MACHINE LEARNING NOTES

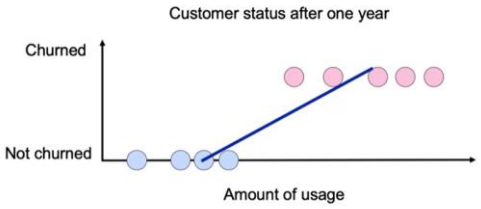
17/07/2023

Modeling Classification:

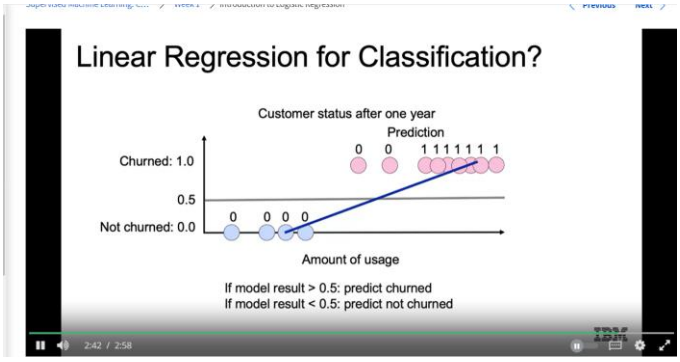
Examples of models used for Supervised Learning: Classification:
- Logistic Regression
- K-Nearest Neighbors
- Support Vector Machines
- Neural Networks
- Decision Tree
- Random Forests
- Boosting
- Ensemble Models

Each of these models can be used for both regression and classification.

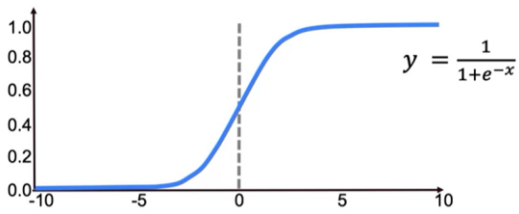
Introduction to Logistic Regression



$$y_\beta(x) = \beta_0 + \beta_1 x + \varepsilon$$

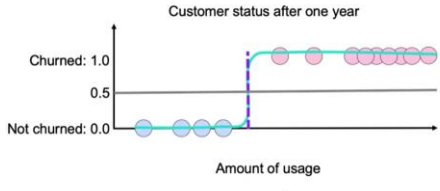


Introducing the Sigmoid Function



$$y = \frac{1}{1+e^{-x}}$$

Linear Regression for Classification?



$$y_\beta(x) = \frac{1}{1+e^{-(\beta_0+\beta_1 x+\varepsilon)}}$$

Logistic vs. Linear Regression

$$P(x) = \frac{1}{1+e^{-(\beta_0+\beta_1 x+\varepsilon)}} \quad \Rightarrow \quad P(x) = \frac{e^{(\beta_0+\beta_1 x)}}{1+e^{(\beta_0+\beta_1 x)}}$$

Logistic Function

Logistic vs. Linear Regression

$$P(x) = \frac{1}{1+e^{-(\beta_0+\beta_1 x+\varepsilon)}} \quad \Rightarrow \quad P(x) = \frac{e^{(\beta_0+\beta_1 x)}}{1+e^{(\beta_0+\beta_1 x)}}$$

Logistic Function

$$\frac{P(x)}{1-P(x)} = e^{(\beta_0+\beta_1 x)}$$

Odds Ratio

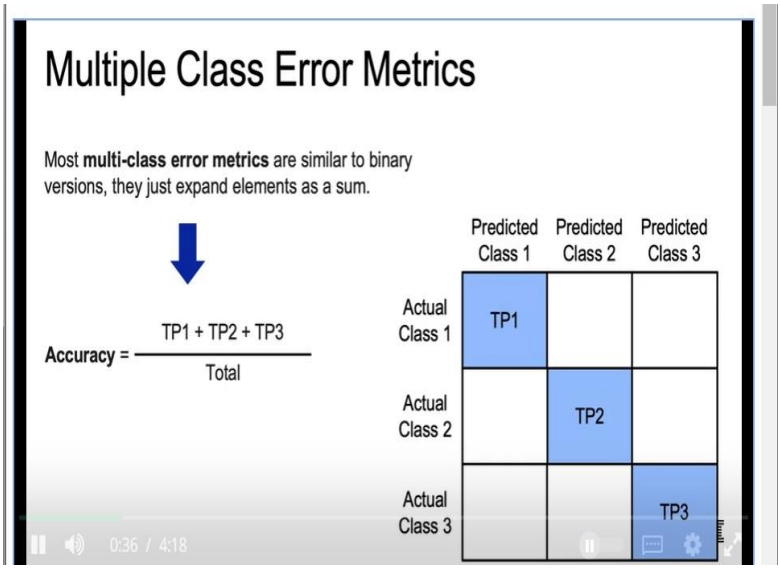
Logistic vs. Linear Regression

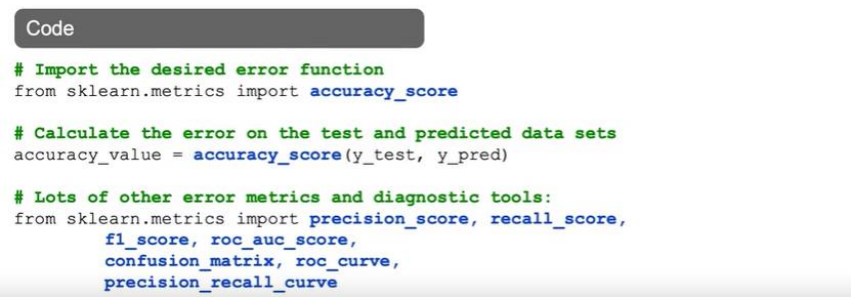
$$P(x) = \frac{1}{1+e^{-(\beta_0+\beta_1 x+\varepsilon)}} \quad \Rightarrow \quad P(x) = \frac{e^{(\beta_0+\beta_1 x)}}{1+e^{(\beta_0+\beta_1 x)}}$$

Logistic Function

$$log\left[\frac{P(x)}{1-P(x)}\right] = \beta_0 + \beta_1 x$$

Odds Ratio





Summary/Review

Classification Problems

The two main types of supervised learning models are:

- Regression models, which predict a continuous outcome
- Classification models, which predict a categorical outcome.
  The most common models used in supervised learning are:

- Logistic Regression
- K-Nearest Neighbors
- Support Vector Machines
- Decision Tree
- Neural Networks
- Random Forests
- Boosting
- Ensemble Models
  With the exception of logistic regression, these models are commonly used for both regression and classification. Logistic regression is most common for dichotomous and nominal dependent variables.

Logistic Regression

Logistic regression is a type of regression that models the probability of a certain class occurring given other independent variables.It uses a logistic or logit function to model a dependent variable. It is a very common predictive model because of its high interpretability.

Classification Error Metrics

A confusion matrix tabulates true positives, false negatives, false positives and true negatives. Remember that the false positive rate is also known as a type I error. The false negatives are also known as a type II error.

Accuracy is defined as the ratio of true postives and true negatives divided by the total number of observations. It is a measure related to predicting correctly positive and negative instances.

Recall or sensitivity identifies the ratio of true positives divided by the total number of actual positives. It quantifies the percentage of positive instances correctly identified.

Precision is the ratio of true positive divided by total of predicted positives. The closer this value is to 1.0, the better job this model does at identifying only positive instances.

Specificity is the ratio of true negatives divided by the total number of actual negatives. The closer this value is to 1.0, the better job this model does at avoiding false alarms.

The receiver operating characteristic (ROC) plots the true positive rate (sensitivity) of a model vs. its false positive rate (1-sensitivity).

The area under the curve of a ROC plot is a very common method of selecting a classification methods.T

he precision-recall curve measures the trade-off between precision and recall.

The ROC curve generally works better for data with balanced classes, while the precision-recall curve generally works better for data with unbalanced classes.

Certainly, I'd be happy to provide you with detailed information and example code for each of the machine learning algorithms you mentioned. Let's start with Logistic Regression.

Logistic Regression:

Logistic Regression is a binary classification algorithm that is used to predict the probability of a binary outcome (1 / 0, Yes / No, True / False) based on one or more predictor variables.

Algorithm Overview:

1. Compute the weighted sum of the input features.

2. Apply the logistic function (sigmoid) to the weighted sum to transform it into a probability score.

3. Classify the example based on a threshold (usually 0.5).

Example Code:

Here's a simple example of logistic regression using Python and the `scikit-learn` library:

```python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Sample data
X = np.array([[1.2], [2.4], [3.1], [4.8], [5.2]])
y = np.array([0, 0, 1, 1, 1])

# Splitting data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a logistic regression model
model = LogisticRegression()

# Fit the model to the training data
model.fit(X_train, y_train)

# Make predictions on the test data
predictions = model.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, predictions)
print("Accuracy:", accuracy)
```

```

Remember, Logistic Regression is primarily used for binary classification problems.

Next, let's move on to K-Nearest Neighbors (KNN).

## K-Nearest Neighbors (KNN):

K-Nearest Neighbors is a simple and intuitive classification algorithm that assigns a class label to a data point based on the majority class labels of its k-nearest neighbors in the feature space.

**Algorithm Overview:**

1. Choose the number of neighbors (k).

2. Calculate the distance between the target data point and all other data points.

3. Select the k-nearest data points.

4. Assign the class label based on majority vote among the k-nearest neighbors.

**Example Code:**

Here's a basic example of KNN using Python and `scikit-learn`:

```python

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy_score

# Sample data

X = np.array([[1, 2], [2, 3], [3, 4], [4, 5], [5, 6]])

y = np.array([0, 0, 1, 1, 1])
```
```

```
# Splitting data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Create a KNN model with k=3

model = KNeighborsClassifier(n_neighbors=3)


# Fit the model to the training data

model.fit(X_train, y_train)


# Make predictions on the test data

predictions = model.predict(X_test)


# Calculate accuracy

accuracy = accuracy_score(y_test, predictions)

print("Accuracy:", accuracy)
```

KNN is simple to understand but can be computationally expensive for large datasets.

