

IBM DA0101EN

## Analyzing Data with Python

Data Analysis with Python will be delivered through lectures, labs, and assignments. It includes the following parts:

Data Analysis libraries: will learn to use Pandas, Numpy, and Scipy libraries to work with a sample dataset. We will introduce you to pandas, an open-source library, and we will use it to load, manipulate, analyze, and visualize cool datasets. Then we will introduce you to another open-source library, scikit-learn, and we will use some of its machine learning algorithms to build smart models and make cool predictions.

### Course Overview

 Bookmark this page

### Course Overview

Learn how to analyze data using Python. This course will take you from the basics of Python to exploring many different types of data. You will learn how to prepare data for analysis, perform simple statistical analysis, create meaningful data visualizations, predict future trends from data, and more!

Topics covered:

- 1) Importing Datasets
- 2) Cleaning the Data
- 3) Data frame manipulation
- 4) Summarizing the Data
- 5) Building machine learning Regression models
- 6) Building data pipelines

## **Syllabus**

### **Module 1 – Importing Data Sets**

- The Problem
- Understanding the Data
- Practice Quiz
- Python Packages for Data Science
- Practice Quiz
- Importing and Exporting Data in Python
- Practice Quiz
- Getting Started Analyzing Data in Python
- Practice Quiz
- Accessing Databases with Python
- Lesson Summary
- Hands-on Lab: Importing Datasets
- Graded Quiz: Importing Datasets

### **Module 2 – Data Wrangling**

- Pre-processing Data in Python
- Dealing with Missing Values in Python
- Practice Quiz: Dealing with Missing Values in Python
- Data Formatting in Python
- Practice Quiz: Data Formatting in Python
- Data Normalization in Python
- Practice Quiz: Data Normalization in Python
- Binning in Python
- Turning Categorical Variables into Quantitative Variables in Python
- Practice Quiz: Turning categorical variables into quantitative variables in Python
- Lesson Summary
- Hands-on Lab: Data Wrangling
- Graded Quiz: Data Wrangling

## **Module 3 - Exploratory Data Analysis**

- Exploratory Data Analysis
- Descriptive Statistics
- Practice Quiz: Descriptive Statistics
- GroupBy in Python
- Practice Quiz: GroupBy in Python
- Correlation
- Practice Quiz: Correlation
- Correlation - Statistics
- Practice Quiz: Correlation - Statistics
- Association between two categorical variables: Chi-Square
- Lesson Summary
- Hands-on Lab: Exploratory Data Analysis
- Graded Quiz: Exploratory Data Analysis

## **Module 4 – Model Development**

- Model Development
- Linear Regression and Multiple Linear Regression
- Practice Quiz: Linear Regression and Multiple Linear Regression
- Model Evaluation using Visualization
- Practice Quiz: Model Evaluation using Visualization
- Polynomial Regression and Pipelines
- Practice Quiz: Polynomial Regression and Pipelines
- Measures for In-Sample Evaluation
- Practice Quiz: Measures for In-Sample Evaluation
- Prediction and Decision Making
- Lesson Summary
- Hands-on Lab: Model Development
- Graded Quiz: Model Development

## **Module 5 - Model Evaluation**

- Model Evaluation and Refinement
- Practice Quiz: Model Evaluation
- Overfitting, Underfitting, and Model Selection
- Practice Quiz: Overfitting, Underfitting and Model Selection
- Ridge Regression Introduction
- Ridge Regression
- Practice Quiz: Ridge Regression
- Grid Search
- Lesson Summary
- Hands-on Lab: Model Evaluation and Refinement
- Graded Quiz: Model Refinement

## **Module 6 - Final Assignment**

- Project Case Scenario
- Project Overview
- (Optional) Create or Login into IBM cloud to use Watson Studio.
- Share your Jupyter Notebook
- Peer Review: House Sales in King County, USA
- Final Exam
- Credits and Acknowledgments

## Grading Scheme

 [Bookmark this page](#)

### GRADING SCHEME

This section contains information for those earning a certificate. Those auditing the course can skip this section and click next.

- This course contains 5 Graded Quizzes, 1 per module, as well as a Final Assignment and a Final Exam.
  - The Graded Quizzes carries a weight of 50% of the total grade.
  - The Final Assignment carries a weight of 14% of the total grade.
  - The Final Exam carries a weight of 36% of the total grade.
- The minimum passing mark for the **course** is 70%.
- Permitted attempts are per **question**:
  - One attempt - For True/False questions
  - Two attempts - For any question other than True/False
- There are no penalties for incorrect attempts.
- Check your grades in the course at any time by clicking on the "Progress" tab.

## Copyrights and Trademarks

 [Bookmarked](#)

### Copyrights and Trademarks

IBM®, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at: [ibm.com/legal/copytrade.shtml](http://ibm.com/legal/copytrade.shtml)

References to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates.

Other product, company or service names may be trademarks or service marks of others.

## Module Introduction & Learning Objectives

 Bookmarked

### Module Introduction

In this week's module, you will learn how to understand data and learn about how to use the libraries in Python to help you import data from multiple sources. You will then learn how to perform some basic tasks to start exploring and analyzing the imported data set.

### Learning Objectives

- Analyze Python data using a dataset
- Identify three Python libraries and describe their uses
- Read data using Python's Pandas package
- Demonstrate how to import and export data in Python

## The Problem

In this video, we'll be talking about data analysis and the scenario in which we'll be playing the data analyst or data scientist.

But before we begin, talking about the problem, used car prices, we should first understand **the importance of data analysis**. As you know, data is collected everywhere around us. Whether it's collected manually by scientists or collected digitally, every time you click on a website, or your mobile device.

But data does not mean information. **Data analysis and, in essence, data science, helps us unlock the information and insights from raw data to answer our questions. So data analysis plays an important role by helping us to discover useful information from the data, answer questions, and even predict the future or the unknown. So let's begin with our scenario.**

Let's say we have a friend named Tom. And Tom wants to sell his car. But the problem is he doesn't know how much he should sell his car for. Tom wants to sell his car for as much as he can. But he also wants to set the price reasonably, so someone would want to purchase it. So the price he sets should represent the value of the car. How can we help Tom determine the best price for his car?

Let's think like data scientists and clearly define some of his problems. For example, is there data on the prices of other cars and their characteristics? What features of cars affect their prices? Color? Brand?

Does horsepower also effect the selling price, or perhaps something else? As a data analyst or data scientist, these are some of the questions we can start thinking about.

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

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

## The Problem

## Why Data Analysis?

- **Data is everywhere.**
- **Data analysis/data science helps us answer questions from data.**
- **Data analysis plays an important role in:**
  - Discovering useful information
  - Answering questions
  - Predicting future or the unknown

## Tom wants to sell his car



How much  
money should he  
sell his car for?

The price he sets should not be too high,  
but not too low either.

## Estimate used car prices

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

- Is there data on the prices of other cars and their characteristics?
- What features of cars affect their prices?
  - Color? Brand? Horsepower? Something else?
- Asking the right questions in terms of data



## Data on used car prices

```
3,7,alfa-romero,gas,std,two,convertible,rwd,front,88.60,169.80,64.10,48.80,2548,dohc,four,130,mpfi,2.47,2.68,9.00,111,5000,21,27,13495
3,7,alfa-romero,gas,std,two,convertible,rwd,front,88.60,168.80,64.10,48.80,2548,dohc,four,130,mpfi,2.47,2.68,9.00,111,5000,21,27,16500
1,7,alfa-romero,gas,std,two,hatchback,rwd,front,94.50,171.20,65.50,52.40,2823,ohc,six,152,mpfi,2.68,3.47,9.00,154,5000,19,26,16500
2,164,audi,gas,std,four,sedan,fwd,front,99.80,176.60,66.20,54.30,2337,ohc,four,109,mpfi,3.19,3.40,10.60,102,5500,24,30,13950
2,164,audi,gas,std,four,sedan,fwd,front,99.40,176.60,66.40,54.30,2824,ohc,five,136,mpfi,3.19,3.40,8.00,115,5500,18,22,17450
2,7,audi,gas,std,two,sedan,fwd,front,99.80,177.30,66.30,53.10,2507,ohc,five,136,mpfi,3.19,3.40,8.50,110,5500,19,25,15250
1,158,audi,gas,std,four,sedan,fwd,front,105.80,192.70,71.40,55.70,2844,ohc,five,136,mpfi,3.19,3.40,8.50,110,5500,19,25,17710
1,7,audi,gas,std,four,wagon,fwd,front,105.80,192.70,71.40,55.70,2954,ohc,five,136,mpfi,3.19,3.40,8.50,110,5500,19,25,1892
1,158,audi,gas,turbo,four,sedan,fwd,front,105.80,192.70,71.40,55.90,3086,ohc,five,131,mpfi,3.13,3.40,8.30,140,5500,17,20,23875
0,7,audi,gas,turbo,two,hatchback,4wd,front,99.50,178.20,67.90,52.00,3053,ohc,five,131,mpfi,3.13,3.40,7.00,160,5500,16,22,?
2,192,bmw,gas,std,two,sedan,rwd,front,101.20,176.80,64.80,54.30,2304,ohc,four,108,mpfi,3.50,2.80,8.80,101,5800,23,29,16430
0,121,bmw,gas,std,four,sedan,rwd,front,101.20,176.80,64.80,54.30,3394,ohc,four,108,mpfi,3.50,2.80,8.80,101,5800,23,29,16430
0,188,bmw,gas,std,four,sedan,rwd,front,101.20,176.80,64.80,54.30,2710,ohc,five,164,mpfi,3.31,3.19,9.00,121,4250,21,28,2099
0,188,bmw,gas,std,four,sedan,rwd,front,101.20,176.80,64.80,54.30,2745,ohc,six,164,mpfi,3.31,3.19,9.00,121,4250,21,28,21105
1,7,bmw,gas,std,four,sedan,rwd,front,103.50,189.00,66.90,55.70,3055,ohc,six,164,mpfi,3.31,3.19,9.00,121,4250,20,25,24565
0,7,bmw,gas,std,four,sedan,rwd,front,103.50,189.00,66.90,55.70,3230,ohc,six,209,mpfi,3.62,3.39,8.00,182,5400,16,22,30760
0,7,bmw,gas,std,two,sedan,rwd,front,103.50,193.80,67.90,53.70,3380,ohc,six,209,mpfi,3.62,3.19,8.00,182,5400,16,22,41315
0,7,bmw,gas,std,four,sedan,rwd,front,110.00,197.00,70.90,56.30,3505,ohc,six,209,mpfi,3.62,3.39,8.00,182,5400,15,20,36880
2,121,chevrolet,gas,std,two,hatchback,fwd,front,88.40,141.10,60.30,53.20,1488,1,three,61,2bb1,2.91,3.03,9.50,48,5100,47,53,5151
```

<https://archive.ics.uci.edu/ml/machine-learning-databases/autos/>

## Understanding the Data

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

Each line represents a row in the dataset. In the hands-on lab for this module, you'll be able to download and use the CSV file.

Do you notice anything different about the first row? Sometimes the first row is a header, which contains a column name for each of the 26 columns. But in this example, it's just another row of data.

So, here's the documentation on what each of the 26 columns represent. There are a lot of columns and I'll just go through a few of the column names, but you can also check out the link at the bottom of the slide to go through the descriptions yourself.

The first attribute, symboling, corresponds to the insurance risk level of a car. Cars are initially assigned a risk factor symbol associated with their price. Then, if an automobile is more risky, this symbol is adjusted by moving it up the scale.

A value of plus three indicates that the auto is risky. Minus three, that is probably pretty safe. The second attribute, normalized-losses, is the relative average loss payment per insured vehicle year. This value is normalized for all autos within a particular size classification, two door small, station wagons, sports specialty, etc., and represents the average loss per car per year.

The values range from 65 to 256. The other attributes are easy to understand. If you would like to check out more details, refer to the link at the bottom of the slide. Okay, after we understand the meaning of each feature, we'll notice that the 26 attribute is price.

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

Thus, the goal of this project is to predict price in terms of other car features. Just a quick note. This dataset is actually from 1985.

So, the car prices for the models may seem a little low. But just bear in mind that the goal of this exercise is to learn how to analyze the data.

## Understanding the Data

# Dataset – Used Automobiles (CSV)

```
3,7,alfa-romero,gas,std,two,convertible,rwd,front,88.60,168.80,64.10,48.80,2548,dohc,four,130,mpfi,3.47,2.68,9.00,111,5000,21,27,12495
3,7,alfa-romero,gas,std,two,convertible,rwd,front,88.60,168.80,64.10,48.80,2548,dohc,four,130,mpfi,3.47,2.68,9.00,111,5000,21,27,16500
1,7,alfa-romero,gas,std,two,hatchback,rwd,front,94.50,171.20,65.50,52.40,2823,ohcv,six,152,mpfi,2.68,3.47,9.00,154,5000,19,26,16500
2,164,audi,gas,std,four,sedan,fwd,front,99.80,176.60,66.20,54.30,2337,ohc,four,109,mpfi,3.19,3.40,10.00,102,5500,24,30,13950
2,164,audi,gas,std,four,sedan,fwd,front,99.80,176.60,66.40,54.30,2824,ohc,five,136,mpfi,3.19,3.40,8.00,115,5500,18,22,17450
2,?,audi,gas,std,two,sedan,fwd,front,99.80,177.30,66.30,53.10,2507,ohc,five,136,mpfi,3.19,3.40,8.50,110,5500,19,25,15250
1,158,audi,gas,std,four,sedan,fwd,front,105.80,192.70,71.40,55.70,2844,ohc,five,136,mpfi,3.19,3.40,8.50,110,5500,19,25,17710
1,7,audi,gas,std,four,wagon,fwd,front,105.80,192.70,71.40,55.70,2954,ohc,five,136,mpfi,3.19,3.40,8.50,110,5500,19,25,18920
1,158,audi,gas,turbo,four,sedan,fwd,front,105.80,192.70,71.40,55.90,3086,ohc,five,131,mpfi,3.13,3.40,8.30,140,5500,17,20,23875
0,?,audi,gas,turbo,two,hatchback,fwd,front,99.50,178.20,67.90,52.00,3053,ohc,five,131,mpfi,3.13,3.40,7.00,160,5500,16,22,?
2,192,bmw,gas,std,two,sedan,rwd,front,101.20,176.80,64.80,54.30,2395,ohc,four,108,mpfi,3.50,2.80,8.80,101,5800,23,29,16430
0,192,bmw,gas,std,four,sedan,rwd,front,101.20,176.80,64.80,54.30,2395,ohc,four,108,mpfi,3.50,2.80,8.80,101,5800,23,29,16925
0,188,bmw,gas,std,two,sedan,rwd,front,101.20,176.80,64.80,54.30,2710,ohc,six,164,mpfi,3.31,3.19,9.00,121,4250,21,28,20970
0,188,bmw,gas,std,four,sedan,rwd,front,101.20,176.80,64.80,54.30,2765,ohc,six,164,mpfi,3.31,3.19,9.00,121,4250,21,28,21105
1,7,bmw,gas,std,four,sedan,rwd,front,103.50,189.00,66.90,55.70,3055,ohc,six,164,mpfi,3.31,3.19,9.00,121,4250,20,25,24565
0,7,bmw,gas,std,four,sedan,rwd,front,103.50,189.00,66.90,55.70,3230,ohc,six,209,mpfi,3.62,3.39,8.00,182,5400,16,22,30760
0,?,bmw,gas,std,two,sedan,rwd,front,103.50,193.80,67.90,53.70,3380,ohc,six,209,mpfi,3.62,3.39,8.00,182,5400,16,22,41315
0,7,bmw,gas,std,four,sedan,rwd,front,110.00,197.00,70.90,56.30,3505,ohc,six,209,mpfi,3.62,3.39,8.00,182,5400,15,20,36880
2,121,chevrolet,gas,std,two,hatchback,fwd,front,88.40,141.10,60.30,53.20,1488,1,three,61,2bbl,2.91,3.03,9.50,48,5100,47,53,5151
```

The dataset used in this course is an open dataset by Jeffrey C. Schlemmer. This dataset is in CSV format, which separates each of the values with commas, making it very easy to import in most tools or applications. Each line represents a row in the dataset.

## Each of the attributes in the dataset

No.	Attribute name	attribute range	No.	Attribute name	attribute range
1	symboling	-3, -2, -1, 0, 1, 2, 3.	14	curb-weight	continuous from 1488 to 4066.
2	normalized-losses	continuous from 65 to 256.	15	engine-type	dohc, dohcvt, i, ohc, ohcf, ohcv, rotor.
3	make	audi, bmw, etc.	16	num-of-cylinders	eight, five, four, six, three, twelve, two.
4	fuel-type	diesel, gas.	17	engine-size	continuous from 61 to 326.
5	aspiration	std, turbo.	18	fuel-system	1bbl, 2bbl, 4bbl, idl, mfi, mpfi, spdi, spfi.
6	num-of-doors	four, two.	19	bore	continuous from 2.54 to 3.94.
7	body-style	hardtop, wagon, etc..	20	stroke	continuous from 2.07 to 4.17.
8	drive-wheels	4wd, fwd, rwd.	21	compression-ratio	continuous from 7 to 23.
9	engine-location	front, rear.	22	horsepower	continuous from 48 to 288.
10	wheel-base	continuous from 86.6 120.9.	23	peak-rpm	continuous from 4150 to 6600.
11	length	continuous from 141.1 to 208.1.	24	city-mpg	continuous from 13 to 49.
12	width	continuous from 60.3 to 72.3.	25	highway-mpg	continuous from 16 to 54.
13	height	continuous from 47.8 to 59.8.	26	price	continuous from 5118 to 45400.

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

## Each of the attributes in the dataset

No.	Attribute name	attribute range	No.	Attribute name	attribute range
1	symboling	-3, -2, -1, 0, 1, 2, 3.	14	curb-weight	continuous from 1488 to 4066.
2	normalized-losses	continuous from 65 to 256.	15	engine-type	dohc, dohcvt, i, ohc, ohcf, ohcv, rotor.
3	make	audi, bmw, etc.	16	num-of-cylinders	eight, five, four, six, three, twelve, two.
4	fuel-type	diesel, gas.	17	engine-size	continuous from 61 to 326.
5	aspiration	std, turbo.	18	fuel-system	1bbl, 2bbl, 4bbl, idl, mfi, mpfi, spdi, spfi.
6	num-of-doors	four, two.	19	bore	continuous from 2.54 to 3.94.
7	body-style	hardtop, wagon, etc.	20	stroke	continuous from 2.07 to 4.17.
8	drive-wheels	4wd, fwd, rwd.	21	compression-ratio	continuous from 7 to 23.
9	engine-location	front, rear.	22	horsepower	continuous from 48 to 288.
10	wheel-base	continuous from 86.6 to 120.9.	23	peak-rpm	continuous from 4150 to 6600.
11	length	continuous from 141.1 to 208.1.	24	city-mpg	continuous from 13 to 49.
12	width	continuous from 60.3 to 72.3.	25	highway-mpg	continuous from 16 to 54.
13	height	continuous from 47.8 to 59.8.	26	price	continuous from 5118 to 45400.

The second attribute, **normalized-losses**, is the relative average loss payment per insured vehicle year. This value is normalized for all autos within a particular size classification, two door small, station wagons, sports specialty, etc., and represents the average loss per car per year. The values range from 65 to 256.

## Each of the attributes in the dataset

No.	Attribute name	attribute range	No.	Attribute name	attribute range
1	symboling	-3, -2, -1, 0, 1, 2, 3.	14	curb-weight	continuous from 1488 to 4066.
2	normalized-losses	continuous from 65 to 256.	15	engine-type	dohc, dohcvt, i, ohc, ohcf, ohcv, rotor.
3	make	audi, bmw, etc.	16	num-of-cylinders	eight, five, four, six, three, twelve, two.
4	fuel-type	diesel, gas.	17	engine-size	continuous from 61 to 326.
5	aspiration	std, turbo.	18	fuel-system	1bbl, 2bbl, 4bbl, idl, mfi, mpfi, spdi, spfi.
6	num-of-doors	four, two.	19	bore	continuous from 2.54 to 3.94.
7	body-style	hardtop, wagon, etc.	20	stroke	continuous from 2.07 to 4.17.
8	drive-wheels	4wd, fwd, rwd.	21	compression-ratio	continuous from 7 to 23.
9	engine-location	front, rear.	22	horsepower	continuous from 48 to 288.
10	wheel-base	continuous from 86.6 to 120.9.	23	peak-rpm	continuous from 4150 to 6600.
11	length	continuous from 141.1 to 208.1.	24	city-mpg	continuous from 13 to 49.
12	width	continuous from 60.3 to 72.3.	25	highway-mpg	continuous from 16 to 54.
13	height	continuous from 47.8 to 59.8.	26	price	continuous from 5118 to 45400.

Target (Label)

This means price is the value that we want to predict from the dataset and the predictors should be all the other variables listed like symboling, normalized-losses, make, and so on. Thus, the goal of this project is to predict price in terms of other car features. Just a quick note. This dataset is actually from 1985.

## Practice Quiz: Understanding the Data

Bookmark

### Question 1

1/1 point (ungraded)

Each column contains a:

different used car

attribute or feature



#### Answer

Correct: Correct

Submit

You have used 1 of 1 attempt

---

✓ Correct (1/1 point)

# Python Packages for Data Science

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

A Python library is a collection of functions and methods that allow you to perform lots of actions without writing any code. The libraries usually contain built in modules providing different functionalities which you can use directly.

And there are extensive libraries offering a broad range of facilities. We have divided the Python data analysis libraries into three groups.

1. The first group is called scientific computing libraries.

1.1 Pandas offers data structure and tools for effective data manipulation and analysis.

It provides facts, access to structured data. The primary instrument of Pandas is the two dimensional table consisting of column and row labels, which are called a data frame. It is designed to provide easy indexing functionality.

1.2 The NumPy library uses arrays for its inputs and outputs. It can be extended to objects for matrices and with minor coding changes, developers can perform fast array processing.

1.3 SciPy includes functions for some advanced math problems as listed on this slide, as well as data visualization.

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

2.1 The Matplotlib package is the most well known library for data visualization. It is great for making graphs and plots. The graphs are also highly customizable.

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

With machine learning algorithms, we're able to develop a model using our data set and obtain predictions.

3. The algorithmic libraries tackles the machine learning tasks from basic to complex.

Here we introduce two packages,

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

3.2. Statsmodels is also a Python module that allows users to explore data, estimate statistical models and perform statistical tests.

Python Packages for  
Data Science

## Scientific Computing Libraries in Python

### 1. Scientifics Computing Libraries



## Visualization Libraries in Python

### 2. Visualization Libraries



## Algorithmic Libraries in Python

### 3. Algorithmic libraries



## Practice Quiz: Python Packages for Data Science

Bookmarked

### Question 1

1/1 point (ungraded)

What description best describes the library Pandas?

- Includes functions for some advanced math problems as listed in the slide as well as data visualization.
- Uses arrays as their inputs and outputs. It can be extended to objects for matrices, and with a little change of coding, developers perform fast array processing.
- Offers data structure and tools for effective data manipulation and analysis. It provides fast access to structured data. The primary instrument of Pandas is a two-dimensional table consisting of columns and rows labels which are called a DataFrame. It is designed to provide an easy indexing function.



#### Answer

Correct: Correct

Submit

You have used 1 of 2 attempts

Reset

Show answer

---

✓ Correct (1/1 point)

# Importing and Exporting Data in Python

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

then we can perform all the subsequent data analysis procedures we need. Data acquisition is a process of loading and reading data into notebook from various sources. To read any data using Python's pandas package, there are two important factors to consider, format and file path. Format is the way data is encoded.

We can usually tell different encoding schemes by looking at the ending of the file name.

Some common encodings are: CSV, JSON, XLSX, HDF and so forth. The path tells us where the data is stored.

Usually, it is stored either on the computer we are using or online on the internet. In our case, we found a dataset of used cars which was obtained from the web address shown on the slide.

When Jerry entered the web address in his web browser, he saw something like this. Each row is one datapoint.

A large number of properties are associated with each datapoint. Because the properties are separated from each other by commas, we can guess the data format is CSV, which stands for comma separated values.

At this point, these are just numbers and don't mean much to humans, but once we read in this data we can try to make more sense out of it. In pandas, the read\_CSV method can read in files with columns separated by commas into a pandas data frame.

Reading data in pandas can be done quickly in three lines. First, import pandas, then define a variable with a file path and then use the read\_CSV method to import the data. However, read\_CSV assumes the data contains a header. Our data on used cars has no column headers. So, we need to specify read\_CSV to not assign headers by setting header to none.

After reading the dataset, it is a good idea to look at the data frame to get a better intuition and to ensure that everything occurred the way you expected. Since printing the entire dataset may take up too much time and resources to save time, we can just use dataframe.head to show the first n rows of the data frame.

Similarly, dataframe.tail shows the bottom end rows of data frame. Here, we printed out the first five rows of data. It seems that the dataset was read successfully. We can see that pandas automatically set the column header as a list of integers because we set header equals none when we read the data. It is difficult to work with the data frame without having meaningful column names. However, we can assign column names in pandas. In our present case, it turned out that we have the column names in a separate file online. We first put the column names in a list called headers, then we set df.columns equals headers to replace the default integer headers by the list. If we use the head method introduced in the last slide to check the dataset, we see the correct headers inserted at the top of each column.

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

You can do this using the method to\_CSV. To do this, specify the file path which includes the file name that you want to write to.

For example, if you would like to save dataframe df as automobile.CSV to your own computer, you can use the syntax `df.to_CSV`. For this course, we will only read and save CSV files. However, pandas also supports importing and exporting of most data file types with different dataset formats. The code syntax for reading and saving other data formats is very similar to read or save CSV file. Each column shows a different method to read and save files into a different format.

## Importing and Exporting Data in Python

### 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/ml/machine-learning-databases/autos/imports-85.data>

### Getting Data

```
3,7,alfa-romero,gas,std,two,convertible,rwd,front,88.40,168.80,64.10,48.80,2548,dohc,four,130,mpfi,3.47,2.68,9.00,111,5000,21,27,13495
3,7,alfa-romero,gas,std,two,convertible,rwd,front,88.40,168.80,64.10,48.80,2548,dohc,four,130,mpfi,3.47,2.68,9.00,111,5000,21,27,16500
1,7,alfa-romero,gas,std,two,hatchback,rwd,front,94.50,171.20,65.50,52.40,2823,ohcv,six,152,mpfi,2.68,3.47,9.00,154,5000,19,26,16500
2,164,audi,gas,std,four,sedan,fwd,front,99.80,176.60,66.20,54.30,2337,ohc,four,109,mpfi,3.19,3.40,10.00,102,5500,24,30,13950
2,164,audi,gas,std,four,sedan,4wd,front,99.40,176.60,66.40,54.30,2824,ohc,five,136,mpfi,3.19,3.40,8.00,115,5500,18,22,17450
2,7,audi,gas,std,two,sedan,fwd,front,99.80,177.30,66.30,53.10,2587,ohc,five,136,mpfi,3.19,3.40,8.50,110,5500,19,25,15250
1,158,audi,gas,std,four,sedan,fwd,front,105.80,192.70,71.40,55.70,2844,ohc,five,136,mpfi,3.19,3.40,8.50,110,5500,19,25,17710
1,7,audi,gas,std,four,four,wagon,fwd,front,105.80,192.70,71.40,55.90,70,2954,ohc,five,136,mpfi,3.19,3.40,8.50,110,5500,19,25,18920
1,158,audi,gas,turbo,four,sedan,fwd,front,105.80,192.70,71.40,55.90,3086,ohc,five,131,mpfi,3.13,3.40,8.30,140,5500,17,20,23875
0,7,hmw,gas,std,two,turbo,two,hatchback,fwd,front,99.50,178.20,69.67,90.52,00,3053,ohc,five,131,mpfi,3.13,3.40,7.00,160,5500,16,22,?
2,192,hmw,gas,std,two,sedan,rwd,front,101.20,176.80,64.80,54.30,2395,ohc,four,108,mpfi,3.50,2.80,8.80,101,5800,23,29,16430
0,192,hmw,gas,std,four,sedan,rwd,front,101.20,176.80,64.80,54.30,2395,ohc,four,108,mpfi,3.50,2.80,8.80,101,5800,23,29,16925
0,188,hmw,gas,std,two,sedan,rwd,front,101.20,176.80,64.80,54.30,2710,ohc,six,164,mpfi,3.31,3.19,9.00,121,4250,21,28,20970
0,188,hmw,gas,std,four,sedan,rwd,front,101.20,176.80,64.80,54.30,2765,ohc,six,164,mpfi,3.31,3.19,9.00,121,4250,21,28,21105
1,7,hmw,gas,std,four,sedan,rwd,front,103.50,189.00,66.90,55.70,3055,ohc,six,164,mpfi,3.31,3.19,9.00,121,4250,20,25,24565
0,7,hmw,gas,std,four,sedan,rwd,front,103.50,189.00,66.90,55.70,3230,ohc,six,209,mpfi,3.62,3.39,8.00,182,5400,16,22,30780
0,7,hmw,gas,std,two,sedan,rwd,front,103.50,193.80,67.90,53.70,3380,ohc,six,209,mpfi,3.62,3.39,8.00,182,5400,16,22,41315
0,7,hmw,gas,std,four,sedan,rwd,front,110.00,197.00,70.90,56.30,3505,ohc,six,209,mpfi,3.62,3.39,8.00,182,5400,15,20,36880
2,121,chevrolet,gas,std,two,hatchback,fwd,front,88.40,141.10,60.30,53.20,1488,1,three,61,2bb1,2.91,3.03,9.50,48,5100,47,53,5151
1,98,chevrolet,gas,std,two,hatchback,fwd,front,94.50,159.90,61.60,52.00,1909,ohc,four,90,2bb1,3.03,3.11,9.40,70,5400,38,43,6295
0,81,chevrolet,gas,std,four,sean,fwd,front,94.50,159.80,63.60,52.00,1909,ohc,four,90,2bb1,3.03,3.11,9.40,70,5400,38,43,6295
1,118,dodge,gas,std,two,hatchback,fwd,front,93.70,157.30,63.80,50.80,1876,ohc,four,90,2bb1,2.97,3.23,9.41,68,5500,37,43,5572
1,118,dodge,gas,std,two,hatchback,fwd,front,93.70,157.30,63.80,50.80,1876,ohc,four,90,2bb1,2.97,3.23,9.40,68,5500,31,38,6377
1,118,dodge,gas,turbo,two,hatchback,fwd,front,93.70,157.30,63.80,50.80,2128,ohc,four,98,mpfi,3.03,3.39,7.40,102,5500,24,30,7957
1,148,dodge,gas,std,four,sedan,fwd,front,93.70,157.30,63.80,50.60,1967,ohc,four,90,2bb1,2.97,3.23,9.40,68,5500,31,38,6229
1,148,dodge,gas,std,four,sedan,fwd,front,93.70,157.30,63.80,50.60,1989,ohc,four,98,2bb1,2.97,3.23,9.40,68,5500,31,38,6692
1,148,dodge,gas,std,four,sedan,fwd,front,93.70,157.30,63.80,50.60,1989,ohc,four,98,2bb1,2.97,3.23,9.40,68,5500,31,38,7669
1,148,dodge,gas,turbo,two,sedan,fwd,front,93.70,157.30,63.80,50.60,2191,ohc,four,98,mpfi,3.03,3.39,7.60,102,5500,24,30,8558
```

data source : <https://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.data>

A large number of properties are associated with each datapoint. Because the properties are separated from each other by commas, we can guess the data format is CSV, which stands for comma separated values.

## Importing a CSV into Python

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



First, import pandas, then define a variable with a file path and then use the read\_CSV method to import the data.

However, read\_CSV assumes the data contains a header.

## Importing a CSV without a header

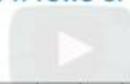
```
import pandas as pd  
  
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-  
85.data"  
  
df = pd.read_csv(url, header = None)
```



Our data on used cars has no column headers. So, we need to specify read\_CSV to not assign headers by setting header to none.

## Printing the dataframe in Python

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



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

## Printing the dataframe in Python

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

df.head()

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

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

## Adding headers

- Replace default header (by `df.columns = headers`)

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

It is difficult to work with the data frame without having meaningful column names. However, we can assign column names in pandas. In our present case, it turned out that we have the column names in a separate file online. We first put the column names in a list called `headers`, then we set `df.columns equals headers` to replace the default integer headers by the list. If we use the head method introduced in the last slide to check the dataset, we see the correct headers inserted at the top of each column.

## Adding headers

- Replace default header (by `df.columns = headers`)

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

```
df.columns=headers
```

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

## Adding headers

- Replace default header (by `df.columns = headers`)

symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke	compression-ratio	hp	
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	1
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	1
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	9.0	1
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	10.0	11
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	8.0	1

## Exporting a Pandas dataframe to CSV

- Preserve progress anytime by saving modified dataset using

```
path="C:/Windows/.../automobile.csv"  
df.to_csv(path)
```

## Exporting to different formats in Python



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

## Importing and Exporting Data in Python

Bookmarked

### Graded Question 1

1/1 point (ungraded)

Some common encodings are ...

csv

xlsx

Pandas



Submit

You have used 1 of 2 attempts

✓ Correct (1/1 point)

### Graded Question 2

1/1 point (ungraded)

What does the following method do to the dataframe? df : df.head(12)

Show the first 12 rows of dataframe.

Shows the bottom 12 rows of dataframe.



Submit

You have used 1 of 1 attempt

✓ Correct (1/1 point)

# Getting Started Analyzing Data in Python

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

At this point, we assume that the data has been loaded. It's time for us to explore the dataset. Pandas has several built-in methods that can be used to understand the datatype or features or to look at the distribution of data within the dataset.

Using these methods, gives an overview of the dataset and also point out potential issues such as the wrong data type of features which may need to be resolved later on. Data has a variety of types. The main types stored in Pandas' objects are `object`, `float`, `Int`, and `datetime`.

The data type names are somewhat different from those in native Python. This table shows the differences and similarities between them.

Some are very similar such as the numeric data types, int and float. The `object` pandas type function's similar to string in Python, save for the change in name.

While the `datetime` Pandas type, is a very useful type for handling time series data. There are two reasons to check data types in a dataset.

Pandas automatically assigns types based on the encoding it detects from the original data table. For a number of reasons, this assignment may be incorrect. For example, it should be awkward if the car price column which we should expect to contain continuous numeric numbers, is assigned the data type of object. It would be more natural for it to have the float type.

Jerry may need to manually change the data type to float. The second reason, is that allows an experienced data scientist to see which Python functions can be applied to a specific column. For example, some math functions can only be applied to numerical data. If these functions are applied to non-numerical data an error may result. When the `dtype` method is applied to the data set, the data type of each column is returned in a series.

A good data scientists intuition tells us that most of the data types make sense. They make of cars for example are names. So, this information should be of type `object`. The last one on the list could be an issue. As bore is a dimension of an engine, we should expect a numerical data type to be used.

Instead, the `object` type is used. In later sections, Jerry will have to correct these type mismatches. Now, we would like to check the statistical summary of each column to learn about the distribution of data in each column.

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

we use the `describe` method. It returns the number of terms in the column as `count`, average column value as `mean`, column standard deviation as `std`, the maximum minimum values, as well as the boundary of each of the quartiles.

By default, the `dataframe.describe` functions skips rows and columns that do not contain numbers. It is possible to make the `describe` method work for `object` type columns as well. To enable a summary of all the columns, we could add an argument.

Include `equals all` inside the `describe` function bracket. Now, the outcome shows the summary of all the 26 columns,

including object typed attributes. We see that for the object type columns, a different set of statistics is evaluated, like unique, top, and frequency. Unique is the number of distinct objects in the column. Top is most frequently occurring object, and freq is the number of times the top object appears in the column.

Some values in the table are shown here as NaN which stands for not a number. This is because that particular statistical metric cannot be calculated for that specific column data type.

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

This function shows the top 30 rows and bottom 30 rows of the data frame.

## Getting Started Analyzing Data in Python

### Basic insights from the data

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

## 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 <a href="#">datetime</a> module in Python's standard library)	time data.

name. While the datetime Pandas type, is a very useful type for handling time.

The data type names are somewhat different from those in native Python. This table shows the differences and similarities between them. Some are very similar such as the numeric data types, int and float. The object pandas type function's similar to string in Python, save for the change in

# 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 <a href="#">datetime</a> module in Python's standard library)	time data.

Why check data types?

- potential info and type mismatch
- compatibility with python methods

# Basic Insights of Dataset - Data Types

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

`df.dtypes`



	symboling	int64
	normalized-losses	object
name	object	
fuel-type	object	
aspiration	object	
num-of-doors	object	
body-style	object	
drive-wheels	object	
engine-location	object	
wheel-base	float64	
length	float64	
width	float64	
height	float64	
curb-weight	int64	
engine-type	object	
num-of-cylinders	object	
engine-size	int64	
fuel-system	object	
bore	object	
stroke	object	
compression-ratio	float64	
horsepower	object	
peak-rpm	object	
city-mpg	int64	
highway-mpg	int64	
price	object	
dtype	object	

A good data scientist's intuition tells us that most of the data types make sense. They make of cars for example are names. So, this information should be of type object. The last one on the list could be an issue. As bore is a dimension of an engine, we should expect a numerical data type to be used. Instead, the object type is used.

## 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	41.842693	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

## Methods to check data set

A.

### `dataframe.describe(include="all")`

- Provides full summary statistics

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

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	engine-size	fuel-system	bore	stroke
count	205.000000	205	205	205	205	205	205	205	205	205.000000	205	205	205	205
unique	NaN	52	22	2	2	3	5	3	2	NaN	NaN	8	39	37
top	NaN	?	toyota	gas	alt.	four	sedan	fwd	front	NaN	NaN	mpfi	3.62	3.40
freq	NaN	41	39	166	166	114	98	120	202	NaN	NaN	94	23	80
mean	0.834148	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	68.796985	126.972317	NaN	NaN	NaN
std	1.245307	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	1.021776	41.642893	NaN	NaN	NaN
min	-2.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	66.800000	61.000000	NaN	NaN	NaN
25%	0.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	44.500000	97.000000	NaN	NaN	NaN
50%	1.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	97.000000	120.000000	NaN	NaN	NaN
75%	2.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	102.400000	141.000000	NaN	NaN	NaN
max	3.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	109.000000	326.000000	NaN	NaN	NaN

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

`Unique` is the number of distinct objects in the column. `Top` is most frequently occurring object, and `freq` is the number of times the top object appears in the column. Some values in the table are shown here as `NaN` which stands for not a number. This is because that particular statistical metric cannot be calculated for that specific column data type.

B.

## Basic Insights of Dataset - Info

`dataframe.info()` provides a concise summary of your DataFrame.

```
df.info()
```

Another method you can use to check your dataset, is the `dataframe.info` function. This function shows the top 30 rows and bottom 30 rows of the data frame.

## Practice Quiz: Getting Started Analyzing Data in Python

 [Bookmark this page](#)

### Question 1

1/1 point (ungraded)

To enable a summary of all the columns, what must the parameter include be set to for the method describe?

df.describe(include="None")

df.describe(include="all")



#### Answer

Correct: Correct

[Submit](#)

You have used 1 of 2 attempts

[Reset](#)

[Show answer](#)

---

Correct (1/1 point)

## Accessing Databases with Python

Hello, in this video you will learn how to access databases using Python. Databases are powerful tools for data scientists.

After completing this module, you'll be able to explain the basic concepts related to using Python to connect to databases.

This is how a typical user accesses databases using Python code written on a Jupyter notebook, a web based editor.

There is a mechanism by which the Python program communicates with the DBMS. **The Python code connects to the database using API calls.** We will explain the basics of SQL APIs and Python DB APIs. An application programming interface is a set of functions that you can call to get access to some type of service.

The SQL API consists of library function calls as an application programming interface, API, for the DBMS. To pass SQL statements to the DBMS, an application program calls functions in the API, and it calls other functions to retrieve query results and status information from the DBMS. The basic operation of a typical SQL API is illustrated in the figure.

The application program begins its database access with one or more API calls that connect the program to the DBMS.

To send the SQL statement to the DBMS, the program builds the statement as a text string in a buffer and then makes an API call to pass the buffer contents to the DBMS.

The application program makes API calls to check the status of its DBMS request and to handle errors. The application program ends its database access with an API call that disconnects it from the database. DB-API is Python's standard API for accessing relational databases. It is a standard that allows you to write a single program that works with multiple

kinds of relational databases instead of writing a separate program for each one. So, if you learn the DB-API functions, then you can apply that knowledge to use any database with Python. The two main concepts in the Python DB-API are connection objects and query objects. You use connection objects to connect to a database and manage your transactions. Cursor objects are used to run queries. You open a cursor object and then run queries. The cursor works similar to a cursor in a text processing system where you scroll down in your result set and get your data into the application.

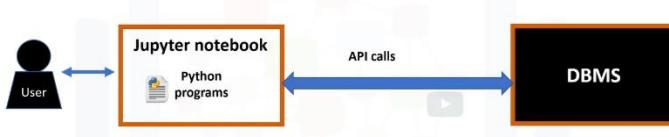
Cursors are used to scan through the results of a database. Here are the methods used with connection objects.

- The `cursor()` method returns a new cursor object using the connection.
- The `commit()` method is used to commit any pending transaction to the database.
- The `rollback()` method causes the database to roll back to the start of any pending transaction.
- The `close()` method is used to close a database connection.

Let's walk through a Python application that uses the DB-API to query a database. First, you import your database module by using the `connect` API from that module. To open a connection to the database, you use the `connection` function and pass in the parameters that is, the database name, username, and password. The `connect` function returns `connection` object. After this, you create a `cursor` object on the `connection` object.

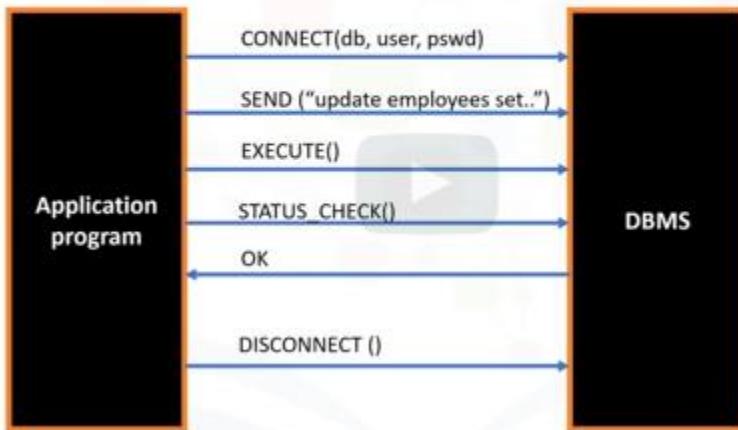
The `cursor` is used to run queries and fetch results. After running the queries using the `cursor`, we also use the `cursor` to fetch the results of the query. Finally, when the system is done running the queries, it frees all resources by closing the connection. Remember that it is always important to close connections to avoid unused connections taking up resources.

## Accessing databases using Python



The Python code connects to the database using API calls.

## What is a SQL API?



An application programming interface is a set of functions that you can call to get access to some type of service.

The SQL API consists of library function calls as an application programming interface, API, for the DBMS.

To pass SQL statements to the DBMS, an application program calls functions in the API, and it calls other functions to retrieve query results and status information from the DBMS.

The basic operation of a typical SQL API is illustrated in the figure. The application program begins its database access with one or more API calls that connect the program

to the DBMS. To send the SQL statement to the DBMS, the program builds the statement as a text string in a buffer and then makes an API call to pass the buffer contents to the DBMS. The application program makes API calls to check the status of its DBMS request and to handle errors. The application program ends its database access with an API call that disconnects it from the database.

## What is a DB-API?

DB-API is Python's standard API for accessing relational databases.

It is a standard that allows you to write a single program that works with multiple kinds of relational databases instead of writing a separate program for each one.

So, if you learn the DB-API functions, then you can apply that knowledge to use any database with Python.

## Concepts of the Python DB API

### Connection Objects

- Database connections
- Manage transactions

The two main concepts in the Python DB-API are connection objects and query objects.

You use connection objects to connect to a database and manage your transactions.

Cursor objects are used to run queries.

You open a cursor object and then run queries. The cursor works similar to a cursor in a text processing system where you scroll down in your result set and get your data into the application.

Cursors are used to scan through the results of a database.

### Cursor Objects

- Database Queries

# What are Connection methods?

- cursor()
- commit()
- rollback()
- close()



The `cursor()` method returns a new cursor object using the connection.

The `commit()` method is used to commit any pending transaction to the database.

The `rollback()` method causes the database to roll back to the start of any pending transaction.

The `close()` method is used to close a database connection.

## Writing code using DB-API

```
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()
```



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

## Examples of libraries used by database systems to connect to Python applications

Database	DB API
DB2 Warehouse on Cloud	Ibm_db
Compose for MySQL	MySQL Connector/Python
Compose for PostgreSQL	psycopg2
Compose for MongoDB	PyMongo

## Lesson Summary

In this lesson, you have learned how to:

- **Define the Business Problem:** Look at the data and make some high-level decision on what kind of analysis should be done
- **Import and Export Data in Python:** How to import data from multiple data sources using the Pandas library and how to export files into different formats.
- **Analyze Data in Python:** How to do some introductory analysis in Python using functions like `dataframe.head()` to view the first few lines of the dataset, `dataframe.info()` to view the column names and data types.

# 7.1.Importing Datasets

Seongjoo Brenden Song edited this page on Nov 5, 2021 · 4 revisions

---

## Learning Objectives

- Analyze Python data using a dataset
  - Identify three Python libraries and describe their uses
  - Read data using Python's Pandas package
  - Demonstrate how to import and export data in Python
- 

- [Importing Datasets](#)
- [Lab 1: Importing Datasets](#)

## 7.1.1.Importing Datasets

Seongjoo Brenden Song edited this page on Nov 5, 2021 · 3 revisions

### The Problem

#### Why Data Analysis?

- Data is everywhere
- Data analysis/data science helps us answer questions from data
- Data analysis plays an important role in:
  - Discovering useful information
  - Answering questions
  - Predicting future or the unknown

### Python Packages for Data Science

1. Scientific Computing Libraries
  - i. Pandas (Data structures & tools)
  - ii. NumPy (Arrays & matrices)
  - iii. SciPy (Integrals, solving differential equations, optimizations)
2. Visualization Libraries
  - i. Matplotlib (plots & graphs, most popular)
  - ii. Seaborn (plots: heat maps, time series, violin plots)
3. Algorithmic Libraries
  - i. Scikit-learn (Machine Learning: regression, classification, ... )
  - ii. Statsmodels (Explore data, estimate statistical models, and perform statistical tests)

### Question

What description best describes the library Pandas?

- ~~Uses arrays as their inputs and outputs. It can be extended to objects for matrices, and with a little change of coding, developers perform fast array processing.~~
- ~~Includes functions for some advanced math problems as listed in the slide as well as data visualization.~~
- Offers data structure and tools for effective data manipulation and analysis. It provides fast access to structured data. The primary instrument of Pandas is a two-dimensional table consisting of columns and rows labels which are called a DataFrame. It is designed to provide an easy indexing function.

Correct.

## Importing and Exporting Data in Python

### Importing a CSV into Python

```
import pandas as pd  
url = "https://www....."  
df = pd.read_csv(url)
```

### Importing a CSV without a header

```
import pandas as pd  
url = "https://www....."  
df = pd.read_csv(url, header = None)
```

### Printing the dataframe in Python

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

### Adding headers

- Replace default header (by `df.columns = headers`)

```
headers = ["header_1", "header_2", ... "header_n"]  
df.columns = headers
```

## Exporting a Pandas dataframe to CSV

- Preserve progress anytime by saving modified dataset using

```
path = "c:/Windows/.../data_file.csv"  
df.to_csv(path)
```

## Exporting to different formats in Python

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

# Getting Started Analyzing Data in Python

---

## Basic insights from the data

- Understand your data before you begin any analysis
- Should check:
  - Data Types
    - why check data types?
      - a. potential info and type mismatch
      - b. compatibility with python methods
  - Data Distribution
- Locate potential issues with the data

## Basic Insights of Dataset - Data Types

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

`dataframe.describe()`

- Returns a statistical summary
  - count, mean, std, min, 25%, 50%, 75%, max ...

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

- Provides full summary statistics
  - count, unique, top, freq, mean, std, min, 25%, 50%, 75%, max ...

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

- Provides full summary statistics
  - count, unique, top, freq, mean, std, min, 25%, 50%, 75%, max ...

## Basic Insights of Dataset - Info

- `dataframe.info()` provides a concise summary(top 30 rows & bottom 30 rows) of data frame

## Accessing Databases with Python

---

What is a SQL API?

What is a DB-API?

## Concepts of the Python DB API

- Connection Objects
  - Database connections
  - Manage transaction
- Cursor Objects
  - Database Queries

### What are Connection methods?

- `cursor()` : returns a new cursor object using the connection
- `commit()` : is used to commit any pending transaction to the database
- `rollback()` : causes the database to roll back to the start of any pending transaction
- `close()` : is used to close a database connection.

## Writing code using DB-API

```
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()
```



# Introduction Notebook

Estimated time needed: **10** minutes

## Objectives

After completing this lab you will be able to:

- Acquire data in various ways
- Obtain insights from data with Pandas library

## Table of Contents

1. Data Acquisition
2. Basic Insight of Dataset

## Data Acquisition

There are various formats for a dataset: .csv, .json, .xlsx etc. The dataset can be stored in different places, on your local machine or sometimes online.

In this section, you will learn how to load a dataset into our Jupyter Notebook.

In our case, the Automobile Dataset is an online source, and it is in a CSV (comma separated value) format. Let's use this dataset as an example to practice data reading.

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

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

```
[3]: #install specific version of libraries used in lab
#! mamba install pandas==1.3.3 -y
#! mamba install numpy=1.21.2 -y
```

```
[4]: # import pandas Library
import pandas as pd
import numpy as np
```

## Read Data

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

This dataset was hosted on IBM Cloud object. Click [HERE](#) for free storage.

```
[5]: # Import pandas Library
import pandas as pd

# Read the online file by the URL provides above, and assign it to variable "df"
other_path = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/auto.csv"
df = pd.read_csv(other_path, header=None)
```

After reading the dataset, we can use the dataframe.head(n) method to check the top n rows of the dataframe, where n is an integer. Contrary to dataframe.head(n), dataframe.tail(n) will show you the bottom n rows of the dataframe.

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

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

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

5 rows × 26 columns

### Question #1:

Check the bottom 10 rows of data frame "df".

```
[7]: # Write your code below and press Shift+Enter to execute...
print("The last 10 rows of the dataframe\n")
df.tail(10)
```

The last 10 rows of the dataframe

[7]:	0	1	2	3	4	5	6	7	8	9	...	16	17	18	19	20	21	22	23	24	25
195	-1	74	volvo	gas	std	four	wagon	rwd	front	104.3	...	141	mpfi	3.78	3.15	9.5	114	5400	23	28	13415
196	-2	103	volvo	gas	std	four	sedan	rwd	front	104.3	...	141	mpfi	3.78	3.15	9.5	114	5400	24	28	15985
197	-1	74	volvo	gas	std	four	wagon	rwd	front	104.3	...	141	mpfi	3.78	3.15	9.5	114	5400	24	28	16515
198	-2	103	volvo	gas	turbo	four	sedan	rwd	front	104.3	...	130	mpfi	3.62	3.15	7.5	162	5100	17	22	18420
199	-1	74	volvo	gas	turbo	four	wagon	rwd	front	104.3	...	130	mpfi	3.62	3.15	7.5	162	5100	17	22	18950
200	-1	95	volvo	gas	std	four	sedan	rwd	front	109.1	...	141	mpfi	3.78	3.15	9.5	114	5400	23	28	16845
201	-1	95	volvo	gas	turbo	four	sedan	rwd	front	109.1	...	141	mpfi	3.78	3.15	8.7	160	5300	19	25	19045
202	-1	95	volvo	gas	std	four	sedan	rwd	front	109.1	...	173	mpfi	3.58	2.87	8.8	134	5500	18	23	21485
203	-1	95	volvo	diesel	turbo	four	sedan	rwd	front	109.1	...	145	idi	3.01	3.40	23.0	106	4800	26	27	22470
204	-1	95	volvo	gas	turbo	four	sedan	rwd	front	109.1	...	141	mpfi	3.78	3.15	9.5	114	5400	19	25	22625

10 rows × 26 columns

## Add Headers

Take a look at our dataset. Pandas automatically set the header with an integer starting from 0.

To better describe our data, we can introduce a header. This information is available

at: <https://archive.ics.uci.edu/ml/datasets/Automobile>.

Thus, we have to add headers manually.

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

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

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

We replace headers and recheck our dataframe:

```
[9]: df.columns = headers
df.head(10)
```

[9]:	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke	compression-ratio	horsepower
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	9.0	154
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	10.0	102
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	8.0	115
5	2	?	audi	gas	std	two	sedan	fwd	front	99.8	...	136	mpfi	3.19	3.40	8.5	110
6	1	158	audi	gas	std	four	sedan	fwd	front	105.8	...	136	mpfi	3.19	3.40	8.5	110
7	1	?	audi	gas	std	four	wagon	fwd	front	105.8	...	136	mpfi	3.19	3.40	8.5	110
8	1	158	audi	gas	turbo	four	sedan	fwd	front	105.8	...	131	mpfi	3.13	3.40	8.3	140
9	0	?	audi	gas	turbo	two	hatchback	4wd	front	99.5	...	131	mpfi	3.13	3.40	7.0	160

10 rows × 26 columns

peak-rpm	city-mpg	highway-mpg	price
5000	21	27	13495
5000	21	27	16500
5000	19	26	16500
5500	24	30	13950
5500	18	22	17450
5500	19	25	15250
5500	19	25	17710
5500	19	25	18920
5500	17	20	23875
5500	16	22	?

We need to replace the "?" symbol with NaN so the dropna() can remove the missing values:

```
[10]: df1=df.replace('?',np.Nan)
```

We can drop missing values along the column "price" as follows:

```
[12]: df=df1.dropna(subset=["price"], axis=0)
df.head(20)
```

[12]:	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke	
0	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	
1	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	
2	1	NaN	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	
5	2	NaN	audi	gas	std	two	sedan	fwd	front	99.8	...	136	mpfi	3.19	3.40	
6	1	158	audi	gas	std	four	sedan	fwd	front	105.8	...	136	mpfi	3.19	3.40	
7	1	NaN	audi	gas	std	four	wagon	fwd	front	105.8	...	136	mpfi	3.19	3.40	
8	1	158	audi	gas	turbo	four	sedan	fwd	front	105.8	...	131	mpfi	3.13	3.40	
10	2	192	bmw	gas	std	two	sedan	rwd	front	101.2	...	108	mpfi	3.50	2.80	
11	0	192	bmw	gas	std	four	sedan	rwd	front	101.2	...	108	mpfi	3.50	2.80	
12	0	188	bmw	gas	std	two	sedan	rwd	front	101.2	...	164	mpfi	3.31	3.19	
13	0	188	bmw	gas	std	four	sedan	rwd	front	101.2	...	164	mpfi	3.31	3.19	
14	1	NaN	bmw	gas	std	four	sedan	rwd	front	103.5	...	164	mpfi	3.31	3.19	
15	0	NaN	bmw	gas	std	four	sedan	rwd	front	103.5	...	209	mpfi	3.62	3.39	
16	0	NaN	bmw	gas	std	two	sedan	rwd	front	103.5	...	209	mpfi	3.62	3.39	
17	0	NaN	bmw	gas	std	four	sedan	rwd	front	110.0	...	209	mpfi	3.62	3.39	
18	2	121	chevrolet	gas	std	two	hatchback	fwd	front	88.4	...	61	2bbl	2.91	3.03	
19	1	98	chevrolet	gas	std	two	hatchback	fwd	front	94.5	...	90	2bbl	3.03	3.11	
20	0	81	chevrolet	gas	std	four	sedan	fwd	front	94.5	...	90	2bbl	3.03	3.11	
compression-ratio	horsepower	peak-rpm	city-mpg	highway-mpg	price											
9.0	111	5000	21	27	13495											
9.0	111	5000	21	27	16500											
9.0	154	5000	19	26	16500											
10.0	102	5500	24	30	13950											
8.0	115	5500	18	22	17450											
8.5	110	5500	19	25	15250											
8.5	110	5500	19	25	17710											
8.5	110	5500	19	25	18920											
8.3	140	5500	17	20	23875											
8.8	101	5800	23	29	16430											
8.8	101	5800	23	29	16925											
9.0	121	4250	21	28	20970											
9.0	121	4250	21	28	21105											
9.0	121	4250	20	25	24565											
8.0	182	5400	16	22	30760											
8.0	182	5400	16	22	41315											
8.0	182	5400	15	20	36880											
9.5	48	5100	47	53	5151											
9.6	70	5400	38	43	6295											
9.6	70	5400	38	43	6575											

Now, we have successfully read the raw dataset and added the correct headers into the dataframe.

## Question #2:

Find the name of the columns of the dataframe.

```
[13]: # Write your code below and press Shift+Enter to execute..  
print(df.columns)  
  
Index(['symboling', 'normalized-losses', 'make', 'fuel-type', 'aspiration',  
       'num-of-doors', 'body-style', 'drive-wheels', 'engine-location',  
       'wheel-base', 'length', 'width', 'height', 'curb-weight', 'engine-type',  
       'num-of-cylinders', 'engine-size', 'fuel-system', 'bore', 'stroke',  
       'compression-ratio', 'horsepower', 'peak-rpm', 'city-mpg',  
       'highway-mpg', 'price'],  
      dtype='object')
```

► [Click here for the solution](#)

## Save Dataset

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

For example, if you would save the dataframe `df` as **automobile.csv** to your local machine, you may use the syntax below, where `index = False` means the row names will not be written.

## Read/Save Other Data Formats

Data Format	Read	Save
csv	<code>pd.read_csv()</code>	<code>df.to_csv()</code>
json	<code>pd.read_json()</code>	<code>df.to_json()</code>
excel	<code>pd.read_excel()</code>	<code>df.to_excel()</code>
hdf	<code>pd.read_hdf()</code>	<code>df.to_hdf()</code>
sql	<code>pd.read_sql()</code>	<code>df.to_sql()</code>
...	...	...

## Basic Insight of Dataset

After reading data into Pandas dataframe, it is time for us to explore the dataset.

There are several ways to obtain essential insights of the data to help us better understand our dataset.

## Data Types

Data has a variety of types.

