

DATA ANALYSIS WITH PYTHON(2)

<https://www.coursera.org/learn/data-analysis-with-python/home/welcome>

- Importing dataset
 - o Data Acquisition
 - o Basic Insight of Dataset

- Data wrangling
 - o Identify and handle missing values
 - Identify missing values
 - Deal with missing values
 - Correct data format
 - o Data standardization
 - o Data Normalization (centering/scaling)
 - o Binning
 - o Indicator variable

- Exploratory data analysis
 - o Import Data from Module
 - o Analyzing Individual Feature Patterns using Visualization
 - o Descriptive Statistical Analysis
 - o Basics of Grouping
 - o Correlation and Causation
 - o ANOVA

- Model development
 - o Linear Regression and multiple linear regression
 - o Model Evaluation using Visualization
 - o Polynomial Regression and Pipeline
 - o Measure for in sample evaluation
 - o Prediction and sample evaluation

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

Scientifics Computing Libraries in Python

1. Scientifics Computing Libraries



Pandas

(Data structures & tools)



NumPy

(Arrays & matrices)



SciPy

(Integrals, solving differential equations, optimization)

Visualization Libraries in Python

2. Visualization Libraries



Matplotlib

(plots & graphs, most popular)





Seaborn

(plots : heat maps, time series, violin plots)

Algorithmic Libraries in Python

3. **Algorithmic libraries**

-  **Scikit-learn**
(Machine Learning : regression, classification,...)
-  **Statsmodels**
(Explore data, estimate statistical models, and perform statistical tests.)

Play

Data acquisition

Importing Data

- Process of loading and reading data into Python from various resources.
- **Two important properties:**
 - **Format**
 - various formats: .csv, .json, .xlsx, .hdf
 - **File Path of dataset**
 - Computer: /Desktop/mydata.csv
 - Internet: <https://archive.ics.uci.edu/autos/imports-85.data>

Play

Import panda as pd

```
# Reading csv file
url = http://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.data
df = pd.read_csv(url, header=None)

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

```


headers = ["symboling", "normalized-losses", .....]
df.columns = headers

df.head()

# Exporting panda dataframe to csv after working on it
Path = "C:\Users\...automobile.csv"
df.to_csv(path)

```

Exporting to different formats in Python



Data Format	Read	Save
csv	pd.read_csv()	df.to_csv()
json	pd.read_json()	df.to_json()
Excel	pd.read_excel()	df.to_excel()
sql	pd.read_sql()	df.to_sql()

CHECK DATA TYPES

Basic Insights of Dataset - Data Types

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.

Basic Insights of Dataset - Data Types

- 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
```

`dataframe.describe()`

- Returns a statistical summary

```
df.describe()
```

	symboling	wheel-base	length	width	height	curb-weight	engine-size	compression-ratio	city-mpg	highway-mpg
count	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000
mean	0.834146	98.756585	174.049268	65.907805	53.724878	2555.565854	126.907317	10.142537	25.219512	30.751220
std	1.245307	6.021776	12.337289	2.145204	2.443522	520.680204	41.642693	3.972040	6.542142	6.886443
min	-2.000000	86.600000	141.100000	60.300000	47.800000	1488.000000	61.000000	7.000000	13.000000	16.000000
25%	0.000000	94.500000	166.300000	64.100000	52.000000	2145.000000	97.000000	8.600000	19.000000	25.000000
50%	1.000000	97.000000	173.200000	65.500000	54.100000	2414.000000	120.000000	9.000000	24.000000	30.000000
75%	2.000000	102.400000	183.100000	66.900000	55.500000	2935.000000	141.000000	9.400000	30.000000	34.000000
max	3.000000	120.900000	208.100000	72.300000	59.800000	4066.000000	326.000000	23.000000	49.000000	54.000000

```
# full summary statistics
df.describe(include="all")
```

WEEK 2

Data Pre-processing

The process of converting or mapping data from the initial “raw” form into another format, in order to prepare the data for further analysis.

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

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

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

Deal with missing data

How to deal with missing data?

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

Whole columns should be dropped only if most entries in the column are empty. In our dataset, none of the columns are empty enough to drop entirely. We have some freedom in choosing which method to replace data; however, some methods may

seem more reasonable than others. We will apply each method to many different columns:

Replace by mean:

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

Replace by frequency:

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

Drop the whole row:

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

How to drop missing values in Python

- 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)
```


Don't Forget

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



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

<http://pandas.pydata.org/>

How to replace missing values in Python

Use `dataframe.replace(missing_value, new_value):`

normalized-losses	make		normalized-losses	make
...
164	audi		164	audi
164	audi		164	audi
NaN	audi	→	162	audi
158	audi		158	audi
...

```
mean = df["normalized-losses"].mean()
```

```
df["normalized-losses"].replace(np.nan, mean)
```

COGNITIVE



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

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.

Methods of normalizing data

Several approaches for normalization:

①

$$x_{new} = \frac{x_{old}}{x_{max}}$$

Simple Feature scaling

②

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

Min-Max

③


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

Z-score

Simple Feature Scaling in Python

With Pandas:

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




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

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

Min-max in Python

With Pandas:

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




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

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

Z-score in Python

With Pandas:

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



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

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

Data Formatting

- Data are usually collected from different places and stored in different formats.
- Bringing data into a common standard of expression allows users to make meaningful comparison.

Non-formatted:

- confusing
- hard to aggregate
- hard to compare

City
NY
New York
N.Y
N.Y




City
New York
New York
New York
New York

Formatted:

- more clear
- easy to aggregate
- easy to compare

Applying calculations to an entire column

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



city-mpg
21
21
19
...

city-L/100km
11.2
11.2
12.4
...

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

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

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
```

Data Types in Python and Pandas

- There are many data types in pandas
- Objects : "A", "Hello"..
- Int64 : 1,3,5
- Float64 : 2.123, 632.31,0.12

Correcting data types

To *identify* data types:

- Use `dataframe.dtypes()` to identify data type.

To *convert* data types:

- Use `dataframe.astype()` to convert data type.

Example: convert data type to integer in column "price"

```
df["price"] = df["price"].astype("int")
```

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

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)
```

Categorical value to quantitative

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)



The diagram illustrates the process of creating dummy variables from a categorical variable. On the left, a table with a single column 'fuel' contains the values 'gas', 'diesel', 'gas', and 'gas'. A blue arrow points to the right, where a new table is shown. This table has two columns, 'gas' and 'diesel'. The rows correspond to the original data: the first row is 'gas' (1, 0), the second is 'diesel' (0, 1), and the next two rows are 'gas' (1, 0).

fuel
gas
diesel
gas
gas

gas	diesel
1	0
0	1
1	0
1	0

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

Exploratory Data Analysis (Week 3)

Exploratory Data Analysis (EDA)

- Preliminary step in data analysis to:
 - Summarize main characteristics of the data
 - Gain better understanding of the data set
 - Uncover relationships between variables
 - Extract important variables
- Question:
"What are the characteristics that have the most impact on the car price?"

Learning Objectives

In this module you will learn about:

- Descriptive Statistics
- GroupBy
- ANOVA
- Correlation
- Correlation - Statistics

Descriptive Statistics

- Describe basic features of data
- Giving short summaries about the sample and measures of the data

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

Descriptive Statistics - Value_Counts()

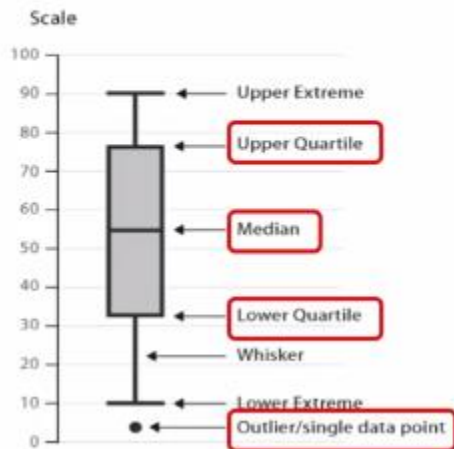
- summarize the categorical data is by using the **value_counts()** method

