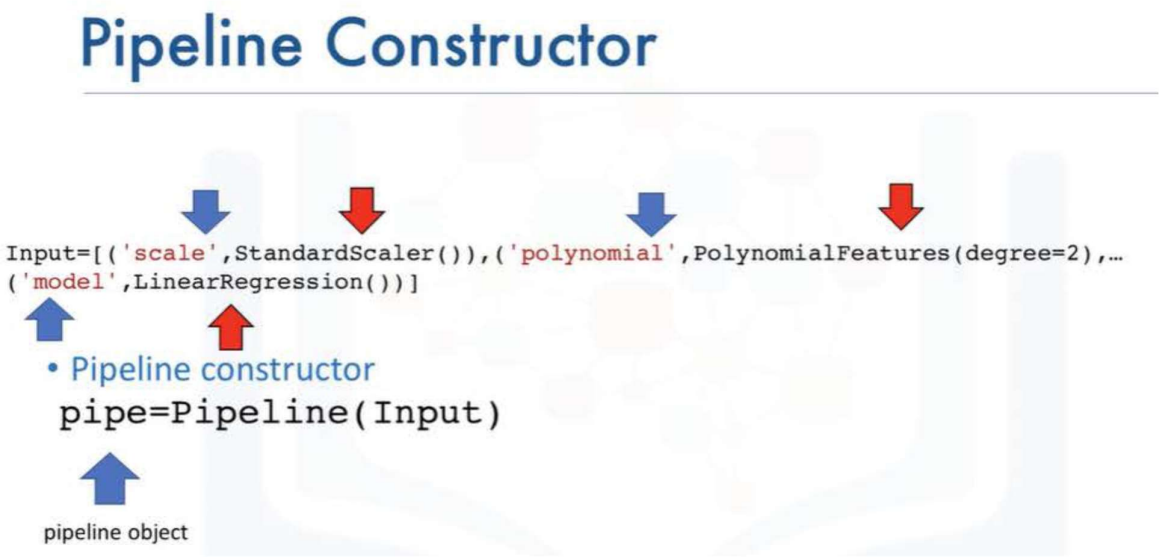
The last step carries out a prediction.

First we import all the modules we need.
Then we import the library Pipeline.

from sklearn.preprocessing import PolynomialFeatures

from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline ..Jnport Pipeline

We create a list of tuples, the first element in the tuple contains the name of the estimator: model.



Input= [('scale' , StandardScaler ( )) , ('polynomial' , PolynomialFeatures (degree=2) , ... ('model' ,LinearRegression())]

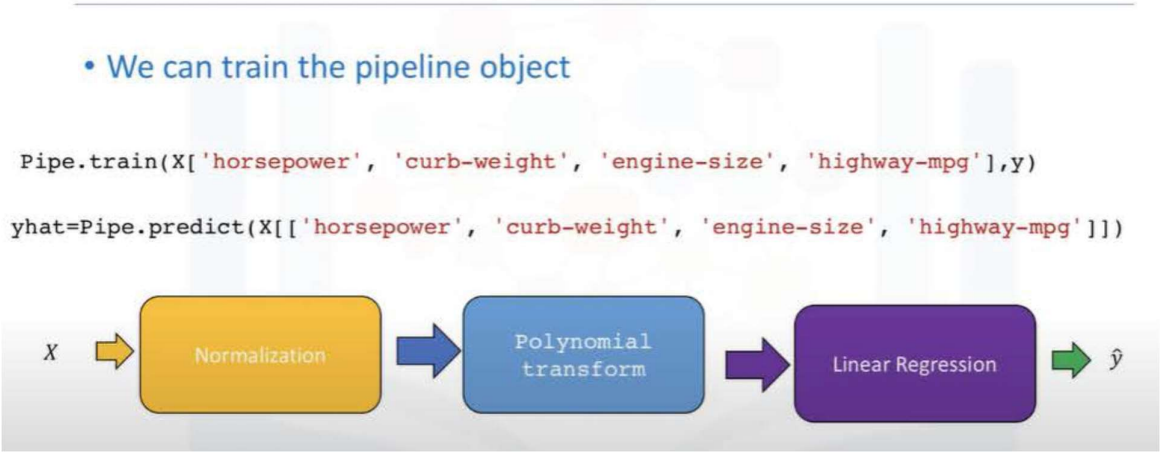
pipe=Pipeline(Input)

The second element contain model constructor.
We input the list in the pipeline constructor.
We now have a pipeline object.

We can train the pipeline by applying the train method to the Pipeline object.

Pipe.train(X('horsepower', 'curb-weight', 'engine-size', 'highway-mpg'J,y)

yhat=Pipe.predict(X(['horsepower', 'curb-weight', 'engine-size', 'highway-mpg')])



We can also produce a prediction as well.

The method normalizes the data, performs a polynomial transform, then outputs a prediction.

Measures For In-Sample Evaluation

Now that we've seen how we can evaluate a model by using visualization, we want to numerically evaluate our models.

Let's look at some of the measures that we use for in-sample evaluation.

These measures are a way to numerically determine how good the model fits on our data.

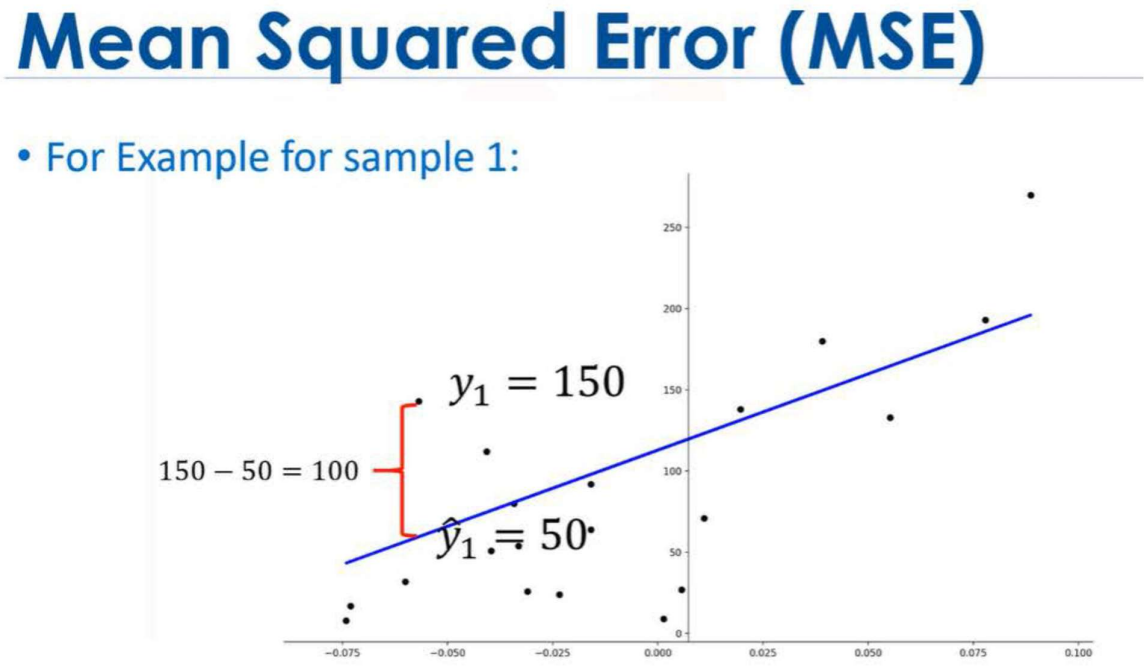
Two important measures that we often use to determine the fit of a model are:

- Mean Square

- Error (MSE), and R-squared.

To measure the MSE, we find the difference between the actual value y and the predicted value yhat then square it.

In this case, the actual value is 150; the predicted value is 50. Subtracting these points we get 100.

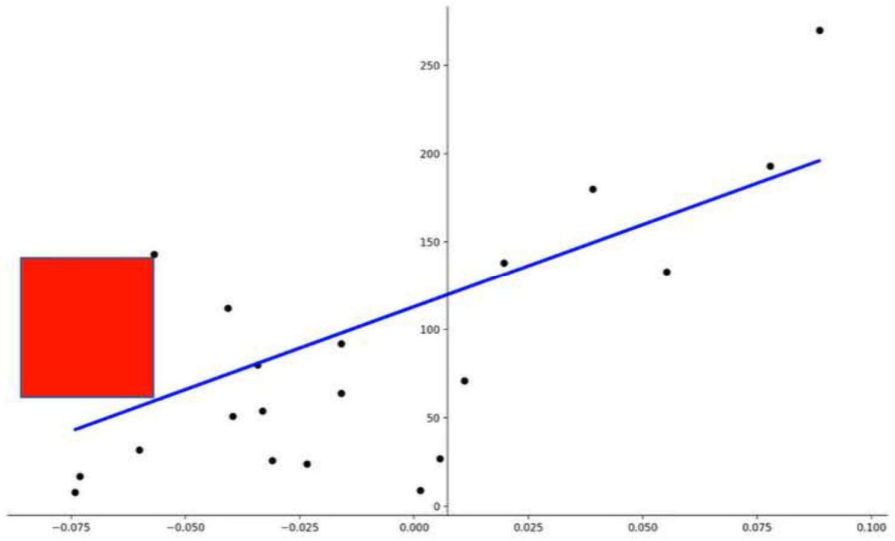


We then square the number.

Mean Squared Error (MSE)

- To make all the values positive we square it

$(\ 50)^2$

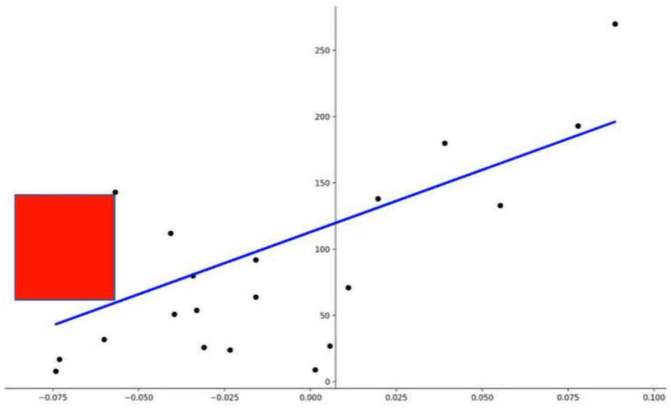


We then take the Mean or average of all the errors by adding then all together and dividing by the number of samples.

Mean Squared Error (MSE)

- To make all the values positive we square it

$(\ 50)^2$

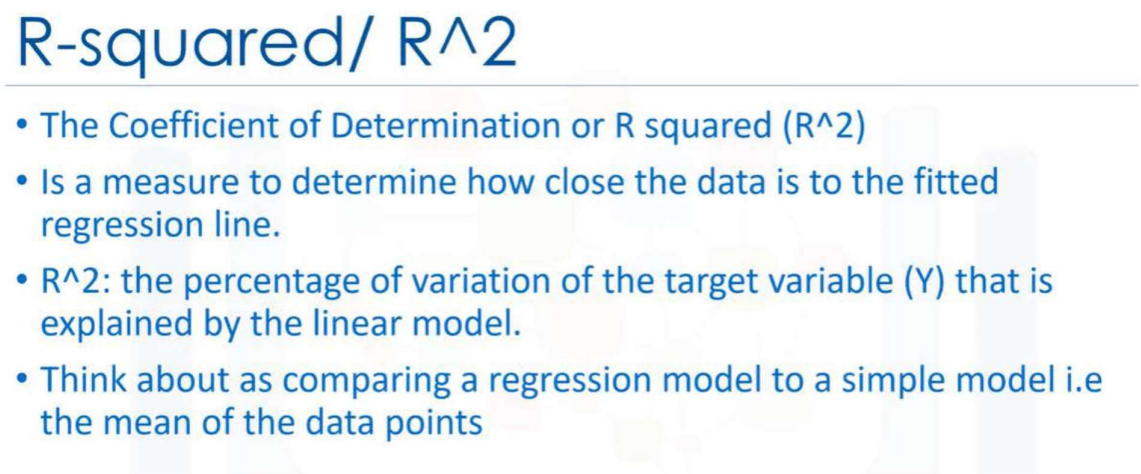


To find the MSE in Python, we can import the “mean_Squared_error()” from “scikit-learn.metrics”.

```
from sklearn.metrics import mean_squared_error

mean_squared_error(df['price'],Y_predict_simple fit)

3163502.944639888
```

The “mean_Squared_error()” function gets two inputs: the actual value of target variable and the predicted value of target variable.

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?

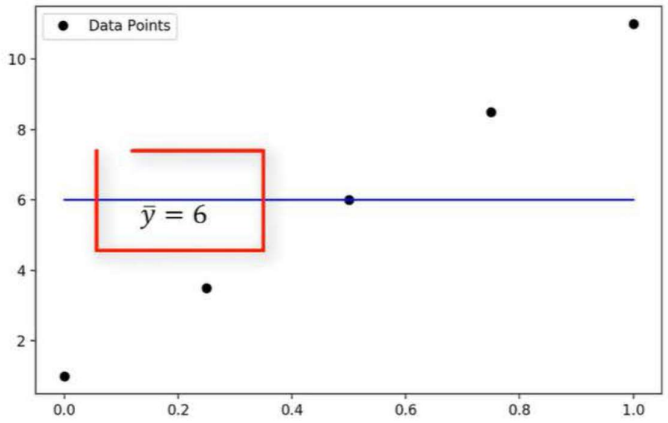


Think about it as comparing a regression model to a simple model, i.e., the mean of the data points. If the variable x is a good predictor our model should perform much better than [with] just the mean.

In this example the average of the data points y | is 6.

Coefficient of Determination (R^2)

- In this example the average of the data points $\bar{y}$ is 6

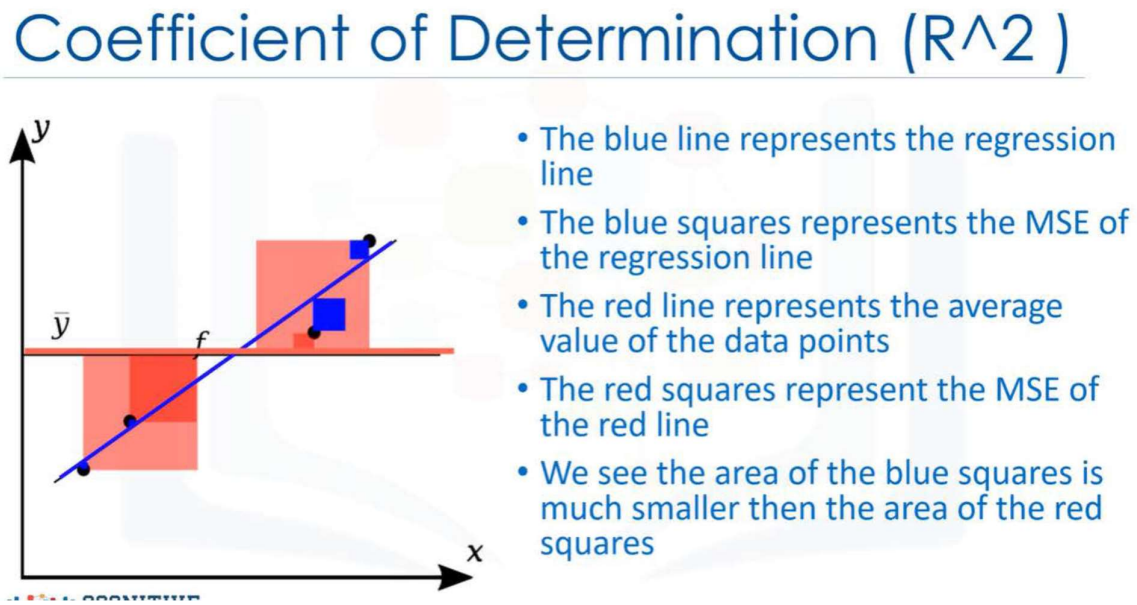


Coefficient of Determination or R^2 is 1 minus the ratio of the MSE of the regression line divided by the MSE of the average of the data points. For the most part, it takes values between 0 and 1.

Coefficient of Determination (R^2)

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

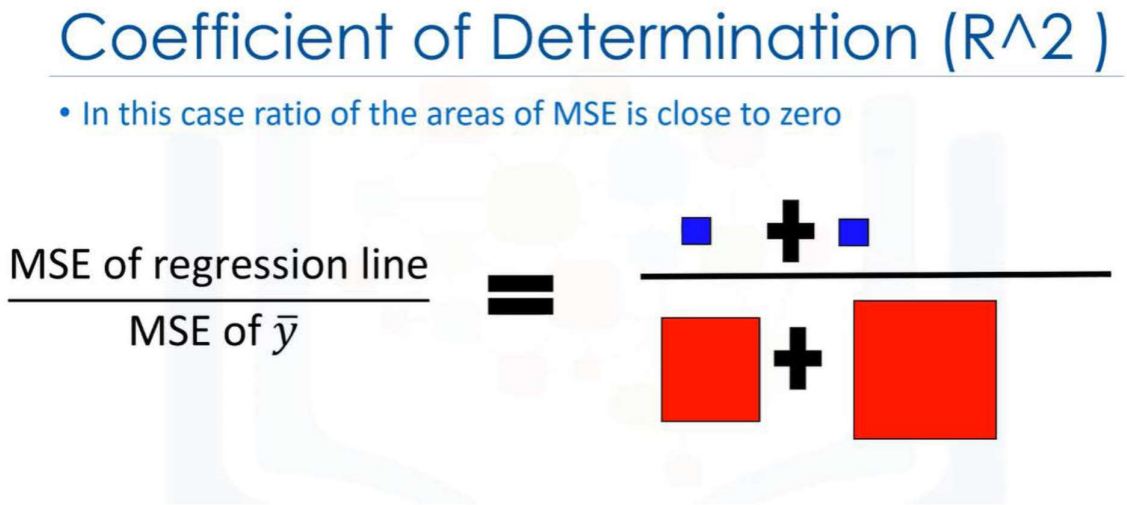
Let's look at a case where the line provides a relatively good fit.



The blue line represents the regression line.
The blue squares represent 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, because the line is a good fit, the Mean squared error is small, therefore the numerator is small.



The Mean squared error of the line is relatively large, as such the numerator is large.

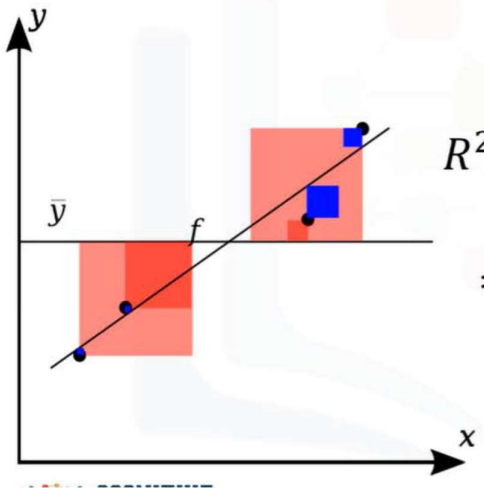
Coefficient of Determination (R^2)

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

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

A small number divided by a larger number is an even smaller number. Taken to an extreme this value tends to zero.

Coefficient of Determination (R^2)



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

$$= (1 - 0)$$

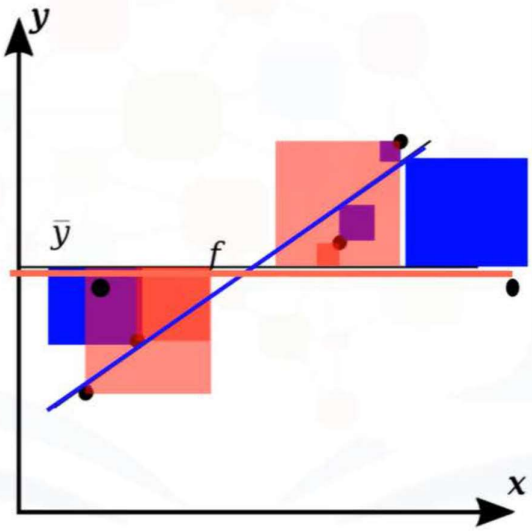
$$= (1 - 0)$$

$$= 1$$

If we Plug in this value from the previous slide for R^2, we get a value near one, this means the line is a good fit for the data.

Here is an example of a line that does not fit the data well.

Coefficient of Determination (R^2)



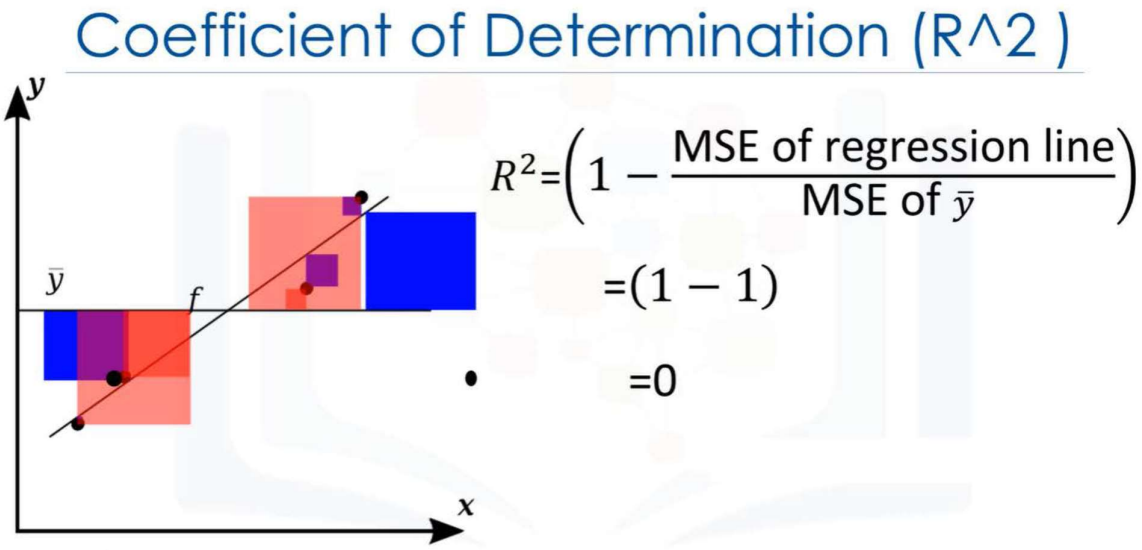
If we just examine the area of the red squares compared to the blue squares, we see the area is almost identical.

The ratio of the areas is close to one.

Coefficient of Determination (R^2)

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

In this case the R^2 is near zero.

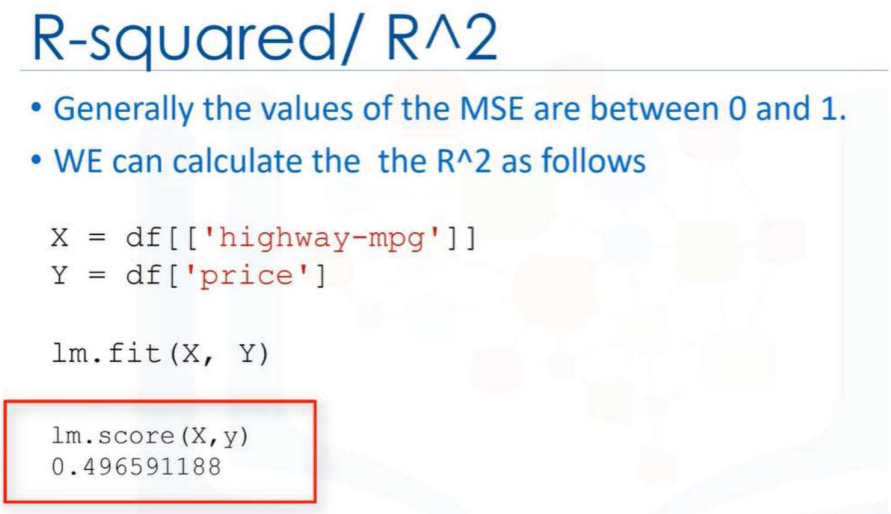


This line performs about the same as just using the average of the data points, therefore, this line did not perform well.

We find the R-squared value in Python by using the score() method, in the linear regression object.

X - df[['highway-mpg']]
Y - df ['price']
lm.fit(X, Y)

lm.score(X,y)
0.496591188



From the value that we get from this example, we can say that approximately 49.695% of the variation of price is explained by this simple linear model.

Your R^2 value is usually between 0 and 1, if your R^2 is negative it can be due to overfitting that we will discuss in the next module.

Prediction And Decision Making

In this video, our final topic will be on 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 between different models

Let's look at an example of prediction; if you recall we train the model using the fit method.

First we train the model

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

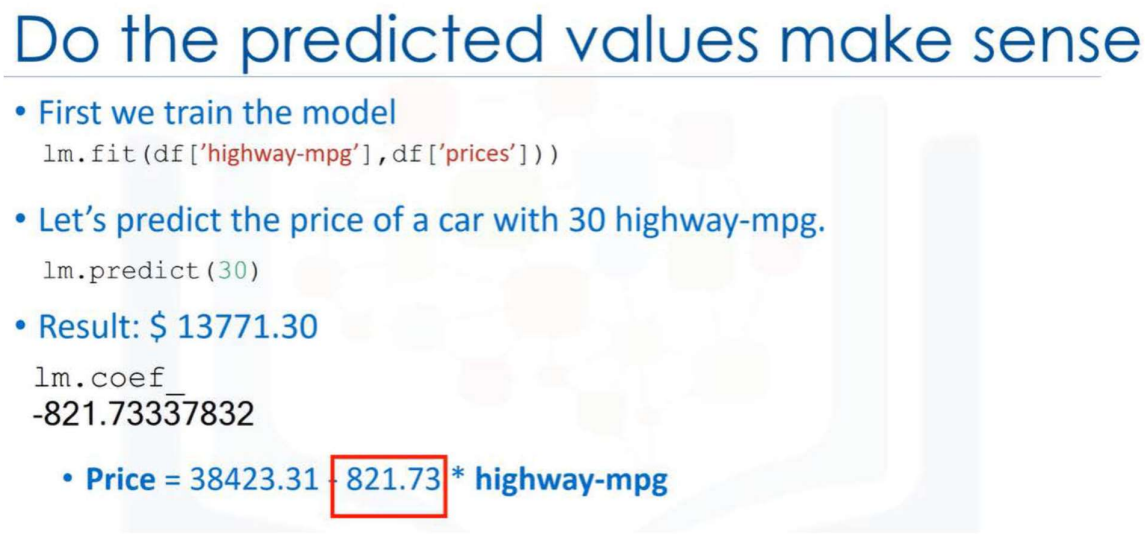
Let's predict the price of a car with 30 highway-mpg

lm.predict (30)

Result: $ 13771.30

lrn.coef

-821.73337832



Now we want to find out what the price would be for a car that has a highway-mpg of 30.

Plugging this value into the predict() method, gives us a resulting price of $13,771.30.

This seems to make sense, for example, the value is not negative, extremely high or extremely low.

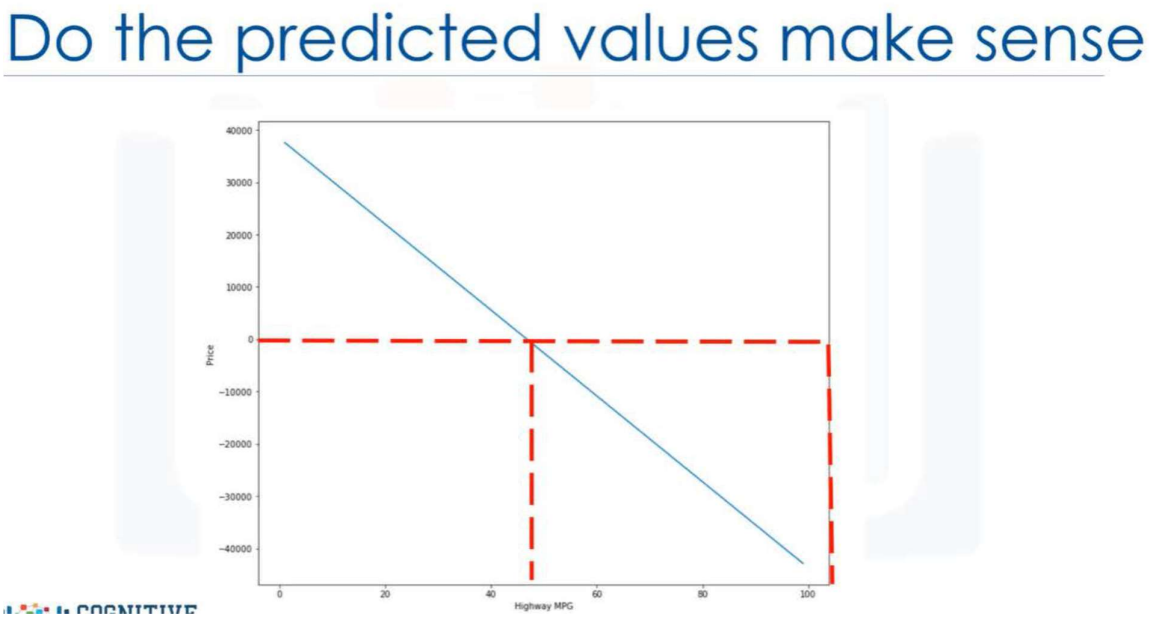
We can look at the coefficients by examining the "coef_" attribute.