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

```
[14]: df.dtypes
```

```
[14]: symboling      int64
normalized-losses   object
make                object
fuel-type          object
aspiration         object
num-of-doors       object
body-style          object
drive-wheels        object
engine-location     object
wheel-base          float64
length              float64
width               float64
height              float64
curb-weight         int64
engine-type         object
num-of-cylinders   object
engine-size         int64
fuel-system         object
bore                object
stroke              object
compression-ratio   float64
horsepower          object
peak-rpm             object
city-mpg            int64
highway-mpg         int64
price               object
dtype: object
```

A series with the data type of each column is returned.

```
[16]: #.check_the_data_type_of_data_frame "df" by .dtypes
print(df.dtypes)

symboling          int64
normalized-losses   object
make              object
fuel-type          object
aspiration         object
num-of-doors       object
body-style          object
drive-wheels        object
engine-location     object
wheel-base          float64
length              float64
width               float64
height              float64
curb-weight         int64
engine-type         object
num-of-cylinders    object
engine-size          int64
fuel-system          object
bore                  object
stroke                object
compression-ratio    float64
horsepower           object
peak-rpm              object
city-mpg              int64
highway-mpg           int64
price                 object
dtype: object
```

As shown above, it is clear to see that the data type of "symboling" and "curb-weight" are int64, "normalized-losses" is object, and "wheel-base" is float64, etc.

These data types can be changed; we will learn how to accomplish this in a later module.

## Describe

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

```
dataframe.describe()
```

This method will provide various summary statistics, excluding `NaN` (Not a Number) values.

	<code>symboling</code>	<code>wheel-base</code>	<code>length</code>	<code>width</code>	<code>height</code>	<code>curb-weight</code>	<code>engine-size</code>	<code>compression-ratio</code>	<code>city-mpg</code>	<code>highway-mpg</code>
<code>count</code>	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000
<code>mean</code>	0.840796	98.797015	174.200995	65.889055	53.766667	2555.666667	126.875622	10.164279	25.179104	30.686567
<code>std</code>	1.254802	6.066366	12.322175	2.101471	2.447822	517.296727	41.546834	4.004965	6.423220	6.815150
<code>min</code>	-2.000000	86.600000	141.100000	60.300000	47.800000	1488.000000	61.000000	7.000000	13.000000	16.000000
<code>25%</code>	0.000000	94.500000	166.800000	64.100000	52.000000	2169.000000	98.000000	8.600000	19.000000	25.000000
<code>50%</code>	1.000000	97.000000	173.200000	65.500000	54.100000	2414.000000	120.000000	9.000000	24.000000	30.000000
<code>75%</code>	2.000000	102.400000	183.500000	66.600000	55.500000	2926.000000	141.000000	9.400000	30.000000	34.000000
<code>max</code>	3.000000	120.900000	208.100000	72.000000	59.800000	4066.000000	326.000000	23.000000	49.000000	54.000000

This shows the statistical summary of all numeric-typed (int, float) columns.

For example, the attribute "symboling" has 205 counts, the mean value of this column is 0.83, the standard deviation is 1.25, the minimum value is -2, 25th percentile is 0, 50th percentile is 1, 75th percentile is 2, and the maximum value is 3.

However, what if we would also like to check all the columns including those that are of type object?

You can add an argument include = "all" inside the bracket. Let's try it again.

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

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke
count	201.000000	164	201	201	201	199	201	201	201	201.000000	...	201.000000	201	197	197
unique	NaN	51	22	2	2	2	5	3	2	NaN	...	NaN	8	38	36
top	NaN	161	toyota	gas	std	four	sedan	fwd	front	NaN	...	NaN	mpfi	3.62	3.40
freq	NaN	11	32	181	165	113	94	118	198	NaN	...	NaN	92	23	19
mean	0.840796	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	98.797015	...	126.875622	NaN	NaN	NaN
std	1.254802	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	6.066366	...	41.546834	NaN	NaN	NaN
min	-2.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	86.600000	...	61.000000	NaN	NaN	NaN
25%	0.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	94.500000	...	98.000000	NaN	NaN	NaN
50%	1.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	97.000000	...	120.000000	NaN	NaN	NaN
75%	2.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	102.400000	...	141.000000	NaN	NaN	NaN
max	3.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	120.900000	...	326.000000	NaN	NaN	NaN

11 rows × 26 columns

compression-ratio	horsepower	peak-rpm	city-mpg	highway-mpg	price
201.000000	199	199	201.000000	201.000000	201
NaN	58	22	NaN	NaN	186
NaN	68	5500	NaN	NaN	8921
NaN	19	36	NaN	NaN	2
10.164279	NaN	NaN	25.179104	30.686567	NaN
4.004965	NaN	NaN	6.423220	6.815150	NaN
7.000000	NaN	NaN	13.000000	16.000000	NaN
8.600000	NaN	NaN	19.000000	25.000000	NaN
9.000000	NaN	NaN	24.000000	30.000000	NaN
9.400000	NaN	NaN	30.000000	34.000000	NaN
23.000000	NaN	NaN	49.000000	54.000000	NaN

Now it provides the statistical summary of all the columns, including object-typed attributes.

We can now see how many unique values there, which one is the top value and the frequency of top value in the object-typed columns.

Some values in the table above show as "NaN". This is because those numbers are not available regarding a particular column type.

## Question #3:

You can select the columns of a dataframe by indicating the name of each column. For example, you can select the three columns as follows:

```
dataframe[['column 1','column 2','column 3']]
```

Where "column" is the name of the column, you can apply the method ".describe()" to get the statistics of those columns as follows:

```
dataframe[['column 1','column 2','column 3']].describe()
```

Apply the method to ".describe()" to the columns 'length' and 'compression-ratio'.

```
[19]: # Write your code below and press Shift+Enter to execute...
print(df.columns)

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

► [Click here for the solution](#)

## Info

Another method you can use to check your dataset is:

```
dataframe.info()
```

It provides a concise summary of your DataFrame.

This method prints information about a DataFrame including the index dtype and columns, non-null values and memory usage.

```
[20]: # Look at the info of "df"
df.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 201 entries, 0 to 204
Data columns (total 26 columns):
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 201 entries, 0 to 204
Data columns (total 26 columns):
 #   Column            Non-Null Count  Dtype  
--- 
 0   symboling         201 non-null    int64  
 1   normalized-losses 164 non-null    object  
 2   make              201 non-null    object  
 3   fuel-type          201 non-null    object  
 4   aspiration         201 non-null    object  
 5   num-of-doors       199 non-null    object  
 6   body-style         201 non-null    object  
 7   drive-wheels       201 non-null    object  
 8   engine-location    201 non-null    object  
 9   wheel-base         201 non-null    float64 
 10  length             201 non-null    float64 
 11  width              201 non-null    float64 
 12  height             201 non-null    float64 
 13  curb-weight        201 non-null    int64  
 14  engine-type        201 non-null    object  
 15  num-of-cylinders   201 non-null    object  
 16  engine-size         201 non-null    int64  
 17  fuel-system         201 non-null    object  
 18  bore               197 non-null    object  
 19  stroke              197 non-null    object  
 20  compression-ratio   201 non-null    float64 
 21  horsepower          199 non-null    object  
 22  peak-rpm             199 non-null    object  
 23  city-mpg            201 non-null    int64  
 24  highway-mpg          201 non-null    int64  
 25  price               201 non-null    object  
dtypes: float64(5), int64(5), object(16)
memory usage: 42.4+ KB
```

## Graded Quiz: Importing Data Sets

Bookmarked

Graded Quiz due Feb 7, 2022 09:32 +08

### Question 1

1/1 point (graded)

What do we want to predict from the dataset?

make

price

colour



#### Answer

Correct: Correct

Submit

You have used 1 of 2 attempts

---

✓ Correct (1/1 point)

---

### Question 2

1/1 point (graded)

Select the libraries you will use for this course:

pandas

matplotlib

scikit-learn



#### Answer

Correct: Correct

Submit

You have used 1 of 2 attempts

---

✓ Correct (1/1 point)

---

### Question 3

1/1 point (graded)

What task does the following command perform?

```
df.to_csv("A.csv")
```

Save the dataframe df to a csv file called "A.csv"

change the name of the column to "A.csv"

load the data from a csv file called "A" into a dataframe



#### Answer

Correct: Correct

Submit

You have used 2 of 2 attempts

Correct (1/1 point)

### Question 4

1/1 point (graded)

Consider the segment of the following dataframe:

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

What is the type of the column **make**?

int64

object

float64



Submit

You have used 1 of 2 attempts

Correct (1/1 point)

---

## Question 5

1/1 point (graded)

How would you generate descriptive statistics for all the columns for the dataframe df?

df.info

df.describe(include = "all")

df.describe()



### Answer

Correct: Correct

Submit

You have used 2 of 2 attempts

---

✓ Correct (1/1 point)

## 7.1.2.Lab 1: Importing Datasets

### Data Acquisition

There are various formats for a dataset: .csv, .json, .xlsx etc. The dataset can be stored in different places, on your local machine or sometimes online.

In this section, you will learn how to load a dataset into our Jupyter Notebook.

In our case, the Automobile Dataset is an online source, and it is in a CSV (comma separated value) format. Let's use this dataset as an example to practice data reading.

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

The Pandas Library is a useful tool that enables us to read various datasets into a dataframe; our Jupyter notebook platforms have a built-in

### Pandas Library

so that all we need to do is import Pandas without installing.

```
*# import pandas library*
**import** pandas **as** pd
**import** numpy **as** np
```

### Read Data

We use pandas.read\_csv() function to read the csv file. In the brackets, we put the file path along with a quotation mark so that pandas will read the file into a dataframe from that address. The file path can be either an URL or your local file address.

Because the data does not include headers, we can add an argument headers = None inside the read\_csv() method so that pandas will not automatically set the first row as a header.

You can also assign the dataset to any variable you create.

This dataset was hosted on IBM Cloud object. Click [HERE](#) for free storage.

```
*# Import pandas library*
**import** pandas **as** pd

*# Read the online file by the URL provided above, and assign it to variable "df"*
other_path **=** "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DA010:
df **=** pd.read_csv(other_path, header**=**None**)
```

After reading the dataset, we can use the dataframe.head(n) method to check the top n rows of the dataframe, where n is an integer. Contrary to dataframe.head(n), dataframe.tail(n) will show you the bottom n rows of the dataframe.

```
*# show the first 5 rows using dataframe.head() method*
print("The first 5 rows of the dataframe")
df.head(5)
```

Question #1: Check the bottom 10 rows of data frame "df".

```
*# Write your code below and press Shift+Enter to execute*
print("The last 10 rows of the dataframe\n")
df.tail(10)
```

## Add Headers

Take a look at our dataset. Pandas automatically set the header with an integer starting from 0.

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

Thus, we have to add headers manually.

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

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

print("headers\n", headers)
```

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

We replace headers and recheck our dataframe:

```
df.columns **=** headers
df.head(10)
```

We replace headers and recheck our dataframe:

```
df.columns ***=*** headers  
df.head(10)
```

We need to replace the "?" symbol with NaN so the dropna() can remove the missing values:

```
df1***=***df.replace('?',np.NaN)
```

We can drop missing values along the column "price" as follows:

```
df***=***df1.dropna(subset***=***["price"], axis***=***0)  
df.head(20)
```

Now, we have successfully read the raw dataset and added the correct headers into the dataframe.

**Question #2: Find the name of the columns of the dataframe.**

```
*# Write your code below and press Shift+Enter to execute*  
df.columns
```

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

## Save Dataset

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

For example, if you would save the dataframe **df** as **automobile.csv** to your local machine, you may use the syntax below, where index = False means the row names will not be written.

```
df.to_csv("automobile.csv", index=False)
```

## Basic Insight of Dataset

After reading data into Pandas dataframe, it is time for us to explore the dataset.

There are several ways to obtain essential insights of the data to help us better understand our dataset.

## Data Types

Data has a variety of types.

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

```
df.dtypes
```

```
symboling          int64
normalized-losses   object
make                object
fuel-type           object
aspiration          object
num-of-doors        object
body-style          object
drive-wheels        object
engine-location     object
wheel-base          float64
length              float64
width               float64
height              float64
curb-weight         int64
engine-type         object
num-of-cylinders    object
engine-size          int64
fuel-system          object
bore                 object
stroke               object
compression-ratio    float64
horsepower          object
peak-rpm             object
city-mpg             int64
highway-mpg          int64
price                object
dtype: object
```

A series with the data type of each column is returned.

```
*# check the data type of data frame "df" by .dtypes*
print(df.dtypes)
```

```
symboling          int64
normalized-losses   object
make                object
fuel-type           object
aspiration          object
num-of-doors        object
body-style          object
drive-wheels        object
engine-location     object
wheel-base          float64
length              float64
width               float64
height              float64
curb-weight         int64
engine-type         object
num-of-cylinders    object
engine-size         int64
fuel-system         object
bore                object
stroke              object
compression-ratio   float64
horsepower          object
peak-rpm             object
city-mpg            int64
highway-mpg          int64
price               object
dtype: object
```

As shown above, it is clear to see that the data type of "symboling" and "curb-weight" are int64, "normalized-losses" is object, and "wheel-base" is float64, etc.

These data types can be changed; we will learn how to accomplish this in a later module.

## Describe

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

```
dataframe.describe()
```

This method will provide various summary statistics, excluding NaN (Not a Number) values.

```
df.describe()
```

This shows the statistical summary of all numeric-typed (int, float) columns.

For example, the attribute "symboling" has 205 counts, the mean value of this column is 0.83, the standard deviation is 1.25, the minimum value is -2, 25th percentile is 0, 50th percentile is 1, 75th percentile is 2, and the maximum value is 3.

However, what if we would also like to check all the columns including those that are of type object?

You can add an argument include = "all" inside the bracket. Let's try it again.

```
*# describe all the columns in "df"*
df.describe(include **=** "all")
```

Now it provides the statistical summary of all the columns, including object-typed attributes.

We can now see how many unique values there, which one is the top value and the frequency of top value in the object-typed columns.

Some values in the table above show as "NaN". This is because those numbers are not available regarding a particular column type.

## Question #3:

---

You can select the columns of a dataframe by indicating the name of each column. For example, you can select the three columns as follows:

```
dataframe[['column 1', 'column 2', 'column 3']]
```

Where "column" is the name of the column, you can apply the method ".describe()" to get the statistics of those columns as follows:

```
dataframe[['column 1', 'column 2', 'column 3']].describe()
```

Apply the method to ".describe()" to the columns 'length' and 'compression-ratio'.

```
*# Write your code below and press Shift+Enter to execute*
df[['length', 'compression-ratio']].describe()
```

## Info

Another method you can use to check your dataset is:

```
dataframe.info()
```

It provides a concise summary of your DataFrame.

This method prints information about a DataFrame including the index dtype and columns, non-null values and memory usage.

```
*# look at the info of "df"*
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 201 entries, 0 to 204
Data columns (total 26 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   symboling        201 non-null    int64  
 1   normalized-losses 164 non-null    object  
 2   make             201 non-null    object  
 3   fuel-type        201 non-null    object  
 4   aspiration       201 non-null    object  
 5   num-of-doors     199 non-null    object  
 6   body-style       201 non-null    object  
 7   drive-wheels     201 non-null    object  
 8   engine-location   201 non-null    object  
 9   wheel-base       201 non-null    float64 
 10  length           201 non-null    float64 
 11  width            201 non-null    float64 
 12  height           201 non-null    float64 
 13  curb-weight      201 non-null    int64  
 14  engine-type      201 non-null    object  
 15  num-of-cylinders 201 non-null    object  
 16  engine-size      201 non-null    int64  
 17  fuel-system      201 non-null    object  
 18  bore              197 non-null    object  
 19  stroke            197 non-null    object  
 20  compression-ratio 201 non-null    float64 
 21  horsepower        199 non-null    object  
 22  peak-rpm          199 non-null    object  
 23  city-mpg          201 non-null    int64  
 24  highway-mpg       201 non-null    int64  
 25  price             201 non-null    object  
dtypes: float64(5), int64(5), object(16)
memory usage: 52.4+ KB
```

## Module Introduction & Learning Objectives

Bookmarked

### Module Introduction

In this week's module, you will learn how to perform some fundamental data wrangling tasks that, together, form the pre-processing phase of data analysis. These tasks include handling missing values in data, formatting data to standardize it and make it consistent, normalizing data, grouping data values into bins, and converting categorical variables into numerical quantitative variables.

### Learning Objectives

- Describe how to handle missing values
- Describe data formatting techniques
- Describe data normalization
- Demonstrate the use of binning
- Demonstrate the use of categorical variables

## Pre-Processing Data With Python

In this video, we'll be going through some data preprocessing techniques. If you're unfamiliar with the term, data preprocessing is a necessary step in data analysis. It is the process of converting or mapping data from one raw form into another format to make it ready for further analysis. Data preprocessing is often called data cleaning or data wrangling, and there are likely other terms. Here are the topics that we'll be covering in this module.

First, we'll show you how to identify and handle missing values. A missing value condition occurs whenever a data entry is left empty. Then we'll cover data formats. Data from different sources maybe in various formats, in different units, or in various conventions.

We will introduce some methods in Python Pandas that can standardize the values into the same format, or unit, or convention. After that, we'll cover data normalization. Different columns of numerical data may have very different ranges and direct comparison is often not meaningful. Normalization is a way to bring all data into a similar range for more useful comparison. Specifically, we'll focus on the techniques of centering and scaling. Then, we'll introduce data binning.

Binning creates bigger categories from a set of numerical values. It is particularly useful for comparison between groups of data. Lastly, we'll talk about categorical variables and show you how to convert categorical values into numeric variables to make statistical modeling easier. In Python, we usually perform operations along columns. Each row of the column represents a sample, I.e, a different used car in the database. You access a column by specifying the name of the column.

For example, you can access symboling and body style. Each of these columns is a Panda series. There are many ways to manipulate Dataframes in Python. For example, you can add a value to each entry off a column. To add one to each symboling entry, use this command. This changes each value of the Data frame column by adding one to the current value.

### Pre-processing Data in Python

#### Data Pre-processing

Also known as:

Data Cleaning, Data Wrangling

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

# Learning Objectives

- Identify and handle missing values
- Data Formatting
- Data Normalization (centering /scaling)
- Data Binning
- Turning Categorical values to numeric variables

• How to identify and handle missing values. A missing value condition occurs whenever a data entry is left empty.

• Data from different sources maybe in various formats, in different units, or in various conventions.

• Data normalization. Different columns of numerical data may have very different ranges and direct comparison is often not meaningful.

Normalization is a way to bring all data into a similar

range for more useful comparison. Specifically, we'll focus on the techniques of centering and scaling.

- Binning creates bigger categories from a set of numerical values. It is particularly useful for comparison between groups of data.
- Categorical variables. how to convert categorical values into numeric variables to make statistical modeling easier

## Simple Dataframe Operations

	df["symboling"]	df["body-style"]
0	3	alfa-romero
1	3	alfa-romero
2	1	alfa-romero
3	2	audi
4	2	audi
5	2	audi
6	1	audi
7	1	audi
8	1	audi
9	0	audi

Each row of the column represents a sample, i.e., a different used car in the database. You access a column by specifying the name of the column.

For example, you can access symboling and body style. Each of these columns is a Panda series.

To Add a Value:

## Simple Dataframe Operations

```
df["symboling"] = df["symboling"] + 1
```

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

There are many ways to manipulate Dataframes in Python. For example, you can add a value to each entry off a column. To add one to each symboling entry, use this command. This changes each value of the Data frame column by adding one to the current value.

## Dealing with Missing Values in Python

We will introduce the pervasive problem of missing values as well as strategies on what to do when you encounter missing values in your data. When no data value is stored for feature for a particular observation, we say this feature has a missing value. Usually missing value in data set appears as question mark and a zero or just a blank cell.

In the example here, the normalized losses feature has a missing value which is represented with NaN. But how can you deal with missing data? There are many ways to deal with missing values and this is regardless of Python, R or whatever tool you use.

Of course, each situation is different and should be judged differently. However, these are the typical options you can consider. The first is to check if the person or group that collected the data can go back and find what the actual value should be.

Another possibility is just to remove the data where that missing value is found. When you drop data, you could either drop the whole variable or just the single data entry with the missing value. If you don't have a lot of observations with missing data, usually dropping the particular entry is the best. If you're removing data, you want to look to do something that has the least amount of impact. Replacing data is better since no data is wasted. However, it is less accurate since we need to replace missing data with a guess of what the data should be. One standard for placement technique is to replace missing values by the average value of the entire variable. As an example, suppose we have some entries that have missing values for the normalized losses column and the column average for entries with data is 4500.

While there is no way for us to get an accurate guess of what the missing value is under the normalized losses column should have been, you can approximate their values using the average value of the column 4500. But what if the values cannot be averaged as with categorical variables? For a variable like fuel type, there isn't an average fuel type since the variable values are not numbers. In this case, one possibility is to try using the mode, the most common like gasoline.

Finally, sometimes we may find another way to guess the missing data. This is usually because the data gathered knows something additional about the missing data. For example, he may know that the missing values tend to be old cars and the normalized losses of old cars are significantly higher than the average vehicle.

And of course, finally, in some cases you may simply want to leave the missing data as missing data. For one reason or another, it may be useful to keep that observation even if some features are missing. Now, let's go into how to drop missing values or replace missing values in Python.

To remove data that contains missing values Panda's library has a built-in method called dropna. Essentially, with the dropna method, you can choose to drop rows or columns that contain missing values like NaN. So you'll need to specify axis equal zero to drop the rows or axis equals one to drop the columns that contain the missing values.

In this example, there is a missing value in the price column. Since the price of used cars is what we're trying to predict in our upcoming analysis, we have to remove the cars, the rows, that don't have a listed price.

It can simply be done in one line of code using `dataframe.dropna`. Setting the argument `inplace` to true, allows the modification to be done on the data set directly. In place equals true, just writes the result back into the data frame. This is equivalent to this line of code. Don't forget that this line of code does not change the data frame but is a good way to make sure that you are performing the correct operation.

To modify the data frame, you have to set the parameter `inplace` equal to true. You should always check the documentation if you are not familiar with the function or method. The pandas web page has lots of useful resources. To replace missing values like NaNs with actual values, Pandas library has a built-in method called `replace` which can be used to fill in the missing values with the newly calculated values.

As an example, assume that we want to replace the missing values of the variable normalized losses by the mean value of the variable. Therefore, the missing value should be replaced by the average of the entries within that column.

In Python, first we calculate the mean of the column. Then we use the method replace to specify the value we would like to be replaced as the first parameter, in this case NaN. The second parameter is the value we would like to replace it with i.e the mean in this example. This is a fairly simplified way of replacing missing values. There are of course other techniques such as replacing missing values for the average of the group instead of the entire data set.

So, we've gone through two ways in Python to deal with missing data. We learnt to drop problematic rows or columns containing missing values and then we learnt how to replace missing values with other values. But don't forget the other ways to deal with missing data. You can always check for a higher quality data set or source or in some cases you may want to leave the missing data as missing data.

## Missing Values

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

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

## How to deal with missing data?

Check with the data collection source

Drop the missing values

- drop the variable
- drop the data entry



Replace the missing values

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

Leave it as missing data

## How to deal with missing data:

### Check with the data collection source

- The first is to check if the person or group that collected the data can go back and find what the actual value should be.
- Another possibility is just to remove the data where that missing value is found.

### Drop the missing values

- Drop the variable
- Drop the data entry

When you drop data, you could either drop the whole variable or just the single data entry with the missing value. If you don't have a lot of observations with missing data, usually dropping the particular entry is the best.

### Replace the missing values

- Replace it with an average.

As an example, suppose we have some entries that have missing values for the normalized losses column and the column average for entries with data is 4500. While there is no way for us to get an accurate guess of what the missing value is under the normalized losses column should have been, you can approximate their values using the average value of the column 4500. But what if the values cannot be averaged as with categorical variables?

- Replace it by frequency (mode). For a variable like fuel type, there isn't an average fuel type since the variable values are not numbers. In this case, one possibility is to try using the mode, the most common like gasoline.
- Replace it based on other functions. Find another way to guess the missing data. This is usually because the data gathered knows something additional about the missing data. For example, he may know that the missing values tend to be old cars and the normalized losses of old cars are significantly higher than the average vehicle.

### Leave it as missing data

- In some cases you may simply want to leave the missing data as missing data.
- For one reason or another, it may be useful to keep that observation even if some features are missing.

## How to drop missing values in Python

- Use `dataframes.dropna()`:

The diagram shows a data frame with two columns: 'highway-mpg' and 'price'. The data consists of several rows. A specific row is highlighted with a red dashed border, containing the value 'NaN' in the 'price' column. An arrow points from this row to a second data frame, which is identical except that the entire row is missing. Below the arrows, text indicates: 'axis=0 drops the entire row' and 'axis=1 drops the entire column'. At the bottom, the code `df.dropna(subset=["price"], axis=0, inplace = True)` is shown.

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

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

### Steps:

1. To remove data that contains missing values Panda's library has a built-in method called `dropna`. Essentially, with the `dropna` method, you can choose to drop rows or columns that contain missing values like `NaN`.
2. So you'll need to specify `axis=0` to drop the rows or `axis=1` to drop the columns that contain the missing values.

In this example, there is a missing value in the price column. Since the price of used cars is what we're trying to predict in our upcoming analysis, we have to remove the cars, the rows, that don't have a listed price.

3. It can simply be done in one line of code using `dataframe.dropna`.
4. Setting the argument `inplace` to `True`, allows the modification to be done on the data set directly. In place equals `True`, just writes the result back into the data frame. This is equivalent to this line of code.

### Don't Forget

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

5. Don't forget that this line of code does not change the data frame but is a good way to make sure that you are performing the correct operation.

### To Modify the dataframe

### Don't Forget

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

To modify the data frame, you have to set the parameter `inplace` equal to `True`. You should always check the documentation if you are not familiar with the function or method.

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

## How to replace missing values in Python

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

The diagram illustrates the process of replacing a missing value in a DataFrame. On the left, a table shows a row with 'normalized-losses' containing 'NaN'. An arrow points to the right, where the same table is shown with the 'NaN' value replaced by '162'. This visualizes the Pandas `replace` method.

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

→

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

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

\*To replace missing values like NaNs with actual values, Pandas library has a built-in method called `replace` which can be used to fill in the missing values with the newly calculated values. As an example, assume that we want to replace the missing values of the variable `normalized losses` by the mean value of the variable. Therefore, the missing value should be replaced by the average of the entries within that column.

Steps:

1. first we calculate the mean of the column.
2. Then we use the method `replace` to specify the value we would like to be replaced as the first parameter, in this case `NaN`.
3. The second parameter is the value we would like to replace it with i.e the mean in this example.

This is a fairly simplified way of replacing missing values.

## 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

## Practice Quiz: Dealing with Missing Values in Python

Bookmarked

### Question 1

1/1 point (ungraded)

How would you access the column "body-style" from the dataframe `df`?

`df[ "body-style"]`

`df=="bodystyle"`



Submit

You have used 1 of 1 attempt

✓ Correct (1/1 point)

### Question 2

1/1 point (ungraded)

What is the correct symbol for missing data?

`nan`

`no-data`



Submit

You have used 1 of 1 attempt

✓ Correct (1/1 point)

## Data Wrangling / Data Formatting with Python

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

Data is usually collected from different places by different people which may be stored in different formats. Data formatting means bringing data into a common standard of expression that allows users to make meaningful comparisons.

As a part of dataset cleaning, data formatting ensures the data is consistent and easily understandable. For example, people may use different expressions to represent New York City, such as uppercase N uppercase Y, uppercase N lowercase y, uppercase N uppercase Y and New York. Sometimes, this unclean data is a good thing to see.

For example, if you're looking at the different ways people tend to write New York, then this is exactly the data that you want. Or if you're looking for ways to spot fraud, perhaps writing N.Y. is more likely to predict an anomaly than if someone wrote out New York in full. But perhaps more often than not, we just simply want to treat them all as the same entity or format to make statistical analyses easier down the road.

Referring to our used car dataset, there's a feature named city-miles per gallon in the dataset, which refers to a car fuel consumption in miles per gallon unit. However, you may be someone who lives in a country that uses metric units.

So, you would want to convert those values to liters per 100 kilometers, the metric version. To transform miles per gallon to liters per 100 kilometers, we need to divide 235 by each value in the city-miles per gallon column.

In Python, this can easily be done in one line of code. You take the column and set it to equal to 235, divide it by the entire column. In the second line of code, rename column name from city-miles per gallon to city-liters per 100 kilometers using the data frame rename method.

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

For example, here we noticed the assigned data type to the price feature is object. Although the expected data type should really be an integer or float type. It is important for later analysis to explore the features data type and convert them to the correct data types. Otherwise, the developed models later on may behave strangely, and totally valid data may end up being treated like missing data. There are many data types in pandas. Objects can be letters or words.

Int64 are integers and floats are real numbers. There are many others that we will not discuss. To identify features data type, in Python we can use the `dataframe.dtypes` method and check the data type of each variable in a data frame.

In the case of wrong data types, the method `dataframe.astype` can be used to convert a data type from one format to another. For example, using `astype int` for the price column,

you can convert the object column into an integer type variable.

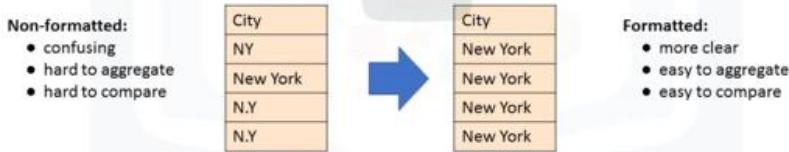
## Data Formatting

- 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.

data formatting ensures the data is consistent and easily understandable.

# 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.



## Applying calculations to an entire column

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

To transform miles per gallon to liters per 100 kilometers, we need to divide 235 by each value in the city-miles per gallon column. In Python, this can easily be done in one line of code. You take the column and set it to equal to 235, divide it by the entire column.

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

## Applying calculations to an entire column

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

In the second line of code, rename column name from city-miles per gallon to city-liters per 100 kilometers using the data frame rename method.

```
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
```

It is important for later analysis to explore the features data type and convert them to the correct data types.

Otherwise, the developed models later on may behave strangely, and totally valid data may end up being treated like missing data.

## 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")
```

## Practice Quiz: Data Formatting in Python

Bookmarked

### Question 1

1/1 point (ungraded)

How would you rename the column "city\_mpg" to "city-L/100km"?

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



#### Answer

Correct: Correct

Submit

You have used 1 of 1 attempt

---

✓ Correct (1/1 point)

## Data Wrangling: Data Normalization in Python

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

When we take a look at the used car data set, we notice in the data that the feature length ranges from 150-250, while feature width and height ranges from 50-100. We may want to normalize these variables so that the range of the values is consistent.

This normalization can make some statistical analyses easier down the road. By making the ranges consistent between variables, normalization enables a fair comparison between the different features, making sure they have the same impact.

It is also important for computational reasons. Here is another example that will help you understand why normalization is important. Consider a data set containing two features, age and income. Where age ranges from 0-100, while income ranges from 0-20,000 and higher.

Income is about 1,000 times larger than age and ranges from 20,000-500,000. So, these two features are in very different ranges.

When we do further analysis, like linear regression for example, the attribute income will intrinsically influence the result more due to its larger value. But this doesn't necessarily mean it is more important as a predictor. So, the nature of the data biases the linear regression model to weigh income more heavily than age. To avoid this, we can normalize these two variables into values that range from zero to one. Compare the two tables at the right. After normalization, both variables now have a similar influence on the models we will build later. There are several ways to normalize data.

I will just outline three techniques. The first method called simple feature scaling just divides each value by the maximum value for that feature. This makes the new values range between zero and one. The second method called min-max takes each value  $X_{old}$  subtract it from the minimum value of that feature, then divides by the range of that feature.

Again, the resulting new values range between zero and one. The third method is called z-score or standard score. In this formula for each value you subtract the mu which is the average of the feature, and then divide by the standard deviation sigma. The resulting values hover around zero, and typically range between negative three and positive three but can be higher or lower.

Following our earlier example, we can apply the normalization method on the length feature. First, we use the simple feature scaling method, where we divide it by the maximum value in the feature. Using the pandas method max,

this can be done in just one line of code. Here's the min-max method on the length feature. We subtract each value by the minimum of that column, then divide it by the range of that column.

The max minus the min. Finally, we apply the z-score method on length feature to normalize the values.

Here we apply the mean and STD method on the length feature.

Mean method will return the average value of the feature in the data set,

and STD method will return the standard deviation of the features in the data set.

## Data Normalization in Python

### 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



We may want to normalize these variables so that the range of the values is consistent.

### 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

Normalization enables a fair comparison between different Features making sure they have the same impact. Also important for computational reasons.

### 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

## Methods of normalizing data

Several approaches for normalization:

①

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



Simple Feature scaling

The first method called simple feature scaling just divides each value by the maximum value for that feature. This makes the new values range between zero and one.

## Methods of normalizing data

Several approaches for normalization:

②

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

Min-Max

The second method called min-max takes each value X\_old subtract it from the minimum value of that feature, then divides by the range of that feature. Again, the resulting new values range between zero and one.

## Methods of normalizing data

Several approaches for normalization:

③

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

Z-score

The third method is called z-score or standard score. In this formula for each value you subtract the mu which is the average of the feature, and then divide by the standard deviation sigma. The resulting values hover around zero, and typically range between negative three and positive three but can be higher or lower.

With Panda, the methods are:

1.

## 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()
```

2.

## 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())
```

3.

## 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()
```

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

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

Here's the min-max method on the length feature. We subtract each value by the minimum of that column, then divide it by the range of that column. The max minus the min.

we apply the z-score method on length feature to normalize the values. Here we apply the mean and STD method on the length feature. Mean method will return the average value of the feature in the data set, and STD method will return the standard deviation of the features in the data set.

## Practice Quiz: Data Normalization in Python

Bookmarked

### Question 1

1/1 point (ungraded)

Which of the following is the correct formula for z -score or data standardization?

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

a

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

b

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

c

c



Submit

You have used 1 of 2 attempts

---

✓ Correct (1/1 point)

# Binning

In this video, we'll talk about binning as a method of data pre-processing. Binning is when you group values together into bins. For example, you can bin "age" into [0 to 5], [6 to 10], [11 to 15] and so on.

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

As example, "price" here is an attribute range from 5,000 to 45,500. Using binning, we categorize the price into three bins: low price, medium price, and high prices.

In the actual car dataset, "price" is a numerical variable ranging from 5188 to 45400, it has 201 unique values. We can categorize them into 3 bins: low, medium, and high-priced cars. In Python we can easily implement the binning: We would like 3 bins of equal binwidth, so we need 4 numbers as dividers that are equal distance apart.

First we use the numpy function "linspace" to return the array "bins" that contains 4 equally spaced numbers over the specified interval of the price. We create a list "group\_names" that contains the different bin names. We use the pandas function "cut" to segment and sort the data values into bins. You can then use histograms to visualize the distribution of the data after they've been divided into bins.

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

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

## 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 is when you group values together into bins. For example, you can bin "age" into [0 to 5], [6 to 10], [11 to 15] and so on. Sometimes, binning can improve accuracy of the predictive models. In addition, sometimes we use data binning to group a set of numerical values into a smaller number of bins to have a better understanding of the data distribution.

# Binning in Python pandas

price
13495
16500
18920
41315
5151
6295
...

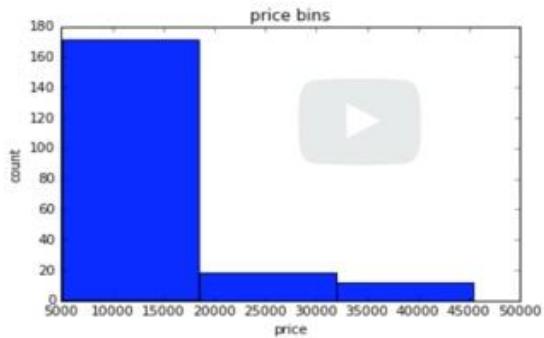


price	price-binned
13495	Low
16500	Low
18920	Medium
41315	High
5151	Low
6295	Low
...	...

```
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 )
```

## Visualizing binned data

- E.g., Histograms



## Turning Categorical Variables into Quantitative Variables in Python

In this video, we'll discuss how to turn categorical variables into quantitative variables in Python.

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

In the car data set, the fuel type feature as a categorical variable has two values, gas or diesel, which are in string format. For further analysis, Jerry has to convert these variables into some form of numeric format.

We encode the values by adding new features corresponding to each unique element in the original feature we would like to encode.

In the case where the feature fuel has two unique values, gas and diesel, we create two new features, gas and diesel. When a value occurs in the original feature, we set the corresponding value to one in the new feature. The rest of the features are set to zero.

In the fuel example, for car B, the fuel value is diesel. Therefore we set the feature diesel equal to one and the gas feature to zero. Similarly, for Car D, the fuel value is gas. Therefore we set the feature gas equal to one and the feature diesel equal to zero. This technique is often called "One-hot encoding".

In Pandas, we can use get\_dummies method to convert categorical variables to dummy variables.

In Python, transforming categorical variables to dummy variables is simple.

Following the example, `pd.get_dummies` method gets the fuel type column and creates the data frame `dummy_variable_1`. The `get_dummies` method automatically generates a list of numbers, each one corresponding to a particular category of the variable.

### Turning categorical variables into quantitative variables in Python

## 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'])
```

## Practice Quiz: Turning Categorical Variables into Quantitative Variables in Python

Bookmarked

### Question 1

1/1 point (ungraded)

Why do we convert values of Categorical Variables into numerical values?

Most statistical models cannot take in objects or strings as inputs

To save memory

✓  
**Answer**  
Correct: Correct

Submit

You have used 1 of 1 attempt

Show answer

✓ Correct (1/1 point)

## Lesson Summary

Bookmarked

In this lesson, you have learned how to:

- **Identify and Handle Missing Values:** Drop rows with incomplete information and impute missing data using the mean values.
- **Understand Data Formatting:** Wrangle features in a dataset and make them meaningful for data analysis.
- **Apply normalization to a data set:** By understanding the relevance of using feature scaling on your data and how normalization and standardization have varying effects on your data analysis.

## 7.2.Data Wrangling

Seongjoo Brenden Song edited this page on Nov 5, 2021 · 2 revisions

---

### Learning Objectives

- Describe how to handle missing values
  - Describe data formatting techniques
  - Describe data normalization
  - Demonstrate the use of binning
  - Demonstrate the use of categorical variables
- 

- [Data Wrangling](#)
- [Lab 2: Data Wrangling](#)

## 7.2.1.Data Wrangling

Seongjoo Brenden Song edited this page on Nov 5, 2021 · 2 revisions

### 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

- 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

#### Dealing 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

### Question

how would you deal with missing values for categorical data

- replace the missing value with the mode of the particular column
- replace the missing value with the mean of the particular column
- replace the missing value with the value that appears most often of the particular column

*Correct. this is called the mode*

### Drop missing values in Python

- Use `dataframes.dropna()`
  - `axis=0` : drops the entire row
  - `axis=1` : drops the entire column

```
df.dropna(subset=['column_name'], axis=0, inplace=True)
```

```
df = df.dropna(subset=['column_name'], axis=0)
```

## Replace missing values in Python

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

```
mean = df['column_name'].mean()  
df['column_name'].replace(np.nan, mean)
```

## Data Formatting in Python

### 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.

### Applying calculations to an entire column

- Example: 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.

### Correcting data types

- To *identify* datatypes:
  - 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')
```

## Data Normalization in Python

### Data Normalization

- Uniform the features value with different range.

## Methods of normalizing data

- Several approaches for normalization:
  - Simple Feature scaling
  - Min-Max
  - Z-score

### Simple Feature Scaling in Python

Example:

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

### Min-Max in Python

Example:

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

### Z-score in Python

Example:

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

## Binning in Python

### 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)

### Binning in Python pandas

Example

```
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)
```

# Turning categorical variables into quantitative variables in Python

## Categorical Variables

### Problem:

- Most statistical models cannot take in the objects/strings as input

### Categorical → Numeric

### Solution:

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

## Dummy variables in Python pandas

- Use pandas.get\_dummies() method
- Convert categorical variables to dummy variables (0 or 1)

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

## Lesson Summary

In this lesson, you have learned how to:

**Identify and Handle Missing Values:** Drop rows with incomplete information and impute missing data using the mean values.

**Understand Data Formatting:** Wrangle features in a dataset and make them meaningful for data analysis.

**Apply normalization to a data set:** By understanding the relevance of using feature scaling on your data and how normalization and standardization have varying effects on your data analysis.



IBM Developer  
SKILLS NETWORK

## Data Wrangling

Estimated time needed: **30** minutes

### Objectives

After completing this lab you will be able to:

- Handle missing values
- Correct data format
- Standardize and normalize data

### Table of Contents

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

## What is the purpose of data wrangling?

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

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

#### Import data

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

#### Import pandas

In [1]:

```
import pandas as pd
import matplotlib.pyplot as plt
```

## Reading the dataset from the URL and adding the related headers

First, we assign the URL of the dataset to "filename".

This dataset was hosted on IBM Cloud object. Click [HERE](#) for free storage.

```
In [2]: filename = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/auto."
```

Then, we create a Python list **headers** containing name of headers.

```
In [3]: headers = ["symboling", "normalized-losses", "make", "fuel-type", "aspiration", "num-of-doors", "body-style", "drive-wheels", "engine-location", "wheel-base", "length", "width", "height", "curb-weight", "engine-type", "num-of-cylinders", "engine-size", "fuel-system", "bore", "stroke", "compression-ratio", "horsepower", "peak-rpm", "city-mpg", "highway-mpg", "price"]
```

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

```
In [4]: df = pd.read_csv(filename, names = headers)
```

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

```
In [5]: # To see what the data set looks like, we'll use the head() method.  
df.head()
```

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke	compression-ratio	horsepower	peak-rpm	city-mpg
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	9.0	154	5000	19
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	10.0	102	5500	24
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	8.0	115	5500	18

5 rows × 26 columns

highway-mpg price

27 13495

27 16500

26 16500

30 13950

22 17450

As we can see, several question marks appeared in the dataframe; those are missing values which may hinder our further analysis.

So, how do we identify all those missing values and deal with them?

## How to work with missing data?

Steps for working with missing data:

1. Identify missing data
2. Deal with missing data
3. Correct data format

## Identify and handle missing values

### Identify missing values

#### Convert "?" to NaN

In the car dataset, missing data comes with the question mark "?". We replace "?" with NaN (Not a Number), Python's default missing value marker for reasons of computational speed and convenience. Here we use the function:

```
.replace(A, B, inplace = True)
```

to replace A by B.

```
In [6]:  
import numpy as np  
  
# replace "?" to NaN  
df.replace("?", np.nan, inplace = True)  
df.head(5)
```

Out[6]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke	compression-ratio	horsepower	peak-rpm	city-mpg
0	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21
1	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21
2	1	NaN	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	9.0	154	5000	19
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	10.0	102	5500	24
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	8.0	115	5500	18

5 rows × 26 columns

highway-  
mpg price

27 13495

27 16500

26 16500

30 13950

22 17450

## Evaluating for Missing Data

The missing values are converted by default. We use the following functions to identify these missing values. There are two methods to detect missing data:

1. `.isnull()`

2. `.notnull()`

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

In [7]:

```
missing_data = df.isnull()
missing_data.head(5)
```

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

In [7]:

```
missing_data = df.isnull()
missing_data.head(5)
```

Out[7]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke	compression-ratio	horsepower	peak-rpm	city-mpg
0	False	True	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	
1	False	True	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	
2	False	True	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	
3	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	
4	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	

5 rows × 26 columns

highway-  
mpg price

False False

False False

False False

False False

False False

"True" means the value is a missing value while "False" means the value is not a missing value.

## Count missing values in each column

Using a for loop in Python, we can quickly figure out the number of missing values in each column. As mentioned above, "True" represents a missing value and "False" means the value is present in the dataset. In the body of the for loop the method ".value\_counts()" counts the number of "True" values.

```
In [8]: for column in missing_data.columns.values.tolist():
    print(column)
    print(missing_data[column].value_counts())
    print("")
```

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

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

make
False 205
Name: make, dtype: int64

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

aspiration
False 205
Name: aspiration, dtype: int64

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

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

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

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

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

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

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

length
False 205
Name: length, dtype: int64

width
False 205
Name: width, dtype: int64

height
False 205
Name: height, dtype: int64

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

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

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

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

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

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

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

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

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

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

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

Based on the summary above, each column has 205 rows of data and seven of the 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

## Calculate the mean value for the "normalized-losses" column

```
In [9]: avg_norm_loss = df["normalized-losses"].astype("float").mean(axis=0)
print("Average of normalized-losses:", avg_norm_loss)

Average of normalized-losses: 122.0
```

Replace "NaN" with mean value in "normalized-losses" column

```
In [10]: df["normalized-losses"].replace(np.nan, avg_norm_loss, inplace=True)
```

Calculate the mean value for the "bore" column

```
In [11]: avg_bore=df['bore'].astype('float').mean(axis=0)
print("Average of bore:", avg_bore)

Average of bore: 3.3297512437810943
```

Replace "NaN" with the mean value in the "bore" column

```
In [12]: df["bore"].replace(np.nan, avg_bore, inplace=True)
```

## Question #1:

Based on the example above, replace NaN in "stroke" column with the mean value.

```
[n 14]: # Write your code below and press Shift+Enter to execute
avg_stroke = df['stroke'].astype('float').mean(axis=0)
print('Average of stroke:', avg_stroke)

df['stroke'].replace(np.nan, avg_stroke, inplace=True)

Average of stroke: 3.255422885572139
```

Click here for the solution ``python #Calculate the mean vaule for "stroke" column avg\_stroke = df["stroke"].astype("float").mean(axis = 0) print("Average of stroke:", avg\_stroke) # replace NaN by mean value in "stroke" column df["stroke"].replace(np.nan, avg\_stroke, inplace = True) ````

## Calculate the mean value for the "horsepower" column

```
In [15]: avg_horsepower = df['horsepower'].astype('float').mean(axis=0)
print("Average horsepower:", avg_horsepower)
```

```
Average horsepower: 104.25615763546799
```

Replace "NaN" with the mean value in the "horsepower" column

```
In [16]: df['horsepower'].replace(np.nan, avg_horsepower, inplace=True)
```

Calculate the mean value for "peak-rpm" column

```
In [17]: avg_peakrpm=df['peak-rpm'].astype('float').mean(axis=0)
print("Average peak rpm:", avg_peakrpm)
```

```
Average peak rpm: 5125.369458128079
```

Replace "NaN" with the mean value in the "peak-rpm" column

```
In [18]: df['peak-rpm'].replace(np.nan, avg_peakrpm, inplace=True)
```

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

```
In [19]: df['num-of-doors'].value_counts()
```

```
Out[19]: four    114
two      89
Name: num-of-doors, dtype: int64
```

We can see that four doors are the most common type. We can also use the ".idxmax()" method to calculate the most common type automatically:

```
In [20]: df['num-of-doors'].value_counts().idxmax()
```

```
Out[20]: 'four'
```

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

```
In [21]: #replace the missing 'num-of-doors' values by the most frequent
df["num-of-doors"].replace(np.nan, "four", inplace=True)
```

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

```
In [22]: # simply drop whole row with NaN in "price" column
df.dropna(subset=["price"], axis=0, inplace=True)

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

```
In [23]: df.head()
```

```
Out[23]:
```

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke	compression-ratio	horsepower	peak-rpm	city-mpg
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	9.0	154	5000	19
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	10.0	102	5500	24
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	8.0	115	5500	18

5 rows × 26 columns

highway-mpg	price
27	13495
27	16500
26	16500
30	13950
22	17450

**Good!** Now, we have a dataset with no missing values.

## Correct data format

We are almost there!

The last step in data cleaning is checking and making sure that all data is in the correct format (int, float, text or other).

In Pandas, we use:

**.dtype()** to check the data type

**.astype()** to change the data type

**Let's list the data types for each column**

```
In [24]: df.dtypes
```

```
Out[24]: symboling      int64
normalized-losses    object
make                 object
fuel-type            object
aspiration           object
num-of-doors         object
body-style            object
drive-wheels          object
engine-location       object
wheel-base            float64
length                float64
width                 float64
height                float64
curb-weight           int64
engine-type           object
num-of-cylinders     object
engine-size            int64
fuel-system            object
bore                  object
stroke                object
compression-ratio     float64
horsepower             object
peak-rpm               object
city-mpg               int64
highway-mpg             int64
price                 object
dtype: object
```

As we can see above, some columns are not of the correct data type. Numerical variables should have type 'float' or 'int', and variables with strings such as categories should have type 'object'. For example, 'bore' and 'stroke' variables are numerical values that describe the engines, so we should expect them to be of the type 'float' or 'int'; however, they are shown as type 'object'. We have to convert data types into a proper format for each column using the "astype()" method.

### Convert data types to proper format

```
In [25]: df[["bore", "stroke"]] = df[["bore", "stroke"]].astype("float")
df[["normalized-losses"]] = df[["normalized-losses"]].astype("int")
df[["price"]] = df[["price"]].astype("float")
df[["peak-rpm"]] = df[["peak-rpm"]].astype("float")
```

Let us list the columns after the conversion

```
In [26]: df.dtypes
```

```
Out[26]: symboling      int64
normalized-losses   int64
make                object
fuel-type          object
aspiration         object
num-of-doors       object
body-style          object
drive-wheels        object
engine-location     object
wheel-base          float64
length              float64
width               float64
height              float64
curb-weight         int64
engine-type         object
num-of-cylinders    object
engine-size          int64
fuel-system          object
bore                float64
stroke              float64
compression-ratio   float64
horsepower          object
peak-rpm             float64
city-mpg             int64
highway-mpg          int64
price               float64
dtype: object
```

Wonderful!

Now we have finally obtained the cleaned dataset with no missing values with all data in its proper format.

## Data Standardization

Data is usually collected from different agencies in different formats. (Data standardization is also a term for a particular type of data normalization where we subtract the mean and divide by the standard deviation.)

### What is standardization?

Standardization is the process of transforming data into a common format, allowing the researcher to make the meaningful comparison.

### Example

Transform mpg to L/100km:

In our dataset, the fuel consumption columns "city-mpg" and "highway-mpg" are represented by mpg (miles per gallon) unit. Assume we are developing an application in a country that accepts the fuel consumption with L/100km standard.

We will need to apply **data transformation** to transform mpg into L/100km.

The formula for unit conversion is:

$$\text{L/100km} = 235 / \text{mpg}$$

We can do many mathematical operations directly in Pandas.

In [27]:

```
df.head()
```

Out[27]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke	compression-ratio	horsepower	peak-rpm	city-mpg
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000.0	21
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000.0	21
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	9.0	154	5000.0	19
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	10.0	102	5500.0	24
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	8.0	115	5500.0	18

5 rows × 26 columns

highway-mpg price

27	13495.0
27	16500.0
26	16500.0
30	13950.0
22	17450.0

Click here for the solution ````python # transform mpg to L/100km by mathematical operation (235 divided by mpg)  
`df["highway-mpg"] = 235/df["highway-mpg"] # rename column name from "highway-mpg" to "highway-L/100km"`  
`df.rename(columns={"highway-mpg":'highway-L/100km'}, inplace=True) # check your transformed data df.head() `````

## Data Normalization

### Why normalization?

Normalization is the process of transforming values of several variables into a similar range. Typical normalizations include scaling the variable so the variable average is 0, scaling the variable so the variance is 1, or scaling the variable so the variable values range from 0 to 1.

### Example

To demonstrate normalization, let's say we want to scale the columns "length", "width" and "height".

**Target:** would like to normalize those variables so their value ranges from 0 to 1

**Approach:** replace original value by (original value)/(maximum value)

```
In [30]: # replace (original value) by (original value)/(maximum value)
df['length'] = df['length']/df['length'].max()
df['width'] = df['width']/df['width'].max()
```

## Question #3:

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

```
In [33]: # Write your code below and press Shift+Enter to execute
df['height'] = df['height']/df['height'].max()

df[['length', 'width', 'height']].head()
```

Click here for the solution `python df['height'] = df['height']/df['height'].max() # show the scaled columns df[['length", "width", "height']].head()`

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

## Binning

### Why binning?

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

#### Example:

In our dataset, "horsepower" is a real valued variable ranging from 48 to 288 and it has 57 unique values. What if we only care about the price difference between cars with high horsepower, medium horsepower, and little horsepower (3 types)? Can we rearrange them into three 'bins' to simplify analysis?

We will use the pandas method 'cut' to segment the 'horsepower' column into 3 bins.

#### Example of Binning Data In Pandas

Convert data to correct format:

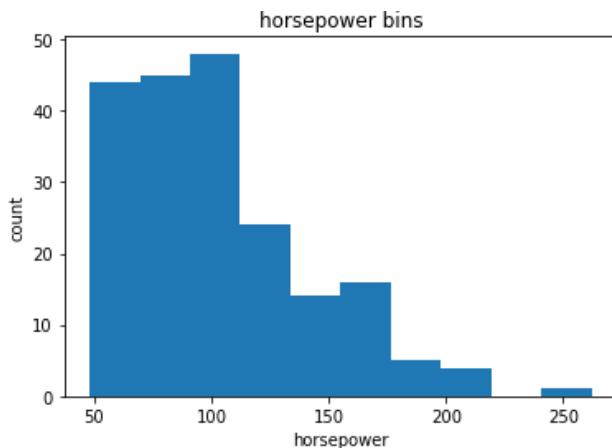
```
In [34]: df["horsepower"] = df["horsepower"].astype(int, copy=True)
```

Let's plot the histogram of horsepower to see what the distribution of horsepower looks like

```
In [35]: %matplotlib inline
import matplotlib as plt
from matplotlib import pyplot
plt.pyplot.hist(df["horsepower"])

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

```
Out[35]: Text(0.5, 1.0, 'horsepower bins')
```



We would like 3 bins of equal size bandwidth so we use numpy's `linspace(start_value, end_value, numbers_generated)` function.

Since we want to include the minimum value of horsepower, we want to set `start_value = min(df["horsepower"])`.

Since we want to include the maximum value of horsepower, we want to set `end_value = max(df["horsepower"])`.

Since we are building 3 bins of equal length, there should be 4 dividers, so `numbers_generated = 4`.

We build a bin array with a minimum value to a maximum value by using the bandwidth calculated above. The values will determine when one bin ends and another begins.

```
In [36]: bins = np.linspace(min(df["horsepower"]), max(df["horsepower"]), 4)
bins
```

```
Out[36]: array([ 48.          , 119.33333333, 190.66666667, 262.          ])
```

We set group names:

```
In [37]: group_names = ['Low', 'Medium', 'High']
```

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

```
In [38]: df['horsepower-binned'] = pd.cut(df['horsepower'], bins, labels=group_names, include_lowest=True )
df[['horsepower','horsepower-binned']].head(20)
```

Out[38]:

	horsepower	horsepower-binned
0	111	Low
1	111	Low
2	154	Medium
3	102	Low
4	115	Low
5	110	Low
6	110	Low
7	110	Low
8	140	Medium
9	101	Low
10	101	Low

11	121	Medium
12	121	Medium
13	121	Medium
14	182	Medium
15	182	Medium
16	182	Medium
17	48	Low
18	70	Low
19	70	Low

Let's see the number of vehicles in each bin:

In [39]:

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

Out[39]:

Low	153
Medium	43
High	5
Name: horsepower-binned, dtype: int64	

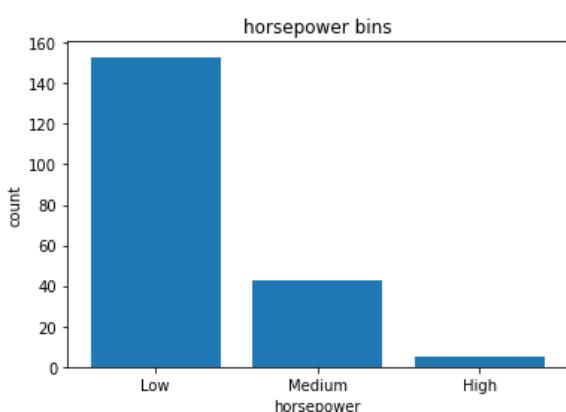
Let's plot the distribution of each bin:

In [40]:

```
%matplotlib inline
import matplotlib as plt
from matplotlib import pyplot
pyplot.bar(group_names, df["horsepower-binned"].value_counts())

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

Out[40]:



Look at the dataframe above carefully. You will find that the last column provides the bins for "horsepower" based on 3 categories ("Low", "Medium" and "High").

We successfully narrowed down the intervals from 57 to 3!

## Bins Visualization

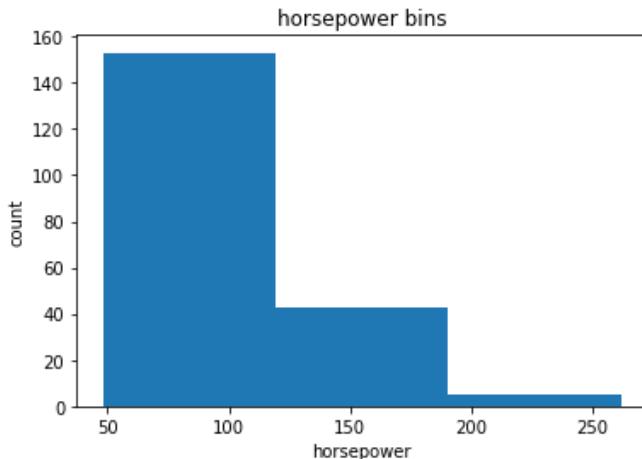
Normally, a histogram is used to visualize the distribution of bins we created above.

```
In [41]: %matplotlib inline
import matplotlib as plt
from matplotlib import pyplot

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

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

Out[41]: Text(0.5, 1.0, 'horsepower bins')



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

## Indicator Variable (or Dummy Variable)

### What is an indicator variable?

An indicator variable (or dummy variable) is a numerical variable used to label categories. They are called 'dummies' because the numbers themselves don't have inherent meaning.

### Why we use indicator variables?

We use indicator variables so we can use categorical variables for regression analysis in the later modules.

### Example

We see the column "fuel-type" has two unique values: "gas" or "diesel". Regression doesn't understand words, only numbers. To use this attribute in regression analysis, we convert "fuel-type" to indicator variables.

We will use pandas' method 'get\_dummies' to assign numerical values to different categories of fuel type.

In [42]:

```
df.columns
```

Out[42]:

```
Index(['symboling', 'normalized-losses', 'make', 'fuel-type', 'aspiration',  
       'num-of-doors', 'body-style', 'drive-wheels', 'engine-location',  
       'wheel-base', 'length', 'width', 'height', 'curb-weight', 'engine-type',  
       'num-of-cylinders', 'engine-size', 'fuel-system', 'bore', 'stroke',  
       'compression-ratio', 'horsepower', 'peak-rpm', 'city-mpg',  
       'highway-L/100km', 'price', 'city-L/100km', 'horsepower-binned'],  
      dtype='object')
```

Get the indicator variables and assign it to data frame "dummy\_variable\_1":

In [43]:

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

Out[43]:

	diesel	gas
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1

Change the column names for clarity:

In the dataframe, column 'fuel-type' has values for 'gas' and 'diesel' as 0s and 1s now.