```
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'
```

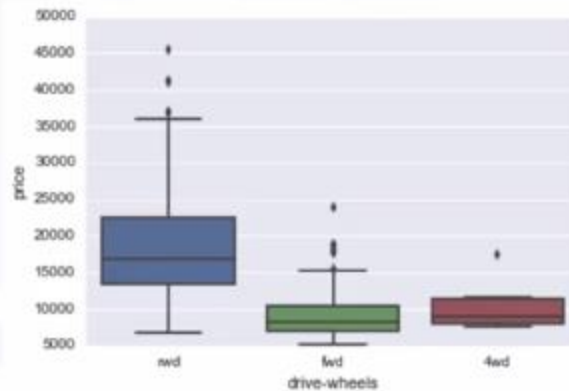
	value_counts
drive-wheels	
fwd	118
rwd	75
4wd	8

Descriptive Statistics - Box Plots



Box Plot - Example

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



Descriptive Statistics - Scatter Plot

- Each observation represented as a point.
 - Scatter plot Show the relationship between two variables.
1. Predictor/independent variables on x-axis.
 2. Target/dependent variables on y-axis.

Scatterplot - Example

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

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



Grouping data

- Use Panda **dataframe.groupby()** method:
 - Can be applied on categorical variables
 - Group data into categories
 - Single or multiple variables

Groupby()- Example

```
df_test = df[['drive-wheels', 'body-style', 'price']]
df_grp = df_test.groupby(['drive-wheels', 'body-style'], as_index=False).mean()
df_grp
```



	drive-wheels	body-style	price
0	4wd	hatchback	7603.000000
1	4wd	sedan	12647.333333
2	4wd	wagon	9095.750000
3	fwd	convertible	11595.000000
4	fwd	hardtop	8249.000000
5	fwd	hatchback	8396.387755
6	fwd	sedan	9811.800000
7	fwd	wagon	9997.333333
8	rwd	convertible	23949.600000
9	rwd	hardtop	24202.714286
10	rwd	hatchback	14337.777778
11	rwd	sedan	21711.833333
12	rwd	wagon	16994.222222

Pandas method - Pivot()

- One variable displayed along the columns and the other variable displayed along the rows.

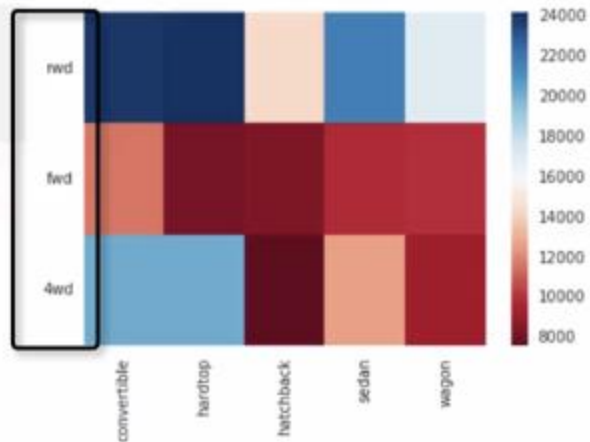
```
df_pivot = df_grp.pivot(index= 'drive-wheels', columns='body-style')
```

	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

Heatmap

- Plot target variable over multiple variables