If you'd like, I can continue with explanations and code examples for the remaining algorithms: Support Vector Machines, Decision Trees, Neural Networks, Random Forests, Boosting, and Ensemble Models. Just let me know!

……..

K Nearest Neighbors: The Syntax

Code

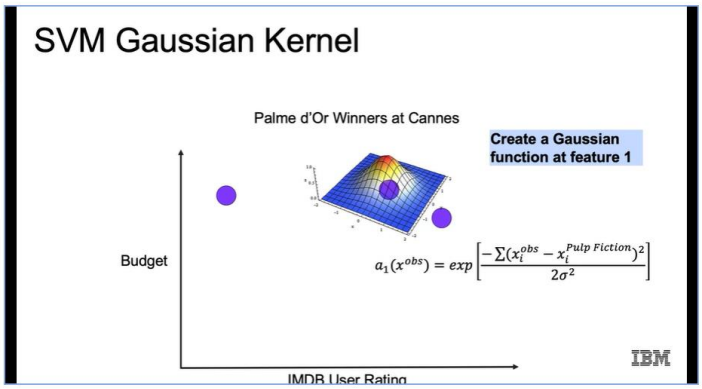
```
# Import the class containing the classification method
from sklearn.neighbors import KNeighborsClassifier

# Create an instance of the class
KNN = KNeighborsClassifier(n_neighbors=3)

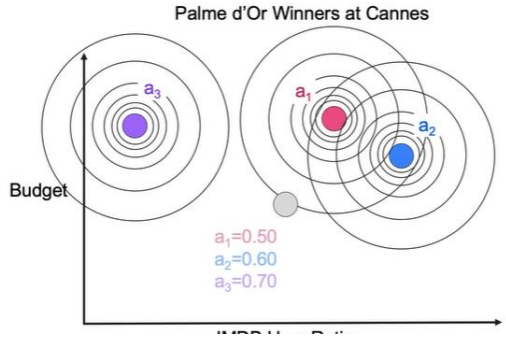
# Fit the instance on the data and then predict the expected value
KNN = KNN.fit(X_train, y_train)
y_predict = KNN.predict(X_test)

The fit and predict/transform syntax will show up throughout the course.
```

SVM Gaussian Kernel

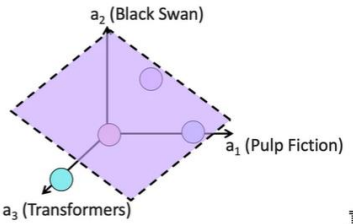


SVM Gaussian Kernel



Classification in the New Space

Transformation:
$[x_1, x_2] \rightarrow [a_1, a_2, a_3]$



Code

```
# Import the class containing the classification method
from sklearn.svm import SVC

# Create an instance of the class
rbfSVC = SVC(kernel='rbf', gamma=1.0, C=10.0)

# Fit the instance on the data and then predict the expected value
rbfSVC = rbfSVC.fit((X_train, y_train)
y_predict = rbfSVC.predict(X_test)

Tune kernel and associated parameters with cross-validation.
```

"C" is penalty associated with the error term

Data Analysis using Python 8/11/23

Writing code using DB-API

```python
from dbmodule import connect

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

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

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

#Free resources
Cursor.close()
connection.close()
```

Data Analysis with Python

Cheat Sheet: Importing Data Sets

Package/Method	Description	Code Example

Read CSV data set	Read the CSV file containing a data set to a pandas data frame	`df = pd.read_csv(<CSV_path>, header = None)`
		`# load without header`
		`df = pd.read_csv(<CSV_path>, header = 0)`
		`# load using first row as header`

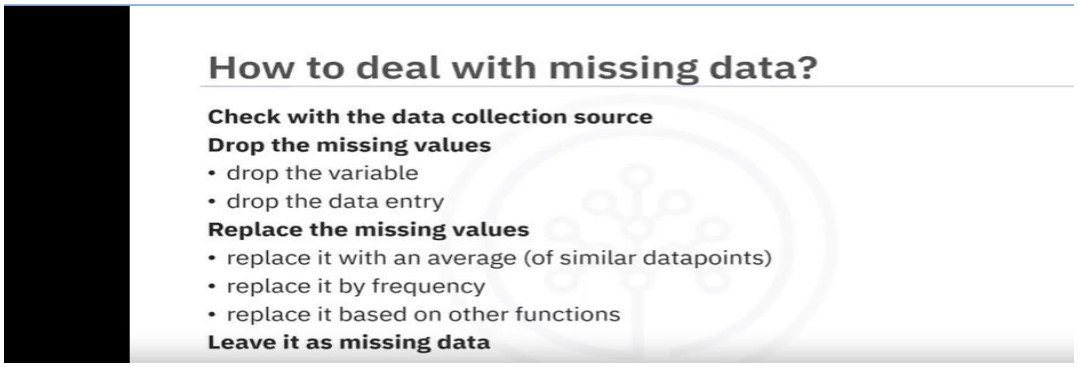
1.
```python
df = pd.read_csv(<CSV_path>, header = None)
# load without header
df = pd.read_csv(<CSV_path>, header = 0)
# load using first row as header
```

other Python compiler on your local machine, you can use the URL of the re

Print first few entries	Print the first few entries (default 5) of the pandas data frame	`df.head(n) #n=number of entries; default 5`
Print last few entries	Print the last few entries (default 5) of the pandas data frame	`df.tail(n) #n=number of entries; default 5`
Assign header names	Assign appropriate header names to the data frame	`df.columns = headers`
Replace "?" with NaN	Replace the entries "?" with NaN entry from Numpy library	`df = df.replace("?", np.nan)`
Retrieve data types	Retrieve the data types of the data frame columns	`df.dtypes`

Retrieve statistical description	Retrieve the statistical description of the data set. Defaults use is for only numerical data types. Use include="all" to create summary for all variables	`df.describe() #default use df.describe(include="all")`
Retrieve data set summary	Retrieve the summary of the data set being used, from the data frame	`df.info()`
Save data frame to CSV	Save the processed data frame to a CSV file with a specified path	`df.to_csv(<output CSV path>)`

Menu



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)
```

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

0

How to replace missing values in Python

Use dataframe.replace(missing_value, new_value):

normalized-losses	make
...	...
164	audi
164	audi
NaN	audi

→

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

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

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

Data Formatting

- Data is 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
N.Y.
Ny
NY
New York

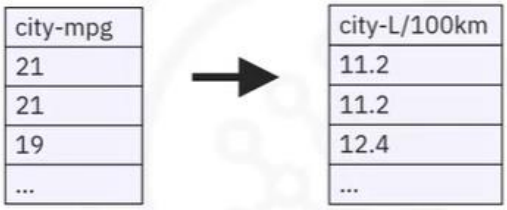
City
New York
New York
New York
New York

Formatted:

- more clear
- easy to aggregate
- easy to compare

Applying calculations to an entire column

- Convert “mpg” to “L/100km” in Car dataset




```
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
```

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")
```

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**

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.

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()
```

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 an feature range from 5,000 to 45,500

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

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 → 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'])
```

https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/jupyterlite/latest/lab/index.html?notebook_url=https%3A%2F%2Fcf-courses-data.s3.us.cloud-object-storage.appdomain.cloud%2FIBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork%2Fjupyterlite%2Ffiles%2FDA0101EN-2-Review-Data-Wrangling.jupyterlite.ipynb

---

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 which have the most impact on the car price?"

Learning Objectives

In this lesson you will learn about:

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

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	201.00000(
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

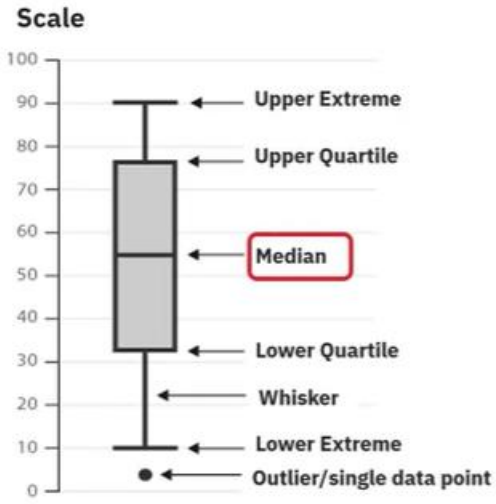
Descriptive Statistics: Value_Counts()

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

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

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

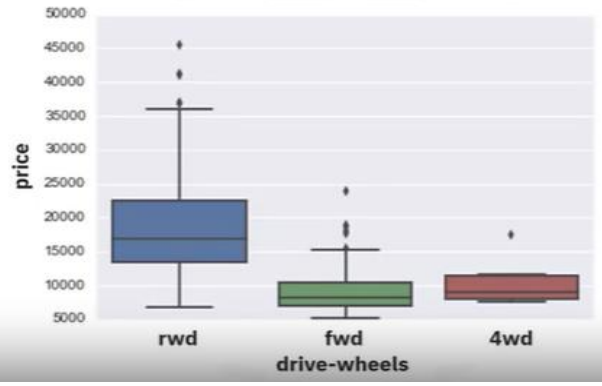
	value_counts
drive-wheels	
fwd	118
rwd	75
4wd	8

Descriptive Statistics: Box Plots



Box Plot: Example

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

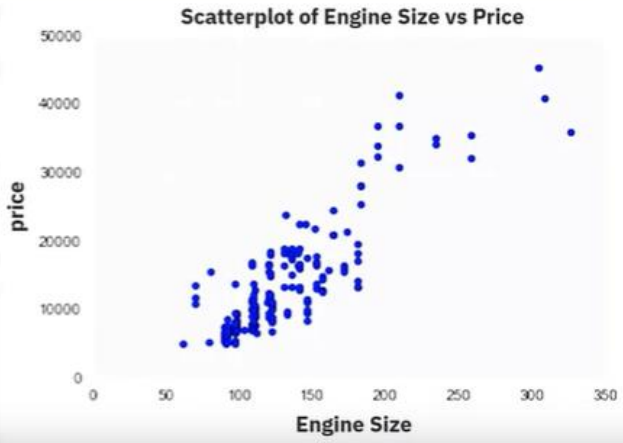


Descriptive Statistics: Scatter Plot

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

Scatterplot: Example

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

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



Grouping data

Question:

Is there any relationship between the different types of "drive system" and the "price" of the vehicles?

