

The Python Guide for New Data Scientists

By

Jeffrey Strickland

The Python Guide for New Data Scientists

Jeffrey Strickland

Copyright © 2022, Jeffrey S. Strickland

ISBN 978-1-4583-2161-9

This work is licensed under a Standard Copyright License. All rights reserved. Any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the author except for the use of brief quotations in a book review.

Printed in the United States of America



Acknowledgements

I would like to thank the faculty and students that I worked with at the Vellore Institute of Technology (VIT) in Vellore, India, from 2016 to 2018. My experience with them was extraordinary. VIT is the top private engineering university in India and they are kind enough to consider me one of their own. I would particularly like to acknowledge my friends Aswani Kumar Cherukuri, Professor & Dean School of Information Technology & Engineering (SITE), and Chandra Mouliswaran S., Assistant Professor (SG), SITE. Along with the students I taught there in September 2016 and 2018, they inspired this book. My love for India is not surpassed by my love for its people.

This is indeed India! ... The land of dreams and romance, of fabulous wealth and fabulous poverty, of splendour and rags, of palaces and hovels, of famine and pestilence, of genii and giants and Aladdin lamps, of tigers and elephants, the cobra and the jungle, the country of hundred nations and a hundred tongues, of a thousand religions and two million gods, cradle of the human race, birthplace of human speech, mother of history, grandmother of legend, great-grandmother of traditions, whose yesterday's bear date with the moderating antiquities for the rest of nations-the one sole country under the sun that is endowed with an imperishable interest for alien prince and alien peasant, for lettered and ignorant, wise and fool, rich and poor, bond and free, the one land that all men desire to see, and having seen once, by even a glimpse, would not give that glimpse for the shows of all the rest of the world combined.

— Mark Twain

I owe a great deal of gratitude to my family for their support and endurance, Laurie, Mariah, and Evie. This is my 35th book along this life journey.

Table of Contents

1. DATA WRANGLING	1
OBJECTIVES	1
AUTOMOBILE DATASET.....	1
ACQUIRING DATA.....	1
IMPORTING DATA.....	2
ADDING HEADERS.....	5
READING & SAVING MULTIPLE FORMATS.....	8
DATA TYPES.....	8
DESCRIBING DATAFRAMES	10
DATAFRAME INFORMATION.....	12
DATA WRANGLING?	13
<i>Identifying and Imputing Missing Values</i>	13
<i>Data Standardization</i>	20
<i>Data Normalization</i>	23
<i>Data Binning</i>	24
<i>Indicator Variable (or DummyVariable)</i>	29
2. EXPLORATORY DATA ANALYSIS.....	33
OBJECTIVES	33
IMPORT DATA.....	33
ANALYZING FEATURE USING VISUALIZATION	34
CONTINUOUS NUMERICAL VARIABLES:	36
<i>Positive Linear Relationship</i>	36
<i>Weak Linear Relationship</i>	38
CATEGORICAL VARIABLES	40
DESCRIPTIVE STATISTICAL ANALYSIS	43
VALUE COUNTS.....	44
BASICS OF GROUPING.....	45
CORRELATION AND CAUSATION	51
<i>Pearson Correlation</i>	51
<i>P-value</i>	52
<i>Wheel-Base vs. Price</i>	53
<i>Horsepower vs. Price</i>	53
<i>Length vs. Price</i>	53
<i>Width vs. Price</i>	54

<i>Curb-Weight vs. Price</i>	54
<i>Engine-Size vs. Price</i>	55
<i>Bore vs. Price</i>	55
<i>City-mpg vs. Price</i>	56
<i>Highway-mpg vs. Price</i>	56
ANOVA	56
<i>ANOVA: Analysis of Variance</i>	56
<i>Drive Wheels</i>	57
<i>Conclusion: Important Variables</i>	59
3. MODEL DEVELOPMENT	61
OBJECTIVES	61
SETUP.....	61
LINEAR AND MULTIPLE LINEAR REGRESSION.....	62
<i>Linear Regression</i>	62
<i>Multiple Linear Regression</i>	65
MODEL EVALUATION USING VISUALIZATION	67
<i>Regression Plot</i>	67
<i>Residual Plot</i>	70
<i>Multiple Linear Regression</i>	71
POLYNOMIAL REGRESSION AND PIPELINES	73
PIPELINE	77
MEASURES FOR IN-SAMPLE EVALUATION	79
<i>Model 1: Simple Linear Regression</i>	79
<i>Model 2: Multiple Linear Regression</i>	80
<i>Model 3: Polynomial Fit</i>	81
<i>MSE</i>	81
PREDICTION AND DECISION MAKING	81
<i>Prediction</i>	81
<i>Decision Making: Determining a Good Model Fit</i>	82
<i>Simple Linear Regression Model (SLR) vs Multiple Linear Regression Model (MLR)</i>	84
<i>Simple Linear Model (SLR) vs. Polynomial Fit</i>	84
<i>Multiple Linear Regression (MLR) vs. Polynomial Fit</i>	84
CONCLUSION	85
4. MODEL EVALUATION AND REFINEMENT	87
<i>Objectives</i>	87
<i>Functions for Plotting</i>	88

PART 1: TRAINING AND TESTING	89
<i>Cross-Validation Score</i>	92
PART 2: OVERFITTING, UNDERFITTING AND MODEL SELECTION.....	94
<i>Overfitting</i>	96
PART 3: CROSS-VALIDATION.....	103
<i>K-Folds Cross Validation:</i>	103
PART 4: RIDGE REGRESSION	105
PART 5: GRID SEARCH	108
5. DATA VISUALIZATION.....	111
<i>Objectives</i>	111
<i>Introduction</i>	111
<i>The Data</i>	111
PART 1: EXPLORING DATASETS WITH PANDAS	111
<i>pandas Basics</i>	112
<i>Filtering based on a criteria</i>	122
PART 2: VISUALIZING DATA USING MATPLOTLIB	125
<i>Matplotlib: Standard Python Visualization Library</i>	125
<i>Matplotlib.Pyplot</i>	125
<i>Plotting in pandas</i>	126
<i>Line Pots (Series/Dataframe)</i>	126
<i>Other Plots</i>	135
6. AREA PLOTS, HISTOGRAMS, & BAR PLOTS.....	137
<i>Objectives</i>	137
EXPLORING DATA WITH PANDAS & MATPLOTLIB	137
<i>Downloading and Prepping Data</i>	137
VISUALIZING DATA USING MATPLOTLIB.....	142
<i>Area Plots</i>	142
<i>Two types of plotting</i>	147
<i>Plotting Parameters</i>	148
<i>Histograms</i>	154
<i>Bar Charts (Dataframe)</i>	163
7. PIE CHARTS, BOX PLOTS, SCATTER PLOTS, AND BUBBLE PLOTS	173
<i>Objectives</i>	173
EXPLORING DATASETS WITH PANDAS AND MATPLOTLIB	173
<i>Downloading and Prepping Data</i>	173
VISUALIZING DATA USING MATPLOTLIB.....	175