```
plt.pcolor(df_pivot, cmap='RdBu')  
plt.colorbar()  
plt.show()
```



Correlation

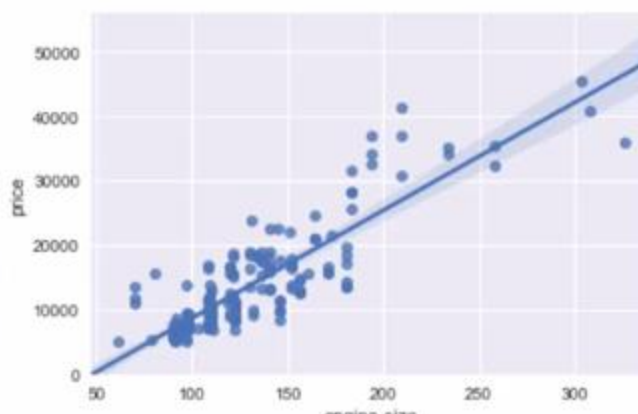
What is Correlation?

- Measures to what extent different variables are interdependent.
- For example:
 - Lung cancer → Smoking
 - Rain → Umbrella
- Correlation doesn't imply causation.

Correlation - Positive Linear Relationship

- Correlation between two features (engine-size and price).

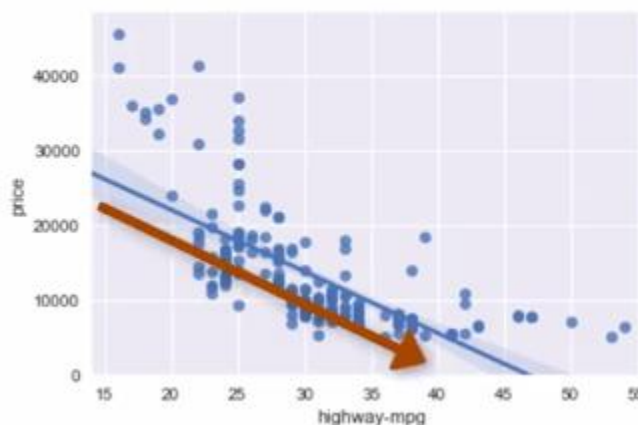
```
sns.regplot(x= "engine-size", y= "prices", data=df)  
plt.ylim(0,)
```



Correlation - Negative Linear Relationship

- Correlation between two features (highway-mpg and price).

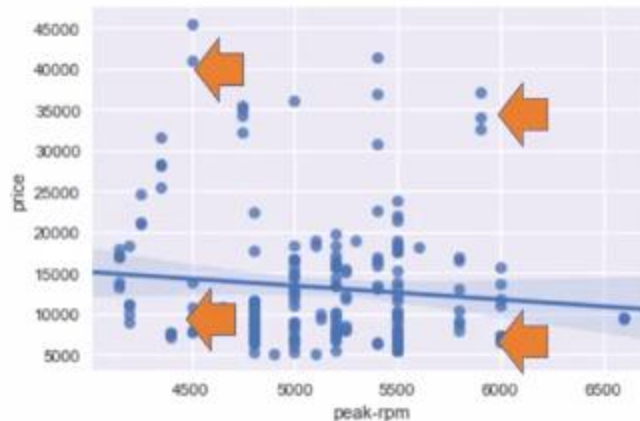
```
sns.regplot(x= "highway-mpg", y= "prices", data=df)  
plt.ylim(0,)
```



Correlation - Negative Linear Relationship

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

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



Pearson Correlation

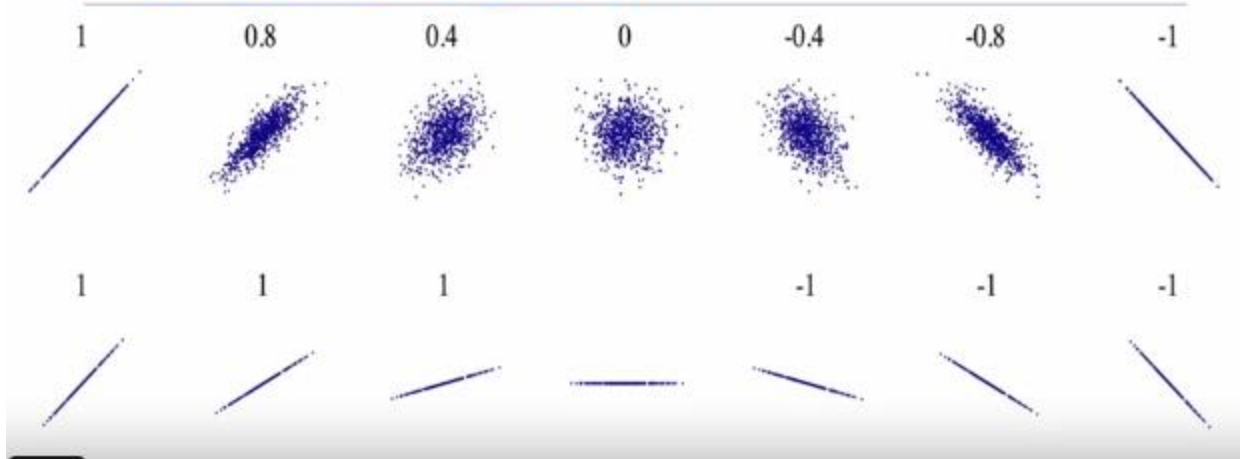
- Measure the strength of the correlation between two features.
 - Correlation coefficient
 - P-value
- Correlation coefficient
 - Close to +1: Large Positive relationship
 - Close to -1: Large Negative relationship
 - Close to 0: No relationship
- P-value
 - P-value < 0.001 **Strong** certainty in the result
 - P-value < 0.05 **Moderate** certainty in the result
 - P-value < 0.1 **Weak** certainty in the result
 - P-value > 0.1 **No** certainty in the result
- Strong Correlation:
 - Correlation coefficient close to 1 or -1
 - P value less than 0.001

Pearson Correlation- Example

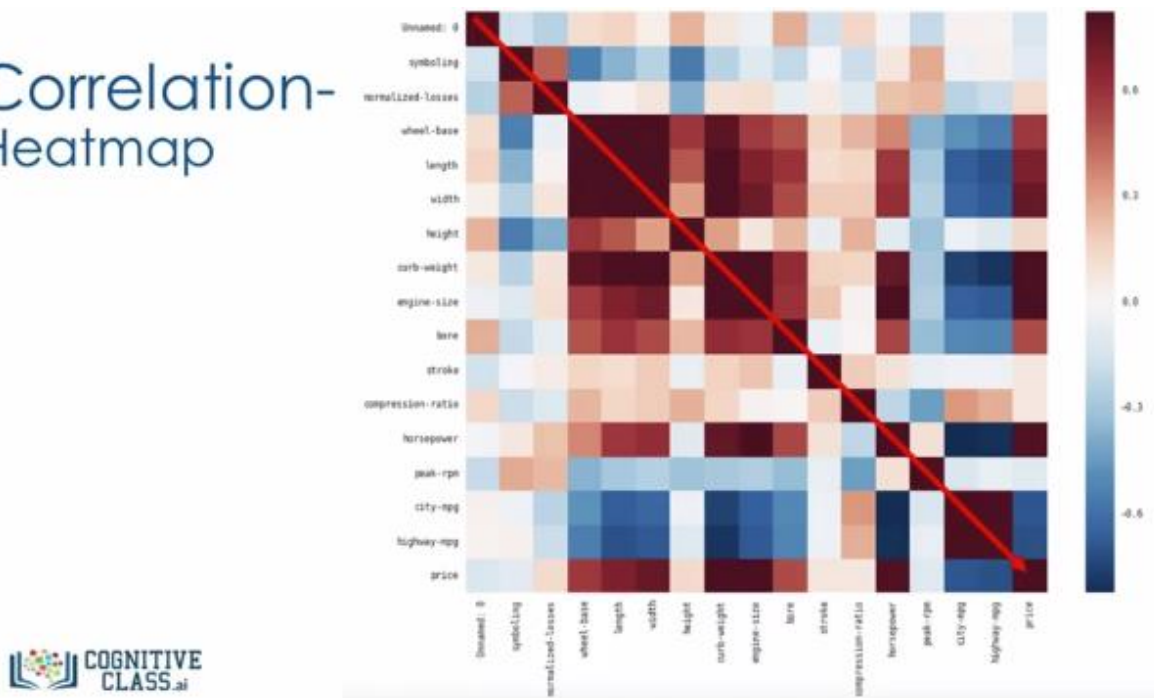
```
pearson_coef, p_value = stats.pearsonr(df['horsepower'], df['price'])
```

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

Pearson Correlation



Correlation-Heatmap



Diagonal =1, coz each variable related to itself

Analysis of Variance (ANOVA)

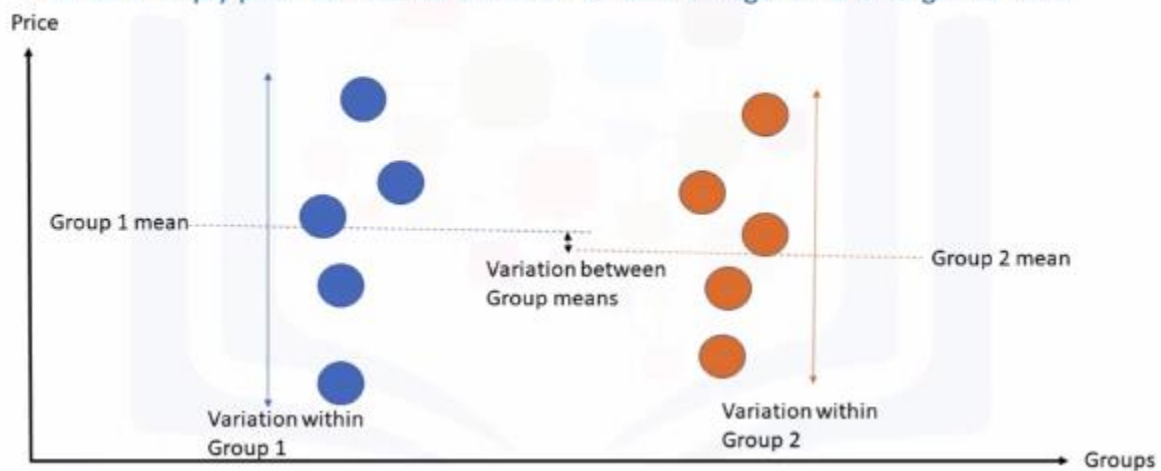
- Statistical comparison of groups
- Example: average price of different vehicle makes.

ANOVA

- Analysis Of Variance (ANOVA)
- Why do we perform ANOVA?
 - Finding correlation between different groups of a categorical variable.
- What we obtain from ANOVA?
 - F-test score : variation between sample group means divided by variation within sample group
 - p-value : confidence degree

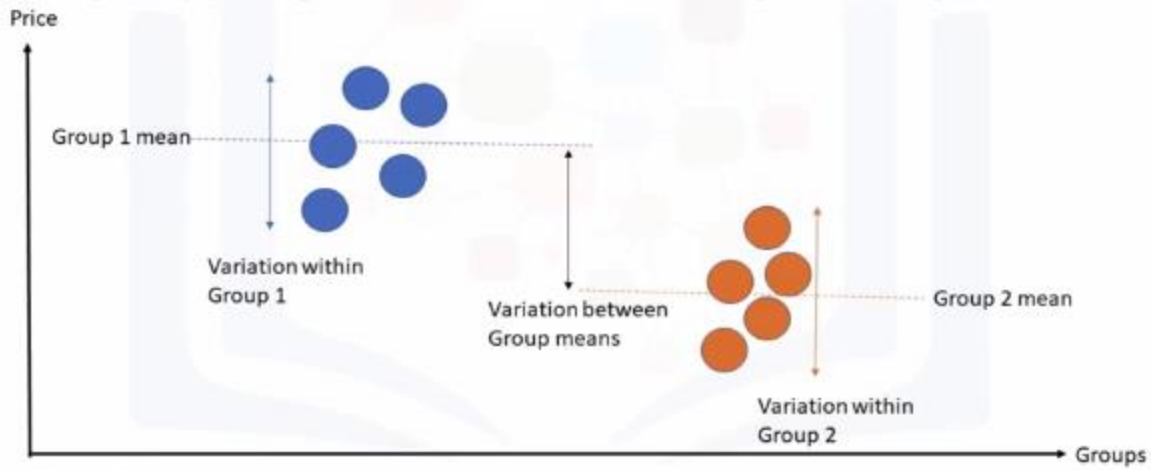
ANOVA – F-test

- Small F imply poor correlation between variable categories and target variable.

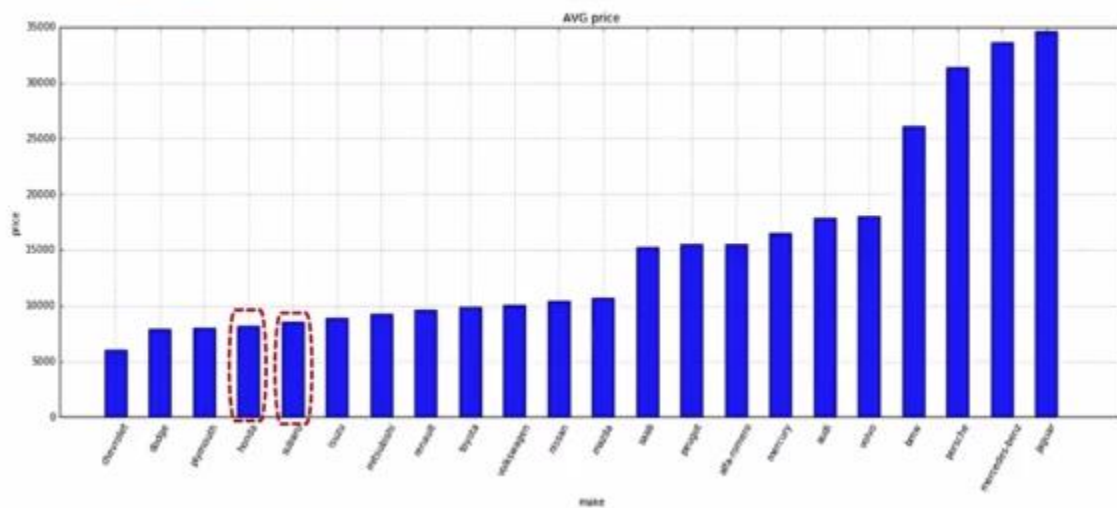


ANOVA – F-test

- Large F imply strong correlation between variable categories and target variable.



ANOVA



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)
```