- Use Panda **dataframe.groupby()** method:
  - Can be applied to 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

Pandas method: Pivot()

- One variable is displayed along the columns, and the other variable is displayed along the rows

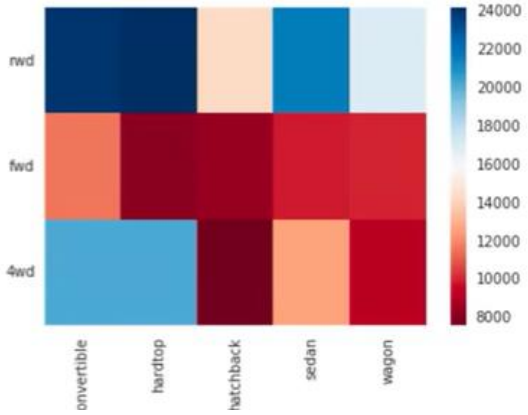
```
df_pivot = df_grp.pivot(index='drive-wheels', columns='body-style')
```

	price				
body-style	convertible	hardtop	hatchback	sedan	wagon
drive-wheels					
4wd	20239.229524	20239.229524	7603.000000	12647.333333	9095.750000
fwd	11595.000000	8429.000000	8396.387755	9811.8000000	9997.333333
rwd	23949.600000	24202.714286	14337.777778	21711.833333	16994.222222

Heatmap

- Plot target variable against multiple variables

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



Correlation

What is Correlation?

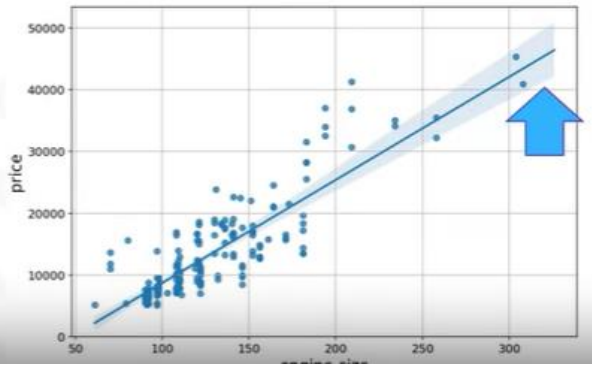
- Measures to what extent different variables are interdependent

- For example:
  - Lung cancer → Smoking
  - Rain → Umbrella
- Correlation doesn't imply causation

Correlation: Positive linear relationship

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

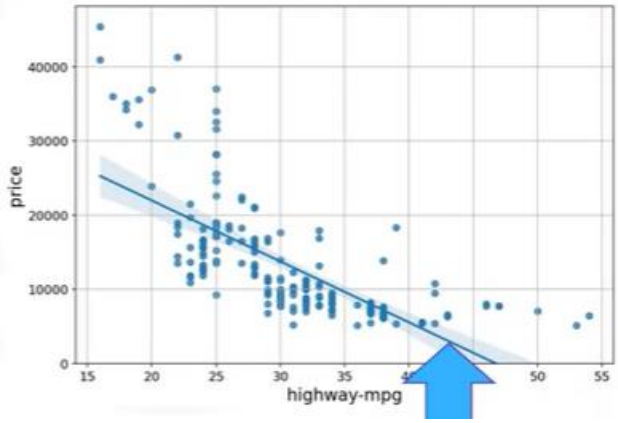
```
sns.regplot(x="engine-size", y="price",
data=df)
plt.ylim(0,)
```



Correlation: Negative linear relationship

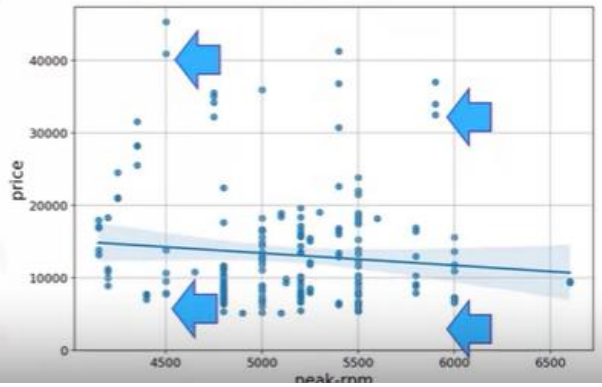
- Correlation between two features (highway-mpg and price)

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



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

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



Pearson Correlation

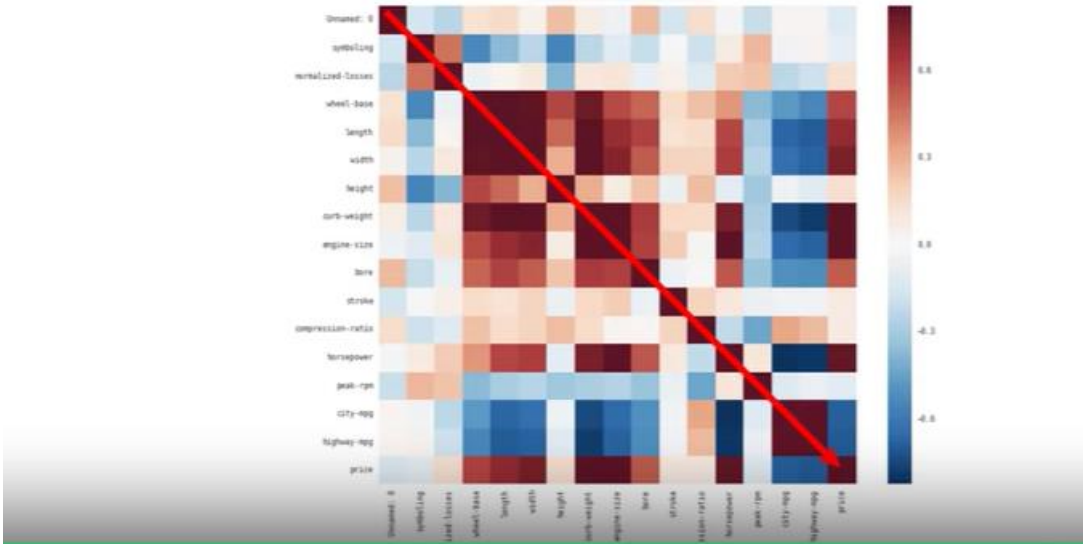
- Measure the strength of the correlation between two features
  - Correlation coefficient
  - P-value
- Correlation coefficient
  - Close to +1: Large Positive relationship
  - Close to -1: Large Negative relationship
  - Close to 0: No relationship
- Strong correlation
  - Correlation coefficient close to 1 or -1
  - P value less than 0.001
- 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

Pearson Correlation- Example

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

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

Correlation- Heatmap



Lesson Summary

Congratulations! You have completed this lesson. At this point in the course, you know:

- Tools like the '**describe**' function in pandas can quickly calculate key statistical measures like mean, standard deviation, and quartiles for all numerical variables in your data frame.
- Use the '**value_counts**' function to summarize data into different categories for categorical data.
- Box plots offer a more visual representation of the data's distribution for numerical data, indicating features like the median, quartiles, and outliers.
- Scatter plots are excellent for exploring relationships between continuous variables, like engine size and price, in a car data set.
- Use Pandas' '**groupby**' method to explore relationships between categorical variables.
- Use pivot tables and heat maps for better data visualizations.
- Correlation between variables is a statistical measure that indicates how the changes in one variable might be associated with changes in another variable.
- When exploring correlation, use scatter plots combined with a regression line to visualize relationships between variables.
- Visualization functions like **regplot,** from the **seaborn** library, are especially useful for exploring correlation.
- The **Pearson correlation**, a key method for assessing the correlation between continuous numerical variables, provides two critical values—the coefficient, which indicates the strength and direction of the correlation, and the P-value, which assesses the certainty of the correlation.
- A correlation coefficient close to 1 or -1 indicates a strong positive or negative correlation, respectively, while one close to zero suggests no correlation.
- For P-values, values less than .001 indicate strong certainty in the correlation, while larger values indicate less certainty. Both the coefficient and P-value are important for confirming a strong correlation.
- Heatmaps provide a comprehensive visual summary of the strength and direction of correlations among multiple variables.

Data Analysis with Python

Cheat Sheet: Exploratory Data Analysis

Package/Method	Description	Code Example

Complete dataframe correlation	Correlation matrix created using all the attributes of the dataset.	`1. 1`
		`1. df.corr()`
		`Copied!`

Specific Attribute correlation	Correlation matrix created using	`1. 1`
		`1. df[['attribute1','attribute2',...]].corr()`
		`Copied!`

	specific attributes of the dataset.	
Scatter Plot	Create a scatter plot using the data points of the dependent variable along the x-axis and the independent variable along the y-axis.	1. 1 2. 2 1. from matlplotlib import pyplot as 2. plt plt.scatter(df[['attribute_1']],df[['attribute_2']]) Copied!
Regression Plot	Uses the dependent and independent variables in a Pandas data frame to create a scatter plot with a generated linear regression line for the data.	1. 1 2. 2 1. import seaborn as sns 2. sns.regplot(x='attribute_1',y='attribute_2', data=df) Copied!
Box plot	Create a box-and-whisker plot that uses the pandas dataframe, the dependent, and the independent variables.	1. 1 2. 2 1. import seaborn as sns 2. sns.boxplot(x='attribute_1',y='attribute_2', data=df) Copied!
Grouping by attributes	Create a group of	1. 1

	different attributes of a dataset to create a subset of the data.	1. df_group = df[['attribute_1','attribute_2',...]] Copied!
GroupBy statements	a. Group the data by different categories of an attribute, displaying the average value of numerical attributes with the same category. b. Group the data by different categories of multiple attributes, displaying the average value of numerical attributes with the same category.	1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 1. a. 2. df_group = 3. df_group.groupby(['attribute_1'],as_index=False).mean() 4. b. 5. df_group = df_group.groupby(['attribute_1', 6. 'attribute_2'],as_index=False).mean() Copied!
Pivot Tables	Create Pivot tables for better representation of data based on parameters	1. 1 2. 2 1. grouped_pivot = 2. df_group.pivot(index='attribute_1',columns='attribute_2') Copied!
Pseudocolor plot	Create a heatmap image using a PsuedoColor plot (or pcolor)	1. 1 2. 2 1. from matlplotlib import pyplot as plt 2. plt.pcolor(grouped_pivot, cmap='RdBu') Copied!