If you recall.

the expression for the simple linear model that predicts price from highway-mpg, this value corresponds to the multiple of the highway-mpg feature.

As such, an increase of one unit in highway-mpg, the value of the car decreases approximately 821 dollars; this value also seems reasonable.

Sometimes your model will produce values that don't make sense, for example, if we plot the model out for highway-mpg, in the ranges of 0 to 100, we get negative values for the price.



This could be because the values in that range are not realistic, the linear assumption is incorrect, or we don't have data for cars in that range.

In this case, it is unlikely that a car will have fuel mileage in that range, so our model seems valid.

To generate a sequence of values in a specified range, import numpy, then use the numpy "arrange" function to generate the sequence.

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(l,101,1) .reshape(-1,1)
```

Do the predicted values make sense

- 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)
```

The sequence starts at one and increments by one till we reach 100.
The first parameter is the starting point of the sequence.
The second parameter is the end point plus one of the sequence.
The final parameter is the step size between elements in the sequence, in this case, it's one, so we increment the sequence one step at a time, from 1 to 2, and so on.

We can use the output to predict new values; the output is a numpy array.
Many of the values are negative.

yhat=lm.predict(new_input)

Do the predicted values make sense

- We can predict new values

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

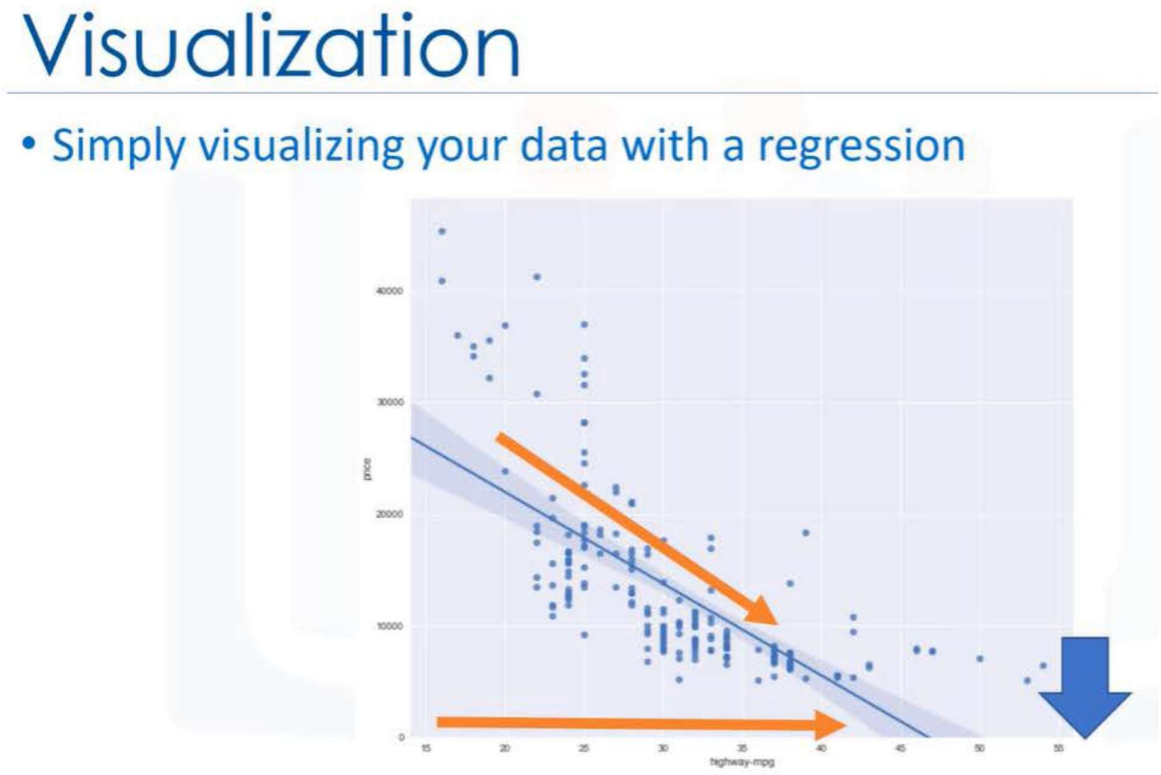
```
array([ 37601.57247984,  36779.83910151,  35958.10572319,  35136.37234487,
        34314.63896655,  33492.90558823,  32671.1722099 ,  31849.43883158,
        31027.70545326,  30205.97207494,  29384.23869662,  28562.50531829,
        27740.77193997,  26919.03856165,  26097.30518333,  25275.57180501,
        24453.83842668,  23632.10504836,  22810.37167004,  21988.63829172,
        21166.9049134 ,  20345.17153508,  19523.43815675,  18701.70477843,
        17879.97140011,  17058.23802179,  16236.50464347,  15414.77126514,
        14593.03788682,  13771.3045085 ,  12949.57113018,  12127.83775186,
        11306.10437353,  10484.37099521,   9662.63761689,   8840.90423857,
         8019.17086025,   7197.43748192,   6375.7041036 ,   5553.97072528,
         4732.23734696,   3910.50396864,   3088.77059031,   2267.03721199,
        -1841.62967962,  -2663.36305794,  -3485.09643626,  -4306.82981458,
        -5128.5631929 ,  -5950.29657123,  -6772.02994955,  -7593.76332787,
        -8415.49670619,  -9237.23008451, -10058.96346284, -10880.69684116,
       -11702.43021948, -12524.1635978 , -13345.89697612, -14167.63035445,
       -14989.36373277, -15811.09711109, -16632.83048941, -17454.56386773,
       -18276.29724606, -19098.03062438, -19919.7640027 , -20741.49738102,
       -21563.23075934, -22384.96413767, -23206.69751599, -24028.43089431,
       -24850.16427263, -25671.89765095, -26493.63102927, -27315.3644076 ,
       -28137.09778592, -28958.83116424, -29780.56454256, -30602.29792088,
       -31424.03129921, -32245.76467753, -33067.49805585, -33889.23143417,
       -34710.96481249, -35532.69819082, -36354.43156914, -37176.16494746,
       -37997.89832578, -38819.6317041 , -39641.36508243, -40463.09846075,
       -41284.83183907, -42106.56521739, -42928.29859571])
```

COGNITIVE

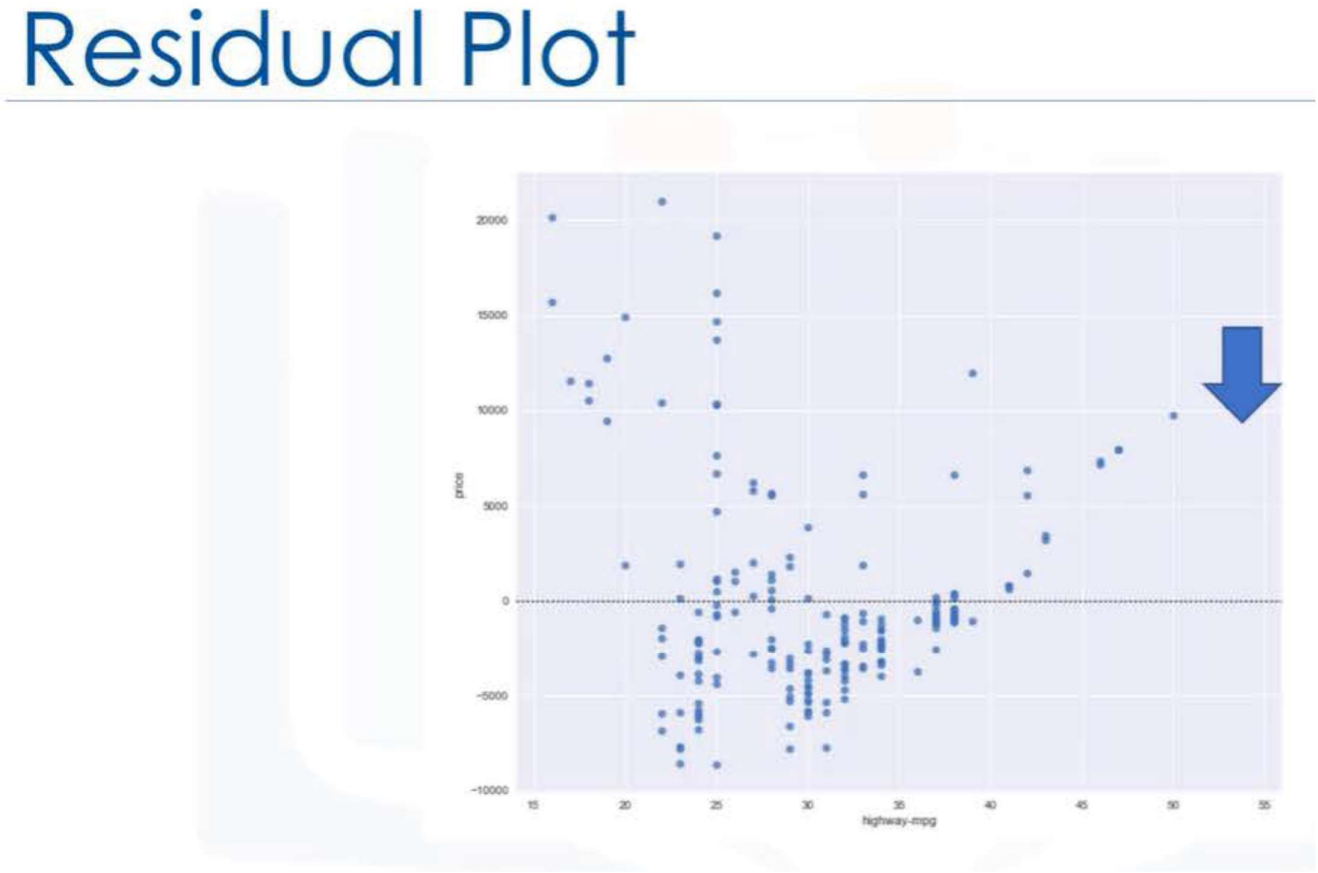
Using a regression plot to visualize your data is the first method you should try. See the labs for examples of how to plot polynomial regression.

For this example, the effect of the independent variable is evident in this case. The data trends down as the dependent variable increases.

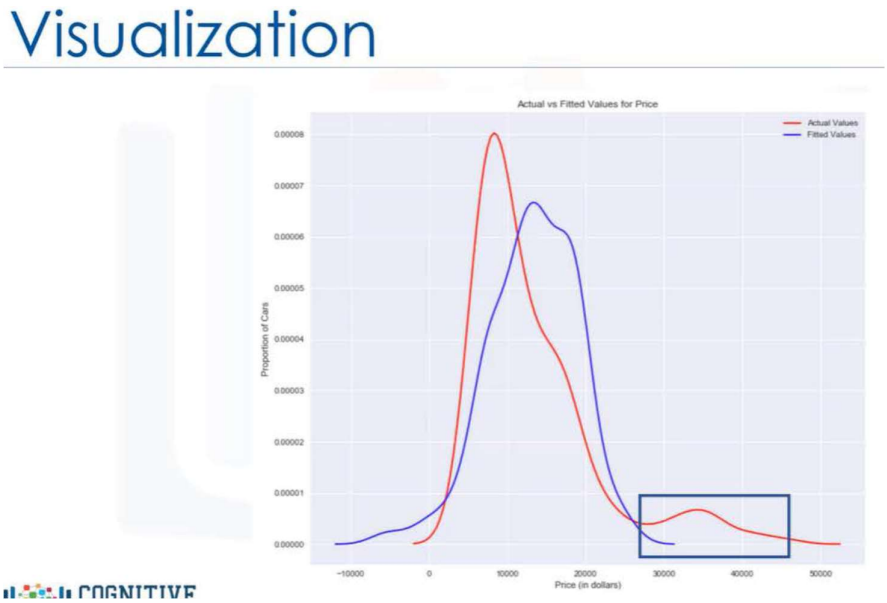
The plot also shows some non-linear behavior.



Examining the Residual Plot We see in this case the Residuals have a curvature suggesting non-linear behavior.



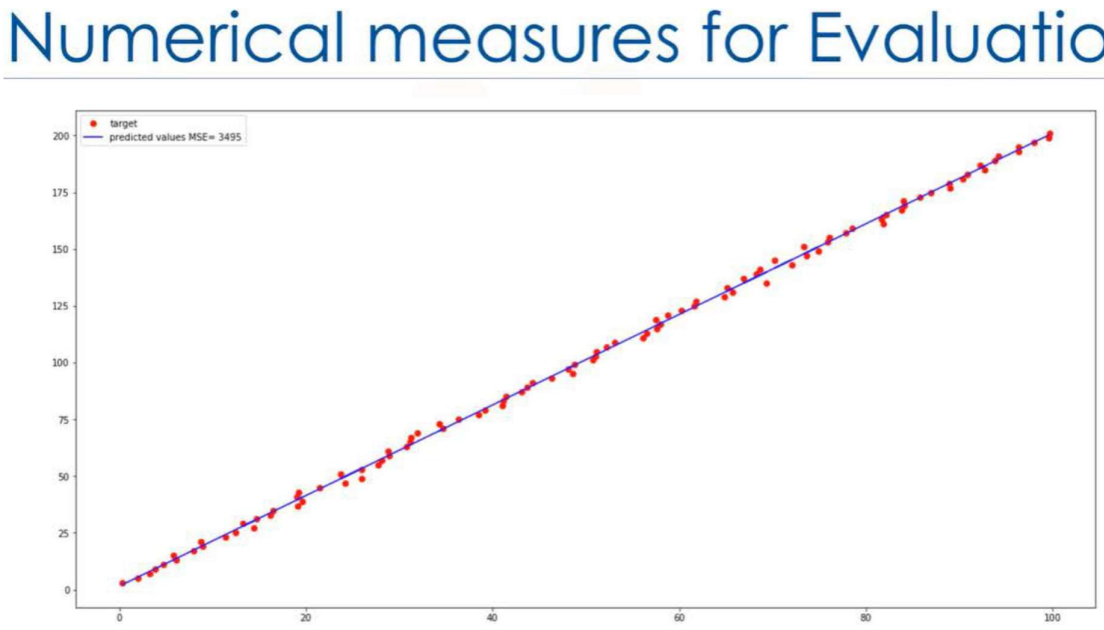
A distribution plot, is a good method for Multiple Linear Regression.



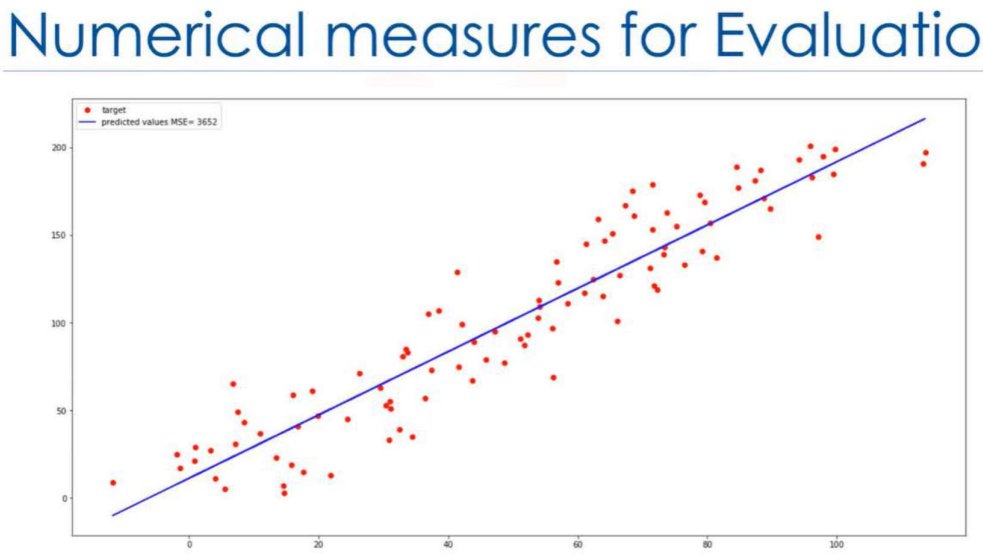
For example: We see the predicted values for prices in the range from $3,0000 to $50,000 are inaccurate This suggests a non-linear model may be more suitable or we need more data in this range.

The mean square error is perhaps the most intuitive Numerical measure for determining if a model is good or not; let's see how different measures of Mean square error impact the model.

The figure shows an example of a mean square error of 3,495.

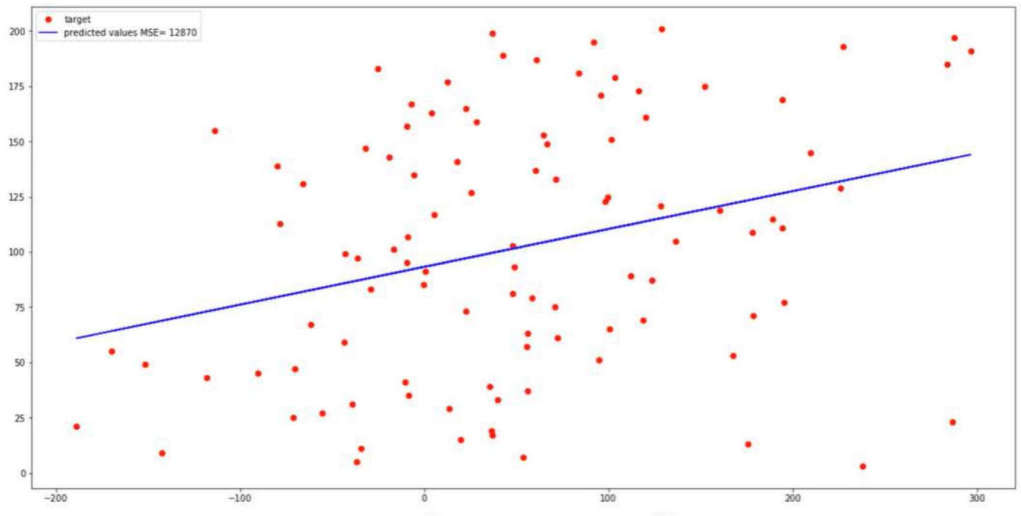


This example has a mean square error of 3,652.



The final plot has a mean square error of 12,870.

Numerical measures for Evaluation

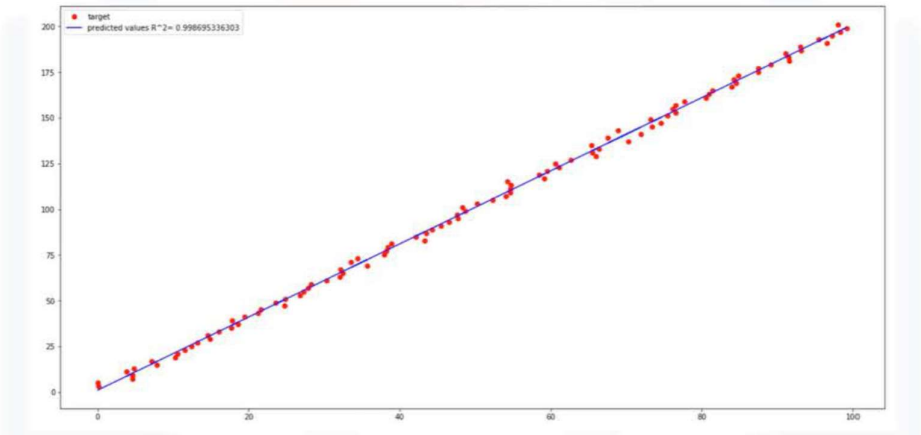


As the square error increases, the targets get further from the predicted points.

As we discussed, R^2 (R-squared) is another popular method to evaluate your model.

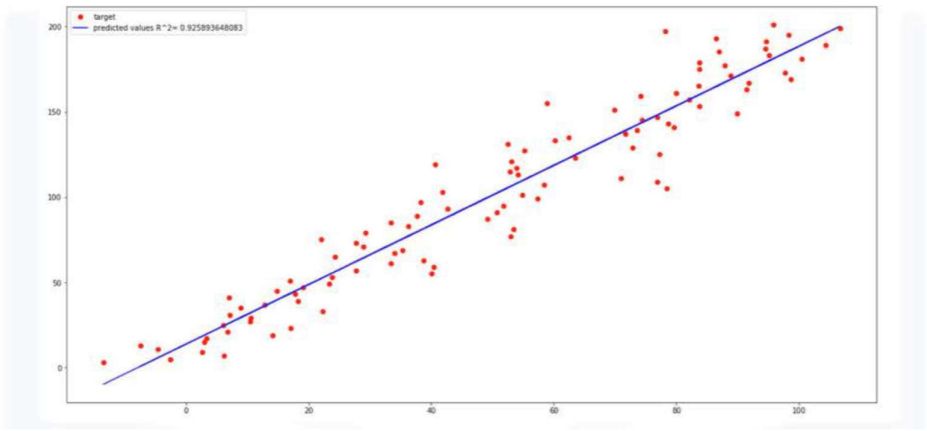
In this plot, we see the target points in red and the predicted line in blue, an R^2 of 0.9986; the model appears to be a good fit.

Numerical measures for Evaluation



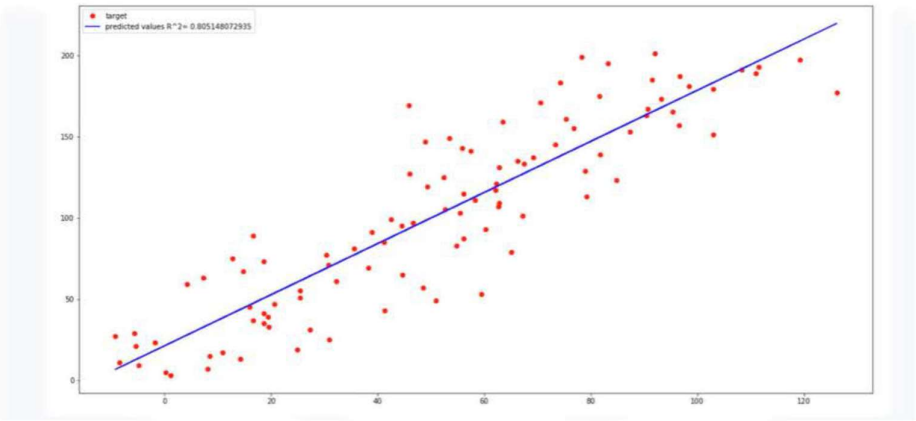
This model has a R^2 of 0.9226; there still is a strong linear relationship.

Numerical measures for Evaluation



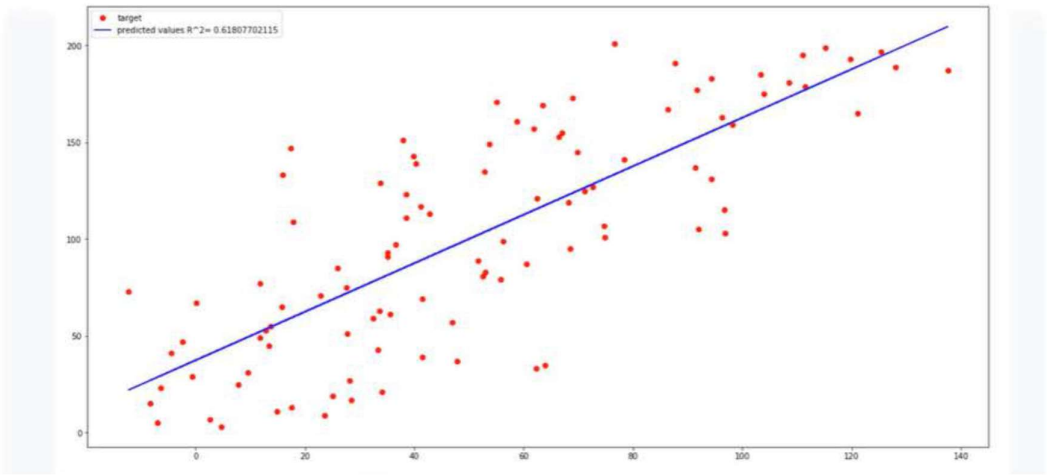
An R^2 of 0806 the data is a lot more messy but the linear relation is evident.

Numerical measures for Evaluation

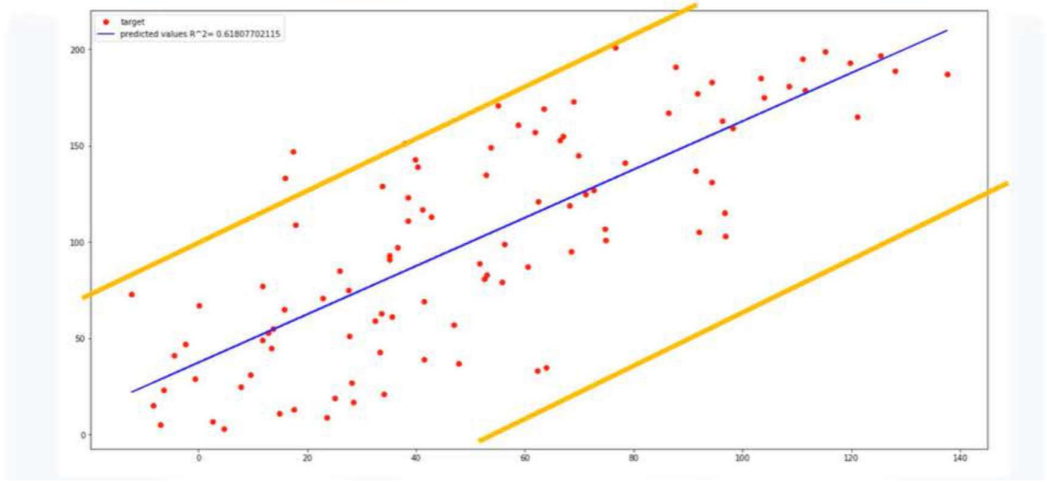


An R^2 of 0.61 the linear function is harder to see, but, on closer inspection, we see the data is increasing with the independent variable.

Numerical measures for Evaluation



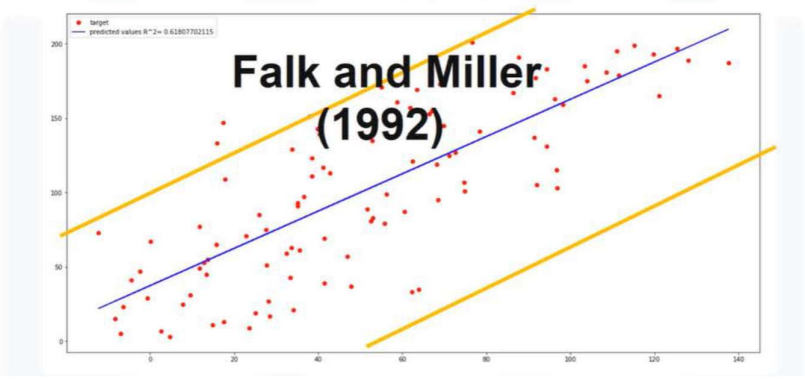
Numerical measures for Evaluation



An acceptable value for R^2 depends on what field you're studding.

Some authors suggest a value should be equal to or greater than 0.10.

Numerical measures for Evaluation



Comparing MLR and SLR:

1. Is a lower MSE always implying a better fit?

Not necessarily.

2. MSE for an MLR model will be smaller than the MSE for an SLR model, since the errors of the data will decrease when more variables are included in the model.

3. Polynomial regression will also have a smaller MSE then regular regression.

4. A similar inverse relationship holds for R^2.

In the next section we'll look at better ways to evaluate the model.

Module 5 - Model Evaluation

Model Evaluation And Refinement

In this module, we are going to talk about model evaluation.
In this lesson you will learn about:

- Model Evaluation
- Over-fitting, Under-fitting and Model Selection
- Ridge Regression
- Grid Search

And answer the Question:
How can you be certain your model works in the real world and performs optimally.

Model Evaluation

Model Evaluation tells us how our model preforms in the real world.

In the previous module, we talked about the in-sample evaluation.

- In-sample evaluation tells us how well our model fits the data already given to train it.

Problem ?

- It does not give us an estimate of how well the trained model can predict new data.

Solution ?