WEEK 4: MODEL DEVELOPMENT

Model Development

- A model 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 to dependent variables.

independent variables or features

'highway-mpg'

55 mpg



Model



dependent variables

'predicted price'

\$5000

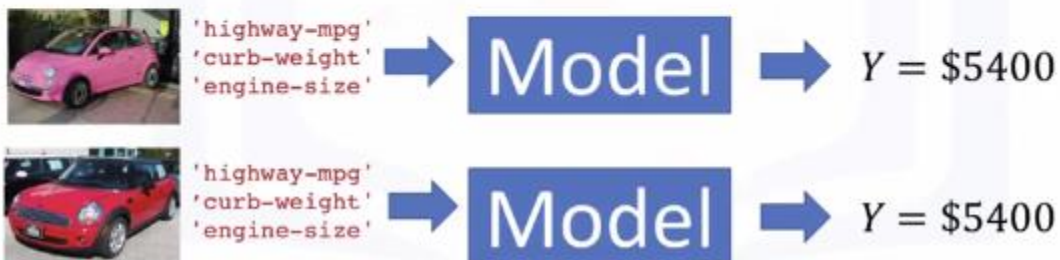
Model Development

- Usually the more **relevant data** you have the more accurate your model is



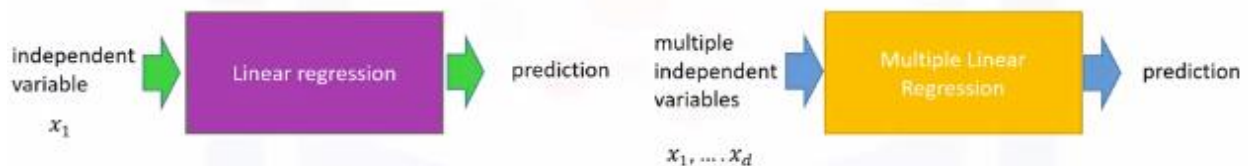
Model Development

- To understand why more data is important consider the following situation:
 - you have two almost identical cars
 - Pink cars sell for significantly less



Introduction

- 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

1. The predictor (independent) variable - x
2. The target (dependent) variable - y

$$y = b_0 + b_1 x$$

Two blue arrows point upwards from below the equation to the terms b_0 and b_1 .

- b_0 : the intercept
- b_1 : the slope

Fitting a Simple Linear Model Estimator

- X :Predictor variable
 - Y: Target variable
1. Import linear_model from scikit-learn

```
from sklearn.linear_model import LinearRegression
```
 2. Create a Linear Regression Object using the constructor :

```
lm=LinearRegression()
```

Fitting a Simple Linear Model

- We define the predictor variable and target variable

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

- Then use `lm.fit(X, Y)` to fit the model, i.e. find the parameters b_0 and b_1

```
lm.fit(X, Y)
```

- We can obtain a prediction

```
Yhat=lm.predict(X)
```

Yhat	X
2	5
:	
3	4



SLR – Estimated Linear Model

- We can view the intercept (b_0): `lm.intercept_`
38423.305858
- We can also view the slope (b_1): `lm.coef_`
-821.73337832
- The Relationship between Price and Highway MPG is given by:
- **Price = 38423.31 - 821.73 * highway-mpg**

$$\hat{Y} = b_0 + b_1x$$

Multiple Linear Regression (MLR)

This method is used to explain the relationship between:

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

Multiple Linear Regression (MLR)

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

- b_0 : **intercept** ($X=0$)
- b_1 : the **coefficient** or **parameter** of x_1
- b_2 : the **coefficient** of **parameter** x_2 and so on..

Fitting a Multiple Linear Model Estimator

1. We can extract the for 4 predictor variables and store them in the variable Z

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

2. Then train the model as before:

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

3. We can also obtain a prediction

```
Yhat=lm.predict(X)
```

x_1	x_2	x_3	x_4	Yhat
3	5	-4	3	2
:	:	:	:	:
2	4	2	-4	3

MLR – Estimated Linear Model

1. Find the intercept (b_0)

```
lm.intercept_  
-15678.742628061467
```

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

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

The Estimated Linear Model:

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

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

MODEL EVALUATION USING visualisation

Regression Plot

Why use regression plot?

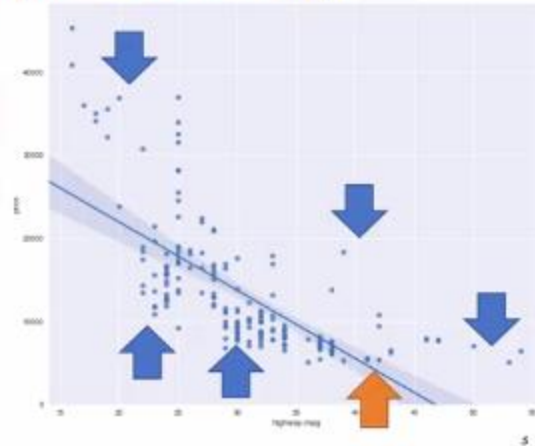
It gives us a good estimate of:

1. The relationship between two variables
2. The strength of the correlation
3. The direction of the relationship (positive or negative)

Regression Plot

Regression Plot shows us a combination of:

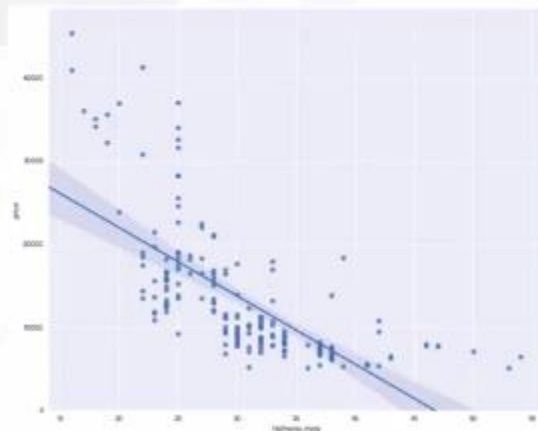
- The scatterplot: where each point represents a different y
- The fitted linear regression line (\hat{y}).