In [45]:

```
# merge data frame "df" and "dummy_variable_1"  
df = pd.concat([df, dummy_variable_1], axis=1)  
  
# drop original column "fuel-type" from "df"  
df.drop("fuel-type", axis = 1, inplace=True)
```

In [46]:

```
df.head()
```

Out[46]:

	symboling	normalized-losses	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	length	...	compression-ratio	horsepower	peak-rpm	city-mpg	highway-L/100km	price	city-L/100km
0	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	...	9.0	111	5000.0	21	8.703704	13495.0	11.190476
1	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	...	9.0	111	5000.0	21	8.703704	16500.0	11.190476
2	1	122	alfa-romero	std	two	hatchback	rwd	front	94.5	0.822681	...	9.0	154	5000.0	19	9.038462	16500.0	12.368421
3	2	164	audi	std	four	sedan	fwd	front	99.8	0.848630	...	10.0	102	5500.0	24	7.833333	13950.0	9.791667
4	2	164	audi	std	four	sedan	4wd	front	99.4	0.848630	...	8.0	115	5500.0	18	10.681818	17450.0	13.055556

5 rows × 29 columns

horsepower-binned	fuel-type-diesel	fuel-type-gas
Low	0	1
Low	0	1
Medium	0	1
Low	0	1
Low	0	1

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

## Question #4:

Similar to before, create an indicator variable for the column "aspiration"

```
In [48]: # Write your code below and press Shift+Enter to execute
dummy_variable_2 = pd.get_dummies(df['aspiration'])

dummy_variable_2.rename(columns={'std':'aspiration-std', 'turbo':'aspiration-turbo'}, inplace=True)

dummy_variable_2.head()
```

```
Out[48]: aspiration-std aspiration-turbo
0 1 0
1 1 0
2 1 0
3 1 0
4 1 0
```

Click here for the solution ``python # get indicator variables of aspiration and assign it to data frame  
``dummy\_variable\_2" dummy\_variable\_2 = pd.get\_dummies(df['aspiration']) # change column names for clarity  
dummy\_variable\_2.rename(columns={'std':'aspiration-std', 'turbo': 'aspiration-turbo'}, inplace=True) # show first 5  
instances of data frame "dummy\_variable\_1" dummy\_variable\_2.head() ````

## Question #5:

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

```
In [49]: # Write your code below and press Shift+Enter to execute
df = pd.concat([df, dummy_variable_2], axis=1)

df.drop('aspiration', axis=1, inplace=True)
```

Click here for the solution ``python # merge the new dataframe to the original datafram df = pd.concat([df, dummy\_variable\_2], axis=1) # drop original column "aspiration" from "df" df.drop('aspiration', axis = 1, inplace=True) ````

Save the new csv:

```
In [50]: df.to_csv('clean_df.csv')
```

## Graded Quiz: Data Wrangling

Bookmark

Graded Quiz due Feb 10, 2022 21:32 +08 Completed

### Question 1

0/1 point (graded)

What task do the following lines of code perform?

```
avg=df['horsepower'].mean(axis=0)
```

```
df['horsepower'].replace(np.nan, avg)
```

- calculate the mean value for the 'horsepower' column and replace all the NaN values of that column by the mean value
- nothing; because the parameter inplace is not set to true
- replace all the NaN values with the mean.

#### Answer

Incorrect: Incorrect, check the parameters of the method replace.

Submit

You have used 2 of 2 attempts

Show answer

**Correct answer is “nothing: because the parameter inplace is not set to true”**

### Question 2

1/1 point (graded)

Consider the dataframe df; convert the column df["city-mpg"] to df["city-L/100km"] by dividing 235 by each element in the column 'city-mpg'.

- `df['city-L/100km'] = 235/df['city-mpg']`
- `df['city-L/100km'] = df['city-mpg'].div(235)`



#### Answer

Correct: Correct

Submit

You have used 1 of 1 attempt

Show answer

### Question 3

1/1 point (graded)

What data type is the following set of numbers? 666, 1.1,232,23.12

int

object

float



#### Answer

Correct: Correct

[Submit](#)

You have used 1 of 2 attempts

[Reset](#)

[Show answer](#)

---

### Question 4

1/1 point (graded)

The following code is an example of:

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

min-max scaling

simple feature scaling

z-score



#### Answer

Correct: Correct

[Submit](#)

You have used 1 of 2 attempts

## Question 5

1/1 point (graded)

Consider the two columns 'horsepower', and 'horsepower-binned'; from the dataframe df; how many categories are there in the 'horsepower-binned' column?

	horsepower	horsepower-binned
0	111.0	Medium
1	111.0	Medium
2	154.0	Medium
3	102.0	Medium
4	115.0	Medium
5	110.0	Medium
18	70.0	Low
19	70.0	Low
3		✓
3		

## Module 3 - Exploratory Data Analysis

---

Module Introduction & Learning Objectives 1 min

Video: Exploratory Data Analysis (1:24) 2 min

Video: Descriptive Statistics (4:44) 5 min

Practice Quiz: Descriptive Statistics 1 activity

Video: GroupBy in Python (3:26) 4 min

Practice Quiz: GroupBy in Python 1 activity

Video: Correlation (2:33) 3 min

Practice Quiz: Correlation 1 activity

Video: Correlation - Statistics (2:42) 3 min

Practice Quiz: Correlation - Statistics 1 activity

Reading: Lesson Summary 1 min

Hands-on Lab: Exploratory Data Analysis 1 min + 1 activity

Graded Quiz: Exploratory Data Analysis (5 Questions) 1 activity

Graded Quiz due Feb 14, 2022, 9:32 AM GMT+8

## Module Introduction & Learning Objectives

 Bookmarked

### Module Introduction

In this week's module, you will learn what is meant by exploratory data analysis, and you will learn how to perform computations on the data to calculate basic descriptive statistical information, such as mean, median, mode, and quartile values, and use that information to better understand the distribution of the data. You will learn about putting your data into groups to help you visualize the data better, you will learn how to use the Pearson correlation method to compare two continuous numerical variables, and you will learn how to use the Chi-square test to find the association between two categorical variables and how to interpret them.

### Learning Objectives

- Descriptive Statistics
- Basic of Grouping
- ANOVA
- Correlation

## Exploratory Data Analysis

In this module we're going to cover the basics of exploratory data analysis using python.

Exploratory data analysis or in short EDA is an approach to analyze data in order to summarize main characteristics of the data gain better understanding of the data set, uncover relationships between different variables, and extract important variables for the problem we're trying to solve.

The main question we are trying to answer in this module is what are the characteristics that have the most impact on the car price? We will be going through a couple of different useful exploratory data analysis techniques in order to answer this question: In this module you will learn about descriptive statistics, which describe basic features of a data set and obtains a short summary about the sample and measures of the data. Basic of grouping data using group by and how this can help to transform our data set, the correlation between different variables, and lastly advanced correlation. Where we'll introduce you to various correlation statistical methods namely pearson correlation and correlation heat maps

# Exploratory Data Analysis

Module 3 Introduction

## 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
- Correlation
- Correlation - Statistics

## Descriptive Statistics

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

Descriptive statistical analysis helps to describe basic features of a data set, and obtains a short summary about the sample and measures of the data.

Let's show you a couple different useful methods. One way in which we can do this is by using the describe function in pandas. Using the describe function and applying it on your data frame, the describe function automatically computes basic statistics for all numerical variables.

It shows the mean, the total number of data points, the standard deviation, the quartiles and the extreme values. Any NAN values are automatically skipped in these statistics.

This function will give you a clear idea of the distribution of your different variables. You could have also categorical variables in your data set. These are variables that can be divided up into different categories or groups, and have discrete values. For example, in our data set we have the drive system as a categorical variable, which consists of the categories, forward wheel drive, rear wheel drive and four wheel drive.

One way you can summarize the categorical data, is by using the function value\_counts. We can change the name of the column to make it easier to read. We see that we have 118 cars in the front wheel drive category. 75 cars in the rear wheel drive category, and 8 cars in the four wheel drive category. Box plots are a great way to visualize numeric data, since you can visualize the various distributions of the data. The main features that the box plot shows, are the median of the data, which represents where the middle data point is. The upper quartile shows where the 75th percentile is. The lower quartile shows where the 25th percentile is.

The data between the upper and lower quartile represents the interquartile range. Next you have the lower and upper extremes. These are calculated as 1.5 times the interquartile range, above the 75th percentile, and as 1.5 times the IQR below the 25th percentile. Finally, box plots also display outliers as individual dots that occur outside the upper and lower extremes. With box plots, you can easily spot outliers, and also see the distribution and skewness of the data. Box plots make it easy to compare between groups. In this example, using box plot we can see the distribution of different categories of the drive wheels feature over price feature.

We can see that the distribution of price between the rear wheel drive, and the other categories are distinct. But the price for front wheel drive and four wheel drive are almost indistinguishable. Often times we tend to see continuous variables in our data. These data points are numbers contained in some range. For example, in our data set price and engine size are continuous variables.

What if we want to understand the relationship between engine size and price. Could engine size possibly predict the price of a car? One good way to visualize this is using a scatter plot. Each observation in the scatter plot is represented as a point. This plot shows the relationship between two variables. The predictor variable, is the variable that you are using to predict an outcome. In this case our predictor variable is the engine size. The target variable is the variable that you are trying to predict. In this case, our target variable is the price. Since this would be the outcome.

In a scatter plot, we typically set the predictor variable on the x-axis or horizontal axis, and we set the target variable on the y-axis or vertical axis. In this case, we will thus plot the engine size on the x-axis and the price on the y-axis. We are using, the matplotlib functions scatter here, taking in x and y variable. Something to note is that it's always important to label your axes, and write a general plot title, so that you know what you're looking at. Now how is the variable engine size related to price? From the scatter plot, we see that as the engine size goes up, the price of the car also goes up. This is giving us an initial indication that there is a positive linear relationship between these two variables.

## 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

## Descriptive Statistics- Describe()

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

```
df.describe()
```

## 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	58.157861	1.254802	35.442168	6.066366	12.322175	2.101471	2.447822	517.296727	41.546834	0.280130	0.316048
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

For Categorical Variables in the dataset:

## Descriptive Statistics - Value\_Counts()

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

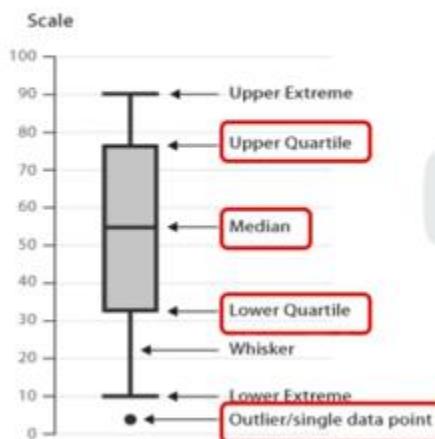
```
drive_wheels_counts=df["drive-wheels"].value_counts().to_frame()  
  
drive_wheels_counts.rename(columns={'drive-wheels':'value_counts'}, inplace=True)  
drive_wheels_counts
```

	value_counts
drive-wheels	
fwd	118
rwd	75
4wd	8

DM Developers

DATA SCIENCE

## Descriptive Statistics - Box Plots

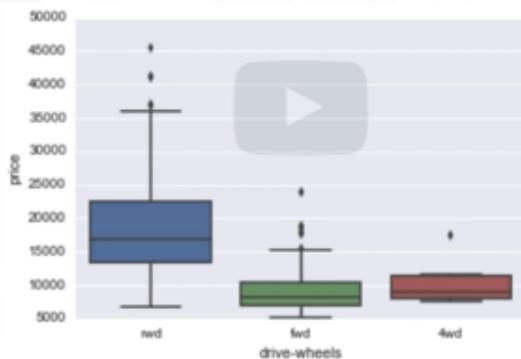


The main features that the box plot shows, are the median of the data, which represents where the middle data point is.

The upper quartile shows where the 75th percentile is. The lower quartile shows where the 25th percentile is. The data between the upper and lower quartile represents the interquartile range. Next you have the lower and upper extremes. These are calculated as 1.5 times the interquartile range, above the 75th percentile, and as 1.5 times the IQR below the 25th percentile. Finally, box plots also display outliers as individual dots that occur outside the upper and lower extremes.

## Box Plot - Example

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



With box plots, you can easily spot outliers, and also see the distribution and skewness of the data. Box plots make it easy to compare between groups. In this example, using box plot we can see the distribution of different categories of the drive wheels feature over price feature. We can see that the distribution of price between the rear wheel drive, and the other categories are distinct. But the price for front wheel drive and four wheel drive are almost indistinguishable.

## 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.

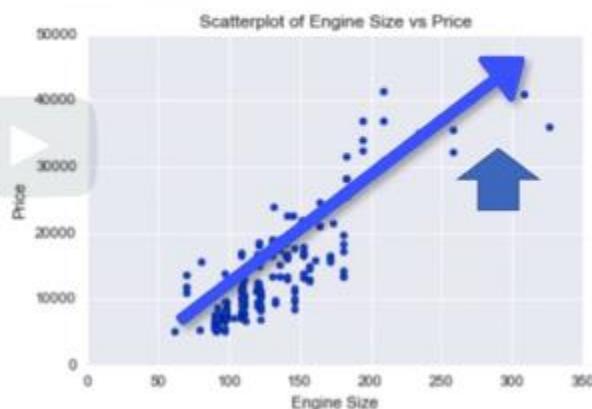
In a scatter plot, we typically set the predictor variable on the x-axis or horizontal axis, and we set the target variable on the y-axis or vertical axis. In this case, we will thus plot the engine size on the x-axis and the price on the y-axis.

In this case our predictor variable is the engine size. The target variable is the variable that you are trying to predict. In this case, our

target variable is the price. Since this would be the outcome. In a scatter plot, we typically set the predictor variable on the x-axis or horizontal axis, and we set the target variable on the y-axis or vertical axis. In this case, we will thus plot the engine size on the x-axis and the price on the y-axis. We are using, the matplotlib functions

## 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" )
```



scatter here, taking in x and y variable. Something to note is that it's always important to label your axes, and write a general plot title, so that you know what you're looking at. Now how is the variable engine size related to price? From the scatter plot, we see that as the engine size goes up, the price of the car also goes up. This is giving us an initial indication that there is a positive linear relationship between these two variables.

## Practice Quiz: Descriptive Statistics

Bookmarked

### Question 1

1/1 point (ungraded)

Consider the following scatter plot; what kind of relationship do the two variables have?



- negative linear relationship
- positive linear relationship



#### Answer

Correct: Correct

## GroupBy in Python

In this video, we'll cover the basics of grouping and how this can help to transform our dataset. Assume you want to know, is there any relationship between the different types of drive system, forward, rear, and four-wheel drive, and the price of the vehicles?

If so, which type of drive system adds the most value to a vehicle? It would be nice if we could group all the data by the different types of drive wheels and compare the results of these different drive wheels against each other. In Pandas, this can be done using the group by method. The group by method is used on categorical variables, groups the data into subsets according to the different categories of that variable.

You can group by a single variable or you can group by multiple variables by passing in multiple variable names. As an example, let's say we are interested in finding the average price of vehicles and observe how they differ between different types of body styles and drive wheels variables. To do this, we first pick out the three data columns we are interested in, which is done in the first line of code. We then group the reduced data according to drive wheels and body style in the second line. Since we are interested in knowing how the average price differs across the board, we can take the mean of each group and append it this bit at the very end of the line too.

The data is now grouped into subcategories and only the average price of each subcategory is shown. We can see that according to our data, rear wheel drive convertibles and rear wheel drive hardtops have the highest value while four wheel drive hatchbacks have the lowest value. A table of this form isn't the easiest to read and also not very easy to visualize. To make it easier to understand, we can transform this table to a pivot table by using the pivot method.

In the previous table, both drive wheels and body style were listening columns. A pivot table has one variable displayed along the columns and the other variable displayed along the rows. Just with one line of code and by using the Panda's pivot method, we can pivot the body style variable so it is displayed along the columns and the drive wheels will be displayed along the rows. The price data now becomes a rectangular grid, which is easier to visualize.

This is similar to what is usually done in Excel spreadsheets. Another way to represent the pivot table is using a heat map plot. Heat map takes a rectangular grid of data and assigns a color intensity based on the data value at the grid points. It is a great way to plot the target variable over multiple variables and through this get visual clues with the relationship between these variables and the target. In this example, we use pyplot's p color method to plot heat map and convert the previous pivot table into a graphical form. We specify the red-blue color scheme.

In the output plot, each type of body style is numbered along the x-axis and each type of drive wheels is numbered along the y-axis. The average prices are plotted with varying colors based on their values. According to the color bar, we see that the top section of the heat map seems to have higher prices than the bottom section.

## GroupBy in Python

### Grouping data

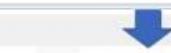
- Use Panda **dataframe. Groupby()** method:
  - Can be applied on categorical variables
  - Group data into categories
  - Single or multiple variables

## 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	convertible	20239.229524
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

IBM Developer



SKILLS NETWORK

## Pandas method - Pivot()

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

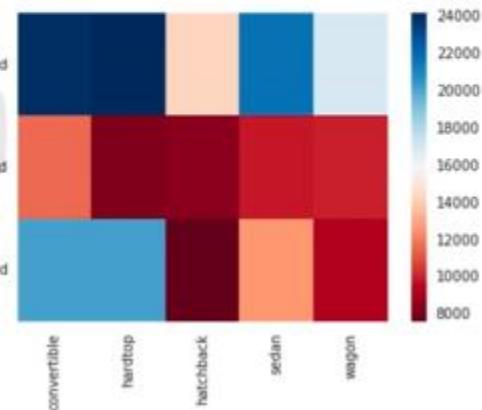
```
df_pivot = df_grp.pivot(index= 'drive-wheels', columns='body-style')
```

	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()
```



## Practice Quiz: GroupBy in Python

Bookmarked

### Question 1

1/1 point (ungraded)

Select the appropriate description of a pivot table:

- A pivot table has one variable displayed along the columns and the other variable displayed along the rows.
- A pivot table contains statistical information for each column.



#### Answer

Correct: Correct

Submit

You have used 1 of 1 attempt

Show answer

---

✓ Correct (1/1 point)

## Correlation

Correlation between different variables. Correlation is a statistical metric for measuring to what extent different variables are interdependent. In other words, when we look at two variables over time, if one variable changes how does this affect change in the other variable?

For example, smoking is known to be correlated to lung cancer since you have a higher chance of getting lung cancer if you smoke. In another example, there is a correlation between umbrella and rain variables where more precipitation means more people use umbrellas. Also, if it doesn't rain people would not carry umbrellas. Therefore, we can say that umbrellas and rain are interdependent and by definition they are correlated.

It is important to know that correlation doesn't imply causation. In fact, we can say that umbrella and rain are correlated but we would not have enough information to say whether the umbrella caused the rain or the rain caused the umbrella.

In data science we usually deal more with correlation. Let's look at the correlation between engine size and price. This time we'll visualize these two variables using a scatter plot and an added linear line called a regression line, which indicates the relationship between the two. The main goal of this plot is to see whether the engine size has any impact on the price.

In this example, you can see that the straight line through the data points is very steep which shows that there's a positive linear relationship between the two variables. With increase in values of engine size, values of price go up as well and the slope of the line is positive. So there is a positive correlation between engine size and price.

We can use `seaborn.regplot` to create the scatter plot. As another example, now let's look at the relationship between highway miles per gallon to see its impact on the car price. As we can see in this plot, when highway miles per gallon value goes up the value price goes down. Therefore there is a negative linear relationship between highway miles per gallon and price. Although this relationship is negative the slope of the line is steep which means that the highway miles per gallon is still a good predictor of price. These two variables are said to have a negative correlation.

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

# Correlation

## Correlation

### What is Correlation?

- Measures to what extent different variables are interdependent

- For example:

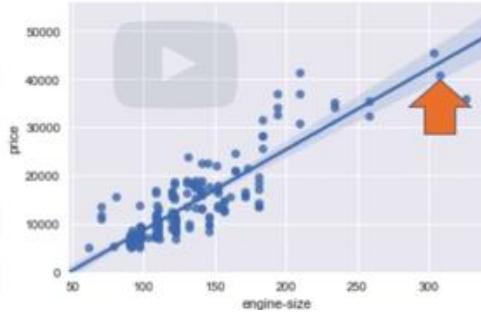
- Lung cancer → Smoking
- Rain → Umbrella

- 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,)
```

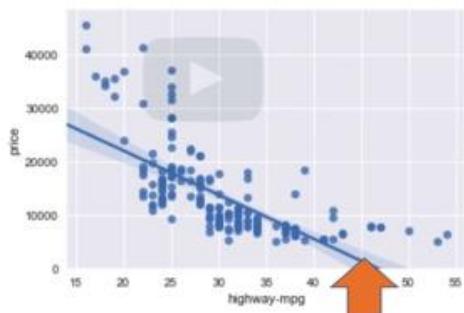


The main goal of this plot is to see whether the engine size has any impact on the price. In this example, you can see that the straight line through the data points is very steep which shows that there's a positive linear relationship between the two variables. With increase in values of engine size, values of price go up as well and the slope of the line is positive. So there is a positive correlation between engine size and price.

## Correlation - Negative Linear Relationship

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

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

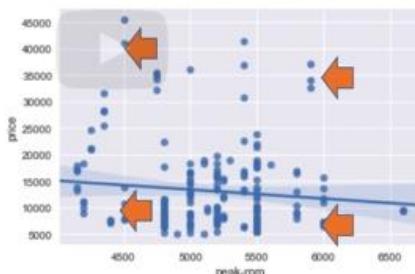


The relationship between highway miles per gallon to see its impact on the car price. As we can see in this plot, when highway miles per gallon value goes up the value price goes down. Therefore there is a negative linear relationship between highway miles per gallon and price. Although this relationship is negative the slope of the line is steep which means that the highway miles per gallon is still a good predictor of price. These two variables are said to have a negative correlation.

## Correlation - Negative Linear Relationship

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

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



an example of a weak correlation. For example, both low peak RPM and high values of peak RPM have low and high prices.

Therefore, we cannot use RPM to predict the values.

## Practice Quiz: Correlation

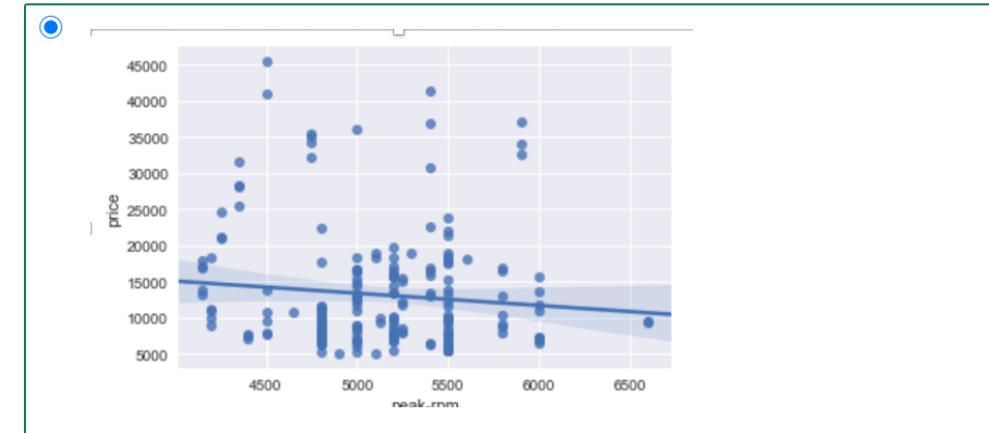
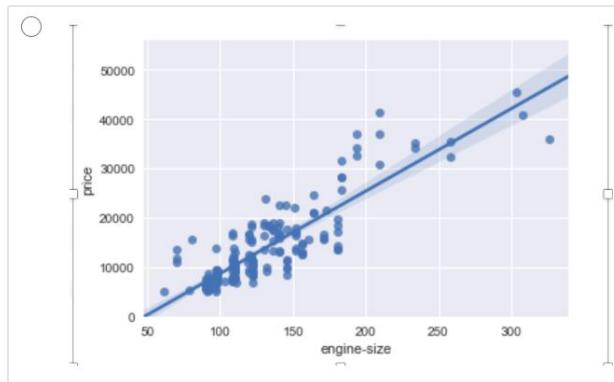
Bookmark this page

### Question 1

1/1 point (ungraded)

Select the scatter plot with weak correlation:

Select the scatter plot with weak correlation:



#### Answer

Correct: Correct

You have used 1 of 1 attempt

Show answer

✓ Correct (1/1 point)

## Correlation Statistics

In this video, we'll introduce you to various correlations statistical methods. One way to measure the strength of the correlation between continuous numerical variable is by using a method called Pearson correlation. Pearson correlation method will give you two values: the correlation coefficient and the P-value.

So how do we interpret these values? For the correlation coefficient,

- a value close to 1 implies a large positive correlation, while a
- value close to negative 1 implies a large negative correlation, and a value close to zero implies no correlation between the variables.

Next, the P-value will tell us how certain we are about the correlation that we calculated.

- For the P-value, a value less than .001 gives us a strong certainty about the correlation coefficient that we calculated.
- A value between .001 and .05 gives us moderate certainty.
- A value between .05 and .1 will give us a weak certainty.
- And a P-value larger than .1 will give us no certainty of correlation at all. We can say that there is a strong correlation when the correlation coefficient is close to 1 or negative 1, and the P-value is less than .001.

The following plot shows data with different correlation values. In this example, we want to look at the correlation between the variable's horsepower and car price. See how easy you can calculate the Pearson correlation using the SI/PI stats package? We can see that the correlation coefficient is approximately .8, and this is close to 1.

So there is a strong positive correlation. We can also see that the P-value is very small, much smaller than .001. And so we can conclude that we are certain about the strong positive correlation. Taking all variables into account, we can now create a heatmap that indicates the correlation between each of the variables with one another.

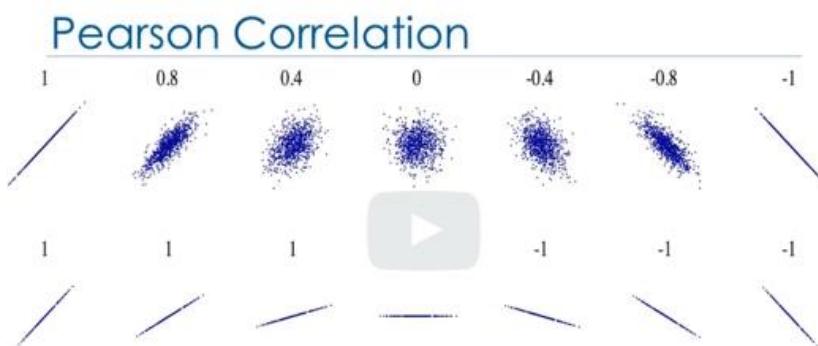
The color scheme indicates the Pearson correlation coefficient, indicating the strength of the correlation between two variables. We can see a diagonal line with a dark red color, indicating that all the values on this diagonal are highly correlated. This makes sense because when you look closer, the values on the diagonal are the correlation of all variables with themselves, which will be always 1.

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

# Correlation - Statistics

## 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



[https://en.wikipedia.org/wiki/Correlation\\_and\\_dependence](https://en.wikipedia.org/wiki/Correlation_and_dependence)

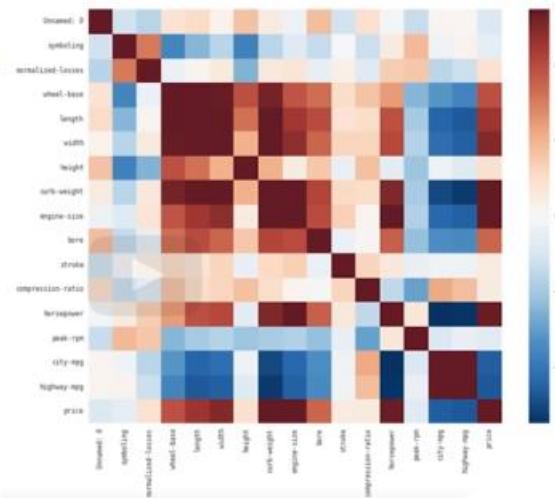
## Pearson Correlation- Example

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

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

We can see that the **correlation coefficient is approximately .8**, and this is close to **1**. So there is a **strong positive correlation**. We can also see that the P-value is very small, much smaller than **.001**. And so **we can conclude that we are certain about the strong positive correlation**.

## Correlation-Heatmap



This makes sense because when you look closer, the values on the diagonal are the correlation of all variables with themselves, which will be always 1. We can see a diagonal line with a dark red color, indicating that all the values on this diagonal are highly correlated.



## Chi-square Test for Independence

We recently learnt about the Pearson correlation test for two continuous variables. In some cases, our data may contain categorical variables - to test for the relationship between two categorical variables. The **chi-square** tests how one categorical variable influences the other categorical variable using the distribution of the frequencies within each group.

We will use the cars dataset. Assuming we want to test the relationship between **fuel-type** and **aspiration**. These are **two categorical variables** and do not have any continuous points to hold them to. It is either the car is a standard car with diesel fuel, a standard car with gas fuel, a turbo car with diesel fuel, or a turbo car with gas fuel. We will like to know if the fuel type is associated (or related) to the aspiration of the car. Before we do let's go through some important points:

The **Chi-square** tests a null hypothesis that the variables are independent. The test compares the observed data to the values that the model expects if the data was distributed in different categories by chance. Anytime the observed data doesn't fit within the model of the expected values, the probability that the variables are dependent becomes stronger, thus proving the null hypothesis incorrect.

The Chi-square does not tell you the type of relationship that exists between both variables only that a relationship exists.

Now let us test if there is an association between fuel-type and aspiration.

First, we will find the observed values of cars in each category. This can be done by using the crosstab in the pandas library.

	<b>Standard</b>	<b>Turbo</b>	<b>Total</b>
<b>diesel</b>	7	13	20
<b>gas</b>	161	24	185
<b>Total</b>	168	37	205

We will then calculate the expected values, i.e. the values that we will expect if each car fell into the categories randomly. We do that by using the following formula:

$$\frac{\text{RowTotal} * \text{ColumnTotal}}{\text{GrandTotal}}$$

Using the formula above, expected values for each group will be as follows:

- Standard car with diesel fuel =  $(20 * 168)/205$
- Standard car with gas fuel =  $(185 * 168)/205$
- Turbo car with diesel fuel =  $(20 * 37)/205$  and
- Turbo car with gas fuel =  $(185 * 37)/205$

We can also look at the expected values in a cross tabular form:

	<b>Standard</b>	<b>Turbo</b>	<b>Total</b>
<b>diesel</b>	16.39	3.61	20
<b>gas</b>	151.61	33.39	185
<b>Total</b>	168	37	205

We can see what the model expects and we can also see that the sub-totals and totals are equal to that of the observed values. We will now perform the Chi-square test. The formula is as follows:

$$\chi^2 = \sum_{k=1}^n \frac{(O_i - E_i)^2}{E_i}$$

This will produce a Chi-square  $\chi^2$  value, in this case is **29.6**. We will use the chi-square table and find the corresponding p-value which is done by finding the degree of freedom **(row-1)\*(column-1) = 1** and then the corresponding p-value to 29.6, using the chi-square table, we will see that the p-value is very close to 0. This means that there is an **association** between fuel-type and aspiration.

You can do this in python using the `scipy.stats` package, first we create a cross tab: `cont_table = pd.crosstab(df['fuel-type'], df['aspiration'])` then we use the `chi2_contingency` in the `scipy.stats` library: `scipy.stats.chi2_contingency(cont_table, correction = True)`

This will print out the chi-square statistic, the p-value, the degree of freedom, and the expected values.

## Lesson Summary



In this lesson, you have learned how to:

- **Describe Exploratory Data Analysis:** By summarizing the main characteristics of the data and extracting valuable insights.
- **Compute basic descriptive statistics:** Calculate the mean, median, and mode using python and use it as a basis in understanding the distribution of the data.
- **Create data groups:** How and why you put continuous data in groups and how to visualize them.
- **Define correlation as the linear association between two numerical variables:** Use Pearson correlation as a measure of the correlation between two continuous variables
- **Define the association between two categorical variables:** Understand how to find the association of two variables using the Chi-square test for association and how to interpret them.

## 7.3.Exploratory Data Analysis

Seongjoo Brenden Song edited this page on Nov 6, 2021 · 2 revisions

---

### Learning Objectives

- Implement descriptive statistics
  - Demonstrate the basics of grouping
  - Describe data correlation processes
  - Describe why and how to apply the Chi-Squared test
- 

- [Exploratory Data Analysis](#)
- [Lab3: Exploratory Data Analysis](#)

## 7.3.1.Exploratory Data Analysis

Seongjoo Brenden Song edited this page on Nov 6, 2021 · [2 revisions](#)

### 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?"

### 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()**`

### Question

What happens if the method `describe` is applied to a dataframe with NaN values

- ~~an error will occur~~
- ~~all the statistics calculated using NaN values will also be NaN~~
- ~~NaN values will be excluded~~

*Correct*

### Descriptive Statistics - value\_counts()

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

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

```
drive_wheels_counts.rename(columns={'dirve-wheels':'value_counts'}, inplace=True)
```

### 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

## GroupBy in Python

### 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()
```

## Question

How would you use the `groupby` function to find the average "price" of each car based on "body-style" ?

- `**df[['price', 'body-style']].groupby(['body-style'],as_index= False).mean()**`
- `~~df.groupby(['price" ],as_index= False).mean()~~`
- `~~mean(df.groupby(['price', 'body-style'],as_index= False))~~`

*Correct*

## 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')`

## Heatmap

- Plot target variable over multiple variables

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

## Question

Select the appropriate description of a pivot table:

- A pivot table has one variable displayed along the columns and the other variable displayed along the rows.
- A pivot table contains statistical information for each column

*Correct*

## 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)
- 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, )
```

### Correlation - Negative Linear Relationship

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

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

## Correlation - Statistics

### 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

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

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

### Correlation - Heatmap

## Association between two categorical variables: Chi-Square

### Categorical variables

- Use the Chi-square Test for Association (denoted as  $\chi^2$ )
- The test is intended to test how likely it is that an observed distribution is due to chance.
-

## Chi-square Test for Association

- The Chi-square tests a null hypothesis that the variables are independent.
- The Chi-square does not tell you the type of relationship that exists between both variables; but only that relationship exists.

## Categorical variables

$$\chi^2 = \sum_{i=1}^n \frac{(O_i - E_i)^2}{E_i}$$

```
scipy.stats.chi2_contingency(cont_table, correction=True)
```

## Lesson Summary

In this lesson, you have learned how to:

**Describe Exploratory Data Analysis:** By summarizing the main characteristics of the data and extracting valuable insights.

**Compute basic descriptive statistics:** Calculate the mean, median, and mode using python and use it as a basis in understanding the distribution of the data.

**Create data groups:** How and why you put continuous data in groups and how to visualize them.

**Define correlation as the linear association between two numerical variables:** Use Pearson correlation as a measure of the correlation between two continuous variables

**Define the association between two categorical variables:** Understand how to find the association of two variables using the Chi-square test for association and how to interpret them.

7.3.1



## Data Analysis with Python

Estimated time needed: **30** minutes

### Objectives

After completing this lab you will be able to:

- Explore features or characteristics to predict price of car

### Table of Contents

1. Import Data from Module
2. Analyzing Individual Feature Patterns using Visualization
3. Descriptive Statistical Analysis
4. Basics of Grouping
5. Correlation and Causation
6. ANOVA

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

## 1. Import Data from Module 2

### Setup

Import libraries:

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

Load the data and store it in dataframe df:

This dataset was hosted on IBM Cloud object. Click [HERE](#) for free storage.

```
In [2]: path='https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/automobileE  
df = pd.read_csv(path)  
df.head()
```

Out[2]:

	symboling	normalized-losses	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	length	...	compression-ratio	horsepower	peak-rpm	city-mpg	highway-mpg	price	city-L/100km
0	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	...	9.0	111.0	5000.0	21	27	13495.0	11.190476
1	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	...	9.0	111.0	5000.0	21	27	16500.0	11.190476
2	1	122	alfa-romero	std	two	hatchback	rwd	front	94.5	0.822681	...	9.0	154.0	5000.0	19	26	16500.0	12.368421
3	2	164	audi	std	four	sedan	fwd	front	99.8	0.848630	...	10.0	102.0	5500.0	24	30	13950.0	9.791667
4	2	164	audi	std	four	sedan	4wd	front	99.4	0.848630	...	8.0	115.0	5500.0	18	22	17450.0	13.055556

5 rows × 29 columns

horsepower-binned diesel gas

Medium	0	1

## 2. Analyzing Individual Feature Patterns Using Visualization

To install Seaborn we use pip, the Python package manager.

```
In [3]: %%capture
! pip install seaborn
```

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

```
In [4]: import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

### How to choose the right visualization method?

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

```
In [5]: # List the data types for each column
print(df.dtypes)
```

```
symboling          int64
normalized-losses   int64
make                object
aspiration         object
num-of-doors        object
body-style          object
drive-wheels        object
engine-location     object
wheel-base          float64
length              float64
width               float64
height              float64
curb-weight         int64
engine-type         object
num-of-cylinders    object
engine-size          int64
fuel-system          object
bore                 float64
stroke               float64
compression-ratio    float64
horsepower           float64
peak-rpm             float64
city-mpg              int64
highway-mpg            int64
price                float64
city-L/100km          float64
horsepower-binned     object
diesel               int64
gas                  int64
dtype: object
```

## Question #1:

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

```
In [7]: # Write your code below and press Shift+Enter to execute
print(df['peak-rpm'].dtypes)
```

float64

Click here for the solution `python float64`

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

```
In [8]: df.corr()
```

Out[8]:

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore	stroke	compression-ratio	horsepower	peak-rpm	city-mpg	
symboling	1.000000	0.466264	-0.535987	-0.365404	-0.242423	-0.550160	-0.233118	-0.110581	-0.140019	-0.008245	-0.182196	0.075819	0.279740	-0.035527	
normalized-losses	0.466264	1.000000	-0.056661	0.019424	0.086802	-0.373737	0.099404	0.112360	-0.029862	0.055563	-0.114713	0.217299	0.239543	-0.225016	
wheel-base	-0.535987	-0.056661	1.000000	0.876024	0.814507	0.590742	0.782097	0.572027	0.493244	0.158502	0.250313	0.371147	-0.360305	-0.470606	
length	-0.365404	0.019424	0.876024	1.000000	0.857170	0.492063	0.880665	0.685025	0.608971	0.124139	0.159733	0.579821	-0.285970	-0.665192	
width	-0.242423	0.086802	0.814507	0.857170	1.000000	0.306002	0.866201	0.729436	0.544885	0.188829	0.189867	0.615077	-0.245800	-0.633531	
height	-0.550160	-0.373737	0.590742	0.492063	0.306002	1.000000	0.307581	0.074694	0.180449	-0.062704	0.259737	-0.087027	-0.309974	-0.049800	
curb-weight	-0.233118	0.099404	0.782097	0.880665	0.866201	0.307581	1.000000	0.849072	0.644060	0.167562	0.156433	0.757976	-0.279361	-0.749543	
engine-size	-0.110581	0.112360	0.572027	0.685025	0.729436	0.074694	0.849072	1.000000	0.572609	0.209523	0.028889	0.822676	-0.256733	-0.650546	
bore	-0.140019	-0.029862	0.493244	0.608971	0.544885	0.180449	0.644060	0.572609	1.000000	-0.055390	0.001263	0.566936	-0.267392	-0.582027	
stroke	-0.008245	0.055563	0.158502	0.124139	0.188829	-0.062704	0.167562	0.209523	-0.055390	1.000000	0.187923	0.098462	-0.065713	-0.034696	
compression-ratio	-0.182196	-0.114713	0.250313	0.159733	0.189867	0.259737	0.156433	0.028889	0.001263	0.187923	1.000000	-0.214514	-0.435780	0.331425	
horsepower	0.075819	0.217299	0.371147	0.579821	0.615077	-0.087027	0.757976	0.822676	0.566936	0.098462	-0.214514	1.000000	0.107885	-0.822214	
peak-rpm	0.279740	0.239543	-0.360305	-0.285970	-0.245800	-0.309974	-0.279361	-0.256733	-0.267392	-0.065713	-0.435780	0.107885	1.000000	-0.115413	
city-mpg	-0.035527	-0.225016	-0.470606	-0.665192	-0.633531	-0.049800	-0.749543	-0.650546	-0.582027	-0.034696	0.331425	-0.822214	-0.115413	1.000000	
highway-mpg	0.036233	-0.181877	-0.543304	-0.698142	-0.680635	-0.104812	-0.794889	-0.679571	-0.591309	-0.035201	0.268465	-0.804575	-0.058598	0.972044	
price	-0.082391	0.133999	0.584642	0.690628	0.751265	0.135486	0.834415	0.872335	0.543155	0.082310	0.071107	0.809575	-0.101616	-0.686571	
city-L/100km	0.066171	0.238567	0.476153	0.657373	0.673363	0.003811	0.785353	0.745059	0.554610	0.037300	-0.299372	0.889488	0.115830	-0.949713	
diesel	-0.196735	-0.101546	0.307237	0.211187	0.244356	0.281578	0.221046	0.070779	0.054458	0.241303	0.985231	-0.169053	-0.475812	0.265676	
gas	0.196735	0.101546	-0.307237	-0.211187	-0.244356	-0.281578	-0.221046	-0.070779	-0.054458	-0.241303	-0.985231	0.169053	0.475812	-0.265676	
highway-mpg	0.036233	-0.082391	0.066171	-0.196735	0.196735										
price	-0.181877	0.133999	0.238567	-0.101546	0.101546										
city-L/100km	-0.543304	0.584642	0.476153	0.307237	-0.307237										
diesel	-0.698142	0.690628	0.657373	0.211187	-0.211187										
gas	-0.680635	0.751265	0.673363	0.244356	-0.244356										
highway-mpg	-0.104812	0.135486	0.003811	0.281578	-0.281578										
price	-0.794889	0.834415	0.785353	0.221046	-0.221046										
city-L/100km	-0.679571	0.872335	0.745059	0.070779	-0.070779										
diesel	-0.591309	0.543155	0.554610	0.054458	-0.054458										
gas	-0.035201	0.082310	0.037300	0.241303	-0.241303										
highway-mpg	0.268465	0.071107	-0.299372	0.985231	-0.985231										
price	-0.804575	0.809575	0.889488	-0.169053	0.169053										
city-L/100km	-0.058598	-0.101616	0.115830	-0.475812	0.475812										
diesel	0.972044	-0.686571	-0.949713	0.265676	-0.265676										
gas	1.000000	-0.704692	-0.930028	0.198690	-0.198690										
highway-mpg	-0.704692	1.000000	0.789898	0.110326	-0.110326										
price	-0.930028	0.789898	1.000000	-0.241282	0.241282										
city-L/100km	0.198690	0.110326	-0.241282	1.000000	-1.000000										
diesel	-0.198690	-0.110326	0.241282	-1.000000	1.000000										
gas	0.196735	0.101546	-0.307237	-0.211187	-0.244356										

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

## Question #2:

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

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

```
In [9]: # Write your code below and press Shift+Enter to execute  
df[['bore', 'stroke', 'compression-ratio', 'horsepower']].corr()
```

```
Out[9]:
```

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

Click here for the solution [```python df\[\['bore', 'stroke', 'compression-ratio', 'horsepower'\]\].corr\(\)```](#)

## Continuous Numerical Variables:

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

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

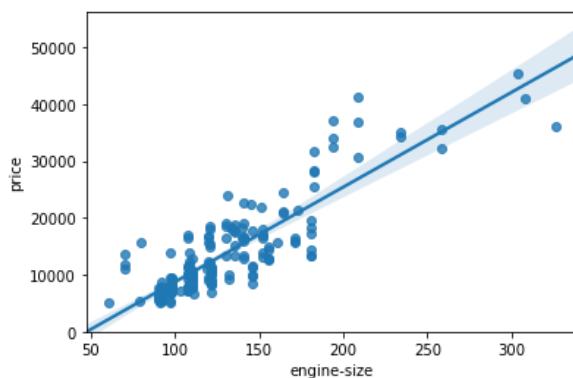
Let's see several examples of different linear relationships:

### Positive Linear Relationship</h4>

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

```
In [14]: # Engine size as potential predictor variable of price  
sns.regplot(x="engine-size", y="price", data=df)  
plt.ylim(0,)
```

```
Out[14]: (0.0, 56152.43776707113)
```



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

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

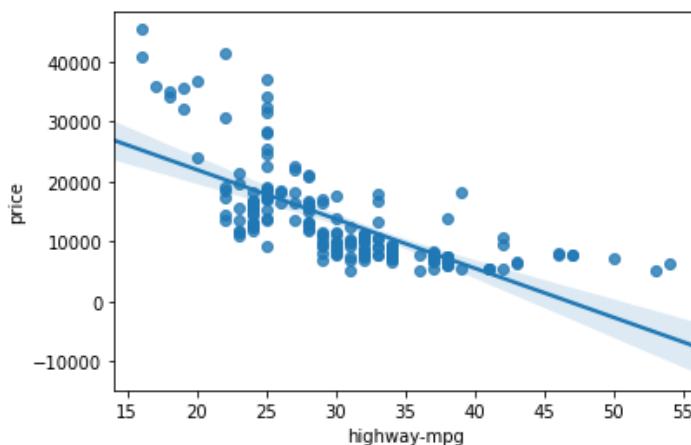
```
In [11]: df[["engine-size", "price"]].corr()
```

```
Out[11]:      engine-size    price
engine-size    1.000000  0.872335
               price     0.872335  1.000000
```

Highway mpg is a potential predictor variable of price. Let's find the scatterplot of "highway-mpg" and "price".

```
In [12]: sns.regplot(x="highway-mpg", y="price", data=df)
```

```
Out[12]: <AxesSubplot:xlabel='highway-mpg', ylabel='price'>
```



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

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

```
In [15]: df[['highway-mpg', 'price']].corr()
```

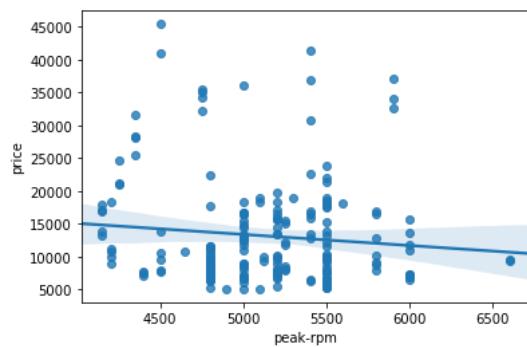
```
Out[15]:      highway-mpg    price
highway-mpg    1.000000 -0.704692
               price     -0.704692  1.000000
```

## Weak Linear Relationship

Let's see if "peak-rpm" is a predictor variable of "price".

```
In [16]: sns.regplot(x="peak-rpm", y="price", data=df)
```

```
Out[16]: <AxesSubplot:xlabel='peak-rpm', ylabel='price'>
```



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

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

```
In [17]: df[['peak-rpm', 'price']].corr()
```

```
Out[17]:
```

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

## Question 3 a):

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

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

```
In [18]: # Write your code below and press Shift+Enter to execute  
df[['stroke', 'price']].corr()
```

```
Out[18]:
```

	stroke	price
stroke	1.000000	0.08231
price	0.08231	1.000000

Click here for the solution ``python #The correlation is 0.0823, the non-diagonal elements of the table.  
df[["stroke", "price"]].corr() ``

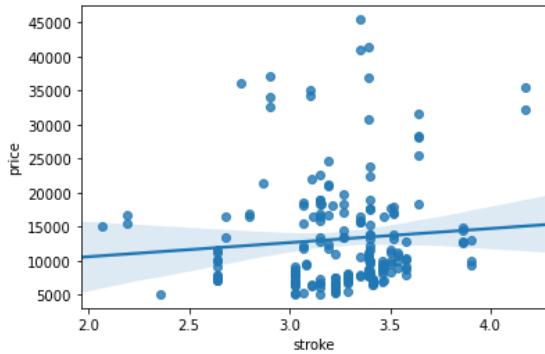
## Question 3 b):

Given the correlation results between "price" and "stroke", do you expect a linear relationship?

Verify your results using the function "regplot()".

```
In [19]: # Write your code below and press Shift+Enter to execute  
sns.regplot(x='stroke', y='price', data=df)
```

```
Out[19]: <AxesSubplot:xlabel='stroke', ylabel='price'>
```



Click here for the solution ``python #There is a weak correlation between the variable 'stroke' and 'price.' as such regression will not work well. We can see this using "regplot" to demonstrate this. #Code: sns.regplot(x="stroke", y="price", data=df)``

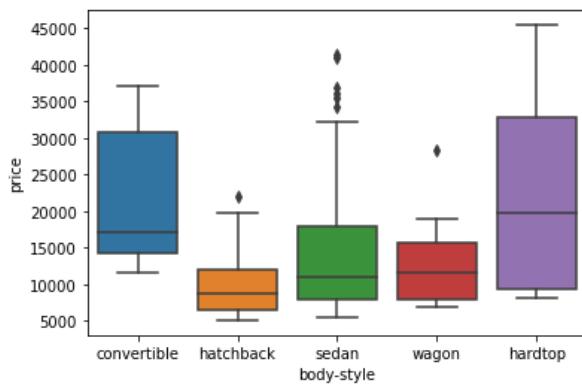
## Categorical Variables

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

Let's look at the relationship between "body-style" and "price".

```
In [20]: sns.boxplot(x="body-style", y="price", data=df)
```

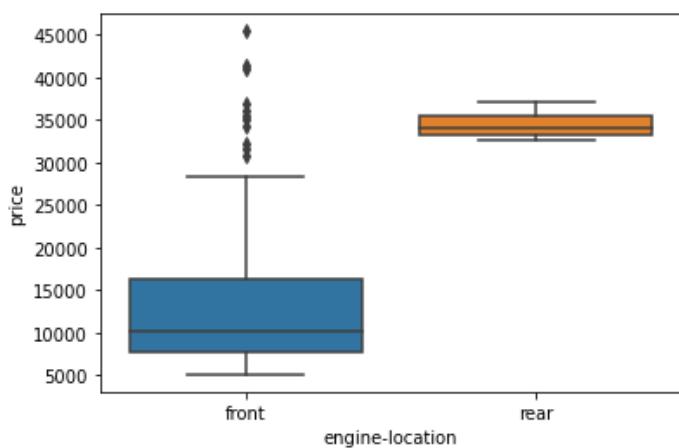
```
Out[20]: <AxesSubplot:xlabel='body-style', ylabel='price'>
```



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

```
In [21]: sns.boxplot(x="engine-location", y="price", data=df)
```

```
Out[21]: <AxesSubplot:xlabel='engine-location', ylabel='price'>
```

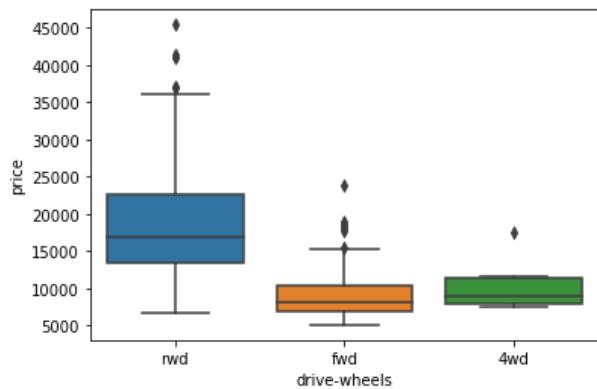


Here we see that the distribution of price between these two engine-location categories, front and rear, are distinct enough to take engine-location as a potential good predictor of price.

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

```
In [22]: # drive-wheels  
sns.boxplot(x="drive-wheels", y="price", data=df)
```

```
Out[22]: <AxesSubplot:xlabel='drive-wheels', ylabel='price'>
```



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

### 3. Descriptive Statistical Analysis

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

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

This will show:

- the count of that variable

- the mean
- the standard deviation (std)
- the minimum value
- the IQR (Interquartile Range: 25%, 50% and 75%)
- the maximum value

We can apply the method "describe" as follows:

In [23]:	df.describe()														
Out[23]:	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore	stroke	compression-ratio	horsepower	peak-rpm	city-mpg	
	<b>count</b>		201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	197.000000	201.000000	201.000000	201.000000	201.000000	
	<b>mean</b>		0.840796	122.000000	98.797015	0.837102	0.915126	53.766667	2555.666667	126.875622	3.330692	3.256904	10.164279	103.405534	5117.665368
	<b>std</b>		1.254802	31.99625	6.066366	0.059213	0.029187	2.447822	517.296727	41.546834	0.268072	0.319256	4.004965	37.365700	478.113805
	<b>min</b>		-2.000000	65.000000	86.600000	0.678039	0.837500	47.800000	1488.000000	61.000000	2.540000	2.070000	7.000000	48.000000	4150.000000
	<b>25%</b>		0.000000	101.000000	94.500000	0.801538	0.890278	52.000000	2169.000000	98.000000	3.150000	3.110000	8.600000	70.000000	4800.000000
	<b>50%</b>		1.000000	122.000000	97.000000	0.832292	0.909722	54.100000	2414.000000	120.000000	3.310000	3.290000	9.000000	95.000000	5125.369458
	<b>75%</b>		2.000000	137.000000	102.400000	0.881788	0.925000	55.500000	2926.000000	141.000000	3.580000	3.410000	9.400000	116.000000	5500.000000
	<b>max</b>		3.000000	256.000000	120.900000	1.000000	1.000000	59.800000	4066.000000	326.000000	3.940000	4.170000	23.000000	262.000000	6600.000000
highway-mpg	price	city-L/100km	diesel	gas											
201.000000	201.000000	201.000000	201.000000	201.000000											
30.686567	13207.129353	9.944145	0.099502	0.900498											
6.815150	7947.066342	2.534599	0.300083	0.300083											
16.000000	5118.000000	4.795918	0.000000	0.000000											
25.000000	7775.000000	7.833333	0.000000	1.000000											
30.000000	10295.000000	9.791667	0.000000	1.000000											
34.000000	16500.000000	12.368421	0.000000	1.000000											
54.000000	45400.000000	18.076923	1.000000	1.000000											

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

In [24]:	df.describe(include=['object'])										
Out[24]:	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	engine-type	num-of-cylinders	fuel-system	horsepower-binned	
	<b>count</b>	201	201	201	201	201	201	201	201	201	200
	<b>unique</b>	22	2	2	5	3	2	6	7	8	3
	<b>top</b>	toyota	std	four	sedan	fwd	front	ohc	four	mpfi	Low
	<b>freq</b>	32	165	115	94	118	198	145	157	92	115

## Value Counts

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

```
In [25]: df['drive-wheels'].value_counts()
```

```
Out[25]: fwd    118  
rwd     75  
4wd      8  
Name: drive-wheels, dtype: int64
```

We can convert the series to a dataframe as follows:

```
In [26]: df['drive-wheels'].value_counts().to_frame()
```

```
Out[26]:   drive-wheels  
fwd        118  
rwd         75  
4wd          8
```

Let's repeat the above steps but save the results to the dataframe "drive\_wheels\_counts" and rename the column 'drive-wheels' to 'value\_counts'.

```
In [27]: drive_wheels_counts = df['drive-wheels'].value_counts().to_frame()  
drive_wheels_counts.rename(columns={'drive-wheels': 'value_counts'}, inplace=True)  
drive_wheels_counts
```

```
Out[27]:   value_counts  
fwd        118  
rwd         75  
4wd          8
```

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

```
In [28]: drive_wheels_counts.index.name = 'drive-wheels'  
drive_wheels_counts
```

```
Out[28]:   value_counts  
drive-wheels  
fwd        118  
rwd         75  
4wd          8
```

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

```
In [30]: # engine-location as variable  
engine_loc_counts = df['engine-location'].value_counts().to_frame()  
engine_loc_counts.rename(columns={'engine-location': 'value_counts'}, inplace=True)  
engine_loc_counts.index.name = 'engine-location'  
engine_loc_counts
```

```
Out[30]:   value_counts  
engine-location  
front       198  
rear         3
```

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

## 4. Basics of Grouping

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

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

```
In [31]: df['drive-wheels'].unique()  
Out[31]: array(['rwd', 'fwd', '4wd'], dtype=object)
```

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

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

```
In [32]: df_group_one = df[['drive-wheels','body-style','price']]
```

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

```
In [33]: # grouping results  
df_group_one = df_group_one.groupby(['drive-wheels'],as_index=False).mean()  
df_group_one  
  
Out[33]:   drive-wheels      price  
0          4wd    10241.000000  
1          fwd    9244.779661  
2          rwd    19757.613333
```

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

You can also group by multiple variables. For example, let's group by both 'drive-wheels' and 'body-style'. This groups the dataframe by the unique combination of 'drive-wheels' and 'body-style'. We can store the results in the variable 'grouped\_test1'.