	using the pivot table as data.	
Pearson Coefficient and p-value	Calculate the Pearson Coefficient and p-value of a pair of attributes	1. 1 2. 2 3. 3 1. From scipy import stats 2. pearson_coef,p_value=stats.pearsonr(df['attribute_1'], 3. df['attribute_2'])

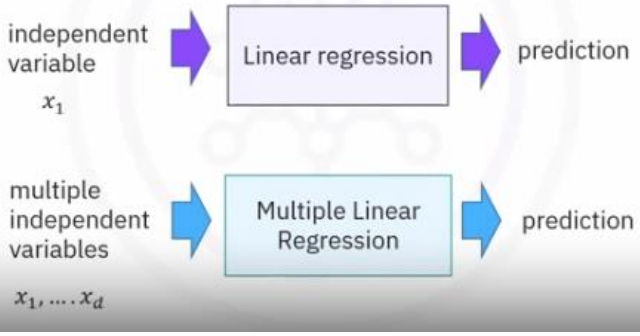
17/12/2023

……

11/01/2024

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

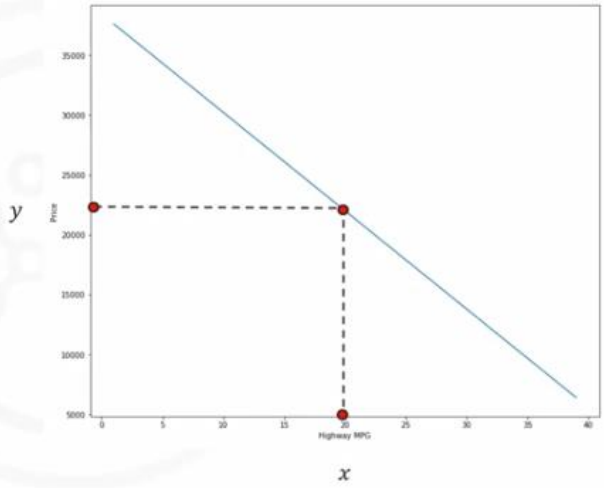
1. The predictor (independent) variable - x
2. The target (dependent) variable - y

$$y = b_0 + b_1 x$$

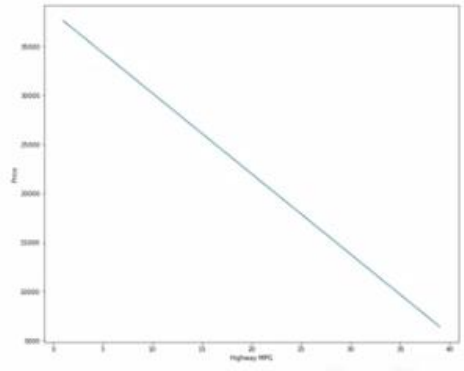
- b_0: **the intercept**
- b_1 : **the slope**

Simple linear regression: Prediction

$$y = 38423 - 821x$$
$$= 38423 - 821\,(20)$$
$$= 22\,003$$



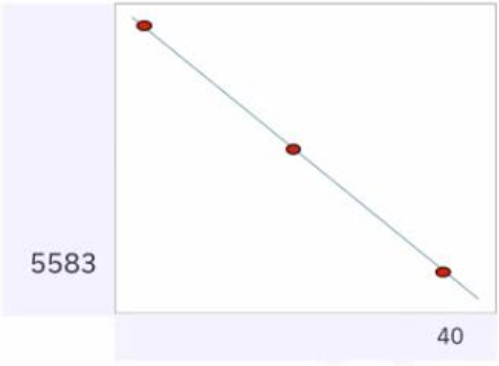
Simple linear regression: Fit



Fit

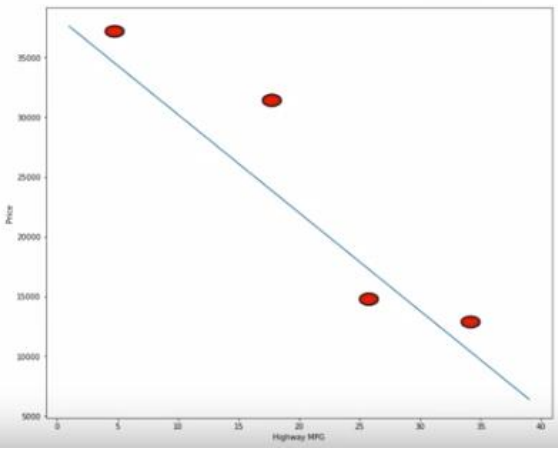
(b_0, b_1)

Simple linear regression: Fit



$$X = \begin{bmatrix} 0 \\ 20 \end{bmatrix} \qquad Y = \begin{bmatrix} 38423 \\ 22003 \end{bmatrix}$$

Simple linear regression

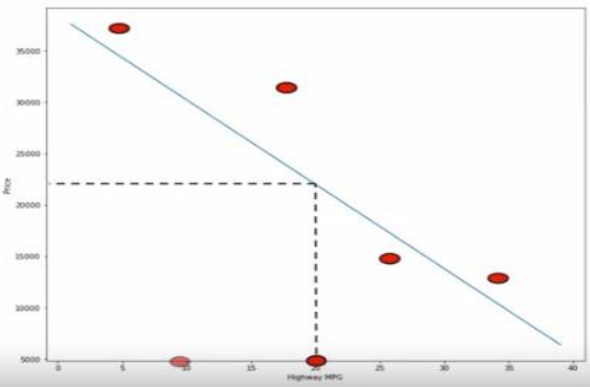


Fit

(b_0, b_1)

Predict

Simple linear regression



Fit

Predict

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

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
```

1. 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**

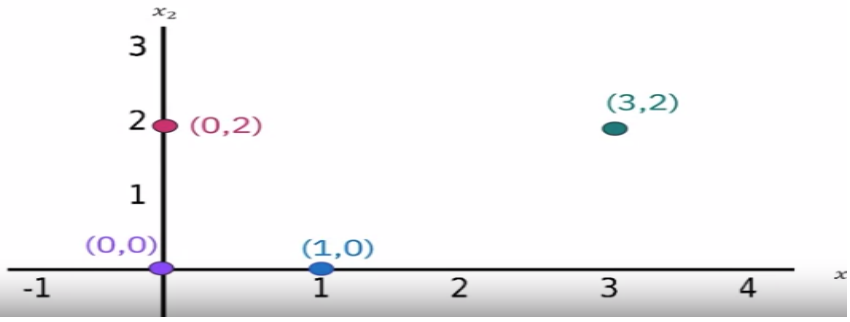
$$\hat{Y} = b_0 + b_1 x$$

Multiple Linear Regression (MLR)

$$\hat{Y} = b_0 + b_1 x_1 + b_2 x_2 + b_3 x_3 + b_4 x_4$$

- 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..

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

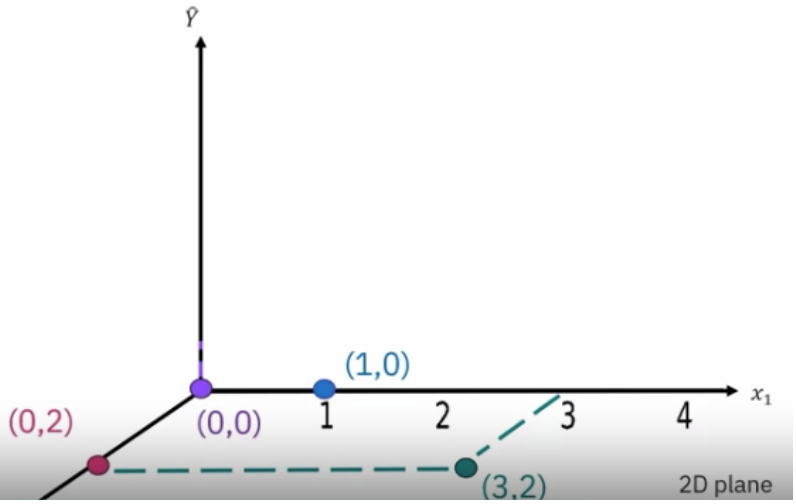


- This is shown below where

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

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

$\hat{Y}$
1
6
2
13



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
3	5	-4	3
:	:	:	:
2	4	2	-4

Yhat
2
:
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**

$$\hat{Y} = b_0 + b_1 x_1 + b_2 x_2 + b_3 x_3 + b_4 x_4$$

Regression plot

Why use regression plot?

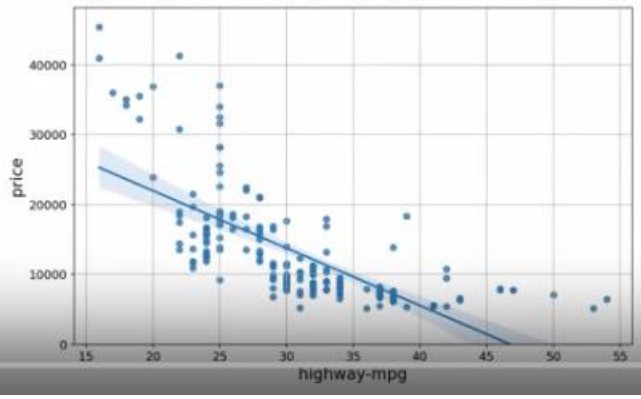
It gives us a good estimate of:

- The relationship between two variables
- The strength of the correlation
- The direction of the relationship (positive or negative)

Regression plot