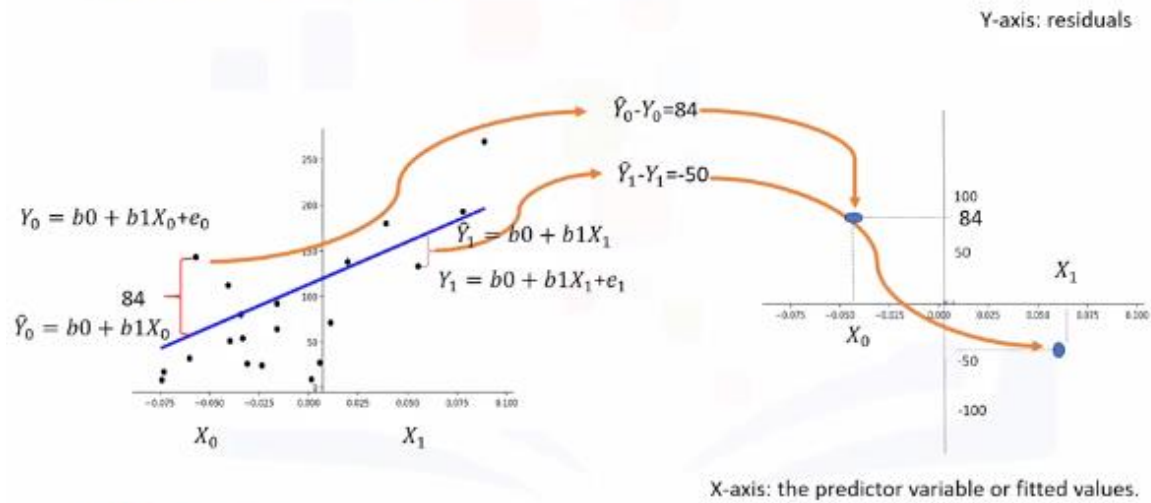
Regression Plot

```
import seaborn as sns
```

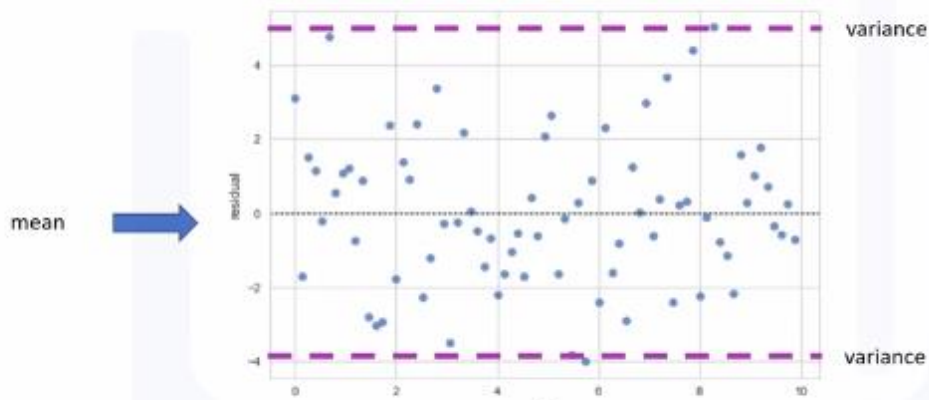
```
sns.regplot(x="highway-mpg", y="price", data=df)  
plt.ylim(0,)
```



Residual Plot

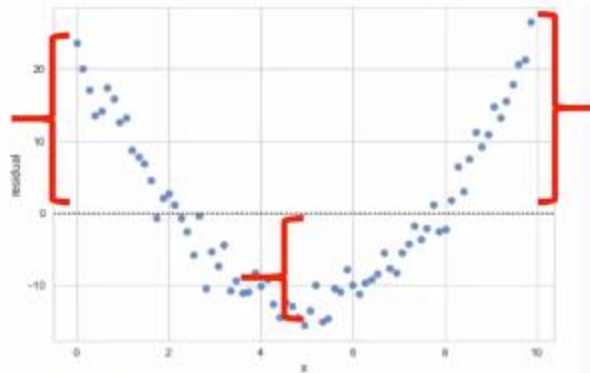


Residual Plot



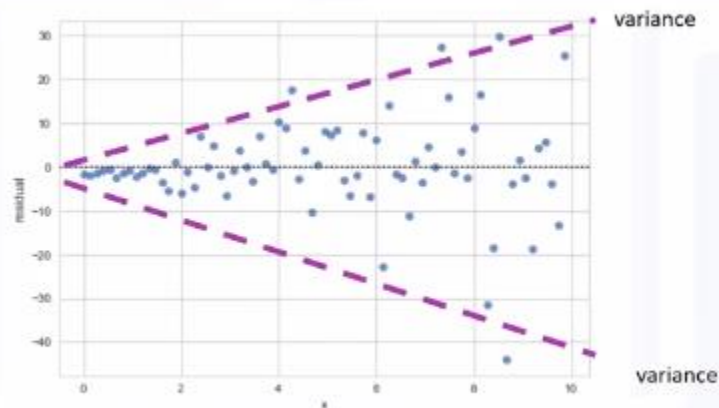
- Look at the **spread of the residuals**:
 - Randomly spread out around x-axis then a linear model is appropriate.

Residual Plot



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

Residual Plot

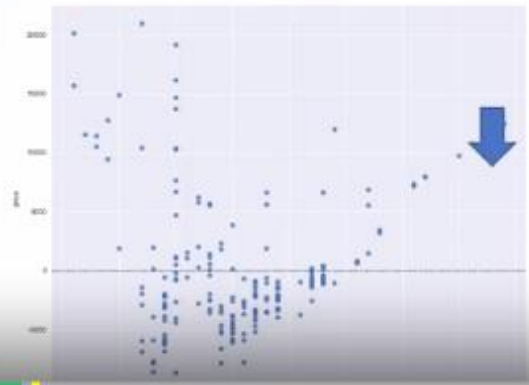


- Not randomly spread out around the x-axis
- Variance appears to change with x axis