In [35]:

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

Out[35]:

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

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

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

In [36]:

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

Out[36]:

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

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

In [37]:

```
grouped_pivot = grouped_pivot.fillna(0) #fill missing values with 0
grouped_pivot
```

Out[37]:

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

## Question 4:

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

```
In [38]: # Write your code below and press Shift+Enter to execute
df_gptest2 = df[['body-style', 'price']]
grouped_test_bodystyle = df_gptest2.groupby(['body-style'], as_index=False).mean()
grouped_test_bodystyle
```

```
Out[38]:   body-style      price
0   convertible  21890.500000
1     hardtop    22208.500000
2   hatchback    9957.441176
3     sedan     14459.755319
4     wagon     12371.960000
```

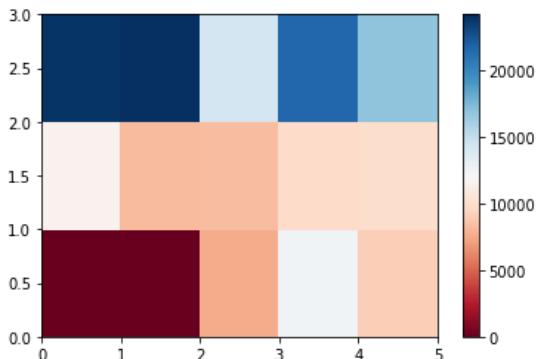
Click here for the solution ``python # grouping results df\_gptest2 = df[['body-style','price']] grouped\_test\_bodystyle = df\_gptest2.groupby(['body-style'],as\_index= False).mean() grouped\_test\_bodystyle ``  
If you did not import "pyplot", let's do it again.

```
In [39]: import matplotlib.pyplot as plt
%matplotlib inline
```

Variables: Drive Wheels and Body Style vs. Price

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

```
In [40]: #use the grouped results
plt.pcolor(grouped_pivot, cmap='RdBu')
plt.colorbar()
plt.show()
```



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

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

```
In [43]: fig, ax = plt.subplots()
im = ax.pcolor(grouped_pivot, cmap='RdBu')

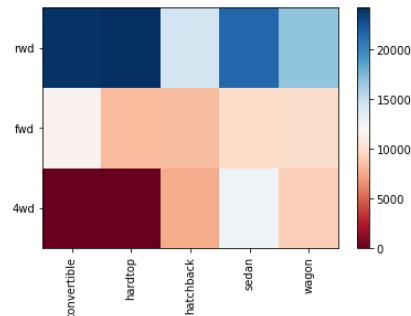
#label names
row_labels = grouped_pivot.columns.levels[1]
col_labels = grouped_pivot.index

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

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

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

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



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

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

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

## 5. Correlation and Causation

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

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

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

### Pearson Correlation

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

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

- **1**: Perfect positive linear correlation.
- **0**: No linear correlation, the two variables most likely do not affect each other.
- **-1**: Perfect negative linear correlation.

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

```
In [44]: df.corr()
```

Out[44]:

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore	stroke	compression-ratio	horsepower	peak-rpm	city-mpg
<b>symboling</b>	1.000000	0.466264	-0.535987	-0.365404	-0.242423	-0.550160	-0.233118	-0.110581	-0.140019	-0.008245	-0.182196	0.075819	0.279740	-0.035527
<b>normalized-losses</b>	0.466264	1.000000	-0.056661	0.019424	0.086802	-0.373737	0.099404	0.112360	-0.029862	0.055563	-0.114713	0.217299	0.239543	-0.225016
<b>wheel-base</b>	-0.535987	-0.056661	1.000000	0.876024	0.814507	0.590742	0.782097	0.572027	0.493244	0.158502	0.250313	0.371147	-0.360305	-0.470606
<b>length</b>	-0.365404	0.019424	0.876024	1.000000	0.857170	0.492063	0.880665	0.685025	0.608971	0.124139	0.159733	0.579821	-0.285970	-0.665192
<b>width</b>	-0.242423	0.086802	0.814507	0.857170	1.000000	0.306002	0.866201	0.729436	0.544885	0.188829	0.189867	0.615077	-0.245800	-0.633531
<b>height</b>	-0.550160	-0.373737	0.590742	0.492063	0.306002	1.000000	0.307581	0.074694	0.180449	-0.062704	0.259737	-0.087027	-0.309974	-0.049800
<b>curb-weight</b>	-0.233118	0.099404	0.782097	0.880665	0.866201	0.307581	1.000000	0.849072	0.644060	0.167562	0.156433	0.757976	-0.279361	-0.749543
<b>engine-size</b>	-0.110581	0.112360	0.572027	0.685025	0.729436	0.074694	0.849072	1.000000	0.572609	0.209523	0.028889	0.822676	-0.256733	-0.650546
<b>bore</b>	-0.140019	-0.029862	0.493244	0.608971	0.544885	0.180449	0.644060	0.572609	1.000000	-0.055390	0.001263	0.566936	-0.267392	-0.582027
<b>stroke</b>	-0.008245	0.055563	0.158502	0.124139	0.188829	-0.062704	0.167562	0.209523	-0.055390	1.000000	0.187923	0.098462	-0.065713	-0.034696
<b>compression-ratio</b>	-0.182196	-0.114713	0.250313	0.159733	0.189867	0.259737	0.156433	0.028889	0.001263	0.187923	1.000000	-0.214514	-0.435780	0.331425
<b>horsepower</b>	0.075819	0.217299	0.371147	0.579821	0.615077	-0.087027	0.757976	0.822676	0.566936	0.098462	-0.214514	1.000000	0.107885	-0.822214
<b>peak-rpm</b>	0.279740	0.239543	-0.360305	-0.285970	-0.245800	-0.309974	-0.279361	-0.256733	-0.267392	-0.065713	-0.435780	0.107885	1.000000	-0.115413
<b>city-mpg</b>	-0.035527	-0.225016	-0.470606	-0.665192	-0.633531	-0.049800	-0.749543	-0.650546	-0.582027	-0.034696	0.331425	-0.822214	-0.115413	1.000000
<b>highway-mpg</b>	0.036233	-0.181877	-0.543304	-0.698142	-0.680635	-0.104812	-0.794889	-0.679571	-0.591309	-0.035201	0.268465	-0.804575	-0.058598	0.972044
<b>price</b>	-0.082391	0.133999	0.584642	0.690628	0.751265	0.135486	0.834415	0.872335	0.543155	0.082310	0.071107	0.809575	-0.101616	-0.686571
<b>city-L/100km</b>	0.066171	0.238567	0.476153	0.657373	0.673363	0.003811	0.785353	0.745059	0.554610	0.037300	-0.299372	0.889488	0.115830	-0.949713
<b>diesel</b>	-0.196735	-0.101546	0.307237	0.211187	0.244356	0.281578	0.221046	0.070779	0.054458	0.241303	0.985231	-0.169053	-0.475812	0.265676
<b>gas</b>	0.196735	0.101546	-0.307237	-0.211187	-0.244356	-0.281578	-0.221046	-0.070779	-0.054458	-0.241303	-0.985231	0.169053	0.475812	-0.265676

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

highway-mpg	price	city-L/100km	diesel	gas
0.036233	-0.082391	0.066171	-0.196735	0.196735
-0.181877	0.133999	0.238567	-0.101546	0.101546
-0.543304	0.584642	0.476153	0.307237	-0.307237
-0.698142	0.690628	0.657373	0.211187	-0.211187
-0.680635	0.751265	0.673363	0.244356	-0.244356
-0.104812	0.135486	0.003811	0.281578	-0.281578
-0.794889	0.834415	0.785353	0.221046	-0.221046
-0.679571	0.872335	0.745059	0.070779	-0.070779
-0.591309	0.543155	0.554610	0.054458	-0.054458
-0.035201	0.082310	0.037300	0.241303	-0.241303
0.268465	0.071107	-0.299372	0.985231	-0.985231
-0.804575	0.809575	0.889488	-0.169053	0.169053
-0.058598	-0.101616	0.115830	-0.475812	0.475812
0.972044	-0.686571	-0.949713	0.265676	-0.265676
1.000000	-0.704692	-0.930028	0.198690	-0.198690
-0.704692	1.000000	0.789898	0.110326	-0.110326
-0.930028	0.789898	1.000000	-0.241282	0.241282
0.198690	0.110326	-0.241282	1.000000	-1.000000
-0.198690	-0.110326	0.241282	-1.000000	1.000000

## P-value

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

By convention, when the

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

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

```
In [45]: from scipy import stats
```

## Wheel-Base vs. Price

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

```
In [46]: pearson_coef, p_value = stats.pearsonr(df['wheel-base'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, "with a P-value of P =", p_value)
```

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

**Conclusion:**

Since the p-value is \$

## Horsepower vs. Price

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

```
In [47]: pearson_coef, p_value = stats.pearsonr(df['horsepower'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P = ", p_value)

The Pearson Correlation Coefficient is 0.8095745670036562 with a P-value of P = 6.369057428259195e-48
```

Conclusion:

Since the p-value is \$

## Length vs. Price

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

```
In [48]: pearson_coef, p_value = stats.pearsonr(df['length'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P = ", p_value)

The Pearson Correlation Coefficient is 0.6906283804483639 with a P-value of P = 8.016477466159328e-30
```

Conclusion:

Since the p-value is \$

## Width vs. Price

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

```
In [49]: pearson_coef, p_value = stats.pearsonr(df['width'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P = ", p_value)

The Pearson Correlation Coefficient is 0.7512653440522675 with a P-value of P = 9.200335510481123e-38
```

Conclusion:

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

## Curb-Weight vs. Price

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

```
In [50]: pearson_coef, p_value = stats.pearsonr(df['curb-weight'], df['price'])
print( "The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P = ", p_value)

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

## Conclusion:

Since the p-value is \$

## Engine-Size vs. Price

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

```
In [51]: pearson_coef, p_value = stats.pearsonr(df['engine-size'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P =", p_value)

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

## Conclusion:

Since the p-value is \$

## Bore vs. Price

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

```
In [52]: pearson_coef, p_value = stats.pearsonr(df['bore'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P = ", p_value )
```

The Pearson Correlation Coefficient is 0.5431553832626603 with a P-value of P = 8.049189483935261e-17

## Conclusion:

Since the p-value is \$

We can relate the process for each 'city-mpg' and 'highway-mpg':

## City-mpg vs. Price

```
In [53]: pearson_coef, p_value = stats.pearsonr(df['city-mpg'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P =", p_value)

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

## Conclusion:

Since the p-value is \$

## Highway-mpg vs. Price

```
In [54]: pearson_coef, p_value = stats.pearsonr(df['highway-mpg'], df['price'])
print( "The Pearson Correlation Coefficient is", pearson_coef, " with a P-value of P = ", p_value )

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

## Conclusion:

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

## 6. ANOVA

### ANOVA: Analysis of Variance

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

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

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

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

### Drive Wheels

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

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

```
In [57]: grouped_test2=df_gptest[['drive-wheels', 'price']].groupby(['drive-wheels'])  
grouped_test2.head(2)
```

```
Out[57]:    drive-wheels      price  
0            rwd    13495.0  
1            rwd    16500.0  
3            fwd    13950.0  
4            fwd    17450.0  
5            fwd    15250.0  
136           fwd    7603.0
```

```
In [58]: df_gptest
```

```
Out[58]:   drive-wheels body-style    price
0           rwd  convertible  13495.0
1           rwd  convertible  16500.0
2           rwd  hatchback   16500.0
3           fwd    sedan     13950.0
4           4wd    sedan     17450.0
...
196          rwd    sedan     16845.0
197          rwd    sedan     19045.0
198          rwd    sedan     21485.0
199          rwd    sedan     22470.0
200          rwd    sedan     22625.0
```

201 rows × 3 columns

We can obtain the values of the method group using the method "get\_group".

```
In [59]: grouped_test2.get_group('4wd')['price']
```

```
Out[59]: 4      17450.0
136     7603.0
140     9233.0
141     11259.0
144     8013.0
145     11694.0
150     7898.0
151     8778.0
Name: price, dtype: float64
```

We can use the function 'f\_oneway' in the module 'stats' to obtain the **F-test score** and **P-value**.

```
In [60]:
```

```
# ANOVA
f_val, p_val = stats.f_oneway(grouped_test2.get_group('fwd')['price'], grouped_test2.get_group('rwd')['price'], grouped_test2.get_group('4wd')['price'])

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

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

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

Let's examine them separately.

### fwd and rwd

```
In [61]: f_val, p_val = stats.f_oneway(grouped_test2.get_group('fwd')['price'], grouped_test2.get_group('rwd')['price'])

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

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

Let's examine the other groups.

## 4wd and rwd

```
In [62]: f_val, p_val = stats.f_oneway(grouped_test2.get_group('4wd')['price'], grouped_test2.get_group('rwd')['price'])

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

ANOVA results: F= 8.580681368924756 , P = 0.004411492211225333

## 4wd and fwd

```
In [63]: f_val, p_val = stats.f_oneway(grouped_test2.get_group('4wd')['price'], grouped_test2.get_group('fwd')['price'])

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

ANOVA results: F= 0.665465750252303 , P = 0.41620116697845666

## Conclusion: Important Variables

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

Continuous numerical variables:

- Length
- Width
- Curb-weight
- Engine-size
- Horsepower
- City-mpg
- Highway-mpg
- Wheel-base
- Bore

Categorical variables:

- Drive-wheels

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

## Graded Quiz: Exploratory Data Analysis

Bookmarked

Graded Quiz due Feb 14, 2022 09:32 +08

### Question 1

1/1 point (graded)

What task does the method value\_counts perform?

- Returns the first five columns of a dataframe
- Returns summary statistics
- Returns counts of unique values



#### Answer

Correct: Correct

---

### Question 2

1/1 point (graded)

What is the largest possible element resulting in the operation df.corr()?

- 1000
- 100
- 1



#### Answer

Correct: Correct

Submit

You have used 1 of 2 attempts

---

✓ Correct (1/1 point)

### Question 3

1/1 point (graded)

If the Pearson Correlation between two variables is zero, then ...

The two variables have zero mean

The two variables are not correlated



#### Answer

Correct: Correct

Submit

You have used 1 of 1 attempt

---

✓ Correct (1/1 point)

### Question 4

1/1 point (graded)

Consider the dataframe df; what method displays the first five rows of a dataframe?

df.describe()

df.tail()

df.head()



#### Answer

Correct: Correct

Submit

You have used 1 of 2 attempts

---

✓ Correct (1/1 point)

---

## Question 5

1/1 point (graded)

What is the Pearson Correlation between variables X and Y, if  $X=-Y$ ?

-1

0

1

**Answer**

Correct: Correct

You have used 1 of 2 attempts

---

✓ Correct (1/1 point)

## Module 4 – Model Development

- Model Development
- Linear Regression and Multiple Linear Regression
- Practice Quiz: Linear Regression and Multiple Linear Regression
- Model Evaluation using Visualization
- Practice Quiz: Model Evaluation using Visualization
- Polynomial Regression and Pipelines
- Practice Quiz: Polynomial Regression and Pipelines
- Measures for In-Sample Evaluation
- Practice Quiz: Measures for In-Sample Evaluation
- Prediction and Decision Making
- Lesson Summary
- Hands-on Lab: Model Development
- Graded Quiz: Model Development

Module 4 - Model Development / Module Introduction & Learning Objectives

JS

≡

### Module Introduction & Learning Objectives

 Bookmark this page

#### Module Introduction

In this week's module, you will learn how to define the explanatory variable and the response variable and understand the differences between the simple linear regression and multiple linear regression models. You will learn how to evaluate a model using visualization and learn about polynomial regression and pipelines. You will also learn how to interpret and use the R-squared and the mean square error measures to perform in-sample evaluations to numerically evaluate our model. And lastly, you will learn about prediction and decision making when determining if our model is correct.

#### Learning Objectives

- Describe how to process linear regression in Python
- Apply model evaluation using visualization in Python
- Apply polynomial regression techniques to Python
- Evaluate a data model by using visualization
- Describe the use of R-squared and MSE for in-sample evaluation
- Apply prediction and decision making to Python model creation

## Model Development

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

In this module, you'll learn about simple and multiple linear regression, model evaluation using visualization, polynomial regression and pipelines, R-squared and MSE for in-sample evaluation, prediction and decision making, and how you can determine a fair value for a used car.

A model or estimator can be thought of as a mathematical equation used to predict the value given one or more other values. Relating one or more independent variables or features to dependent variables.

For example, you input a car model's highway miles per gallon as the independent variable or feature, the output of the model or dependent variable is the price. Usually, the more relevant data you have,

the more accurate your model is. For example, you input multiple independent variables or features to your model. Therefore, your model may predict a more accurate price for the car. To understand why more data is important, consider the following situation.

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

If your models independent variables or features do not include color, your model will predict the same price for cars that may sell for much less. In addition to getting more data, you can try different types of models.

In this course, you will learn about simple linear regression, multiple linear regression and polynomial regression.

## Model Development

### Learning Objectives

In this module you will learn about:

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

- Question

- How can you determine a fair value for a used car?

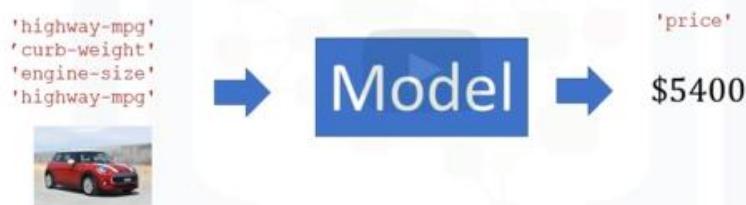
## Model Development

- 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.



## Model Development

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

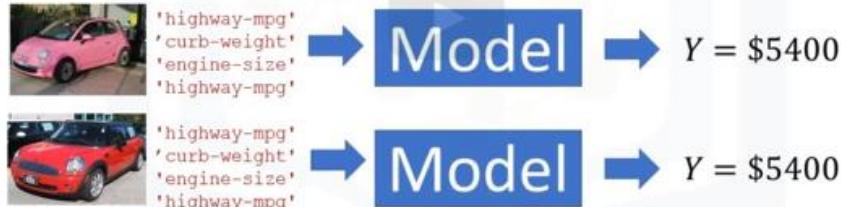


## Model Development

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

1. you have two almost identical cars
2. Pink cars sell for significantly less

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



## Model Development

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

In this course you will learn about:

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

## Linear Regression and Multiple Linear Regression

In this video, we'll be talking about simple linear regression and multiple linear regression. Linear regression will refer to one independent variable to make a prediction. Multiple linear regression will refer to multiple independent variables to make a prediction.

Simple linear regression or SLR is a method to help us understand the relationship between two variables.

The predictor independent variable  $x$  and the target dependent variable  $y$ . We would like to come up with a linear relationship between the variables shown here. The parameter  $b_0$  is the intercept. The parameter  $b_1$  is the slope. When we fit or train the model, we will come up with these parameters. This step requires lots of math, so we will not focus on this part.

Let's clarify the prediction step. It's hard to figure out how much a car costs, but the highway miles per gallon is in the owner's manual. If we assume there is a linear relationship between these variables, we can use this relationship to formulate a model to determine the price of the car. If the highway miles per gallon is 20, we can input this value into the model to obtain a prediction of \$22,000.

In order to determine the line, we take data points from our data set marked in red here. We then use these training points to fit our model. The results of the training points are the parameters. We usually store the data points into data frame or numpy arrays. The value we would like to predict is called the target that we store in the array  $y$ . We store the independent variable in the data frame or array  $x$ . Each sample corresponds to a different row in each data frame or array. In many cases, many factors influence how much people pay for a car.

For example, make or how old the car is. In this model, this uncertainty is taken into account by assuming a small random value is added to the point on the line. This is called noise. The figure on the left shows the distribution of the noise. The vertical axis shows the value added and the horizontal axis illustrates the probability that the value will be added. Usually a small positive value is added or a small negative value.

Sometimes, large values are added. But for the most part, the values added are near zero. We can summarize the process like this. We have a set of training points. We use these training points to fit or train the model and get parameters. We then use these parameters in the model. We now have a model. We use the hat on the  $y$  to denote the model is an estimate. We can use this model to predict values that we haven't seen.

For example, we have no car with 20 highway miles per gallon. We can use our model to make a prediction for the price of this car. But don't forget our model is not always correct. We can see this by comparing the predicted value to the actual value.

We have a sample for 10 highway miles per gallon but the predictive value does not match the actual value. If the linear assumption is correct, this error is due to the noise. But there can be other reasons. To fit the model in Python, first we import linear model from sklearn then create a linear regression object using the constructor. We define the predictor variable and target variable then use the method fit to fit the model and find the parameters  $b_0$  and  $b_1$ . The input are the features and the targets. We can obtain prediction using the method predict.

The output is an array. The array has the same number of samples as the input  $x$ . The intercept  $b_0$  is an attribute of the object  $lm$ . The slope  $b_1$  is also an attribute of the object  $lm$ . The relationship between price and highway miles per gallon is given by this equation in bold, price equals 38,423.31 minus 821.73 times highway miles per gallon, like the equation we discussed before.

Multiple linear regression is used to explain the relationship between one continuous target  $y$  variable and two or more predictor  $x$  variables. If we have for example 4 predictor variables then  $b_0$  intercept  $x$  equal zero  $b_1$  the coefficient or parameter of  $x_1$ ,  $b_2$  the coefficient of parameter  $x_2$  and so on.

If there are only two variables then we can visualize the values. Consider the following function: The variables  $x_1$  and  $x_2$  can be visualized on a 2D plane. Let's do an example on the next slide. The table contains different values of the predictor variables  $x_1$  and  $x_2$ . The position of each point is placed on the 2D plane color-coded accordingly. Each value of the predictor variables  $x_1$  and  $x_2$  will be mapped to a new value  $y$ ,  $\hat{y}$ . The new values of  $y$   $\hat{y}$  are mapped in the vertical direction with height proportional to the value that  $\hat{y}$  takes.

We can fit the multiple linear regression as follows. We can extract the four predictor variables and store them in the variable  $z$  then train the model as before using the method `train` with the features or dependent variables and the targets colon. We can also obtain a prediction using the method `predict`. In this case, the input is an array or data frame with four columns. The number of rows corresponds to the number of samples.

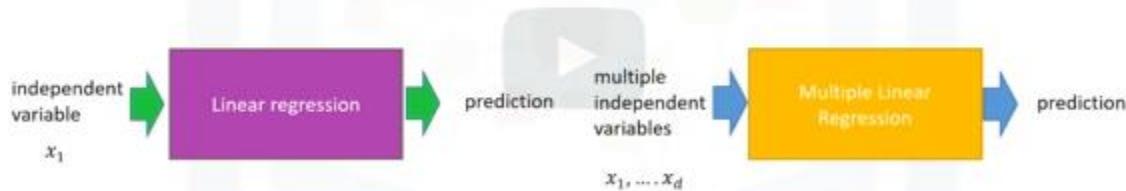
The output is an array with the same number of elements as number of samples. The intercept is an attribute of the object and the coefficients are also attributes. It is helpful to visualize the equation, replacing the dependent variable names with actual names.

This is identical to the form we discussed earlier.

## Linear Regression and Multiple Linear Regression

### 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

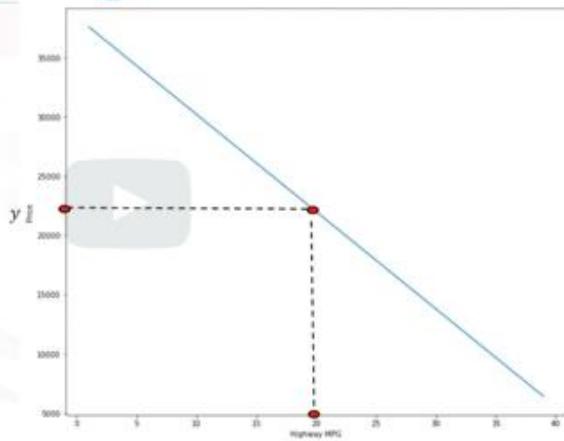
1. The predictor (independent) variable -  $x$
2. The target (dependent) variable -  $y$

$$y = b_0 + b_1 x$$

- $b_0$ : the intercept
- $b_1$ : the slope

### Simple Linear Regression: Prediction

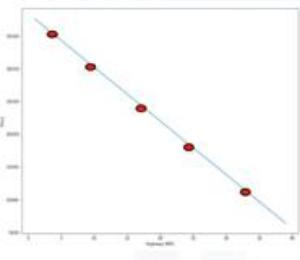
$$\begin{aligned}y &= 38423 - 821x \\&= 38423 - 821(20) \\&= 22\,003\end{aligned}$$



It's hard to figure out how much a car costs, but the highway miles per gallon is in the owner's manual. If we assume there is a linear relationship between these variables, we can use this relationship to formulate a model to determine the price of the car. If the highway miles per gallon is 20, we can input this value into the model to obtain a prediction of \$22,000. In order to determine the line, we take data points from our data set marked in red here. We

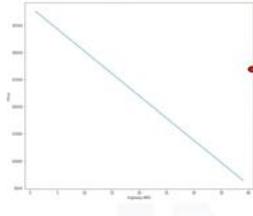
then use these training points to fit our model. The results of the training points are the parameters.

### Simple Linear Regression: Fit



Fit

### Simple Linear Regression: Fit

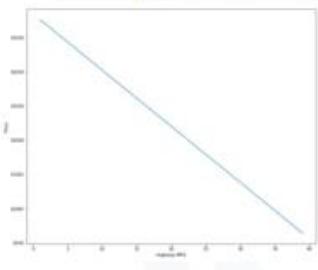


Fit

In order to determine the line, we take data points from our data set marked in red here. We then use these training points to fit our model. The results of the training points are the parameters.

1.

## Simple Linear Regression: Fit



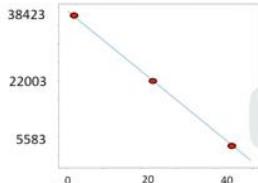
Fit

 $(b_0, b_1)$ 

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

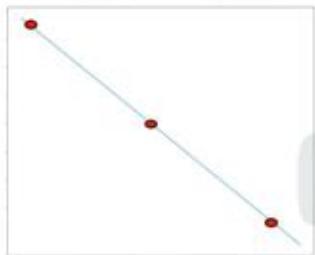
2.

## Simple Linear Regression: Fit



$$X = \begin{bmatrix} \quad \\ \quad \\ \quad \end{bmatrix} \quad Y = \begin{bmatrix} \quad \\ \quad \\ \quad \end{bmatrix}$$

## Simple Linear Regression: Fit

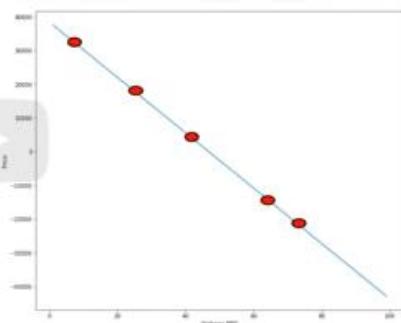
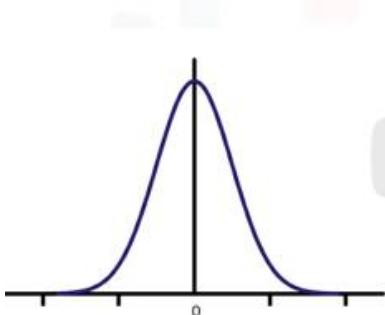


$$X = \begin{bmatrix} 0 \\ 20 \\ 40 \end{bmatrix} \quad Y = \begin{bmatrix} 38423 \\ 22003 \\ 5583 \end{bmatrix}$$

We usually store the data points into data frame or numpy arrays. The value we would like to predict is called the target that we store in the array  $y$ . We store the independent variable in the data frame or array  $x$ . Each sample corresponds to a different row in each data frame or array. In many cases, many factors influence how much people pay for a car.

3.

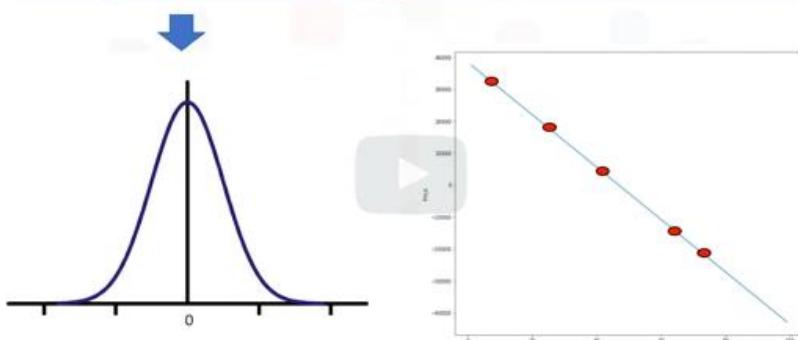
## Simple Linear Regression: Fit



In many cases, many factors influence how much people pay for a car. For example, make or how old the car is. In this model, this uncertainty is taken into account by assuming a small random value is added to the point on the line. This is called noise.

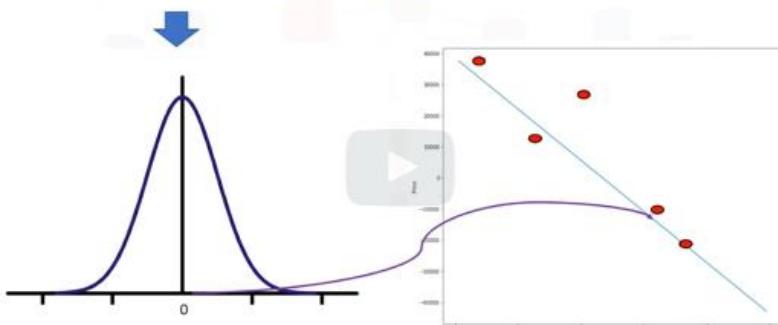
4.

## Simple Linear Regression: Fit



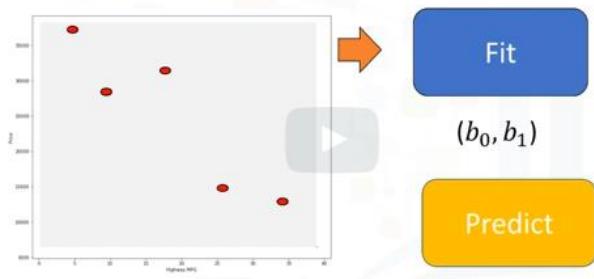
5.

## Simple Linear Regression: Fit



6.

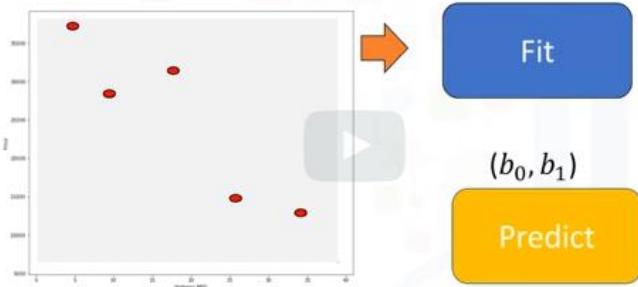
## Simple Linear Regression



We can summarize the process like this. We have a set of training points. We use these training points to fit or train the model and get parameters.

7.

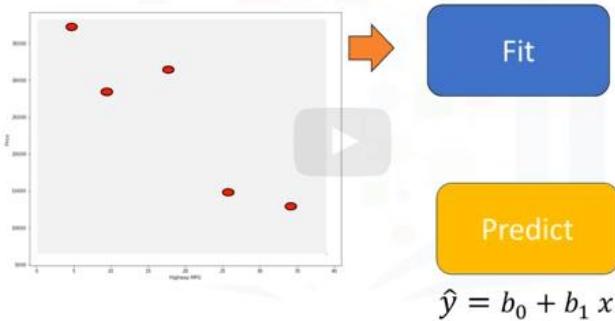
## Simple Linear Regression



We then use these parameters in the model. We now have a model.

8.

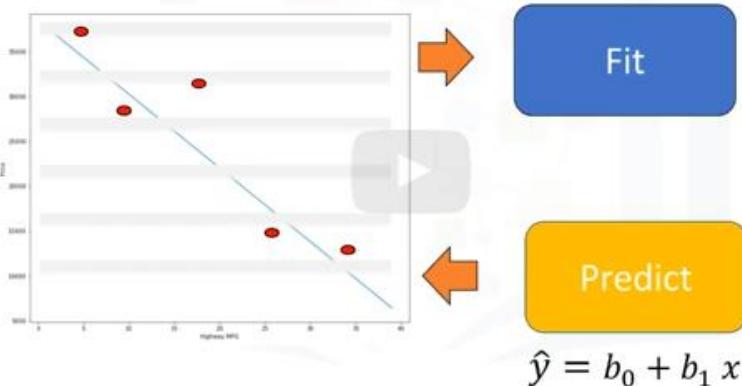
## Simple Linear Regression



We then use these parameters in the model. We now have a model. We use the hat on the y to denote the model is an estimate.

9.

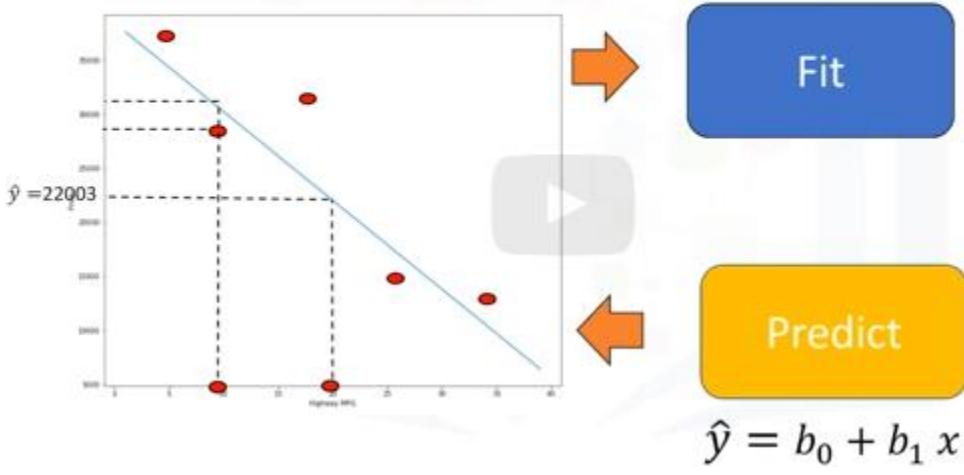
## Simple Linear Regression



We can use this model to predict values that we haven't seen. We can use our model to make a prediction for the price of this car.

But don't forget our model is not always correct. We can see this by comparing the predicted value to the actual value.

## Simple Linear Regression



We can use our model to make a prediction for the price of this car. But don't forget our model is not always correct. We can see this by comparing the predicted value to the actual value. We have a sample for 10 highway miles per gallon but the predictive value does not match the actual value. If the linear assumption is correct, this error is due to the noise. But there can be other reasons.

## 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 fine 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

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

The intercept  $b_0$  is an attribute of the object lm. The slope  $b_1$  is also an attribute of the object lm. The relationship between price and highway miles per gallon is given by this equation in bold, price equals 38,423.31 minus 821.73 times highway miles per gallon, like the equation we discussed before.

# 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)

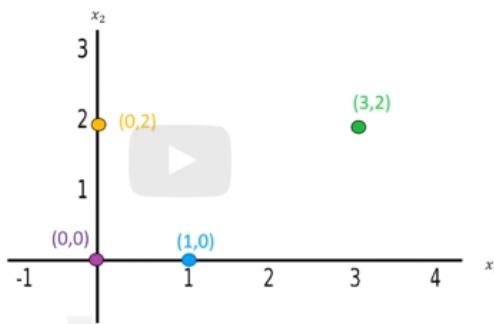
$$\hat{Y} = b_0 + b_1x_1 + b_2x_2 + b_3x_3 + b_4x_4$$

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

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

- The variables  $x_1$  and  $x_2$  can be visualized on a 2D plane, lets do an example on the next slide

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

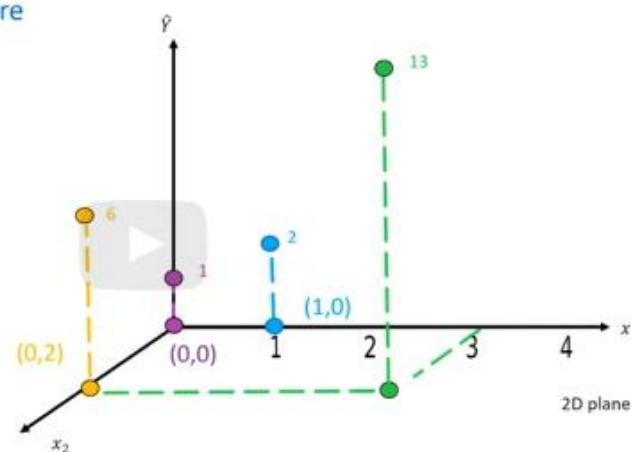


The table contains different values of the predictor variables  $x\_1$  and  $x\_2$ . The position of each point is placed on the 2D plane color-coded accordingly.

- This is shown below where

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

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



Each value of the predictor variables  $x\_1$  and  $x\_2$  will be mapped to a new value  $y$ ,  $\hat{y}$ . The new values of  $y$ ,  $\hat{y}$  are mapped in the vertical direction with height proportional to the value that  $\hat{y}$  takes.

## Fitting a Multiple Linear Model Estimator

1. We can extract the four 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)
```

The diagram illustrates a linear transformation. On the left, there is a 4x4 matrix labeled  $X$  with columns  $x_1, x_2, x_3, x_4$ . To its right is a blue arrow pointing to the right. To the right of the arrow is a vertical column labeled  $\hat{Y}$  with elements 2, :, and 3. This represents the mapping  $\hat{Y} = X \cdot \text{coefficients}$ .

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

$\rightarrow$

$\hat{Y}$
2
:
3

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

## 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

We can fit the multiple linear regression as follows. We can extract the four predictor variables and store them in the variable z then train the model as before using the method train with the features or dependent variables and the targets colon. We can also obtain a prediction using the method predict. In this case, the input is an array or data frame with four columns. The number of

The intercept is an attribute of the object and the coefficients are also attributes. It is helpful to visualize the equation, replacing the dependent variable names with actual names.

## Practice Quiz: Linear Regression and Multiple Linear Regression

Bookmarked

### Question 1

1/1 point (ungraded)

Consider the following lines of code, what variable contains the predicted values

```
1 from sklearn.linear_model import LinearRegression
2 lm=LinearRegression()
3 X = df[['highway-mpg']]
4 Y = df['price']
5 lm.fit(X, Y)
6 Yhat=lm.predict(X)
7
8
```

Y

Yhat

X



#### Answer

Correct: Correct

You have used 2 of 2 attempts

✓ Correct (1/1 point)

### Question 2

1/1 point (ungraded)

consider the following equation:

$$y = b_0 + b_1 x$$

the variable y is?

the predictor or independent variable

the target or dependent variable

the intercept



#### Answer

Correct: Correct

You have used 2 of 2 attempts

✓ Correct (1/1 point)

## 7.4. Model Development

Seongjoo Brenden Song edited this page on Nov 6, 2021 · 3 revisions

---

### Learning Objectives

- Describe how to process linear regression in Python
  - Apply model evaluation using visualization in Python
  - Apply polynomial regression techniques to Python
  - Evaluate a data model by using visualization
  - Describe the use of R-squared and MSE for in-sample evaluation
  - Apply prediction and decision making to Python model creation
- 

- [Model Development](#)
- [Lab 4: Model Development](#)

## 7.4.1. Model Development

Seongjoo Brenden Song edited this page on Nov 6, 2021 · 3 revisions

- A model can be thought of as a mathematical equation used to predict a value one or more other values
- Relating one or more independent variables to dependent variables

Example:

```
independent variables of features ('highway-mpg': 55mpg)
→ MODEL
→ dependent variables ('predicted price': $5000)
```

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

```
('highway-mpg', 'curb-weight', 'engine-size')
→ MODEL
→ ('price': $5400)
```

- In addition to getting more data you can try different types of models.
- In this course you will learn about:
  - i. Simple Linear Regression
  - ii. Multiple Linear Regression
  - iii. Polynomial Regression

## Linear Regression and Multiple Linear Regression

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

### Simple Linear Regression (SLR)

1. The predictor (independent) variable - x

2. The target (dependent) variable - y

$$y = b_0 + b_1x$$

- $b_0$ : 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
```

- 
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 fine the parameters  $b_0$  and  $b_1$

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

- We can obtain a prediction

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

## 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:  
$$\text{Price} = 38423.31 - 821.73 * \text{highway-mpg} / \hat{Y} = b_0 + b_1 x$$

## Multiple Linear Regression (MLR)

This method is used to explain the relationship between:

- One continuous target (Y) variable

- Two or more predictor (X) variables

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

- $b_0$ : intercept ( $X=0$ )

- $b_1$ : the coefficient or parameter of  $x_1$

- $b_2$ : the coefficient or parameter of  $x_2$  and so on...

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

- The variables  $x_1$  and  $x_2$  can be visualized on a 2D plane

## 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']]
```

1. Then train the model as before:

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

1. We can also obtain a prediction

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

## MLR - Estimated Linear Model

1. Find the intercept ( $b_0$ )

```
lm.intercept_
-15678.742628061467
```

1. 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 =  $-15678.74 + (52.66) * \text{horsepower} + (4.70) * \text{curb-weight} + (81.96) * \text{engine-size} + (33.58) * \text{highway-mpg}$
- $\hat{Y} = b_0 + b_1x_1 + b_2x_2 + b_3x_3 + b_4x_4$

## Practice Quiz: Linear Regression and Multiple Linear Regression

---

TOTAL POINTS 2

### Question 1

consider the following lines of code, what variable contains the predicted values :

```
from sklearn.linear_model import LinearRegression
lm=LinearRegression()
X = df[['highway-mpg']]
Y = df['price']
lm.fit(X, Y)
Yhat=lm.predict(X)
```

- $Y$
- $X$
- $Yhat$

*Correct*

## Question 2

consider the following equation:

$$y = b_0 + b_1 x$$

what is the parameter  $b_0$  (b subscript 0)

- the predictor or independent variable
- the target or dependent variable
- the intercept
- the slope

*Correct*

## Model Evaluation using Visualization

---

### 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 shows us a combination of:

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

```
import seaborn as sns

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

## Residual Plot

- Look at the spread of the residuals:
  - Randomly spread out around x-axis then a linear model is appropriate
- Not randomly spread out around the x-axis
- Nonlinear model may be more appropriate
- Not randomly spread out around the x-axis
- Variance appears to change with x-axis

```
import seaborn as sns  
  
sns.residplot(df['highway_mpg'], df['price'])
```

## Distribution Plots

Compare the distribution plots:

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

## MLR - 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 and Pipelines

## Polynomial Regression

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

Curvilinear relationship:

By squaring or setting higher-order terms of the predictor variables

- Quadratic - 2nd order

- $\hat{Y} = b_0 + b_1x_1 + b_2(x_1)^2$

- Cubic - 3rd order

- $\hat{Y} = b_0 + b_1x_1 + b_2(x_1)^2 + b_3(x_1)^3$

- Higher order

- $\hat{Y} = b_0 + b_1X_1 + b_2(x_1)^2 + b_3(x_1)^3 + \dots$

Example:

1. Calculate Polynomial of 3rd order

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

1. Print out the model

```
print(p)
```

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

## Polynomial Regression with More than One Dimension

- There are also multi dimensional polynomial linear regression

$$\hat{Y} = b_0 + b_1X_1 + b_2X_2 + b_3X_1X_2 + b_4(X_1)^2 + b_5(X_2)^2 + \dots$$

- 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']])

**pr = polynomialFeatures(degree=2)**

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

## 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

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

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

- Pipeline constructor

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

- 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

- 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:

- In python

```
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)

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

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

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

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

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

- We get a value near one, this means the line is a good fit for the data.
- An Example of a line that does not fit the data well

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

- The ratio of the areas is close to one. In this case the R^2 is near zero. This line performs about the same as just using the average of the data points, therefore, this line did not perform well.
- Generally the values of the MSE are between 0 and 1.
- We can calculate the R^2 as follows

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

lm.fit(X, Y)

lm.score(X, Y)
```

```
0.496591188
```

- From the value that we get from this example, we can say that approximately 49.695% of the variation of price is explained by this simple linear model.
- Your R^2 value is usually between 0 and 1. If your R^2 is negative, it can be due to over fitting.

## Prediction and 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['price'])
```

- 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
- Using the numpy function arange to generate a sequence from 1 to 100

```
import numpy as np

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

- Can predict new values

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

## Visualization

- Simply visualizing your data with a regression

## Residual Plot

### Visualization - Multiple linear regression

#### Numerical measures for Evaluation

The figure shows an example of a mean square error of 3495.

This example has a mean square error of 3652.

The one has a mean square error of 12870.

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

r squared is another popular method to evaluate your model. It tells you how well your line fits into the model. r squared values range from zero to one. r squared tells us what percent of the variability in the dependent variable is accounted for by the regression on the independent variable. An r squared of 1 means that all movements of another dependent variable are completely explained by movements in the independent variables.

In this plot we see the target points in red and the predicted line in blue. An r squared of 0.9986 the model appears to be a good fit. That means that more than 99 of the variability of the predicted variable is explained by the independent variables.

This model has an r squared of 0.9226 there still is a strong linear relationship model is still a good fit.

An r squared of 0.806 of the data we can visually see that the values are scattered around the line. They are still close to the line and we can say that 80 percent of the variability of the predicted variable is explained by the independent variables.

An r squared 0.61 means that approximately 61 percent of the observed variation can be explained by the independent variables.

- An acceptable value for r squared depends on what field you are studying and what your use case is. Falcon Miller 1992 suggests that an acceptable r squared value should be at least 0.1.

## Comparing MLR and SLR

Does a lower Mean Square Error imply better fit?

- Not necessarily
1. Mean Square Error for a Multiple Linear Regression Model will be smaller than the Mean Square Error for a Simple Linear Regression Model, since the errors of the data will decrease when more variables are include in the model.
  2. Polynomial regression will also have a smaller Mean Square Error than the Linear Regression

## Lesson Summary

In this lesson, you have learned how to:

**Define the explanatory variable and the response variable:** Define the response variable ( $y$ ) as the focus of the experiment and the explanatory variable ( $x$ ) as a variable used to explain the change of the response variable. Understand the differences between Simple Linear Regression because it concerns the study of only one explanatory variable and Multiple Linear Regression because it concerns the study of two or more explanatory variables.

**Evaluate the model using Visualization:** By visually representing the errors of a variable using scatterplots and interpreting the results of the model.

**Identify alternative regression approaches:** Use a Polynomial Regression when the Linear regression does not capture the curvilinear relationship between variables and how to pick the optimal order to use in a model.

**Interpret the R-square and the Mean Square Error:** Interpret R-square ( $\times 100$ ) as the percentage of the variation in the response variable  $y$  that is explained by the variation in explanatory variable(s)  $x$ . The Mean Squared Error tells you how close a regression line is to a set of points. It does this by taking the average distances from the actual points to the predicted points and squaring them.

## Model Evaluation using Visualization

In this video, we'll look at Model Evaluation using Visualization. Regression plots are a good estimate of the relationship between two variables, the strength of the correlation, and the direction of the relationship (positive or negative).

The horizontal axis is the independent variable. The vertical axis is the dependent variable. Each point represents a different target point. The fitted line represents the predicted value. There are several ways to plot a regression plot. A simple way to use regplot from the seaborn library. First, "import seaborn." Then use the "regplot" function. The parameter x is the name of the column that contains the independent variable or feature.

The parameter y, contains the name of the column that contains the name of the dependent variable or target. The parameter data is the name of the dataframe. The result is given by the plot. The residual plot represents the error between the actual value. Examining the predicted value and actual value we see a difference.

We obtain that value by subtracting the predicted value, and the actual target value. We then plot that value on the vertical axis with the independent variable as the horizontal axis. Similarly, for the second sample, we repeat the process. Subtracting the target value from the predicted value. Then plotting the value accordingly. Looking at the plot gives us some insight into our data. We expect to see the results to have zero mean, distributed evenly around the x axis with similar variance. There is no curvature. This type of residual plot suggests a linear plot is appropriate. In this residual plot, there is a curvature. The values of the error change with x. For example, in the region, all the residual errors are positive. In this area, the residuals are negative. In the final location, the error is large. The residuals are not randomly separated. This suggests the linear assumption is incorrect. This plot suggests a nonlinear function. We will deal with this in the next section. In this plot, we see the variance of the residuals increases with x. Therefore, our model is incorrect. We can use seaborn to create a residual plot. First, "import seaborn." We use the "residplot" function. The first parameter is a series of dependent variable or feature. The second parameter is a series of dependent variable or target. We see in this case, the residuals have a curvature. A distribution plot counts the predicted value versus the actual value.

These plots are extremely useful for visualizing models with more than one independent variable or feature. Let's look at a simplified example. We examined the vertical axis. We then count and plot the number of predicted points that are approximately equal to one. We then, count and plot the number of predicted points that are approximately equal to two. We repeat the process. For predicted points, they are approximately equal to three. Then we repeat the process for the target values. In this case, all the target values are approximately equal to two.

The values of the targets and predicted values are continuous. A histogram is for discrete values. Therefore, pandas will convert them to a distribution. The vertical axis is scaled to make the area under the distribution equal to one. This is an example of using a distribution plot. The dependent variable or feature is price. The fitted values that result from the model are in blue. The actual values are in red. We see the predicted values for prices in the range from 40,000 to 50,000 are inaccurate. The prices in the region from 10,000 to 20,000 are much closer to the target value.

In this example, we use multiple features or independent variables. Comparing it to the plot on the last slide, we see predicted values are much closer to the target values. Here's the code to create a distribution plot. The actual values are used as a parameter. We wanted distribution instead of a histogram. So we want the hist parameters set to false. The color is red. The label is also included. The predicted values are included for the second plot. The rest of the parameters are set accordingly.

## Model Evaluation Using Visualization

### Regression Plot

Why use regression plot?

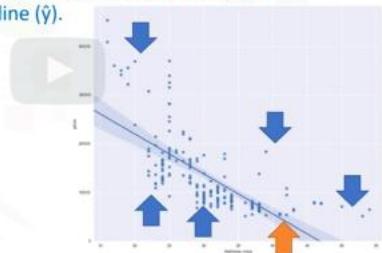
It gives us a good estimate of:

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 ( $\hat{y}$ )



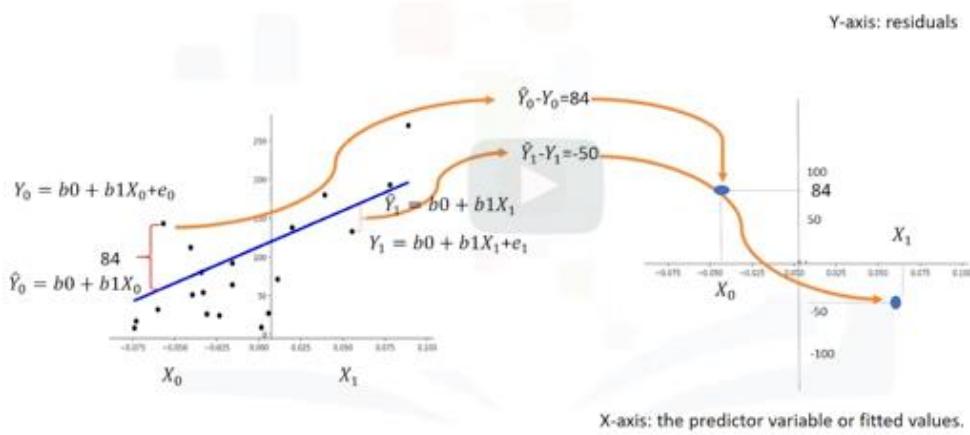
The horizontal axis is the independent variable. The vertical axis is the dependent variable. Each point represents a different target point. The fitted line represents the predicted value.

### Regression Plot

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

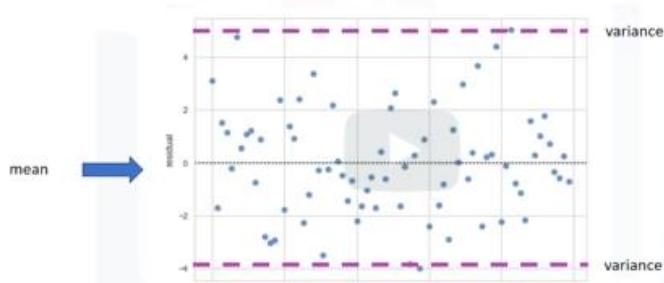
There are several ways to plot a regression plot. A simple way is to use `regplot` from the `seaborn` library. First, import `seaborn`. Then use the `"regplot"` function. The parameter `x` is the name of the column that contains the independent variable or feature. The parameter `y`, contains the name of the column that contains the name of the dependent variable or target. The parameter `data` is the name of the dataframe. The result is given by the plot.

## Residual Plot



The residual plot represents the error between the actual value. Examining the predicted value and actual value we see a difference. We obtain that value by subtracting the predicted value, and the actual target value. We then plot that value on the vertical axis with the independent variable as the horizontal axis. Similarly, for the second sample, we repeat the process. Subtracting the target value from the predicted value. Then plotting the value accordingly.

## Residual Plot



We expect to see the results to have zero mean, distributed evenly around the x axis with similar variance. There is no curvature. This type of residual plot suggests a linear plot is appropriate.

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

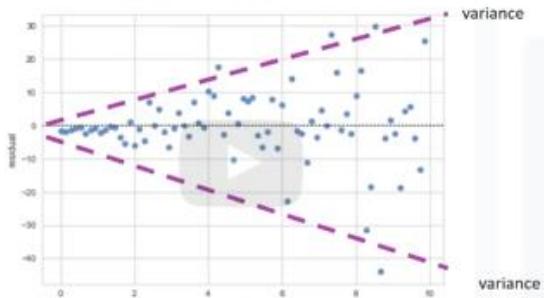
## Residual Plot



In this residual plot, there is a curvature. The values of the error change with x. For example, in the region, all the residual errors are positive. In this area, the residuals are negative. In the final location, the error is large. The residuals are not randomly separated. This suggests the linear assumption is incorrect. This plot suggests a nonlinear function.

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

## Residual Plot

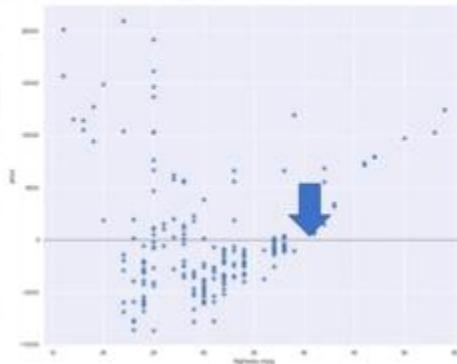


- Not randomly spread out around the x-axis
- Variance appears to change with x axis

In this plot, we see the variance of the residuals increases with x. Therefore, our model is incorrect. We can use seaborn to create a residual plot.

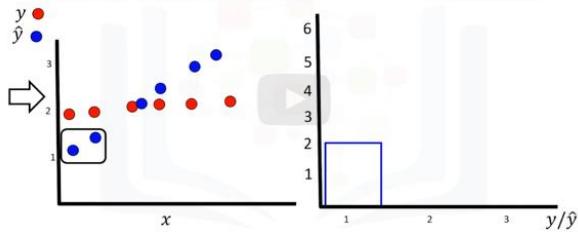
## Residual Plot

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

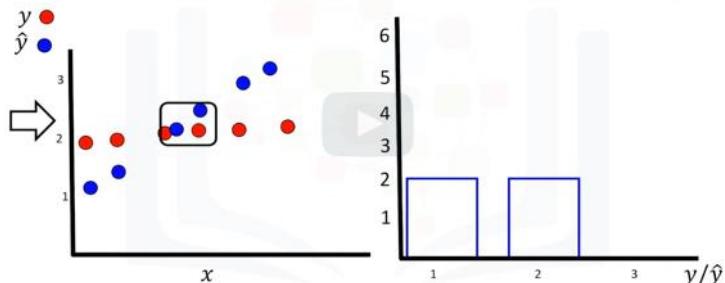


We can use **seaborn** to create a residual plot. First, "import seaborn." We use the "residplot" function. The first parameter is a series of dependent variable or feature. The second parameter is a series of dependent variable or target. We see in this case, the residuals have a curvature

## Distribution Plots



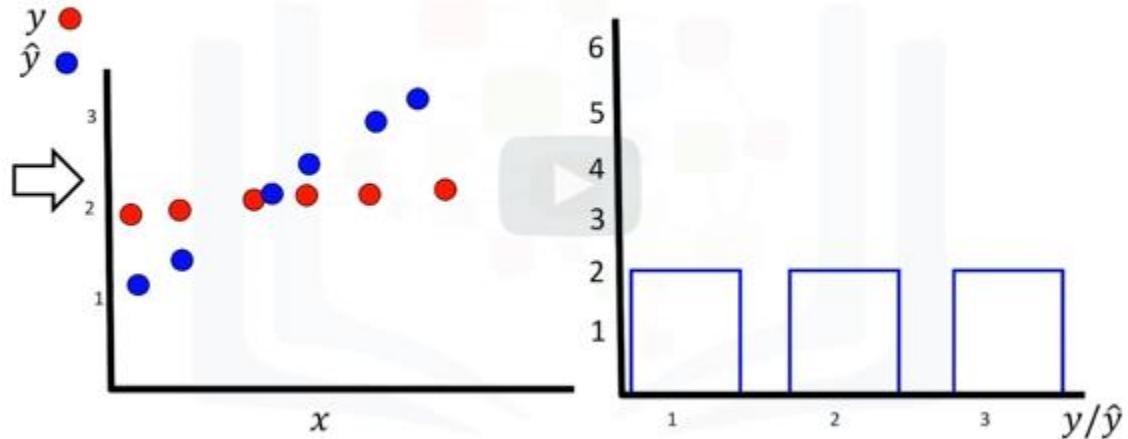
## Distribution Plots



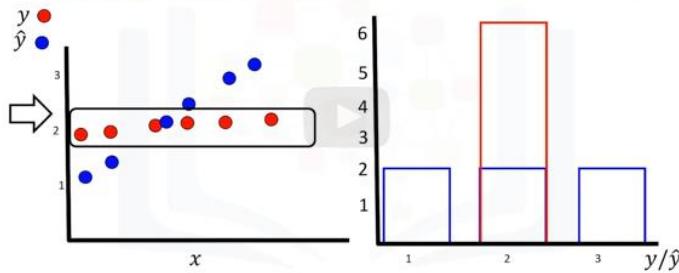
A distribution plot counts the predicted value versus the actual value. These plots are extremely useful for visualizing models with more than one independent variable or feature. Let's look at a simplified example. We examined the vertical axis. We then count and plot the number of predicted points that are approximately equal to one. We then, count and plot the number of predicted points that are approximately equal to two. We repeat the process. For

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

## Distribution Plots

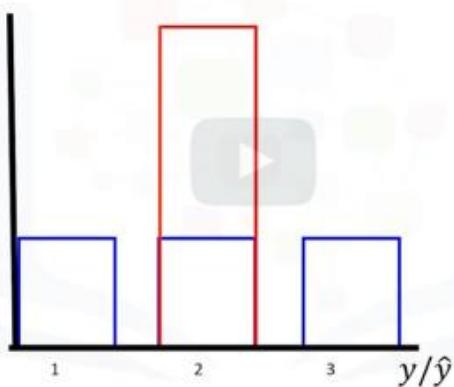


## Distribution Plots



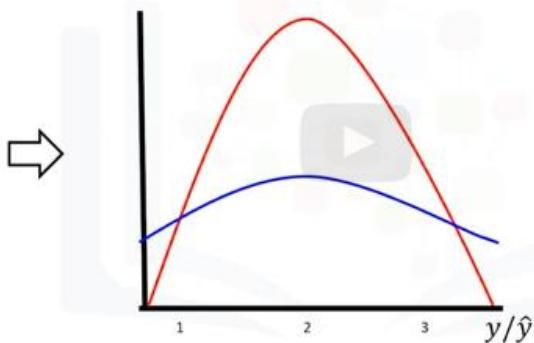
In this case, all the target values are approximately equal to two.

## Distribution Plots



The values of the targets and predicted values are continuous.

## Distribution Plots



A histogram is for discrete values.

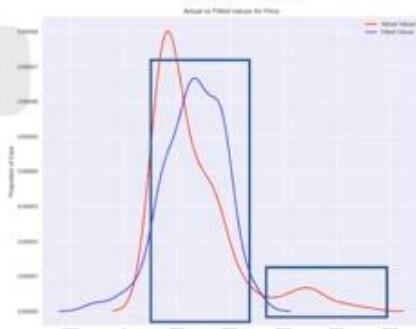
Therefore, pandas will convert them to a distribution.

The vertical axis is scaled to make the area under the distribution equal to one.

## Distribution Plots

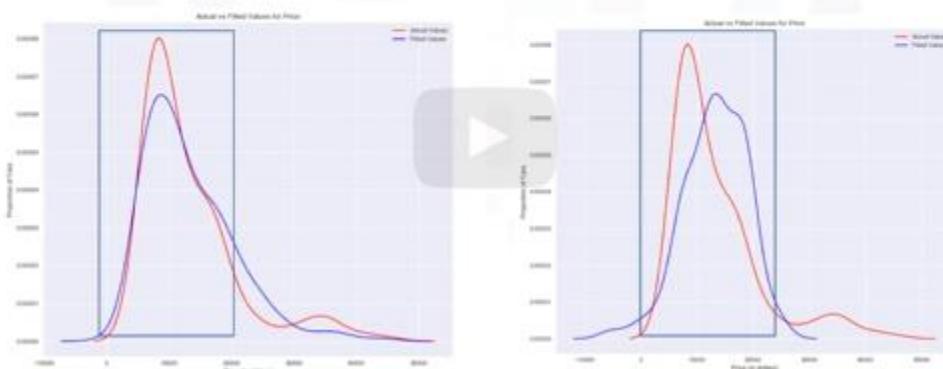
Compare the distribution plots:

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



This is an example of using a distribution plot. The dependent variable or feature is price. The fitted values that result from the model are in blue. The actual values are in red. We see the predicted values for prices in the range from 40,000 to 50,000 are inaccurate. The prices in the region from 10,000 to 20,000 are much closer to the target value.

## MLR – Distribution Plots



In this example, we use multiple features or independent variables. Comparing it to the plot on the last slide, we see predicted values are much closer to the target values.

# 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)  
↑
```

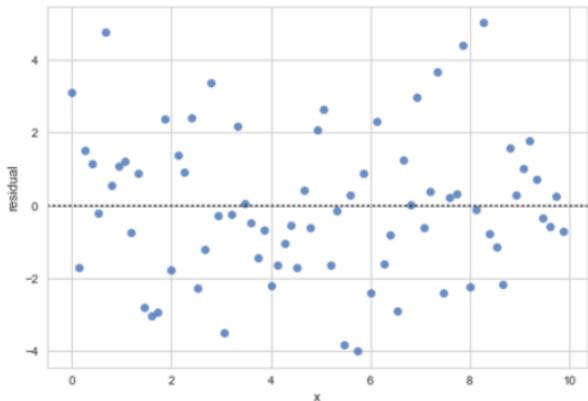
Here's the code to create a distribution plot. The actual values are used as a parameter. We wanted distribution instead of a histogram. So we want the hist parameters set to false. The color is red. The label is also included. The predicted values are included for the second plot. The rest of the parameters are set accordingly.