```
import seaborn as sns
```

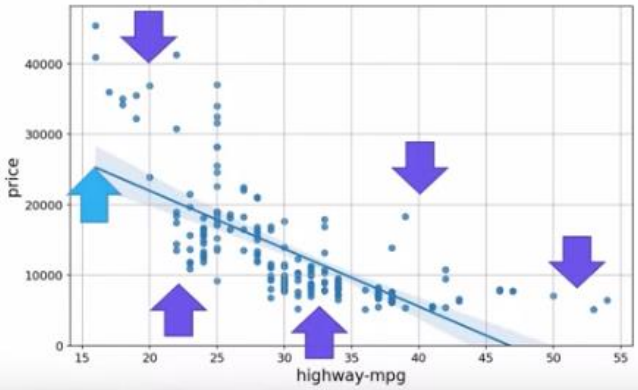
```
sns.regplot(x="highway-mpg", y="price", data=df)
plt.ylim(0,)
```



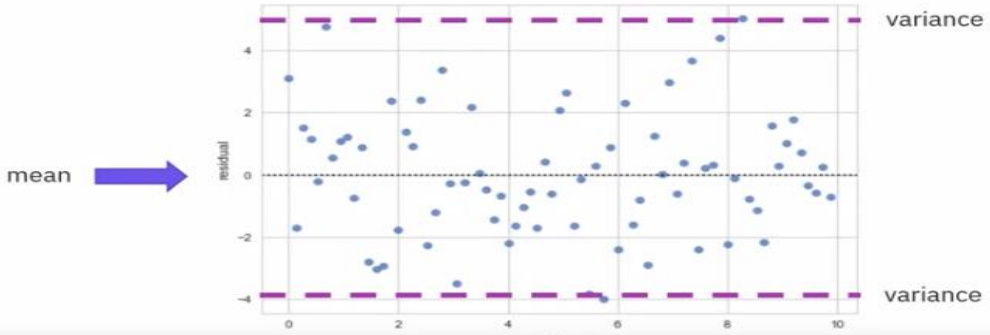
Regression plot

Regression Plot shows us a combination of:

- The scatterplot: where each point represents a different y
- The fitted linear regression line (ŷ)



Residual plot

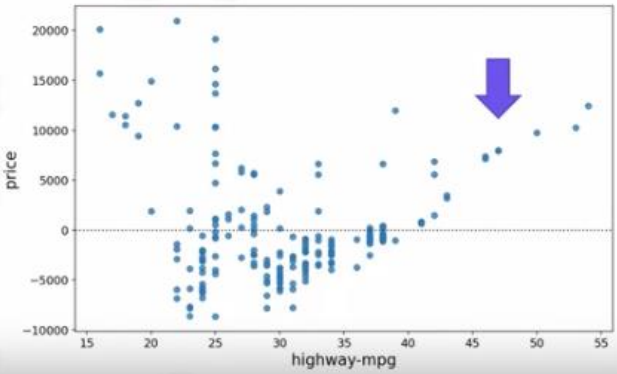


- Look at the **spread of the residuals**:
  - Randomly spread out around x-axis then a linear model is appropriate

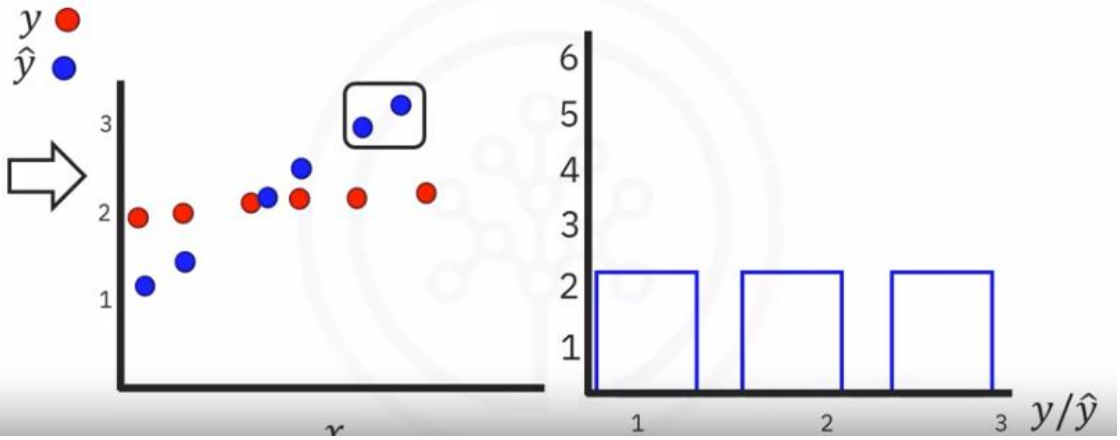
Residual plot

```
import seaborn as sns
```

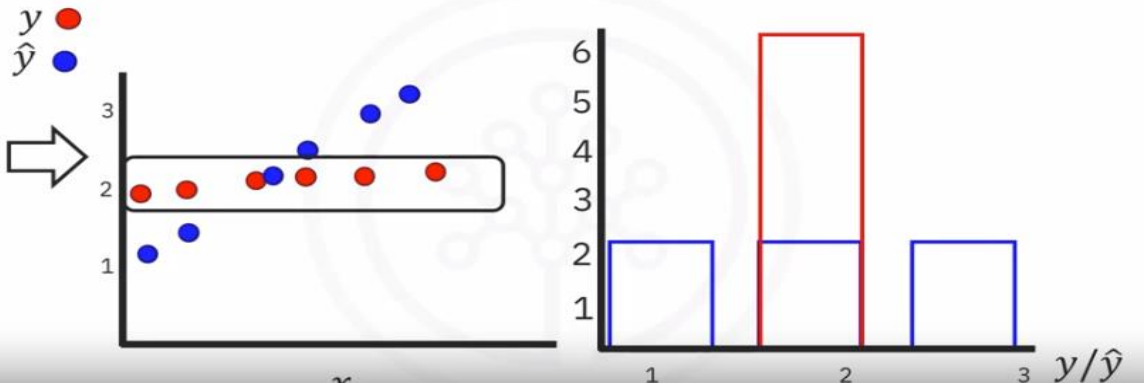
```
sns.residplot(df['highway-mpg'],
df['price'])
```



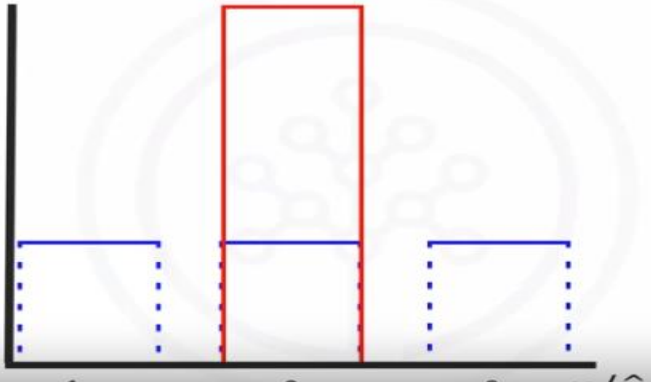
Distribution plots



Distribution plots



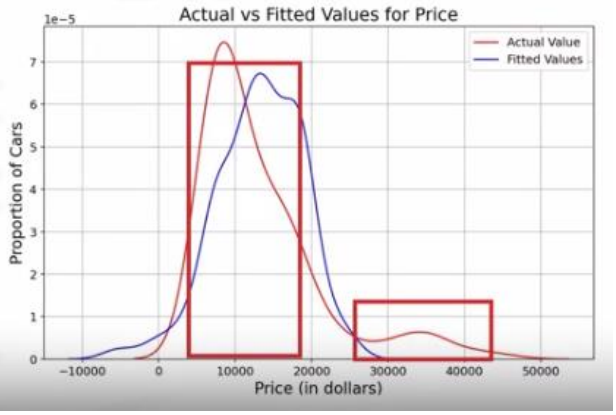
Distribution plots



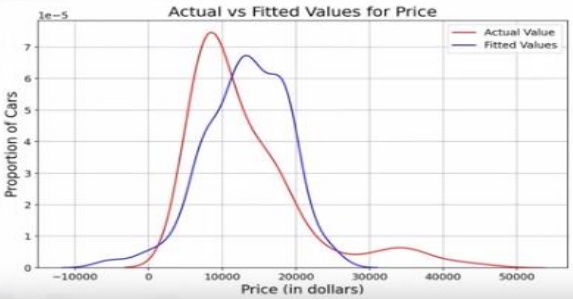
Distribution plots

Compare the distribution plots:

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



MLR: Distribution plots



Distribution plots

```
import seaborn as sns
```

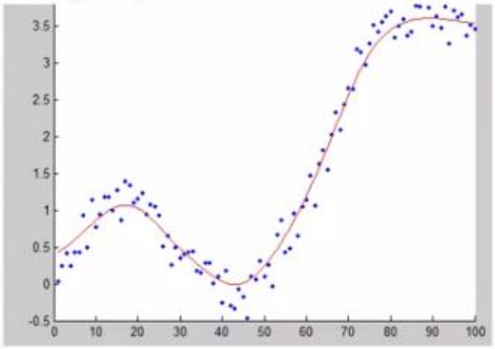
```
ax1 = sns.distplot(df['price'], hist=False, color="r", label="Actual Value")

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

Polynomial Regression

- 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

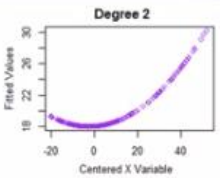
$$\hat{Y} = b_0 + b_1 x_1 + b_2 (x_1)^2$$

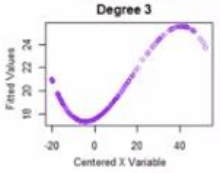
- Cubic – 3rd order

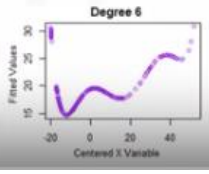
$$\hat{Y} = b_0 + b_1 x_1 + b_2 (x_1)^2 + b_3 (x_1)^3$$

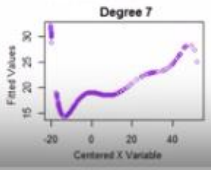
- Higher order

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









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)
```

$$-1.557(x_1)^3 + 204.8(x_1)^2 + 8965\,x_1 + 1.37\text{x}10^5$$

Polynomial Regression with more than 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']])
```