- The solution is 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 evaluation or test set

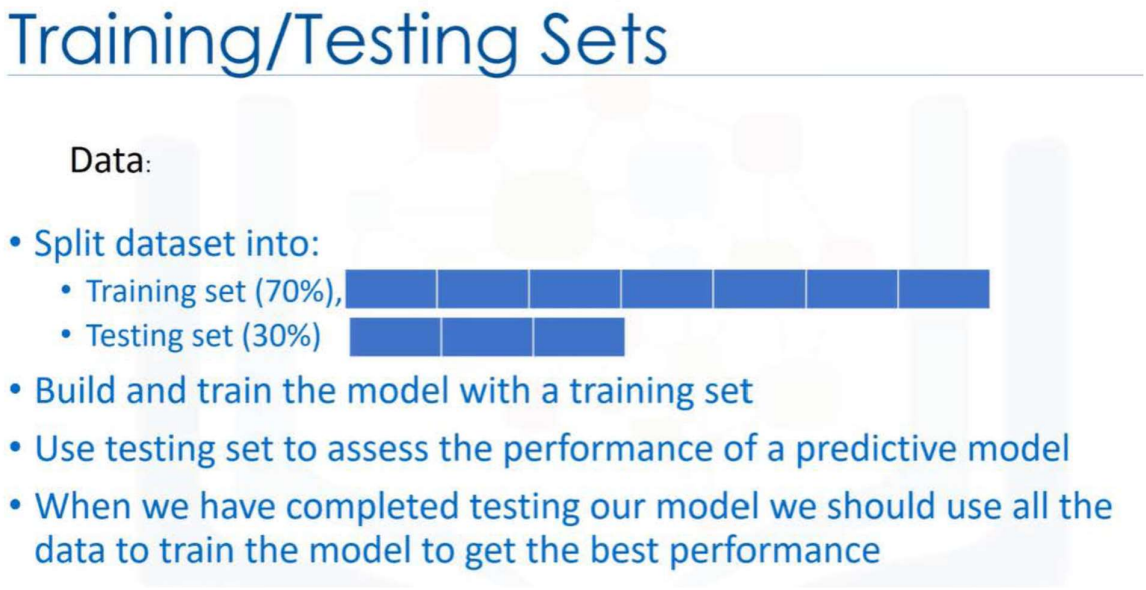
This data is then used to approximate how the model preforms in the real world.

Separating data into training and testing sets is an important part of model evaluation.

We use the test data to get an idea how our model will perform in the real world.

When we split a data set, usually the larger portion of data is used for training and a smaller part is used for testing.

For example we can use 70% of the data for training; we then use 30% for testing.
We use a training set to build a model and discover predictive relationships.
We then 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.



A popular function in the sci-kit learn package for splitting datasets is the "train test split" function.

This function randomly splits a dataset into training and testing subsets.

From the example code snippet, this method is imported from "sklearn.cross validation."

The input parameters y_data is the target variable (in the car appraisal example, it would be the price), and "x_data", the list of predictor variables. In this case, it would be all the other variables in the car data set that we are using to try to predict the price.

Function train_test_split()

- 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_state=0)
```

- **x_data**: features or independent variables
- **y_data**: dataset target: df['price']

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_state=O)

The output is an array: "x_train" and "y_train", the subsets for training; "x_test" and "y_test", the subsets for testing.

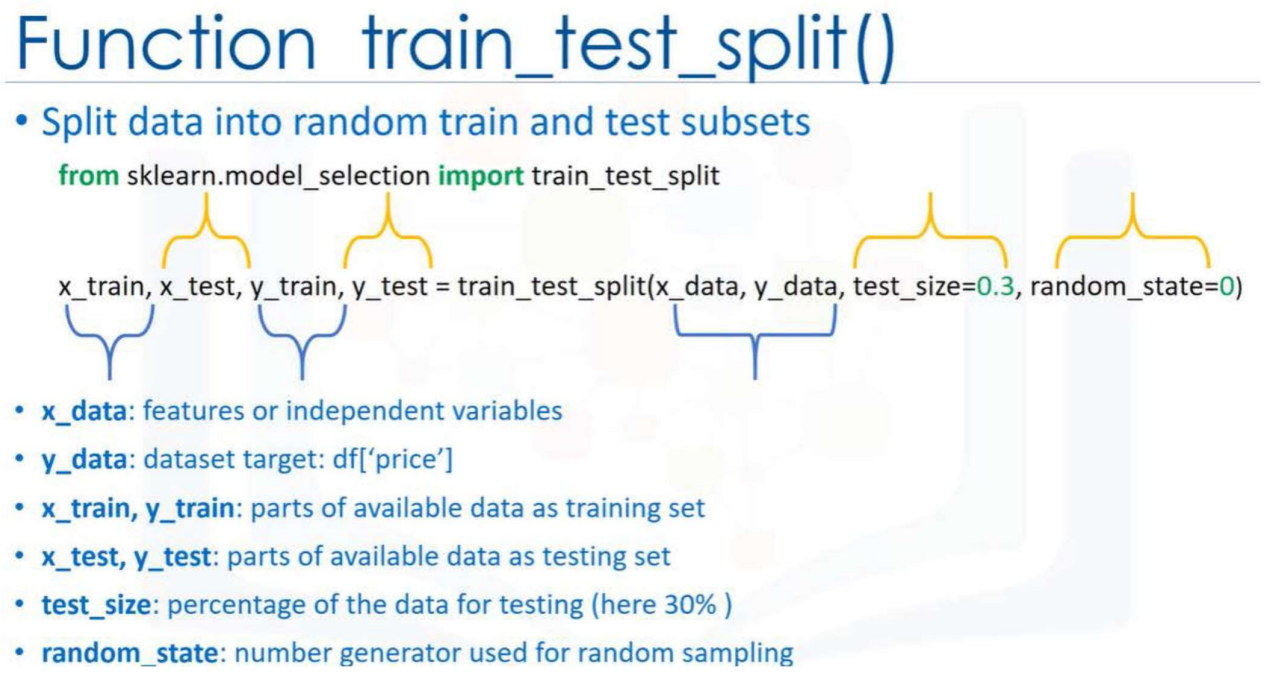
Function train_test_split()

- 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_state=0)
```

- **x_data**: features or independent variables
- **y_data**: dataset target: df['price']
- **x_train, y_train**: parts of available data as training set
- **x_test, y_test**: parts of available data as testing set

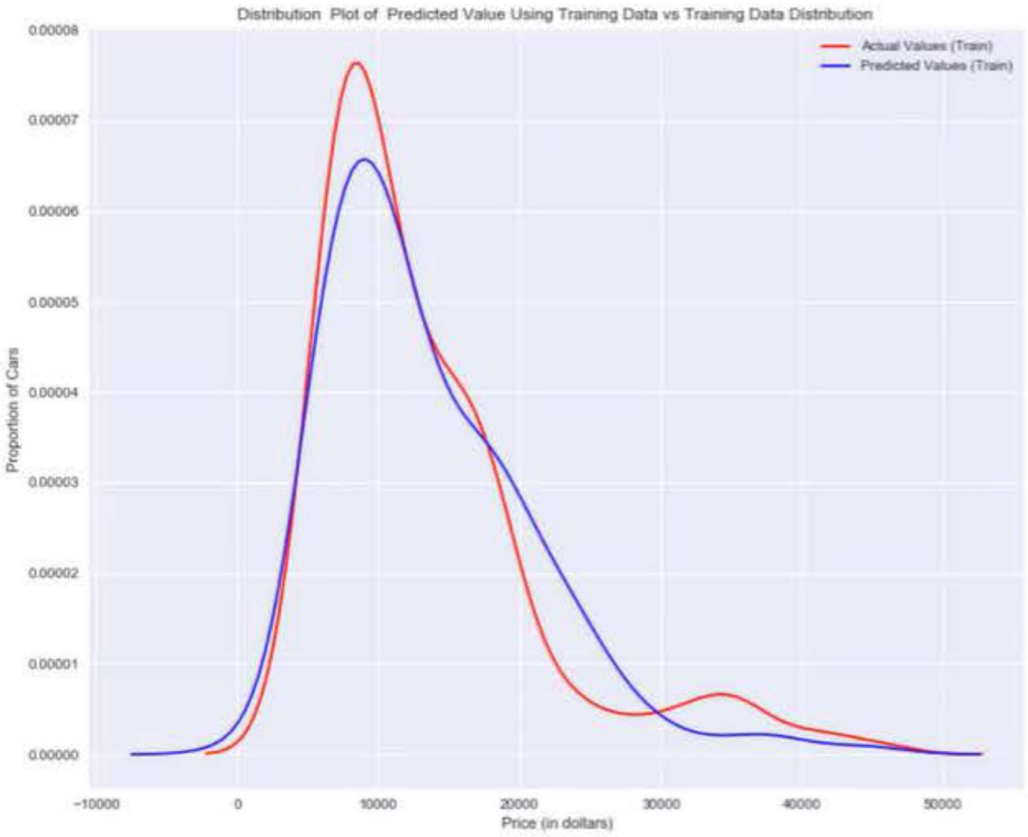
In this case, the "test size" percentage of the data for the testing set. Here it is 30%. The random state is a random seed for random dataset splitting.



- 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.

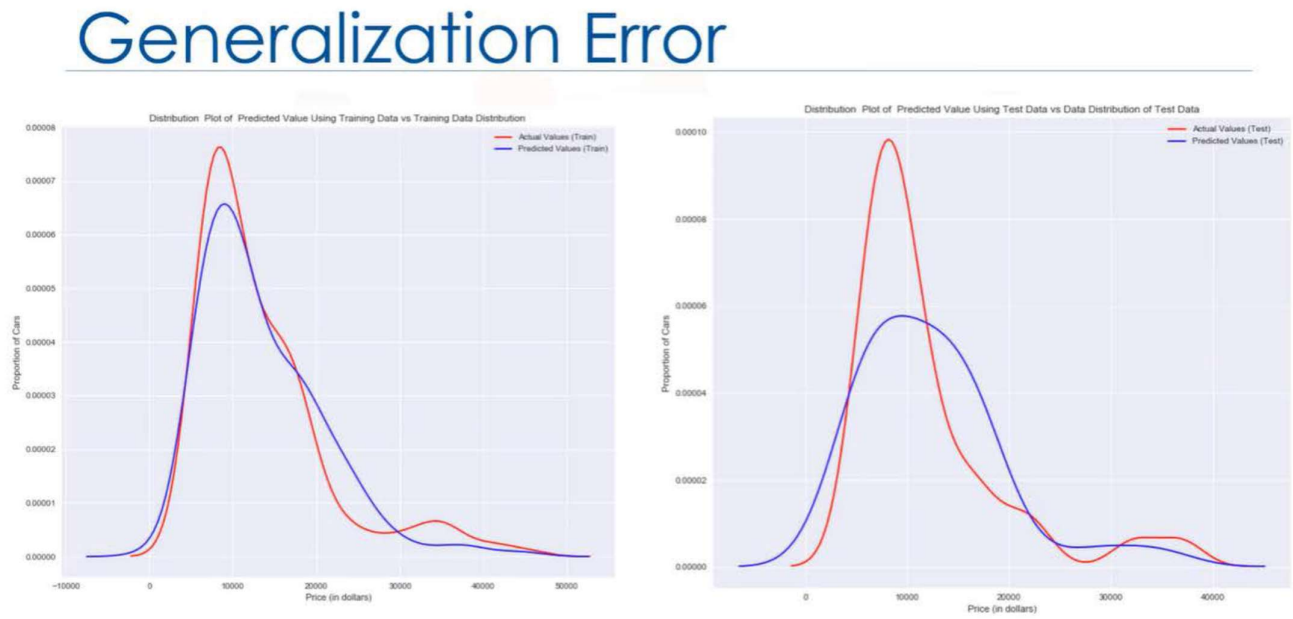
This figure shows the distribution of the actual values in red compared to the predicted values from a linear regression in blue.

Generalization Error



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.

Using a lot of data for training gives us an accurate means of determining how our model will perform in the real world, but the precision of the performance will be low.

Let's clarify this with an example.

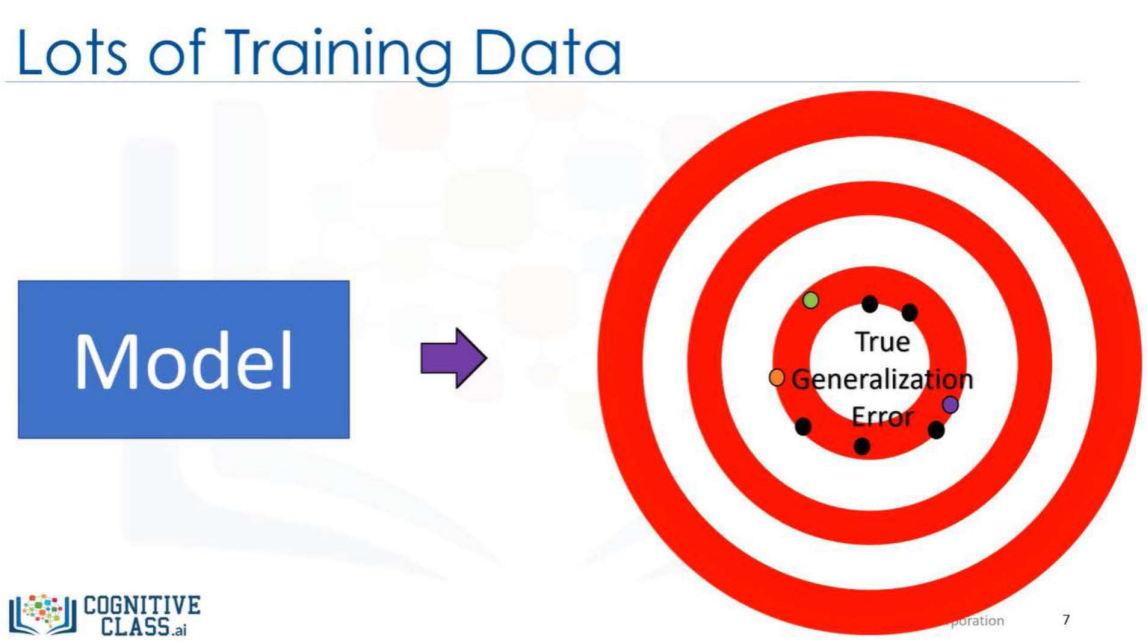
The center of this bullseye represents the correct generalization error; let's say we take a random sample of the data using 90% of the data for training and 10% for testing.

The first time we experiment we get a good estimate of the training data.

If we experiment again, training the model with a different combination of samples, we also get a good result, but the results will be different relative to the first time we run the experiment.

Repeating the experiment again with a different combination of training and testing samples, the results are relatively close to the Generalization error, but distinct from each other.

Repeating the process, we get good approximation of the generalization error, but the precision is poor .i.e., all the results are extremely different from one another.



If we use fewer data points to train the model and more to test the model, the accuracy of the generalization performance will be less, but the model will have good precision.

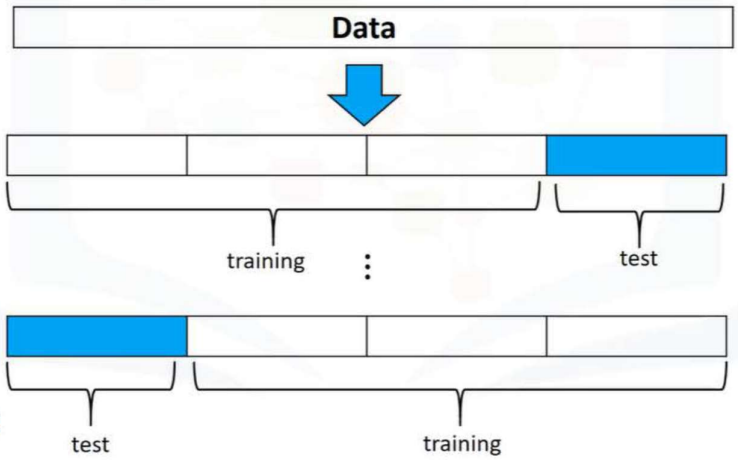
The figure above demonstrates this; all our error estimates are relatively close together, but they are further away from the true generalization performance.

To overcome this problem, we use **cross validation**.

One of the most common '**out-of-sample evaluation metrics**' is cross validation. In this method, the dataset is split into k-equal groups; each group is referred to as a fold.

Cross Validation

- Most common out-of-sample evaluation metrics
- More effective use of data (each observation is used for both training and testing)



COGNITIVE CLASS.ai

9

For example 4 folds.

Some of the folds can be used as a training set, which we use to train the model, and the remaining parts are used as a test set, which we use to test the model.

For example, we can 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.
The Simplest way to apply cross validation is to call the cross_val_score() function, which performs multiple 'out-of-sample' evaluations.

Function cross_val_score()

```
from sklearn.model_selection import cross_val_score

scores= cross_val_score(lr, x_data, y_data, cv=3)
```

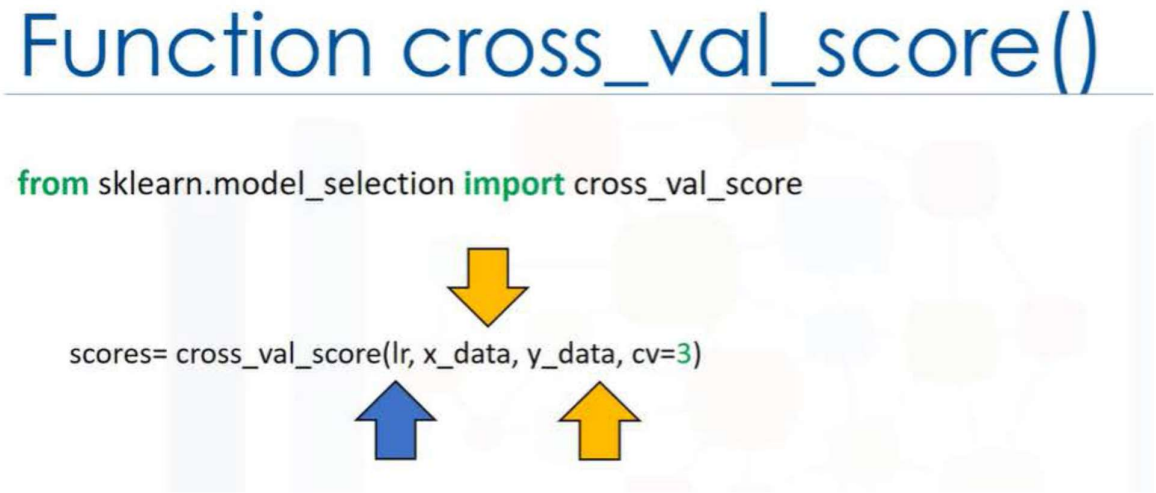
```
from sklearn.model_selection import cross_val_score

scores= cross_val_score(lr, x_data, y_data, cv=3)
```

This method is imported from sklearn's model selection package.

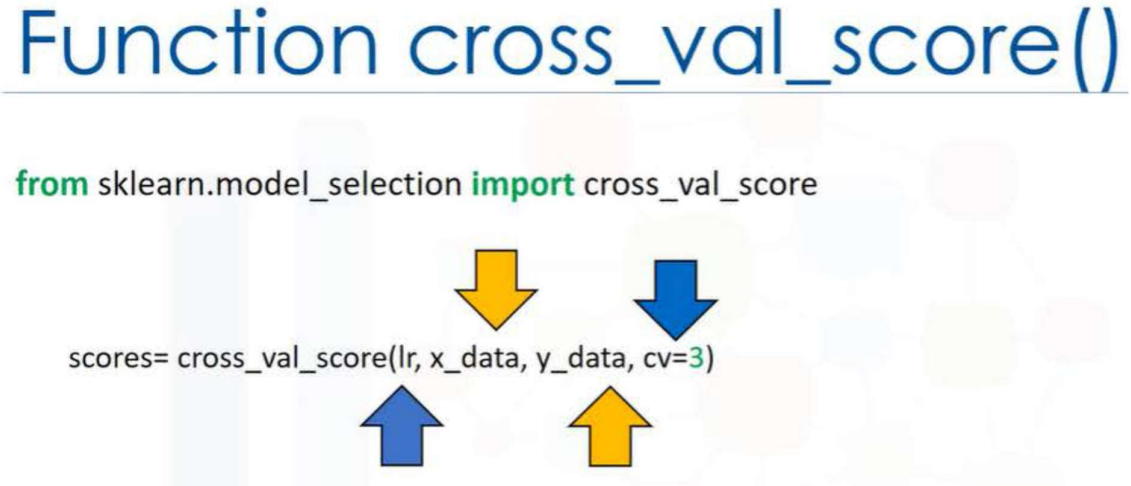
We then use the function cross_val_score(). The first input parameter is the type of model we are using to do the cross validation.

In this example, we initialized a linear regression model or object lr, which we passed to the cross_val_score function.

The other parameters are x_data, the predictor variable data, and y_data, the target variable data.

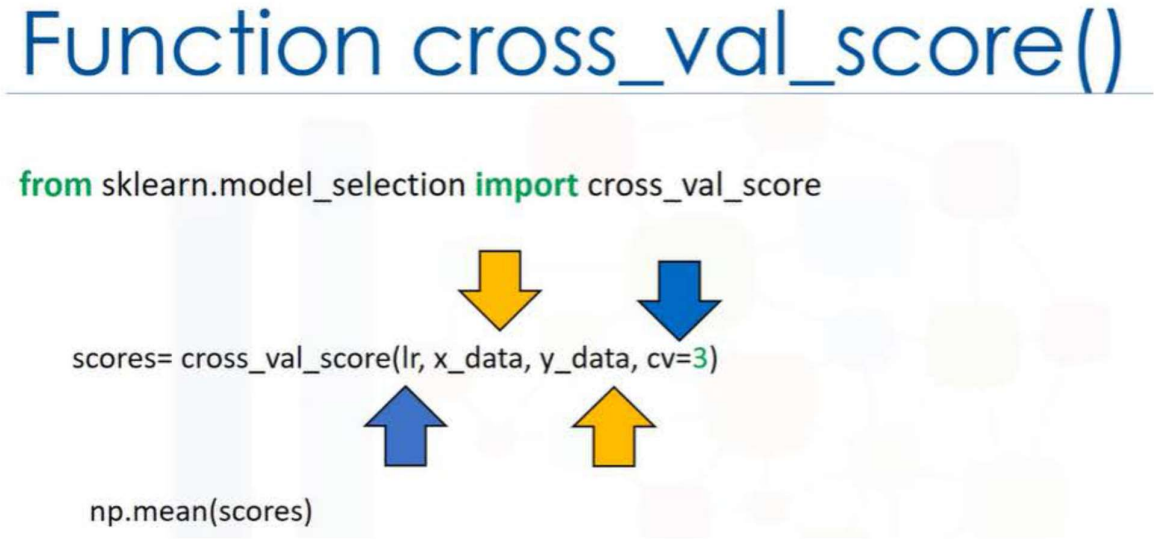


We can manage the number of partitions with the cv parameter. Here, cv = 3, which means the data set is split into 3 equal partitions.



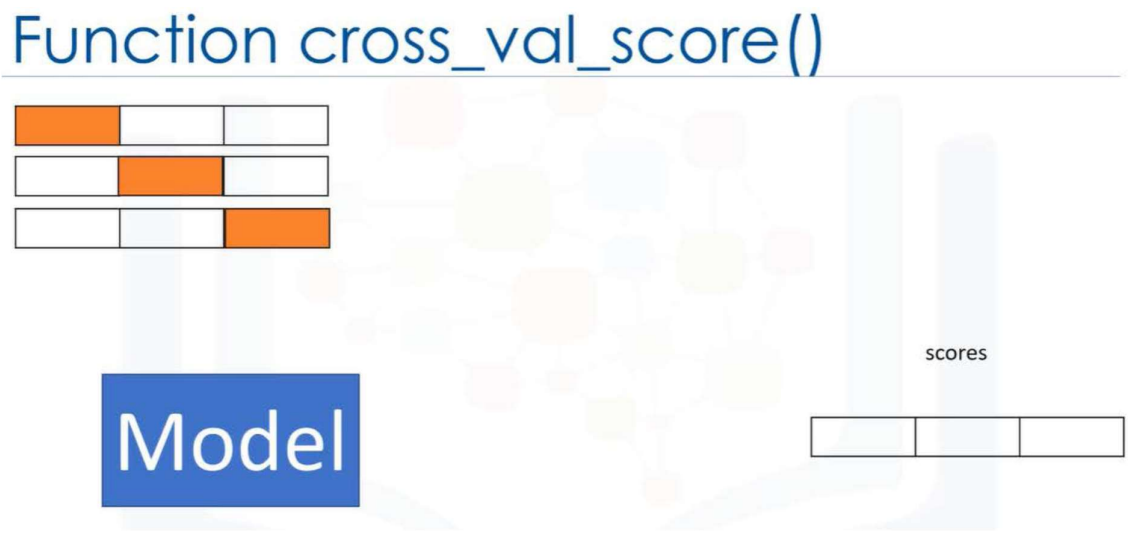
The function returns an array of scores, one for each partition that was chosen as the testing set.

We can average the result together to estimate out-of-sample R-squared using the mean function in numpy.



Let's see an animation.

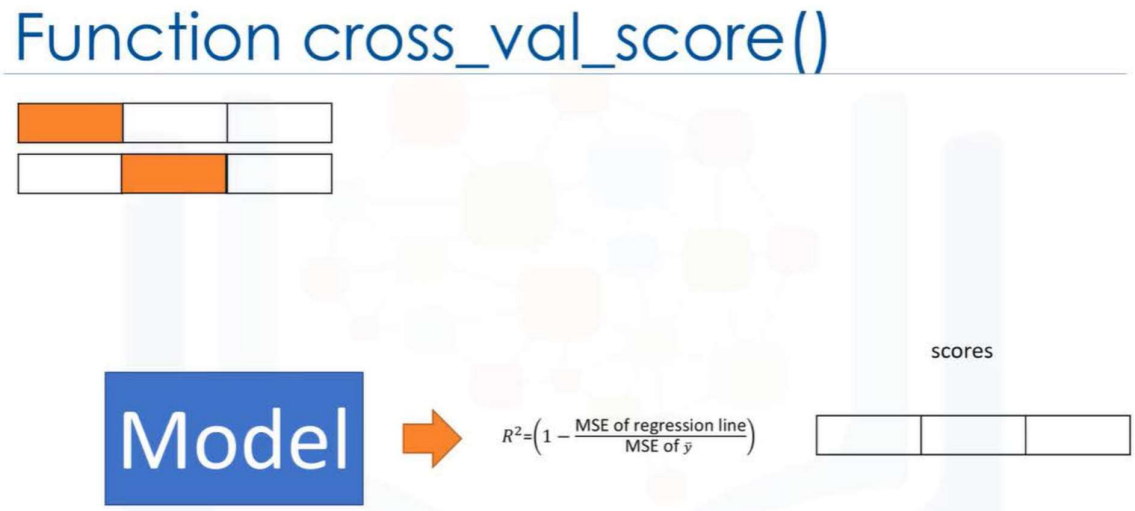
Let's see the result of the score array in the last slide.



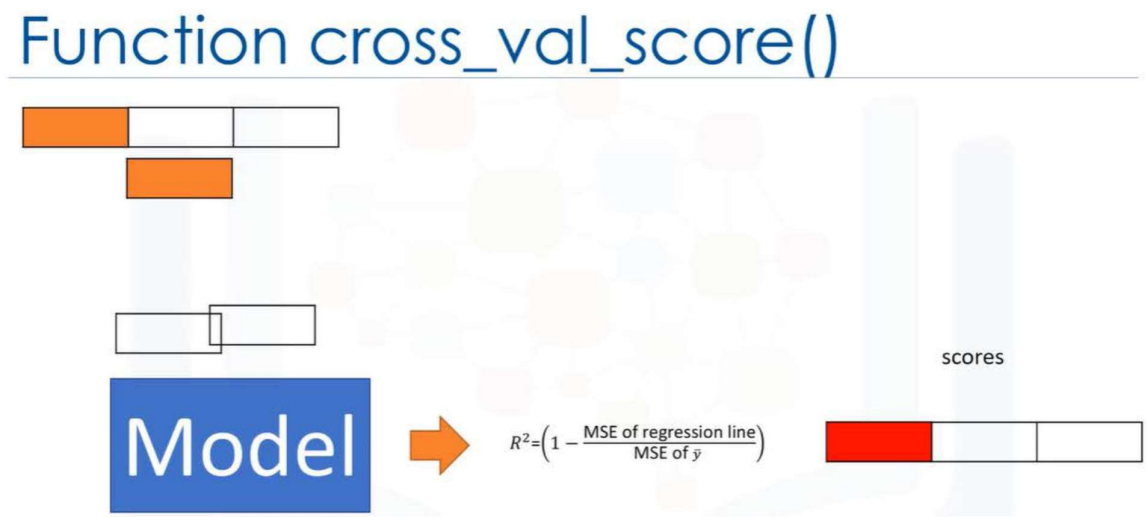
First, we split the data into three folds. We use two folds for training; the remaining fold for testing.

The model will produce an output.