## Practice Quiz: Model Evaluation using Visualization

### Question 1

1/1 point (ungraded)

Consider the following Residual Plot, is our linear model correct:



Yes

Incorrect



### Answer

Correct: Correct

Submit

You have used 1 of 1 attempt

✓ Correct (1/1 point)

## Polynomial Regression and Pipelines

In this video, we will cover polynomial regression and pipelines. What do we do when a linear model is not the best fit for our data? Let's look into another type of regression model. The polynomial regression. We transform our data into a polynomial, then use linear regression to fit the parameter. Then we will discuss pipelines.

Pipelines are a way to simplify your code. Polynomial regression is a special case of the general linear regression.

This method is beneficial for describing curvilinear relationships. What is a curvilinear relationship? It's what you get by squaring or setting higher order terms of the predictor variables in the model transforming the data. The model can be quadratic, which means that the predictor variable in the model is squared. We use a bracket to indicate it as an exponent. This is a second order polynomial regression, with a figure representing the function. The model can be cubic, which means that the predictor variable is cubed.

This is the third order polynomial regression. We see by examining the figure that the function has more variation.

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

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

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

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

Let's look at an example from our data where we generate a polynomial regression model. In Python we do this by using the polyfit function. In this example, we develop a third order polynomial regression model base.

We can print out the model. Symbolic form for the model is given by the following expression: negative 1.557  $x_1$  cubed plus 204.8  $x_1$  squared plus 8965  $x_1$  plus 1.37 times 10 to the power of five. We could also have multi-dimensional polynomial linear regression. The expression can get complicated.

Here are just some of the terms for a two dimensional second order polynomial. Numpy's polyfit function cannot perform this type of regression. We use the preprocessing library in scikit-learn to create a polynomial feature object. The constructor takes the degree of the polynomial as a parameter. Then we transform the features into a polynomial feature with the fit underscore transform method. Let's do a more intuitive example.

Consider the features shown here. Applying the method we transform the data, we now have a new set of features that are a transformed version of our original features. As the dimension of the data gets larger, we may want to normalize multiple features in scikit-learn. Instead we can use the preprocessing module to simplify many tasks.

For example, we can standardize each feature simultaneously. We import StandardScaler. We train the object, fit the scale object, then transform the data into a new data frame on array x\_scale.

There are more normalization methods available in the preprocessing library as well as other transformations. We can simplify our code by using a pipeline library. There are many steps to getting a prediction. For example, normalization, polynomial transform, and linear regression. We simplify the process using a pipeline. Pipeline sequentially perform a series of transformations.

The last step carries out a prediction. First we import all the modules we need, then we import the library pipeline. We create a list of tuples, the first element in the tuple contains the name of the estimator model. The second element contains model constructor. We input the list in the pipeline constructor. We now have a pipeline object. We can train the pipeline by applying the train method to the pipeline object. We can also produce a prediction as well. The method normalizes the data, performs a polynomial transform, then outputs a prediction.

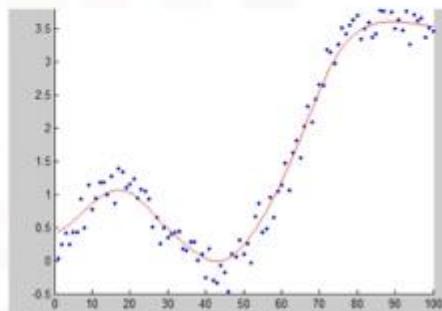
# Polynomial Regression and Pipelines

## Polynomial Regressions

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

**Curvilinear relationships:**

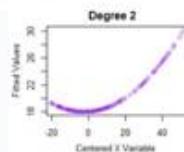
By squaring or setting higher-order terms  
of the predictor variables



## Polynomial Regression

- Quadratic – 2<sup>nd</sup> order

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



means that the predictor variable in the model is squared. We use a bracket to indicate it as an exponent. This is a second order polynomial regression, with a figure representing the function.

What is a **curvilinear relationship**? It's what you get by squaring or setting higher order terms of the predictor variables in the model transforming the data. The model can be quadratic, which

# Polynomial Regression

- Quadratic – 2<sup>nd</sup> order

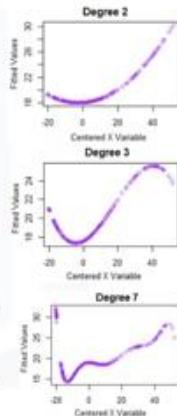
$$\hat{Y} = b_0 + b_1 x_1 + b_2 (x_1)^2$$

- Cubic – 3<sup>rd</sup> order

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

- Higher order

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



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

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

We see by examining the figure that the function has more variation. There also exists higher order polynomial regressions. When a good fit hasn't been achieved by second or third order. We can see in figures how much the graphs change, when we change the order of the polynomial regression. The degree of the regression makes a big difference and can result in a better fit if you pick the

# Polynomial Regression

1. Calculate Polynomial of 3<sup>rd</sup> order

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

```
p=np.poly1d(f)
```

2. We can print out the model

```
print (p)
```

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

Let's look at an example from our data where we generate a polynomial regression model. In Python we do this by using the **polyfit function**. In this example, we develop a third order polynomial regression model base. We can print out the model. Symbolic form for the model is given by the following expression:  
negative 1.557 x1 cubed plus 204.8 x one squared plus 8965 x1 plus 1.37 times 10 to the power of five.

# Polynomial Regression with More than One Dimension

- We can also have multi dimensional polynomial linear regression

$$\hat{Y} = b_0 + b_1 X_1 + b_2 X_2 + b_3 X_1 X_2 + b_4 (X_1)^2 + b_5 (X_2)^2 + ..$$

function cannot perform this type of regression.

We could also have multi-dimensional polynomial linear regression. The expression can get complicated. Here are just some of the terms for a two dimensional second order polynomial. Numpy's polyfit

## 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']])
```

## Polynomial Regression with More than One Dimension

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

$X_1$	$X_2$
1	2

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



$X_1$	$X_2$	$X_1X_2$	$X_1^2$	$X_2^2$
1	2	(1)2	1	(2) <sup>2</sup>
1	2	2	1	4

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

## 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']])
```

For example, we can standardize each feature simultaneously. We import StandardScaler. We train the object, fit the scale object, then transform the data into a new data frame or array x\_scale. There are more normalization methods available in the preprocessing library as well as other transformations.

## Pipelines

1.

## Pipelines

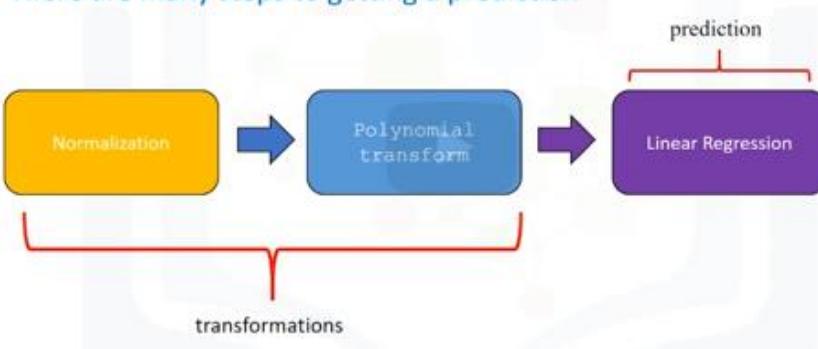
- There are many steps to getting a prediction



2.

## Pipelines

- There are many steps to getting a prediction



There are many steps to getting a prediction. For example, normalization, polynomial transform, and linear regression.

3.

## Pipelines

```

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

First we import all the modules we need, then we import the library pipeline.

4.

## Pipeline Constructor

```

Input=[('scale',StandardScaler()),('polynomial',PolynomialFeatures(degree=2),...,
('mode',LinearRegression()))
• Pipeline constructor
pipe=Pipeline(Input)
  
```

pipeline object

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

We input the list in the pipeline constructor.

We now have a 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']])
```



We can train the pipeline by applying the `train` method to the pipeline object. We can also produce a prediction as well.

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

## Practice Quiz: Polynomial Regression and Pipelines

Bookmarked

### Question 1

1/1 point (ungraded)

what is the order of the following Polynomial

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

1

2

3



#### Answer

Correct: Correct

Submit

You have used 1 of 2 attempts

✓ Correct (1/1 point)

## Measures for In-Sample Evaluation

Now that we've seen how we can evaluate a model by using visualization, we want to numerically evaluate our models. Let's look at some of the measures that we use for in-sample evaluation.

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

To measure the MSE, we find the difference between the actual value  $y$  and the predicted value  $\hat{y}$ , then square it. In this case, the actual value is 150; the predicted value is 50. Subtracting these points we get 100. We then square the number. We then take the Mean or average of all the errors by adding them all together and dividing by the number of samples.

To find the MSE in Python, we can import the "mean\_squared\_error" from "scikit-learn.metrics". The "mean\_squared\_error" function gets two inputs: the actual value of target variable and the predicted value of target variable. R-squared is also called the coefficient of determination. It's a measure to determine how close the data is to the fitted regression line. So how close is our actual data to our estimated model?

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

In this example the average of the data points  $y$  is 6. Coefficient of Determination  $R^2$  is 1 minus the ratio of the MSE of the regression line divided by the MSE of the average of the data points. For the most part, it takes values between 0 and 1.

Let's look at a case where the line provides a relatively good fit. The blue line represents the regression line. The blue squares represent the MSE of the regression line. The red line represents the average value of the data points. The red squares represent the MSE of the red line.

We see the area of the blue squares is much smaller than the area of the red squares. In this case, because the line is a good fit, the Mean squared error is small, therefore the numerator is small.

The Mean squared error of the line is relatively large, as such the numerator is large. A small number divided by a larger number is an even smaller number. Taken to an extreme this value tends to zero. If we plug in this value from the previous slide for  $R^2$ , we get a value near one, this means the line is a good fit for the data. Here is an example of a line that does not fit the data well.

If we just examine the area of the red squares compared to the blue squares, we see the area is almost identical. The ratio of the areas is close to one.

In this case the  $R^2$  is near zero. This line performs about the same as just using the average of the data points, therefore, this line did not perform well. We find the R-squared value in Python by using the score method, in the linear regression object.

From the value that we get from this example, we can say that approximately 49.695% of the variation of price is explained by this simple linear model. Your  $R^2$  value is usually between 0 and 1. If your  $R^2$  is negative, it can be due to over fitting .

## Measures for In-Sample Evaluation

### Measures for In-Sample Evaluation

- A way to numerically determine how good the model fits on dataset.

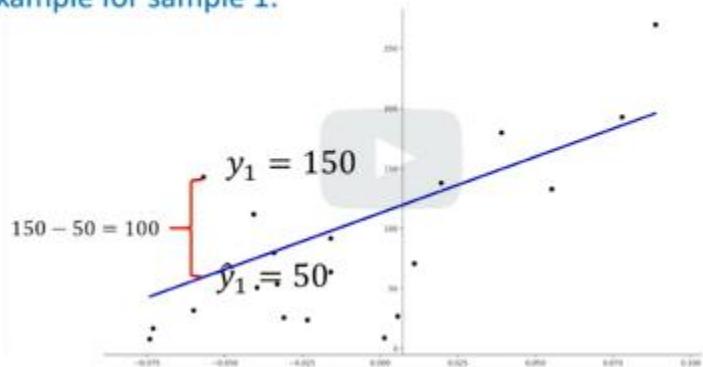
- Two important measures to determine the fit of a model:
  - Mean Squared Error (MSE)
  - R-squared (R<sup>2</sup>)

To Measure the MSE:

1.

### Mean Squared Error (MSE)

- For Example for sample 1:

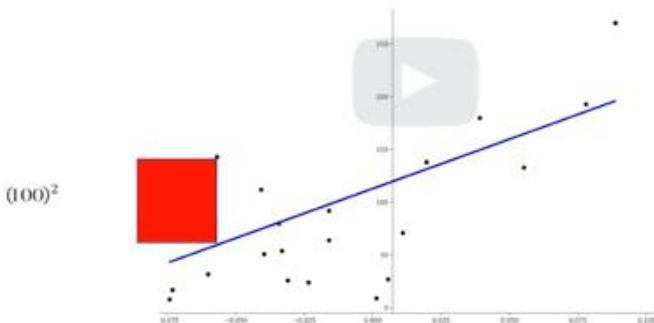


To measure the MSE, we find the difference between the actual value  $y$  and the predicted value  $\hat{y}$  then square it. In this case, the actual value is 150; the predicted value is 50. Subtracting these points we get 100

2.

### Mean Squared Error (MSE)

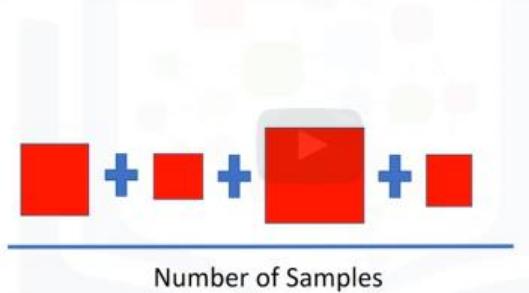
- To make all the values positive we square it



We then square the number.

3.

## Mean Squared Error (MSE)



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

4.

### To find the MSE in Python

## 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
```

To find the MSE in Python, we can import the “mean\_squared\_error” from “scikit-learn.metrics”. The `mean_squared_error` function gets two inputs: the actual value of target variable and the predicted value of target variable.

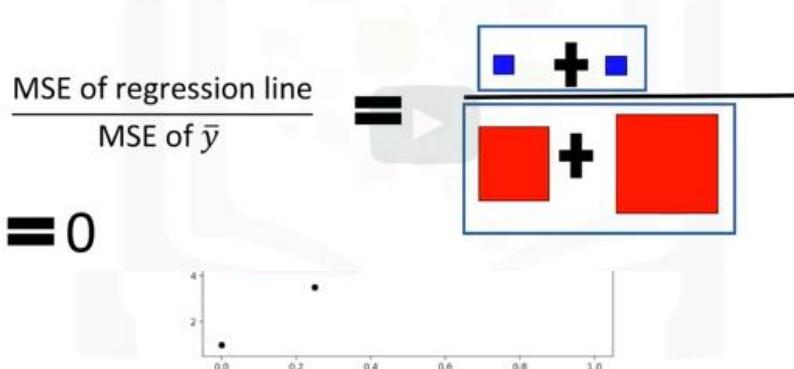
## R-squared/ R<sup>2</sup>

- 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

### Example:

## Coefficient of Determination ( $R^2$ )

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



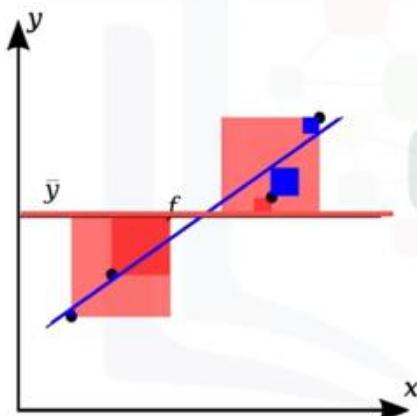
In this example the average of the data points  $y$  is 6.

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

## Coefficient of Determination ( $R^2$ )

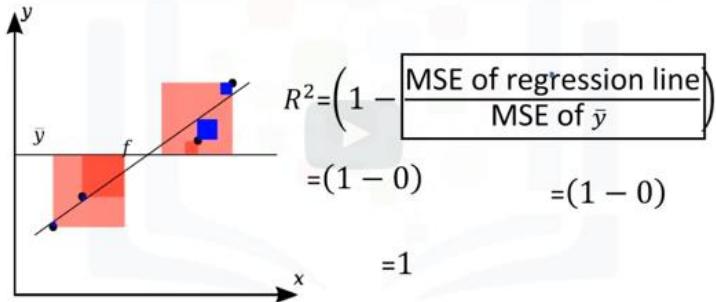


- The blue line represents the regression line
- The blue squares represent the MSE of the regression line
- The red line represents the average value of the data points
- The red squares represent the MSE of the red line
- We see the area of the blue squares is much smaller than the area of the red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

Let's look at a case where the line provides a relatively good fit. The blue line represents the regression line. The blue squares represent the MSE of the regression line. The red line represents the average value of the data points. The red squares represent the MSE of the red line. We see the area of the blue squares is much smaller than the area of the red squares.

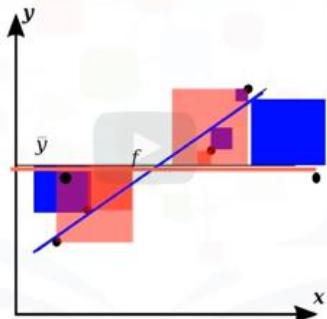
In this case, because the line is a good fit, the Mean squared error is small, therefore the numerator is small. The Mean squared error of the line is relatively large, as such the numerator is large. A small number divided by a larger number is an even smaller number. Taken to an extreme this value tends to zero.

## Coefficient of Determination ( $R^2$ )



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

## Coefficient of Determination ( $R^2$ )



Here is an example of a line that does not fit the data well. If we just examine the area of the red squares compared to the blue squares, we see the area is almost identical.

## Coefficient of Determination ( $R^2$ )

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

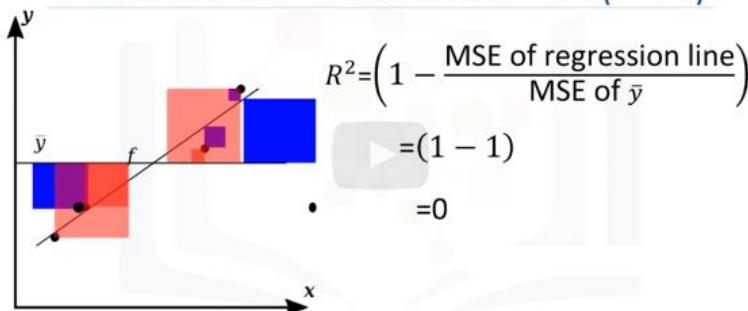
If we just examine the area of the red squares compared to the blue squares, we see the area is almost identical.

## Coefficient of Determination ( $R^2$ )

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

If we just examine the area of the red squares compared to the blue squares, we see the area is almost identical. The ratio of the areas is close to one.

## Coefficient of Determination ( $R^2$ )



In this case the  $R^2$  is near zero. This line performs about the same as just using the average of the data points, therefore, this line did not perform well.

## R-squared/ $R^2$

- Generally the values of the MSE are between 0 and 1.

- We can calculate the  $R^2$  as follows

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

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

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

We find the R-squared value in Python by using the score method, in the linear regression object. From the value that we get from this example, we can say that approximately 49.695% of the variation of price is explained by this simple linear model. Your  $R^2$  value is usually between 0 and 1. If your  $R^2$  is negative, it can be due to over fitting that we will discuss in the next module.

## Practice Quiz: Measures for In-Sample Evaluation

Bookmarked

### Question 1

1/1 point (ungraded)

Consider the following lines of code; what value does the variable out contain?

```
1 lm = LinearRegression()
2 lm.score(X,y)
3 X = df[['highway-mpg']]
4 Y = df['price']
5 lm.fit(X, Y)
6 out=lm.score(X,y)
```

Mean Squared Error

The Coefficient of Determination or R^2



#### Answer

Correct: Correct

Submit

You have used 1 of 1 attempt

---

✓ Correct (1/1 point)

## Prediction and Decision Making

In this video our final topic will be on prediction and decision making. How can we determine if our model is correct?

The first thing you should do is make sure your model results make sense. You should always use visualization, numerical measures for evaluation, and comparing between different models.

Let's look at an example of prediction. If you recall we trained the model using the fit method. Now, we want to find out what the price would be for a car that has a highway miles per gallon of 30. Plugging this value into the predict method gives us a resulting price of \$13,771.30 this seems to make sense. For example, the value is not negative, extremely high, or extremely low. We can look at the coefficients by examining the coeff underscore attribute. If you recall the expression for the simple linear model that predicts price from highway miles per gallon.

This value corresponds to the multiple of the highway miles per gallon feature. As such an increase of one unit in highway miles per gallon the value of the car decreases approximately 821 dollars. This value also seems reasonable sometimes your model will produce values that don't make sense. For example, if we plot the model out for highway miles per gallon in the ranges of 0 to 100 we get negative values for the price. This could be because the values in that range are not realistic the linear assumption is incorrect or we don't have data for cars in that range. In this case it is unlikely that a car will have fuel mileage in that range so our model seems valid. To generate a sequence of values in a specified range import numpy then use the numpy arrange function to generate the sequence the sequence starts at 1 and increments by 1 until we reach 100.

The first parameter is the starting point of the sequence. The second parameter is the endpoint plus one of the sequence the final parameter is the step size between elements in the sequence.

In this case it's one so we increment the sequence one step at a time from one to two and so on. We can use the output to predict new values the output is a numpy array many of the values are negative. Using a regression plot to visualize your data is the first method you should try. See the labs for examples of how to plot polynomial regression for this example the effect of the independent variable is evident in this case. The data trends down as the dependent variable increases the plot also shows some non-linear behavior.

Examining the residual plot we see in this case, the residuals have a curvature suggesting non-linear behavior. A distribution plot is a good method for multiple linear regression. For example, we see the predicted values for prices in the range from thirty thousand to fifty thousand are inaccurate this suggests a non-linear model may be more suitable or we need more data in this range. The mean square error is perhaps the most intuitive numerical measure for determining if a model is good or not. Let's see how different measures of mean square error impact the model.

The figure shows an example of a mean square error of 3495. This example has a mean square error of three thousand six hundred and fifty two. The final plot has a mean square error of twelve thousand eight hundred and seventy. As the square error increases the targets get further from the predicted points. As we discussed r squared is another popular method to evaluate your model.

- It tells you how well your line fits into the model. r squared values range from zero to one.
- r squared tells us what percent of the variability in the dependent variable is accounted for by the regression on the independent variable.
- An r squared of 1 means that all movements of another dependent variable are completely explained by movements in the independent variables.

In this plot we see the target points in red and the predicted line in blue. An r squared of 0.9986 the model appears to be a good fit. That means that more than 99 of the variability of the predicted variable is explained by the independent variables. This model has an r squared of 0.9226 there still is a strong linear relationship model is still a good fit. An r squared of 0.806 of the data we can visually see that the values are scattered around the line.

They are still close to the line and we can say that 80 percent of the variability of the predicted variable is explained by the independent variables. And an r squared 0.61 means that approximately 61 percent of the observed variation can be

explained by the independent variables. An acceptable value for r squared depends on what field you are studying and what your use case is. Falcon Miller 1992 suggests that an acceptable r squared value should be at least 0.1.

Does a lower mean square error imply better fit?

- Not necessarily MSE4 and 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.
- Polynomial regression will also have a smaller MSE than regular regression.

## Prediction and Decision Making

### How can we determine if our model is correct?

## 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

### 1. Do the predicted value makes sense?

#### 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

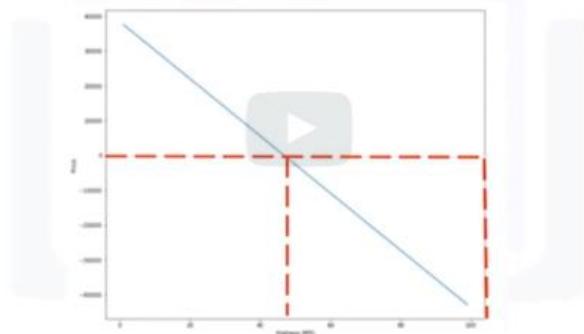
```
lm.coef  
-821.73337832
```

• Price = 38423.31 821.73 \* highway-mpg

If you recall we trained the model using the fit method. Now, we want to find out what the price would be for a car that has a highway miles per gallon of 30. Plugging this value into the predict method gives us a resulting price of \$13,771.30 this seems to make sense. For example, the value is not negative, extremely high, or extremely low. We can look at the coefficients by examining the coeff underscore attribute. If you recall the expression for the simple linear model that predicts price from highway miles per

gallon. This value corresponds to the multiple of the highway miles per gallon feature. As such an increase of one unit in highway miles per gallon the value of the car decreases approximately 821 dollars. This value also seems reasonable sometimes your model will produce values that don't make sense.

## Do the predicted values make sense



Sometimes your model will produce values that don't make sense. For example, if we plot the model out for highway miles per gallon in the ranges of 0 to 100 we get negative values for the price. This could be because the values in that range are not realistic the linear assumption is incorrect or we don't have data for cars in that range. In this case it is unlikely that a car will have fuel mileage in that range so our model seems valid.

## 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
```

To generate a sequence of values in a specified range import numpy then use the numpy `arrange` function to generate the sequence the sequence starts at 1 and increments by 1 until we reach 100. The first parameter is the starting point of the sequence. The second parameter is the endpoint plus one of the sequence the final parameter is the step size between elements in the sequence. In this case it's one so we increment the sequence one step at a time from one to two and so on.

## Do the predicted values make sense

- We can predict new values

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

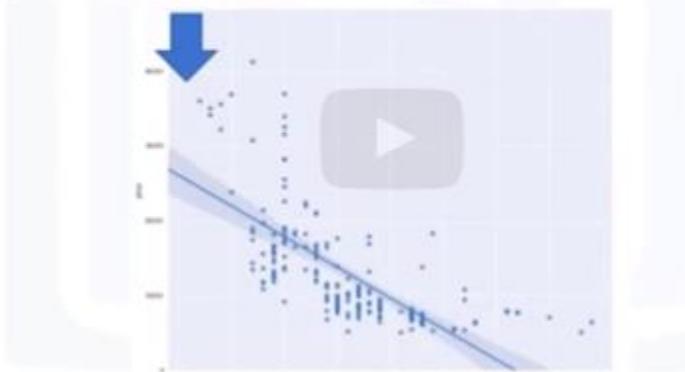
```
array([[ 20861.37512984, -26778.42361531, -28958.18752319, 35136.37512984,  
    94314.42898821, -24464.99988823, -22671.27120194, -28484.42898824,  
    31027.79585329, -26109.97297984, -29388.33489842, -29542.50518108,  
    27740.77193897, -20918.33956105, -26097.39518331, -21775.37180501,  
    29432.83982488, -20232.19594834, -24819.37387004, -21988.83929172,  
    21512.87194834, -21708.33956105, -24797.39518331, -21775.37180501,  
    17876.87194831, -17058.33956105, -14334.30444347, -15414.77324714,  
    14593.82798882, -13771.0406089, -12949.37112918, -12127.82775388,  
    11208.18279882, -10484.3499521, -9604.77415489, -8833.80423857,  
    40130.18279882, -39307.4199821, -38479.74613489, -37652.87321199,  
    6732.23734499, -1912.58394884, -3588.77388631, -2287.67321199,  
    1485.36382387, -633.57645335, -138.36282387, -1018.8946013,  
    -418.8946013, -489.8946013, -244.8946013, -244.8946013,  
    -5128.7621829, -5893.19657123, -4772.32994805, -7182.76322787,  
    -8411.49818619, -9232.73308451, -10558.94344286, -10880.49464116,  
    -12792.43231346, -12252.18397814, -13343.49487912, -14147.43398465,  
    -15942.59487912, -17132.18397814, -18323.49487912, -19522.60488465,  
    -18274.29724409, -19398.42962438, -19919.76495227, -20741.49793052,  
    -21583.21075934, -22338.94613767, -23206.49751099, -24028.63089831,  
    -23850.18427293, -24871.99793092, -24893.42326297, -27375.36460797,  
    -27375.36460797, -27375.36460797, -27375.36460797, -27375.36460797,  
    -21424.15129921, -52249.74467751, -33047.49055587, -33089.20143617,  
    -34712.94481249, -20332.49819032, -36294.42158914, -37176.1649786,  
    -27997.89832579, -20819.4327084, -39941.38582823, -40463.59988779,  
    -41384.82182303, -40104.5432178, -42309.29809271])
```

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

## 2. Visualization

### Visualization

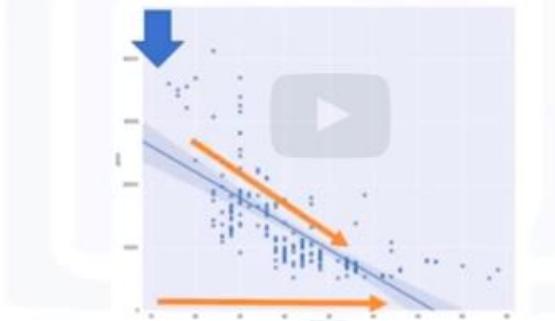
- Simply visualizing your data with a regression



Using a regression plot to visualize your data is the first method you should try.

### Visualization

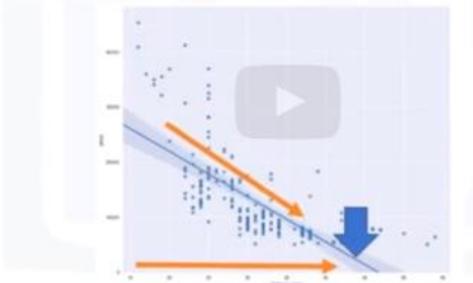
- Simply visualizing your data with a regression



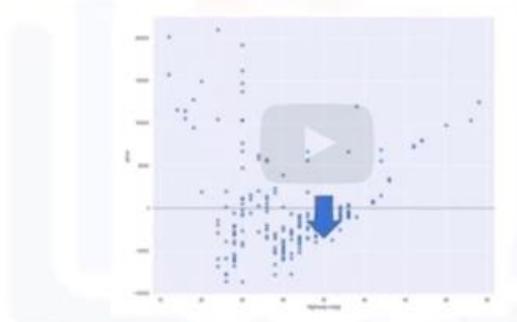
For this example the effect of the independent variable is evident in this case. The data trends down as the dependent variable increases the plot also shows some non-linear behavior.

### Visualization

- Simply visualizing your data with a regression

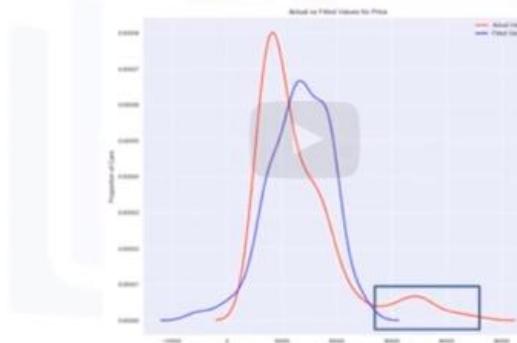


## Residual Plot



Examining the residual plot we see in this case, the residuals have a curvature suggesting non-linear behavior.

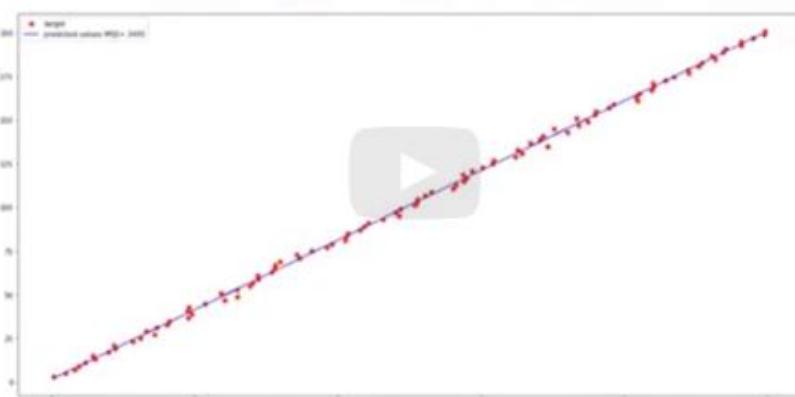
## Visualization



A distribution plot is a good method for multiple linear regression. For example, we see the predicted values for prices in the range from thirty thousand to fifty thousand are inaccurate this suggests a non-linear model may be more suitable or we need more data in this range.

## 3. Numerical Measures of Evaluation

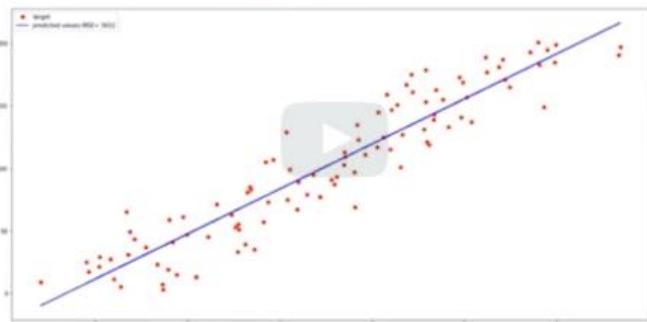
### Numerical measures for Evaluation



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

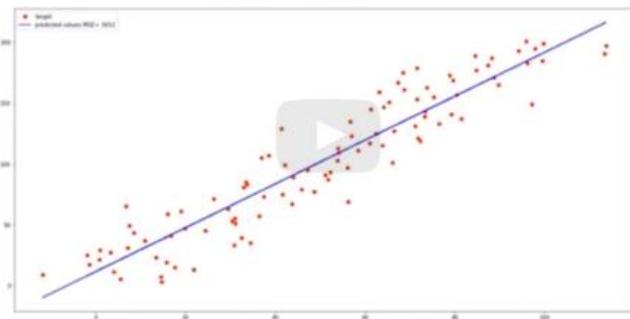
The figure shows an example of a **mean square error of 3495**.

## Numerical measures for Evaluation



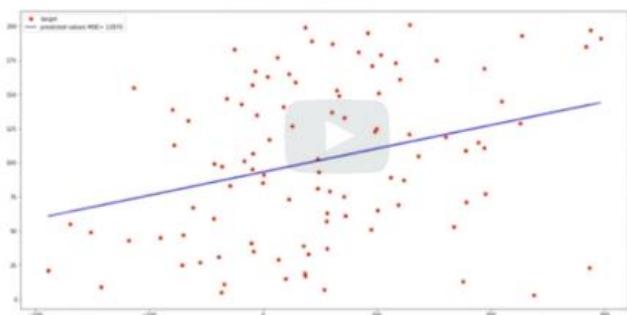
This example has a mean square error of three thousand six hundred and fifty two.

## Numerical measures for Evaluation



The final plot has a mean square error of twelve thousand eight hundred and seventy.

## Numerical measures for Evaluation



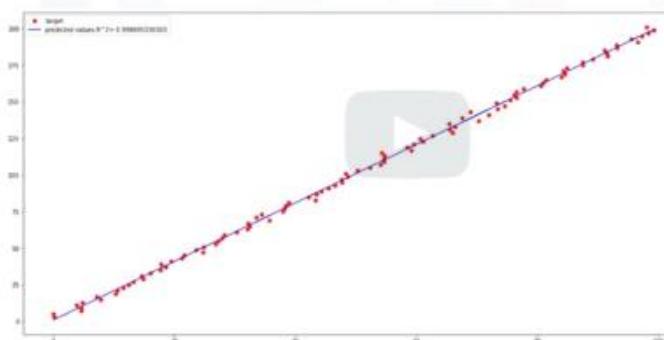
The final plot has a mean square error of twelve thousand eight hundred and seventy.

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

As we discussed r squared is another popular method to evaluate your model.

- It tells you how well your line fits into the model.
- r squared values range from zero to one.
- r squared tells us what percent of the variability in the dependent variable is accounted for by the regression on the independent variable.
- An r squared of 1 means that all movements of another dependent variable are completely explained by movements in the independent variables.

## Numerical measures for Evaluation



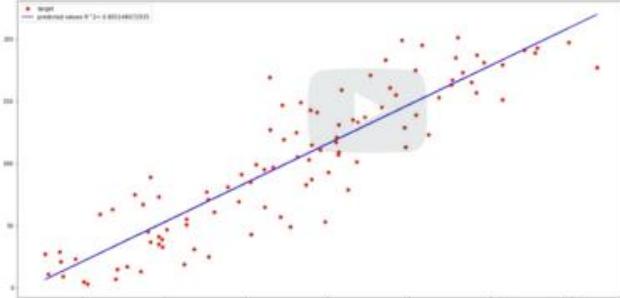
In this plot we see the target points in red and the predicted line in blue. An r squared of 0.9986 the model appears to be a good fit. That means that more than 99 of the variability of the predicted variable is explained by the independent variables.

## Numerical measures for Evaluation



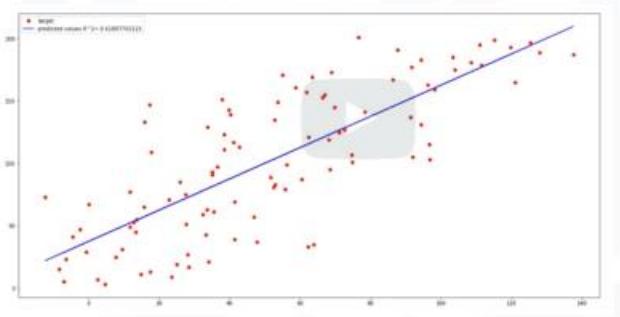
This model has an r squared of 0.9226 there still is a strong linear relationship model is still a good fit.

## Numerical measures for Evaluation



An r squared of 0.806 of the data we can visually see that the values are scattered around the line. They are still close to the line and we can say that 80 percent of the variability of the predicted variable is explained by the independent variables.

## Numerical measures for Evaluation



And an r squared 0.61 means that approximately 61 percent of the observed variation can be explained by the independent variables. An acceptable value for r squared depends on what field you are studying and what your use case is. Falcon Miller 1992 suggests that an acceptable r squared value should be at least 0.1.

## Comparing MLR and SLR

Does a lower Mean Square Error imply better fit?

- Not necessarily
1. Mean Square Error for a Multiple Linear Regression Model will be smaller than the Mean Square Error for a Simple Linear Regression model, since the errors of the data will decrease when more variables are included in the model
  2. Polynomial regression will also have a smaller Mean Square Error than the linear regular regression
  3. In the next section we will look at more accurate ways to evaluate the model

## Lesson Summary

[Bookmark this page](#)

In this lesson, you have learned how to:

- **Define the explanatory variable and the response variable:** Define the response variable ( $y$ ) as the focus of the experiment and the explanatory variable ( $x$ ) as a variable used to explain the change of the response variable. Understand the differences between Simple Linear Regression because it concerns the study of only one explanatory variable and Multiple Linear Regression because it concerns the study of two or more explanatory variables.
- **Evaluate the model using Visualization:** By visually representing the errors of a variable using scatterplots and interpreting the results of the model.
- **Identify alternative regression approaches:** Use a Polynomial Regression when the Linear regression does not capture the curvilinear relationship between variables and how to pick the optimal order to use in a model.
- **Interpret the R-square and the Mean Square Error:** Interpret R-square ( $\times 100$ ) as the percentage of the variation in the response variable  $y$  that is explained by the variation in explanatory variable(s)  $x$ . The Mean Squared Error tells you how close a regression line is to a set of points. It does this by taking the average distances from the actual points to the predicted points and squaring them.

# Model Development

Estimated time needed: **30** minutes

## Objectives

After completing this lab you will be able to:

- Develop prediction models

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

Some questions we want to ask in this module

- Do I know if the dealer is offering fair value for my trade-in?
- Do I know if I put a fair value on my car?

In data analytics, we often use **Model Development** to help us predict future observations from the data we have.

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

## Setup

Import libraries:

```
In [1]: import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt
```

Load the data and store it in dataframe `df`:

This dataset was hosted on IBM Cloud object. Click [HERE](#) for free storage.

```
In [2]: # path of data  
path = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/automobil  
df = pd.read_csv(path)  
df.head()
```

Out[2]:	symboling	normalized-losses	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	length	...	compression-ratio	horsepower	peak-rpm	city-mpg	highway-mpg	price	city-L/100km
0	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	...	9.0	111.0	5000.0	21	27	13495.0	11.190476
1	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	...	9.0	111.0	5000.0	21	27	16500.0	11.190476
2	1	122	alfa-romero	std	two	hatchback	rwd	front	94.5	0.822681	...	9.0	154.0	5000.0	19	26	16500.0	12.368421
3	2	164	audi	std	four	sedan	fwd	front	99.8	0.848630	...	10.0	102.0	5500.0	24	30	13950.0	9.791667
4	2	164	audi	std	four	sedan	4wd	front	99.4	0.848630	...	8.0	115.0	5500.0	18	22	17450.0	13.055556

5 rows × 29 columns

horsepower-binned diesel gas

Medium	0	1

## 1. Linear Regression and Multiple Linear Regression

### Linear Regression

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

#### Simple Linear Regression

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

- The predictor/independent variable (X)
- The response/dependent variable (that we want to predict)(Y)

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

$$Y : \text{Response Variable} \quad X : \text{Predictor Variables}$$

#### Linear Function

$$Y_{hat} = a + bX$$

- a refers to the **intercept** of the regression line, in other words: the value of Y when X is 0
- b refers to the **slope** of the regression line, in other words: the value with which Y changes when X increases by 1 unit

**Let's load the modules for linear regression:**

```
In [3]: from sklearn.linear_model import LinearRegression
```

Create the linear regression object:

```
In [4]: lm = LinearRegression()  
lm
```

```
Out[4]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,  
normalize=False)
```

**How could "highway-mpg" help us predict car price?**

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

```
In [5]: X = df[['highway-mpg']]  
Y = df['price']
```

Fit the linear model using highway-mpg:

```
In [6]: lm.fit(X,Y)
```

```
Out[6]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,  
normalize=False)
```

We can output a prediction:

```
In [7]: Yhat=lm.predict(X)  
Yhat[0:5]
```

```
Out[7]: array([16236.50464347, 16236.50464347, 17058.23802179, 13771.3045085 ,  
20345.17153508])
```

What is the value of the intercept (a)?

```
In [8]: lm.intercept_
```

```
Out[8]: 38423.3058581574
```

What is the value of the slope (b)?

```
In [9]: lm.coef_
```

```
Out[9]: array([-821.73337832])
```

## What is the final estimated linear model we get?

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

$$Y\hat{=} a + bX$$

Plugging in the actual values we get:

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

### Question #1 a):

Create a linear regression object called "lm1".

```
In [10]: # Write your code below and press Shift+Enter to execute
lm1 = LinearRegression()
lm1
```

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

Click here for the solution [```python lm1 = LinearRegression\(\) lm1```](#)

### Question #1 b):

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

```
In [11]: # Write your code below and press Shift+Enter to execute
lm1.fit(df[['engine-size']], df[['price']])
lm1
```

```
Out[11]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                           normalize=False)
```

Click here for the solution [```python lm1.fit\(df\[\['engine-size'\]\], df\[\['price'\]\]\) lm1```](#)

## Question #1 c):

Find the slope and intercept of the model.

### Slope

```
In [12]: # Write your code below and press Shift+Enter to execute  
lm1.coef_  
  
Out[12]: array([[166.86001569]])
```

### Intercept

```
In [13]: # Write your code below and press Shift+Enter to execute  
lm1.intercept_  
  
Out[13]: array([-7963.33890628])
```

[Click here for the solution](#) `'''python # Slope lm1.coef_ # Intercept lm1.intercept_'''`

## Question #1 d):

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

```
In [14]: # Write your code below and press Shift+Enter to execute  
Yhat = -7963.34 + 166.86*X  
Price = -7963.34 + 166.86*engine-size
```

```
NameError                                                 Traceback (most recent call last)  
<ipython-input-14-ae3ba7b8c381> in <module>  
      1 # Write your code below and press Shift+Enter to execute  
      2 Yhat = -7963.34 + 166.86*X  
----> 3 Price = -7963.34 + 166.86*engine-size
```

`NameError: name 'engine' is not defined`

[Click here for the solution](#) `'''python # using X and Y Yhat=-7963.34 + 166.86*X Price=-7963.34 + 166.86*engine-size'''`

## Multiple Linear Regression

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

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

$Y$  : Response Variable  
 $X_1$  : Predictor Variable 1  
 $X_2$  : Predictor Variable 2  
 $X_3$  : Predictor Variable 3  
 $X_4$  : Predictor Variable 4

$a$  : intercept  
 $b_1$  : coefficients of Variable 1  
 $b_2$  : coefficients of Variable 2  
 $b_3$  : coefficients of Variable 3  
 $b_4$  : coefficients of Variable 4

The equation is given by:

$$Y_{\text{hat}} = a + b_1 X_1 + b_2 X_2 + b_3 X_3 + b_4 X_4$$

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

- Horsepower
- Curb-weight
- Engine-size
- Highway-mpg

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

```
In [15]: Z = df[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']]
```

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

```
In [16]: lm.fit(Z, df['price'])
```

```
Out[16]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                           normalize=False)
```

What is the value of the intercept( $a$ )?

```
In [17]: lm.intercept_
```

```
Out[17]: -15806.624626329209
```

What are the values of the coefficients ( $b_1, b_2, b_3, b_4$ )?

```
In [18]: lm.coef_
```

```
Out[18]: array([53.49574423, 4.70770099, 81.53026382, 36.05748882])
```

What is the final estimated linear model that we get?

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

$$Y_{\text{hat}} = a + b_1 X_1 + b_2 X_2 + b_3 X_3 + b_4 X_4$$

What is the linear function we get in this example?

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

## Question #2 a):

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

```
In [19]: # Write your code below and press Shift+Enter to execute
lm2 = LinearRegression()
lm2.fit(df[['normalized-losses', 'highway-mpg']], df['price'])
```

```
Out[19]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                           normalize=False)
```

Click here for the solution [```python lm2 = LinearRegression\(\) lm2.fit\(df\[\['normalized-losses', 'highway-mpg'\]\], df\['price'\]\)```](#)

## Question #2 b):

Find the coefficient of the model.

```
In [20]: # Write your code below and press Shift+Enter to execute
lm2.coef_
```

```
Out[20]: array([ 1.49789586, -820.45434016])
```

Click here for the solution [```python lm2.coef\\_```](#)

## 2. Model Evaluation Using Visualization

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

Import the visualization package, seaborn:

```
In [21]: # import the visualization package: seaborn
import seaborn as sns
%matplotlib inline
```

### Regression Plot

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

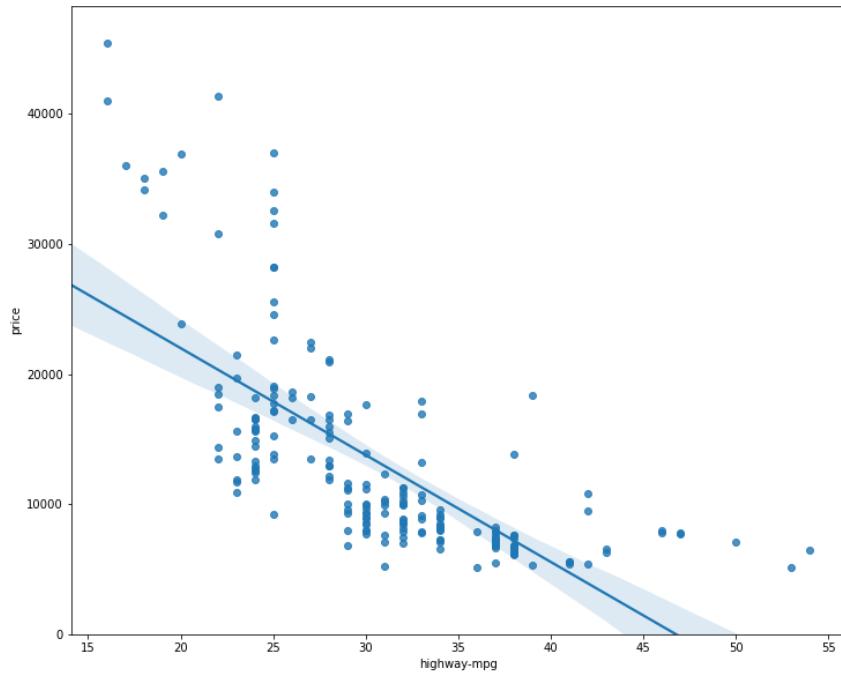
This plot will show a combination of a **scattered data points** (a **scatterplot**), as well as the fitted **linear regression line** going through the data. This will give us a reasonable estimate of the relationship between the two variables, the strength of the correlation, as well as the direction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

Let's visualize **highway-mpg** as potential predictor variable of price:

In [22]:

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

Out[22]: (0.0, 48266.65645753763)

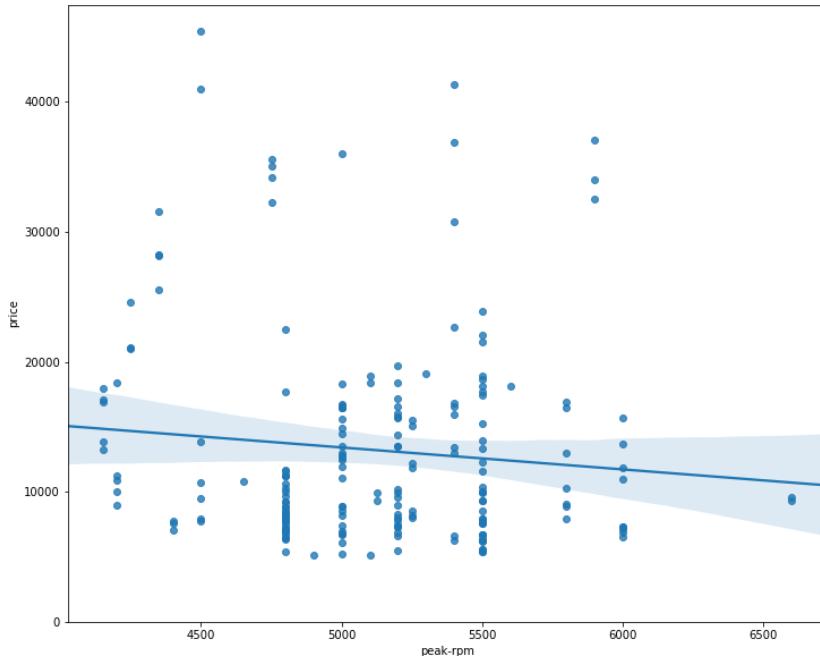


We can see from this plot that price is negatively correlated to highway-mpg since the regression slope is negative. One thing to keep in mind when looking at a regression plot is to pay attention to how scattered the data points are around the regression line. This will give you a good indication of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

In [23]:

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

Out[23]: (0.0, 47414.1)



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

### Question #3:

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

```
In [24]: # Write your code below and press Shift+Enter to execute
df[['peak-rpm', 'highway-mpg', 'price']].corr()
```

	peak-rpm	highway-mpg	price
peak-rpm	1.000000	-0.058598	-0.101616
highway-mpg	-0.058598	1.000000	-0.704692
price	-0.101616	-0.704692	1.000000

Click here for the solution `python # The variable "highway-mpg" has a stronger correlation with "price", it is approximate -0.704692 compared to "peak-rpm" which is approximate -0.101616. You can verify it using the following command: df[["peak-rpm","highway-mpg","price"]].corr() ```

## Residual Plot

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

What is a **residual**?

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

So what is a **residual plot**?

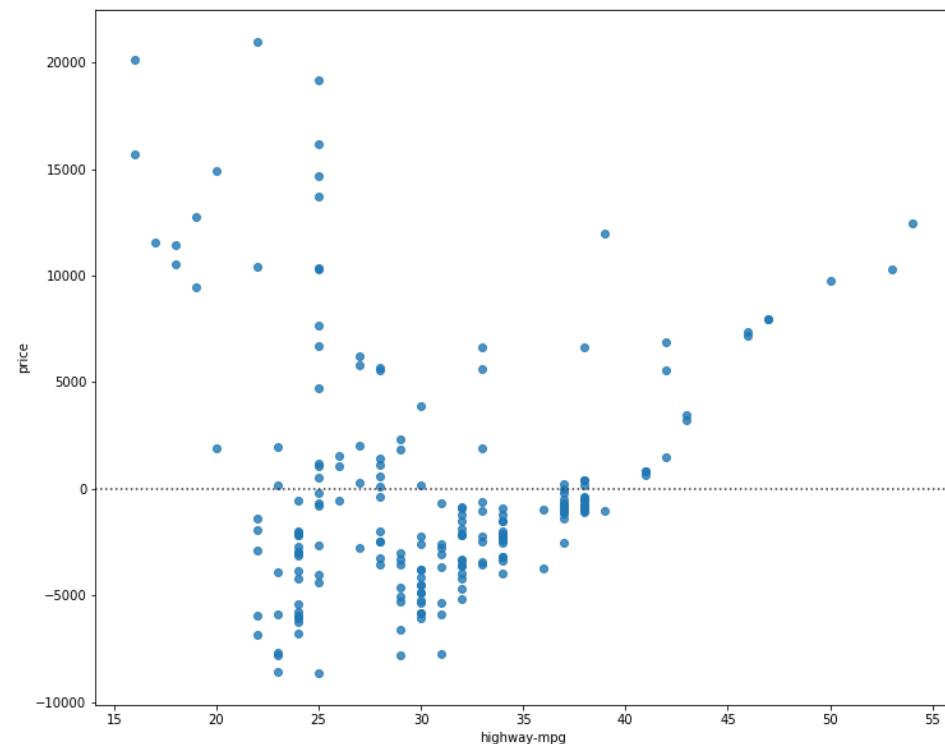
A residual plot is a graph that shows the residuals on the vertical y-axis and the independent variable on the horizontal x-axis.