PIE CHARTS	177
Box PLOTS	185
<i>Customizing your objects</i>	190
<i>Object containers</i>	191
<i>Figure container</i>	191
<i>Axes container</i>	193
<i>Axis containers</i>	195
<i>Tick containers</i>	196
SCATTER PLOTS	214
8. WAFFLE CHARTS, WORD CLOUDS, AND REGRESSION PLOTS	227
<i>Objectives</i>	227
EXPLORING DATASETS WITH PANDAS AND MATPLOTLIB.....	227
<i>Downloading and Prepping Data</i>	227
VISUALIZING DATA USING MATPLOTLIB.....	229
WAFFLE CHARTS.....	230
WORD CLOUDS	241
REGRESSION PLOTS	248
9. SIMPLE LINEAR REGRESSION.....	259
<i>Objectives</i>	259
<i>Importing Needed packages</i>	259
DOWNLOADING DATA	259
<i>Understanding the Data</i>	259
READING THE IN THE DATA	260
DATA EXPLORATION	260
PRACTICE.....	264
<i>Simple Regression Model</i>	266
EXAMPLE	269
10. MULTIPLE LINEAR REGRESSION	271
<i>Downloading Data</i>	271
<i>Understanding the Data</i>	271
<i>Reading the in the Data</i>	272
<i>Creating train and test dataset</i>	274
<i>Train data distribution</i>	275
MULTIPLE REGRESSION MODEL.....	276
<i>Ordinary Least Squares (OLS)</i>	276
PREDICTION	277

<i>Variance Regression Score Explained</i>	277
11. POLYNOMIAL REGRESSION	279
<i>Objectives</i>	279
<i>Table of contents</i>	279
<i>Importing Needed packages</i>	279
DOWNLOADING DATA.....	279
<i>Understanding the Data</i>	279
<i>Reading the in the Data</i>	280
POLYNOMIAL REGRESSION.....	282
EVALUATION.....	285
EXAMPLE	286
12. NON-LINEAR REGRESSION ANALYSIS.....	289
<i>Objectives</i>	289
<i>Importing required libraries</i>	289
<i>Quadratic</i>	291
<i>Exponential</i>	292
<i>Logarithmic</i>	293
<i>Sigmoidal/Logistic</i>	294
<i>Non-Linear Regression example</i>	295
<i>Plotting the Dataset</i>	296
<i>Choosing a model</i>	297
<i>Building The Model</i>	298
<i>Finding the Best Fit</i>	299
13. K-NEAREST NEIGHBORS.....	303
OBJECTIVES	303
TABLE OF CONTENTS	304
ABOUT THE DATASET	304
DATA VISUALIZATION AND ANALYSIS	305
NORMALIZE DATA	307
TRAIN TEST SPLIT	307
CLASSIFICATION – K NEAREST NEIGHBOR (KNN).....	308
<i>Import library</i>	308
<i>Training</i>	308
<i>Predicting</i>	309
<i>Accuracy evaluation</i>	309
14. DECISION TREES	313

<i>Objectives</i>	313
<i>Table of contents</i>	313
ABOUT THE DATASET	313
DOWNLOADING THE DATA	314
PRE-PROCESSING	314
SETTING UP THE DECISION TREE.....	316
MODELING	317
PREDICTION.....	317
EVALUATION.....	318
VISUALIZATION.....	318
ENTROPY.....	319
15. LOGISTIC REGRESSION WITH PYTHON	323
<i>Objectives</i>	323
<i>Table of contents</i>	323
<i>Customer churn with Logistic Regression</i>	325
ABOUT THE DATASET	325
<i>Load the Telco Churn Data</i>	326
<i>Load Data from CSV File</i>	326
DATA PRE-PROCESSING AND SELECTION	327
<i>Example</i>	327
<i>Train/Test dataset</i>	328
MODELING (LOGISTIC REGRESSION WITH SCIKIT-LEARN).....	328
EVALUATION	330
<i>Jaccard index</i>	330
<i>confusion matrix</i>	330
<i>log loss</i>	334
16. SVM (SUPPORT VECTOR MACHINES).....	335
<i>Objectives</i>	335
<i>Table of contents</i>	335
LOAD THE CANCER DATA	335
<i>Load Data from CSV File</i>	336
MODELING (SVM WITH SCIKIT-LEARN).....	339
EVALUATION	340
<i>Classification Accuracy and its Limitations</i>	340
<i>What is a Confusion Matrix?</i>	341
EXAMPLE	345

17. K-MEANS CLUSTERING	347
<i>Objectives</i>	347
<i>Introduction</i>	347
<i>Table of contents</i>	347
<i>Import libraries</i>	348
K-MEANS ON A RANDOMLY GENERATED DATASET	348
SETTING UP K-MEANS.....	349
CREATING THE VISUAL PLOT	350
CUSTOMER SEGMENTATION WITH K-MEANS	353
<i>Pre-processing</i>	354
<i>Modeling</i>	355
<i>Insights</i>	356
18. HIERARCHICAL CLUSTERING	361
<i>Objectives</i>	361
<i>Table of contents</i>	361
HIERARCHICAL CLUSTERING - AGGLOMERATIVE	361
<i>Generating Random Data</i>	362
<i>Agglomerative Clustering</i>	363
<i>Dendrogram Associated for the Agglomerative Hierarchical Clustering</i>	365
<i>Clustering on Vehicle dataset</i>	368
<i>Download data</i>	368
<i>Read data</i>	369
<i>Data Cleaning</i>	369
<i>Feature selection</i>	370
<i>Normalization</i>	370
<i>Clustering using SciPy</i>	371
<i>Clustering using scikit-learn</i>	373
19. DENSITY-BASED CLUSTERING	381
<i>Objectives</i>	381
<i>Import the following libraries:</i>	381
DATA GENERATION	382
MODELING.....	382
<i>Distinguish outliers</i>	383
DATA VISUALIZATION.....	384
WEATHER STATION CLUSTERING USING DBSCAN & SCIKIT-LEARN	386
<i>Loading the data</i>	386