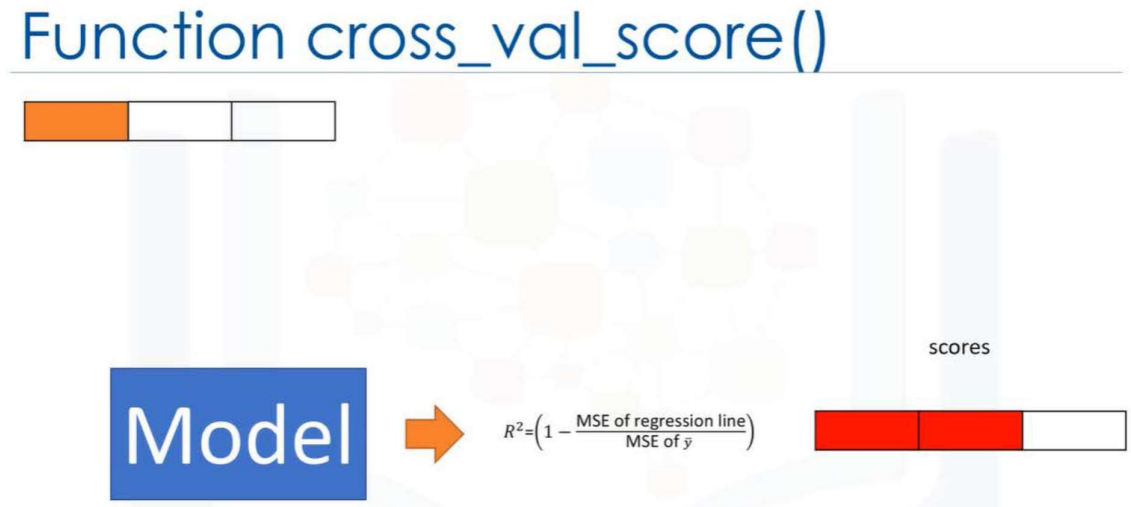
We will use the output to calculate a score. In the case of the R-squared i.e., coefficient of determination.

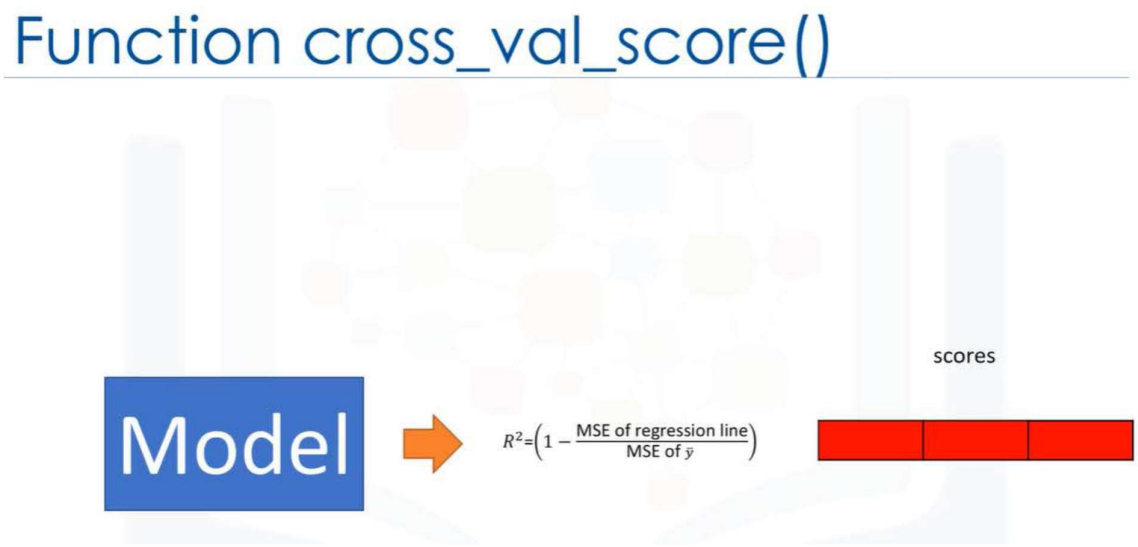


We will store that value in an array.



We will repeat the process using two folds for training, and one fold one for testing, save the score, then use a different combination for training, and the remaining fold for testing. We store the final result.

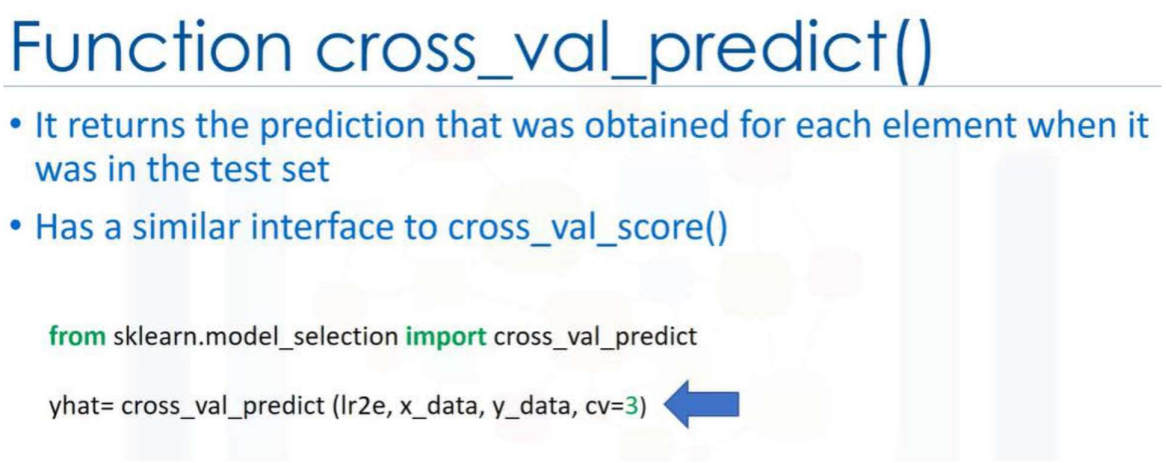




The cross_val_score() function returns a score value to tell us the cross-validation result.

What if we want a little more information: what if we want to know the actual predicted values supplied by our model before the R squared values are calculated?

To do this, we use the **cross_val_predict()** function.



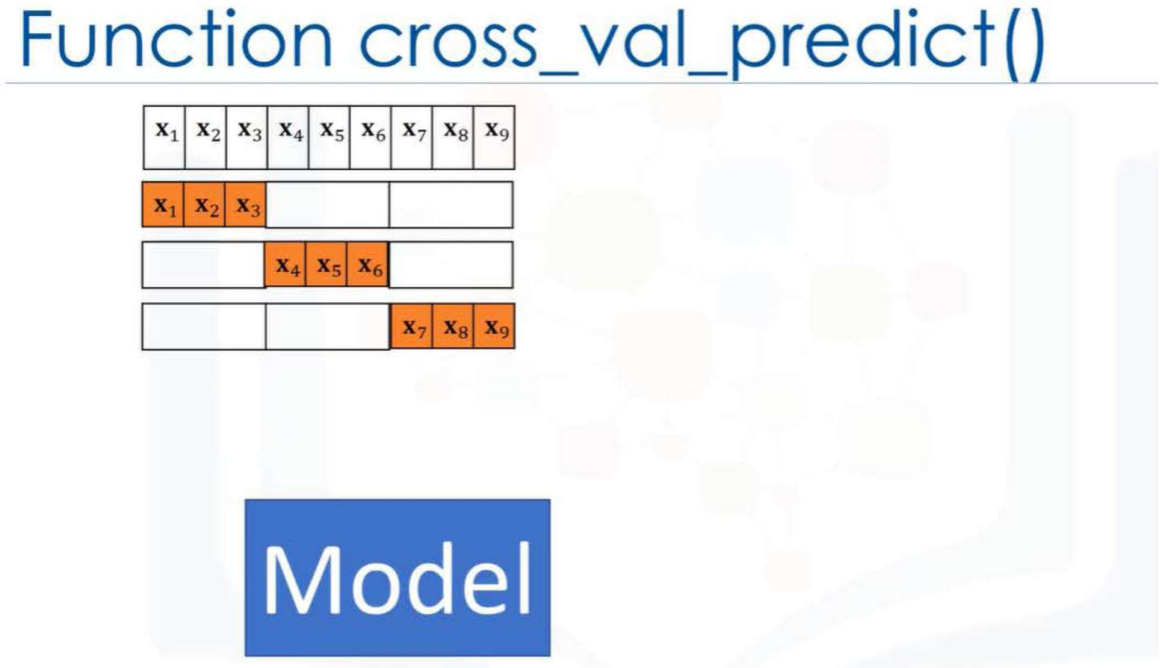
from sklearn.model_selection import cross_val_predict

yhat= cross_val_predict (lr2e, x_data, y_data, cv=3)

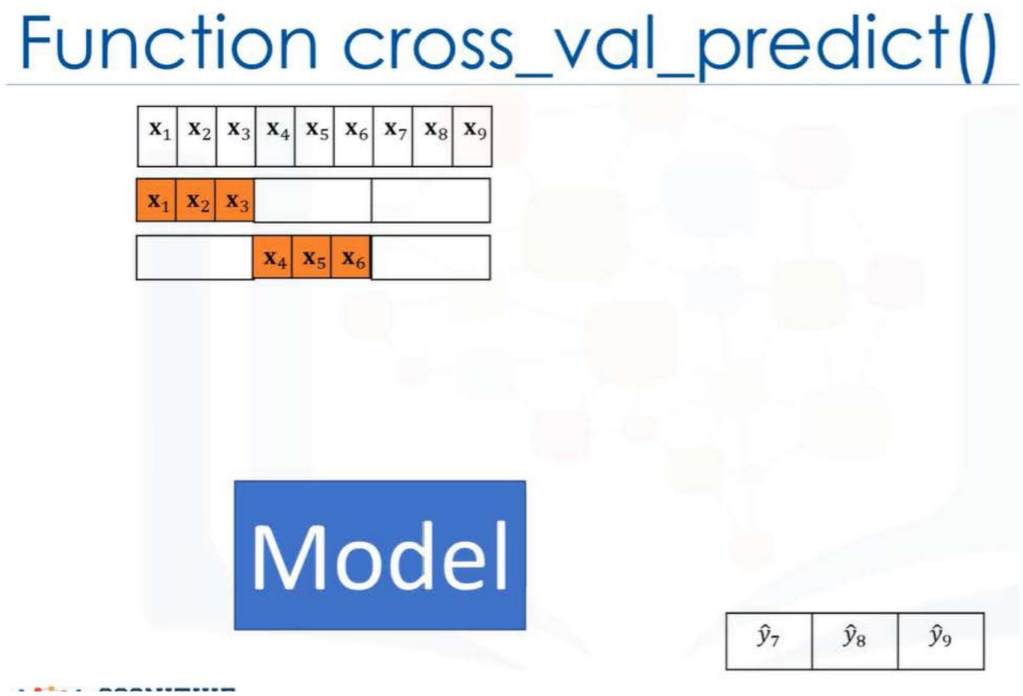
The input parameters are exactly the same as the cross_val_score() function, but the output is a prediction.

Let's illustrate the process.

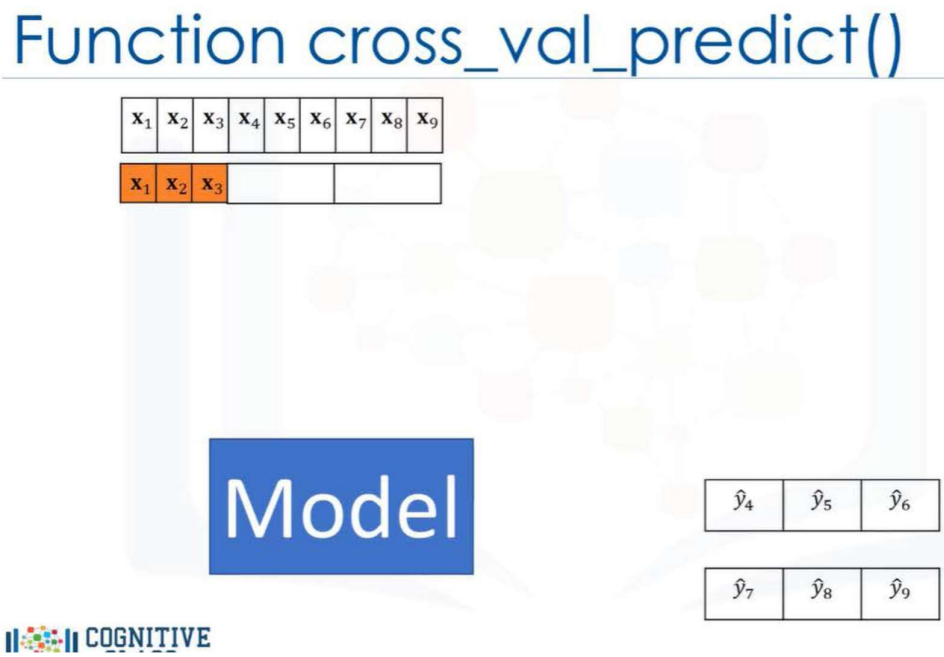
First, we split the data into three folds; we use two folds for training, the remaining fold for testing.



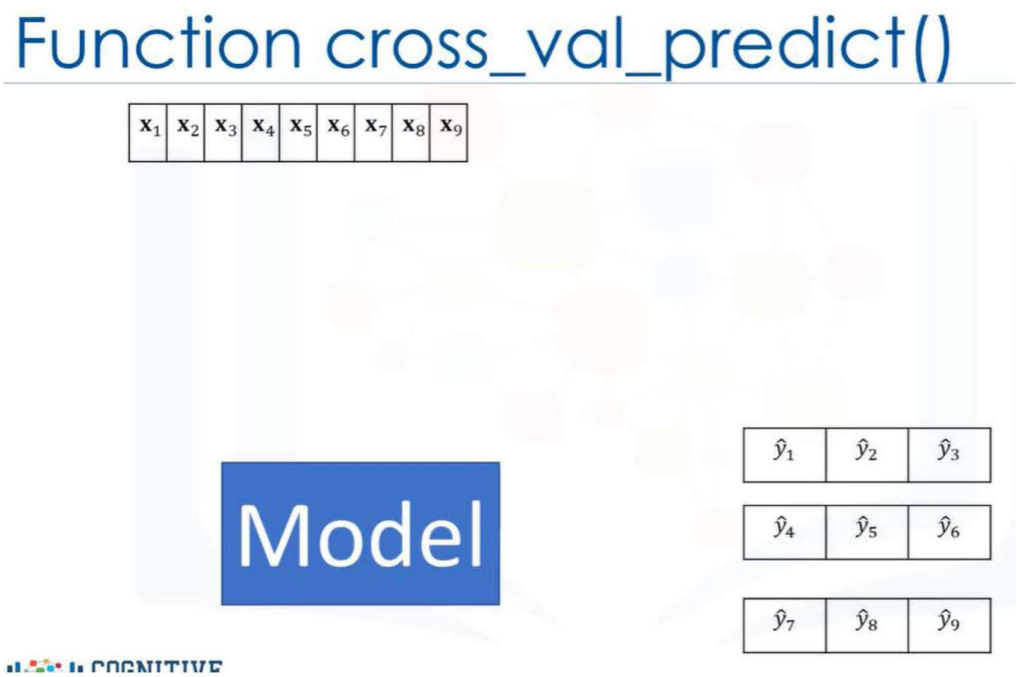
The model will produce an output, and we will store it in and array.



We will repeat the process using two folds for training, one for testing. The model produces an output again.



Finally, we use the last two folds for training, then we use the testing data.
This final testing fold produces an output.
These predictions are stored in an array.



Overfitting, Underfitting, Model Selection

If you recall, in the last Module we discussed polynomial regression.

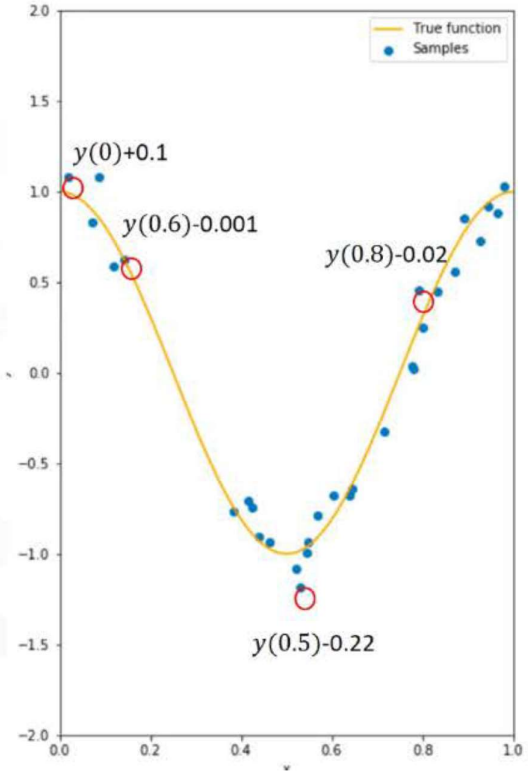
In this section, we will discuss how to pick the best polynomial order and problems that arise with selecting the wrong order polynomial.

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.

Model Selection

$y(x)$+noise



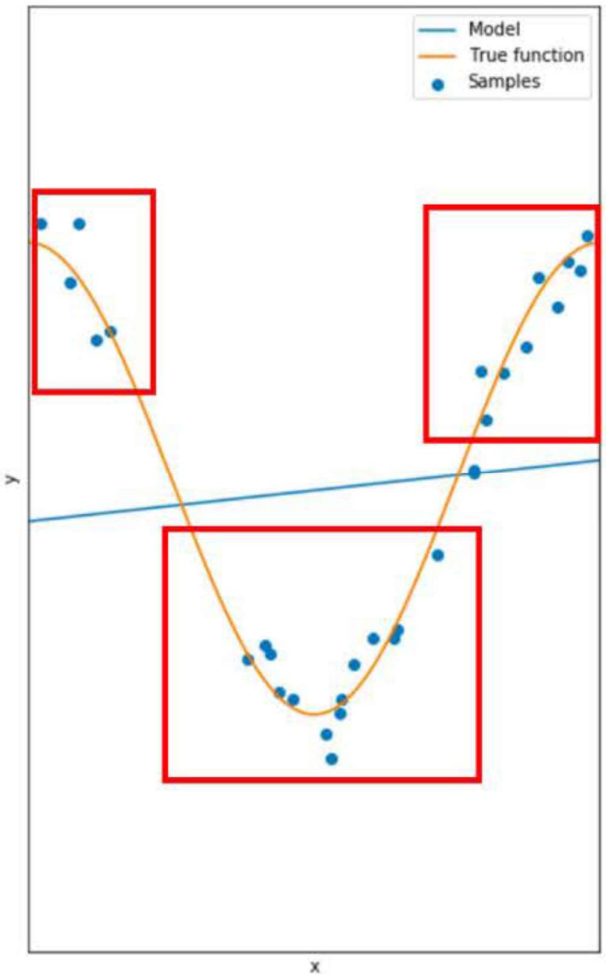
COGNITIVE CLASS.ai

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.

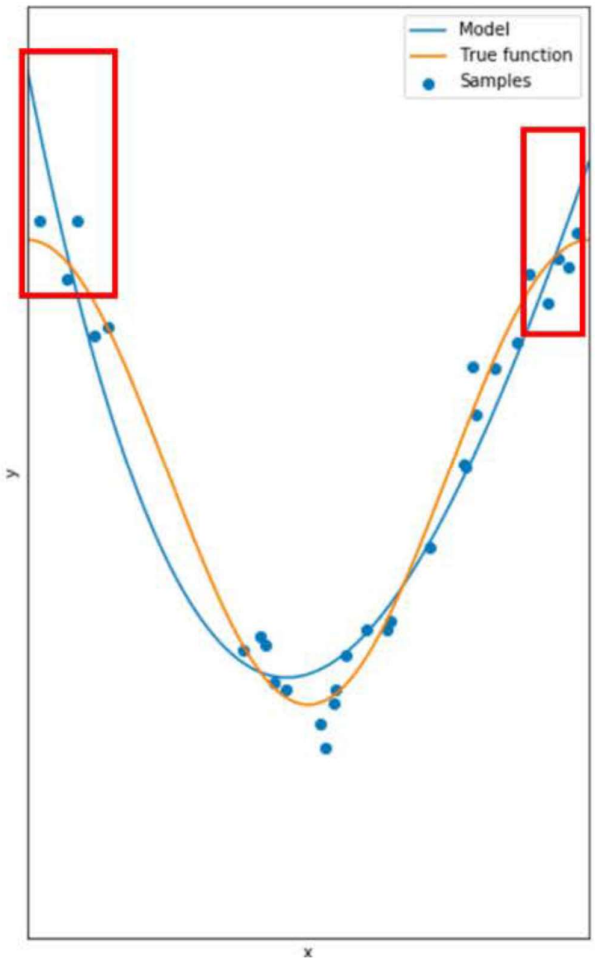
This is called under-fitting, where the model is too simple to fit the data.

$$y = b_0 + b_1 x$$



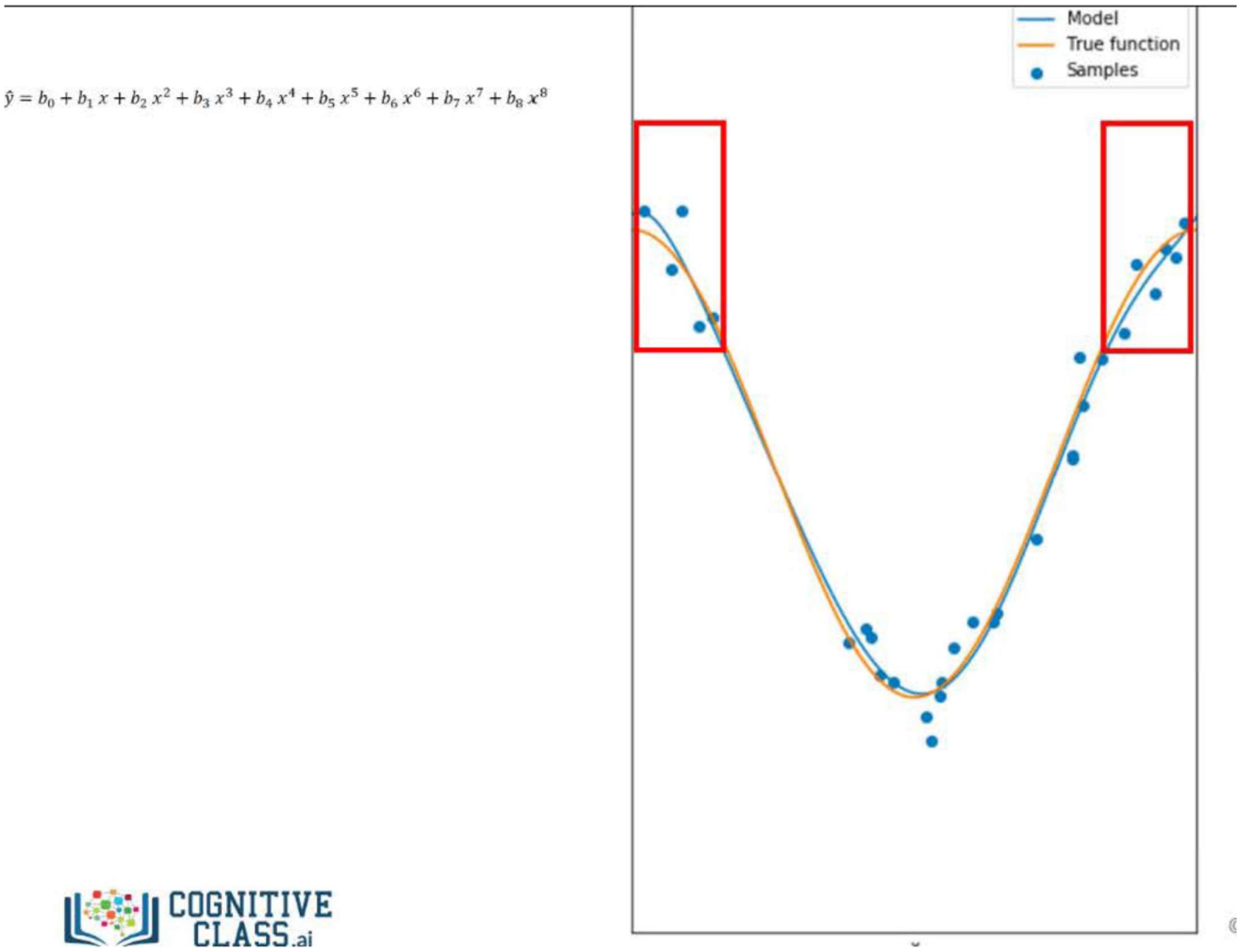
If we increase the order of the polynomial, the model fits better, but the model is still not flexible enough and exhibits under-fitting.

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



COGNITIVE CLASS.ai

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.



Increasing it to a 16th order polynomial, the model does extremely well at tracking the training points, but performs poorly at estimating the function.

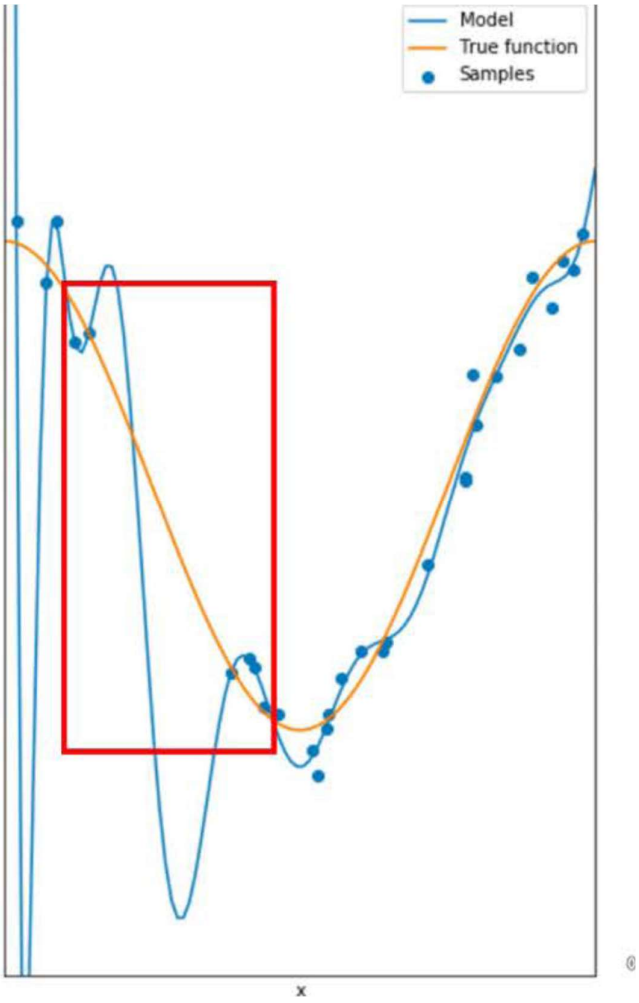
This is especially apparent where there is little training data; the estimated function oscillates not tracking the function.

This is called over-fitting, where the model is too flexible and fits the noise rather

than the function.

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

$$+ b_9 x^9 + b_{10} x^{10} + b_{11} x^{11} + b_{12} x^{12} + b_{13} x^{13} + b_{14} x^{14} + b_{15} x^{15} + b_{16} x^{16}$$



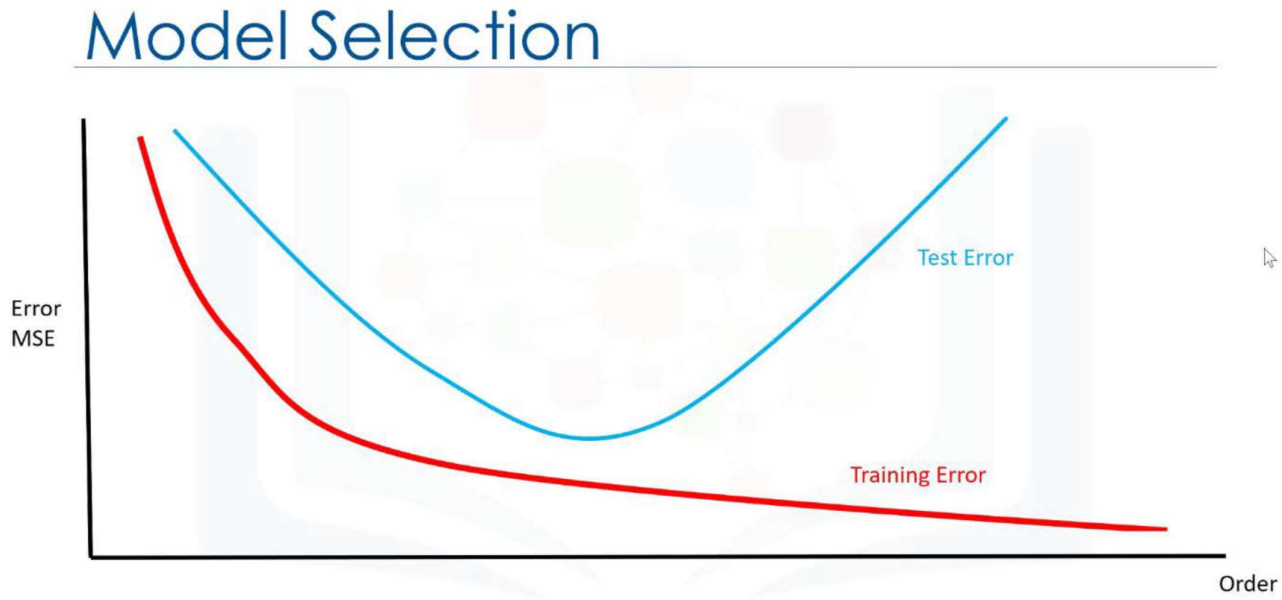
COGNITIVE CLASS.ai

Let's look at a plot of the mean square error for the training and testing set for different order polynomials.