Residual Plot

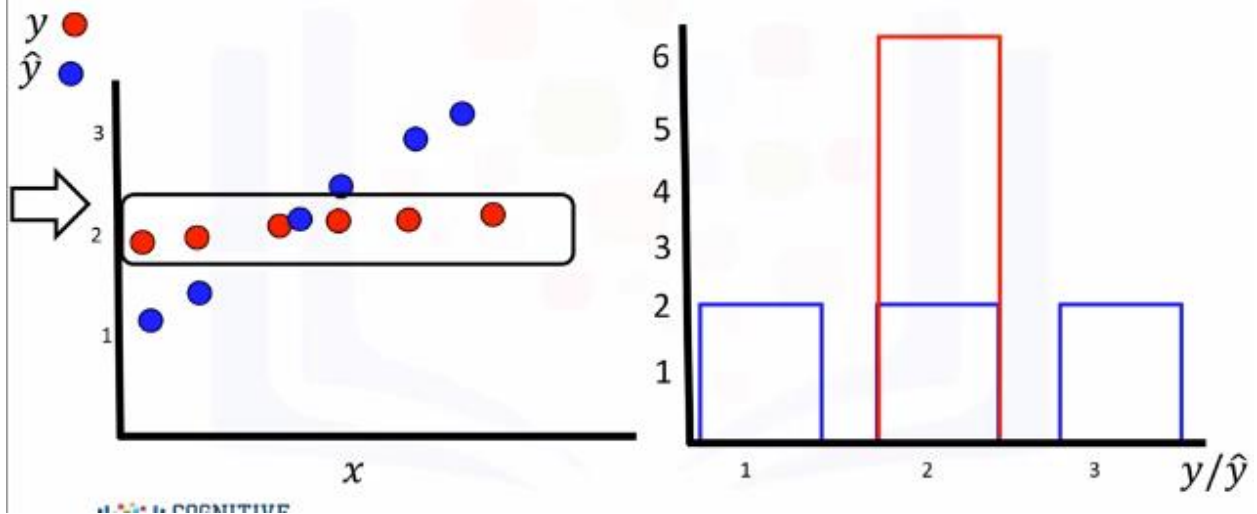
```
import seaborn as sns
```

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



Play

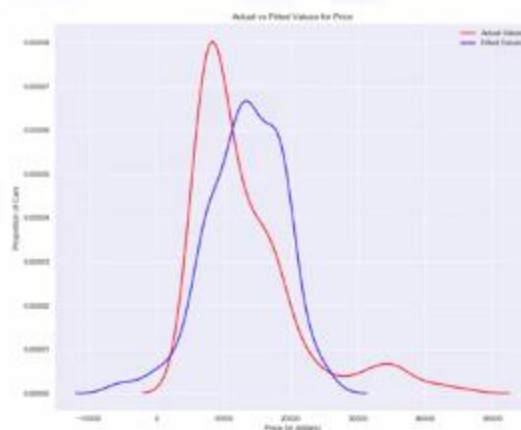
Distribution Plots



Distribution Plots

Compare the distribution plots:

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



Distribution Plots

```
import seaborn as sns
```



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

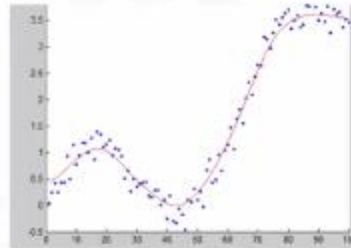
```
sns.distplot(Yhat, hist=False, color="b", label="Fitted Values" , ax=ax1)
```

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

- Quadratic – 2nd order

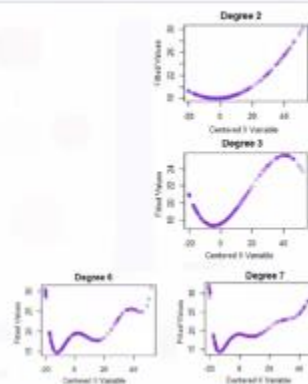
$$\hat{Y} = b_0 + b_1 x_1 + b_2 (x_1)^2$$

- Cubic – 3rd order

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

- Higher order

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



Polynomial Regression

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)
```

$$-1.557(x_1)^3 + 204.8(x_1)^2 + 8965 x_1 + 1.37 \times 10^5$$

If we need multi dim, x1 and x2 ... then

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)
x_polly=pr.fit_transform(x[['horsepower', 'curb-weight']])
```

Polynomial Regression with More the One Dimension

```
pr=PolynomialFeatures(degree=2)
```

X_1	X_2
1	2

```
pr.fit_transform([1,2], include_bias=False)
```



X_1	X_2	X_1X_2	X_1^2	X_2^2
1	2	(1)2	1	(2) ²

1	2	2	1	4
---	---	---	---	---

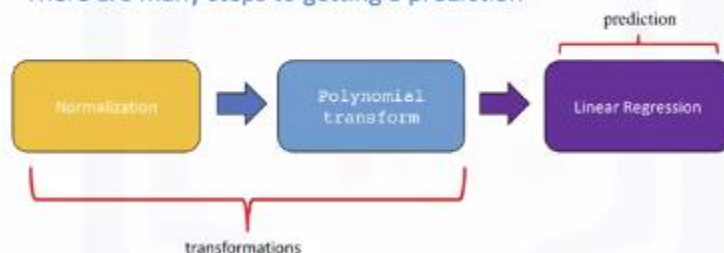
Pre-processing

- For example we can Normalize the each feature simultaneously:

```
from sklearn.preprocessing import StandardScaler
SCALE=StandardScaler()
SCALE.fit(x_data[['horsepower', 'highway-mpg']])
x_scale=SCALE.transform(x_data[['horsepower', 'highway-mpg']])
```

Pipelines

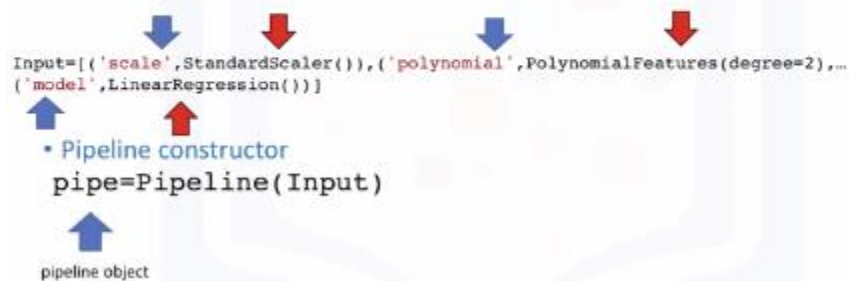
- There are many steps to getting a prediction



Pipelines

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
```

Pipeline Constructor



- We can train the pipeline object

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

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



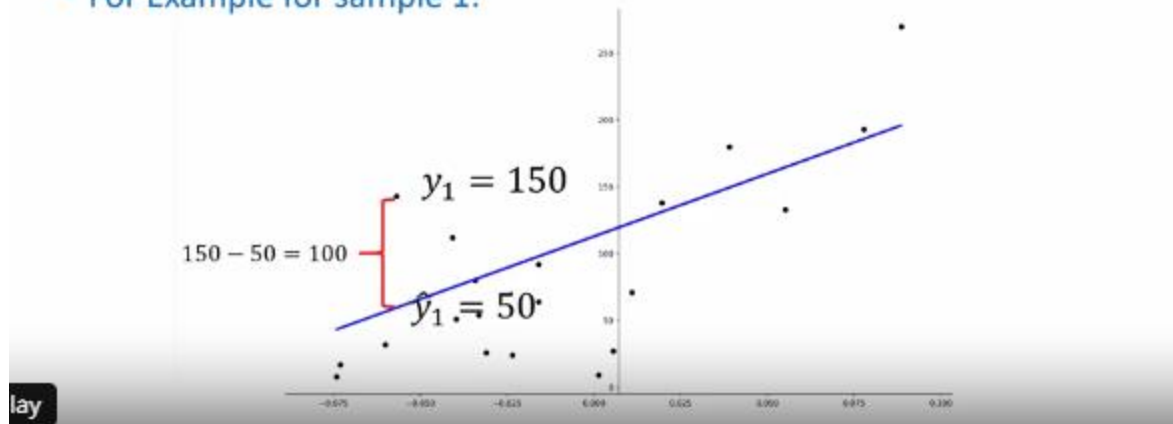
Measure for insample eevaluation

Measures for In-Sample Evaluation

- A way to numerically determine how good the model fits on dataset.
- Two important measures to determine the fit of a model:
 - **Mean Squared Error (MSE)**
 - **R-squared (R^2)**

Mean Squared Error (MSE)

- For Example for sample 1:



And square it and take average, div no of samples

Mean Squared Error (MSE)

- In python we can measure the MSE as follows

```
from sklearn.metrics import mean_squared_error  
  
mean_squared_error(df['price'], Y_predict_simple_fit)  
  