<i>About the dataset</i>	386
<i>1-Download data</i>	387
<i>2- Load the dataset</i>	387
<i>3-Cleaning</i>	388
<i>4-Visualization</i>	389
<i>5- Clustering of stations based on their location i.e., Lat & Lon</i>	392
<i>6- Visualization of clusters based on location</i>	393
<i>7- Clustering of stations based on their location, mean, max, and min Temperature</i>	395
<i>8- Visualization of Clusters Based on Location and Temperature</i>	395
20. CONTENT BASED FILTERING	399
<i>Objectives</i>	399
<i>Table of contents</i>	399
<i>Acquiring the Data</i>	399
<i>Preprocessing</i>	400
<i>Content-Based recommendation system</i>	404
<i>Advantages and Disadvantages of Content-Based Filtering</i>	410
21. COLLABORATIVE FILTERING	411
<i>Objectives</i>	411
<i>Table of contents</i>	411
ACQUIRING THE DATA	411
PREPROCESSING	412
COLLABORATIVE FILTERING	415
<i>Add movielid to input user</i>	416
<i>Users Who have Seen the Same Movies</i>	417
<i>Similarity of Users to Input User</i>	419
<i>Pearson Correlation</i>	419
<i>Calculating the Pearson Correlation</i>	419
<i>The Top X Similar Users to Input User</i>	422
<i>Rating of Selected Users to All Movies</i>	422
<i>Advantages & Disadvantages of Collaborative Filtering</i>	425
REFERENCES.....	427
INDEX.....	431

xi

Preface

To write a single book about data science, at least as I view the discipline, would result in several volumes, and it has. I have come to view Data Science kind of like Engineering. We have all sorts of engineers: mechanical, electrical, civil, aeronautical, industrial, and so on. We still have them, but when we talk about them, we tend to use the general term "engineers" and their field as "engineering." I have thought about this for a few years, and I have concluded that we do something similar with the terms "data scientists" and "data science." Although this book covers a lot of what I include as data science, it is written with the beginner in mind. I have written other books that are more specialized for more experienced users.

Data really powers everything that we do.

– Jeff Weiner, LinkedIn

What Comprises Data Science?

Programming (Computer Science). Data scientist convert data into a valuable format for investigation, inquiry, or analysis. This data conversion involves an extraction, transformation, and loading (ETL) effort to place data into a data repository, like a data warehouse or data lake. One of the data scientist's extremely valuable and sought-after skills is the capability to design, build, and execute computer code that structures data and manipulates unstructured data. Along with ETL, these three conceptual steps are the basis for the design and structure of most data pipelines. They serve as an outline or blueprint the way raw data are transformed to data that is ready for consumption.

Data Preprocessing (Information Technology). Many people who write about data science place preprocessing data in the field of information technology (IT), and indeed IT seems an appropriate. However, I now claim that this may be within the discipline of IT. The lines between IT and data science are blurred, enough so that the data scientist's primary training has been in IT. My data science

students at the Vellore Institute of Technology (VIT) were in the School of Information Technology & Engineering (SITE). Some were working on projects involving the writing of Hindi lexicons for text analytics using the Python programming language—quite different than my earlier image of IT.

Data Mining (Data Analysis, Statistics, Information Technology). Data mining is the semi-automatic or automatic process of taking large quantities of data and extracting data that is pertinent to an organization's data operations. Such mining pulls out interesting patterns such as groups of data records (cluster analysis); identifies items, events or observations which do not conform to an expected pattern (anomaly detection or outlier detection); finds frequent co-occurring associations among a collection of items (association rule mining or market basket analysis); and discovers statistically relevant patterns between data examples where the values are delivered in a sequence (sequential pattern mining). These mining tasks usually involve using database techniques to find patterns, which may be used in further analysis or, for example, in machine learning and predictive analytics. However, the data collection, data preparation, as well as the result interpretation and reporting is not part of the data mining step.

In God we trust. All others must bring data.

— W. Edwards Deming, statistician

Business Intelligence (BI Engineering, BI Development, BI Analysis). Business intelligence (BI), often referred to as business analytics, pursues the transform of data into actionable intelligence that informs an organization's strategic and tactical business decisions. In BI, analysts use software tools, such as Tableau and SAS, to access and analyze pertinent business data and present the results, in a variety of "story telling" forms, to provide stakeholders with detailed intelligence about the state of the business. Some experts distinguish between BI and business analytics, but I treat them as the same, with the additional idea that they are a specialized area of data analytics. Consequently, BI may describe a past or current state of the

business and it also analyzes data to predict what will happen or what could happen by taking a certain approach.

The big technology trend is to make systems intelligent and data is the raw material.

— Amod Malviya, CTO at flipkart

Machine Learning (Data Science, Statistics, Computer Science, Applied Physic, Biomedicine, Cognitive Science). Machine learning (ML) is not the same thing as artificial intelligence (AI). ML is the science of getting computers to act without being explicitly programmed (the domain of AI). In fact, many researchers think that ML is the best way to make progress towards human-level AI. In data science, ML or the algorithms of ML are used to mine data, explore mined data, and predict future outcomes (among other things), and can be considered as a part of data analytics or data mining. I make a distinction here to emphasize that ML does not make use of traditional statistical or mathematics methods. ML algorithms may include artificial neural networks (ANN), random forests (RF), genetic algorithms (GA), and many other methods. ML is particularly useful in text analytics, where statistical methods are inappropriate.

I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted.

— Alan Turing, Computing machinery and intelligence

Data Analytics (Data Scientist, Field Specific Data Analysis). Data analytics is a process of examining, cleaning, transforming, and modeling data with the goal of discovering useful information, informing conclusions, and providing decision support. Data analytics encompasses multiple methods and approaches within the realms of descriptive, predictive, and prescriptive analysis, while being used in different business, science, medical, psychological, and social science domains. The goals of data analytics are discovering useful information, informing conclusions, and supporting decision-

making. Once the data is cleaned, data analysts apply a variety of techniques, referred to as exploratory data analysis, to begin understanding the information contained in the data. Data exploration can result in additional data cleaning or additional data requirements, so these activities may be iterative. In descriptive analytics, unfolding the characteristics of the data with such measures as the mean and variance may help in understanding the data. Predictive analytics is concerned with forecasting future events based on the data that has been mined and explored. Prescriptive analytics is concerned with telling the story of why the data describes the present or predicts the future to help decision makers choose the best courses of action.

Descriptive Analytics, which use data aggregation and data mining to provide insight into the past and answer: "What has happened?"

Predictive Analytics, which use statistical models and forecasts techniques to understand the future and answer: "What could happen?"

The goal is to turn data into information, and information into insight.