Pre-processing

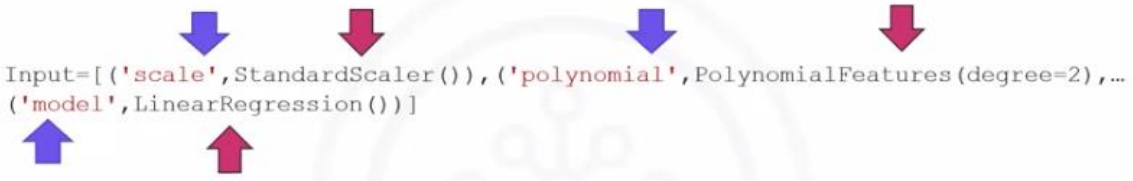
- For example we can Normalize the each feature simultaneously:

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

Pipelines

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

Pipeline constructor



- Pipeline constructor

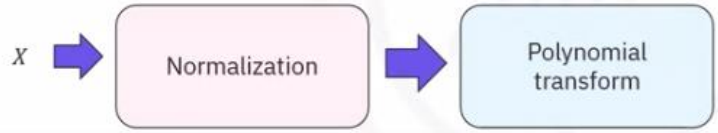
```
pipe=Pipeline(Input)
```

pipeline object

Pipeline constructor

- We can train the pipeline object

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

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



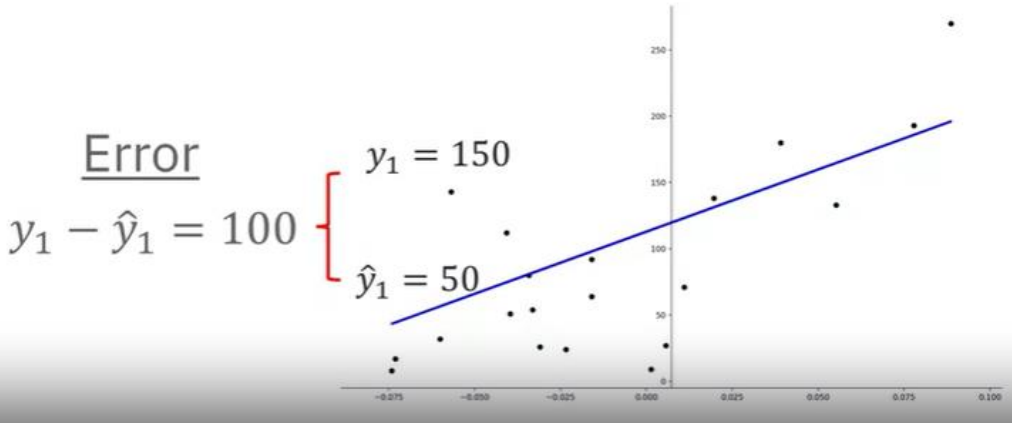
Measures for in-sample evaluation

- Measure how well the model fits our data

- 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:



Mean squared error (MSE)

Find the mean

$$\frac{(y_1 - \hat{y}_1)^2 + (y_2 - \hat{y}_2)^2 \ldots (y_n - \hat{y}_n)^2}{n}$$

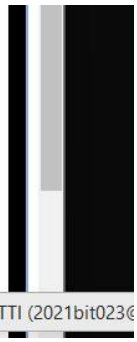
Mean squared error (MSE)

- In Python, we can measure the MSE as follows

```
from sklearn.metrics import mean_squared_error

mean_squared_error(actual_value,predicted_value)
```

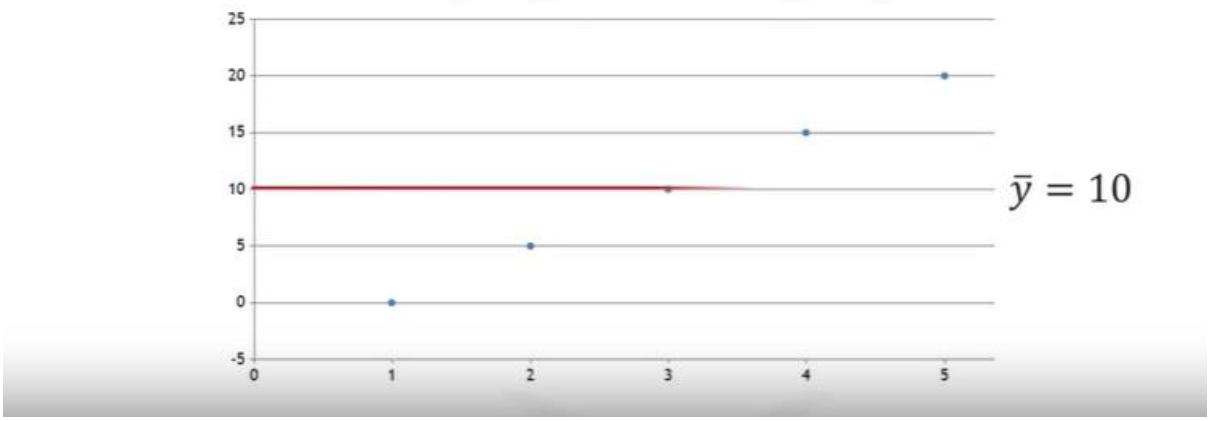
R-squared/ R^2

- The Coefficient of Determination or R squared (R^2)
  - Is a measure to determine how close the data is to the fitted regression line
- The base model is the average of the y-values
- R^2 compares the regression line to this average



Coefficient of Determination (R^2)

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

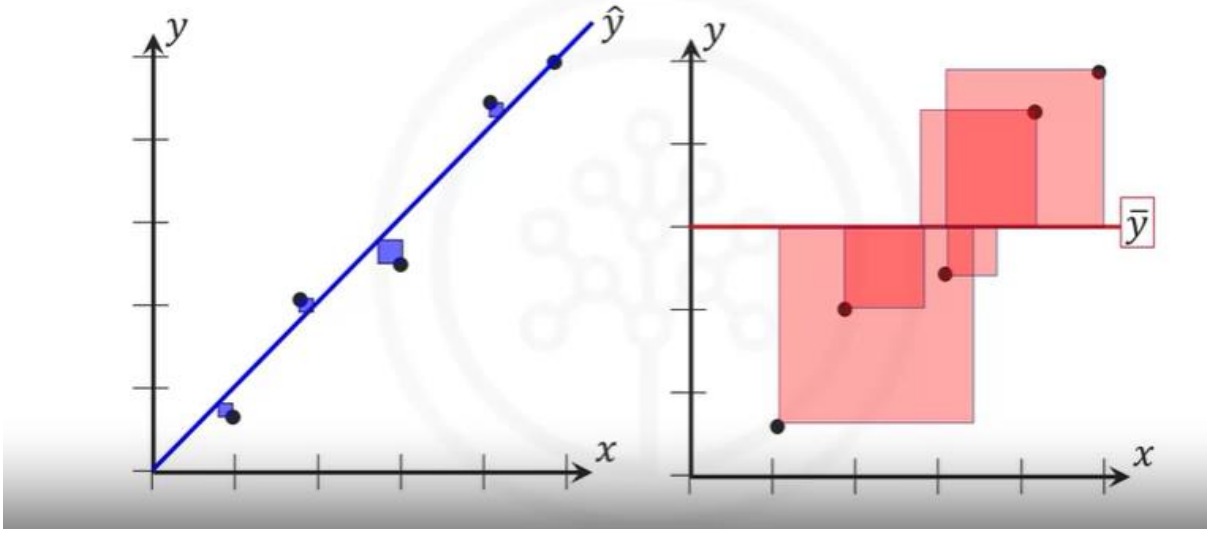


Coefficient of Determination (R^2)

$$R^2 = \left(1 - \frac{MSE(y, \hat{y})}{MSE(y, \bar{y})}\right)$$

R2= 0 or 1 inclusive

Coefficient of Determination (R^2)



Coefficient of Determination (R^2)

$$\frac{MSE(y, \hat{y})}{MSE(y, \bar{y})}$$

Nume—small, denome—big …

Coefficient of Determination (R^2)

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

$$= (1 - 1)$$

$$= 0$$

R-squared/ R^2

We can calculate the R^2 as follows

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

lm.fit(X, Y)

lm.score(X,y)
```

Output:
0.496591188 → 49.7%

Do the predicted values make sense?

- First we train the model

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

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

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

- Result: $ 13771.30

Do the predicted values make sense?

- First we train the model

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

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

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

- Result: $ 13771.30

```
lm.coef_
-821.73337832
```

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

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)
```

1	2	...	99	100

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,
         1445.30383367,    623.57045535,
        -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])
```

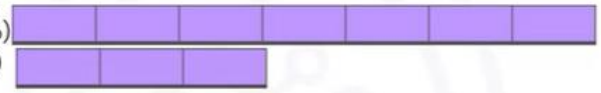
Comparing MLR and SLR

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 than regular regression
4. A similar inverse relationship holds for R^2

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 training_test_split()

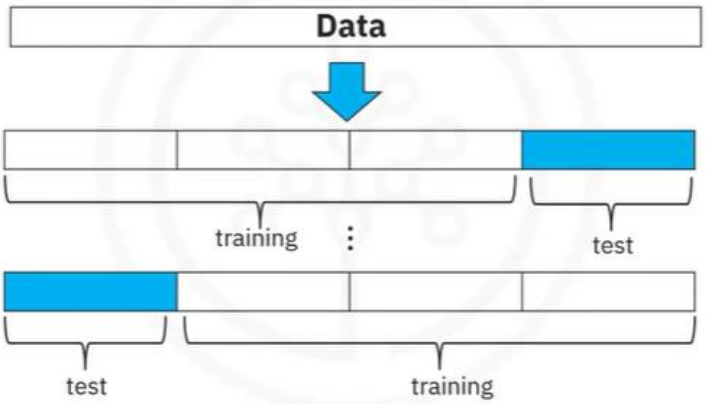
- Split data into random train and test subsets