3163502.944639888
```

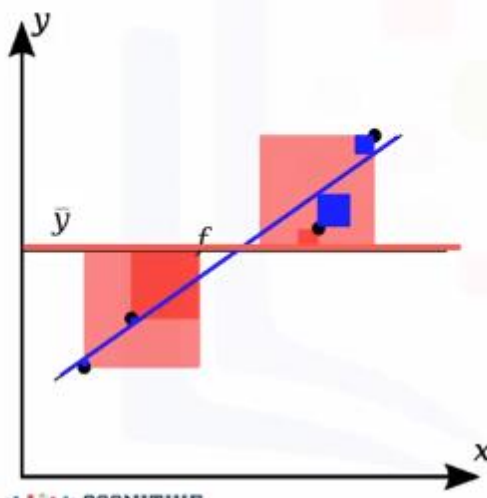
R-squared/ R^2

- The Coefficient of Determination or R squared (R^2)
- Is a measure to determine how close the data is to the fitted regression line.
- R^2 : the percentage of variation of the target variable (Y) that is explained by the linear model.
- Think about as comparing a regression model to a simple model i.e the mean of the data points

Coefficient of Determination (R^2)

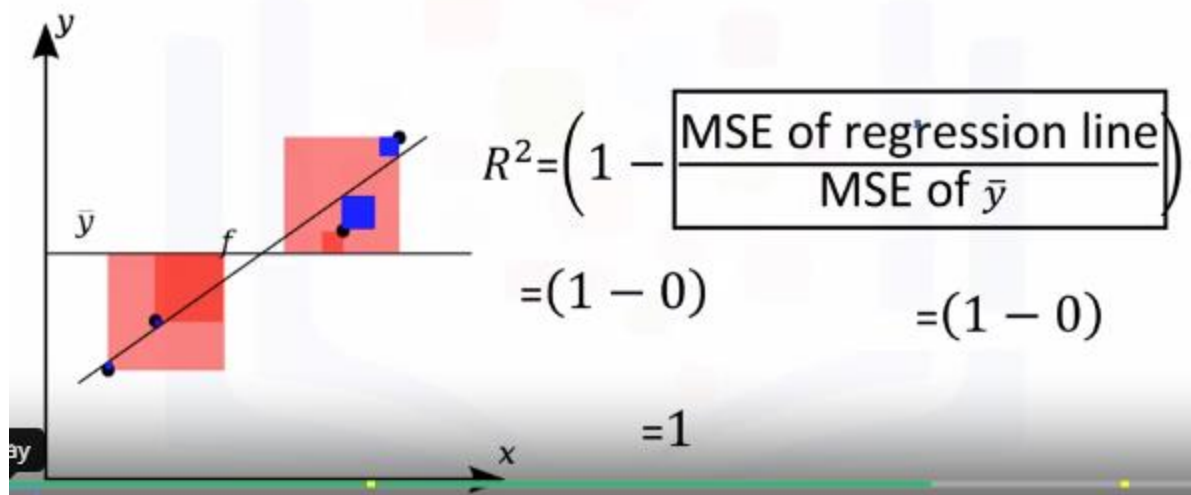
$$R^2 = \left(1 - \frac{\text{MSE of regression line}}{\text{MSE of the average of the data}} \right)$$

Coefficient of Determination (R^2)



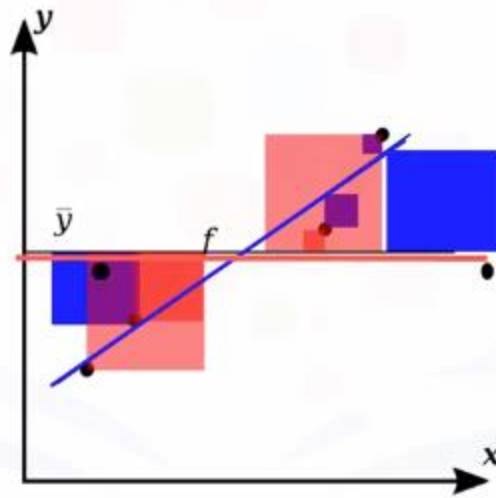
- The blue line represents the regression line
- The blue squares represent the MSE of the regression line
- The red line represents the average value of the data points
- The red squares represent the MSE of the red line
- We see the area of the blue squares is much smaller than the area of the red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

Coefficient of Determination (R^2)

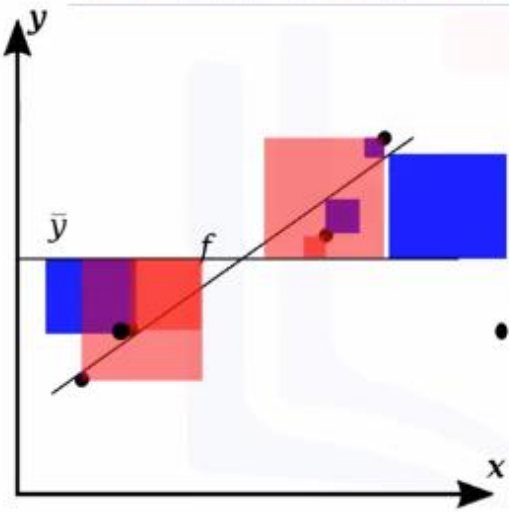


Hence a good fit

Coefficient of Determination (R^2)



Coefficient of Determination (R^2)



$$\begin{aligned} R^2 &= \left(1 - \frac{\text{MSE of regression line}}{\text{MSE of } \bar{y}} \right) \\ &= (1 - 1) \\ &= 0 \end{aligned}$$

R-squared/ R^2

- Generally the values of the MSE are between 0 and 1.
- WE can calculate the the R^2 as follows

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

```
lm.fit(X, Y)
```

```
lm.score(X, y)  
0.496591188
```

Prediction & decision making

Decision Making: Determining a Good Model Fit

To determine final best fit, we look at a combination of:

- Do the predicted values make sense
- Visualization
- Numerical measures for evaluation
- Comparing Models

Do the predicted values make sense

- First we train the model

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

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

```
lm.predict(30)
```

- Result: \$ 13771.30

```
lm.coef_  
-821.73337832
```

- Price = 38423.31 - 821.73 * highway-mpg

Do the predicted values make sense

- First we import numpy

```
import numpy as np
```

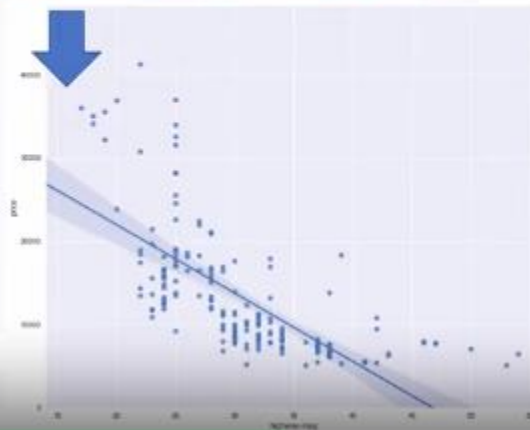
- We use the numpy function `arange` to generate a sequence from 1 to 100

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

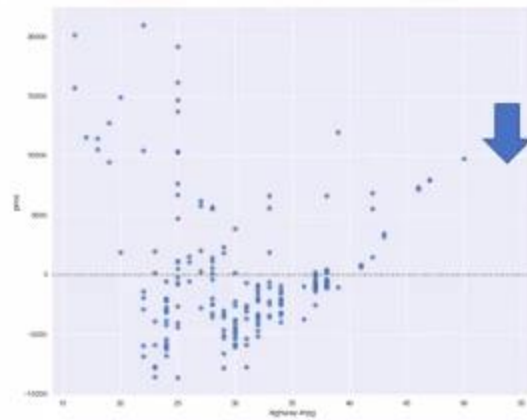
1	2	...	99	100
---	---	-----	----	-----

Visualization

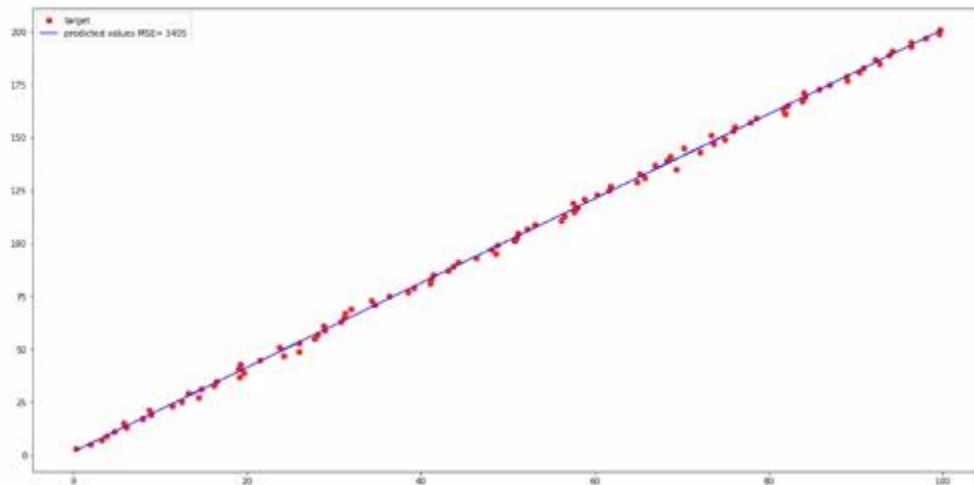
- Simply visualizing your data with a regression



Residual Plot



Numerical measures for Evaluation





WEEK 5 : Model Evalutaion

Model Evaluation

- In-sample evaluation tells us how well our model will fit the data used to train it
- Problem?
 - It does not tell us how well the trained model can be used to predict new data
- Solution?
 - In- sample data or training data
 - Out-of-sample evaluation or test set

Training/Testing Sets

Data:

- Split dataset into:
 - Training set (70%), 
 - Testing set (30%) 
- Build and train the model with a training set
- Use testing set to assess the performance of a predictive model
- When we have completed testing our model we should use all the data to train the model to get the best performance

Function `train_test_split()`

- Split data into random train and test subsets