— Carly Fiorina, former CEO, Hewlett-Packard

Statistics (Statistics, Applied Mathematics). In my assessment, this is the foundation of data science. It is the science that deals with the collection, classification, analysis, and interpretation of numerical facts or data. Supported directly by use of mathematical theories of probability, statistics imposes order and regularity on collections of dissimilar elements. This is the first time I have included a fair coverage of inferential statistics, regression models, and generalized linear models (GLMs) in a data science book. But, people generally do not know how to read, interpret, and use statistics, which must be governed by the following inquiries:

- How was the data collected?
- Does the evidence come from reliable sources?
- What is the data's background?

- Are all data reported?
- Have the data been interpreted correctly?

Scientific Collaboration and Publication (All Data-Centered Scientific Disciplines) GitHub has revolutionized programming in general and data science programming specifically. Git is a free and open-source distributed version control system (configuration management) designed to handle everything from small to very large projects with speed and efficiency. Anyone taking data science courses at an accredited college or university, or as no-credit training through Coursera, will need to be familiar with GitHub. We'll talk more about GitHub in Chapter 5.

Data Dashboards and Apps (Data Science, Business Intelligence, Other Domain-Specific Disciplines). Commercial software tools, like Tableau, has made the creation of dashboards convenient, cost-effective, useful, and should be included in the data scientists' toolbox. Google Analytics is where dashboards were born, and Google Analytics 360 is Google's current powerhouse. The COVID-19 pandemic of 2020 brought other dashboards to the fore, with the Johns Hopkins coronavirus tracker (JHU, 2021) and the UK government coronavirus tracker (GOV.UK, 2022) being good examples. Here, we'll talk about Shiny Apps with R Shiny (Chapter 11). If you want your data science products used customers, you have to build intuitive dashboards for them.

Interdisciplinary. In a nutshell, nearly any scientist may be a data scientist, even though there may be contention with this statement. In data science, we use statistics, which entails the application of mathematical and computational aspect of data science and suggests decision options to take advantage of the results of foundational descriptive and predictive analytics, where statistical methods and models prevail. We also use machine learning, which is highly specialized in terms of coding and algorithm construction. I have also performed data mining while doing the stuff of data science, so our IT friends are involved as well. It's like bringing a combined arms force to bear on a stubborn, defending enemy to drive them from their stronghold and reveal their vulnerabilities.

Torture the data, and it will confess to anything.

– Ronald Coase, winner of the Nobel Prize in Economics

About Python and Jupyter Labs

Python is a high-level general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small- and large-scale projects.

GitHub.

We'll discuss GitHub in Chapter 4, but for now, all the material in this book resides in my repositories there:

<https://github.com/stricje1>.

Related Books by the Author

Time Series Analysis and Forecasting using Python & R. Copyright© 2020, Lulu, Inc. ISBN 978-1-716-45113-3

Data Science Applications using Python and R. Copyright© 2020, Lulu, Inc. ISBN 978-1-716-89644-6

Data Science Applications using R. Copyright© 2020, Lulu, Inc. ISBN 978-0-359-81042-0

Logistic Regression Inside-Out. Copyright © 2020 by Jeffrey S. Strickland. Lulu, Inc. ISBN 978-1-365-81915-5

Predictive Crime Analysis using R. Copyright© 2018, Jeffrey Strickland. Lulu, Inc. ISBN 978-0-359-43159-5

Time Series Analysis using Open-Source Tools. Copyright© 2016, Jeffrey Strickland. Glasstree, Inc. ISBN 978-1-5342-0100-2

Predictive Analytics using R. Copyright © 2016 by Jeffrey S. Strickland. Lulu.com. ISBN 978-1-312-84101-7

Introduction to Crime Analysis and Mapping. © 2016 by Jeffrey S. Strickland. Lulu.com. ISBN 978-1-312-19311-6

Data Analytics Using Open-Source Tools. Copyright © 2015 by Jeffrey Strickland. Lulu Inc. ISBN 978-1-365-21384-7

Data Science and Analytics for Ordinary People. Copyright © 2015 by Jeffrey S. Strickland. Lulu.com. ISBN 978-1-329-28062-5

Operations Research using Open-Source Tools. Copyright © 2015 by Jeffrey Strickland. Lulu Inc. ISBN 978-1-329-00404-7

Missile Flight Simulation - Surface-to-Air Missiles, 2nd Edition. Copyright © 2015 by Jeffrey S. Strickland. Lulu.com. ISBN 978-1-329-64495-3

Predictive Modeling and Analytics. © 2014 by Jeffrey S. Strickland. Lulu.com. ISBN 978-1-312-37544-4

Verification and Validation for Modeling and Simulation. Copyright ©

2014 by Jeffrey S. Strickland. Lulu.com. ISBN 978-1-312-74061-7

Using Math to Defeat the Enemy: Combat Modeling for Simulation. © 2011 by Jeffrey S. Strickland. Lulu.com. ISBN 978-1-257-83225-5

Mathematical Modeling of Warfare and Combat Phenomenon. Copyright © 2011 by Jeffrey S. Strickland. Lulu.com. ISBN 978-1-45839255-8

Simulation Conceptual Modeling. Copyright © 2011 by Jeffrey S. Strickland. Lulu.com. ISBN 978-1-105-18162-7.

Discrete Event Simulation using ExtendSim 8. Copyright © 2010 by Jeffrey S. Strickland. Lulu.com. ISBN 978-0-557-72821-3

1. Data Wrangling

Objectives

In this chapter, we want to:

- Acquire data in various ways
- Obtain insights from data with *Pandas* library
- Handle missing values
- Correct data format
- Standardize and normalize data

Automobile Dataset

The dataset we use in this chapter is an open dataset, by Jeffrey C. Schlemmer, containing data on used car prices. This dataset is in CSV (comma separated value) 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. 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, which we'll deal with below.

Acquiring Data

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, we'll learn how to load a dataset into our Jupyter Notebook.

In our case, the Automobile Dataset is an online source, URL. First, we'll 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.

```
# Install specific version of Libraries used in chapter
# ! conda install pandas==1.3.3 -y
# ! conda install numpy=1.21.2 -y

# Import pandas Library
import pandas as pd
import numpy as np
```

Importing 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. This dataset is hosted on the UC Irvine Machine Learning Repository or we can save it to our locale disc or we can read it directly into the notebook from the repository. The file path can be either the URL list as from the data source URL above, or we can use our 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. We can get the header information from `import-85.names` (in the same directory as the data) or, we can create any column names that make sense to us after seeing the attribute information in the names file:

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

Here we show the attribute information from the `imports-85.names` file:

Attribute:	Attribute Range:
<code>symboling</code> :	-3, -2, -1, 0, 1, 2, 3.
<code>normalized-losses</code> :	continuous from 65 to 256.
<code>make</code> :	alfa-romero, audi, bmw, chevrolet, dodge, honda, isuzu, jaguar, mazda, mercedes-benz, mercury, mitsubishi, nissan, peugot, plymouth, porsche,