The horizontal axis represents the order of the polynomial; the vertical axis is the mean square error.

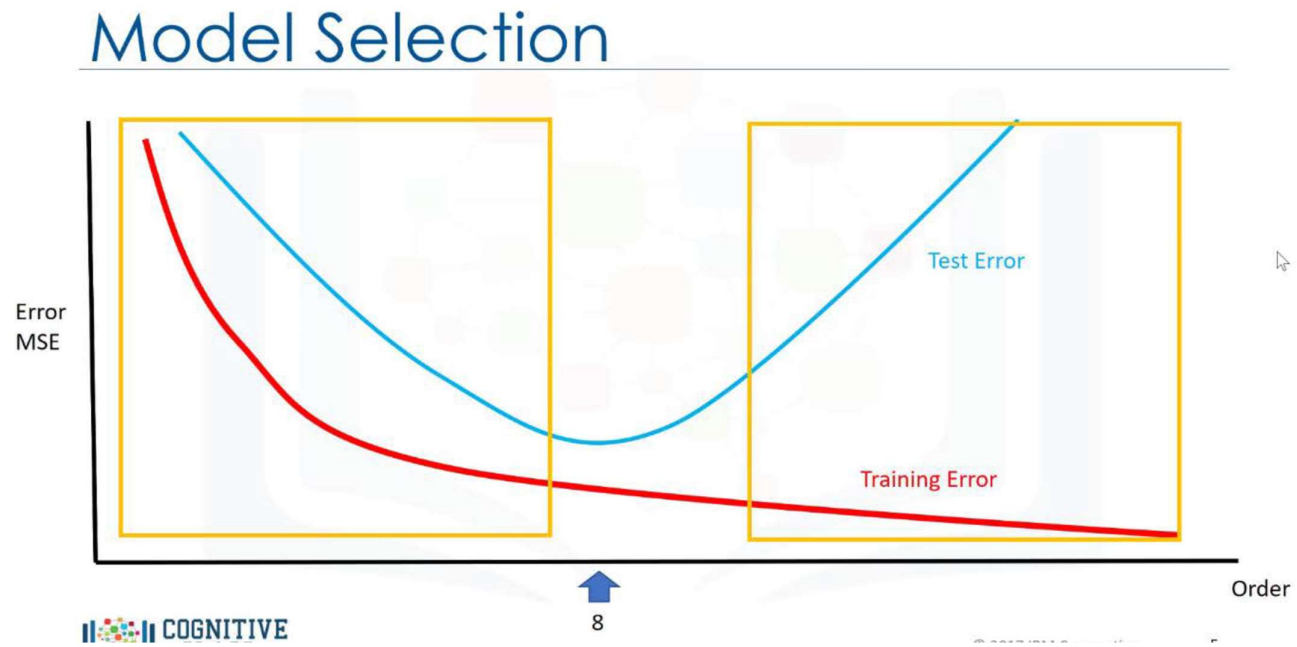
The training error decreases with the order of the polynomial.

The test error is a better means of estimating the error of a polynomial. 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 8.

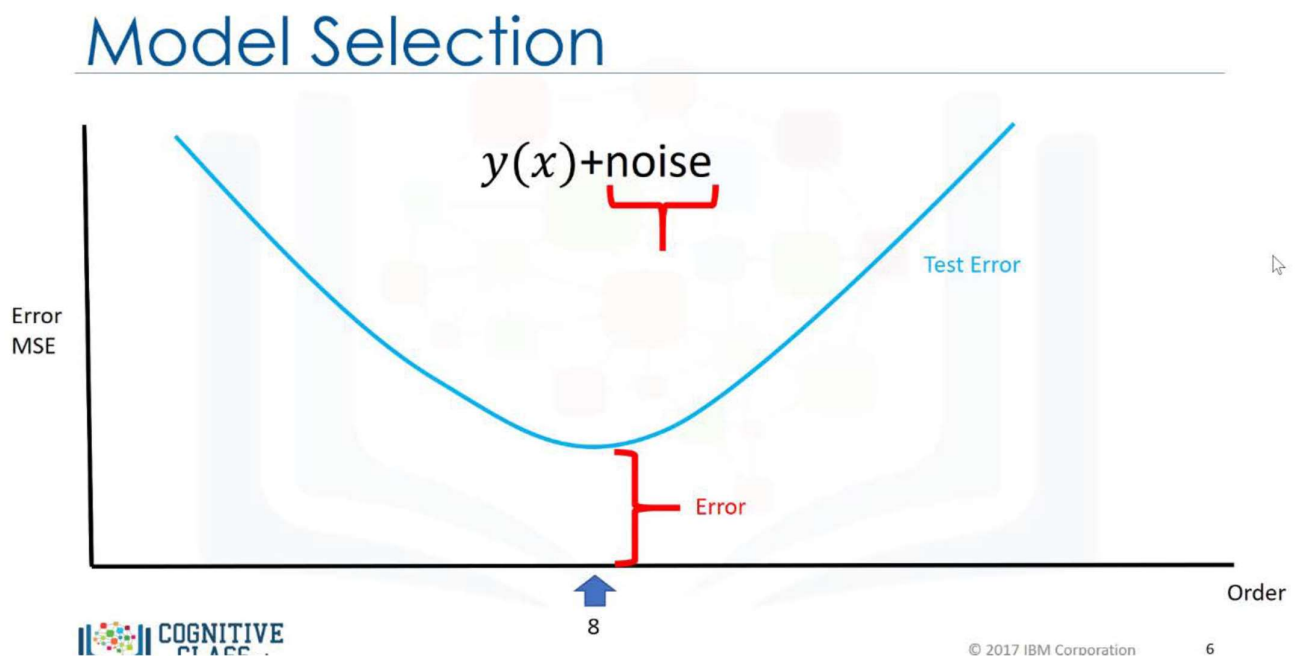
Anything on the left would be considered under-fitting.
Anything on the right is over-fitting.

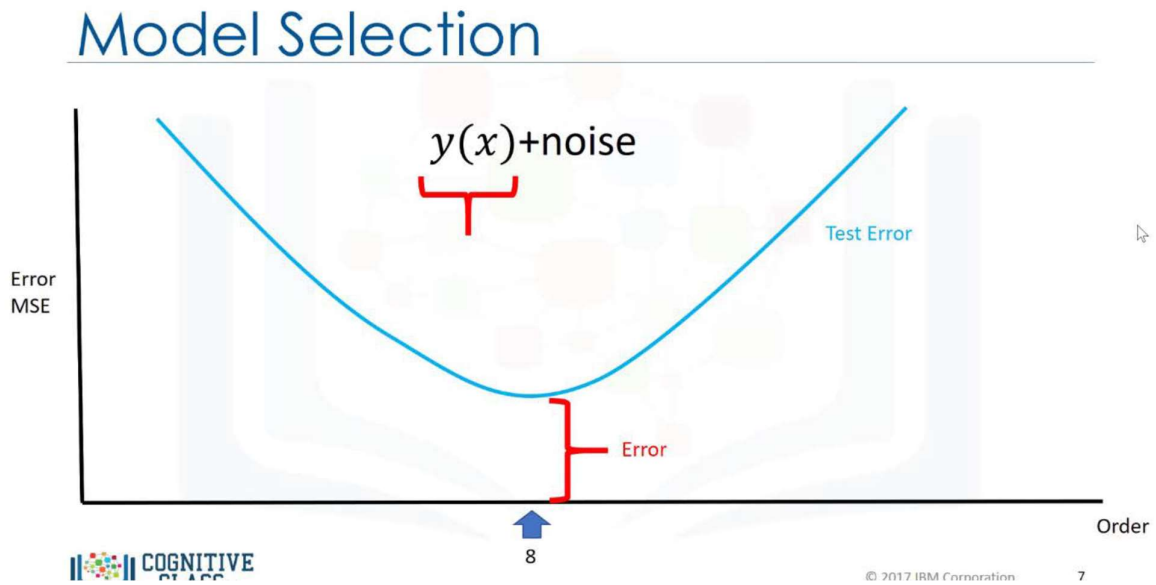


If we select the best order of the polynomial, we will still have some errors, if you recall, the original expression for the training points.

We see a noise term; this term is one reason for the error.

This is because the noise is random and we can't predict it; this is sometimes referred to as an irreducible error.





There are other sources of errors as well.

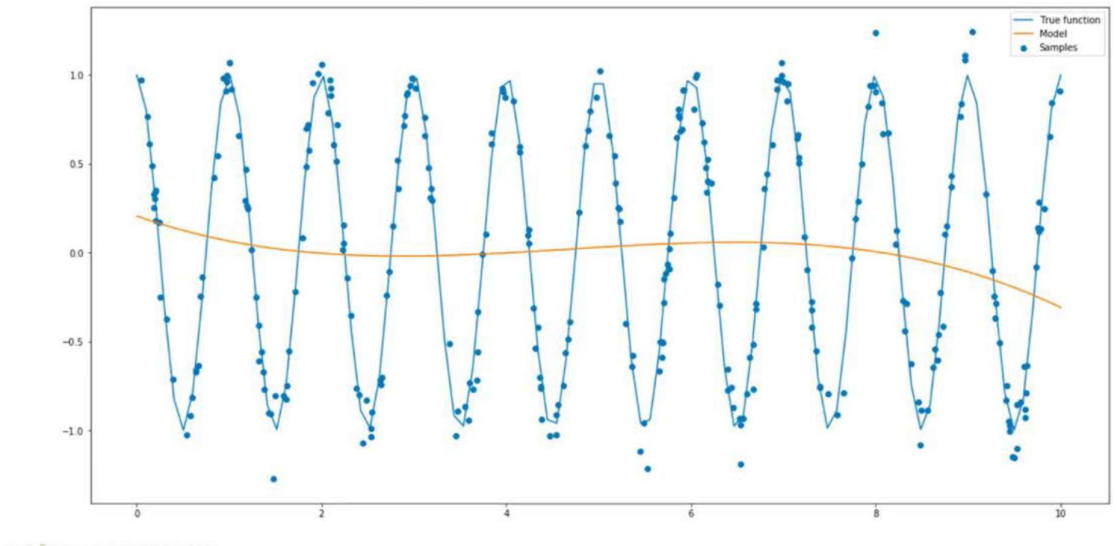
For example, our polynomial assumption may be wrong.

Our sample points may have come from a different function.

For example, in this plot, the data is generated from a sine wave; the polynomial function does not do a good job at fitting the sine wave.

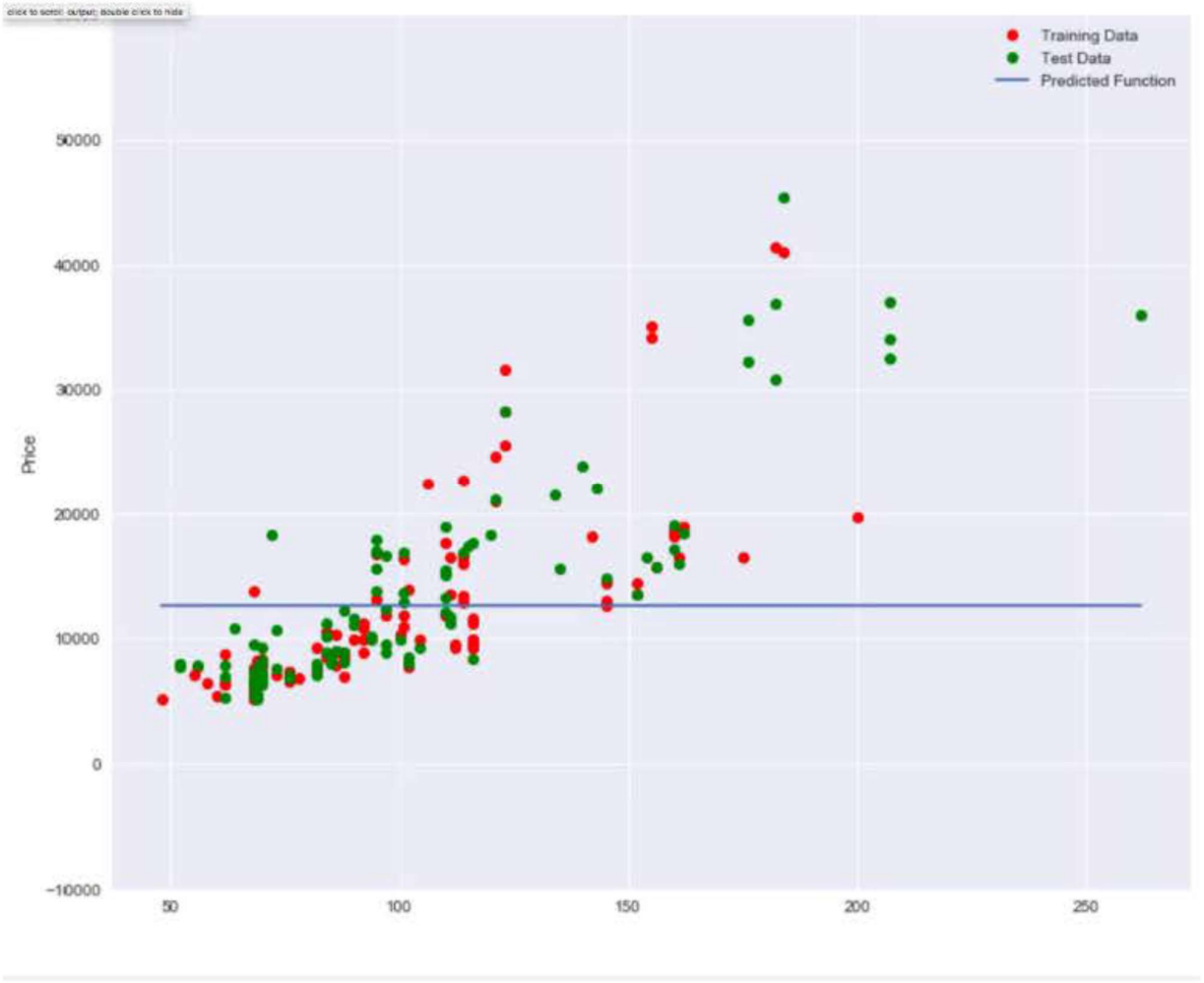
For real data, the model may be too difficult to fit, or we may not have the correct type of data to estimate the function.

Model Selection

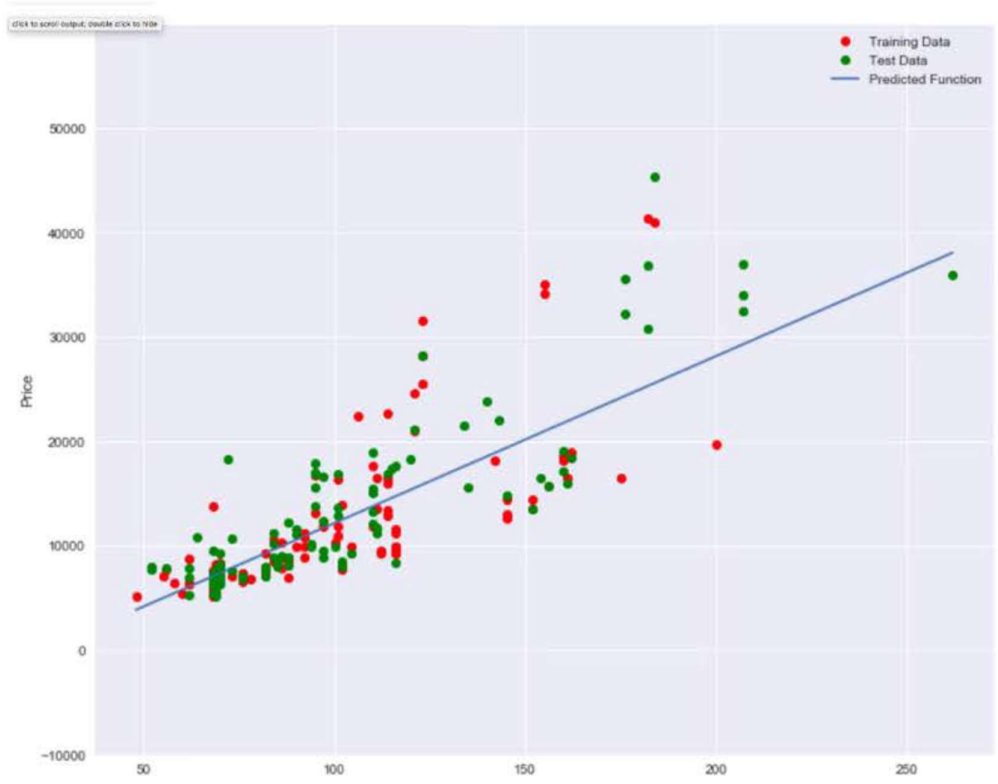


Let's try different order polynomials on the real data using horse power; the red points represent the training data; the green points represent the test data.

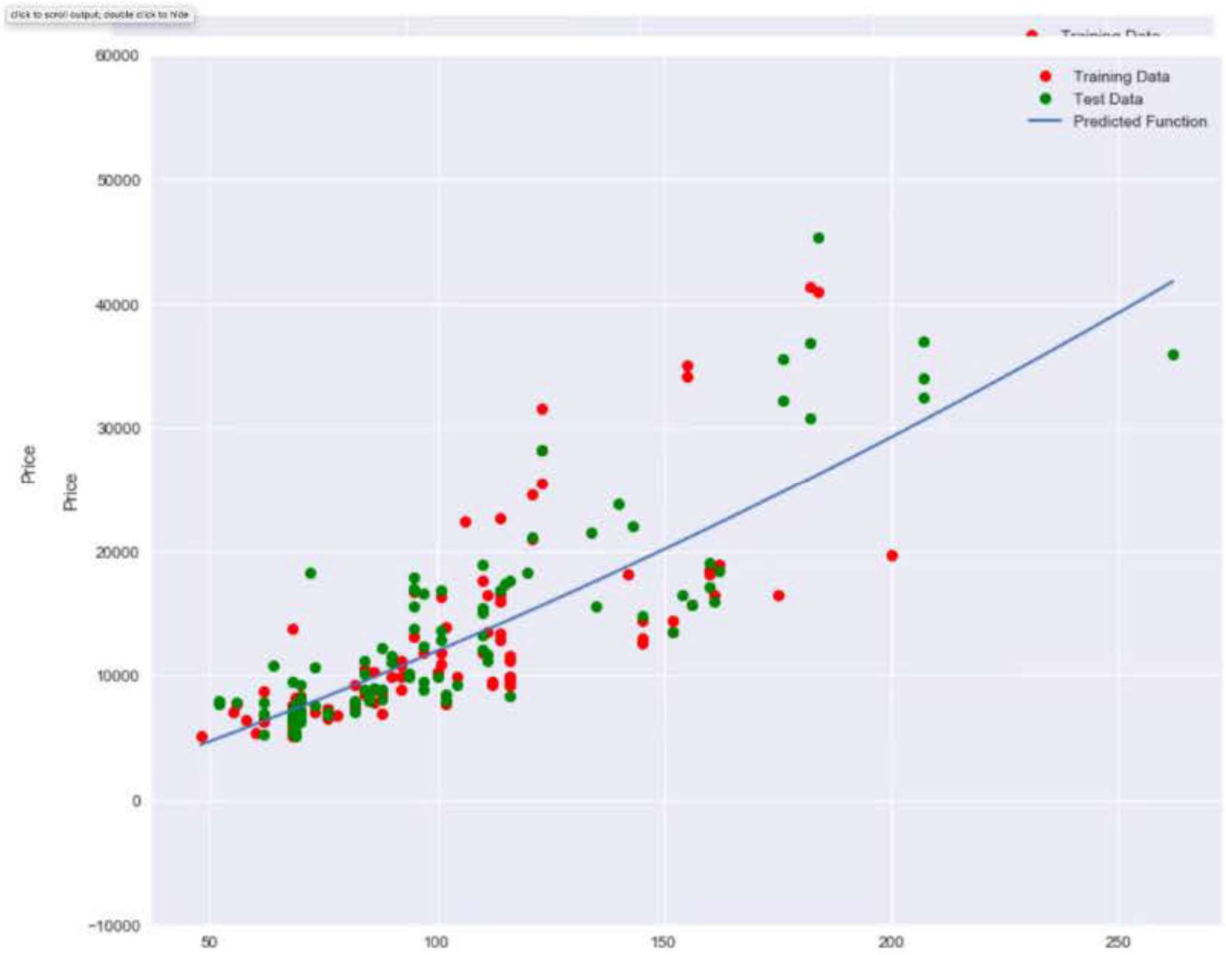
If we just use the mean of the data, our model does not perform well.



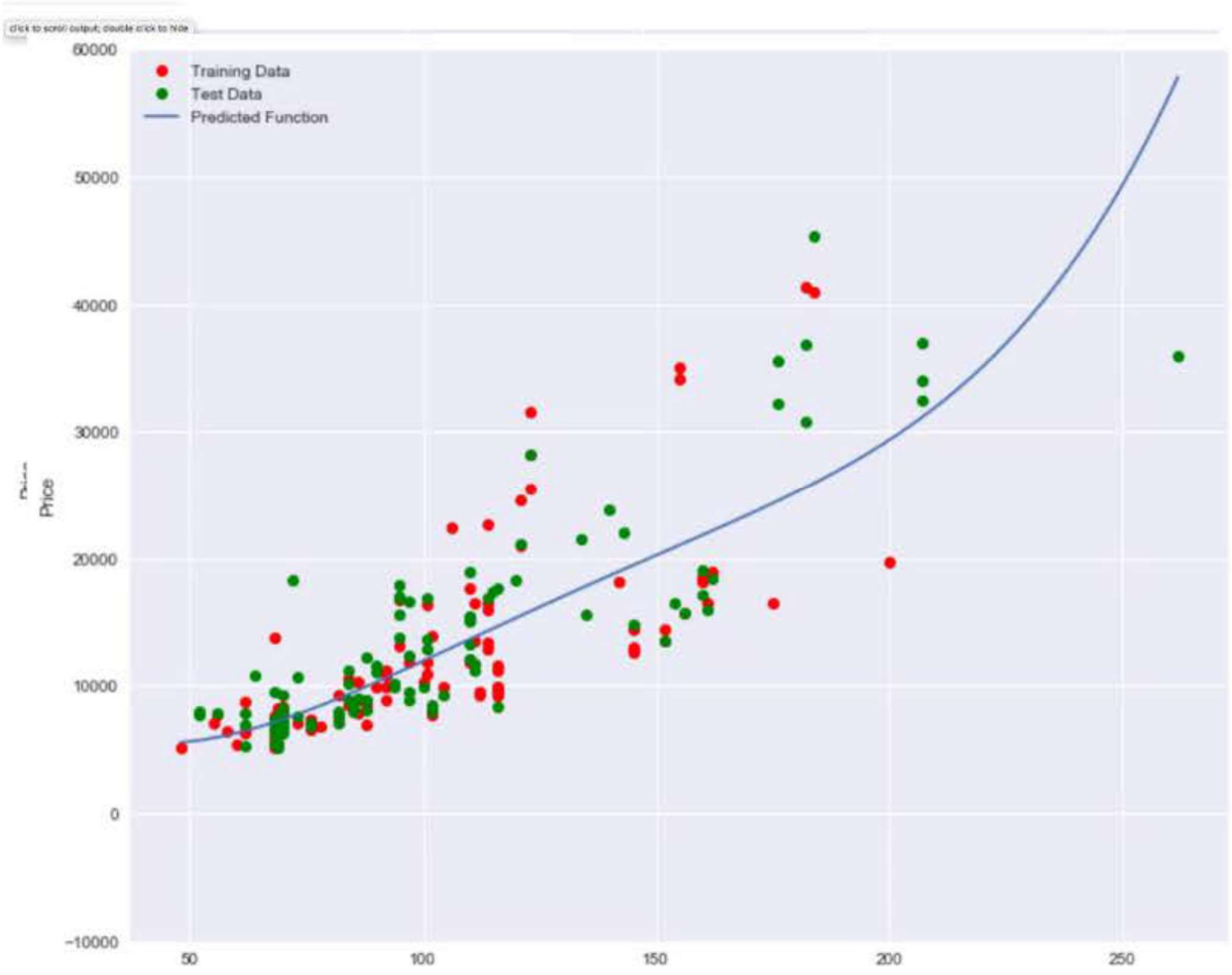
A linear function does fit the data better.



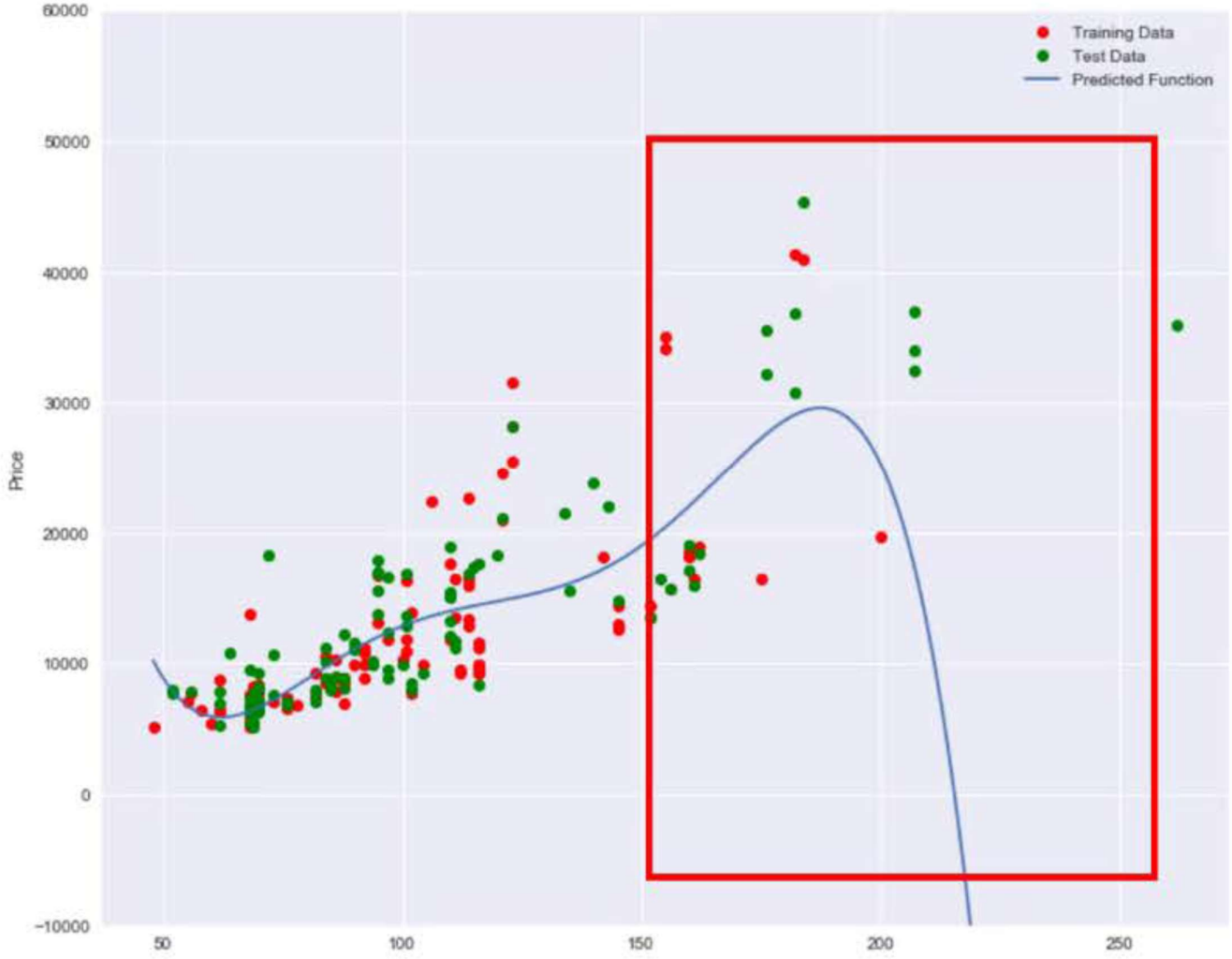
A second order model looks similar to the linear function.



A third order function also appears to increase, like the previous two orders.