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

We look at the spread of the residuals:

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

```
In [25]: width = 12  
height = 10  
plt.figure(figsize=(width, height))  
sns.residplot(df['highway-mpg'], df['price'])  
plt.show()
```



**What is this plot telling us?**

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

## Multiple Linear Regression

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

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

First, let's make a prediction:

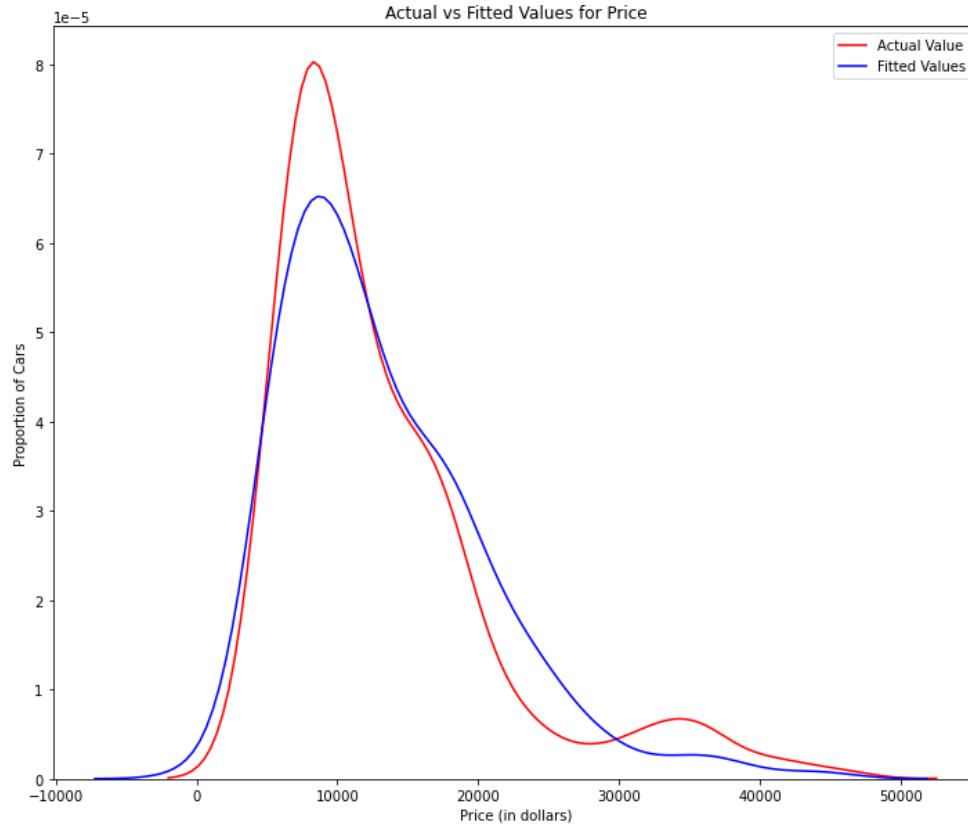
```
In [26]: Y_hat = lm.predict(z)

In [27]: plt.figure(figsize=(width, height))

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

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

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



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

### 3. Polynomial Regression and Pipelines

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

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

There are different orders of polynomial regression:

**Quadratic - 2nd Order**

$$Yhat = a + b_1X + b_2X^2$$

**Cubic - 3rd Order**

$$Yhat = a + b_1X + b_2X^2 + b_3X^3$$

**Higher-Order:**

$$Y = a + b_1X + b_2X^2 + b_3X^3 \dots$$

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

We will use the following function to plot the data:

```
In [28]: def PlotPolly(model, independent_variable, dependent_variabble, Name):
    x_new = np.linspace(15, 55, 100)
    y_new = model(x_new)

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

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

Let's get the variables:

```
In [29]: x = df['highway-mpg']
y = df['price']
```

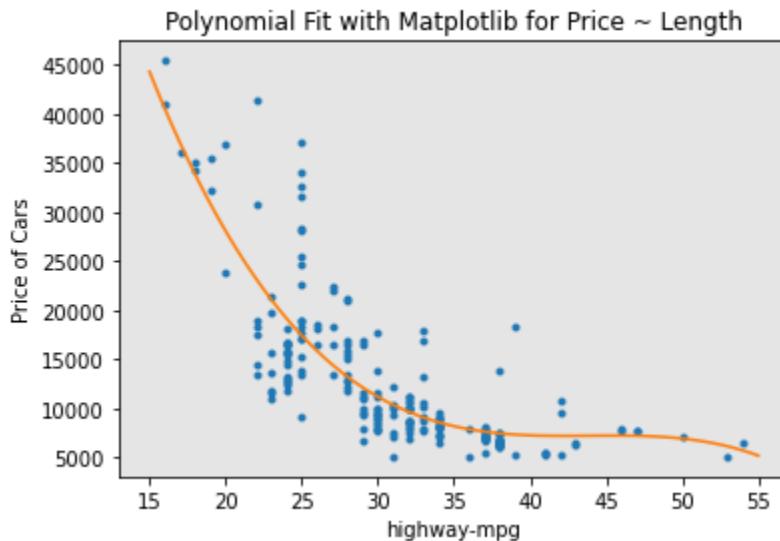
Let's fit the polynomial using the function **polyfit**, then use the function **poly1d** to display the polynomial function.

```
In [30]: # Here we use a polynomial of the 3rd order (cubic)
f = np.polyfit(x, y, 3)
p = np.poly1d(f)
print(p)
```

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

Let's plot the function:

```
In [31]: PlotPolly(p, x, y, 'highway-mpg')
```



```
In [32]: np.polyfit(x, y, 3)
```

```
Out[32]: array([-1.55663829e+00,  2.04754306e+02, -8.96543312e+03,  1.37923594e+05])
```

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

## Question #4:

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

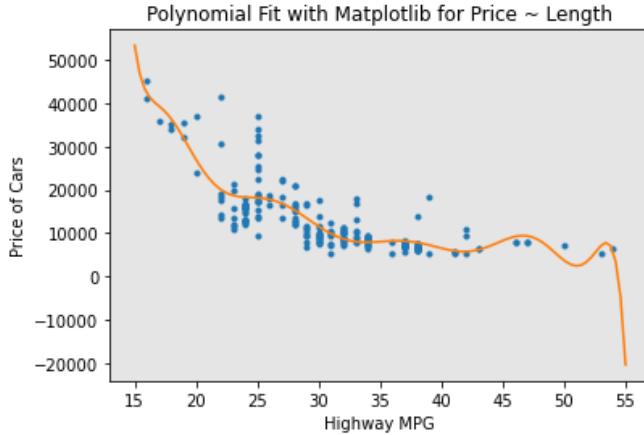
In [35]:

```
# Write your code below and press Shift+Enter to execute
f1 = np.polyfit(x, y, 11)
p1 = np.poly1d(f1)
print(p1)
```

```
11          10          9          8          7
-1.243e-08 x + 4.722e-06 x - 0.0008028 x + 0.08056 x - 5.297 x
6          5          4          3          2
+ 239.5 x - 7588 x + 1.684e+05 x - 2.565e+06 x + 2.551e+07 x - 1.491e+08 x + 3.879e+08
```

In [36]:

```
PlotPolly(p1, x, y, 'Highway MPG')
```



Click here for the solution ``python # Here we use a polynomial of the 11rd order (cubic) f1 = np.polyfit(x, y, 11) p1 = np.poly1d(f1) print(p1) PlotPolly(p1,x,y, 'Highway MPG') ````

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

$$Yhat = a + b_1X_1 + b_2X_2 + b_3X_1X_2 + b_4X_1^2 + b_5X_2^2$$

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

```
In [37]: from sklearn.preprocessing import PolynomialFeatures
```

We create a **PolynomialFeatures** object of degree 2:

```
In [38]: pr=PolynomialFeatures(degree=2)
pr
```

```
Out[38]: PolynomialFeatures(degree=2, include_bias=True, interaction_only=False)
```

```
In [39]: Z_pr=pr.fit_transform(Z)
```

In the original data, there are 201 samples and 4 features.

```
In [40]: Z.shape
```

```
Out[40]: (201, 4)
```

After the transformation, there are 201 samples and 15 features.

```
In [41]: Z_pr.shape
```

```
Out[41]: (201, 15)
```

## Pipeline

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

```
In [42]: from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

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

```
In [43]: Input=[('scale',StandardScaler()), ('polynomial', PolynomialFeatures(include_bias=False)), ('model',LinearRegression())]
```

We input the list as an argument to the pipeline constructor:

```
In [44]: pipe=Pipeline(Input)
pipe
```

```

Out[44]: Pipeline(memory=None,
      steps=[('scale', StandardScaler(copy=True, with_mean=True, with_std=True)), ('polynomial', PolynomialFeatures(degree=2, include_bias=False, interaction_only=False)), ('model', LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False))])

First, we convert the data type Z to type float to avoid conversion warnings that may appear as a result of StandardScaler taking float inputs.

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

In [45]: Z = Z.astype(float)
pipe.fit(Z,y)

Out[45]: Pipeline(memory=None,
      steps=[('scale', StandardScaler(copy=True, with_mean=True, with_std=True)), ('polynomial', PolynomialFeatures(degree=2, include_bias=False, interaction_only=False)), ('model', LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False))])

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

In [46]: ypipe=pipe.predict(Z)
ypipe[0:4]

Out[46]: array([13102.74784201, 13102.74784201, 18225.54572197, 10390.29636555])

```

## Question #5:

Create a pipeline that standardizes the data, then produce a prediction using a linear regression model using the features Z and target y.

```

In [47]: # Write your code below and press Shift+Enter to execute
Input=[('scale',StandardScaler()),('model',LinearRegression())]

pipe=Pipeline(Input)

pipe.fit(Z, y)

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

Out[47]: array([13699.11161184, 13699.11161184, 19051.65470233, 10620.36193015,
       15521.31420211, 13869.66673213, 15456.16196732, 15974.00907672,
       17612.35917161, 10722.32509097])

```

Click here for the solution ``python Input=[('scale',StandardScaler()),('model',LinearRegression())] pipe=Pipeline(Input) pipe.fit(Z,y) ypipe=pipe.predict(Z) ypipe[0:10] ````

## 4. Measures for In-Sample Evaluation

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

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

- R<sup>2</sup> / R-squared
- Mean Squared Error (MSE)

### R-squared

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

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

## Mean Squared Error (MSE)

The Mean Squared Error measures the average of the squares of errors. That is, the difference between actual value (y) and the estimated value ( $\hat{y}$ ).

## Model 1: Simple Linear Regression

Let's calculate the R^2:

```
In [48]: #highway_mpg_fit  
lm.fit(X, Y)  
# Find the R^2  
print('The R-square is: ', lm.score(X, Y))
```

The R-square is: 0.4965911884339176

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

Let's calculate the MSE:

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

```
In [49]: Yhat=lm.predict(X)  
print('The output of the first four predicted value is: ', Yhat[0:4])
```

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

Let's import the function `mean_squared_error` from the module `metrics`:

```
In [50]: from sklearn.metrics import mean_squared_error
```

We can compare the predicted results with the actual results:

```
In [51]:  
    mse = mean_squared_error(df['price'], Yhat)  
    print('The mean square error of price and predicted value is: ', mse)
```

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

## Model 2: Multiple Linear Regression

Let's calculate the R^2:

```
In [52]:  
    # fit the model  
    lm.fit(Z, df['price'])  
    # Find the R^2  
    print('The R-square is: ', lm.score(Z, df['price']))
```

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

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

Let's calculate the MSE.

We produce a prediction:

```
In [53]:  
    Y_predict_multifit = lm.predict(Z)
```

We compare the predicted results with the actual results:

```
In [54]:  
    print('The mean square error of price and predicted value using multifit is: ', \  
        mean_squared_error(df['price'], Y_predict_multifit))
```

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

## Model 3: Polynomial Fit

Let's calculate the R^2.

Let's import the function `r2\score` from the module `metrics` as we are using a different function.

```
In [55]: from sklearn.metrics import r2_score
```

We apply the function to get the value of R<sup>2</sup>:

```
In [56]: r_squared = r2_score(y, p(x))
print('The R-square value is: ', r_squared)
```

```
The R-square value is:  0.6741946663906517
```

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

## MSE

We can also calculate the MSE:

```
In [57]: mean_squared_error(df['price'], p(x))
```

```
Out[57]: 20474146.426361226
```

# 5. Prediction and Decision Making

## Prediction

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

```
In [58]: import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline
```

Create a new input:

```
In [59]: new_input=np.arange(1, 100, 1).reshape(-1, 1)
```

Fit the model:

```
In [60]: lm.fit(X, Y)
lm
```

```
Out[60]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                           normalize=False)
```

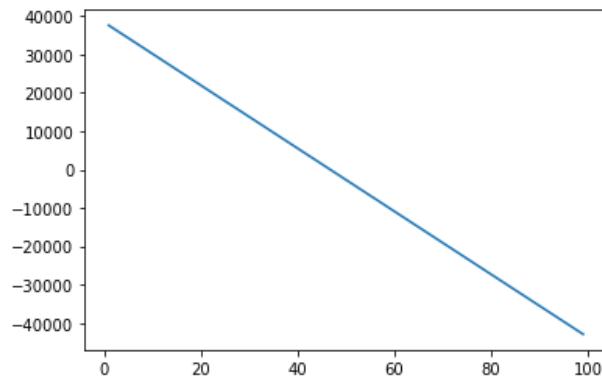
Produce a prediction:

```
In [61]: yhat=lm.predict(new_input)
yhat[0:5]
```

```
Out[61]: array([37601.57247984, 36779.83910151, 35958.10572319, 35136.37234487,
   34314.63896655])
```

We can plot the data:

```
In [62]: plt.plot(new_input, yhat)
plt.show()
```



## Decision Making: Determining a Good Model Fit

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

- *What is a good R-squared value?*

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

- *What is a good MSE?*

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

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

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

- R-squared: 0.49659118843391759
- MSE:  $3.16 \times 10^7$

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

- R-squared: 0.80896354913783497
- MSE:  $1.2 \times 10^7$

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

- R-squared: 0.6741946663906514
- MSE:  $2.05 \times 10^7$

## Simple Linear Regression Model (SLR) vs Multiple Linear Regression Model (MLR)

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

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

- **MSE:** The MSE of SLR is  $3.16 \times 10^7$  while MLR has an MSE of  $1.2 \times 10^7$ . The MSE of MLR is much smaller.
- **R-squared:** In this case, we can also see that there is a big difference between the R-squared of the SLR and the R-squared of the MLR. The R-squared for the SLR ( $\sim 0.497$ ) is very small compared to the R-squared for the MLR ( $\sim 0.809$ ).

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

## Simple Linear Model (SLR) vs. Polynomial Fit

- **MSE:** We can see that Polynomial Fit brought down the MSE, since this MSE is smaller than the one from the SLR.
- **R-squared:** The R-squared for the Polynomial Fit is larger than the R-squared for the SLR, so the Polynomial Fit also brought up the R-squared quite a bit.

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

## Multiple Linear Regression (MLR) vs. Polynomial Fit

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

### Conclusion

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

## Graded Quiz: Model Development

Bookmarked

Graded Quiz due Feb 17, 2022 21:32 +08

### Question 1

1/1 point (graded)

What does the following line of code do?

**lm = LinearRegression()**

- Predict a value
- Fit a regression object lm
- Create a linear regression object



#### Answer

Correct: Correct

### Question 2

1/1 point (graded)

What is the maximum value of R^2 that can be obtained?

- 10
- 1
- 0



#### Answer

Correct: Correct

**Submit**

You have used 1 of 2 attempts

---

✓ Correct (1/1 point)

### Question 3

1/1 point (graded)

We create a polynomial feature as follows "PolynomialFeatures(degree=2)"; what is the order of the polynomial?

0

1

2



#### Answer

Correct: Correct

[Submit](#)

You have used 1 of 2 attempts

[Reset](#)

✓ Correct (1/1 point)

### Question 4

1/1 point (graded)

Which statement is true about Polynomial linear regression?

Polynomial linear regression uses linear Wavelets

Although the predictor variables of Polynomial linear regression are not linear the relationship between the parameters or coefficients is linear

Polynomial linear regression is not linear in any way



#### Answer

Correct: Correct

[Submit](#)

You have used 1 of 2 attempts

[Reset](#)

[Show answer](#)

✓ Correct (1/1 point)

## Question 5

1/1 point (graded)

Consider the following equation:

$$y = b_0 + b_1 x$$

The variable  $y$  is what?

The target or dependent variable

The predictor or independent variable

The intercept



### Answer

Correct: Correct

Submit

You have used 1 of 2 attempts

## Module Introduction & Learning Objectives

 [Bookmark this page](#)

### Module Introduction

In this week's module, you will learn about the importance of model evaluation and discuss different data model refinement techniques. You will learn about model selection and how to identify overfitting and underfitting in a predictive model. You will also learn about using Ridge Regression to regularize and reduce standard errors to prevent overfitting a regression model and how to use the Grid Search method to tune the hyperparameters of an estimator.

### Learning Objectives

- Describe data model refinement techniques
- Explain overfitting, underfitting, and model selection
- Apply ridge regression to regularize and reduce the standard errors to avoid overfitting a regression model
- Apply grid search techniques to Python data

## Model Evaluation

Model evaluation tells us how our model performs in the real world. In the previous module, we talked about in-sample evaluation.

In-sample evaluation tells us how well our model fits the data already given to train it. It does not give us an estimate of how well the train model can predict new data. The solution is to split our data up, use the in-sample data or training data to train the model. The rest of the data, called Test Data, is used as out-of-sample data. This data is then used to approximate, how the model performs in the real world.

Separating data into training and testing sets is an important part of model evaluation. We use the test data to get an idea how our model will perform in the real world. When we split a dataset, usually the larger portion of data is used for training and a smaller part is used for testing.

For example, we can use 70 percent of the data for training. We then use 30 percent for testing. We use training set to build a model and discover predictive relationships. We then use a testing set to evaluate model performance. When we have completed testing our model, we should use all the data to train the model.

A popular function, in the scikit-learn package for splitting datasets, is the train test split function. This function randomly splits a dataset into training and testing subsets. From the example code snippet, this method is imported from `sklearn.cross-validation`.

The input parameters `y_data` is the target variable. In the car appraisal example, it would be the price and `x_data`, the list of predictive variables. In this case, it would be all the other variables in the car dataset that we are using to try to predict the price.

The output is an array. `x_train` and `y_train` the subsets for training. `x_test` and `y_test` the subsets for testing. In this case, the test size is a percentage of the data for the testing set. Here, it is 30 percent. The random state is a random seed for random data set splitting.

Generalization error is a measure of how well our data does at predicting previously unseen data. The error we obtain using our testing data is an approximation of this error. This figure shows the distribution of the actual values in red compared to the predicted values from a linear regression in blue.

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

Using a lot of data for training,gives us an accurate means of determining how well our model will perform in the real world. But the precision of the performance will be low. Let's clarify this with an example. The center of this bull's eye represents the correct generalization error.

Let's say we take a random sample of the data using 90 percent of the data for training and 10 percent for testing.

The first time we experiment, we get a good estimate of the training data. If we experiment again training the model with a different combination of samples, we also get a good result.

But, the results will be different relative to the first time we run the experiment. Repeating the experiment again with a different combination of training and testing samples, the results are relatively close to the generalization error, but distinct from each other.

Repeating the process, we get good approximation of the generalization error, but the precision is poor i.e. all the results are extremely different from one another. If we use fewer data points to train the model and more to test the model, the accuracy of the generalization performance will be less, but the model will have good precision.

The figure above demonstrates this. All our error estimates are relatively close together, but they are further away from the true generalization performance. To overcome this problem, we use cross-validation. One of the most common out of sample evaluation metrics is `cross-validation`.

In this method, the dataset is split into K equal groups. Each group is referred to as a fold. For example, four folds. Some of the folds can be used as a training set which we use to train the model and the remaining parts are used as a test set, which we use to test the model.

For example, we can use three folds for training, then use one fold for testing. This is repeated until each partition is used for both training and testing. At the end, we use the average results as the estimate of out-of-sample error.

The evaluation metric depends on the model, for example, the r squared. The simplest way to apply cross-validation is to call the `cross_val_score` function, which performs multiple out-of-sample evaluations. This method is imported from sklearn's model selection package.

We then use the function `cross_val_score`. The first input parameters, the type of model we are using to do the cross-validation. In this example, we initialize a linear regression model or object `lr` which we passed the `cross_val_score` function.

The other parameters are `x_data`, the predictive variable data, and `y_data`, the target variable data. We can manage the number of partitions with the `cv` parameter.

Here, `cv` equals three, which means the data set is split into three equal partitions. The function returns an array of scores, one for each partition that was chosen as the testing set. We can average the result together to estimate out of sample r squared using the mean function `Numpy`.

Let's see an animation, let's see the result of the score array in the last slide. First, we split the data into three folds. We use two folds for training, the remaining fold for testing. The model will produce an output. We will use the output to calculate a score. In the case of the r squared i.e. coefficient of determination, we will store that value in an array. We will repeat the process using two folds for training and one fold for testing.

Save the score, then use a different combination for training and the remaining fold for testing. We store the final result. The `cross_val_score` function returns a score value to tell us the cross-validation result. What if we want a little more information?

What if we want to know the actual predicted values supplied by our model before the r squared values are calculated? To do this, we use the `cross_val_predict` function. The input parameters are exactly the same as the `cross_val_score` function, but the output is a prediction. Let's illustrate the process. First, we split the data into three folds. We use two folds for training, the remaining fold for testing. The model will produce an output, and we will store it in an array. We will repeat the process using two folds for training, one for testing. The model produces an output again. Finally, we use the last two folds for training.

Then we use the testing data. This final testing fold produces an output. These predictions are stored in an array.

## Model Evaluation

Model evaluation tells us how our model performs in the real world.

# 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

In the previous module, we talked about in-sample evaluation. In-sample evaluation tells us how well our model fits the data already given to train it. It does not give us an estimate of how well the train model can predict new data. The solution is to split our data up, use the in-sample data or training data to train the model. The rest of the data, called Test Data, is used as out-of-sample data. This data is then used to

approximate, how the model performs in the real world. The rest of the data, called Test Data, is used as out-of-sample data. This data is then used to approximate, how the model performs in the real world.

## 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

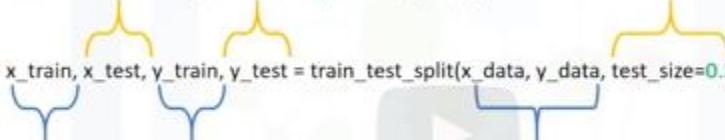
Separating data into training and testing sets is an important part of model evaluation. We use the test data to get an idea how our model will perform in the real world. When we split a dataset, usually the larger portion of data is used for training and a smaller part is used for testing. For example, we can use 70 percent of the data for training. We then use 30 percent for testing. We use training set to build a model and discover predictive relationships. We then use a testing set to evaluate model performance.

## To Split the Data

### Function `train_test_split()`

- Split data into random train and test subsets

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



- `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%)

A popular function, in the scikit-learn package for splitting datasets, is the `train_test_split` function. This function randomly splits a dataset into training and testing subsets. From the example code snippet, this method is imported from `sklearn.cross-validation`. The input parameters `y_data` is the target variable. In the car appraisal example, it would be the price and `x_data`, the list of predictive variables. In this case, it would be all the other variables in

the car dataset that we are using to try to predict the price. The output is an array. `x_train` and `y_train` the subsets for

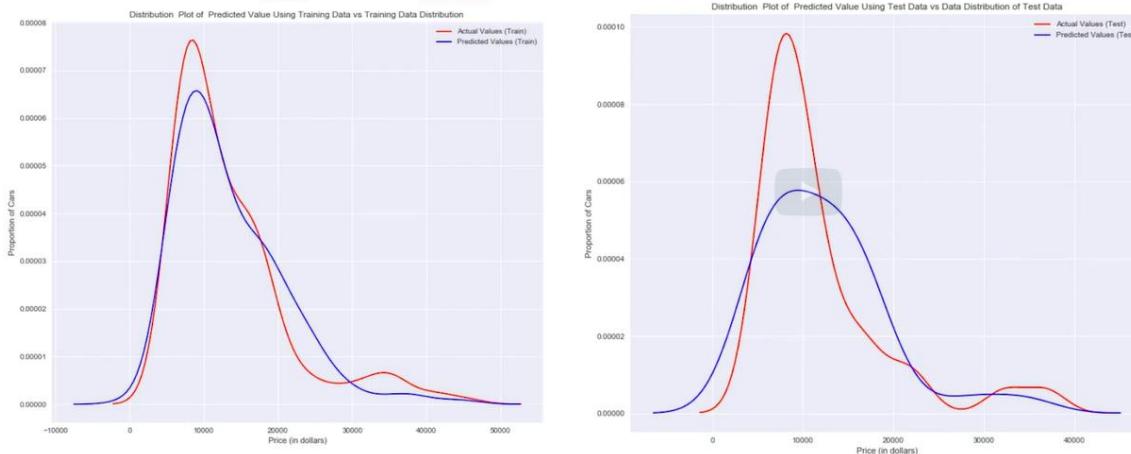
training. `x_test` and `y_test` the subsets for testing. In this case, the test size is a percentage of the data for the testing set. Here, it is 30 percent. The random state is a random seed for random data set splitting.

## Generalization Performance

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

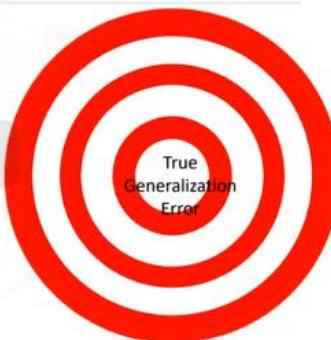
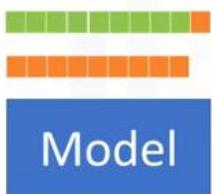
The error we obtain using our testing data is an approximation of this error. This figure shows the distribution of the actual values in

## Generalization Error



This figure shows the distribution of the actual values in red compared to the predicted values from a linear regression in blue. We see the distributions are somewhat similar. If we generate the same plot using the test data, we see the distributions are relatively different. The difference is due to a generalization error and represents what we see in the real world.

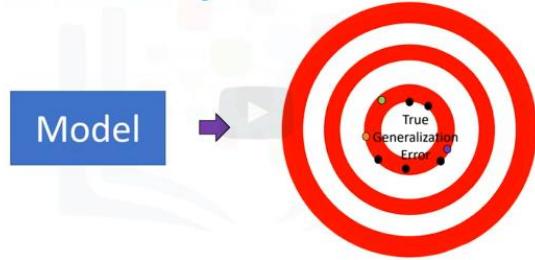
## Lots of Training Data



Using a lot of data for training gives us an accurate means of determining how well our model will perform in the real world. But the precision of the performance will be low. Let's clarify this with an example. The center of this bull's eye represents the correct generalization error. Let's say we take a random sample of the data using 90 percent of the data for training and 10 percent for testing.

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

## Lots of Training Data



Repeating the experiment again with a different combination of training and testing samples, the results are relatively close to the generalization error, but distinct from each other. Repeating the process, we get good approximation of the generalization error, but the precision is poor i.e. all the results are extremely different from one another.

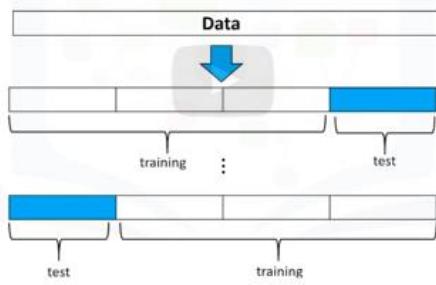
## Lots of Training Data



If we use fewer data points to train the model and more to test the model, the accuracy of the generalization performance will be less, but the model will have good precision. The figure above demonstrates this. All our error estimates are relatively close together, but they are further away from the true generalization performance.

## Cross Validation

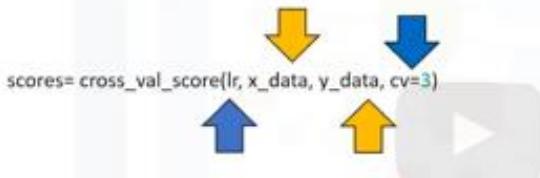
- Most common out-of-sample evaluation metrics
- More effective use of data (each observation is used for both training and testing)



To overcome this problem, we use **cross-validation**. One of the most common out of sample evaluation metrics is cross-validation. In this method, the dataset is split into K equal groups. Each group is referred to as a fold. For example, four folds. Some of the folds can be used as a training set which we use to train the model and the remaining parts are used as a test set, which we use to test the model. For example, we can use three folds for training, then use one fold for testing. This is repeated until each partition is used for both training and testing. At the end, we use the average results as the estimate of out-of-sample error. The evaluation metric depends on the model, for example, the r squared.

## Function cross\_val\_score()

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



The simplest way to apply cross-validation is to call the `cross_val_score` function, which performs multiple out-of-sample evaluations. This method is imported from `sklearn's model selection package`. We then use the function `cross_val_score`. The first input parameters, the type of model we are using to do the cross-validation. In this example, we initialize a linear regression model or object `lr` which we passed the `cross_val_score` function. The other parameters are `x_data`, the predictive variable data, and `y_data`, the target variable data. We can

manage the number of partitions with the `cv` parameter. Here, `cv` equals three, which means the data set is split into three equal partitions. The function returns an array of scores, one for each partition that was chosen as the testing set. We can average the result together to estimate out of sample r squared using the mean function `NnumPi`.

## Function cross\_val\_score()



Let's see the result of the score array in the last slide. First, we split the data into three folds. We use two folds for training, the remaining fold for testing. The model will produce an output. We will use the output to calculate a score. In the case of the r squared i.e. coefficient of determination, we will store that value in an array. We will repeat the process using two folds for training and one fold for testing. Save the score, then use a different

combination for training and the remaining fold for testing. We store the final result. The `cross_val_score` function returns a score value to tell us the cross-validation result.

## 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()`

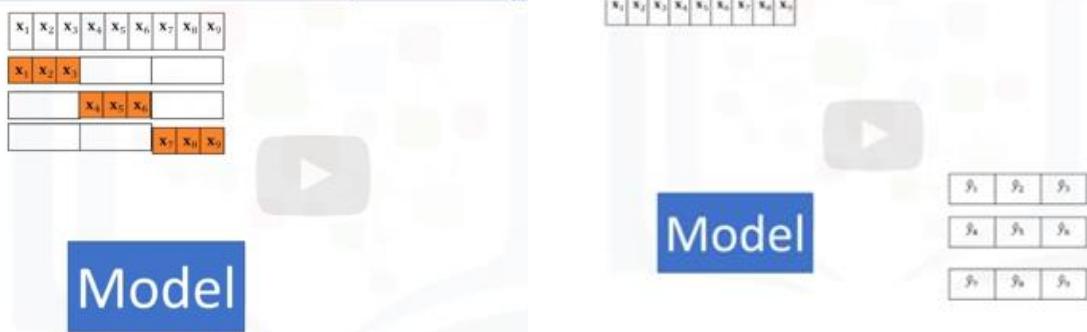
```
from sklearn.model_selection import cross_val_predict
```

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

output is a prediction.

The `cross_val_score` function returns a score value to tell us the cross-validation result. What if we want a little more information? What if we want to know the actual predicted values supplied by our model before the r squared values are calculated? To do this, we use the `cross_val_predict` function. The input parameters are exactly the same as the `cross_val_score` function, but the

## Function cross\_val\_predict()



produce an output, and we will store it in an array. We will repeat the process using two folds for training, one for testing. The model produces an output again. Finally, we use the last two folds for training. Then we use the testing data. This final testing fold produces an output. These predictions are stored in an array.

First, we split the data into three folds.

We use two folds for training,

the remaining fold for testing.

The model will

### Practice Quiz: Model Evaluation

Bookmarked

#### Question 1

1/1 point (ungraded)

What function randomly splits a dataset into training and testing subsets

cross\_val\_predict

cross\_val\_score

train\_test\_split



#### Answer

Correct: Correct

Submit

You have used 2 of 2 attempts

✓ Correct (1/1 point)

## Overfitting, Underfitting and Model Selection

If you recall, in the last module, we discussed polynomial regression. In this section, we will discuss how to pick the best polynomial order and problems that arise when selecting the wrong order polynomial.

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

If we try and fit the function with a linear function, the line is not complex enough to fit the data. As a result, there are many errors. This is called underfitting, where the model is too simple to fit the data. If we increase the order of the polynomial, the model fits better, but the model is still not flexible enough and exhibits underfitting.

This is an example of the 8th order polynomial used to fit the data. We see the model does well at fitting the data and estimating the function even at the inflection points. Increasing it to a 16th order polynomial, the model does extremely well at tracking the training point but performs poorly at estimating the function.

This is especially apparent where there is little training data. The estimated function oscillates not tracking the function. This is called overfitting, where the model is too flexible and fits the noise rather than the function.

Let's look at a plot of the mean square error for the training and testing set of different order polynomials. The horizontal axis represents the order of the polynomial. The vertical axis is the mean square error. The training error decreases with the order of the polynomial. The test error is a better means of estimating the error of a polynomial. The error decreases 'till the best order of the polynomial is determined. Then the error begins to increase.

We select the order that minimizes the test error. In this case, it was eight. Anything on the left would be considered underfitting. Anything on the right is overfitting. If we select the best order of the polynomial, we will still have some errors. If you recall the original expression for the training points we see a noise term. This term is one reason for the error. This is because the noise is random, and we can't predict it. This is sometimes referred to as an irreducible error.

There are other sources of errors as well. For example, our polynomial assumption may be wrong. Our sample points may have come from a different function. For example, in this plot, the data is generated from a sine wave. The polynomial function does not do a good job at fitting the sine wave. For real data, the model may be too difficult to fit or we may not have the correct type of data to estimate the function.

Let's try different order polynomials on the real data using horsepower. The red points represent the training data. The green points represent the test data. If we just use the mean of the data, our model does not perform well.

A linear function does fit the data better. A second order model looks similar to the linear function. A third order function also appears to increase, like the previous two orders. Here, we see a fourth order polynomial. At around 200 horsepower, the predicted price suddenly decreases. This seems erroneous.

Let's use R-squared to see if our assumption is correct. The following is a plot of the R-squared value. The horizontal axis represents the order polynomial models. The closer the R-squared is to one, the more accurate the model is.

Here, we see the R-squared is optimal when the order of the polynomial is three. The R-squared drastically decreases when the order is increased to four, validating our initial assumption. We can calculate different R-squared values as follows.

First, we create an empty list to store the values. We create a list containing different polynomial orders. We then iterate through the list using a loop. We create a polynomial feature object with the order of the polynomial as a parameter. We transform the training and test data into a polynomial using the fit transform method. We fit the regression model using the transform data. We then calculate the R-squared using the test data and store it in the array.

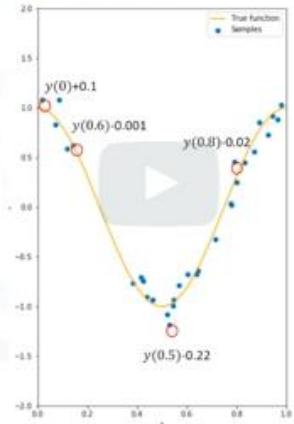
## Overfitting, Underfitting and Model Selection

**Goal: How to pick the best polynomial order and problems that arise when selecting the wrong order polynomial.**

1.

### Model Selection

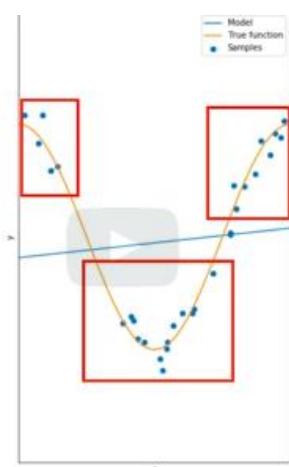
$y(x) + \text{noise}$



The goal of Model Selection is to determine the order of the polynomial to provide the best estimate of the function  $y(x)$ .

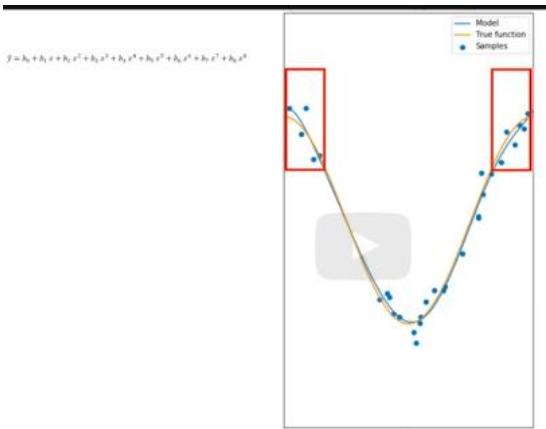
2.

$$y = b_0 + b_1 x$$



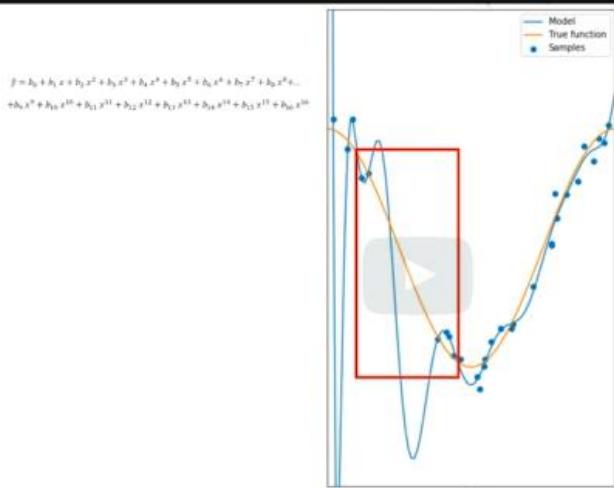
If we try and fit the function with a linear function, the line is not complex enough to fit the data. As a result, there are many errors. This is called **underfitting**, where the model is too simple to fit the data.

3.



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

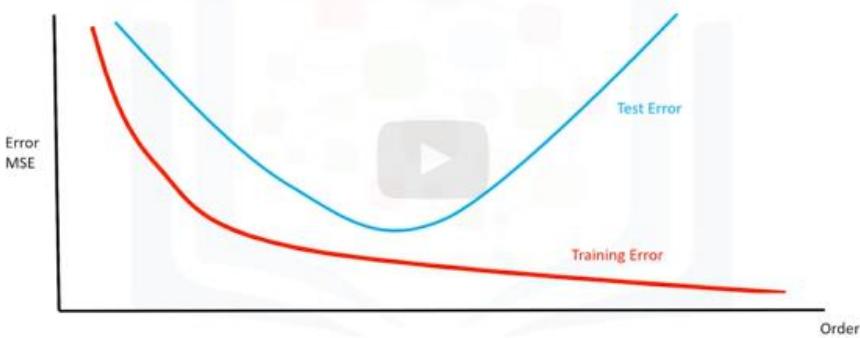
4.



This is an example of the 8th order polynomial used to fit the data. We see the model does well at fitting the data and estimating the function even at the inflection points. Increasing it to a 16th order polynomial, the model does extremely well at tracking the training point but performs poorly at estimating the function. This is especially apparent where there is little training data. The estimated function oscillates not tracking the function. This is called overfitting, where the model is too flexible and fits the noise rather than the function.

5.

## Model Selection

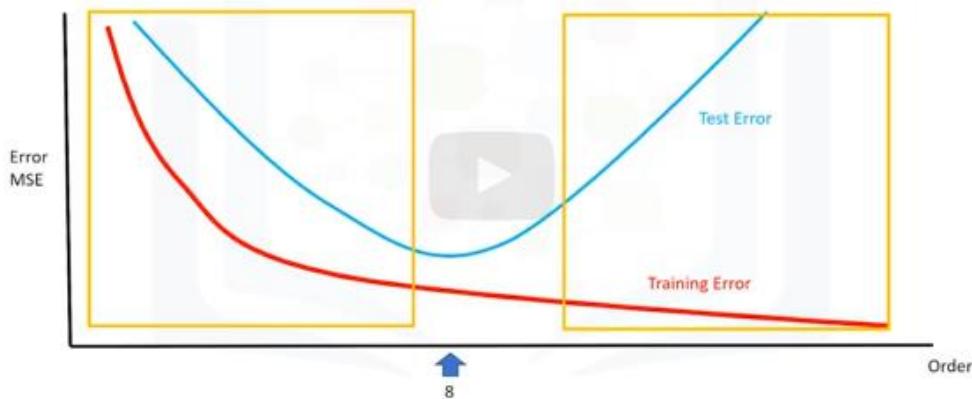


Let's look at a plot of the mean square error for the training and testing set of different order polynomials. The horizontal axis represents the order of the polynomial. The vertical axis is the mean square error.

The training error decreases with the order of the polynomial. The test error is a better means of estimating the error of a polynomial. The error decreases 'till the best order of the polynomial is determined. Then the error begins to increase.

6.

## Model Selection

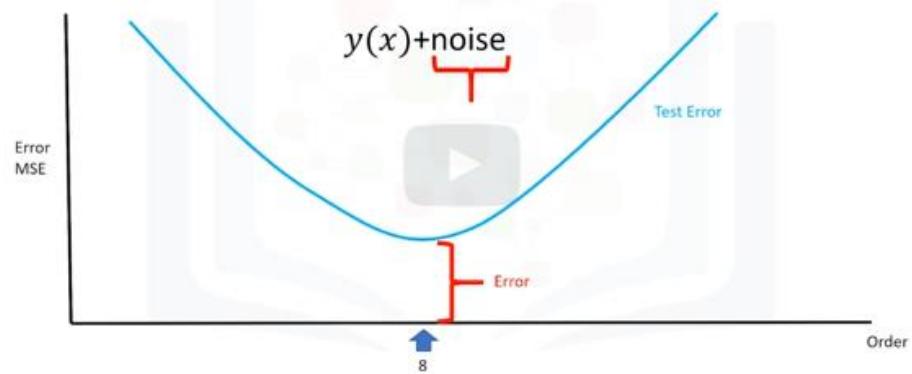


We select the order that minimizes the test error. In this case, it was eight. Anything on the left would be considered underfitting.

Anything on the right is overfitting.

7.

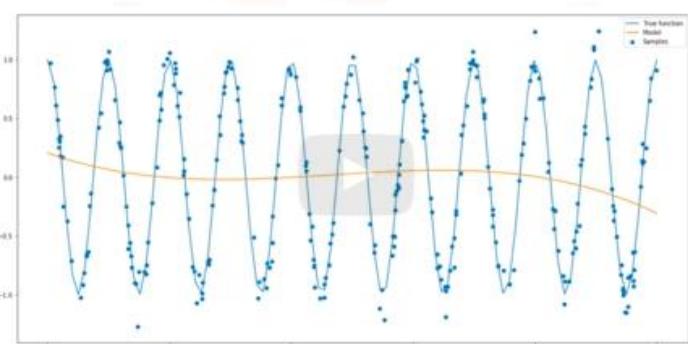
## Model Selection



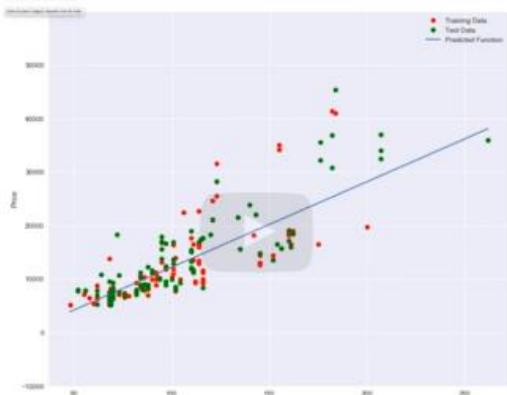
If we select the best order of the polynomial, we will still have some errors. If you recall the original expression for the training points we see a noise term. This term is one reason for the error. This is because the noise is random, and we can't predict it. This is sometimes referred to as an irreducible error. There are other sources of errors as well. For example, our polynomial assumption may be wrong. Our sample points may have come from a different function.

8.

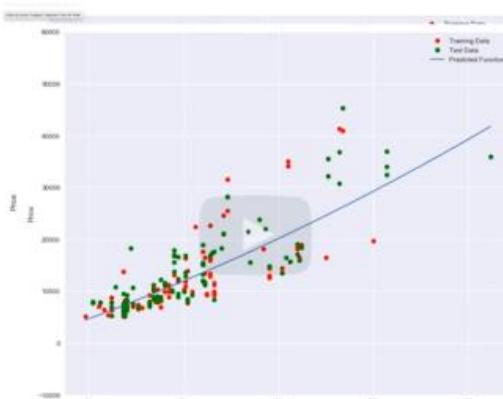
## Model Selection



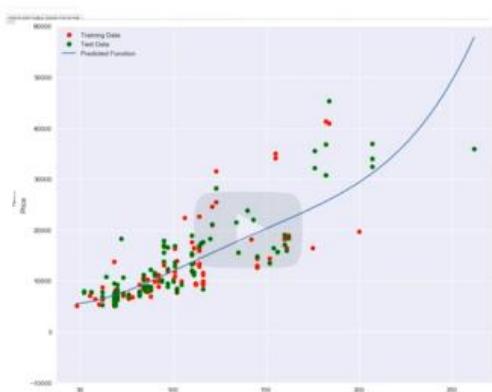
Our sample points may have come from a different function. For example, in this plot, the data is generated from a sine wave. The polynomial function does not do a good job at fitting the sine wave. For real data, the model may be too difficult to fit or we may not have the correct type of data to estimate the function.



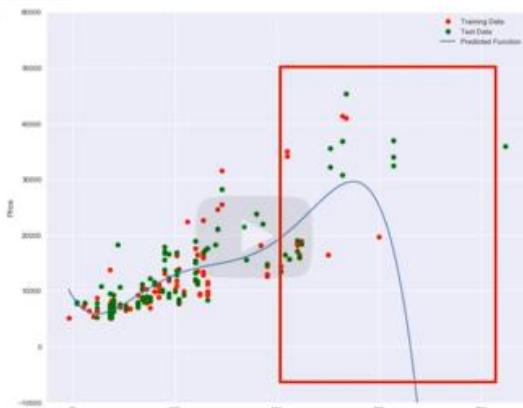
Let's try different order polynomials on the real data using horsepower. The red points represent the training data. The green points represent the test data. If we just use the mean of the data, our model does not perform well. A linear function does fit the data better.



A second order model looks similar to the linear function



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



Here, we see a fourth order polynomial. At around 200 horsepower, the predicted price suddenly decreases. This seems erroneous.

## Model Selection



The closer the R-squared is to one, the more accurate the model is. Here, we see the R-squared is optimal when the order of the polynomial is three. The R-squared drastically decreases when the order is increased to four, validating our initial assumption.

### Calculate different $R^2$ values:

```
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))
```

First, we create an empty list to store the values. We create a list containing different polynomial orders. We then iterate through the list using a loop. We create a polynomial feature object with the order of the polynomial as a parameter. We transform the training and test data into a polynomial using the fit transform method. We fit the regression model using the transform data. We then calculate the R-squared using the test data and store it in the array.

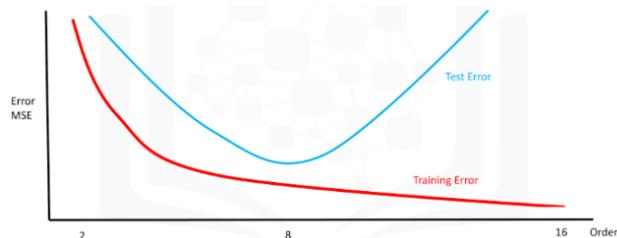
## Practice Quiz: Overfitting, Underfitting and Model Selection

 Bookmarked

### Question 1

1/1 point (ungraded)

In the following plot, the vertical axis shows the mean square error and the horizontal axis represents the order of the polynomial. The red line represents the training error the blue line is the test error. Should you select the 16 order polynomial.



No

Yes



#### Answer

Correct: Correct

## Reading: Ridge Regression Introduction

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

## Ridge Regression

Ridge regression prevents overfitting. We will focus on polynomial regression for visualization, but overfitting is also a big problem when you have multiple independent variables, or features.

Consider the following fourth order polynomial in orange. The blue points are generated from this function. We can use a tenth order polynomial to fit the data. The estimated function in blue does a good job at approximating the true function. In many cases real data has outliers. For example, this point shown here does not appear to come from the function in orange. If we use a tenth order polynomial function to fit the data, the estimated function in blue is incorrect, and is not a good estimate of the actual function in orange.

If we examine the expression for the estimated function, we see the estimated polynomial coefficients have a very large magnitude. This is especially evident for the higher order polynomials.

Ridge regression controls the magnitude of these polynomial coefficients by introducing the parameter alpha. Alpha is a parameter we select before fitting or training the model. Each row in the following table represents an increasing value of alpha.

Let's see how different values of alpha change the model. This table represents the polynomial coefficients for different values of alpha. The column corresponds to the different polynomial coefficients, and the rows correspond to the different values of alpha. As alpha increases, the parameters get smaller. This is most evident for the higher order polynomial features.

But Alpha must be selected carefully. If alpha is too large, the coefficients will approach zero and underfit the data. If alpha is zero, the overfitting is evident. For alpha equal to 0.001, the overfitting begins to subside. For Alpha equal to 0.01, the estimated function tracks the actual function. When alpha equals one, we see the first signs of underfitting. The estimated function does not have enough flexibility. At alpha equals to 10, we see extreme underfitting.

It does not even track the two points. In order to select alpha, we use cross validation. To make a prediction using ridge regression, import ridge from sklearn.linear\_models. Create a ridge object using the constructor. The parameter alpha is one of the arguments of the constructor. We train the model using the fit method. To make a prediction, we use the predict method. In order to determine the parameter alpha, we use some data for training.

We use a second set called validation data. This is similar to test data, but it is used to select parameters like alpha. We start with a small value of alpha. We train the model, make a prediction using the validation data, then calculate the R-squared and store the values. Repeat the value for a larger value of alpha. We train the model again, make a prediction using the validation data, then calculate the R-squared and store the values of R-squared. We repeat the process for a different alpha value, training the model, and making a prediction. We select the value of alpha that maximizes the R-squared.

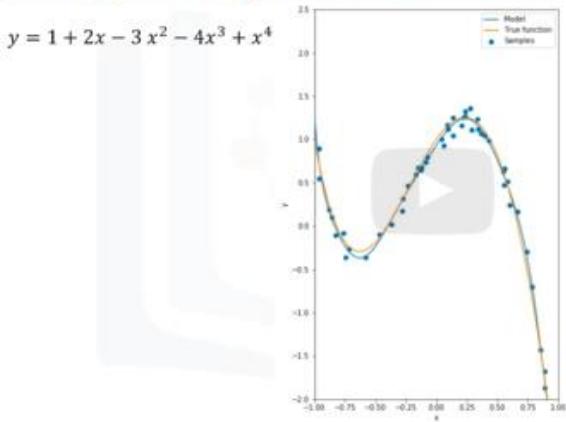
Note that we can use other metrics to select the value of alpha, like mean squared error. The overfitting problem is even worse if we have lots of features. The following plot shows the different values of R-squared on the vertical axis. The horizontal axis represents different values for alpha. We use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

# Ridge Regression



1.

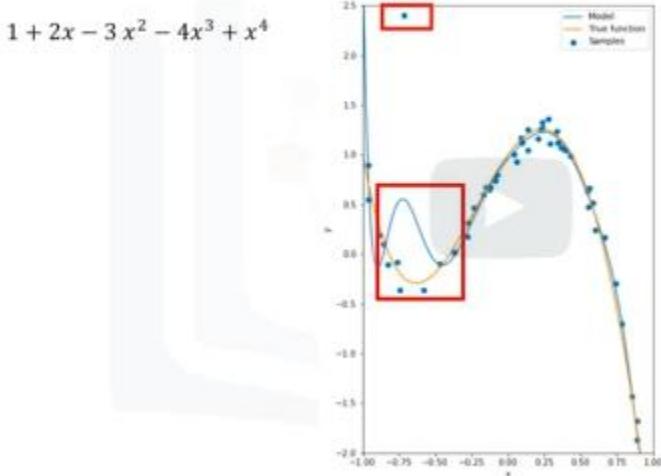
## Ridge Regression



Consider the following fourth order polynomial in orange. The blue points are generated from this function. We can use a tenth order polynomial to fit the data. The estimated function in blue does a good job at approximating the true function.

2.

## Ridge Regression



In many cases real data has outliers. For example, this point shown here does not appear to come from the function in orange. If we use a tenth order polynomial function to fit the data, the estimated function in blue is incorrect, and is not a good estimate of the actual function in orange.

## Ridge Regression

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

examine the expression for the estimated function, we see the estimated polynomial coefficients have a very large magnitude. This is especially evident for the higher order polynomials.

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

# Ridge Regression

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

Alpha	$x$	$x^2$	$x^3$	$x^4$	$x^5$	$x^6$	$x^7$	$x^8$	$x^9$	$x^{10}$
0	2	-3	-2	-12	-40	80	71	-141	-38	75
0.001	2	-3	-7	5	4	-6	4	-4	4	6
0.01	1	-2	-5	-0.04	0.15	-1	1	-0.5	0.3	1
1	0.5	-1	-1	-0.614	0.70	-0.38	-0.56	-0.21	-0.5	-0.1
10	0	-0.5	-0.3	-0.37	-0.30	-0.30	-0.22	-0.22	-0.22	-0.17

Ridge regression controls the magnitude of these polynomial coefficients by introducing the parameter alpha. Alpha is a parameter we select before fitting or training the model. Each row in the following table represents an increasing value of alpha.

Let's see how different values of alpha change the model. This table represents the polynomial coefficients for different values of alpha. The column corresponds to the different polynomial coefficients, and the rows correspond to the different values of alpha. As alpha increases, the parameters get smaller. This is most evident for the higher order polynomial features.

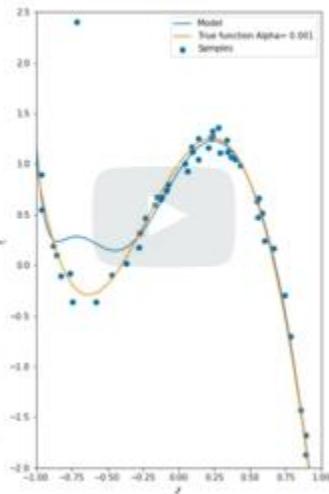
But Alpha must be selected carefully. If alpha is too large, the coefficients will approach zero and underfit the data. If alpha is zero, the overfitting is evident. For alpha equal to 0.001, the overfitting begins to subside. For Alpha equal to 0.01, the estimated function tracks the actual function. When alpha equals one, we see the first signs of underfitting. The estimated function does not have enough flexibility. At alpha equals to 10, we see extreme underfitting.

\*The bigger the alpha, the higher the underfitting

\*As alpha increases, the parameter gets smaller

# Ridge Regression

alpha
0
0.001
0.01
1
10



If alpha is zero, the overfitting is evident. For alpha equal to 0.001, the overfitting begins to subside. For Alpha equal to 0.01, the estimated function tracks the actual function. When alpha equals one, we see the first signs of underfitting. The estimated function does not have enough flexibility. At alpha equals to 10, we see extreme underfitting. It does not even track the two points.

## To Select Alpha

# Ridge Regression

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

In order to select alpha, we use cross validation. To make a prediction using ridge regression, import ridge from sklearn.linear\_models. Create a ridge object using the constructor. The parameter alpha is one of the arguments of the constructor. We train the model using the fit method.

# Ridge Regression



# Ridge Regression



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

## Ridge Regression



## Ridge Regression



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

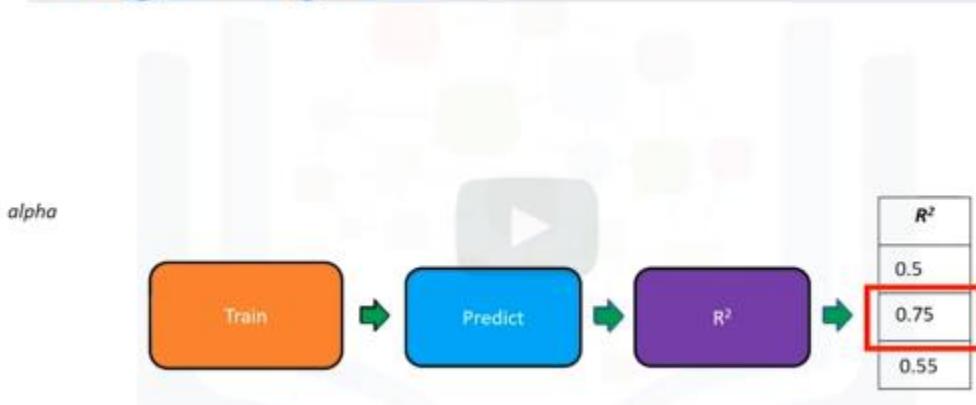
## Ridge Regression



## Ridge Regression

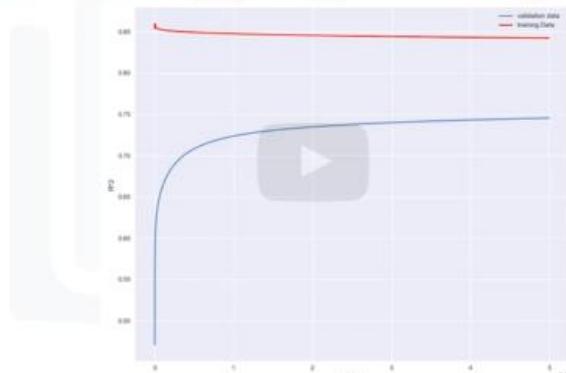


## Ridge Regression



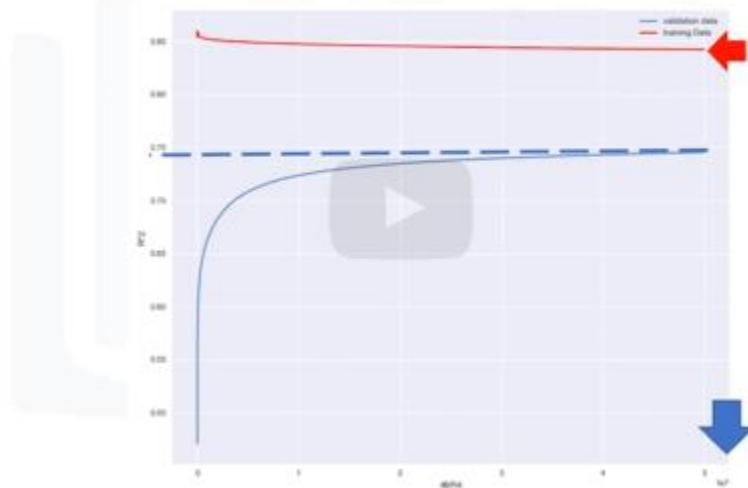
We select the value of alpha that maximizes the R-squared. Note that we can use other metrics to select the value of alpha, like mean squared error. The overfitting problem is even worse if we have lots of features.

## Ridge Regression



The overfitting problem is even worse if we have lots of features. The following plot shows the different values of R-squared on the vertical axis. The horizontal axis represents different values for alpha. We use several features from our used car data set and a second order polynomial function. The training data is in red and validation data is in blue.

## Ridge Regression



We see as the value for alpha increases, the value of R-squared increases and converges at approximately 0.75. In this case, we select the maximum value of alpha because running the experiment for higher values of alpha have little impact. Conversely, as alpha increases, the R-squared on the test data decreases. This is because the term alpha prevents overfitting. This may improve the results in the unseen data, but the model has worse performance on the test data.

The model that exhibits the “most” underfitting is usually the model with highest parameter value for alpha

The higher the alpha, more it is underfitting



**alpha = more underfitting**

The model that exhibits overfitting is usually the model with the lowest parameter value for alpha



**alpha = more overfitting**

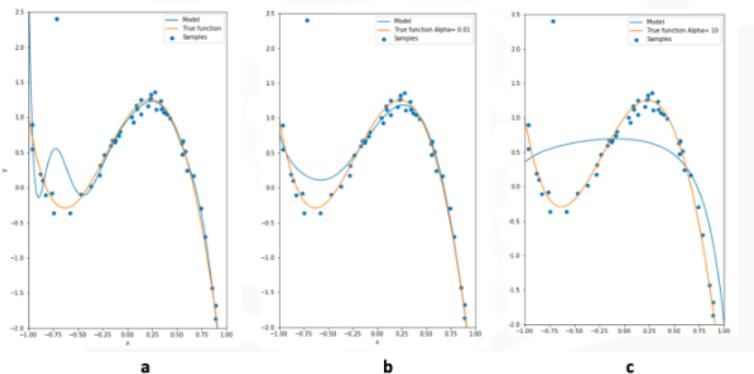
## Practice Quiz: Ridge Regression

Bookmarked

### Question 1

1/1 point (ungraded)

The following models were all trained on the same data. Select the model with the lowest value for alpha:



a

b

c



#### Answer

Correct: Correct

*The model that exhibits the "most" underfitting is usually the model with the highest parameter value for alpha. c -*

*The model that exhibits overfitting is usually the model with the lowest parameter value for alpha*

The model that exhibits the “most” underfitting is usually the model with highest parameter value for alpha

The higher the alpha, more it is underfitting



**alpha = more underfitting**

The model that exhibits overfitting is usually the model with the lowest parameter value for alpha



**alpha = more overfitting**

## Grid Search

Grid Search allows us to scan through multiple free parameters with few lines of code. Parameters like the alpha term discussed in the previous video are not part of the fitting or training process. These values are called hyperparameters.

Scikit-learn has a means of automatically iterating over these hyperparameters using cross-validation. This method is called Grid Search.

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

Let the small circles represent different hyperparameters. We start off with one value for hyperparameters and train the model. We use different hyperparameters to train the model. We continue the process until we have exhausted the different free parameter values.

Each model produces an error. We select the hyperparameter that minimizes the error. To select the hyperparameter, we split our dataset into three parts, the training set, validation set, and test set. We train the model for different hyperparameters.

We use the R-squared or mean square error for each model. We select the hyperparameter that minimizes the mean squared error or maximizes the R-squared on the validation set. We finally test our model performance using the test data. This is the scikit-learn web page, where the object constructor parameters are given. It should be noted that the attributes of an object are also called parameters. We will not make the distinction even though some of the options are not hyperparameters per se.

In this module, we will focus on the hyperparameter alpha and the normalization parameter. The value of your Grid Search is a Python list that contains a Python dictionary. The key is the name of the free parameter. The value of the dictionary is the different values of the free parameter. This can be viewed as a table with various free parameter values.

We also have the object or model. The Grid Search takes on the scoring method. In this case, R-squared the number of folds, the model or object, and the free parameter values. Some of the outputs include the different scores for different free parameter values.

In this case, the R-squared along with a free parameter values that have the best score. First, we import the libraries we need, including GridSearchCV, the dictionary of parameter values. We create a ridge regression object or model. We then create a GridSearchCV object. The inputs are the ridge regression object, the parameter values, and the number of folds.

We will use R-squared. This is the default scoring method. We fit the object. We can find the best values for the free parameters using the attribute best estimator. We can also get information like the mean score on the validation data using the attribute CV result. What are the advantages of Grid Search is how quickly we can test multiple parameters.

For example, ridge regression has the option to normalize the data. To see how to standardize, see module four. The term alpha is the first element in the dictionary. The second element is the normalized option. The key is the name of the parameter. The value is the different options in this case because we can either normalize the data or not.

The values are True or False respectively. The dictionary is a table or grid that contains two different values. As before, we need the ridge regression object or model. The procedure is similar except that we have a table or grid of different parameter values. The output is the score for all the different combinations of parameter values. The code is also similar. The dictionary contains the different free parameter values. We can find the best value for the free parameters.

The resulting scores of the different free parameters are stored in this dictionary, Grid1.cv\_results\_.

We can print out the score for the different free parameter values. The parameter values are stored as shown here.

Grid Search allows us to scan through multiple free parameters with few lines of code. Parameters like the alpha term discussed in the previous video are not part of the fitting or training process. These values are called hyperparameters.

Scikit-learn has a means of automatically iterating over these hyperparameters using cross-validation. This method is called Grid Search. Grid Search takes the model or objects you would like to train and different values of the hyperparameters.

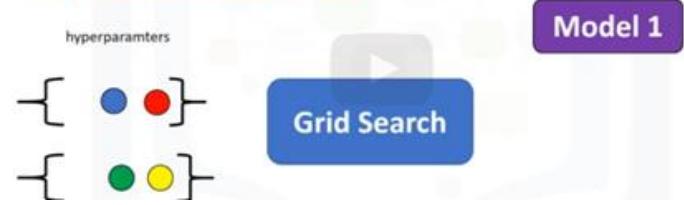
## Hyperparameters

- In the last section, the term alpha in Ridge regression is called a hyperparameter
- Scikit-learn has a means of automatically iterating over these hyperparameters using cross-validation called Grid Search



## Hyperparameters

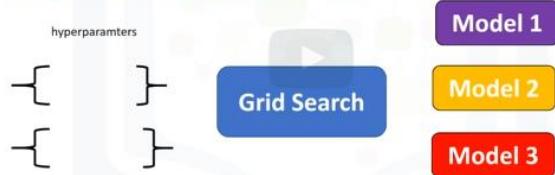
- In the last section, the term alpha in Ridge regression is called a hyperparameter
- Scikit-learn has a means of automatically iterating over these hyperparameters using cross-validation called Grid Search



Let the small circles represent different hyperparameters. We start off with one value for hyperparameters and train the model. We use different hyperparameters to train the model. We continue the process until we have exhausted the different free parameter values.

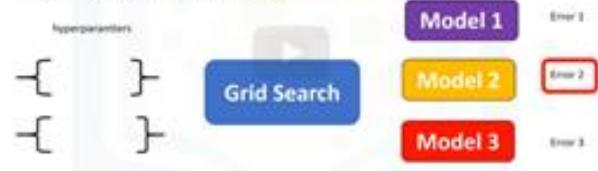
## Hyperparameters

- In the last section, the term alpha in Ridge regression is called a hyperparameter
- Scikit-learn has a means of automatically iterating over these hyperparameters using cross-validation called Grid Search



## Hyperparameters

- In the last section, the term alpha in Ridge regression is called a hyperparameter
- Scikit-learn has a means of automatically iterating over these hyperparameters using cross-validation called Grid Search

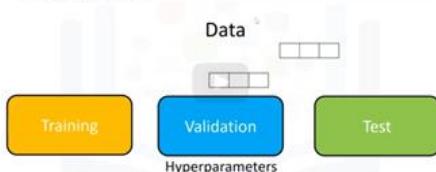


It then calculates the mean square error or R-squared for various hyperparameter values, allowing you to choose the best values. Each model produces an error. We select the hyperparameter that minimizes the error.

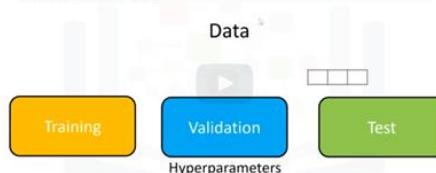
## Grid Search



## Grid Search



## Grid Search



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

The screenshot shows the scikit-learn documentation page for the `sklearn.linear_model.Ridge` class. The URL is [http://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.Ridge.html](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html). The page includes the class definition:

```
class sklearn.linear_model.Ridge(alpha=1.0, fit_intercept=True, normalize=False, copy_X=True, max_iter=None,
tol=0.001, solver='auto', random_state=None)
```

Below the code, there is a detailed description of the Ridge regression model. It highlights the `alpha` parameter as a regularization strength. The `alpha` parameter is described as a positive float that improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to  $C^{-1}$  in other linear models such as LogisticRegression or LinearSVC. If an array is passed, penalties are assumed to be specific to the targets. Hence they must correspond in number.

The `fit_intercept` parameter is described as a boolean indicating whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

The `normalize` parameter is described as a boolean indicating whether the regressors X will be normalized before regression by subtracting the mean and dividing by the 2-norm. If you want to standardize, please use `sklearn.preprocessing.StandardScaler` before calling `fit` on an estimator with `normalize` set to true.

The `copy_X` parameter is described as a boolean indicating whether X will be copied; else, it may be overwritten.

This is the scikit-learn web page, where the object constructor parameters are given. It should be noted that the attributes of an object are also called parameters. We will not make the distinction even though some of the options are not hyperparameters per se. We will focus on the hyperparameter alpha and the normalization parameter.

## Grid Search

```
parameters = [{ 'alpha': [1, 10, 100, 1000] }]
```

Alpha	1	10	100	1000
-------	---	----	-----	------

Ridge()

The value of your Grid Search is a Python list that contains a Python dictionary. The key is the name of the free parameter. The value of the dictionary is the different values of the free parameter. This can be viewed as a table with various free parameter values.

## Grid Search



## Grid Search



The Grid Search takes on the scoring method. In this case, R-squared the number of folds, the model or object, and the free parameter values. Some of the outputs include the different scores for different free parameter values.

## Grid Search



Some of the outputs include the different scores for different free parameter values. In this case, the R-squared along with a free parameter values that have the best score.

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

parameters1=[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 1000000]}]

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']

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

First, we import the libraries we need, including GridSearchCV, the dictionary of parameter values. We create a ridge regression object or model. We then create a GridSearchCV object. The inputs are the ridge regression object, the parameter values, and the number of folds. We will use R-squared. This is the default scoring method. We fit the object. We can find the best values for the free parameters using the attribute best estimator. We can also get info like the mean score on the validation data using the attribute CV result.

## Grid Search

parameters = [{ 'alpha': [1, 10, 100, 1000], 'normalize': [True, False] }]

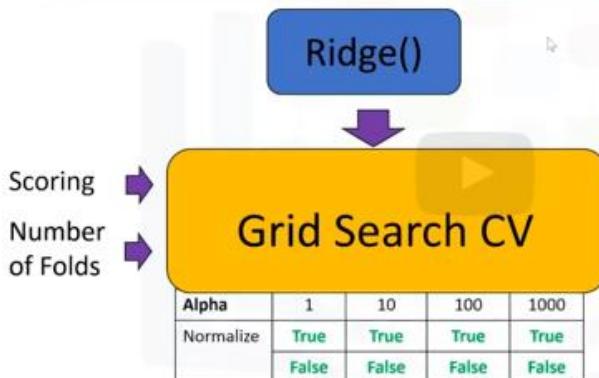
Alpha	1	10	100	1000
Normalize	True	True	True	True
	False	False	False	False

Ridge()

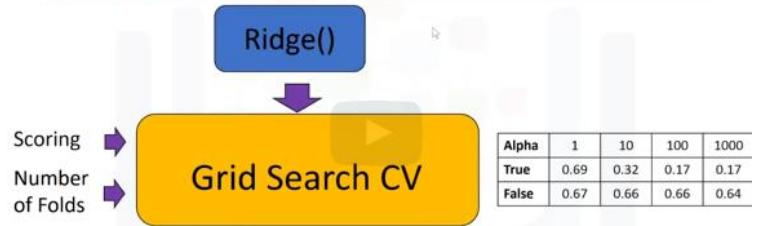
What are the advantages of Grid Search is how quickly we can test multiple parameters. For example, ridge regression has the option to normalize the data. To see how to standardize, see module four. The term alpha is the first element in the dictionary. The second element is the normalized option. The key is the name of the parameter. The value is the different options in this case because we can either

normalize the data or not. The values are True or False respectively. The dictionary is a table or grid that contains two different values. As before, we need the ridge regression object or model.

## Grid Search



## Grid Search



The procedure is similar except that we have a table or grid of different parameter values. The output is the score for all the different combinations of parameter values.

```

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

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

RR=Ridge()

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

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

Grid1.best_estimator_

scores = Grid1.cv_results_
    
```

```

for param, mean_val, mean_test in zip(scores['params'], scores['mean_test_score'], scores['mean_train_score']):
    print(param, "R^2 on test data:", mean_val, "R^2 on train data:", mean_test)

('alpha': 0.001, 'normalize': True) R^2 on test data: 0.646605547293 R^2 on train data: 0.814001968709
('alpha': 0.001, 'normalize': False) R^2 on test data: 0.465488366584 R^2 on train data: 0.8140024998797
('alpha': 0.1, 'normalize': True) R^2 on test data: 0.694179425356 R^2 on train data: 0.8134027698794
('alpha': 0.1, 'normalize': False) R^2 on test data: 0.665488237796 R^2 on train data: 0.814002698794
('alpha': 1, 'normalize': True) R^2 on test data: 0.690486934584 R^2 on train data: 0.7491044440368
('alpha': 1, 'normalize': False) R^2 on test data: 0.665494127178 R^2 on train data: 0.814002698472
('alpha': 10, 'normalize': True) R^2 on test data: 0.321376875232 R^2 on train data: 0.341856042902
('alpha': 10, 'normalize': False) R^2 on test data: 0.665545680812 R^2 on train data: 0.81400266666
('alpha': 100, 'normalize': True) R^2 on test data: 0.0170551710263 R^2 on train data: 0.0496044796826
('alpha': 100, 'normalize': False) R^2 on test data: 0.666029359996 R^2 on train data: 0.8139997198151
('alpha': 1000, 'normalize': True) R^2 on test data: -0.0301961745066 R^2 on train data: 0.005184451599
('alpha': 1000, 'normalize': False) R^2 on test data: 0.666968215369 R^2 on train data: 0.813870488264
('alpha': 10000, 'normalize': True) R^2 on test data: -0.0351687400461 R^2 on train data: 0.000520784757979
('alpha': 10000, 'normalize': False) R^2 on test data: 0.673346350541 R^2 on train data: 0.81258374302
('alpha': 100000, 'normalize': True) R^2 on test data: -0.035666844528 R^2 on train data: 0.2101975528e-05
('alpha': 100000, 'normalize': False) R^2 on test data: 0.67818839432 R^2 on train data: 0.799541446486
('alpha': 1000000, 'normalize': True) R^2 on test data: -0.03566685844558 R^2 on train data: 5.2101975528e-05
('alpha': 1000000, 'normalize': False) R^2 on test data: 0.657818839432 R^2 on train data: 0.789541446486
    
```

The code is also similar. The dictionary contains the different free parameter values. We can find the best value for the free parameters. The resulting scores of the different free parameters are stored in this dictionary, Grid1.cv\_results\_.

We can print out the score for the different free parameter values. The parameter values are stored as shown here.

## Lesson Summary

In this lesson, you have learned how to:

**Identify over-fitting and under-fitting in a predictive model:** Overfitting occurs when a function is too closely fit to the training data points and captures the noise of the data. Underfitting refers to a model that can't model the training data or capture the trend of the data.

**Apply Ridge Regression to linear regression models:** Ridge regression is a regression that is employed in a Multiple regression model when Multicollinearity occurs.

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

# 7.5. Model Evaluation

Seongjoo Brenden Song edited this page on Nov 6, 2021 · 2 revisions

---

## Learning Objectives

- Describe data model refinement techniques
  - Explain overfitting, underfitting and model selection
  - Apply ridge regression to regularize and reduce the standard errors to avoid overfitting a regression model
  - Apply grid search techniques to Python data
- 

- [Model Evaluation and Refinement](#)
- [Lab 5: Model Evaluation and Refinement](#)

## 7.5.1. Model Evaluation and Refinement

### 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

- 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,
```

- **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 user for random sampling

### Generalization Performance

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

## Generalization Error

### 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)
```

### Question

consider the following lines of code, how many partitions or folds are used in the function `cross_val_score`:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(lr, x_data, y_data, cv=10)
```

- 4
- 10
- 5

*Correct*

## Practice Quiz: Model Evaluation

---

### Question 1

What is the correct use of the "train\_test\_split" function such that 90% of the data samples will be utilized for training, the parameter "random\_state" is set to zero, and the input variables for the features and targets are `x_data`, `y_data` respectively.

- `~~train_test_split(x_data, y_data, test_size=0.9, random_state=0)~~`
- `**train_test_split(x_data, y_data, test_size=0.1, random_state=0)**`

*Correct*

## Overfitting, Underfitting and Model Selection

---

### Model Selection

$$y(x) + \text{noise}$$

$$y = b_0 + b_1x$$

$$y = b_0 + b_1x + b_2x^2$$

$$\hat{y} = b_0 + b_1x + b_2x^2 + b_3x^3 + b_4x^4 + b_5x^5 + b_6x^6 + b_7x^7 + b_8x^8$$

$$\hat{y} = b_0 + b_1x + b_2x^2 + b_3x^3 + b_4x^4 + b_5x^5 + b_6x^6 + b_7x^7 + b_8x^8 + b_9x^9 + b_{10}x^{10} + b_{11}x^{11} + b_{12}x^{12} + b_{13}x^{13} + b_{14}x^{14} + b_{15}x^{15} + b_{16}x^{16}$$

## Model Selection

- We select the order that minimizes the test error. In this case, it was **eight**.
- Anything on the **left** would be considered **underfitting**. Anything on the **right** is **overfitting**.

### Question

True or False, the following plot shows that as the order of the polynomial increases the mean square error of our model decreases on the test data:

- **False**
- **True**

*Correct. This plot shows the training error*

## Practice Quiz: Overfitting, Underfitting and Model Selection

---

TOTAL POINTS 1

### Question 1

In the following plot, the vertical axis shows the mean square error and the horizontal axis represents the order of the polynomial. The red line represents the training error the blue line is the test error. Should you select the 16 order polynomial.

- **no**
- **yes**

*Correct. We use the test error to determine the model error. For this order of the polynomial, the training error is smaller but the test error is larger.*

## Ridge Regression

Ridge regression is a regression that is employed in a Multiple regression model when Multicollinearity occurs. Multicollinearity is when there is a strong relationship among the independent variables. Ridge regression is very common with polynomial regression.

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

- Overfitting is a big problem when you have multiple independent variables, or features.
- The estimated function in blue does a good job at approximating the true function.

- In many cases real data has outliers. For example, the blue point shown above does not appear to come from the function in orange. If we use a tenth order polynomial function to fit the data, the estimated function in blue is incorrect, and is not a good estimate of the actual function in orange.

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

- If we examine the expression for the estimated function, we see the estimated polynomial coefficients have a very large magnitude. This is especially evident for the higher order polynomials.
- Ridge regression controls the magnitude of these polynomial coefficients by introducing the parameter alpha. Alpha is a parameter we select before fitting or training the model. Each row in the following table represents an increasing value of alpha. Let's see how different values of alpha change the model. This table represents the polynomial coefficients for different values of alpha.
- The column corresponds to the different polynomial coefficients, and the **rows** correspond to the **different values of alpha**. As alpha increases, the parameters get smaller. This is most evident for the higher order polynomial features. But Alpha must be selected carefully. If alpha is too large, the coefficients will approach zero and underfit the data. If alpha is zero, the overfitting is evident.
- For alpha equal to 0.001, the overfitting begins to subside. For Alpha equal to 0.01, the estimated function tracks the actual function. When alpha equals one, we see the first signs of underfitting. The estimated function does not have enough flexibility. At alpha equals to 10, we see extreme underfitting. It does not even track the two points. In order to select alpha, we use cross validation.

## Question

Consider the following fourth order polynomial, fitted with Ridge Regression; should we increase or decrease the parameter alpha?

- Decrease
- Increase

*Correct. The model seems to be underfitting the data we should decrease the value of the parameter.*

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

- To make a prediction using ridge regression, import ridge from sklearn.linear\_models. Create a ridge object using the constructor. The parameter alpha is one of the arguments of the constructor. We train the model using the fit method. To make a prediction, we use the predict method.
  - The overfitting problem is even worse if we have lots of features. The following plot shows the different values of R-squared on the vertical axis.
- The horizontal axis represents different values for alpha. We use several features from our used car data set and a second order polynomial function. The training data is in red and validation data is in blue. We see as the value for alpha increases, the value of R-squared increases and converges at approximately 0.75.
- In this case, we select the maximum value of alpha because running the experiment for higher values of alpha have little impact. Conversely, as alpha increases, the R-squared on the test data decreases. This is because the term alpha prevents overfitting. This may improve the results in the unseen data, but the model has worse performance on the test data.

## Practice Quiz: Ridge Regression

---

TOTAL POINTS 1

### Question 1

the following models were all trained on the same data, select the model with the highest value for alpha:

- a
- b
- c

*Correct. a, b - The model that exhibits the "most" underfitting is usually the model with the highest parameter value for alpha. c - The model that exhibits overfitting is usually the model with the lowest parameter value for alpha*

## Grid Search

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

### Hyperparameters

- The term alpha in Ridge regression is called a hyperparameter
- Scikit-learn has a means of automatically iterating over these hyperparameters using cross-validation called Grid Search

## Question

What data do we use to pick the best hyperparameter

- Training data
- Validation data
- Test data

Correct. The training data is used to get the model parameters, not the hyperparameters

```
parameters = [{}{'alpha': [1, 10, 100, 1000]}]
```

The value of your Grid Search is a Python list that contains a Python dictionary.

`**'alpha'**` : The key is the name of the free parameter.

`**[1, 10, 100, 1000]**` : The value of the dictionary is the different values of the free parameter.

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

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

What are the advantages of Grid Search is how quickly we can test multiple parameters.

For example, ridge regression has the option to normalize the data.

```
parameters = [{}{'alpha': [1, 10, 100, 1000], 'normalize': [True, False]}]
```

`**'alpha': [1, 10, 100, 1000]**`: The term alpha is the first element in the dictionary.

`**'normalize':[True, False]**`: The second element is the normalized option.

`**'normalize'**`: The key is the name of the parameter.

**\*\*[True, False]\*\*:** The value is the different options in this case because we can either normalize the data or not. The values are True or False respectively.

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

As before, we need the ridge regression object or model. The procedure is similar except that we have a table or grid of different parameter values. The output is the score for all the different combinations of parameter values. The code is also similar.

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

parameters2 = [{'alpha': [1, 10, 100, 1000], 'normalize': [True, False]}]

RR = Ridge()

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

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

Grid1.best_estimator_

scores = Grid1.cv_results_
```

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

```
for param, mean_val, mean_test in zip(score['params'], scores['mean_test_score'],
                                         print(param, 'R^2 on the test data:', mean_val, 'R^2 on train data:', mean_test)
```

## Question

how many types of parameters does the following dictionary contain:

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

- 2
- 9
- 4

*Correct*

## Lesson Summary

In this lesson, you have learned how to:

**Identify over-fitting and under-fitting in a predictive model:** Overfitting occurs when a function is too closely fit to the training data points and captures the noise of the data. Underfitting refers to a model that can't model the training data or capture the trend of the data.

**Apply Ridge Regression to linear regression models:** Ridge regression is a regression that is employed in a Multiple regression model when Multicollinearity occurs.

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

## 7.5.2.Lab 5: Model Evaluation and Refinement

Seongjoo Brenden Song edited this page on Nov 6, 2021 · 2 revisions

```
**import** pandas **as** pd
**import** numpy **as** np

*# Import clean data*
path ***=*** 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DA0101EN-SkillNetwork/module_5_auto.csv'
df ***=*** pd.read_csv(path)

df.to_csv('module_5_auto.csv')
```

First, let's only use numeric data:

```
df ***=*** df._get_numeric_data()
df.head()
```

Libraries for plotting:

```
***%capture
***!pip install ipywidgets

**from** ipywidgets **import** interact, interactive, fixed, interact_manual
```



# Model Evaluation and Refinement

Estimated time needed: **30** minutes

## Objectives

After completing this lab you will be able to:

- Evaluate and refine prediction models

## Table of Contents

- [Model Evaluation](#)
- [Over-fitting, Under-fitting and Model Selection](#)
- [Ridge Regression](#)
- [Grid Search](#)

This dataset was hosted on IBM Cloud object. Click [HERE](#) for free storage.

```
[ ]: #install specific version of libraries used in lab
# !mamba install pandas==1.3.3 -y
# !mamba install numpy=1.21.2 -y
# !mamba install sklearn=0.20.1 -y
# !mamba install ipywidgets=7.4.2 -y
```

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

# Import clean data.
path = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/module_5_auto.csv'
df = pd.read_csv(path)
```

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

First, let's only use numeric data:

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

Libraries for plotting:

```
[ ]: from ipywidgets import interact, interactive, fixed, interact_manual
```

## Functions for Plotting

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

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

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

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

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

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

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

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

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

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

```
label**=**'Predicted Function')
```

## Part 1: Training and Testing

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

```
y_data ***=** df['price']
```

Drop price data in dataframe **x\_data**:

```
x_data***=**df.drop('price',axis***=**1)
```

Now, we randomly split our data into training and testing data using the function **train\_test\_split**.

```
**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***=**0.10,  
  
print("number of test samples :", x_test.shape[0])  
print("number of training samples:",x_train.shape[0])
```

```
number of test samples : 21  
number of training samples: 180
```

The **test\_size** parameter sets the proportion of data that is split into the testing set. In the above, the testing set is 10% of the total dataset.

### Question #1):

Use the function "train\_test\_split" to split up the dataset such that 40% of the data samples will be utilized for testing. Set the parameter "random\_state" equal to zero. The output of the function should be the following: "x\_train1", "x\_test1", "y\_train1" and "y\_test1".

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

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

Let's import `LinearRegression` from the module `linear_model`.

```
**from** sklearn.linear_model **import** LinearRegression
```

We create a Linear Regression object:

```
lre***=***LinearRegression()
```

We fit the model using the feature "horsepower":

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

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

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

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

```
0.36358755750788263
```

We can see the R^2 is much smaller using the test data compared to the training data.

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

```
0.6619724197515104
```

## Question #2):

**Find the R^2 on the test data using 40% of the dataset for testing.**

```
x_train1, x_test1, y_train1, y_test1 ***=*** train_test_split(x_data, y_data, test_size***=***0.4,
```

```
lre.fit(x_train1[['horsepower']],y_train1)
```

```
lre.score(x_test1[['horsepower']],y_test1)
```

```
0.7139364665406973
```

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

## Cross-Validation Score

Let's import **model\_selection** from the module **cross\_val\_score**.

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

We input the object, the feature ("horsepower"), and the target data (y\_data). The parameter 'cv' determines the number of folds. In this case, it is 4.

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

The default scoring is R^2. Each element in the array has the average R^2 value for the fold:

```
Rcross
```

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

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

```
print("The mean of the folds are", Rcross.mean(),
      "and the standard deviation is" , Rcross.std())
```

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

We can use negative squared error as a score by setting the parameter 'scoring' metric to ''neg\_mean\_squared\_error''.

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

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

### Question #3):

Calculate the average R<sup>2</sup> using two folds, then find the average R<sup>2</sup> for the second fold utilizing the "horsepower" feature:

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

```
0.5166761697127429
```

You can also use the function 'cross\_val\_predict' to predict the output. The function splits up the data into the specified number of folds, with one fold for testing and the other folds are used for training. First, import the function:

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

We input the object, the feature "horsepower", and the target data y\_data. The parameter 'cv' determines the number of folds. In this case, it is 4. We can produce an output:

```
yhat **=** cross_val_predict(lre,x_data[['horsepower']], y_data,cv**=**4)
```

```
yhat[0:5]
```

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

## Part 2: Overfitting, Underfitting and Model Selection

It turns out that the test data, sometimes referred to as the "out of sample data", is a much better measure of how well your model performs in the real world. One reason for this is overfitting.

Let's go over some examples. It turns out these differences are more apparent in Multiple Linear Regression and Polynomial Regression so we will explore overfitting in that context.

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

```
lr ***=** LinearRegression()  
lr.fit(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Prediction using training data:

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

```
array([ 7426.6731551 , 28323.75090803, 14213.38819709, 4052.34146983, 34500.19124244])
```

Prediction using training data:

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

```
array([ 7426.6731551 , 28323.75090803, 14213.38819709, 4052.34146983, 34500.19124244])
```

Prediction using test data:

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

```
array([11349.35089149, 5884.11059106, 11208.6928275 , 6641.07786278, 15565.79920282])
```

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

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

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

```
Title **=** 'Distribution Plot of Predicted Value Using Training Data vs Training Data Distribution'  
DistributionPlot(y_train, yhat_train, "Actual Values (Train)", "Predicted Values (Train)", Title)
```

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

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

```
Title**=**'Distribution Plot of Predicted Value Using Test Data vs Data Distribution of Test Data'  
DistributionPlot(y_test,yhat_test,"Actual Values (Test)","Predicted Values (Test)",Title)
```

Figure 2: Plot of predicted value using the test data compared to the actual values of the test data.

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

```
**from** sklearn.preprocessing **import** PolynomialFeatures
```

## Overfitting

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

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

```
x_train, x_test, y_train, y_test **=** train_test_split(x_data, y_data, test_size**=**0.45, random_state**=**0)
```

We will perform a degree 5 polynomial transformation on the feature '**horsepower**'.

```
pr ***=** PolynomialFeatures(degree***=**5)

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

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

pr
```

```
PolynomialFeatures(degree=5, include_bias=True, interaction_only=False)
```

Now, let's create a Linear Regression model "poly" and train it.

```
poly ***=** LinearRegression()

poly.fit(x_train_pr, y_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

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

```
yhat ***=** poly.predict(x_test_pr)

yhat[0:5]
```

```
array([ 6728.68465468,  7308.01690973, 12213.81302023, 18893.19052853, 19995.88231726])
```

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

```
print("Predicted values:", yhat[0:4])

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

```
Predicted values: [ 6728.68465468  7308.01690973 12213.81302023 18893.19052853]
True values: [ 6295. 10698. 13860. 13499.]
```

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

```
PollyPlot(x_train[['horsepower']], x_test[['horsepower']], y_train, y_test, poly,pr)
```

Figure 3: A polynomial regression model where red dots represent training data, green dots represent test data, and the blue line represents the model prediction.

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

R<sup>2</sup> of the training data:

```
poly.score(x_train_pr, y_train)
```

```
0.556771690250259
```

R<sup>2</sup> of the test data:

```
poly.score(x_test_pr, y_test)
```

```
- 29.871506261647205
```

We see the R<sup>2</sup> for the training data is 0.5567 while the R<sup>2</sup> on the test data was -29.87. The lower the R<sup>2</sup>, the worse the model. A negative R<sup>2</sup> is a sign of overfitting.

Let's see how the R<sup>2</sup> changes on the test data for different order polynomials and then plot the results:

```

Rsqu_test **=** []

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

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

    lr.fit(x_train_pr, y_train)

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

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

```

Text(3, 0.75, 'Maximum R^2 ')

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

The following function will be used in the next section. Please run the cell below.

```

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

```

```

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

```

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

```
interact(f, order**=(0, 6, 1), test_data**=(0.05, 0.95, 0.05))
```

## Output

This XML file does not appear to have any style information associated with it. The document tree is shown below

---

```
▼<Error>
  <Code>AccessDenied</Code>
  <Message>Request has expired</Message>
  <X-Amz-Expires>86400</X-Amz-Expires>
  <Expires>2021-11-06T18:17:16Z</Expires>
  <ServerTime>2022-02-11T00:04:30Z</ServerTime>
  <RequestId>AWJ5Z23V68ZM7RMM</RequestId>
  <HostId>3LGRquy59t9RtkW/Z5RpXx782RcK9vUp98n9+W3+QrNh1Pikoafk+IIS03FdFLRw8bvpqx7oVV0=</HostId>
</Error>
```

## Question #4a):

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

```
pr1**=**PolynomialFeatures(degree**=**2)
```

## Question #4b):

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

```
x_train_pr1**=**pr1.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
x_test_pr1**=**pr1.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
```

## Question #4c):

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

```
x_train_pr1.shape *#there are now 15 features*
```

```
(110, 15)
```

### Question #4d):

Create a linear regression model "poly1". Train the object using the method "fit" using the polynomial feature

```
poly1***=**LinearRegression().fit(x_train_pr1,y_train)
```

```
poly1***=**LinearRegression().fit(x_train_pr1,y_train)
```

### Question #4e):

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

```
yhat_test1***=**poly1.predict(x_test_pr1)

Title***='Distribution Plot of Predicted Value Using Test Data vs Data Distribution of Test Data'

DistributionPlot(y_test, yhat_test1, "Actual Values (Test)", "Predicted Values (Test)", Title)
```

### Question #4f):

Using the distribution plot above, describe (in words) the two regions where the predicted prices are less accurate than the actual prices.

- The predicted value is higher than actual value for cars where the price \$10,000 range, conversely the predicted price is lower than the price cost in the \$30,000 to \$40,000 range. As such the model is not as accurate in these ranges.

## Part 3: Ridge Regression

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

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

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

x_train_pr***=**pr.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-size',

x_test_pr***=**pr.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size',
```

Let's import **Ridge** from the module **linear models**.

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

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

```
RidgeModel***=**Ridge(alpha***=**1)
```

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

```
RidgeModel.fit(x_train_pr, y_train)
```

```
Ridge(alpha=1, copy_X=True, fit_intercept=True, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001)
```

Similarly, you can obtain a prediction:

```
yhat ***=** RidgeModel.predict(x_test_pr)
```

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

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

```
predicted: [ 6570.82441941  9636.2489147  20949.92322737 19403.60313256]  
test set : [ 6295. 10698. 13860. 13499.]
```

We select the value of alpha that minimizes the test error. To do so, we can use a for loop. We have also created a progress bar to see how many iterations we have completed so far.

```
**from** tqdm **import** tqdm  
  
Rsqu_test ***=** []  
Rsqu_train ***=** []  
dummy1 ***=** []  
Alpha ***=** 10 ***** np.array(range(0,1000))  
pbar ***=** tqdm(Alpha)  
  
**for** alpha ***in** pbar:  
    RidgeModel ***=** Ridge(alpha***=**alpha)  
    RidgeModel.fit(x_train_pr, y_train)  
    test_score, train_score ***=** RidgeModel.score(x_test_pr, y_test), RidgeModel.score(x_train_pr, y_train)  
  
    pbar.set_postfix({"Test Score": test_score, "Train Score": train_score})  
  
    Rsqu_test.append(test_score)  
    Rsqu_train.append(train_score)
```

```
100%|██████████| 1000/1000 [02:10<00:00,  7.65it/s, Test Score=0.564, Train Score=0.859]
```

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

```

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

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

<matplotlib.legend.Legend at 0x7f38a8e104e0>

```

**Figure 4:** The blue line represents the R<sup>2</sup> of the validation data, and the red line represents the R<sup>2</sup> of the training data. The x-axis represents the different values of Alpha.

Here the model is built and tested on the same data, so the training and test data are the same.

The red line in Figure 4 represents the R<sup>2</sup> of the training data. As alpha increases the R<sup>2</sup> decreases. Therefore, as alpha increases, the model performs worse on the training data

The blue line represents the R<sup>2</sup> on the validation data. As the value for alpha increases, the R<sup>2</sup> increases and converges at a point.

## Question #5):

Perform Ridge regression. Calculate the R<sup>2</sup> using the polynomial features, use the training data to train the model and use the test data to test the model. The parameter alpha should be set to 10.

```

RigeModel ***=** Ridge(alpha***=**10)

RigeModel.fit(x_train_pr, y_train)

RigeModel.score(x_test_pr, y_test)

```

```
0.5418576440207993
```

## Part 4: Grid Search

The term **alpha** is a **hyperparameter**. Sklearn has the class **GridSearchCV** to make the process of finding the best hyperparameter simpler.

Let's import **GridSearchCV** from the module **model\_selection**.

```
**from** sklearn.model_selection **import** GridSearchCV
```

We create a dictionary of parameter values:

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

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

Create a Ridge regression object:

```
RR**=**Ridge()  
RR  
  
Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,  
      normalize=False, random_state=None, solver='auto', tol=0.001)
```

Create a ridge grid search object:

```
Grid1 **=** GridSearchCV(RR, parameters1, cv**=**4, iid**=None**)
```

In order to avoid a deprecation warning due to the iid parameter, we set the value of iid to "None".

Fit the model:

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

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

```
a=1.0, copy_X=True, fit_intercept=True, max_iter=None,
normalize=False, random_state=None, solver='auto', tol=0.001)

0, 1000, 10000, 100000, 1000000}]], train_score='warn',
```

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

```
BestRR**=**Grid1.best_estimator_
BestRR
```

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

We now test our model on the test data:

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

```
0.8411649831036152
```

**Question #6:** Perform a grid search for the alpha parameter and the normalization parameter, then find the best values of the parameters:

```
parameters2 **=** [ {'alpha': [0.001,0.1,1, 10, 100, 1000,10000,100000,1000000], 'normalize':[**True**, **False**]}

Grid2 **=** GridSearchCV(Ridge(), parameters2, cv**=**4)

Grid2.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],y_data)

Grid2.best_estimator_

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

# Model Evaluation and Refinement (MMSTinao Feb. 2022)

Estimated time needed: **30** minutes

## Objectives

After completing this lab you will be able to:

- Evaluate and refine prediction models

## Table of Contents

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

This dataset was hosted on IBM Cloud object. Click [HERE](#) for free storage.

```
[1]: #install specific version of libraries used in lab
#!mamba install pandas==1.3.3 -y
#!mamba install numpy=1.21.2 -y
#!mamba install sklearn=0.20.1 -y
#!mamba install ipywidgets=7.4.2 -y
```

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

# Import clean data
path = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/module_5_auto.csv'
df = pd.read_csv(path)
```

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

First, let's only use numeric data:

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

[4]:	Unnamed: 0	Unnamed: 0.1	symboling	normalized-losses	wheel-base	length	width	height	curb-weight
0	0	0	3	122	88.6	0.811148	0.890278	48.8	2548
1	1	1	3	122	88.6	0.811148	0.890278	48.8	2548
2	2	2	1	122	94.5	0.822681	0.909722	52.4	2823
3	3	3	2	164	99.8	0.848630	0.919444	54.3	2337
4	4	4	2	164	99.4	0.848630	0.922222	54.3	2824

engine-size	...	stroke	compression-ratio	horsepower	peak-rpm	city-mpg	highway-mpg	price	city-L/100km	diesel	gas
130	...	2.68	9.0	111.0	5000.0	21	27	13495.0	11.190476	0	1
130	...	2.68	9.0	111.0	5000.0	21	27	16500.0	11.190476	0	1
152	...	3.47	9.0	154.0	5000.0	19	26	16500.0	12.368421	0	1
109	...	3.40	10.0	102.0	5500.0	24	30	13950.0	9.791667	0	1
136	...	3.40	8.0	115.0	5500.0	18	22	17450.0	13.055556	0	1

5 rows × 21 columns

Libraries for plotting:

```
[6]: from ipywidgets import interact, interactive, fixed, interact_manual
```

## Functions for Plotting

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

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

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

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

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

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

    xmax=max([xtrain.values.max(), xtest.values.max()])
    xmin=min([xtrain.values.min(), xtest.values.min()])
    x=np.arange(xmin, xmax, 0.1)

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

## Part 1: Training and Testing

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

```
[9]: y_data = df['price']
```

Drop price data in dataframe **x\_data**:

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

Now, we randomly split our data into training and testing data using the function **train\_test\_split**.

```
[11]: 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=0.10, random_state=1)
```

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

```
number of test samples : 21
number of training samples: 180
```

The **test\_size** parameter sets the proportion of data that is split into the testing set. In the above, the testing set is 10% of the total dataset.

## Question #1):

Use the function "train\_test\_split" to split up the dataset such that 40% of the data samples will be utilized for testing. Set the parameter "random\_state" equal to zero. The output of the function should be the following: "x\_train1" , "x\_test1" , "y\_train1" and "y\_test1".

```
[12]: # Write your code below and press Shift+Enter to execute
x_train1, x_test1, y_train1, y_test1 = train_test_split(x_data, y_data, test_size=0.4, random_state=0)
print("number of test samples :", x_test1.shape[0])
print("number of training samples:", x_train1.shape[0])
```

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

► Click here for the solution

Let's import LinearRegression from the module linear\_model.

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

```
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/linear_model/least_angle.py:35: DeprecationWarning: 'np.float' is a deprecated alias for the builtin `float`. To silence this warning, use `float` by itself. Doing this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.float64` here.
Deprecation in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
    eps=np.finfo(np.float).eps,
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/linear_model/least_angle.py:597: DeprecationWarning: 'np.float' is a deprecated alias for the builtin `float`. To silence this warning, use `float` by itself. Doing this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.float64` here.
Deprecation in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
    eps=np.finfo(np.float).eps, copy_X=True, fit_path=True,
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/linear_model/least_angle.py:836: DeprecationWarning: 'np.float' is a deprecated alias for the builtin `float`. To silence this warning, use `float` by itself. Doing this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.float64` here.
Deprecation in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
    eps=np.finfo(np.float).eps, copy_X=True, fit_path=True,
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/linear_model/least_angle.py:862: DeprecationWarning: 'np.float' is a deprecated alias for the builtin `float`. To silence this warning, use `float` by itself. Doing this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.float64` here.
Deprecation in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
    eps=np.finfo(np.float).eps, positive=False):
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/linear_model/least_angle.py:1097: DeprecationWarning: 'np.float' is a deprecated alias for the builtin `float`. To silence this warning, use `float` by itself. Doing this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.float64` here.
Deprecation in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
    max_n_alphas=1000, n_jobs=None, eps=np.finfo(np.float).eps,
```

We create a Linear Regression object:

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

We fit the model using the feature "horsepower":

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

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

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

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

```
[16]: 0.3635875575078824
```

We can see the R<sup>2</sup> is much smaller using the test data compared to the training data.

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

```
[17]: 0.6619724197515103
```

## Question #2):

Find the R<sup>2</sup> on the test data using 40% of the dataset for testing.

```
[18]: # Write your code below and press Shift+Enter to execute..  
x_train1, x_test1, y_train1, y_test1 = train_test_split(x_data, y_data, test_size=0.4, random_state=0)  
lre.fit(x_train1[['horsepower']],y_train1)  
lre.score(x_test1[['horsepower']],y_test1)
```

```
[18]: 0.7139364665406973
```

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

## Cross-Validation Score

Let's import `model_selection` from the module `cross_val_score`.

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

We input the object, the feature ("horsepower"), and the target data (y\_data). The parameter 'cv' determines the number of folds. In this case, it is 4.

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

```
/home/jupyterlab/conda/envs/python3.7/site-packages/sklearn/model_selection/_split.py:437: DeprecationWarning: `np.int` is a deprecated alias for the built-in type `int` by itself. Doing this will not modify any behavior and is safe. When replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. Current use, check the release note link for additional information.  
Deprecation in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations  
fold_sizes = np.full(n_splits, n_samples // n_splits, dtype=np.int)  
/home/jupyterlab/conda/envs/python3.7/site-packages/sklearn/model_selection/_split.py:113: DeprecationWarning: `np.bool` is a deprecated alias for the built-in type `bool` by itself. Doing this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.bool_` here.  
Deprecation in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
```

The default scoring is R<sup>2</sup>. Each element in the array has the average R<sup>2</sup> value for the fold:

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

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

```
[23]: print("The mean of the folds are", Rcross.mean(), "and the standard deviation is", Rcross.std())  
The mean of the folds are 0.522009915042119 and the standard deviation is 0.2911839444756029
```

We can use negative squared error as a score by setting the parameter 'scoring' metric to 'neg\_mean\_squared\_error'.

```
[24]: -1 * cross_val_score(lre,x_data[['horsepower']], y_data, cv=4, scoring='neg_mean_squared_error')
```

```
'home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/model_selection/_split.py:437: DeprecationWarning:  
use `int` by itself. Doing this will not modify any behavior and is safe. When replacing `np.int`, you may wish to use e.  
- current use, check the release note link for additional information.  
Deprecation in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations  
    fold_sizes = np.full(n_splits, n_samples // n_splits, dtype=np.int)  
'home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/model_selection/_split.py:113: DeprecationWarning:  
..., use `bool` by itself. Doing this will not modify any behavior and is safe. If you specifically wanted the numpy scalar  
Deprecation in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations  
    test_mask = np.zeros(_num_samples(X), dtype=np.bool)  
'home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/model_selection/_split.py:113: DeprecationWarning:  
[24]: array([20254142.84026704, 43745493.2650517 , 12539630.34014931,  
           17561927.72247591])
```

## Question #3):

Calculate the average  $R^2$  using two folds, then find the average  $R^2$  for the second fold utilizing the "horsepower" feature:

```
[25]: # Write your code below and press Shift+Enter to execute..  
Rc=cross_val_score(lre,x_data[['horsepower']], y_data, cv=2)  
Rc.mean()
```

```
'home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/model  
use `int` by itself. Doing this will not modify any behavior and is safe. Wh  
r current use, check the release note link for additional information.  
Deprecation in NumPy 1.20; for more details and guidance: 

```
\[25\]: 0.5166761697127429
```


```

You can also use the function 'cross\_val\_predict' to predict the output. The function splits up the data into the specified number of folds, with one fold for testing and the other folds are used for training. First, import the function:

```
[26]: from sklearn.model_selection import cross_val_predict
```

We input the object, the feature "**horsepower**", and the target data **y\_data**. The parameter 'cv' determines the number of folds. In this case, it is 4. We can produce an output:

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

/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/model_selection/_split.py:437:
use `int` by itself. Doing this will not modify any behavior and is safe. When replacing `np.int`, your current use, check the release note link for additional information.
Deprecation in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html
    fold_sizes = np.full(n_splits, n_samples // n_splits, dtype=np.int)
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/model_selection/_split.py:113:

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

## Part 2: Overfitting, Underfitting and Model Selection

It turns out that the test data, sometimes referred to as the "out of sample data", is a much better measure of how well your model performs in the real world. One reason for this is **overfitting**.

Let's go over some examples. It turns out these differences are more apparent in Multiple Linear Regression and Polynomial Regression so we will explore overfitting in that context.

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

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

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

Prediction using training data:

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

[29]: array([ 7426.6731551 , 28323.75090803, 14213.38819709,  4052.34146983,
       34500.19124244])
```

Prediction using test data:

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

[30]: array([11349.35089149,  5884.11059106, 11208.6928275 ,  6641.07786278,
       15565.79920282])
```

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

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

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

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

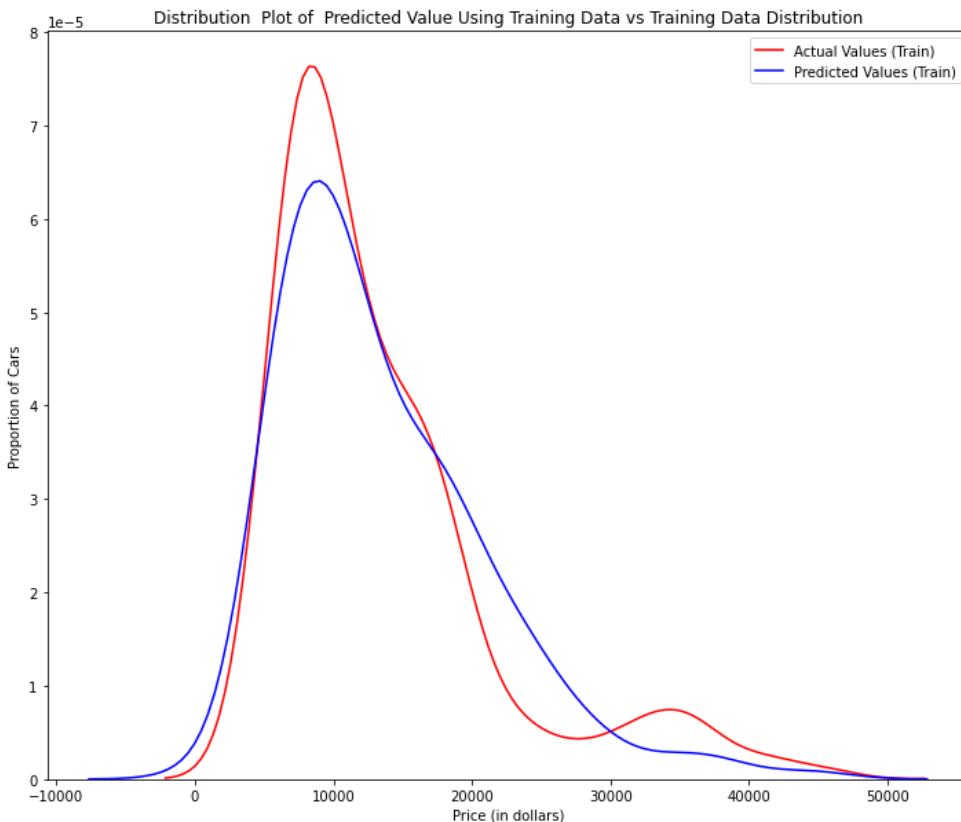


Figure 1: Plot of predicted values using the training data compared to the actual values of the training data.

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

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

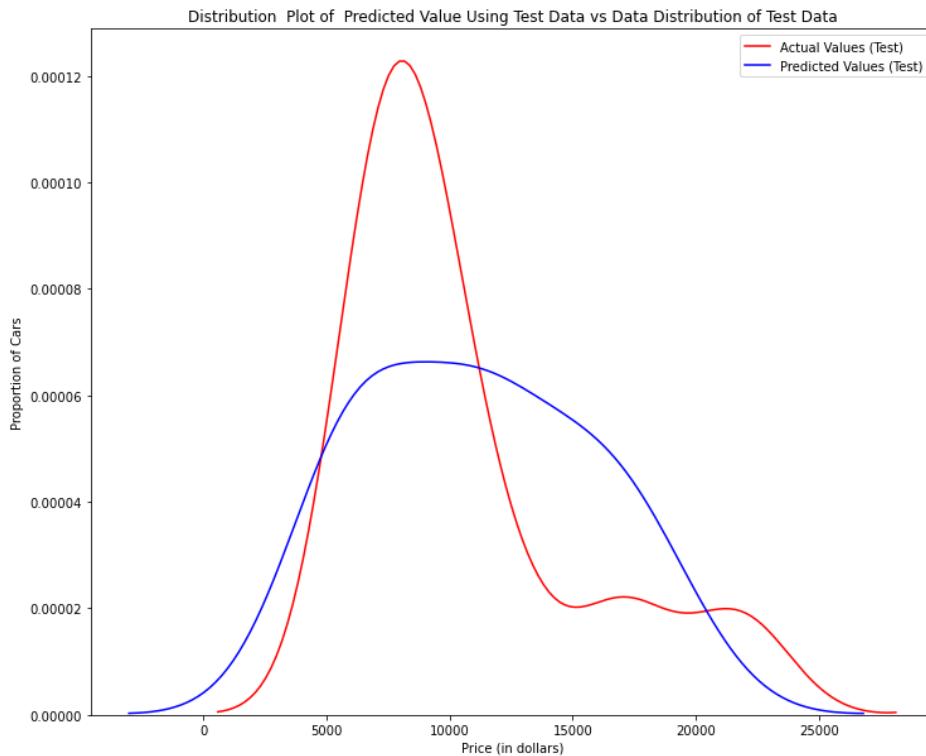


Figure 2: Plot of predicted value using the test data compared to the actual values of the test data.

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

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

## Overfitting

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

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

```
[35]: x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.45, random_state=0)
```

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

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

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

Predicted values: [ 6728.73285076  7308.05589332 12213.80614303 18893.13997531]
True values: [ 6295. 10698. 13860. 13499.]
```

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

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

Predicted values: [ 6728.73285076  7308.05589332 12213.80614303 18893.13997531]
True values: [ 6295. 10698. 13860. 13499.]
```

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

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

Now, let's create a Linear Regression model "poly" and train it.

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

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

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

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

```
[38]: array([ 6728.73285076,  7308.05589332, 12213.80614303, 18893.13997531,
19995.82734265])
```

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

Predicted values: [ 6728.73285076  7308.05589332 12213.80614303 18893.13997531]
True values: [ 6295. 10698. 13860. 13499.]
```

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

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

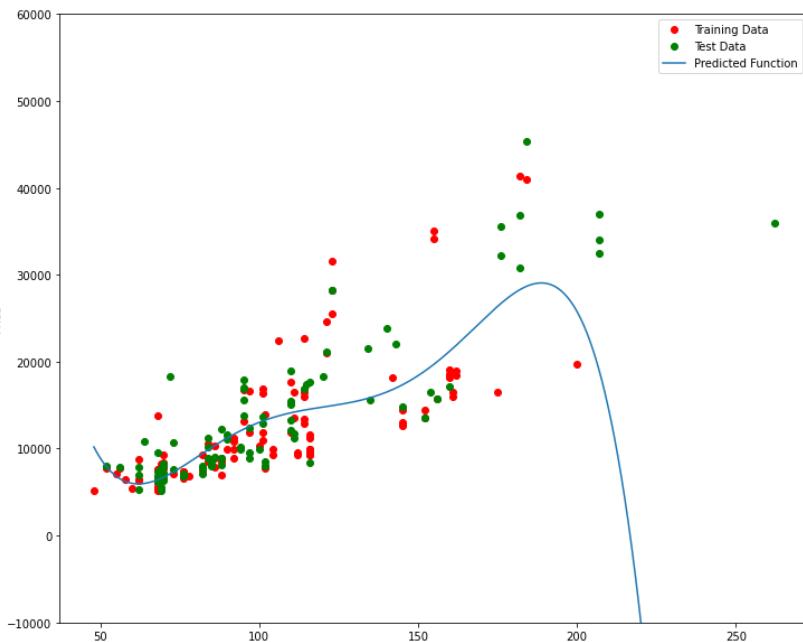


Figure 3: A polynomial regression model where red dots represent training data, green dots represent test data, and the blue line represents the model prediction.

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

R<sup>2</sup> of the training data:

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

```
[41]: 0.5567716902237296
```

R<sup>2</sup> of the test data:

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

```
[42]: -29.87158580724305
```

We see the R<sup>2</sup> for the training data is 0.5567 while the R<sup>2</sup> on the test data was -29.87. The lower the R<sup>2</sup>, the worse the model. A negative R<sup>2</sup> is a sign of overfitting.

Let's see how the R<sup>2</sup> changes on the test data for different order polynomials and then plot the results:

```
[43]: Rsqu_test = []

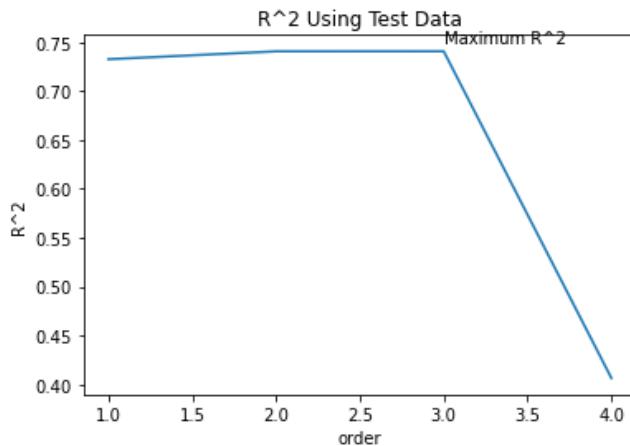
order = [1, 2, 3, 4]
for n in order:
    pr = PolynomialFeatures(degree=n)

    x_train_pr = pr.fit_transform(x_train[['horsepower']])
    x_test_pr = pr.fit_transform(x_test[['horsepower']])...
    lr.fit(x_train_pr, y_train)

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

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

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



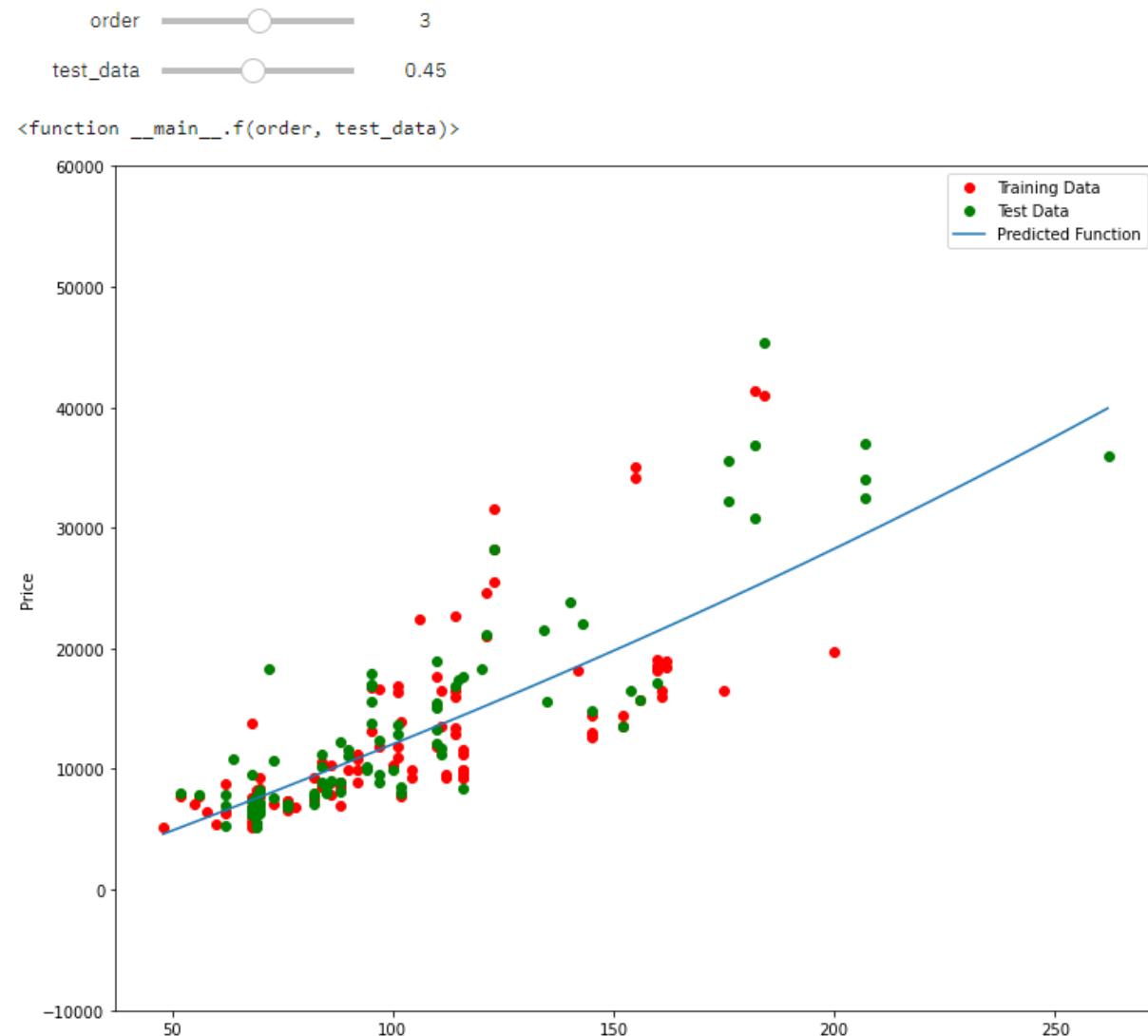
We see the  $R^2$  gradually increases until an order three polynomial is used. Then, the  $R^2$  dramatically decreases at an order four polynomial.

The following function will be used in the next section. Please run the cell below.

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

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

```
[45]: interact(f, order=(0, 6, 1), test_data=(0.05, 0.95, 0.05))
```



## Question #4a):

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

```
[46]: # Write your code below and press Shift+Enter to execute...
pr1=PolynomialFeatures(degree=2)
```

## Question #4b):

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

```
[47]: # Write your code below and press Shift+Enter to execute...
x_train_pr1=pr1.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
x_test_pr1=pr1.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
```

► Click here for the solution

## Question #4c):

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

```
[48]: # Write your code below and press Shift+Enter to execute...
x_train_pr1.shape #there are now 15 features
```

[48]: (110, 15)

## Question #4d):

Create a linear regression model "poly1". Train the object using the method "fit" using the polynomial features.

```
[49]: # Write your code below and press Shift+Enter to execute...
poly1=LinearRegression().fit(x_train_pr1,y_train)
```

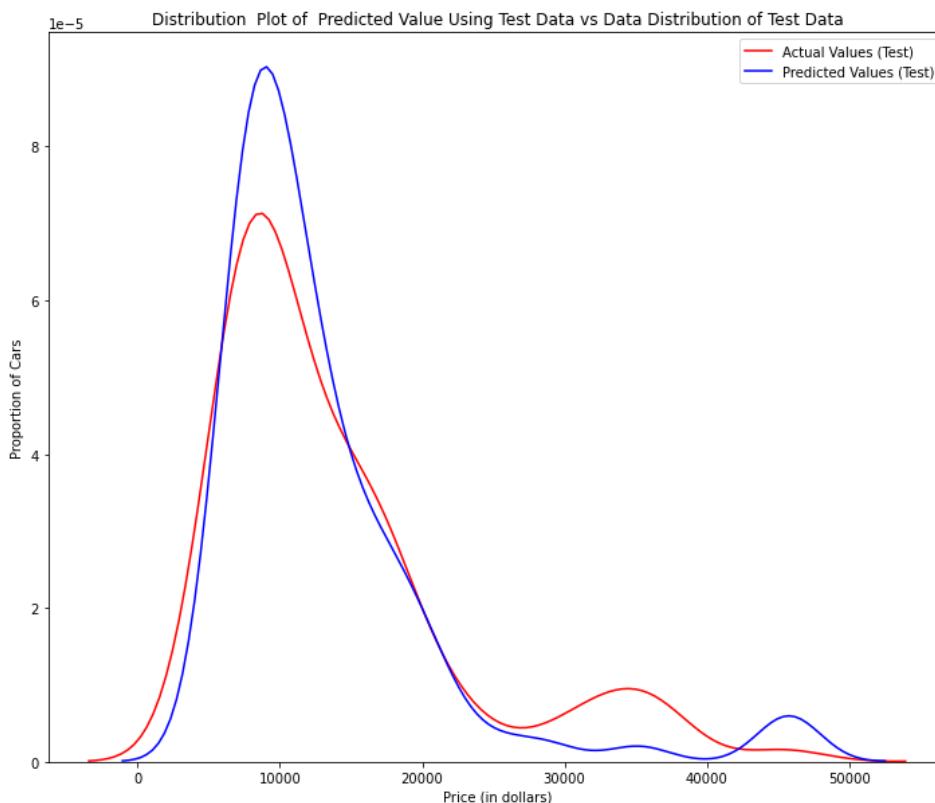
▼ Click here for the solution

```
poly1=LinearRegression().fit(x_train_pr1,y_train)
```

## Question #4e):

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

```
[53]: # Write your code below and press Shift+Enter to execute.  
yhat_test1=poly1.predict(x_test_pr1)  
  
Title='Distribution Plot of Predicted Value Using Test Data vs Data Distribution of Test Data'  
  
DistributionPlot(y_test, yhat_test1, "Actual Values (Test)", "Predicted Values (Test)", Title)
```



[Click here for the solution](#)

## Question #4f):

Using the distribution plot above, describe (in words) the two regions where the predicted prices are less accurate than the actual prices.