Attribute:	Attribute Range:
	renault, saab, subaru, toyota, volkswagen, volvo
fuel-type:	diesel, gas.
aspiration:	std, turbo.
num-of-doors:	four, two.
body-style:	hardtop, wagon, sedan, hatchback, convertible.
drive-wheels:	4wd, fwd, rwd.
engine-location:	front, rear.
wheel-base:	continuous from 86.6 to 120.9.
length:	continuous from 141.1 to 208.1.
width:	continuous from 60.3 to 72.3.
height:	continuous from 47.8 to 59.8.
curb-weight:	continuous from 1488 to 4066.
engine-type:	dohc, dohcv, l, ohc, ohcf, ohcv, rotor.
num-of-cylinders:	eight, five, four, six, three, twelve, two.
engine-size:	continuous from 61 to 326.
fuel-system:	1bbl, 2bbl, 4bbl, idi, mfi, mpfi, spdi, spfi.
bore:	continuous from 2.54 to 3.94.
stroke:	continuous from 2.07 to 4.17.
compression-ratio:	continuous from 7 to 23.
horsepower:	continuous from 48 to 288.
peak-rpm:	continuous from 4150 to 6600.
city-mpg:	continuous from 13 to 49.
highway-mpg:	continuous from 16 to 54.
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 riskier, 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 on the

dataset, refer to the complete `imports-85.names` text file for information on each of the 26 columns (the file is in the same directory as the data).

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.

```
# 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://archive.ics.uci.edu/ml/machine
-learning-databases/autos/imports-85.data"
df = pd.read_csv(other_path, header = 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(n)` to show the first `n` rows of the data frame, where `n` is an integer. Similarly, `dataframe.tail (n)` shows the bottom `n` 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.

```
# 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
0	3	?	alfaromero	gas	std	two	convertible	rwd front
1	3	?	alfaromero	gas	std	two	convertible	rwd front
2	1	?	alfaromero	gas	std	two	hatchback	rwd front

0	1	2	3	4	5	6	7	8
3	2	164	audi	gas	std	four	sedan	fwd front
4	2	164	audi	gas	std	four	sedan	4wd front

5 rows × 26 columns

Now, let's check the bottom 10 rows of data frame "df".

```
print("The last 10 rows of the dataframe\n")
df.tail(10)
```

The last 10 rows of the dataframe

0	1	2	3	4	5	6	7	8
-1	74	volvo	gas	std	four	wagon	rwd	front
-2	103	volvo	gas	std	four	sedan	rwd	front
-1	74	volvo	gas	std	four	wagon	rwd	front
-2	103	volvo	gas	turbo	four	sedan	rwd	front
-1	74	volvo	gas	turbo	four	wagon	rwd	front
-1	95	volvo	gas	std	four	sedan	rwd	front
-1	95	volvo	gas	turbo	four	sedan	rwd	front
-1	95	volvo	gas	std	four	sedan	rwd	front
-1	95	volvo	diesel	turbo	four	sedan	rwd	front
-1	95	volvo	gas	turbo	four	sedan	rwd	front

10 rows × 26 columns

Adding Headers

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. So, now we use the information from the names file from the Data Acquisition section above to add headers manually. First, we create a list of "headers" that include all column names in order. Then, we use `dataframe.columns = headers` function 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', . . . ,
'compression-ratio', 'horsepower', 'peak-rpm', 'city-mpg',
'highway-mpg', 'price']

```

So, we replaced headers in our dataframe. If we use the `dataframe.head` method introduced above 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 our dataframe we may want to export our pandas dataframe with column headings to a new CSV file, but we have a few more things to do to the dataset first. Alternatively, we could wait on loading the data until after we define headers, then we can use the function:

```
df = pd.read_csv(filename, names = headers).
```

```

df.columns = headers
df.head(10)

```

	symboling	normalized-losses	make	fuel-type	aspiration	...	price
0	3	?	alfa-romero	gas	std	...	13495
1	3	?	alfa-romero	gas	std	...	16500
2	1	?	alfa-romero	gas	std	...	16500
3	2	164	audi	gas	std	...	13950
4	2	164	audi	gas	std	...	17450
5	2	?	audi	gas	std	...	15250
6	1	158	audi	gas	std	...	17710
7	1	?	audi	gas	std	...	18920
8	1	158	audi	gas	turbo	...	23875
9	0	?	audi	gas	turbo	...	?

10 rows × 26 columns

Viewing the first 10 rows, we observe question marks in the `normalized-losses` and the `price` columns. Of particular interest is the question mark in row nine for price, since it is the dependent variable. We need to replace the `?` symbol with `NaN` (Not-a-Number) so the `dropna()` can remove the missing values. We can do this using the `dataframe.replace()` function from Pandas. We need to tell the replace function what to replace, `?`, and what to replace it with, `np.NaN`, which is a numpy function.

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

We can drop missing values along the column "price" using the `dataframe.dropna()` function and the data subset, "price", as follows.

```
df = df1.dropna(subset=["price"], axis=0)df.head(14)
```

	symbol- ling	normal- ized-losses	make	fuel- type	aspir- ation	...	price
0	3	NaN	alfa- romero	gas	std	...	13495
1	3	NaN	alfa- romero	gas	std	...	16500
2	1	NaN	alfa- romero	gas	std	...	16500
3	2	164	audi	gas	std	...	13950
4	2	164	audi	gas	std	...	17450
5	2	NaN	audi	gas	std	...	15250
6	1	158	audi	gas	std	...	17710
7	1	NaN	audi	gas	std	...	18920
8	1	158	audi	gas	turbo	...	23875
10	2	192	bmw	gas	std	...	16430
11	0	192	bmw	gas	std	...	16925
12	0	188	bmw	gas	std	...	20970
13	0	188	bmw	gas	std	...	21105
14	1	NaN	bmw	gas	std	...	24565

14 rows × 26 columns

Now, we have successfully read the raw dataset and added the correct headers into the dataframe. We also dropped missing "price" data. Notice that row 9 is no longer in the dataset above.

Now, we want to check the name of the columns of the dataframe.