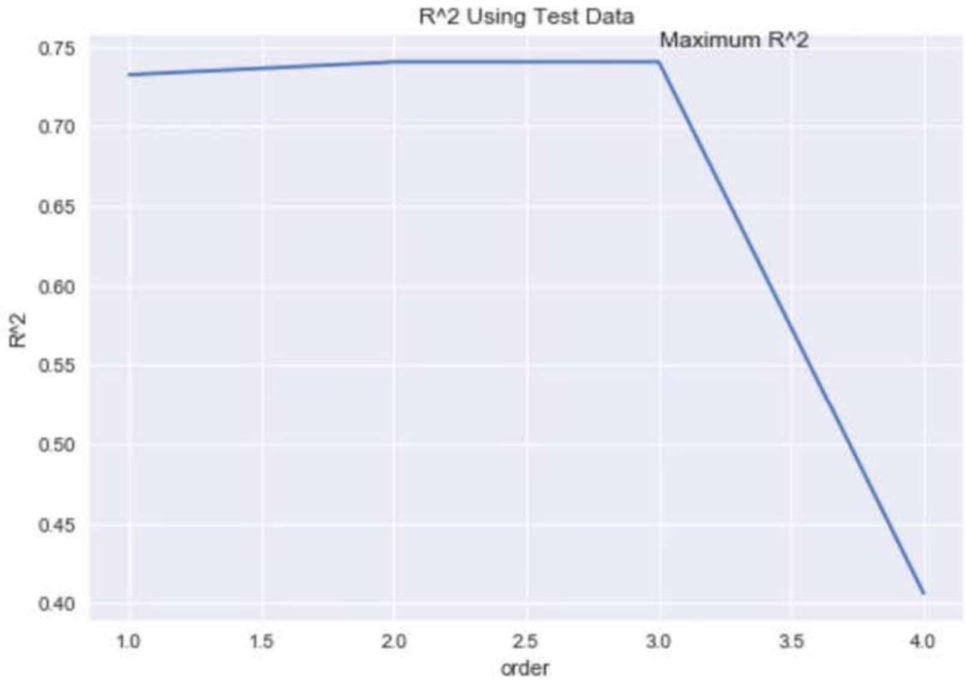
Here we see a 4th order polynomial. At around 200 horse power, the predicted price suddenly decreases; this seems erroneous.



Let's use R-squared to see if our assumption is correct.

The following is a plot of the R-squared value, the horizontal axis represents the order of polynomial models.
The closer the R-squared is to 1, the more accurate the model is.

Model Selection

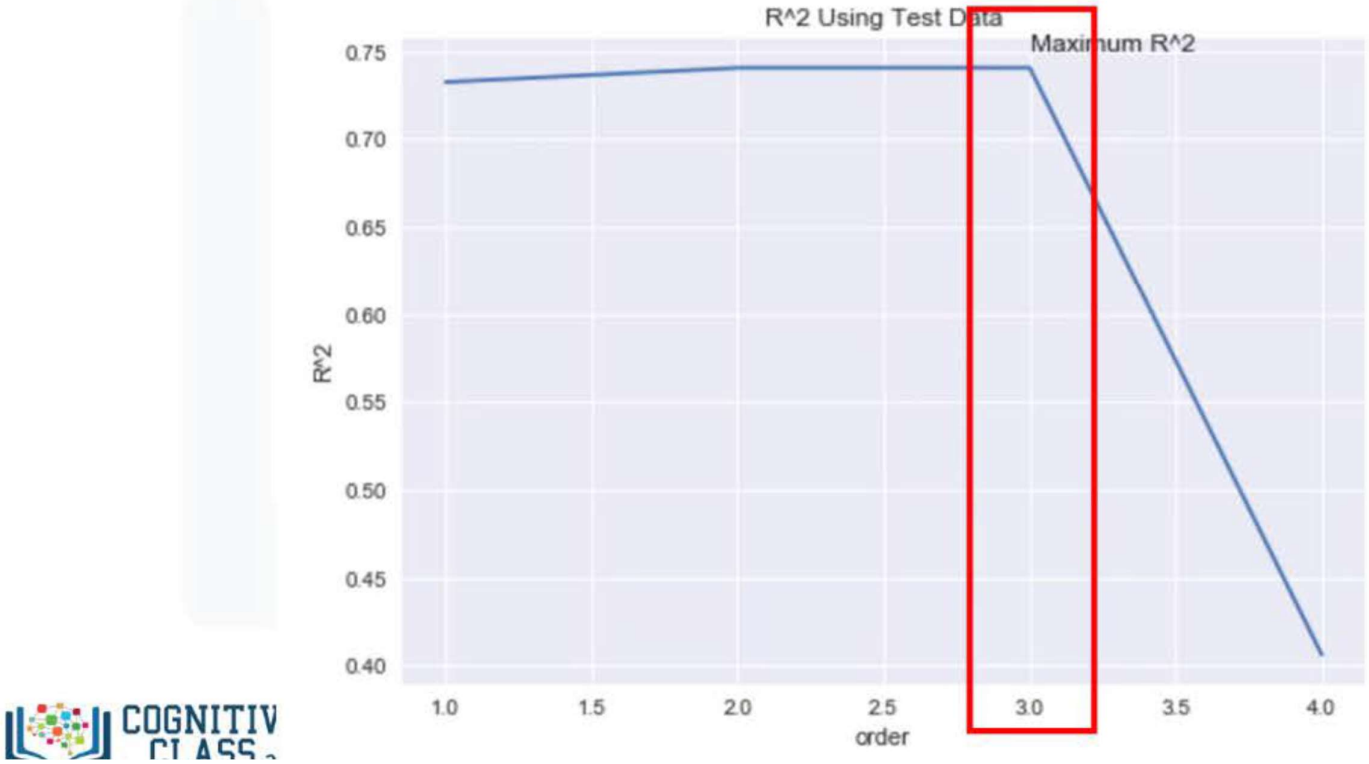


COGNITIV CLASS.a

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 4, validating our initial assumption.

Model Selection



We can calculate different R-squared values as follows:

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.

```
for n in order:
```

We create a polynomial feature object withthe order of the polynomial as a parameter.

```
pr=PolynomialFeatures(degree=n)
```

We transform the training and test data into a polynomial using the fit transform method.

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

We fit the regression model using the transformed data.

```
lr.fit(x _train _pr, y _train)
```

We then calculate the R-squared using the test data and store it in the array.

```
Rsqu_test.append(lr.score(x_test_pr, y _test))
```

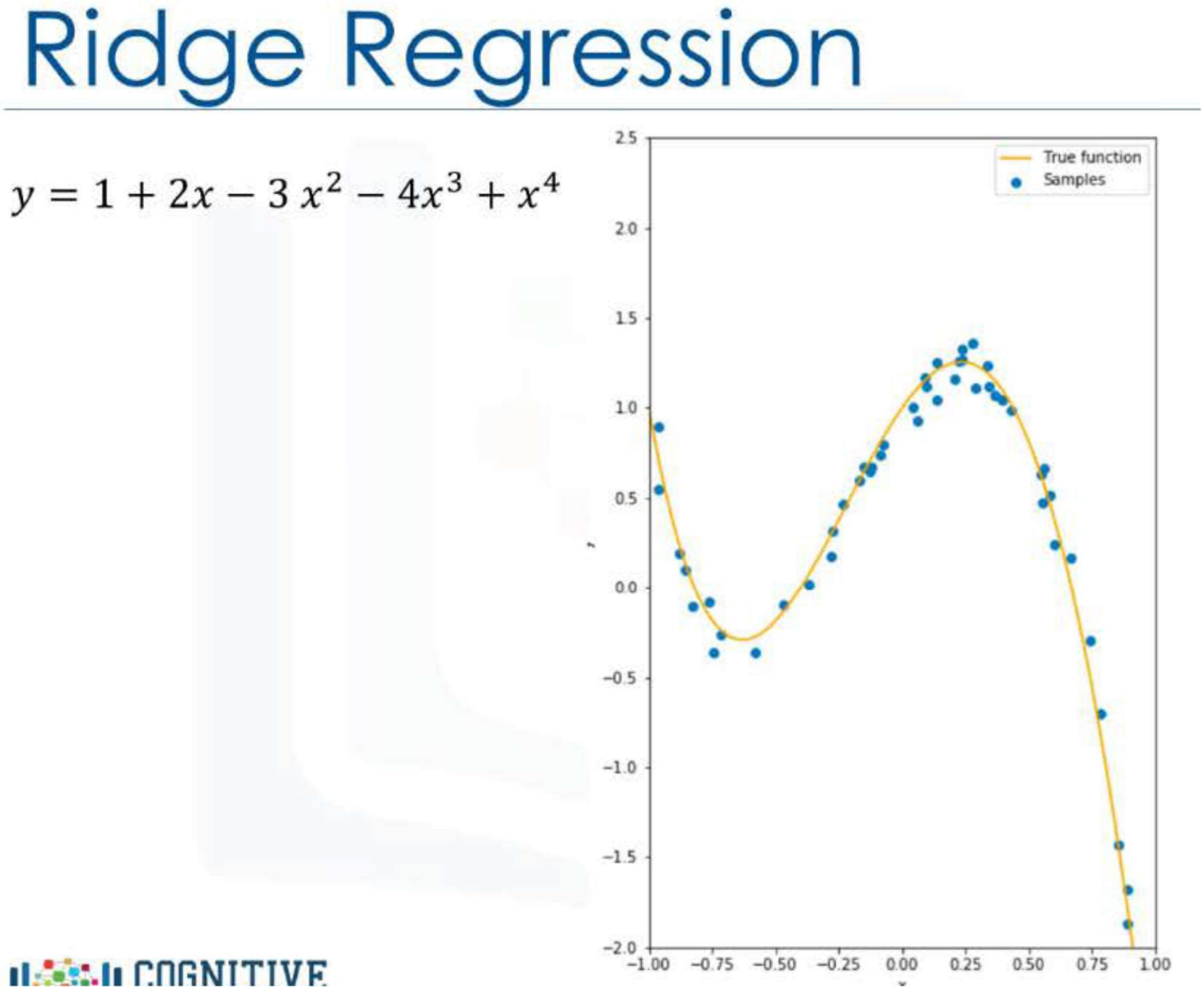
RIDGE REGRESSION

In this video we'll discuss Ridge Regression.

Ridge regression prevents over-fitting.

In this video we will focus on polynomial regression for visualization, but over-fitting is also a big problem when you have multiple independent variables or features.

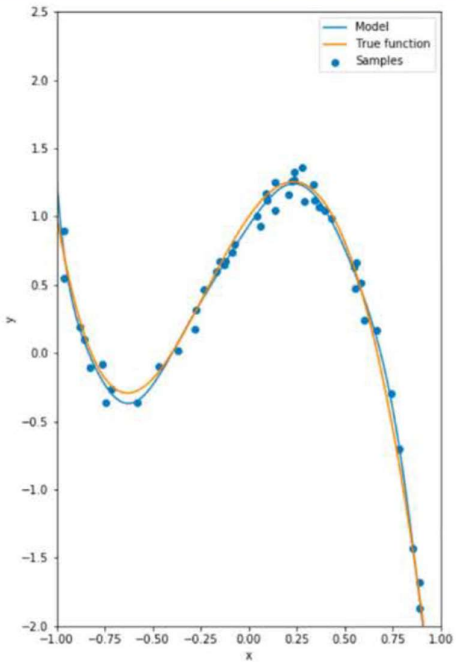
Consider the following 4th order polynomial in orange. The blue points are generated from this function.



We can use a 10th order polynomial to fit the data. The estimated function in blue does a good job at approximating the true function.

Ridge Regression

$y = 1 + 2x - 3\,x^2 - 4x^3 + x^4$

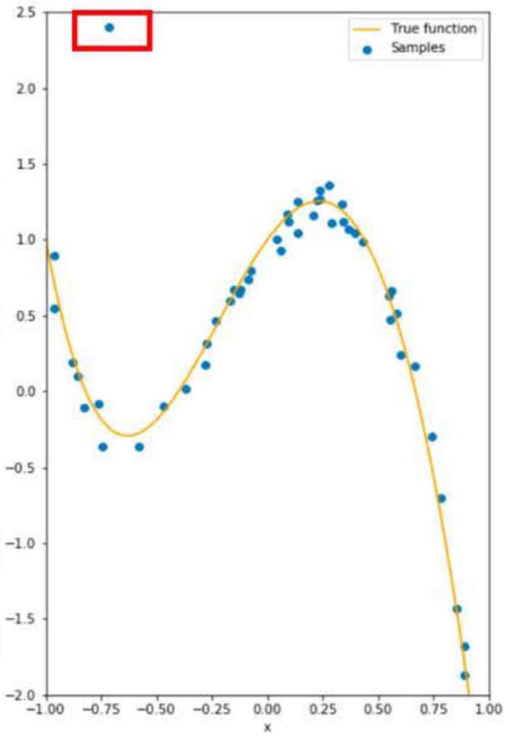


COGNITIVE

In many cases real data has outliers. For example, this point shown here does not appear to come from the function in orange.

Ridge Regression

$1 + 2x - 3\,x^2 - 4x^3 + x^4$

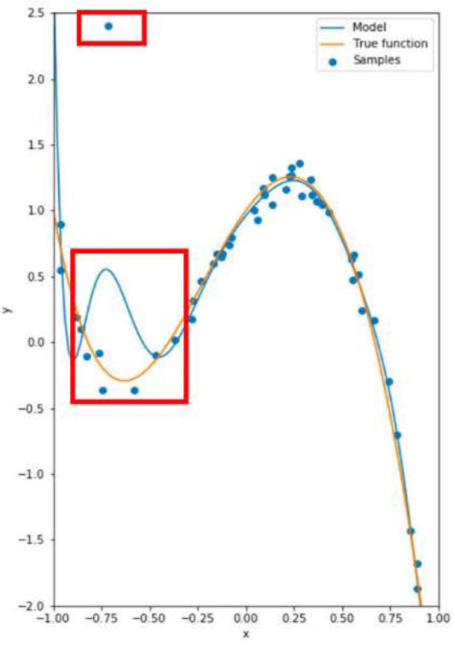


COGNITIVE

If we use a 10th order polynomial function to fit the data, the estimated function in blue is incorrect and is not a good estimate of the actual function in orange.

Ridge Regression

$1 + 2x - 3\,x^2 - 4x^3 + x^4$



If we examine the expression for the estimated function, we see the estimated polynomial coefficients have a very large magnitude.

Ridge Regression

$\hat{y} = 1 + 2\,x - 3\,x^2 - 2x^3 - 12\,x^4 - 40x^5 + 80x^6 + 71x^7 - 141x^8 - 38x^9 + 75x^{10}$

This is especially evident for the higher order polynomials. Ridge regression controls the magnitude of these polynomial coefficients by introducing the parameter alpha.

Alpha is a parameter we select before fitting or training the model. Each row in the following table represents an increasing value of alfa. Let's see how different values of alpha change the model.

Ridge Regression

$$\hat{y} = 1 + 2\,x - 3\,x^2 - 2x^3 - 12\,x^4 - 40x^5 + 80x^6 + 71x^7 - 141x^8 - 38x^9 + 75x^{10}$$

Alpha	x	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^{10}
0	2	-3	-2	-12	-40	80	71	-141	-38	75
0.001	2	-3	-7	5	4	-6	4	-4	4	6
0.01	1	-2	-5	-0.04	0.15	-1	1	-0.5	0.3	1
1	0.5	-1	-1	-0.614	0.70	-0.38	-0. 56	-0.21	-0.5	-0.1
10	0	-0.5	-0.3	-0.37	-0.30	-0.30	-0.22	-0.22	-0.22	-0.17

Ridge Regression

$$\hat{y} = 1 + 2\,x - 3\,x^2 - 2x^3 - 12\,x^4 - 40x^5 + 80x^6 + 71x^7 - 141x^8 - 38x^9 + 75x^{10}$$

Alpha	x	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^{10}
0	2	-3	-2	-12	-40	80	71	-141	-38	75
0.001	2	-3	-7	5	4	-6	4	-4	4	6
0.01	1	-2	-5	-0.04	0.15	-1	1	-0.5	0.3	1
1	0.5	-1	-1	-0.614	0.70	-0.38	-0. 56	-0.21	-0.5	-0.1
10	0	-0.5	-0.3	-0.37	-0.30	-0.30	-0.22	-0.22	-0.22	-0.17

This table represents the polynomial coefficients for different values of alfa. The columns correspond to the different polynomial coefficients and the rows correspond to the different values of alfa.

As alfa increases, the parameters get smaller. This is most evident for thE higher order polynomial features, but alpha must be selected carefully. If alpha is too large, the coefficients will approach zero and under-fit the data.

Ridge Regression

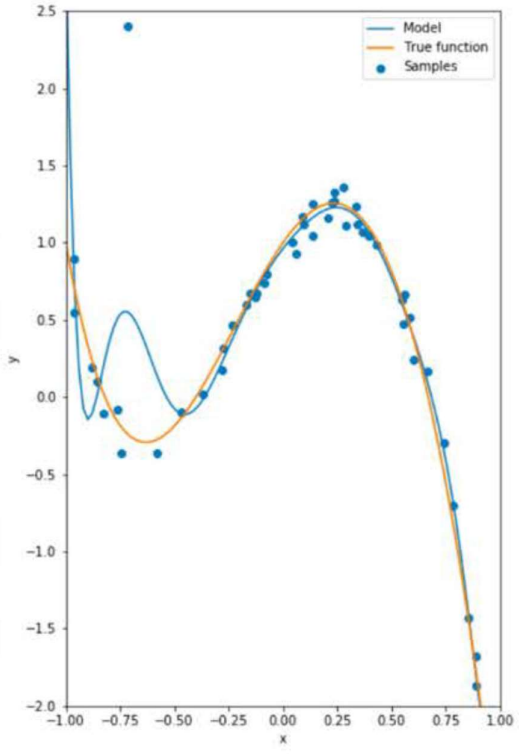
$$\hat{y} = 1 + 2\,x - 3\,x^2 - 2x^3 - 12\,x^4 - 40x^5 + 80x^6 + 71x^7 - 141x^8 - 38x^9 + 75x^{10}$$

Alpha	x	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^{10}
0	2	-3	-2	-12	-40	80	71	-141	-38	75
0.001	2	-3	-7	5	4	-6	4	-4	4	6
0.01	1	-2	-5	-0.04	0.15	-1	1	-0.5	0.3	1
1	0.5	-1	-1	-0.614	0.70	-0.38	-0. 56	-0.21	-0.5	-0.1
10	0	-0.5	-0.3	-0.37	-0.30	-0.30	-0.22	-0.22	-0.22	-0.17

If alpha is zero, the over-fitting is evident.

Ridge Regression

alpha
0
0.001
0.01
1
10

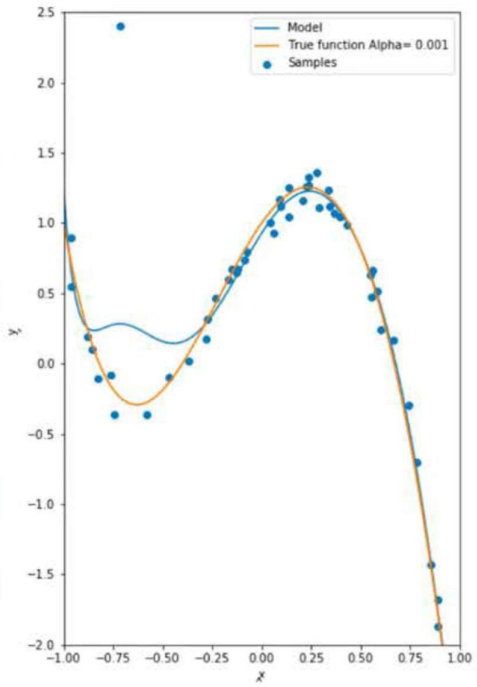


COGNITIVE CLASS.ai

For alpha equal to 0.001, the over fitting begins to subside.

Ridge Regression

alpha
0
0.001
0.01
1
10

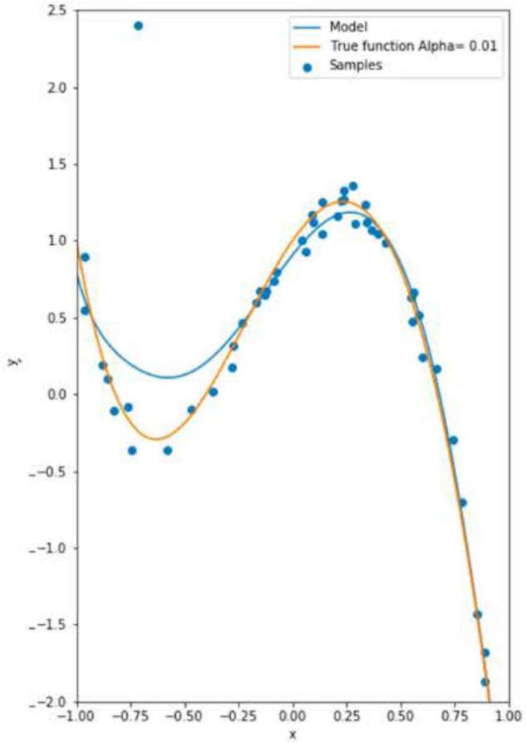


COGNITIVE CLASS.ai

For alpha equal to 0.01, the estimated function tracks the actual function.

Ridge Regression

alpha
0
0.001
0.01
1
10

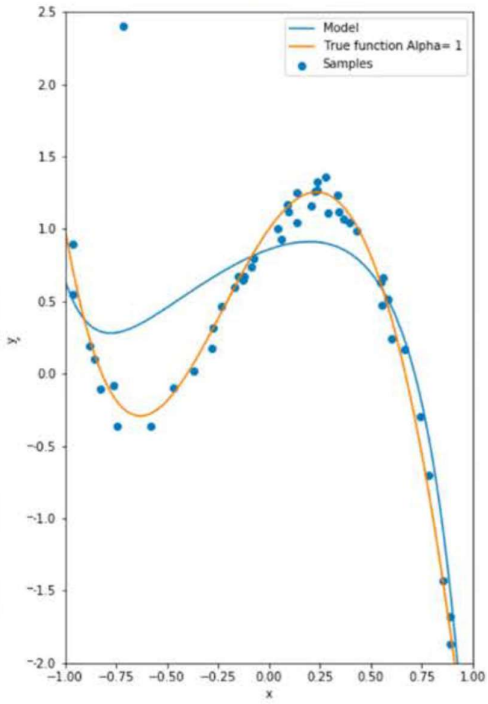


COGNITIVE CLASS.ai

When alpha equals 1, we see the first signs of under-fitting. The estimated function does not have enough flexibility.

Ridge Regression

alpha
0
0.001
0.01
1
10

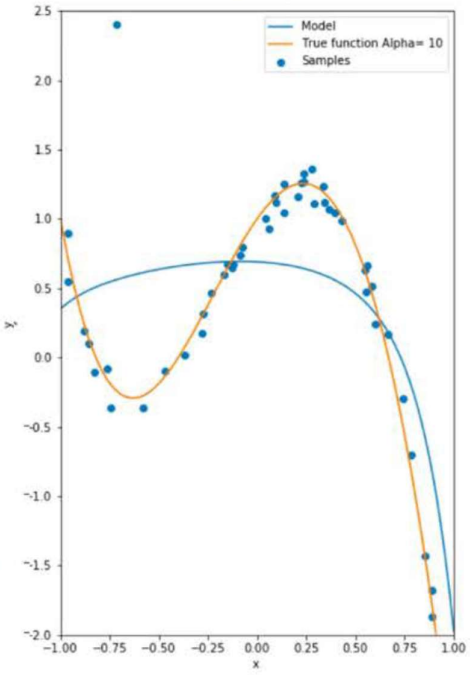


COGNITIVE CLASS.ai

At alpha equals to 10, we see extreme under-fitting; it does not even track the two points.

Ridge Regression

alpha
0
0.001
0.01
1
10



COGNITIVE CLASS.ai

In order to select alpha we use cross-validation.

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. The parameter alpha is one of the arguments of the constructor.

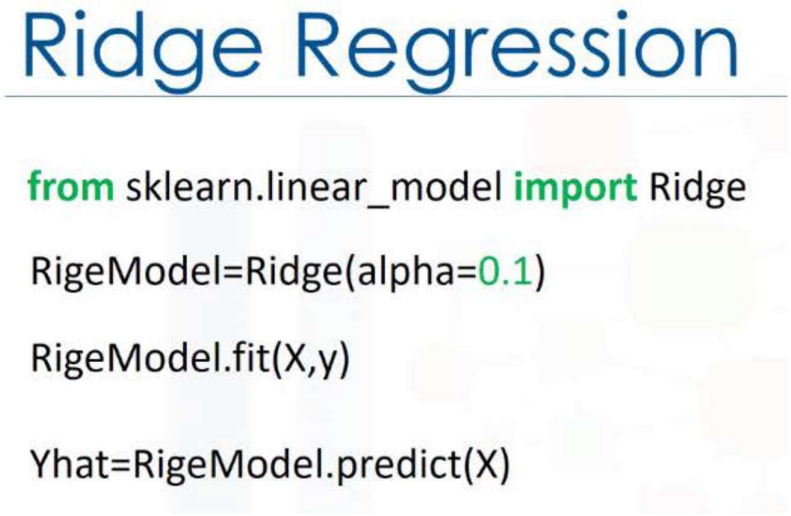
```
RigeModel=Ridge{alpha=O.l)
```

We train the model using the fit method.

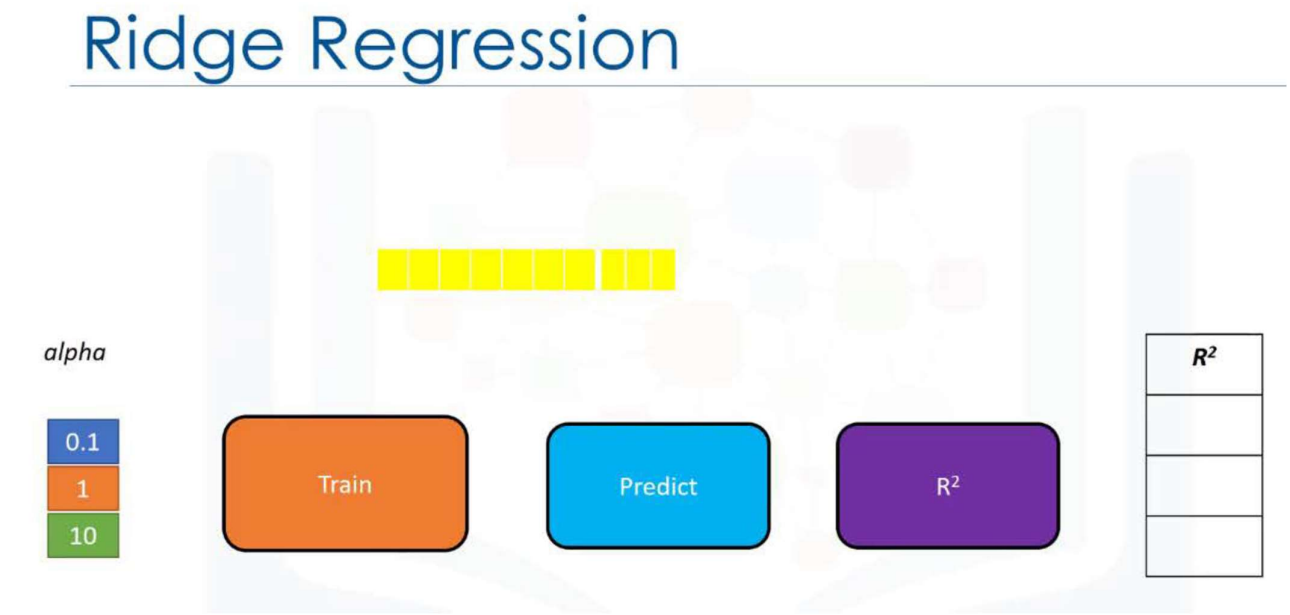
```
RigeModel.fit{X,y)
```

To make a prediction, we use the predict method.