```
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

Cross validation

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



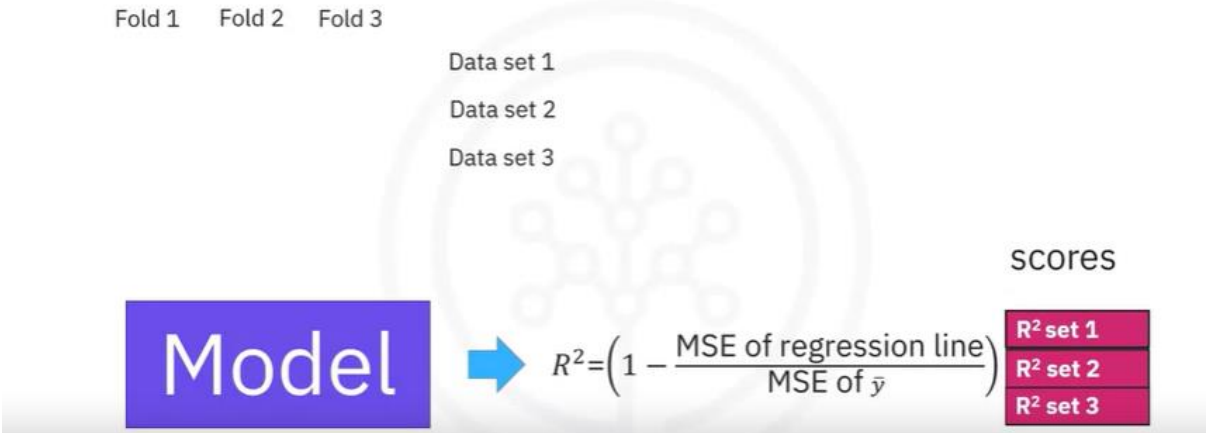
Function cross_val_score()

```
from sklearn.model_selection import cross_val_score


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

np.mean(scores)
```

Evaluating cross-validation score

2 fold for training and 1 fold for testing

Evaluating cross-validation score

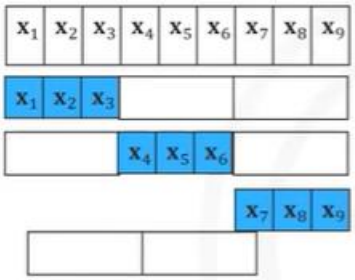


Function cross_val_predict()

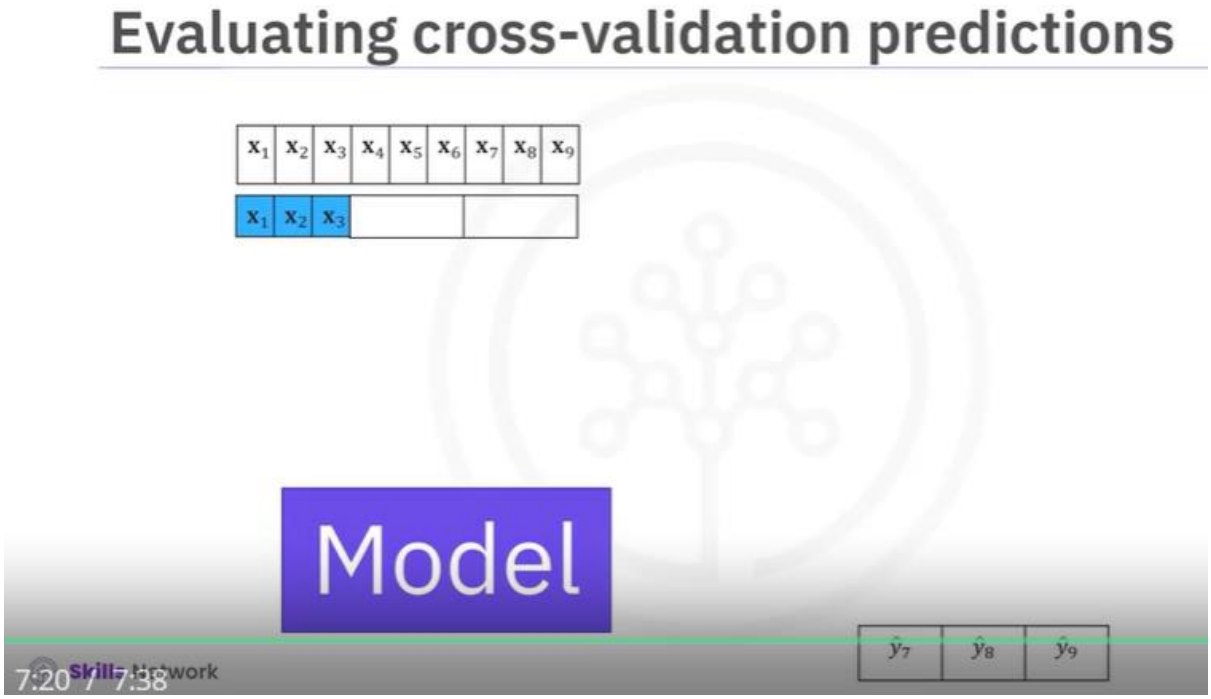
- It returns the prediction that was obtained for each element when it was in the test set.
- Has a similar interface to cross_val_score()

```
from sklearn.model_selection import cross_val_predict

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

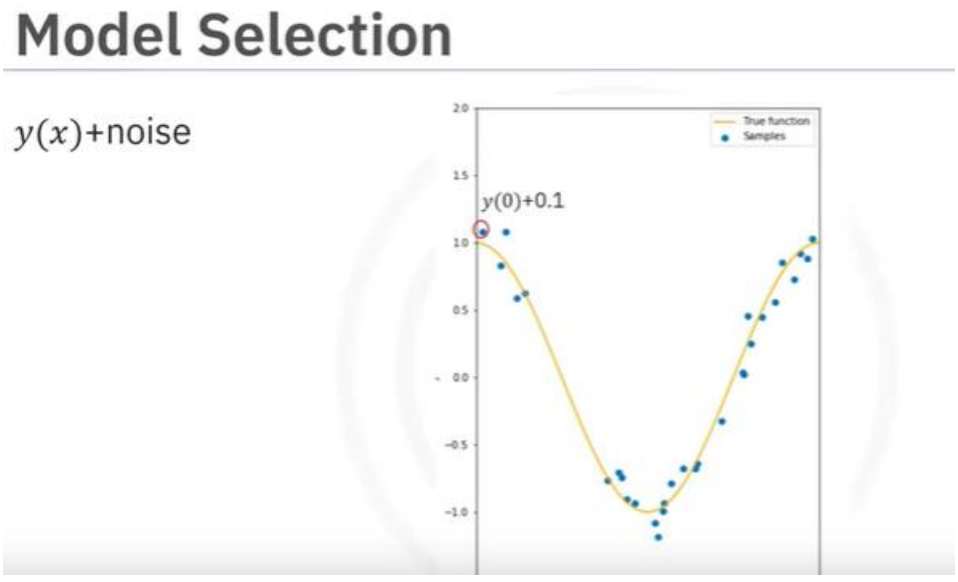
Evaluating cross-validation predictions



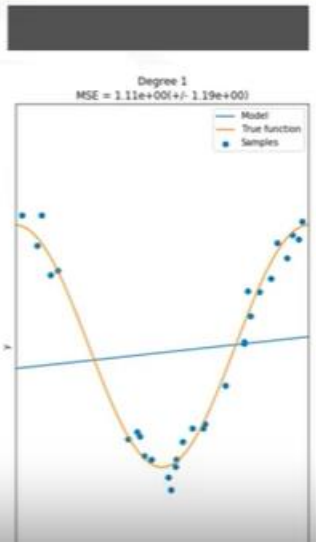
Model



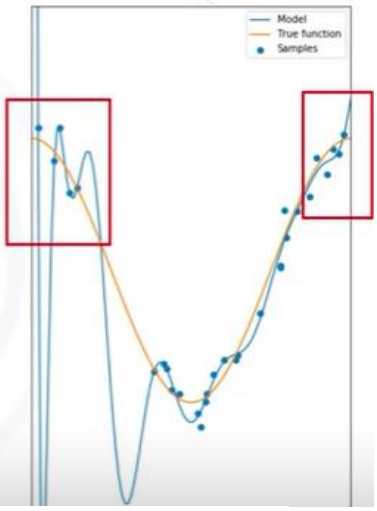
// overfitting and underfitting



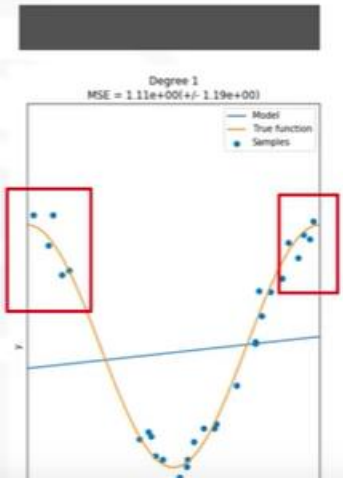
$$y = b_0 + b_1 x$$



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

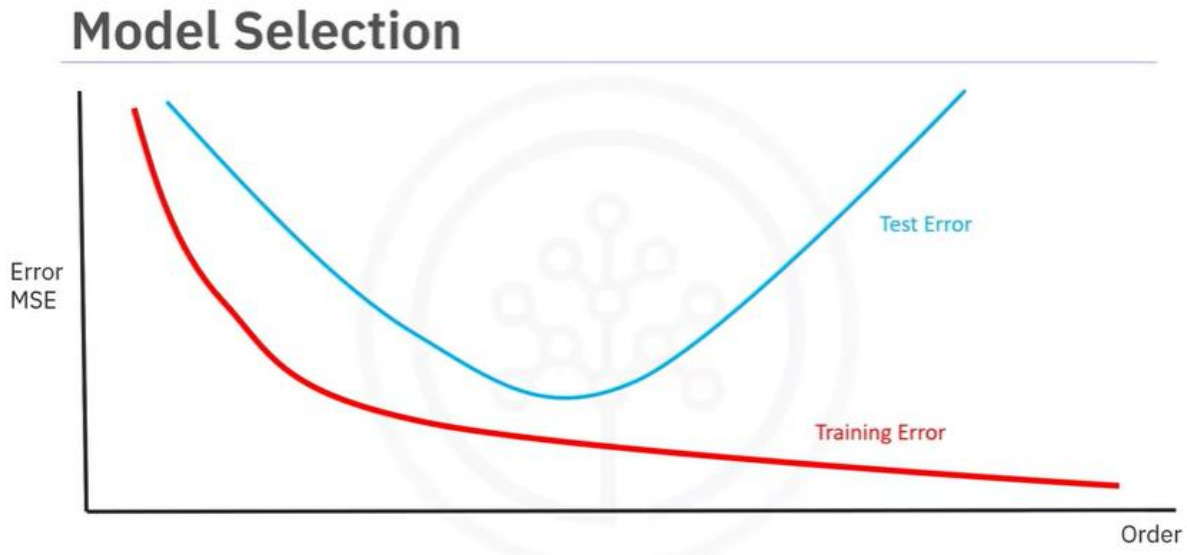


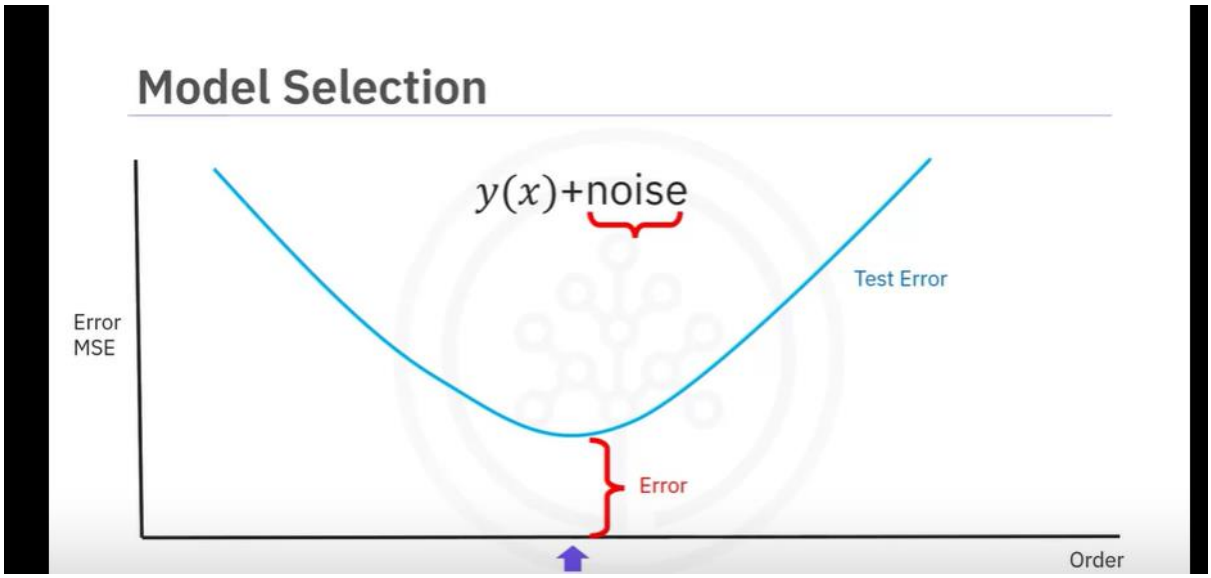
$$y = b_0 + b_1 x + b_2 x^2$$



Anything left to the graph→ underfitting.

Right→ Overfitting

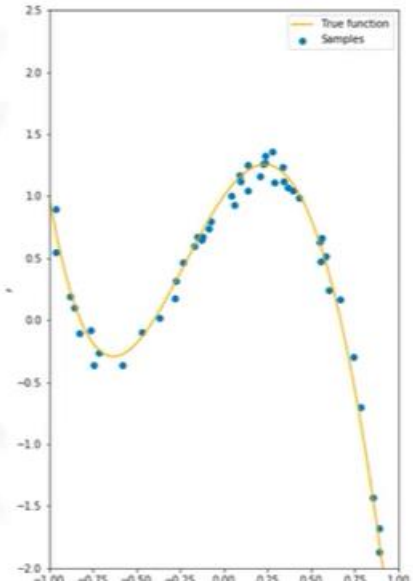




// ridge regression

Ridge Regression

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



Ridge Regression