```
# Here we write code that will print the column names  
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')
```

Reading & Saving Multiple Formats

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 we would save the dataframe `df` as `automobile.csv` to our local machine, we can use the syntax below, where `index = False` means the row names will not be written:

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

We can also read and save other file formats as shown in **Table 1-1**. We can use similar functions like `pd.read_csv()` and `df.to_csv()` for other dataformats. The functions are listed in the following table:

Table 1-1. 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>
...

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` (floating point), `int` (integer), `bool` (Boolean) and `datetime64` (64-bit date-time). In order to better learn about each attribute, it is always good for us to know the data type of each column. In Pandas:

<code>df.dtypes</code>	<code>print(df.dtypes)</code>
<code>symboling</code>	<code>int64</code>
<code>normalized-losses</code>	<code>object</code>
<code>make</code>	<code>object</code>
<code>fuel-type</code>	<code>object</code>
<code>aspiration</code>	<code>object</code>
<code>num-of-doors</code>	<code>object</code>
<code>body-style</code>	<code>object</code>
<code>drive-wheels</code>	<code>object</code>
<code>engine-location</code>	<code>object</code>
<code>wheel-base</code>	<code>float64</code>
<code>length</code>	<code>float64</code>
<code>width</code>	<code>float64</code>
<code>height</code>	<code>float64</code>
<code>curb-weight</code>	<code>int64</code>
<code>engine-type</code>	<code>object</code>
<code>num-of-cylinders</code>	<code>object</code>
<code>engine-size</code>	<code>int64</code>
<code>fuel-system</code>	<code>object</code>
<code>bore</code>	<code>object</code>
<code>stroke</code>	<code>object</code>
<code>compression-ratio</code>	<code>float64</code>
<code>horsepower</code>	<code>object</code>
<code>peak-rpm</code>	<code>object</code>
<code>city-mpg</code>	<code>int64</code>
<code>highway-mpg</code>	<code>int64</code>
<code>price</code>	<code>object</code>
<code>dtype:</code>	<code>object</code>

A series with the data type of each column is returned, so we want to check the data type of data frame, `df`, by `.dtypes`

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.

Describing Dataframes

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()
```

	Symbolic	wheel-base	length	width	...	city-mpg	Highway-mpg
count	201	201	201	201	...	201	201
mean	0.84	98.8	174.2	65.9	...	25.2	30.7
std	1.25	6.1	12.3	2.1	...	6.4	6.8
min	-2	86.6	141.1	60.3	...	13	16
25%	0	94.5	166.8	64.1	...	19	25
50%	1	97.0	173.2	65.5	...	24	30
75%	2	102.4	183.5	66.6	...	30	34
max	3	120.9	208.1	72	...	49	54

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")
```

	Symbolic	Normalized-losses	make	fuel-type	Highway-mpg	price
count	201	164	201	201	201	201
unique	NaN	51	22	2	NaN	186
top	NaN	161	toyota	gas	NaN	7898
freq	NaN	11	32	181	NaN	2
mean	0.84	NaN	NaN	NaN	30.69	NaN
std	1.25	NaN	NaN	NaN	6.82	NaN
min	-2	NaN	NaN	NaN	16	NaN
25%	0	NaN	NaN	NaN	25	NaN
50%	1	NaN	NaN	NaN	30	NaN
75%	2	NaN	NaN	NaN	34	NaN
max	3	NaN	NaN	NaN	54	NaN

11 rows × 26 columns

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.

Gold Nugget

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

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

Where "column" is the name of the column, we 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 column's `length` and `compression-ratio`.

```
df[['length', 'compression-ratio']].describe ()
```

	length	compression-ratio
count	201.000000	201.000000
mean	174.200995	10.164279
std	12.322175	4.004965
min	141.100000	7.000000
25%	166.800000	8.600000
50%	173.200000	9.000000
75%	183.500000	9.400000
max	208.100000	23.000000

Dataframe Information

Another method we can use to check your dataset is:

`dataframe.info()`

It provides a concise summary of our `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):
```

symboling	201 non-null int64
normalized-losses	164 non-null object
make	201 non-null object
fuel-type	201 non-null object
aspiration	201 non-null object
num-of-doors	199 non-null object
body-style	201 non-null object
drive-wheels	201 non-null object
engine-location	201 non-null object
wheel-base	201 non-null float64
length	201 non-null float64
width	201 non-null float64
height	201 non-null float64
curb-weight	201 non-null int64
engine-type	201 non-null object
num-of-cylinders	201 non-null object
engine-size	201 non-null int64

<code>symboling</code>	<code>201 non-null int64</code>
<code>fuel-system</code>	<code>201 non-null object</code>
<code>bore</code>	<code>197 non-null object</code>
<code>stroke</code>	<code>197 non-null object</code>
<code>compression-ratio</code>	<code>201 non-null float64</code>
<code>horsepower</code>	<code>199 non-null object</code>
<code>peak-rpm</code>	<code>199 non-null object</code>
<code>city-mpg</code>	<code>201 non-null int64</code>
<code>highway-mpg</code>	<code>201 non-null int64</code>
<code>price</code>	<code>201 non-null object</code>

`dtypes: float64(5), int64(5), object(16)` memory usage:
 42.4+ KB

Data Wrangling?

Data wrangling is the process of converting data from the initial format to a format that may be better for analysis. As we perform this task, we must remember that we are looking for the fuel consumption (L/100k) rate for the diesel car?

One of the things we are looking for during data wrangling is the presence of missing values. Most of the machine learning models, we want to use will provide an error if we pass NaN values into it. Consequently, we want to impute any missing values. (Eddie, 2021)

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

Identifying and Imputing Missing Values

Steps for working with missing data:

1. Identify missing data
2. Impute missing data
3. Correct data format

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.

```

import numpy as np
# Replace "?" to NaN
df.replace("?", np.nan, inplace = True)
df.head(5)

```

	symbolic ling	normalized- losses	make ...	city- mpg	price	city- L/100km
0	3	122	alfa-romero ...	21	13495	11.1904
1	3	122	alfa-romero ...	21	16500	11.1904
2	1	122	alfa-romero ...	19	16500	12.3684
3	2	164	audi ...	24	13958	9.7917
4	2	164	audi ...	18	17450	13.0556

5 rows × 30 columns

Identifying 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.

```

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

```

	symbolic- ling	normalized- losses	make ... mpg	city- price	city- L/100km
0	False	True	False ... False	False	False
1	False	True	False ... False	False	False
2	False	True	False ... False	False	False
3	False	False	False ... False	False	False
4	False	False	False ... False	False	False

5 rows × 26 columns

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

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.