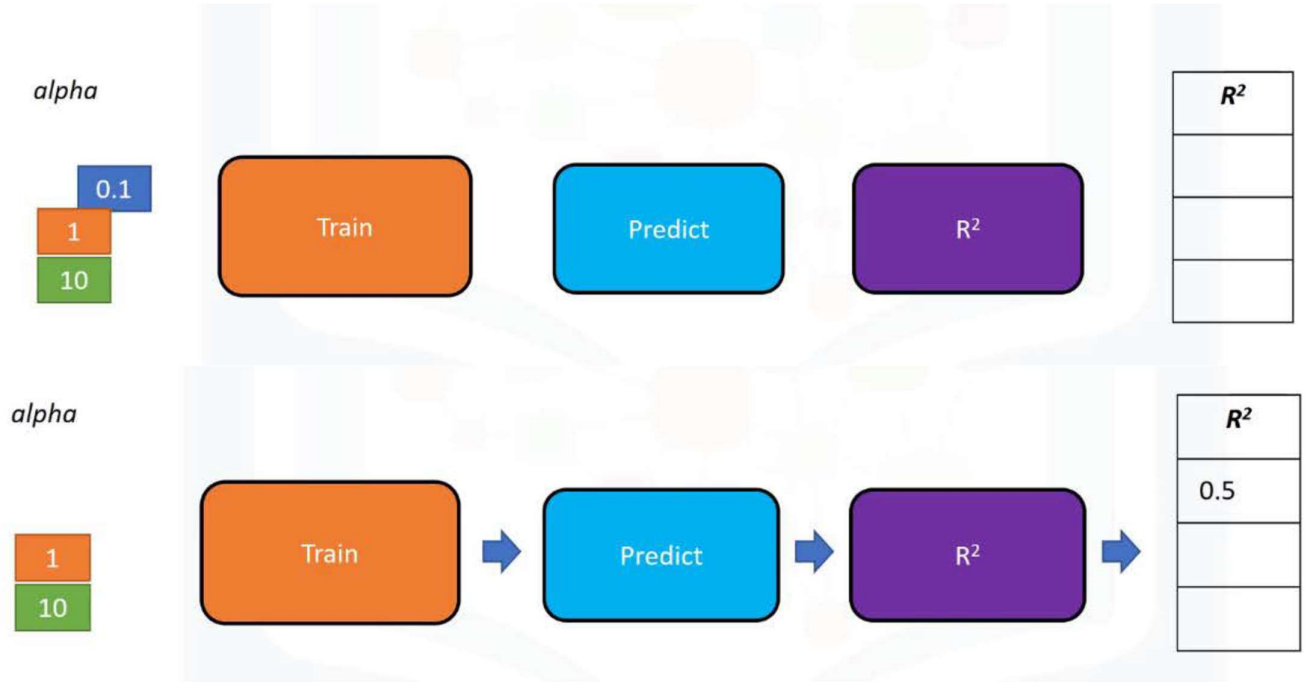
```
Yhat=RigeModel.predict{X)
```



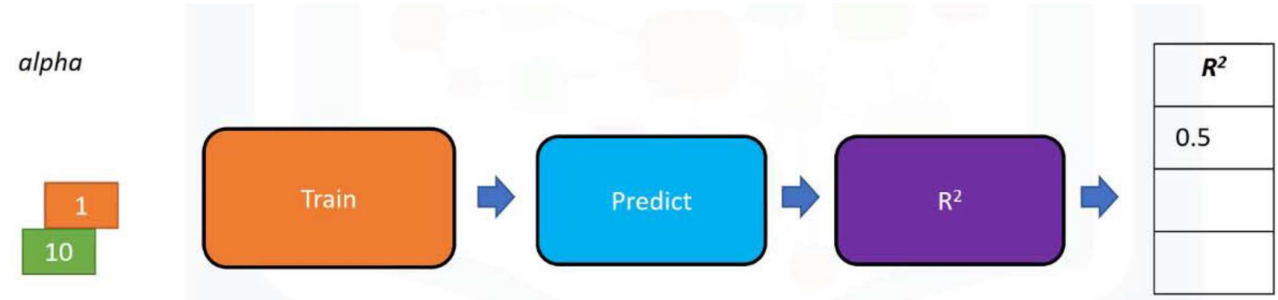
In order to determine the parameter alpha, we use some data for training. We use a second set called validation data; this is similar to test data, but it is used to select parameters like alpha.

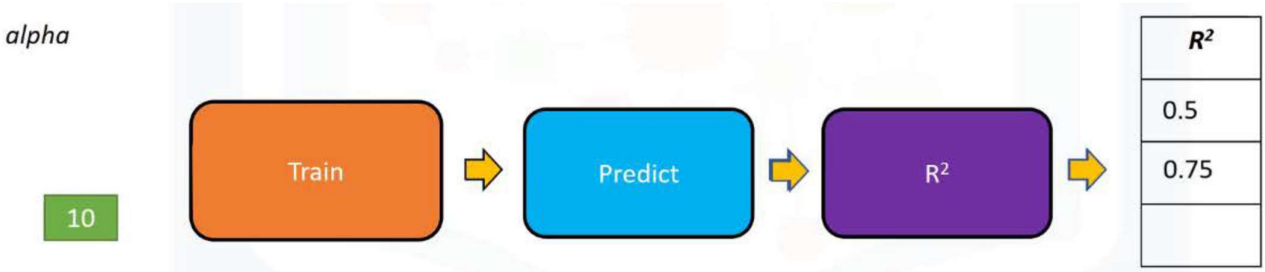


We start with a small value of alpha, we train the model, make a prediction using the validation data, then calculate the R squared and store the values.

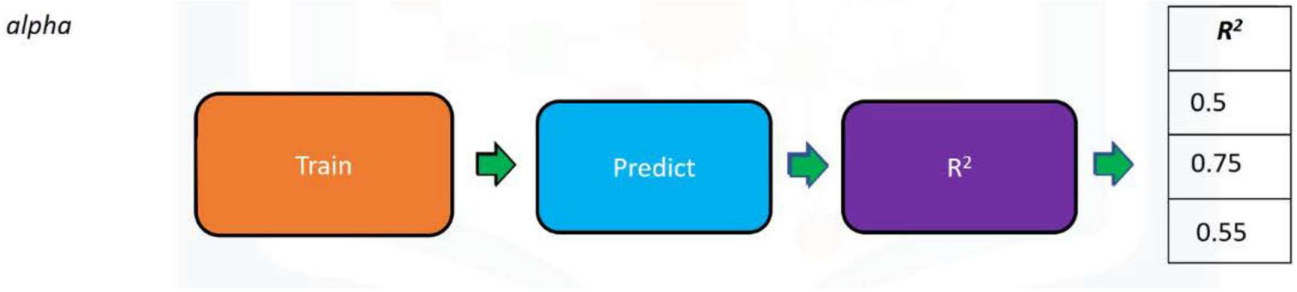


Repeat the value for a larger value of alpha. We train the model again, make a prediction using the validation data, then calculate the R squared and store the values of R squared.

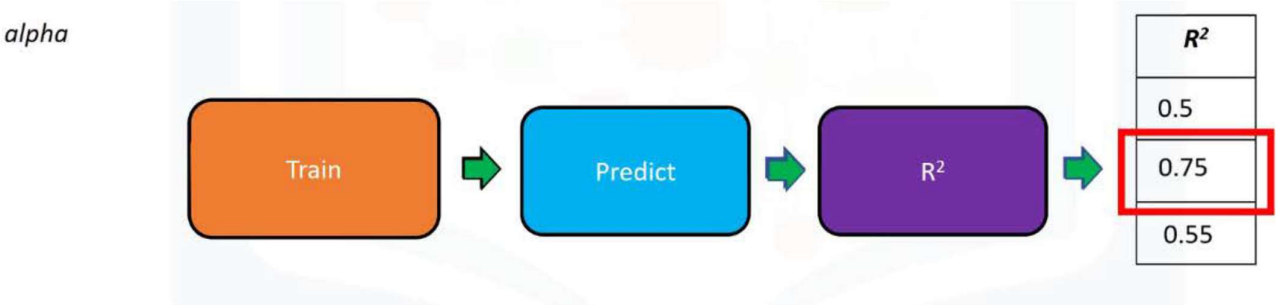




We repeat the process for a different alpha value, training the model, and making a prediction.



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.



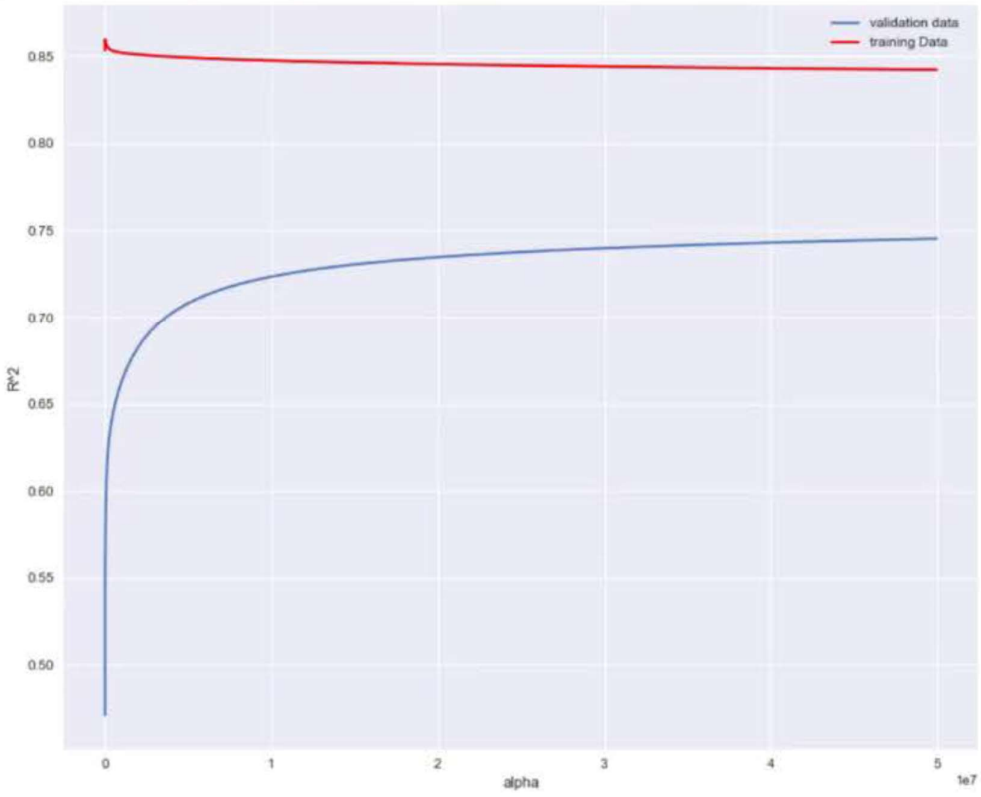
The Overfitting problem is even worse if we have lots of features.

The following plot shows the different values of R squared on the vertical access. The horizontal axis represent 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.

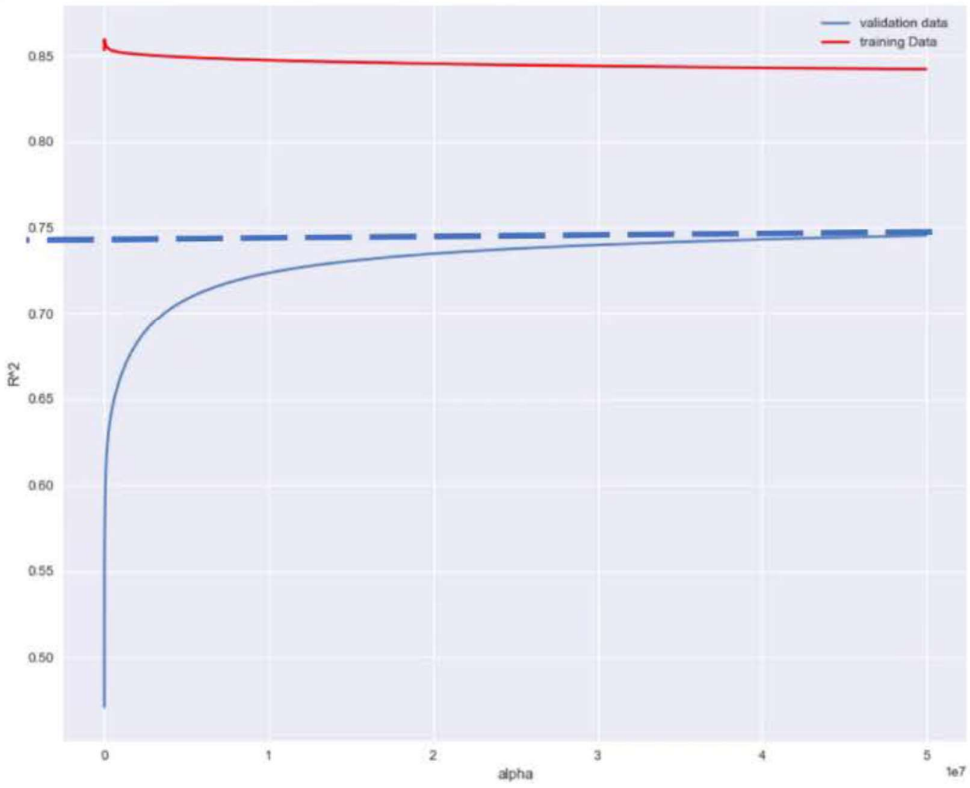
Ridge Regression



COGNITIVE CLASS

We see as the value for alpha increases, the value the R squared increases and converges at approximately 0.75.

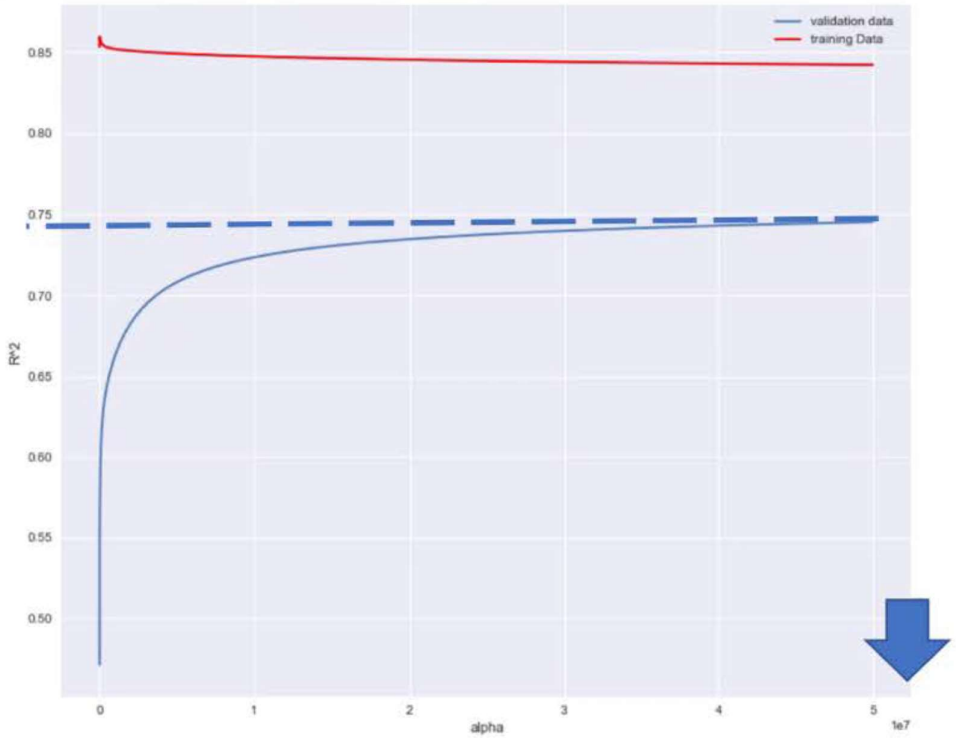
Ridge Regression



COGNITIVE CLASS

In this case, we select the maximum value of alpha because running the experiment for higher values of alpha have little impact.

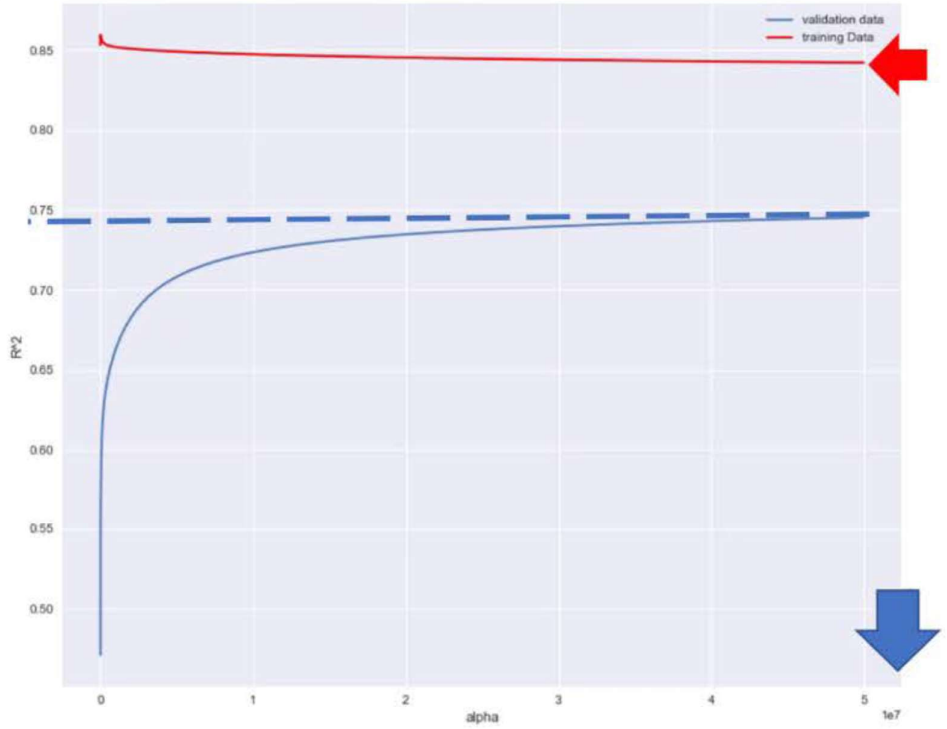
Ridge Regression



COGNITIVE CLASS.ai

Conversely, as alpha increases, the R squared on the training 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. See the lab on how to generate this plot.

Ridge Regression



COGNITIVE CLASS.ai

Grid Search

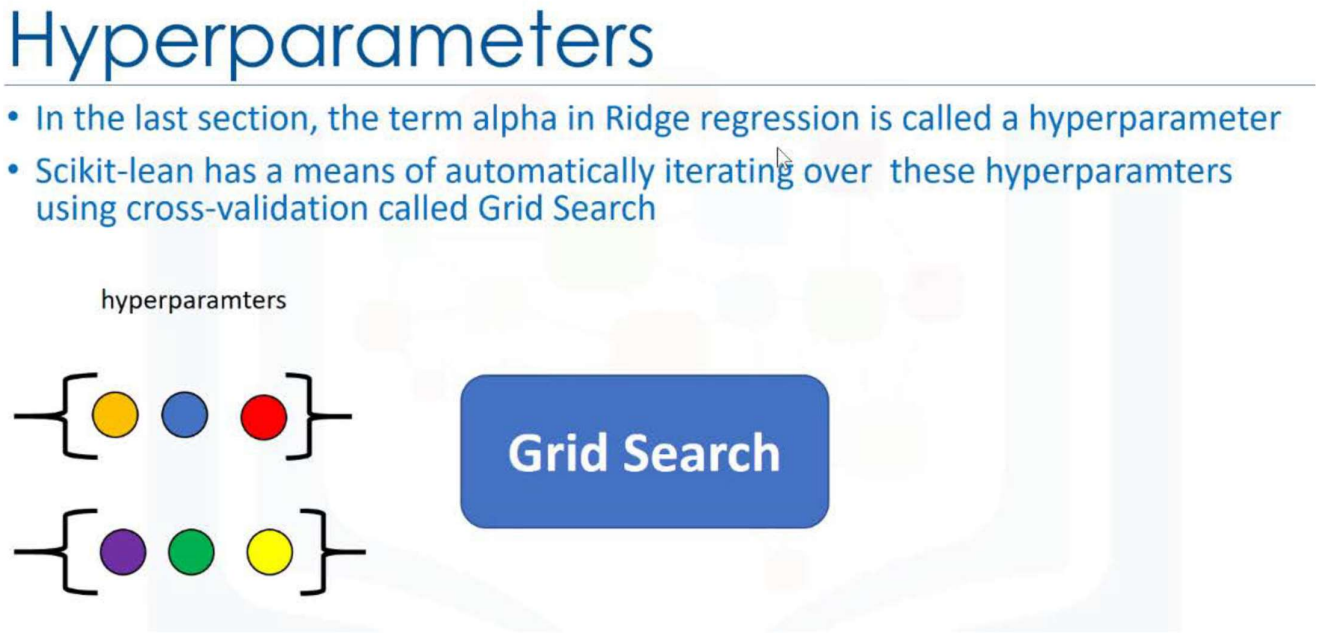
Grid search allows us to scan through multiple free parameters with few lines of code.

Parameters like the alpha term discussed in the previous video are not part of the fitting or training process.

These values are called hyperparameters.

Scikit-learn has a means of automatically iterating over these hyperparameters using cross-validation.

This method is called Grid search.

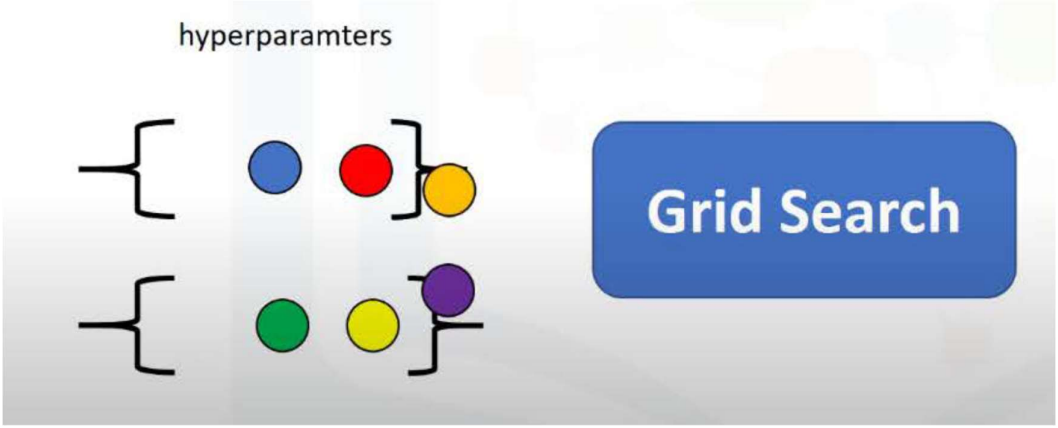


Grid search takes the model or objects you would like to train and different values of the hyperparameters.

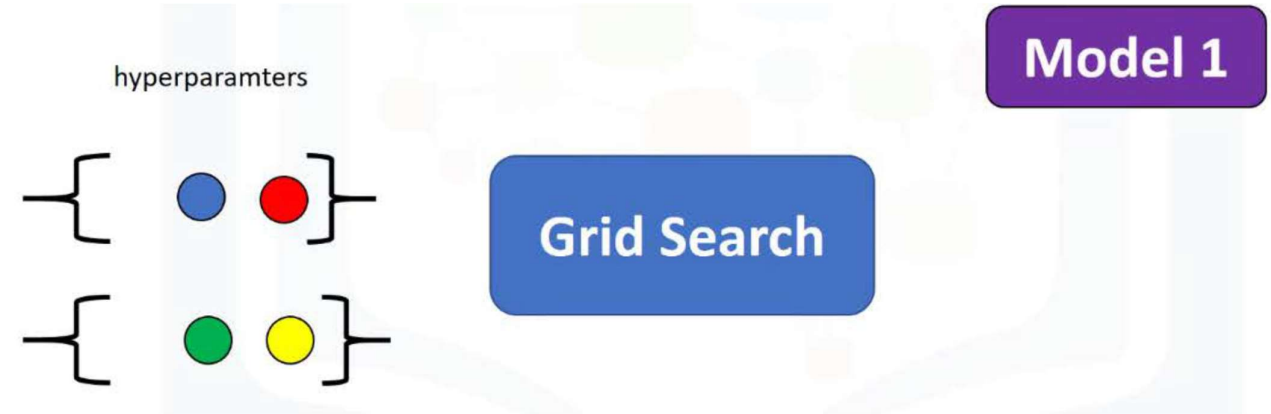
It then calculates the mean square error or R squared for various hyperparameter values, allowing you to choose the best values.

Let the small circles represent different hyperparameters.

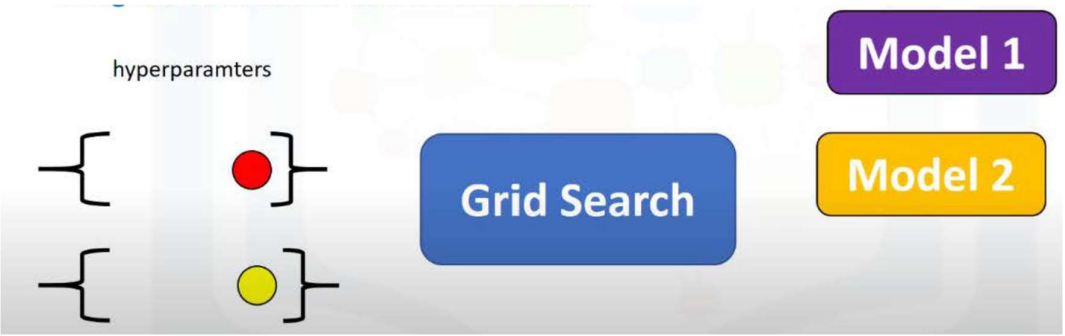
We start off with one value for hyperparameters and train the model.

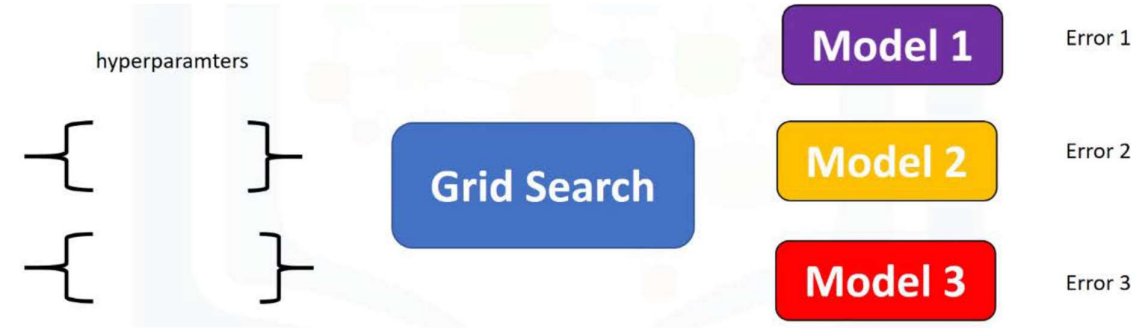


We use different hyperparameters to train the model.

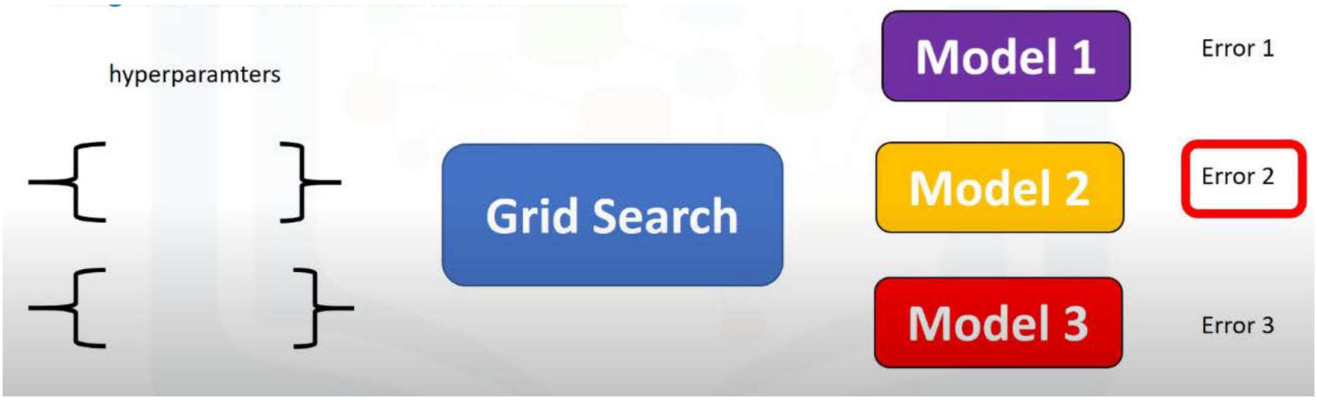


We continue the process until we have exhausted the different free parameter values. Each model produces an error.