```
from sklearn.linear_model import Ridge
RidgeModel=Ridge(alpha=0.1)
RidgeModel.fit(X,y)
Yhat=RidgeModel.predict(X)
```

Ridge Regression

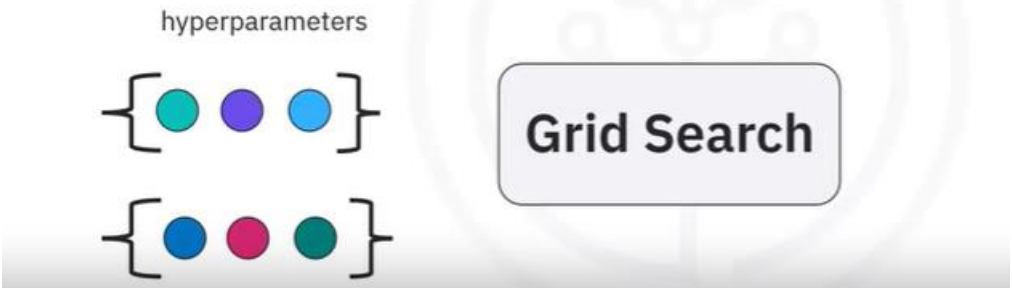
alpha

10

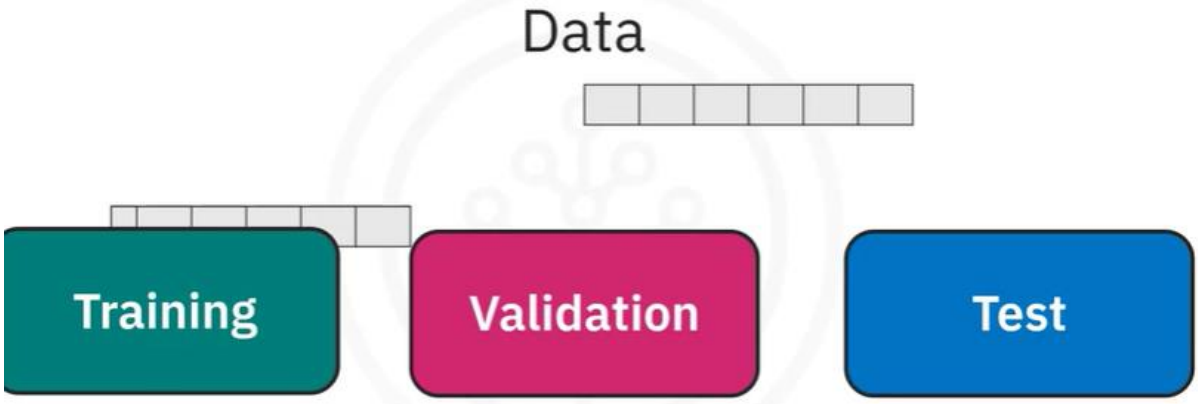
R^2
0.5
0.75

Hyperparameters

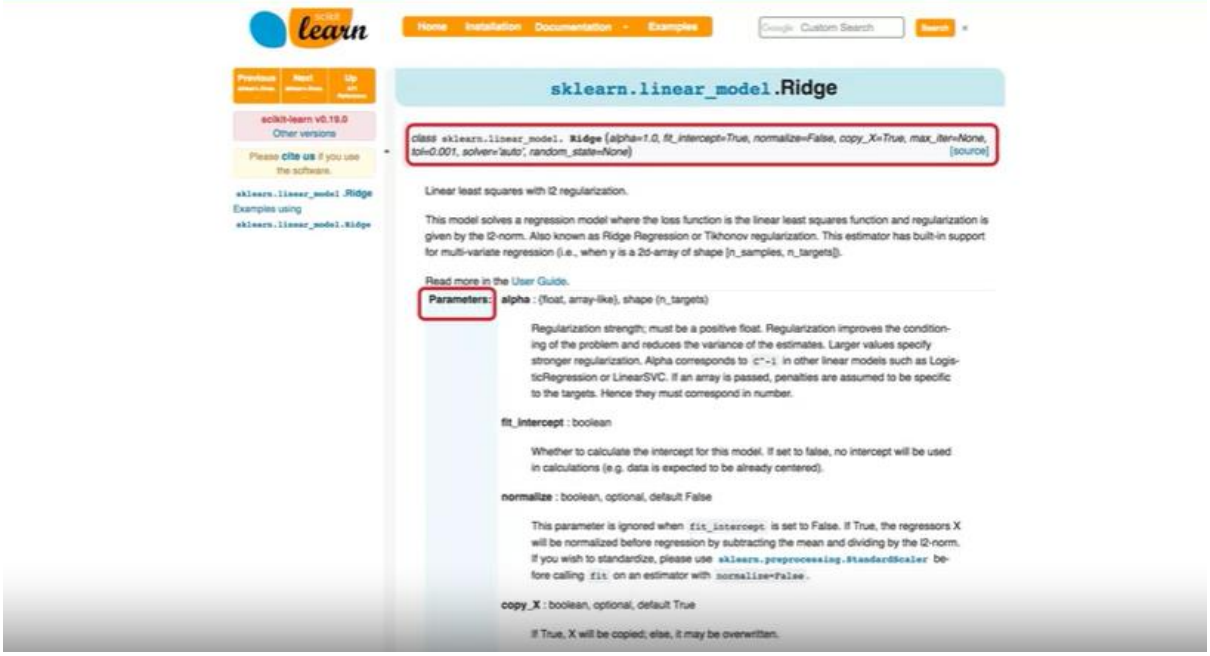
- Alpha in Ridge regression is called a hyperparameter



Grid Search



Data is get tested



Grid Search

Grid Search CV

Alpha	1	10	100	1000
R^2	0.74	0.35	0.073	0.008

```
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']
```