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

```
symboling
False      201
Name: symboling, dtype: int64
normalized-losses
False      164
True       37
Name: normalized-losses, dtype: int64

make
False      201
Name: make, dtype: int64
fuel-type
False      201
Name: fuel-type, dtype: int64
aspiration
False      201
Name: aspiration, dtype: int64
num-of-doors
False      199
True        2
Name: num-of-doors, dtype: int64
body-style
False      201
Name: body-style, dtype: int64
drive-wheels
False      201
Name: drive-wheels, dtype: int64
engine-location
False      201
Name: engine-location, dtype: int64
wheel-base
False      201
Name: wheel-base, dtype: int64
length
False      201
```

```
symboling
False      201
Name: symboling, dtype: int64
length
Name: length, dtype: int64
width
False      201
Name: width, dtype: int64
height
False      201
Name: height, dtype: int64
curb-weight
False      201
Name: curb-weight, dtype: int64
engine-type
False      201
Name: engine-type, dtype: int64
num-of-cylinders
False      201
Name: num-of-cylinders, dtype: int64
engine-size
False      201
Name: engine-size, dtype: int64
fuel-system
False      201
Name: fuel-system, dtype: int64
bore
False      197
True       4
Name: bore, dtype: int64
stroke
False      197
True       4
Name: stroke, dtype: int64
compression-ratio
False      201
Name: compression-ratio, dtype: int64
horsepower
False      199
True       2
Name: horsepower, dtype: int64
peak-rpm
False      199
```

```
symboling
False    201
Name: symboling, dtype: int64
True     2
Name: peak-rpm, dtype: int64
city-mpg
False    201
Name: city-mpg, dtype: int64
highway-mpg
False    201
Name: highway-mpg, dtype: int64
price
False    201
Name: price, 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

Impute Missing Data

Imputation is a technique used for replacing the missing data with some substitute value to retain most of the data/information of the dataset. These techniques are used because removing the data from the dataset every time is not feasible and can lead to a reduction in the size of the dataset to a large extend, which not only raises concerns for biasing the dataset but also leads to incorrect analysis.

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

```
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

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

Calculate the mean value for the "bore" column

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

Average of bore: 3.3307106598984775

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

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

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

```
# Calculate the mean value 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)
```

Average of stroke: 3.2569035532994857

Calculate the mean value for the "horsepower" column

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

Average horsepower: 103.39698492462311

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

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

Calculate the mean value for "peak-rpm" column

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

Average peak rpm: 5117.587939698493

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

```
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:

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

```
four    113  
two     86  
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:

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

```
four
```

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

```
# 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)  
df.head()
```

	Symbolic- losses	make	city- ...mpg	High-way- mpg	price
0	122	alfa-romero	... 21	27	13495
1	122	alfa-romero	... 21	27	16500
2	122	alfa-romero	... 19	26	16500
3	164	audi	... 24	30	13950
4	164	audi	... 18	22	17450

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

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.)

Data standardization is the critical process of bringing data into a common format that allows for collaborative research, large-scale analytics, and sharing of sophisticated tools and methodologies. Why is it so important?

For example, healthcare data can vary greatly from one organization to the next. Data are collected for different purposes, such as provider reimbursement, clinical research, and direct patient care. These data may be stored in different formats using different database systems and information models. And despite the growing use of standard terminologies in healthcare, the same concept (e.g., blood glucose) may be represented in a variety of ways from one setting to the next.

Getting there involves converting that data into a uniform format, with logical and consistent definitions. This includes standardizing the way we label data will improve access to the most relevant and current information

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 acceptsthe fuel consumption with L/100km standard. We will need to apply **data transformation** to transform mpg intoL/100km

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

We can do many mathematical operations directly in *Pandas*.

```
df.head()
```

	Symbolic- ling	Normalized- losses	make	...	city- mpg	High- way-mpg	price
0	3	122	alfa-romero	...	21	27	13495
1	3	122	alfa-romero	...	21	27	16500
2	1	122	alfa-romero	...	19	26	16500
3	2	164	audi	...	24	30	13950
4	2	164	audi	...	18	22	17450

5 rows × 26 columns

```
# Convert mpg to L/100km by mathematical operation (235 divided by mpg)
df['city-L/100km'] = 235/df["city-mpg"]
# Check your transformed data
df.head()
```

	Symbolic losses	Normalized- make	...	city- mpg	Highway- mpg	price	
0	3	122	alfa-romero	...	21	27	13495
1	3	122	alfa-romero	...	21	27	16500
2	1	122	alfa-romero	...	19	26	16500
3	2	164	audi	...	24	30	13950
4	2	164	audi	...	18	22	17450

Gold Nugget

According to the example above, transform mpg to L/100km in the column of "highway-mpg" and change the name of column to

$$\frac{\text{highway-}L}{100\text{km}}$$

```
# 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()
```

	Symbolic losses	Normalized- make	...	city- mpg	Highway- mpg	price	
0	3	122	alfa-romero	...	21	27	13495
1	3	122	alfa-romero	...	21	27	16500
2	1	122	alfa-romero	...	19	26	16500
3	2	164	audi	...	24	30	13950
4	2	164	audi	...	18	22	17450

Now that we have a standardized dataset, we want to normalize it as

well.

Data 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. 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 $(\text{original value}) / (\text{maximum value})$.

```
# Replace (original value) by (original value)/(maximum value)
df['length'] = df['length']/df['length'].max()
df['width'] = df['width']/df['width'].max()
```

Using the example above, we will now normalize the column `height`.

```
# Write your code below and press Shift+Enter to execute
df['height'] = df['height']/df['height'].max()
# Show the scaled columns
df[["length", "width", "height"]].head()
```

	length	width	height
0	0.811148	0.890278	0.816054
1	0.811148	0.890278	0.816054
2	0.822681	0.909722	0.876254
3	0.848630	0.919444	0.908027
4	0.848630	0.922222	0.908027

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

Python provides the `preprocessing` library, which contains the `normalize` function to normalize the data. It takes an array in as an input and normalizes its values between 000 and 111. It then returns

an output array with the same dimensions as the input. Since `normalize()` only normalizes values along rows, we need to convert the column of a dataframe into an array before we apply the method.

```
from sklearn import preprocessing
import numpy as np
housing =
pd.read_csv("https://raw.githubusercontent.com/stricje1/jupyter/main/data/california_housing_train.csv")
x_array = np.array(housing['total_bedrooms'])
normalized_arr = preprocessing.normalize([x_array])
print(normalized_arr)

[[0.01437454  0.02129852  0.00194947 ...  0.00594924
 0.00618453  0.00336115]]
```

Normalizing along the rows, can be very unintuitive. Normalizing along rows means that each individual sample is normalized instead of the features. However, you can specify the axis while calling the method to normalize along a feature (column). The value of axis parameter is set to 1 by default. If we change the value to 0, the process of normalization happens along a column.

```
names = housing.columns
d = preprocessing.normalize(housing, axis=0)
scaled_df = pd.DataFrame(d, columns=names)
scaled_df.head()
```

	longitude	latitude	total_rooms	population	households	median_income
0	-0.007332	0.007347	0.012562	0.004246	0.005730	0.002647
1	-0.007342	0.007393	0.017123	0.004723	0.005621	0.003226
2	-0.007348	0.007240	0.001612	0.001393	0.001420	0.002926
3	-0.007348	0.007229	0.003360	0.002154	0.002744	0.005657
4	-0.007348	0.007214	0.003255	0.002610	0.003181	0.003412