```
from sklearn.model_selection import train_test_split
```

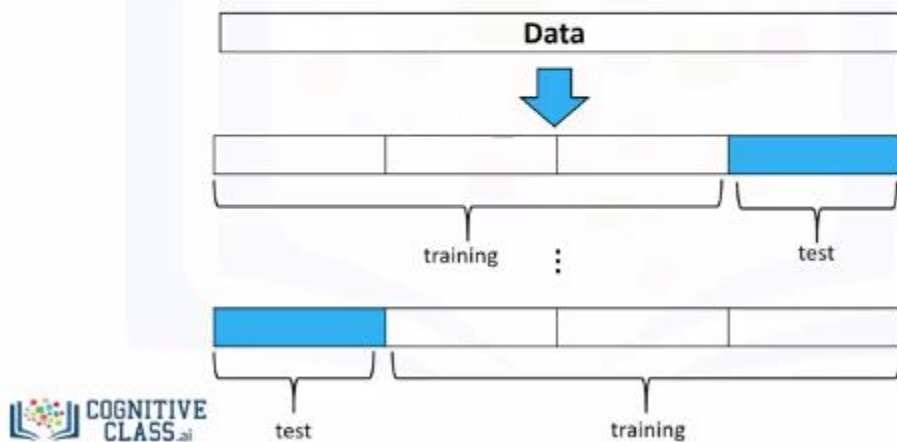
```
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.3, random_state=0)
```

- **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
- **test_size**: percentage of the data for testing (here 30%)
- **random_state**: number generator used for random sampling

Bias-variance problem

Cross Validation

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



Function `cross_val_score()`

```
from sklearn.model_selection import cross_val_score
```

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

```
np.mean(scores)
```

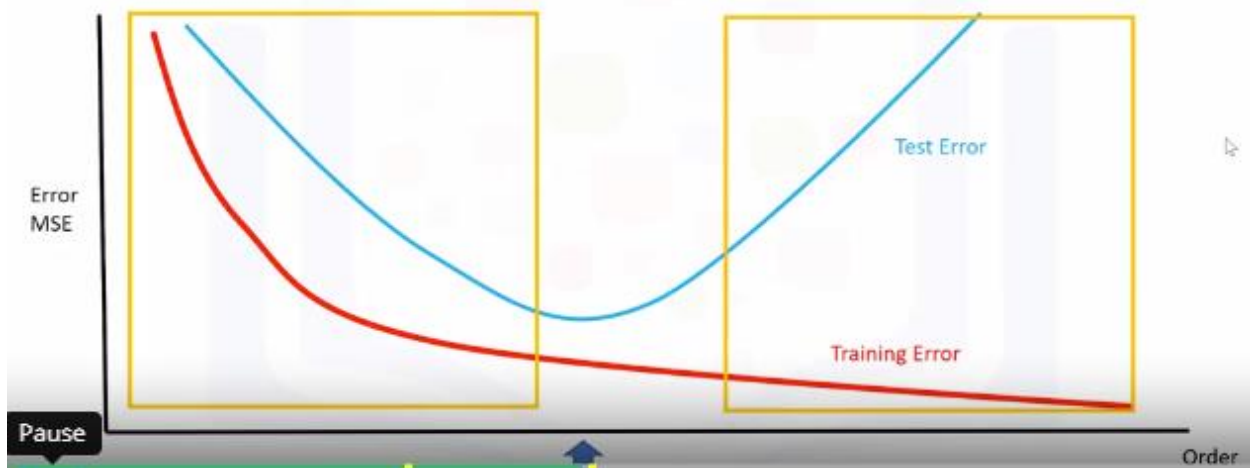
Function `cross_val_predict()`

- It returns the prediction that was obtained for each element when it was in the test set
- Has a similar interface to `cross_val_score()`

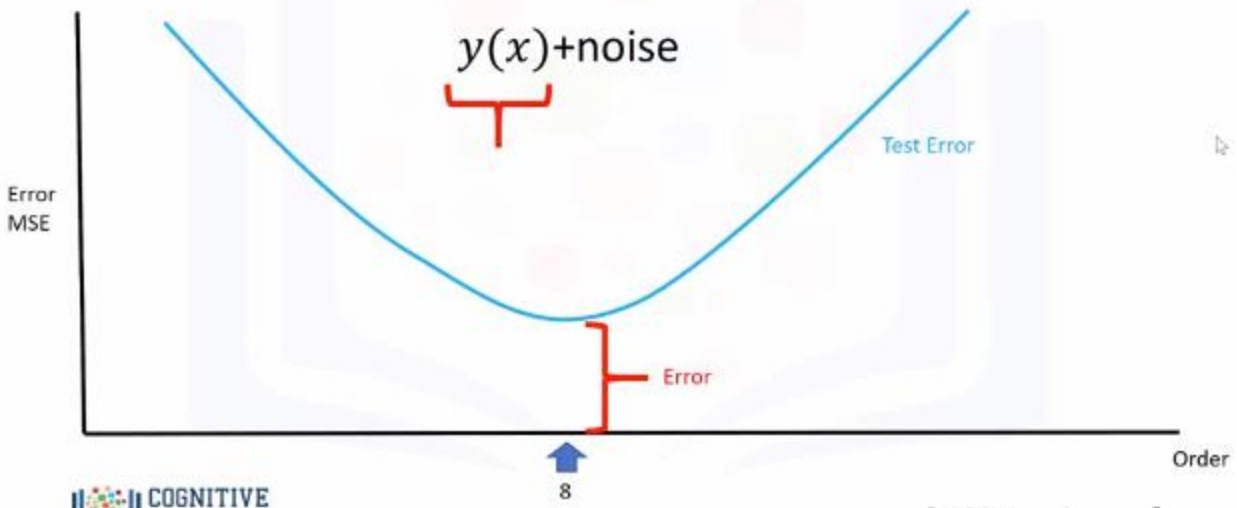
```
from sklearn.model_selection import cross_val_predict
```

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

Model Selection



Model Selection



Calculating R^2 for (higher degree of polynomial bad)

```
Rsqu_test=[]  
order=[1,2,3,4]  
for n in order:  
    pr=PolynomialFeatures(degree=n)  
    x_train_pr=pr.fit_transform(x_train[['horsepower']])  
    x_test_pr=pr.fit_transform(x_test[['horsepower']])  
    lr.fit(x_train_pr,y_train)  
    Rsqu_test.append(lr.score(x_test_pr,y_test))
```

Ridge Regression

```
from sklearn.linear_model import Ridge
```

```
RidgeModel=Ridge(alpha=0.1)
```

```
RidgeModel.fit(X,y)
```

```
Yhat=RidgeModel.predict(X)
```