---

```
[54]: # Write your code below and press Shift+Enter to execute.  
#The predicted value is higher than actual value for cars where the price $10,000 range,  
conversely the predicted price is lower than the price cost in the $30,000 to $40,000 range. As such the model is not as accurate in these ranges.
```

## Part 3: Ridge Regression

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

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

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

Let's import Ridge from the module linear models.

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

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

```
[57]: RidgeModel=Ridge(alpha=1)
```

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

```
[58]: RidgeModel.fit(x_train_pr, y_train)
```

```
[58]: Ridge(alpha=1, copy_X=True, fit_intercept=True, max_iter=None,
normalize=False, random_state=None, solver='auto', tol=0.001)
```

Similarly, you can obtain a prediction:

```
[59]: yhat = RidgeModel.predict(x_test_pr)
```

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

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

predicted: [ 6570.82441941  9636.24891471 20949.92322737 19403.60313255]
test set : [ 6295. 10698. 13860. 13499.]
```

We select the value of alpha that minimizes the test error. To do so, we can use a for loop. We have also created a progress bar to see how many iterations we have completed so far.

```
[61]: from tqdm import tqdm

Rsqu_test = []
Rsqu_train = []
dummy1 = []
Alpha = 10 * np.array(range(0,1000))
pbar = tqdm(Alpha)

for alpha in pbar:
    RidgeModel = Ridge(alpha=alpha)
    RidgeModel.fit(x_train_pr, y_train)
    test_score, train_score = RidgeModel.score(x_test_pr, y_test), RidgeModel.score(x_train_pr, y_train)

    pbar.set_postfix({"Test Score": test_score, "Train Score": train_score})

    Rsqu_test.append(test_score)
    Rsqu_train.append(train_score)

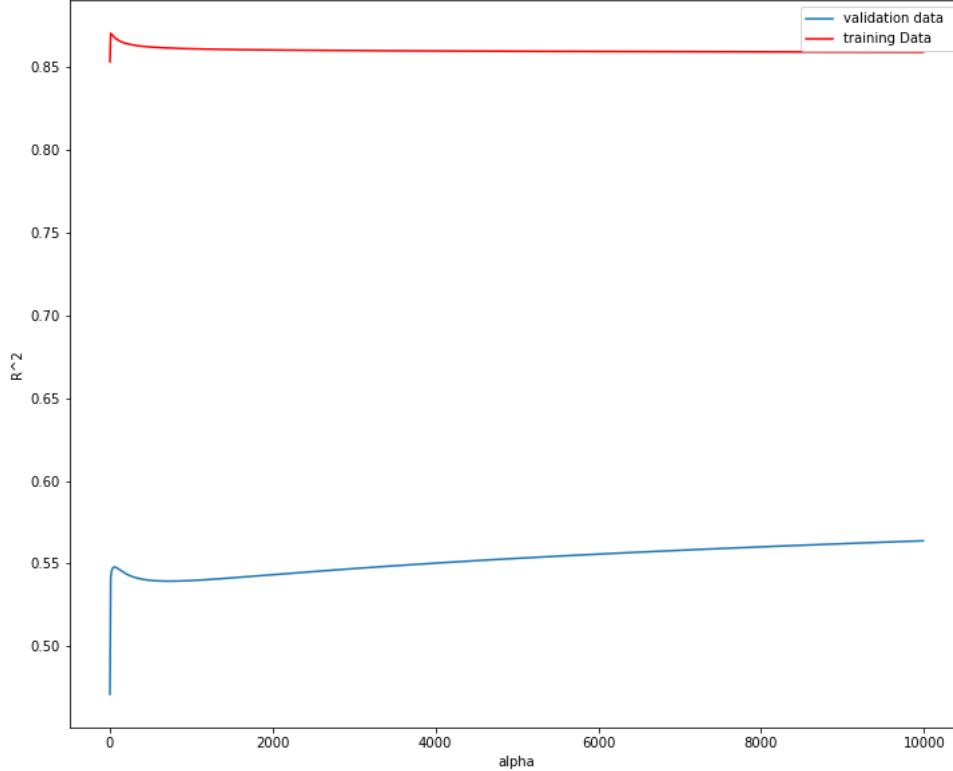
100%|██████████| 1000/1000 [00:03<00:00, 251.12it/s, Test Score=0.564, Train Score=0.859]
```

We can plot out the value of  $R^2$  for different alphas:

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

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

```
[62]: <matplotlib.legend.Legend at 0x7f158585ed50>
```



**Figure 4:** The blue line represents the R<sup>2</sup> of the validation data, and the red line represents the R<sup>2</sup> of the training data. The x-axis represents the different values of Alpha. Here the model is built and tested on the same data, so the training and test data are the same.

The red line in Figure 4 represents the R<sup>2</sup> of the training data. As alpha increases the R<sup>2</sup> decreases. Therefore, as alpha increases, the model performs worse on the training data

The blue line represents the R<sup>2</sup> on the validation data. As the value for alpha increases, the R<sup>2</sup> increases and converges at a point.

## Question #5):

Perform Ridge regression. Calculate the R<sup>2</sup> using the polynomial features, use the training data to train the model and use the test data to test the model. The parameter alpha should be set to 10.

```
[63]: # Write your code below and press Shift+Enter to execute..  
RidgeModel = Ridge(alpha=10)..  
RidgeModel.fit(x_train_pr, y_train)  
RidgeModel.score(x_test_pr, y_test)  
  
[63]: 0.5418576440207269
```

## Part 4: Grid Search

The term `alpha` is a **hyperparameter**. Sklearn has the class **GridSearchCV** to make the process of finding the best hyperparameter simpler.

Let's import **GridSearchCV** from the module **model\_selection**.

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

We create a dictionary of parameter values:

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

Create a Ridge regression object:

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

Create a ridge grid search object:

```
[67]: Grid1 = GridSearchCV(RR, parameters1, cv=4, iid=None)
```

In order to avoid a deprecation warning due to the iid parameter, we set the value of iid to "None".

Fit the model:

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

/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/model_selection/_split.py:100: DeprecationWarning: `use_int` is deprecated. It will be removed in a future version. Please use `int` by itself. Doing this will not modify any behavior and is safe. When replacing `np.int` in your current use, check the release note link for additional information.
  DeprecationWarning)
  fold_sizes = np.full(n_splits, n_samples // n_splits, dtype=np.int)
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/model_selection/_split.py:100: DeprecationWarning: `use_int` is deprecated. Doing this will not modify any behavior and is safe. If you specified
[68]: GridSearchCV(cv=4, error_score='raise-deprecating',
                  estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
                  normalize=False, random_state=None, solver='auto', tol=0.001),
                  fit_params=None, iid=None, n_jobs=None,
                  param_grid=[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 1000000]}],
                  pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
                  scoring=None, verbose=0)
```

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

```
[69]: BestRR=Grid1.best_estimator_
BestRR

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

We now test our model on the test data:

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

[70]: 0.8411649831036152
```

## Question #6):

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

```
[71]: # Write your code below and press Shift+Enter to execute.
parameters2=[{'alpha': [0.001,0.1,1, 10, 100, 1000,10000,100000], 'normalize':[True,False]}]
Grid2 = GridSearchCV(Ridge(), parameters2, cv=4)
Grid2.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],y_data)
Grid2.best_estimator_

/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/model_selection/_split.py:437:
use `int` by itself. Doing this will not modify any behavior and is safe. When replacing `np.int`, your current use, check the release note link for additional information.
DeprecationWarning: NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-not
-fold\_sizes = np.full\(n\_splits, n\_samples // n\_splits, dtype=np.int\)
/home/jupyterlab/conda/envs/python/lib/python3.7/site-packages/sklearn/model\_selection/\_split.py:113:
```

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

► Click here for the solution

Thank you for completing this lab!

## Author

[Joseph Santarcangelo](#)

## **Module Introduction & Learning Objectives**

### **Module Introduction**

Congratulations! You have now completed all the weeks' modules for this course. In this last week, you will complete the final assignment that will be graded by your peers. In this final assignment, you will assume the role of a Data Analyst working at a real estate investment trust organization who wants to start investing in residential real estate. You will be given a dataset containing detailed information about house prices in the region based on a number of property features, and it will be your job to analyze and predict the market price of houses given that information.

### **Learning Objectives**

- Create a Jupyter notebook.
- Apply data analysis and modeling techniques to housing price data

# Peer-Graded Assignment: Final Assignment

**Estimated time needed:** 45 minutes

Congratulations! You have now completed all the modules of this course. This week, you will complete the final assignment that will be graded by your peers. You will be provided a notebook with instructions and questions.

## Software Used in this Assignment

You will be using Jupyter Notebook through IBM Watson studio for the final project and will be required to share the link to your notebook. If you are not familiar with IBM Watson studio, instructions on how to get started has been provided for you.

## Dataset Used in this Assignment

This dataset contains house sale prices for King County, which includes Seattle. It includes homes sold between May 2014 and May 2015. It was taken from [here](#). It was also slightly modified for the purposes of this course. Here is the description of the data:

Variable	Description
id	A notation for a house
date	Date house was sold
price	Price is prediction target
bedrooms	Number of bedrooms
bathrooms	Number of bathrooms
sqft_living	Square footage of the home
sqft_lot	Square footage of the lot
floors	Total floors (levels) in house
waterfront	House which has a view to a waterfront
view	Has been viewed
condition	How good the condition is overall
grade	overall grade given to the housing unit, based on King County grading system
sqft_above	Square footage of house apart from basement
sqft_basement	Square footage of the basement

<b>Variable</b>	<b>Description</b>
yr_built	Built Year
yr_renovated	Year when house was renovated
zipcode	Zip code
lat	Latitude coordinate
long	Longitude coordinate
sqft_living15	Living room area in 2015(implies-- some renovations) This might or might not have affected the lotsize area
sqft_lot15	LotSize area in 2015(implies-- some renovations)

## Assignment Scenario

You are a Data Analyst working at a Real Estate Investment Trust. The Trust will like to start investing in Residential real estate. You are tasked with determining the market price of a house given a set of features. You will analyze and predict housing prices using attributes or features such as square footage, number of bedrooms, number of floors, and so on.

## Guidelines for the Submission

Copy the link to the notebook and paste it in IBM Watson Studio: [https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DA0101EN-SkillsNetwork/labs/FinalModule\\_Coursera/House\\_Sales\\_in\\_King\\_Count\\_USA.ipynb](https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DA0101EN-SkillsNetwork/labs/FinalModule_Coursera/House_Sales_in_King_Count_USA.ipynb)

## Grading Information

You will be required to submit a link to your notebook for peer grading.

**The main grading criteria will be:**

- Have you reproduced the correct information using the functions?
- Have you created the appropriate graphs?
- Did you properly fit a regression model?
- Have you shared the link to your Notebook?

**You will not be judged on:**

- Your English language, including spelling or grammatical mistakes.
- The content of any text or image(s) or where a link is hyperlinked to.

## Author(s)

- Aije Egwaikhide

[https://cocl.us/da0101en\\_coursera\\_labb](https://cocl.us/da0101en_coursera_labb)

[https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DA0101EN-SkillsNetwork/labs/FinalModule\\_Coursera/House\\_Sales\\_in\\_King\\_Count\\_USA.ipynb](https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DA0101EN-SkillsNetwork/labs/FinalModule_Coursera/House_Sales_in_King_Count_USA.ipynb)

Copy the link to the notebook and paste it in IBM Watson Studio:

[https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DA0101EN-SkillsNetwork/labs/FinalModule\\_Coursera/House\\_Sales\\_in\\_King\\_Count\\_USA.ipynb](https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DA0101EN-SkillsNetwork/labs/FinalModule_Coursera/House_Sales_in_King_Count_USA.ipynb)

## 7.6.Final Assignment

Seongjoo Brenden Song edited this page on Nov 7, 2021 · 4 revisions

---

### Learning Objectives

- Create a Jupyter notebook
  - Apply data analysis and modeling techniques to housing price data
- 

- [Final Assignment](#)
- [Final Exam](#)

# Final Assignment



## Data Analysis with Python

### House Sales in King County, USA

This dataset contains house sale prices for King County, which includes Seattle. It includes homes sold between May 2014 and May 2015.

Variable	Description
id	A notation for a house
date	Date house was sold
price	Price is prediction target
bedrooms	Number of bedrooms
bathrooms	Number of bathrooms
sqft_living	Square footage of the home
sqft_lot	Square footage of the lot
floors	Total floors (levels) in house
waterfront	House which has a view to a waterfront
view	Has been viewed
condition	How good the condition is overall
grade	overall grade given to the housing unit, based on King County grading system
sqft_above	Square footage of house apart from basement
sqft_basement	Square footage of the basement
yr_built	Built Year
yr_renovated	Year when house was renovated
zipcode	Zip code
lat	Latitude coordinate
long	Longitude coordinate
sqft_living15	Living room area in 2015(implies-- some renovations) This might or might not have affected the lotsize area
sqft_lot15	LotSize area in 2015(implies-- some renovations)

You will require the following libraries:

```
In [2]: import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler,PolynomialFeatures
from sklearn.linear_model import LinearRegression
%matplotlib inline
```

## Module 1: Importing Data Sets

Load the csv:

```
In [3]: file_name='https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/FinalModule_CourseraData.csv'
df=pd.read_csv(file_name)
```

We use the method `head` to display the first 5 columns of the dataframe.

We use the method `head` to display the first 5 columns of the dataframe.

```
In [4]: df.head()
```

```
Out[4]: Unnamed: 0 id date price bedrooms bathrooms sqft_living sqft_lot floors waterfront ... grade sqft_above sqft_basement yr_built
0 0 7129300520 20141013T000000 221900.0 3.0 1.00 1180 5650 1.0 0 ... 7 1180 0 1955
1 1 6414100192 20141209T000000 538000.0 3.0 2.25 2570 7242 2.0 0 ... 7 2170 400 1951
2 2 5631500400 20150225T000000 180000.0 2.0 1.00 770 10000 1.0 0 ... 6 770 0 1933
3 3 2487200875 20141209T000000 604000.0 4.0 3.00 1960 5000 1.0 0 ... 7 1050 910 1965
4 4 1954400510 20150218T000000 510000.0 3.0 2.00 1680 8080 1.0 0 ... 8 1680 0 1987
```

5 rows × 22 columns

yr_renovated	zipcode	lat	long	sqft_living15	sqft_lot15
0	98178	47.5112	-122.257	1340	5650
1991	98125	47.7210	-122.319	1690	7639
0	98028	47.7379	-122.233	2720	8062
0	98136	47.5208	-122.393	1360	5000
0	98074	47.6168	-122.045	1800	7503

## Question 1

Display the data types of each column using the function `dtypes`, then take a screenshot and submit it, include your code in the image.

In [5]: `df.dtypes`

```
Out[5]: Unnamed: 0      int64
id          int64
date        object
price       float64
bedrooms    float64
bathrooms   float64
sqft_living int64
sqft_lot    int64
floors      float64
waterfront  int64
view        int64
condition   int64
grade        int64
sqft_above  int64
sqft_basement int64
yr_built    int64
yr_renovated int64
zipcode     int64
lat         float64
long        float64
sqft_living15 int64
sqft_lot15   int64
dtype: object
```

We use the method `describe` to obtain a statistical summary of the dataframe.

In [6]: `df.describe()`

	Unnamed: 0	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade
count	21613.00000	2.161300e+04	2.161300e+04	21600.000000	21603.000000	21613.000000	2.161300e+04	21613.000000	21613.000000	21613.000000	...	21613.000000
mean	10806.00000	4.580302e+09	5.400881e+05	3.372870	2.115736	2079.899736	1.510697e+04	1.494309	0.007542	0.234303	...	7.656873
std	6239.28002	2.876566e+09	3.671272e+05	0.926657	0.768996	918.440897	4.142051e+04	0.539989	0.086517	0.766318	...	1.175459
min	0.00000	1.000102e+06	7.500000e+04	1.000000	0.500000	290.000000	5.200000e+02	1.000000	0.000000	0.000000	...	1.000000
25%	5403.00000	2.123049e+09	3.219500e+05	3.000000	1.750000	1427.000000	5.040000e+03	1.000000	0.000000	0.000000	...	7.000000
50%	10806.00000	3.904930e+09	4.500000e+05	3.000000	2.250000	1910.000000	7.618000e+03	1.500000	0.000000	0.000000	...	7.000000
75%	16209.00000	7.308900e+09	6.450000e+05	4.000000	2.500000	2550.000000	1.068800e+04	2.000000	0.000000	0.000000	...	8.000000
max	21612.00000	9.900000e+09	7.700000e+06	33.000000	8.000000	13540.000000	1.651359e+06	3.500000	1.000000	4.000000	...	13.000000

8 rows × 21 columns

sqft_above	sqft_basement	yr_built	yr_renovated	zipcode	lat	long	sqft_living15	sqft_lot15
21613.000000	21613.000000	21613.000000	21613.000000	21613.000000	21613.000000	21613.000000	21613.000000	21613.000000
1788.390691	291.509045	1971.005136	84.402258	98077.939805	47.560053	-122.213896	1986.552492	12768.455652
828.090978	442.575043	29.373411	401.679240	53.505026	0.138564	0.140828	685.391304	27304.179631
290.000000	0.000000	1900.000000	0.000000	98001.000000	47.155900	-122.519000	399.000000	651.000000
1190.000000	0.000000	1951.000000	0.000000	98033.000000	47.471000	-122.328000	1490.000000	5100.000000
1560.000000	0.000000	1975.000000	0.000000	98065.000000	47.571800	-122.230000	1840.000000	7620.000000
2210.000000	560.000000	1997.000000	0.000000	98118.000000	47.678000	-122.125000	2360.000000	10083.000000
9410.000000	4820.000000	2015.000000	2015.000000	98199.000000	47.777600	-121.315000	6210.000000	871200.000000

## Module 2: Data Wrangling

### Question 2

Drop the columns "id" and "Unnamed: 0" from axis 1 using the method drop(), then use the method describe() to obtain a statistical summary of the data. Take a screenshot and submit it, make sure the inplace parameter is set to True

```
In [7]: df.drop(['id', 'Unnamed: 0'], axis = 1, inplace=True)
df.describe()
```

Out[7]:	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade
count	2.161300e+04	21600.000000	21603.000000	21613.000000	2.161300e+04	21613.000000	21613.000000	21613.000000	21613.000000	21613.000000
mean	5.400881e+05	3.372870	2.115736	2079.899736	1.510697e+04	1.494309	0.007542	0.234303	3.409430	7.656873
std	3.671272e+05	0.926657	0.768996	918.440897	4.142051e+04	0.539989	0.086517	0.766318	0.650743	1.175459
min	7.500000e+04	1.000000	0.500000	290.000000	5.200000e+02	1.000000	0.000000	0.000000	1.000000	1.000000
25%	3.219500e+05	3.000000	1.750000	1427.000000	5.040000e+03	1.000000	0.000000	0.000000	3.000000	7.000000
50%	4.500000e+05	3.000000	2.250000	1910.000000	7.618000e+03	1.500000	0.000000	0.000000	3.000000	7.000000
75%	6.450000e+05	4.000000	2.500000	2550.000000	1.068800e+04	2.000000	0.000000	0.000000	4.000000	8.000000
max	7.700000e+06	33.000000	8.000000	13540.000000	1.651359e+06	3.500000	1.000000	4.000000	5.000000	13.000000

sqft_above	sqft_basement	yr_built	yr_renovated	zipcode	lat	long	sqft_living15	sqft_lot15
21613.000000	21613.000000	21613.000000	21613.000000	21613.000000	21613.000000	21613.000000	21613.000000	21613.000000
1788.390691	291.509045	1971.005136	84.402258	98077.939805	47.560053	-122.213896	1986.552492	12768.455652
828.090978	442.575043	29.373411	401.679240	53.505026	0.138564	0.140828	685.391304	27304.179631
290.000000	0.000000	1900.000000	0.000000	98001.000000	47.155900	-122.519000	399.000000	651.000000
1190.000000	0.000000	1951.000000	0.000000	98033.000000	47.471000	-122.328000	1490.000000	5100.000000
1560.000000	0.000000	1975.000000	0.000000	98065.000000	47.571800	-122.230000	1840.000000	7620.000000
2210.000000	560.000000	1997.000000	0.000000	98118.000000	47.678000	-122.125000	2360.000000	10083.000000
9410.000000	4820.000000	2015.000000	2015.000000	98199.000000	47.777600	-121.315000	6210.000000	871200.000000

We can see we have missing values for the columns `bedrooms` and `bathrooms`

```
In [8]: print("number of NaN values for the column bedrooms :", df['bedrooms'].isnull().sum())
print("number of NaN values for the column bathrooms :", df['bathrooms'].isnull().sum())
```

```
number of NaN values for the column bedrooms : 13
number of NaN values for the column bathrooms : 10
```

We can replace the missing values of the column `'bedrooms'` with the mean of the column `'bedrooms'` using the method `replace()`. Don't forget to set the `inplace` parameter to `True`

```
In [9]: mean=df['bedrooms'].mean()
df['bedrooms'].replace(np.nan,mean, inplace=True)
```

We also replace the missing values of the column `'bathrooms'` with the mean of the column `'bathrooms'` using the method `replace()`. Don't forget to set the `inplace` parameter top `True`

```
In [10]: mean=df['bathrooms'].mean()
df['bathrooms'].replace(np.nan,mean, inplace=True)
```

```
In [11]: print("number of NaN values for the column bedrooms :", df['bedrooms'].isnull().sum())
print("number of NaN values for the column bathrooms :", df['bathrooms'].isnull().sum())
```

```
number of NaN values for the column bedrooms : 0
number of NaN values for the column bathrooms : 0
```

## Module 3: Exploratory Data Analysis

### Question 3

Use the method `value_counts` to count the number of houses with unique floor values, use the method `.to_frame()` to convert it to a dataframe.

```
In [14]: floor_values = df['floors'].value_counts()  
floor_values.to_frame()
```

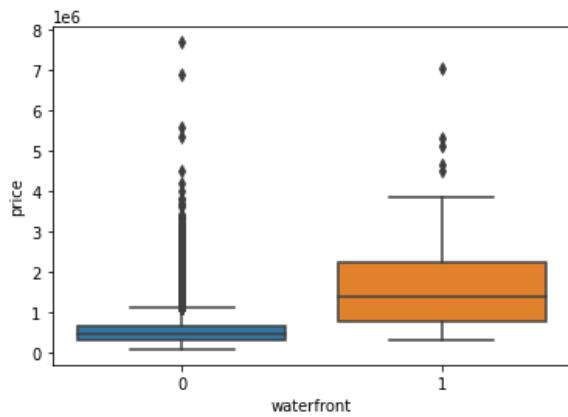
```
Out[14]:    floors  
1.0      10680  
2.0      8241  
1.5      1910  
3.0       613  
2.5       161  
3.5        8
```

## Question 4

Use the function boxplot in the seaborn library to determine whether houses with a waterfront view or without a waterfront view have more price outliers.

```
In [15]: sns.boxplot(x='waterfront', y='price', data=df)
```

```
Out[15]: <AxesSubplot:xlabel='waterfront', ylabel='price'>
```

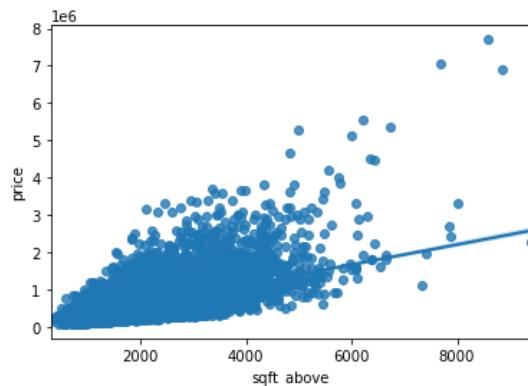


## Question 5

Use the function regplot in the seaborn library to determine if the feature sqft\_above is negatively or positively correlated with price.

```
In [16]: sns.regplot(x='sqft_above', y='price', data=df)
```

```
Out[16]: <AxesSubplot:xlabel='sqft_above', ylabel='price'>
```



We can use the Pandas method corr() to find the feature other than price that is most correlated with price.

```
In [17]: df.corr()['price'].sort_values()
```

```
Out[17]: zipcode      -0.053203
          long         0.021626
          condition    0.036362
          yr_built     0.054012
          sqft_lot15   0.082447
          sqft_lot      0.089661
          yr_renovated 0.126434
          floors        0.256794
          waterfront    0.266369
          lat           0.307003
          bedrooms      0.308797
          sqft_basement 0.323816
          view          0.397293
          bathrooms      0.525738
          sqft_living15 0.585379
          sqft_above     0.605567
          grade          0.667434
          sqft_living    0.702035
          price          1.000000
          Name: price, dtype: float64
```

## Module 4: Model Development

We can Fit a linear regression model using the longitude feature 'long' and calculate the R^2.

```
In [18]: X = df[['long']]
          Y = df['price']
          lm = LinearRegression()
          lm.fit(X,Y)
          lm.score(X, Y)
```

```
Out[18]: 0.00046769430149007363
```

## Question 6

Fit a linear regression model to predict the 'price' using the feature 'sqft\_living' then calculate the R<sup>2</sup>. Take a screenshot of your code and the value of the R<sup>2</sup>.

```
In [36]: X = df[['sqft_living']]  
Y = df['price']  
lm = LinearRegression()  
lm.fit(X, Y)  
lm.score(X, Y)
```

```
Out[36]: 0.4928532179037931
```

## Question 7

Fit a linear regression model to predict the 'price' using the list of features:

```
In [34]: features =["floors", "waterfront","lat" , "bedrooms" , "sqft_basement" , "view" , "bathrooms", "sqft_living15", "sqft_above", "grade", "sqft_living"]
```

Then calculate the R<sup>2</sup>. Take a screenshot of your code.

Then calculate the R<sup>2</sup>. Take a screenshot of your code.

```
In [38]: X = df[features]  
Y = df['price']  
lm = LinearRegression()  
lm.fit(X, Y)  
lm.score(X, Y)
```

```
Out[38]: 0.6576569675583581
```

## This will help with Question 8

Create a list of tuples, the first element in the tuple contains the name of the estimator:

'scale'

'polynomial'

'model'

The second element in the tuple contains the model constructor

StandardScaler()

PolynomialFeatures(include\_bias=False)

LinearRegression()

```
In [42]: Input=[('scale',StandardScaler()),('polynomial', PolynomialFeatures(include_bias=False)),('model',LinearRegression())]
```

## Question 8

Use the list to create a pipeline object to predict the 'price', fit the object using the features in the list features, and calculate the R^2.

```
In [43]: pipe = Pipeline(Input)
pipe
```

```
Out[43]: Pipeline(steps=[('scale', StandardScaler()),
 ('polynomial', PolynomialFeatures(include_bias=False)),
 ('model', LinearRegression())])
```

```
In [44]: pipe.fit(X, Y)
```

```
Out[44]: Pipeline(steps=[('scale', StandardScaler()),
 ('polynomial', PolynomialFeatures(include_bias=False)),
 ('model', LinearRegression())])
```

```
In [45]: pipe.score(X, Y)
```

```
Out[45]: 0.7513417707683823
```

## Module 5: Model Evaluation and Refinement

Import the necessary modules:

```
In [46]: from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
print("done")
```

done

We will split the data into training and testing sets:

```
In [47]: features =["floors", "waterfront","lat", "bedrooms" , "sqft_basement" , "view" , "bathrooms", "sqft_living15", "sqft_above", "grade", "sqft_living"]
X = df[features]
Y = df['price']

x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.15, random_state=1)

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

number of test samples: 3242  
number of training samples: 18371

## Question 9

Create and fit a Ridge regression object using the training data, set the regularization parameter to 0.1, and calculate the R<sup>2</sup> using the test data

```
In [48]: from sklearn.linear_model import Ridge
```

```
In [50]: RidgeModel = Ridge(alpha=0.1)
RidgeModel.fit(x_train, y_train)
RidgeModel.score(x_test, y_test)
```

```
Out[50]: 0.6478759163939113
```

## Question 10

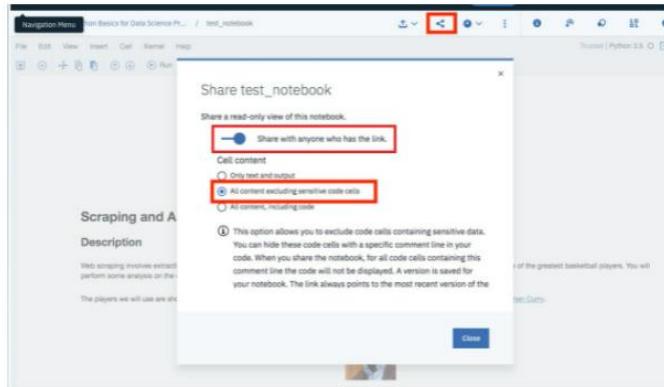
Perform a second order polynomial transform on both the training data and testing data. Create and fit a Ridge regression object using the training data, set the regularisation parameter to 0.1, and calculate the R<sup>2</sup> utilising the test data provided. Take a screenshot of your code and the R<sup>2</sup>.

```
In [51]: pr = PolynomialFeatures(degree=2)
x_train_pr = pr.fit_transform(x_train)
x_test_pr = pr.fit_transform(x_test)

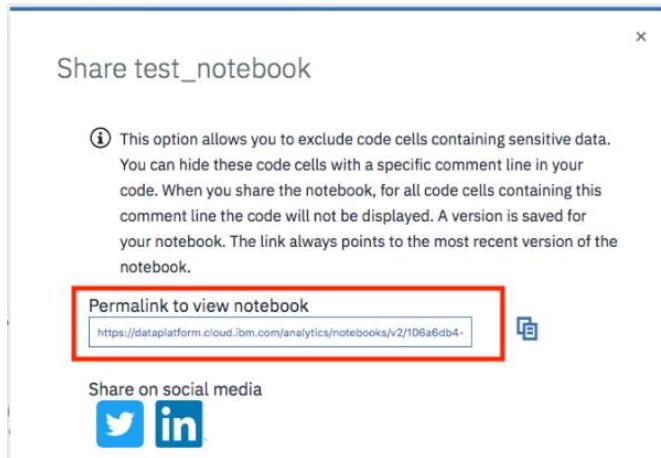
RidgeModel = Ridge(alpha=0.1)
RidgeModel.fit(x_train_pr, y_train)
RidgeModel.score(x_test_pr, y_test)
```

```
Out[51]: 0.7002744273468813
```

Once you complete your notebook you will have to share it. Select the icon on the top right marked in red in the image below, a dialogue box should open, and select the option all content excluding sensitive code cells.



You can then share the notebook via a URL by scrolling down as shown in the following image:



## 7.5. Model Evaluation

Seongjoo Brenden Song edited this page on Nov 6, 2021 · 2 revisions

---

### Learning Objectives

- Describe data model refinement techniques
  - Explain overfitting, underfitting and model selection
  - Apply ridge regression to regularize and reduce the standard errors to avoid overfitting a regression model
  - Apply grid search techniques to Python data
- 

- [Model Evaluation and Refinement](#)
- [Lab 5: Model Evaluation and Refinement](#)

## 7.5.1. Model Evaluation and Refinement

### 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

- Split dataset into:
  - Training set (70%)
  - Testing set (30%)
- Build and train the model with a training get
- 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,
```

- `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 user for random sampling
-

## Generalization Performance

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

## Generalization Error

### 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)
```

### Question

consider the following lines of code, how many partitions or folds are used in the function `cross_val_score`:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(lr, x_data, y_data, cv=10)
```

- 4
- 10
- 5

*Correct*

## Practice Quiz: Model Evaluation

---

### Question 1

What is the correct use of the "train\_test\_split" function such that 90% of the data samples will be utilized for training, the parameter "random\_state" is set to zero, and the input variables for the features and targets are x\_data, y\_data respectively.

- `~~train_test_split(x_data, y_data, test_size=0.9, random_state=0)~~`
- `**train_test_split(x_data, y_data, test_size=0.1, random_state=0)**`

*Correct*

## Overfitting, Underfitting and Model Selection

---

### Model Selection

$$y(x) + \text{noise}$$

$$y = b_0 + b_1x$$

$$y = b_0 + b_1x + b_2x^2$$

$$\hat{y} = b_0 + b_1x + b_2x^2 + b_3x^3 + b_4x^4 + b_5x^5 + b_6x^6 + b_7x^7 + b_8x^8$$

$$\begin{aligned}\hat{y} = & b_0 + b_1x + b_2x^2 + b_3x^3 + b_4x^4 + b_5x^5 + b_6x^6 + b_7x^7 + b_8x^8 + b_9x^9 \\ & b_{10}x^{10} + b_{11}x^{11} + b_{12}x^{12} + b_{13}x^{13} + b_{14}x^{14} + b_{15}x^{15} + b_{16}x^{16}\end{aligned}$$

### Model Selection

- We select the order that minimizes the test error. In this case, it was **eight**.
- Anything on the **left** would be considered **underfitting**. Anything on the **right** is **overfitting**.

### Question

True or False, the following plot shows that as the order of the polynomial increases the mean square error of our model decreases on the test data:

- False
- True

**Correct.** This plot shows the training error

### Practice Quiz: Overfitting, Underfitting and Model Selection

#### TOTAL POINTS 1

##### Question 1

In the following plot, the vertical axis shows the mean square error and the horizontal axis represents the order of the polynomial. The red line represents the training error the blue line is the test error. Should you select the 16 order polynomial.

- no
- yes

**Correct.** We use the test error to determine the model error. For this order of the polynomial, the training error is smaller but the test error is larger.

## Ridge Regression

Ridge regression is a regression that is employed in a Multiple regression model when Multicollinearity occurs. Multicollinearity is when there is a strong relationship among the independent variables. Ridge regression is very common with polynomial regression.

- Overfitting is a big problem when you have multiple independent variables, or features.
- The estimated function in blue does a good job at approximating the true function.
- In many cases real data has outliers. For example, the blue point shown above does not appear to come from the function in orange. If we use a tenth order polynomial function to fit the data, the estimated function in blue is incorrect, and is not a good estimate of the actual function in orange.

Ridge regression is a regression that is employed in a Multiple regression model when Multicollinearity occurs. Multicollinearity is when there is a strong relationship among the independent variables. Ridge regression is very common with polynomial regression.

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

- Overfitting is a big problem when you have multiple independent variables, or features.
- The estimated function in blue does a good job at approximating the true function.

- In many cases real data has outliers. For example, the blue point shown above does not appear to come from the function in orange. If we use a tenth order polynomial function to fit the data, the estimated function in blue is incorrect, and is not a good estimate of the actual function in orange.

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

- If we examine the expression for the estimated function, we see the estimated polynomial coefficients have a very large magnitude. This is especially evident for the higher order polynomials.
- Ridge regression controls the magnitude of these polynomial coefficients by introducing the parameter alpha. Alpha is a parameter we select before fitting or training the model. Each row in the following table represents an increasing value of alpha. Let's see how different values of alpha change the model. This table represents the polynomial coefficients for different values of alpha.
- The column corresponds to the different polynomial coefficients, and the **rows** correspond to the **different values of alpha**. As alpha increases, the parameters get smaller. This is most evident for the higher order polynomial features. But Alpha must be selected carefully. If alpha is too large, the coefficients will approach zero and underfit the data. If alpha is zero, the overfitting is evident.
- For alpha equal to 0.001, the overfitting begins to subside. For Alpha equal to 0.01, the estimated function tracks the actual function. When alpha equals one, we see the first signs of underfitting. The estimated function does not have enough flexibility. At alpha equals to 10, we see extreme underfitting. It does not even track the two points. In order to select alpha, we use cross validation.

## Question

Consider the following fourth order polynomial, fitted with Ridge Regression; should we increase or decrease the parameter alpha?

- Decrease
- Increase

**Correct.** The model seems to be underfitting the data we should decrease the value of the parameter.

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

- To make a prediction using ridge regression, import ridge from sklearn.linear\_models. Create a ridge object using the constructor. The parameter alpha is one of the arguments of the

constructor. We train the model using the fit method. To make a prediction, we use the predict method.

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

## Practice Quiz: Ridge Regression

### TOTAL POINTS 1

#### Question 1

the following models were all trained on the same data, select the model with the highest value for alpha:

- a
- b
- c

**Correct.** **a, b** - The model that exhibits the "most" underfitting is usually the model with the highest parameter value for alpha. **c** - The model that exhibits overfitting is usually the model with the lowest parameter value for alpha

## Grid Search

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

## Hyperparameters

- The term alpha in Ridge regression is called a hyperparameter
- Scikit-learn has a means of automatically iterating over these hyperparameters using cross-validation called Grid Search

## Question

What data do we use to pick the best hyperparameter

- Training data
- Validation data
- Test data

**Correct.** The training data is used to get the model parameters, not the hyperparameters

```
parameters = [{'alpha': [1, 10, 100, 1000]}]
```

The value of your Grid Search is a Python list that contains a Python dictionary.

`**'alpha'**` : The key is the name of the free parameter.

`**[1, 10, 100, 1000]**` : The value of the dictionary is the different values of the free parameter.

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

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

What are the advantages of Grid Search is how quickly we can test multiple parameters.

For example, ridge regression has the option to normalize the data.

```
parameters = [ {'alpha': [1, 10, 100, 1000], 'normalize': [True, False]} ]
```

`**'alpha': [1, 10, 100, 1000]**` : The term alpha is the first element in the dictionary.

`**'normalize':[True, False]**` : The second element is the normalized option.

`**'normalize'**` : The key is the name of the parameter.

`**[True, False]**` : The value is the different options in this case because we can either normalize the data or not. The values are True or False respectively.

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

As before, we need the ridge regression object or model. The procedure is similar except that we have a table or grid of different parameter values. The output is the score for all the different combinations of parameter values. The code is also similar.

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

parameters2 = [ {'alpha': [1, 10, 100, 1000], 'normalize': [True, False]} ]

RR = Ridge()

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

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

Grid1.best_estimator_

scores = Grid1.cv_results_
```

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

```
for param, mean_val, mean_test in zip(score['params'], scores['mean_test_score'],

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

## Question

how many types of parameters does the following dictionary contain:

```
parameters= [{"alpha": [0.001, 0.1, 1, 10, 100], "normalize": [True, False]}]
```

- 2
- 9
- 4

## Correct

### Lesson Summary

In this lesson, you have learned how to:

**Identify over-fitting and under-fitting in a predictive model:** Overfitting occurs when a function is too closely fit to the training data points and captures the noise of the data. Underfitting refers to a model that can't model the training data or capture the trend of the data.

**Apply Ridge Regression to linear regression models:** Ridge regression is a regression that is employed in a Multiple regression model when Multicollinearity occurs.

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

## 7.5.2.Lab 5: Model Evaluation and Refinement

```
**import** pandas **as** pd
**import** numpy **as** np

*# Import clean data*
path ***=*** 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DA0101EN-SkillNetwork/module_5_auto.csv'
df ***=*** pd.read_csv(path)

df.to_csv('module_5_auto.csv')
```

First, let's only use numeric data:

```
df ***=*** df._get_numeric_data()
df.head()
```

Libraries for plotting:

```
***%capture
%%capture
!pip install ipywidgets

from ipywidgets import interact, interactive, fixed, interact_manual
```

### Functions for Plotting

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

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

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

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

```

**def** PollyPlot(xtrain, xtest, y_train, y_test, lr,poly_transform):
    width **=** 12
    height **=** 10
    plt.figure(figsize**=(width, height))

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

    xmax**=**max([xtrain.values.max(), xtest.values.max()])
    xmin**=**min([xtrain.values.min(), xtest.values.min()])
    x**=**np.arange(xmin, xmax, 0.1)

    plt.plot(xtrain, y_train, 'ro', label**='Training Data')
    plt.plot(xtest, y_test, 'go', label**='Test Data')
    plt.plot(x, lr.predict(poly_transform.fit_transform(x.reshape(**-**1, 1))),

    plt.ylim([-10000, 60000])
    plt.ylabel('Price')
    plt.legend()

label**='Predicted Function')

```

## Part 1: Training and Testing

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

```
y_data *** df['price']
```

Drop price data in dataframe `x_data`:

```
x_data***df.drop('price',axis***1)
```

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

```
**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***=**0.10,  
  
print("number of test samples :", x_test.shape[0])  
print("number of training samples:",x_train.shape[0])
```

```
number of test samples : 21  
number of training samples: 180
```

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

### Question #1):

Use the function "train\_test\_split" to split up the dataset such that 40% of the data samples will be utilized for testing. Set the parameter "random\_state" equal to zero. The output of the function should be the following: "`x_train1`", "`x_test1`", "`y_train1`" and "`y_test1`".

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

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

Let's import LinearRegression from the module linear\_model.

```
**from** sklearn.linear_model **import** LinearRegression
```

We create a Linear Regression object:

```
lre**=**LinearRegression()
```

We fit the model using the feature "horsepower":

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

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

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

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

```
0.36358755750788263
```

We can see the R^2 is much smaller using the test data compared to the training data.

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

```
0.6619724197515104
```

## Question #2):

Find the R^2 on the test data using 40% of the dataset for testing.

```
x_train1, x_test1, y_train1, y_test1 **=** train_test_split(x_data, y_data, test_size**=**0.4,  
lre.fit(x_train1[['horsepower']],y_train1)  
lre.score(x_test1[['horsepower']],y_test1)
```

```
0.7139364665406973
```

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

## Cross-Validation Score

---

Let's import `model_selection` from the module `cross_val_score`.

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

We input the object, the feature ("horsepower"), and the target data (`y_data`). The parameter 'cv' determines the number of folds. In this case, it is 4.

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

The default scoring is  $R^2$ . Each element in the array has the average  $R^2$  value for the fold:

```
Rcross
```

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

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

```
print("The mean of the folds are", Rcross.mean(),
      "and the standard deviation is" , Rcross.std())
```

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

We can use negative squared error as a score by setting the parameter 'scoring' metric to 'neg\_mean\_squared\_error'.

```
- 1 ***** cross_val_score(lre,x_data[['horsepower']],
                           y_data, cv**=**4, scoring**=**'r
```

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

```
y_data, cv**=**4, scoring**=**'neg_mean_squared_error')
```

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

## Question #3):

Calculate the average R<sup>2</sup> using two folds, then find the average R<sup>2</sup> for the second fold utilizing the "horsepower" feature:

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

```
0.5166761697127429
```

You can also use the function 'cross\_val\_predict' to predict the output. The function splits up the data into the specified number of folds, with one fold for testing and the other folds are used for training. First, import the function:

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

We input the object, the feature "horsepower", and the target data `y_data`. The parameter 'cv' determines the number of folds. In this case, it is 4. We can produce an output:

```
yhat ***=** cross_val_predict(lre,x_data[['horsepower']], y_data, cv***=**4)
```

```
yhat[0:5]
```

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

## Part 2: Overfitting, Underfitting and Model Selection

It turns out that the test data, sometimes referred to as the "out of sample data", is a much better measure of how well your model performs in the real world. One reason for this is overfitting.

Let's go over some examples. It turns out these differences are more apparent in Multiple Linear Regression and Polynomial Regression so we will explore overfitting in that context.

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

```
lr ***=** LinearRegression()

lr.fit(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y_train)

LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

> prediction using training data:

yhat_train ***=** lr.predict(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])

yhat_train[0:5]

array([ 7426.6731551 , 28323.75090803, 14213.38819709,  4052.34146983, 34500.19124244])
```

```
lr ***=** LinearRegression()

lr.fit(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y_train)

LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Prediction using training data:

```
yhat_train ***=** lr.predict(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])

yhat_train[0:5]

array([ 7426.6731551 , 28323.75090803, 14213.38819709,  4052.34146983, 34500.19124244])
```

Prediction using test data:

```
yhat_test ***=** lr.predict(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])

yhat_test[0:5]

array([11349.35089149,  5884.11059106, 11208.6928275 ,  6641.07786278, 15565.79920282])
```

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

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

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

```
Title **=** 'Distribution Plot of Predicted Value Using Training Data vs Training Data Distribution'  
DistributionPlot(y_train, yhat_train, "Actual Values (Train)", "Predicted Values (Train)", Title)
```

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

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

```
Title**=**'Distribution Plot of Predicted Value Using Test Data vs Data Distribution of Test Data'  
DistributionPlot(y_test,yhat_test,"Actual Values (Test)","Predicted Values (Test)",Title)
```

Figure 2: Plot of predicted value using the test data compared to the actual values of the test data.

Comparing Figure 1 and Figure 2, it is evident that the distribution of the test data in Figure 1 is much better at fitting the data. This difference in Figure 2 is apparent in the range of 5000 to 15,000. This is where the shape of the distribution is extremely

Figure 2: Plot of predicted value using the test data compared to the actual values of the test data.

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

```
**from** sklearn.preprocessing **import** PolynomialFeatures
```

## Overfitting

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

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

```
x_train, x_test, y_train, y_test ***=*** train_test_split(x_data, y_data, test_size***=***0.45, random_state***=***0)
```

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

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

```
PolynomialFeatures(degree=5, include_bias=True, interaction_only=False)
```

Now, let's create a Linear Regression model "poly" and train it.

```
poly ***=*** LinearRegression()  
poly.fit(x_train_pr, y_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

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

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

```
array([ 6728.68465468,  7308.01690973, 12213.81302023, 18893.19052853, 19995.88231726])
```

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

```
print("Predicted values:", yhat[0:4])  
print("True values:", y_test[0:4].values)  
  
Predicted values: [ 6728.68465468  7308.01690973 12213.81302023 18893.19052853]  
True values: [ 6295. 10698. 13860. 13499.]
```

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

```
PollyPlot(x_train[['horsepower']], x_test[['horsepower']], y_train, y_test, poly,pr)
```

Figure 3: A polynomial regression model where red dots represent training data, green dots represent test data, and the blue line represents the model prediction.

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

R<sup>2</sup> of the training data:

```
poly.score(x_train_pr, y_train)
```

```
0.556771690250259
```

R<sup>2</sup> of the test data:

```
poly.score(x_test_pr, y_test)
```

```
- 29.871506261647205
```

We see the R<sup>2</sup> for the training data is 0.5567 while the R<sup>2</sup> on the test data was -29.87. The lower the R<sup>2</sup>, the worse the model. A negative R<sup>2</sup> is a sign of overfitting.

Let's see how the R<sup>2</sup> changes on the test data for different order polynomials and then plot the results:

```

Rsqu_test **=** []

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

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

    lr.fit(x_train_pr, y_train)

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

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

```

```
Text(3, 0.75, 'Maximum R^2 ')
```

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

The following function will be used in the next section. Please run the cell below.

```

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

```

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

```
interact(f, order**=**([0, 6, 1]), test_data**=**([0.05, 0.95, 0.05]))
```

```

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

```

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

```
interact(f, order**=*(0, 6, 1), test_data**=*(0.05, 0.95, 0.05))
```

Output

This XML file does not appear to have any style information associated with it. The document tree is shown below.

---

```

▼<Error>
  <Code>AccessDenied</Code>
  <Message>Request has expired</Message>
  <X-Amz-Expires>86400</X-Amz-Expires>
  <Expires>2021-11-06T18:17:16Z</Expires>
  <ServerTime>2022-02-06T00:07:15Z</ServerTime>
  <RequestId>7HG7NHX4FJKKPE1S</RequestId>
  <HostId>mNqf0TfzDw2udY81t4eKRu0C73LWGfmVmBLxjE3F5gFV2WL701p6vsBxm08w7SfV5auvdjpqLw=</HostId>
</Error>

```

## Question #4a):

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

```
pr1**=**PolynomialFeatures(degree**=**2)
```

## Question #4b):

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

```
x_train_pr1**=**pr1.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])  
x_test_pr1**=**pr1.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
```

## Question #4c):

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

```
x_train_pr1.shape *#there are now 15 features*
```

```
(110, 15)
```

## Question #4d):

Create a linear regression model "poly1". Train the object using the method "fit" using the polynomial features.

```
poly1**=**LinearRegression().fit(x_train_pr1,y_train)
```

```
poly1**=**LinearRegression().fit(x_train_pr1,y_train)
```

## Question #4e):

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

```
yhat_test1**=**poly1.predict(x_test_pr1)  
  
Title**=**'Distribution Plot of Predicted Value Using Test Data vs Data Distribution of Test Data'  
  
DistributionPlot(y_test, yhat_test1, "Actual Values (Test)", "Predicted Values (Test)", Title)
```

## Question #4f):

Using the distribution plot above, describe (in words) the two regions where the predicted prices are less accurate than the actual prices.

- The predicted value is higher than actual value for cars where the price \$10,000 range, conversely the predicted price is lower than the price cost in the \$30,000 to \$40,000 range. As such the model is not as accurate in these ranges.

## Part 3: Ridge Regression

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

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

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

x_train_pr***=**pr.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-size',

x_test_pr***=**pr.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size',
```

Let's import Ridge from the module linear models.

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

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

```
RidgeModel***=**Ridge(alpha***=**1)
```

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

```
RidgeModel.fit(x_train_pr, y_train)
```

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

```
RidgeModel.fit(x_train_pr, y_train)
```

```
Ridge(alpha=1, copy_X=True, fit_intercept=True, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001)
```

Similarly, you can obtain a prediction:

```
yhat *** RidgeModel.predict(x_test_pr)
```

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

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

```
predicted: [ 6570.82441941  9636.2489147  20949.92322737 19403.60313256]
test set : [ 6295. 10698. 13860. 13499.]
```

We select the value of alpha that minimizes the test error. To do so, we can use a for loop. We have also created a progress bar to see how many iterations we have completed so far.

```
**from** tqdm **import** tqdm

Rsqu_test *** []
Rsqu_train *** []
dummy1 *** []
Alpha *** 10 ***** np.array(range(0,1000))
pbar *** tqdm(Alpha)

**for** alpha ***in*** pbar:
    RidgeModel *** Ridge(alpha***alpha)
    RidgeModel.fit(x_train_pr, y_train)
    test_score, train_score *** RidgeModel.score(x_test_pr, y_test), RidgeModel.score(x_train_pr, y_train)

    pbar.set_postfix({"Test Score": test_score, "Train Score": train_score})

    Rsqu_test.append(test_score)
    Rsqu_train.append(train_score)
```

```
100%|██████████| 1000/1000 [02:10<00:00,  7.65it/s, Test Score=0.564, Train Score=0.859]
```

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

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

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

```
<matplotlib.legend.Legend at 0x7f38a8e104e0>
```

**Figure 4:** The blue line represents the R^2 of the validation data, and the red line represents the R^2 of the training data. The x-axis represents the different values of Alpha.

Here the model is built and tested on the same data, so the training and test data are the same.

The red line in Figure 4 represents the R^2 of the training data. As alpha increases the R^2 decreases. Therefore, as alpha increases, the model performs worse on the training data

The blue line represents the R^2 on the validation data. As the value for alpha increases, the R^2 increases and converges at a point.

## Question #5):

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

```
RidgeModel = Ridge(alpha=10)
RidgeModel.fit(x_train_pr, y_train)
RidgeModel.score(x_test_pr, y_test)
```

```
0.5418576440207993
```

## Part 4: Grid Search

The term alpha is a hyperparameter. Sklearn has the class **GridSearchCV** to make the process of finding the best hyperparameter simpler.

Let's import **GridSearchCV** from the module **model\_selection**.

```
**from** sklearn.model_selection **import** GridSearchCV
```

We create a dictionary of parameter values:

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

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

Create a Ridge regression object:

```
RR**=**Ridge()  
RR
```

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

Create a ridge grid search object:

```
Grid1 ***=** GridSearchCV(RR, parameters1, cv***=**4, iid**=None**)
```

In order to avoid a deprecation warning due to the iid parameter, we set the value of iid to "None".

Fit the model:

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

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

```
GridSearchCV(cv=4, error_score='raise-deprecating',
            estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
                           fit_params=None, iid=None, n_jobs=None,
                           param_grid=[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 1000000]}],
                           pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
                           scoring=None, verbose=0)
```

```
normalize=False, random_state=None, solver='auto', tol=0.001),
```

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

```
BestRR***=**Grid1.best_estimator_
BestRR
```

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

We now test our model on the test data:

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

```
0.8411649831036152
```

**Question #6):** Perform a grid search for the alpha parameter and the normalization parameter, then find the best values of the parameters:

```
parameters2 ***=** [{"alpha": [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 1000000], "normalize": [True, False]}] ]
Grid2 ***=** GridSearchCV(Ridge(), parameters2, cv=4)
Grid2.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y_data)
Grid2.best_estimator_
```

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

## 7.6.2.Final Exam

### Final Exam

LATEST SUBMISSION GRADE 100%

#### Question 1

What does **csvfile** stand for?

- ~~car seller values~~
- **comma separated values**

*Correct*

#### Question 2

Scikit-learn is used for?

- **Statistical modelling including regression and classification.**
- ~~Fast array processing.~~

*Correct*

#### Question 3

What tells us the way the data is encoded?

- ~~Data path~~
- ~~File path~~
- **Encoding scheme**

*Correct*

#### Question 4

What attribute or function returns the data types of each column?

- ~~head()~~
- ~~tail()~~
- **dtypes**

*Correct*

## Question 5

In a dataset what is the name of the columns?

- Type
- Row
- Header

*Correct*

## Question 6

The Matplotlib library is mostly used for what?

- Data analysis
- Machine learning
- Data visualization

*Correct*

## Question 7

What would the following code segment output from a dataframe `df`?

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

- It would return the first 5 rows of the dataframe
- It would return the last 5 rows of the dataframe
- It would return all of the rows of the dataframe

*Correct*

## Question 8

What does the following code segment perform in a dataframe?

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

- It drops rows that contain missing values
- It drops all of the rows in the column "normalized-losses"
- It replaces the missing values in the column "normalized-losses" with the mean of that column

*Correct*

## Question 9

How would you multiply each element in the column `df["a"]` by 2 and assign it back to the column `df["a"]`?

- `df["a"] = 2 * df["a"]`
- `2 * df["a"]`
- `df["a"] = df["a"] + 1`

*Correct*

## Question 10

What function returns the maximum of the values requested for the requested column?

- `max()`
- `min()`
- `std()`

*Correct*

## Question 11

Since most statistical models cannot take objects or strings as inputs, what action needs to be performed?

- Convert categorical variables into numerical values
- Convert numerical values into categorical variables

*Correct*

## Question 12

What function will change the name of a column in a dataframe?

- `rename()`
- `exchange()`
- `replace()`

*Correct*

# Final Exam on Analysis of Data with Python Feb. 2022

Final Exam due Feb 24, 2022 21:32 +08

## Question 1

1/1 point (graded)

What is the acronym for comma separated values?

csepv

cosv

csv



**Submit**

You have used 1 of 2 attempts

---

✓ Correct (1/1 point)

## Question 2

1/1 point (graded)

What Python library is used for statistical modelling including regression and classification?

Matplotlib

Scikit-learn

Numpy



**Submit**

You have used 1 of 2 attempts

---

✓ Correct (1/1 point)

## Question 3

1/1 point (graded)

What tells us the way the data is encoded?

Encoding scheme

File path

Data path



**Submit**

You have used 1 of 2 attempts

---

✓ Correct (1/1 point)

## Question 4

1/1 point (graded)

What does the tail() method return?

It returns the data types of each column

It returns the last five rows

It returns the first five rows



**Submit**

You have used 1 of 2 attempts

Correct (1/1 point)

## Question 5

1/1 point (graded)

In a dataset what is the name of the columns?

Type

Header

Row



**Submit**

You have used 1 of 2 attempts

**Submit**

You have used 1 of 2 attempts

---

✓ Correct (1/1 point)

---

## Question 6

1/1 point (graded)

The Scikit-learning library is mostly used for what?

 Machine learning Data visualization Data analysis**Submit**

You have used 1 of 2 attempts

---

✓ Correct (1/1 point)

---

## Question 7

1/1 point (graded)

What would the following code segment output from a dataframe df?

df.tail(10)

It would return all of the rows of the dataframe

It would return the last 10 rows of the dataframe

It would return the first 10 rows of the dataframe



**Submit**

You have used 1 of 2 attempts

Correct (1/1 point)

## Question 8

1/1 point (graded)

What does the following code segment perform in a dataframe?

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

It drops all of the rows in the column "normalized-losses"

It replaces the missing values in the column "normalized-losses" with the mean of that column

It drops rows that contain missing values



**Submit**

You have used 1 of 2 attempts

Save |

Correct (1/1 point)

## Question 9

1/1 point (graded)

How would you multiply each element in the column `df["c"]` by 5 and assign it back to the column `df["c"]`?

`df["a"] = df["c"] * 5`

`df["c"] = 5 * df["c"]`

`5 * df["b"]`



**Submit**

You have used 1 of 2 attempts

[Save](#)

[Show answer](#)

Correct (1/1 point)

## Question 10

1/1 point (graded)

What does the below code segment give an example of for the column "length"?

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

It gives an example of the max-min method

It gives an example of the z-score or standard score



**Submit**

You have used 1 of 1 attempt

Correct (1/1 point)