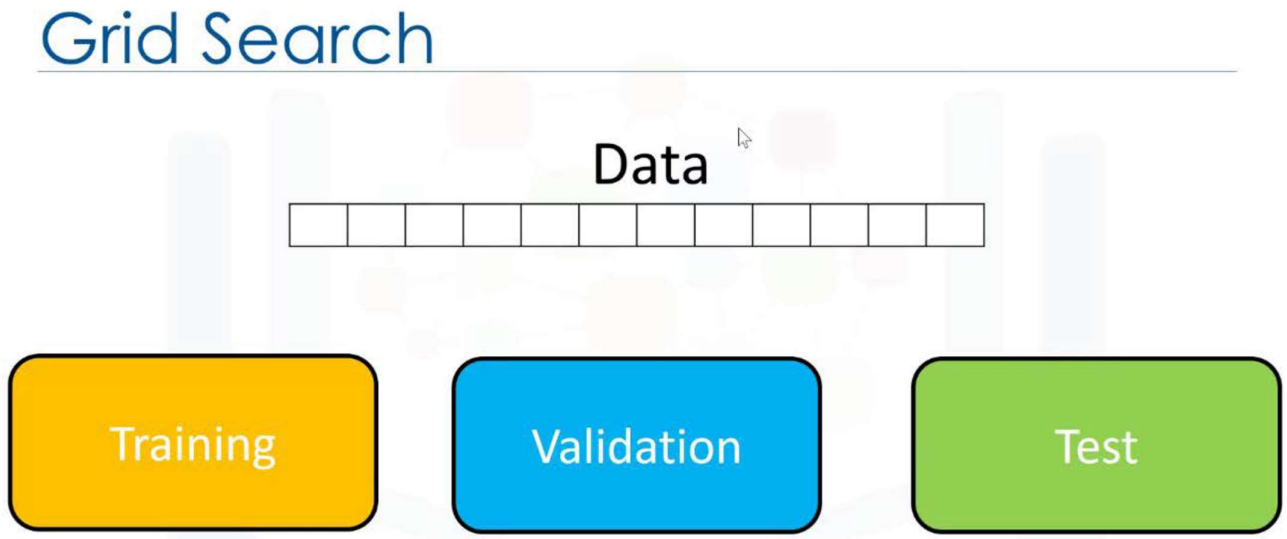




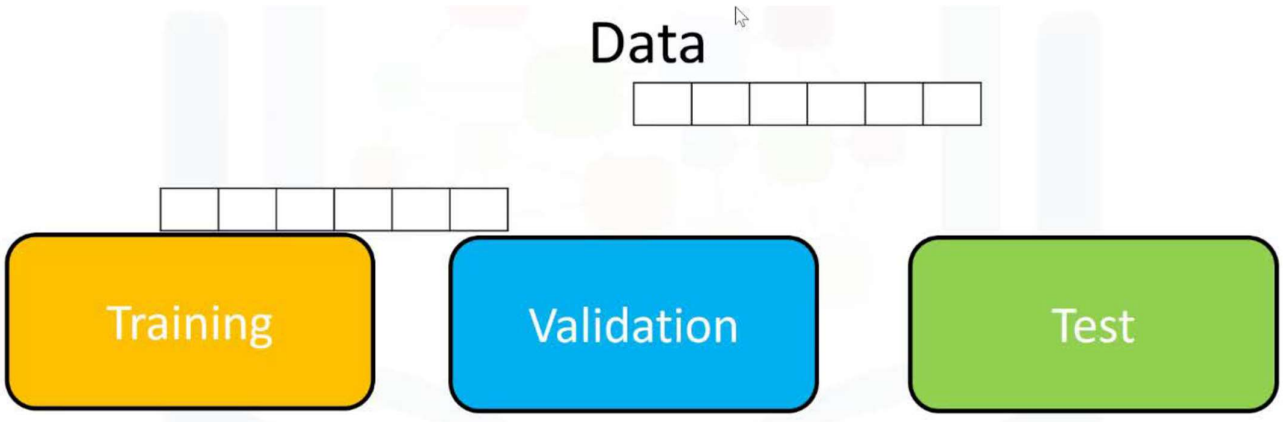
We select the hyperparameter that minimizes the error.

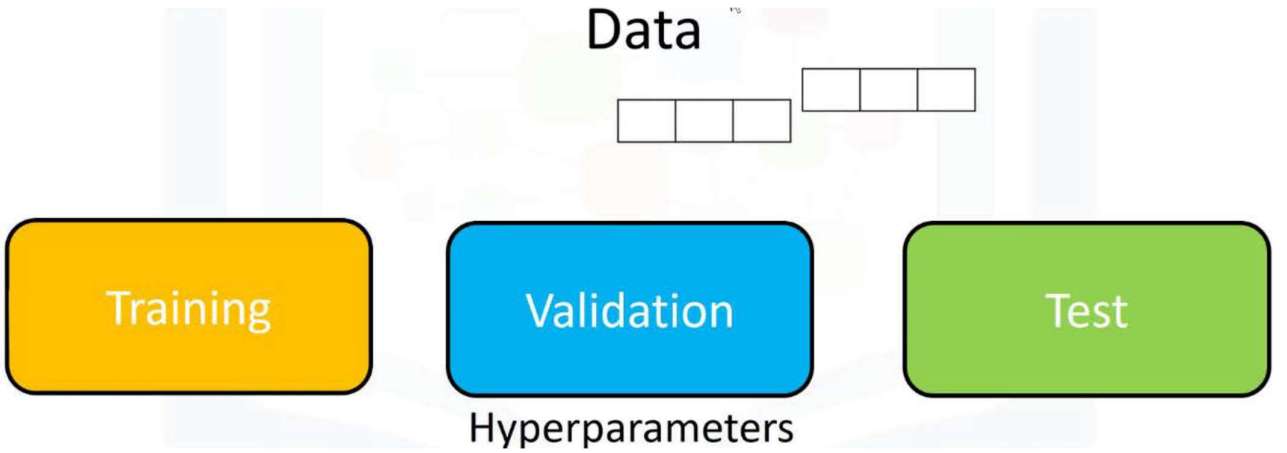


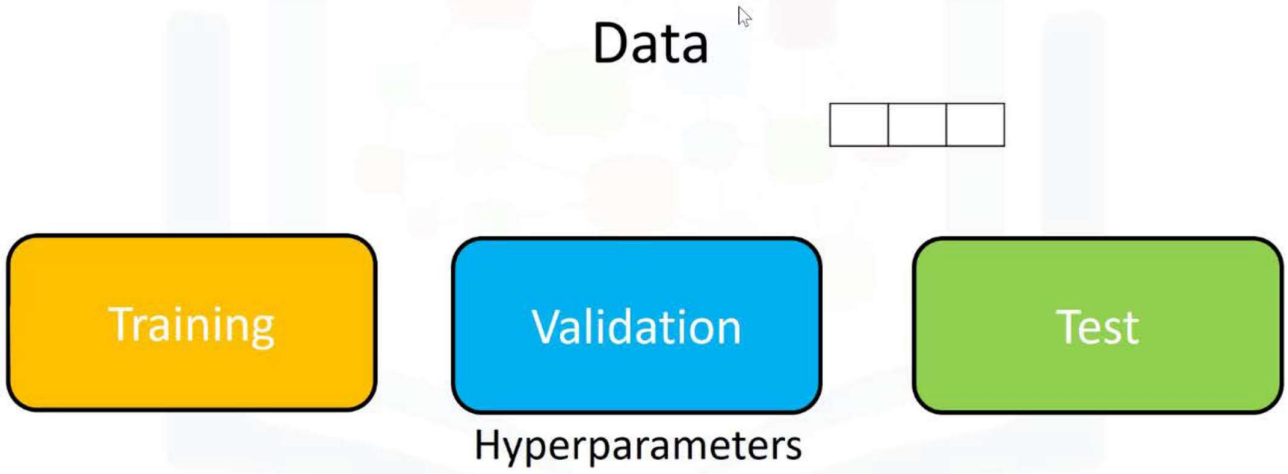
To select the hyperparameter, we split our dataset into three parts, the training set, validation set, and test set.



We train the model for different hyperparameters.







We use the R squared or mean square error for each model.

We select the hyperparameter that minimizes the mean squared error or maximizes the R squared on the validation set.

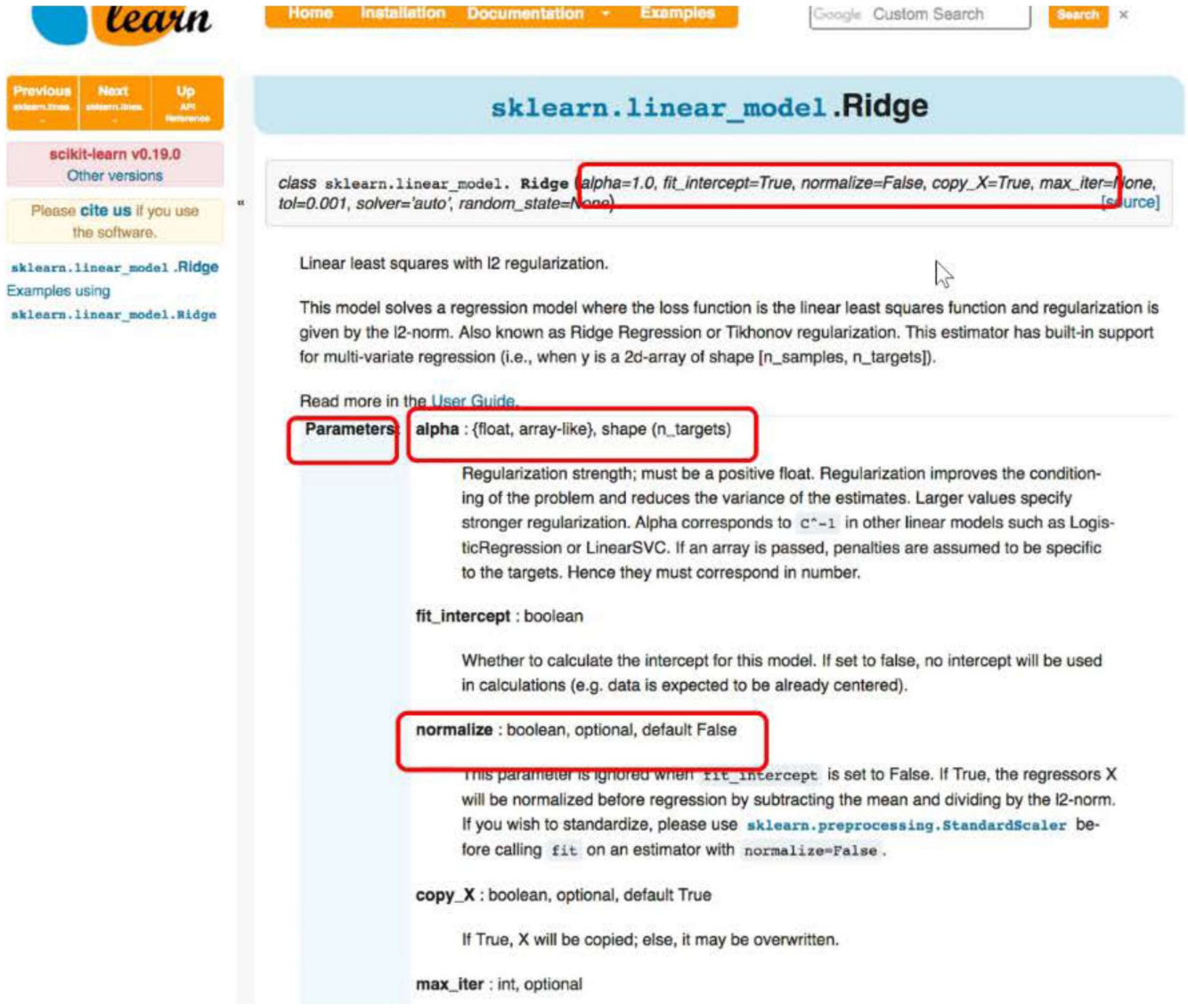
We finally test our model performance using the test data.

This is the scikit learn web-page where the object constructor parameters are given.

It should be noted that the attributes of an object are also called parameters.

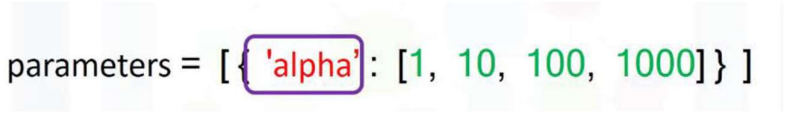
We will not make the distinction even though some of the options are not hyperparameters per say.

In this module, we will focus on the hyperparameter alpha and the normalization parameter.

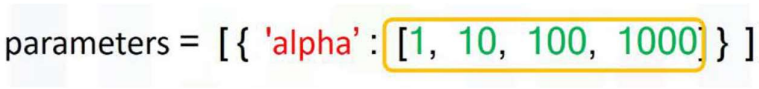


The value of your grid search is a Python list that contains a Python dictionary.

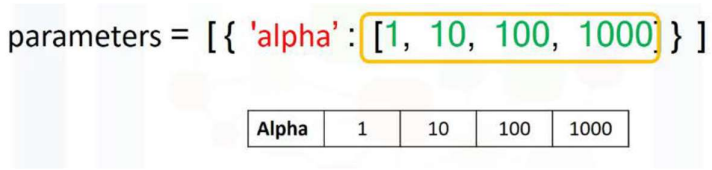
The key is the name of the free parameter.



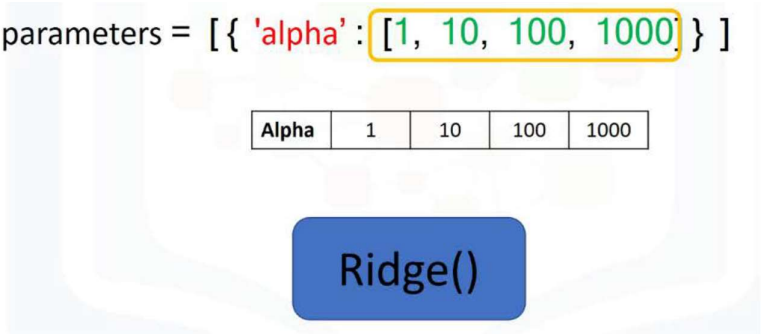
The value of the dictionary is the different values of the free parameter.



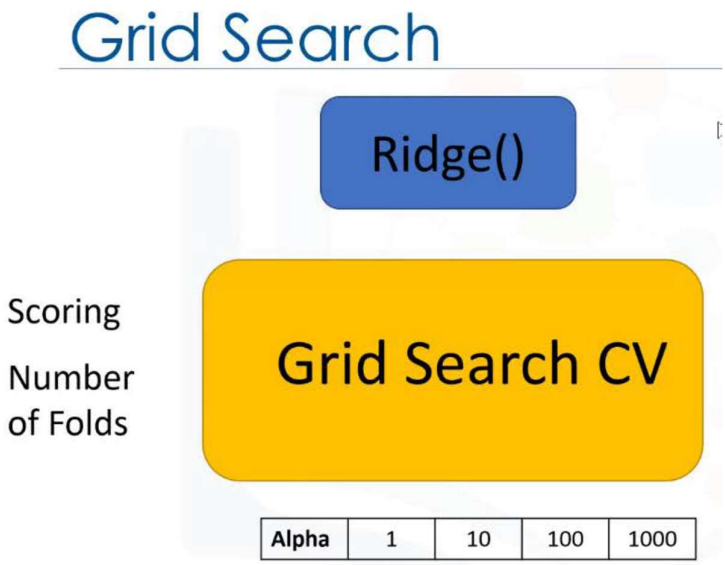
This can be viewed as a table with various free parameter values.



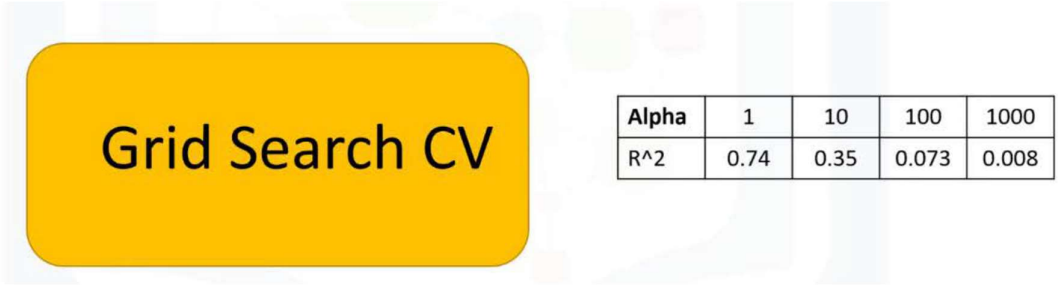
We also have the object or model.



The grid search takes on the scoring method, in this case 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 values;



in this case the R squared along with the free parameter values that have the best score.

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

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

RR=Ridge()

Grid1 = GridSearchCV(RR, parameters1,cv=4)

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

Grid1.best_estimator_

scores  = Grid1.cv_results_
scores['mean_test_score']
```

First, we import the libraries we need including Grid Search CV, the dictionary of parameter values.

```
from sklearn.linear _model import Ridge
from sklearn.model_selection import GridSearchCV

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

We create a ridge regression object or model.

```
RR=Ridge()
```

We then create a Grid Search CV object; the inputs are the ridge regression object, the parameter values and the number of folds.

```
Grid1 == GridSearchCV(RR, parametersl,cv==4)
```

We will use R squared; this is the default scoring method.
We fit the object.

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

We can find the best values for the free parameters using the attribute best estimator.

```
Grid1.best estimator_
```

We can also get information like the mean score on the validation data using the attribute cv result.

```
scores = Grid1.cv_results_
scores['mean_test_score']
```

array([0.66549413, 0.66554568, 0.66602936, 0.66896822, 0.67334636, 0.65781884, 0.65781884])

One of the advantages of Grid search is how quickly we can test multiple parameters.

For example, Ridge regression has the option to normalize the data.
To see how to standardize, see Module 4.

The term alpha is the first element in the dictionary, the second element is the normalize option.

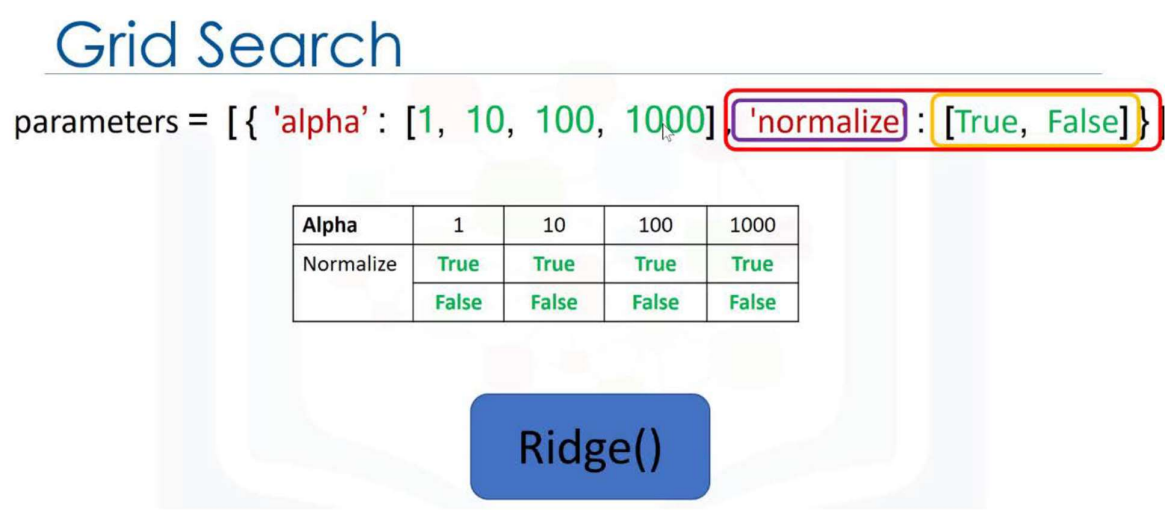
The key is the name of the parameter.

The value is the different options, in this case, because we can either normalize the data or not, the values are true or false, respectively.

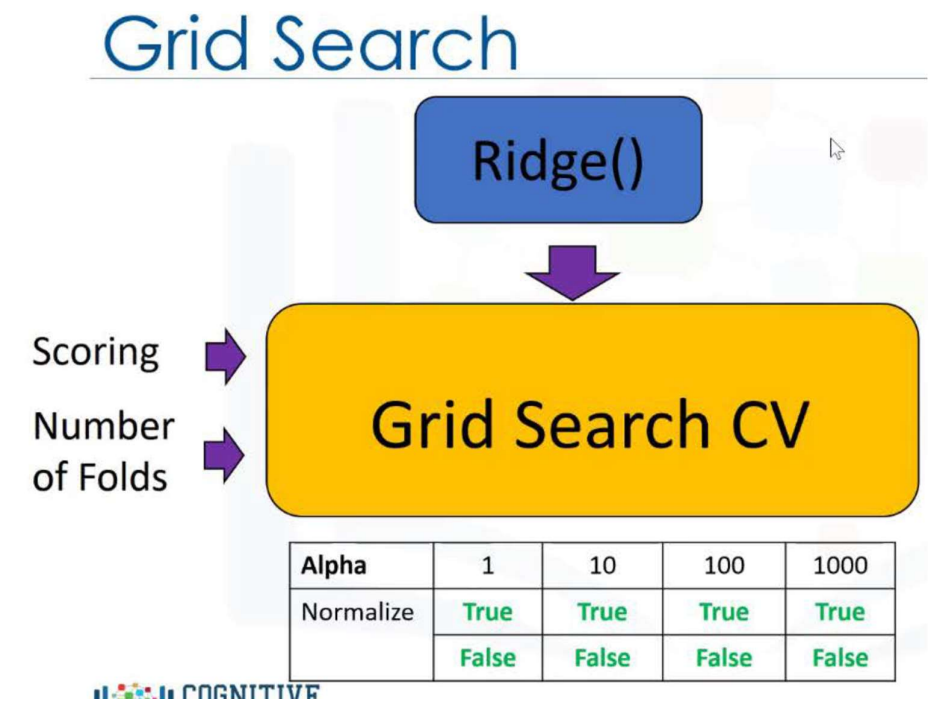
The Dictionary is a table or grid that contains two different values.

Alpha	1	10	100	1000
Normalize	True	True	True	True
	False	False	False	False

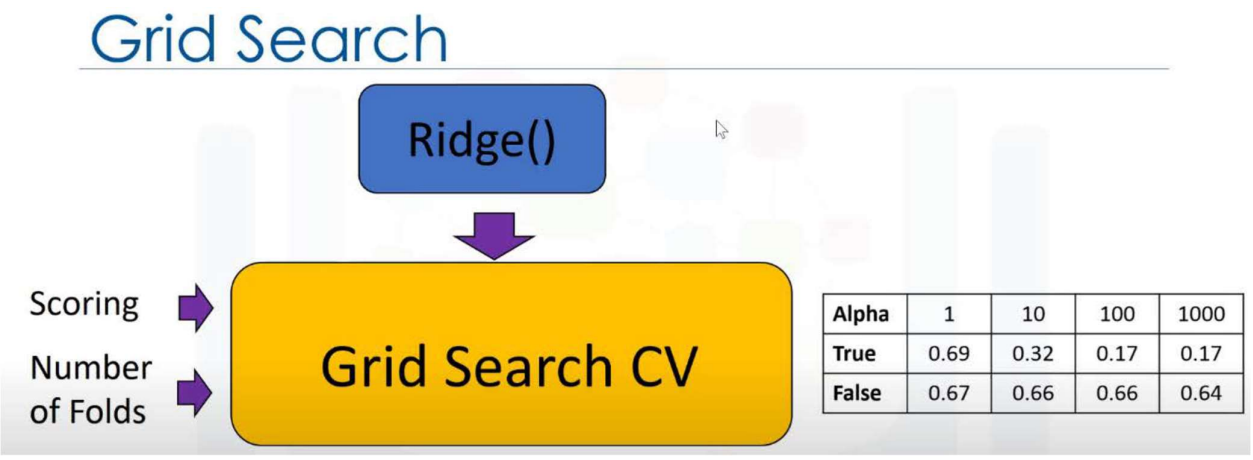
As before, we need the ridge regression object or model.



The procedure is similar, except that we have a table or grid of different parameter values.



The output is the score for all the different combinations of parameter values.



The code is also similar.

```
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_
```

The dictionary contains the different free parameter values.

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

We can find the best value for the free parameters.

Grid1.best_estimator_

The resulting scores of the different free parameters are stored in this dictionary: Grid1.cv_results_.

scores = Grid1.cv_results_

We can print out the score for the different free parameter values.

```
for param,mean_val, mean_test  inzip(scores['params'],scores['mean_test_score'],scores['mean_train_score']):

        print(param, "R^2 on test data:", mean_val,"R^2 on train data:" ,mean_test)
```

```
{'alpha': 0.001, 'normalize': True} R^2 on tesst data: 0.66605547293 R^2 on train data: 0.814001968709
{'alpha': 0.001, 'normalize': False} R^2 on tesst data: 0.665488366584 R^2 on train data: 0.814002698797
{'alpha': 0.1, 'normalize': True} R^2 on tesst data: 0.694175625356 R^2 on train data: 0.810546768311
{'alpha': 0.1, 'normalize': False} R^2 on tesst data: 0.665488937796 R^2 on train data: 0.814002698794
{'alpha': 1, 'normalize': True} R^2 on tesst data: 0.690486934584 R^2 on train data: 0.749104440368
{'alpha': 1, 'normalize': False} R^2 on tesst data: 0.665494127178 R^2 on train data: 0.814002698472
{'alpha': 10, 'normalize': True} R^2 on tesst data: 0.321376875232 R^2 on train data: 0.341856042902
{'alpha': 10, 'normalize': False} R^2 on tesst data: 0.665545680812 R^2 on train data: 0.8140026666
{'alpha': 100, 'normalize': True} R^2 on tesst data: 0.0170551710263 R^2 on train data: 0.0496044796826
{'alpha': 100, 'normalize': False} R^2 on tesst data: 0.666029359996 R^2 on train data: 0.813999791851
{'alpha': 1000, 'normalize': True} R^2 on tesst data: -0.0301961745066 R^2 on train data: 0.005184451599
{'alpha': 1000, 'normalize': False} R^2 on tesst data: 0.668968215369 R^2 on train data: 0.813870488264
{'alpha': 10000, 'normalize': True} R^2 on tesst data: -0.0351687400461 R^2 on train data: 0.000520784757979
{'alpha': 10000, 'normalize': False} R^2 on tesst data: 0.673346359342 R^2 on train data: 0.812583743226
{'alpha': 100000, 'normalize': True} R^2 on tesst data: -0.0356685844558 R^2 on train data: 5.2101975528e-05
{'alpha': 100000, 'normalize': False} R^2 on tesst data: 0.657818838432 R^2 on train data: 0.789541446486
{'alpha': 100000, 'normalize': True} R^2 on tesst data: -0.0356685844558 R^2 on train data: 5.2101975528e-05
{'alpha': 100000, 'normalize': False} R^2 on tesst data: 0.657818838432 R^2 on train data: 0.789541446486
```

The parameter values are stored as shown here.

```
for param,mean_val, mean_test  inzip(scores['params'],scores['mean_test_score'],scores['mean_train_score']):

        print(param, "R^2 on test data:", mean_val,"R^2 on train data:" ,mean_test)
```

```
{'alpha': 0.001, 'normalize': True} R^2 on tesst data: 0.66605547293 R^2 on train data: 0.814001968709
{'alpha': 0.001, 'normalize': False} R^2 on tesst data: 0.665488366584 R^2 on train data: 0.814002698797
{'alpha': 0.1, 'normalize': True} R^2 on tesst data: 0.694175625356 R^2 on train data: 0.810546768311
{'alpha': 0.1, 'normalize': False} R^2 on tesst data: 0.665488937796 R^2 on train data: 0.814002698794
{'alpha': 1, 'normalize': True} R^2 on tesst data: 0.690486934584 R^2 on train data: 0.749104440368
{'alpha': 1, 'normalize': False} R^2 on tesst data: 0.665494127178 R^2 on train data: 0.814002698472
{'alpha': 10, 'normalize': True} R^2 on tesst data: 0.321376875232 R^2 on train data: 0.341856042902
{'alpha': 10, 'normalize': False} R^2 on tesst data: 0.665545680812 R^2 on train data: 0.8140026666
{'alpha': 100, 'normalize': True} R^2 on tesst data: 0.0170551710263 R^2 on train data: 0.0496044796826
{'alpha': 100, 'normalize': False} R^2 on tesst data: 0.666029359996 R^2 on train data: 0.813999791851
{'alpha': 1000, 'normalize': True} R^2 on tesst data: -0.0301961745066 R^2 on train data: 0.005184451599
{'alpha': 1000, 'normalize': False} R^2 on tesst data: 0.668968215369 R^2 on train data: 0.813870488264
{'alpha': 10000, 'normalize': True} R^2 on tesst data: -0.0351687400461 R^2 on train data: 0.000520784757979
{'alpha': 10000, 'normalize': False} R^2 on tesst data: 0.673346359342 R^2 on train data: 0.812583743226
{'alpha': 100000, 'normalize': True} R^2 on tesst data: -0.0356685844558 R^2 on train data: 5.2101975528e-05
{'alpha': 100000, 'normalize': False} R^2 on tesst data: 0.657818838432 R^2 on train data: 0.789541446486
{'alpha': 100000, 'normalize': True} R^2 on tesst data: -0.0356685844558 R^2 on train data: 5.2101975528e-05
{'alpha': 100000, 'normalize': False} R^2 on tesst data: 0.657818838432 R^2 on train data: 0.789541446486
```

See the course labs for more examples.