Data Binning

Data binning, or bucketing, is a data pre-processing method used to minimize the effects of small observation errors. The original data values are divided into small intervals known as bins and then they are replaced by a general value calculated for that bin. This has a

smoothing effect on the input data and may also reduce the chances of overfitting in the case of small datasets

There are 2 methods of dividing data into bins:

1. Equal Frequency Binning: bins have an equal frequency.
2. Equal Width Binning: bins have equal width with a range of each bin are defined as $[min + w], [min + 2w], \dots [min + nw]$, where

$$w = \frac{max - min}{nbr\ of\ bins}.$$

In our dataset, `horsepower` is a real valued variable ranging from 48 to 288 and it has 59 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

First, we convert data to correct format:

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

Let's plot the histogram of `horsepower` to see what the distribution of `horsepower` looks like. We will use the *Matplotlib* package to construct the plot, as seen in **Figure 1-1**. We can learn more about Matplotlib at: <https://matplotlib.org/>

```
%matplotlib inline  
mpl.rcParams['figure.dpi'] = 300  
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")
```

```
Text(0.5, 1.0, 'horsepower bins')
```

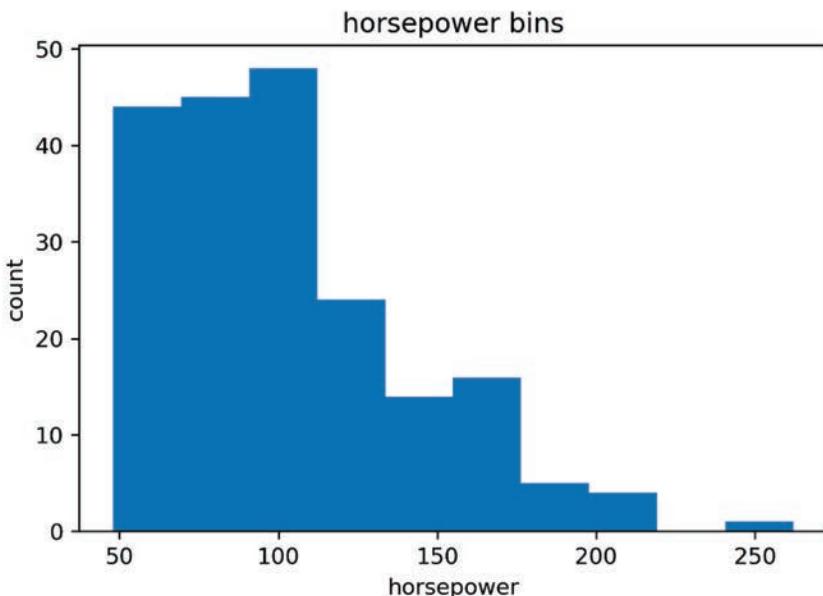


Figure 1-1. Histogram of horsepower from the automobile price data

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.

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

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

Now, let's set group names:

```
group_names = ['Low', 'Medium', 'High']
```

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

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

```
df[['horsepower', 'horsepower-binned']].head(20)
```

horsepower	horsepower-binned
0	111
1	111
2	154
3	102
4	115
5	110
6	110
7	110
8	140
9	101
10	101
11	121
12	121
13	121
14	182
15	182
16	182

Let's see the number of vehicles in each bin and displays the results as low, medium, high, using the `value_counts` function:

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

Low	153
Medium	43
High	5

Name: horsepower-binned, dtype: int64

Now, using the bin distribution from above, let's construct a plot of the distribution of each bin and display it in **Figure 1-2**.

```

import matplotlib as plt
%matplotlib inline
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")

```

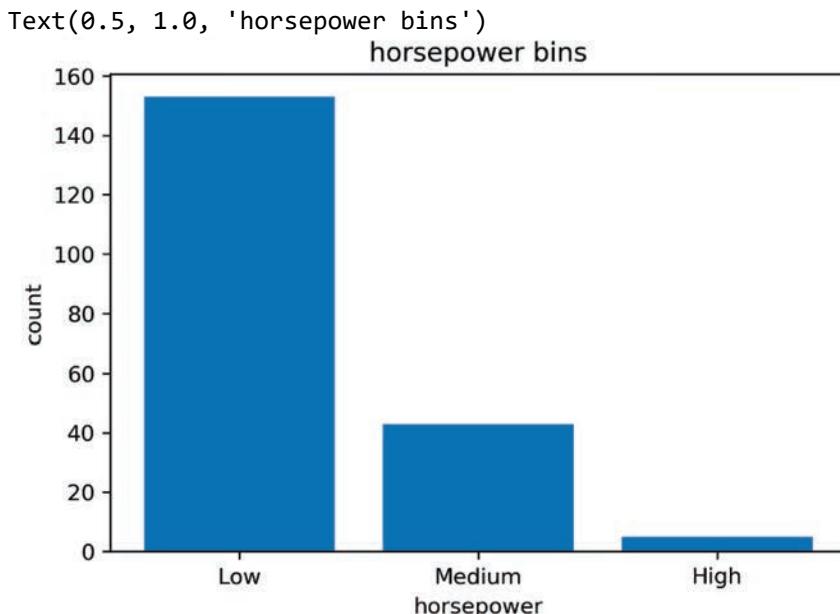


Figure 1-2. Horsepower histogram by low medium and high bins.

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 59 to 3!

Bins Visualization

Normally, a histogram is used to visualize the distribution of bins like we created above. Now, we plot a histogram that clearly shows the low-medium-high binning result by actual horsepower numeric values for the attribute `horsepower`. This generates a much simpler histogram than the original in **Figure 1-1**. So, let's construct the

histogram and display it in **Figure 1-3**.

```
%matplotlib inline
pyplot.rcParams['figure.dpi'] = 300
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")
plt.pyplot.savefig('foo2.png', bbox_inches='tight',
dpi=300)
```

Text(0.5, 1.0, 'horsepower bins')

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.

When you put an indicator variable in a regression model, there are two things you must always keep in mind about interpreting the coefficients associated with the indicator variable:

1. The coefficient on an indicator variable is an estimate of the average **difference** in the dependent variable for the group identified by the indicator variable (after taking into account other variables in the regression) and
2. The **reference group**, which is the set of observations for which the indicator variable is always zero.

If you always remember that the coefficient on an indicator variable is an estimate of a **difference** with respect to a **reference group** (sometimes referred to as the **omitted category**), you're 90% of the way to understanding indicator variables.

Why we use indicator variables?

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

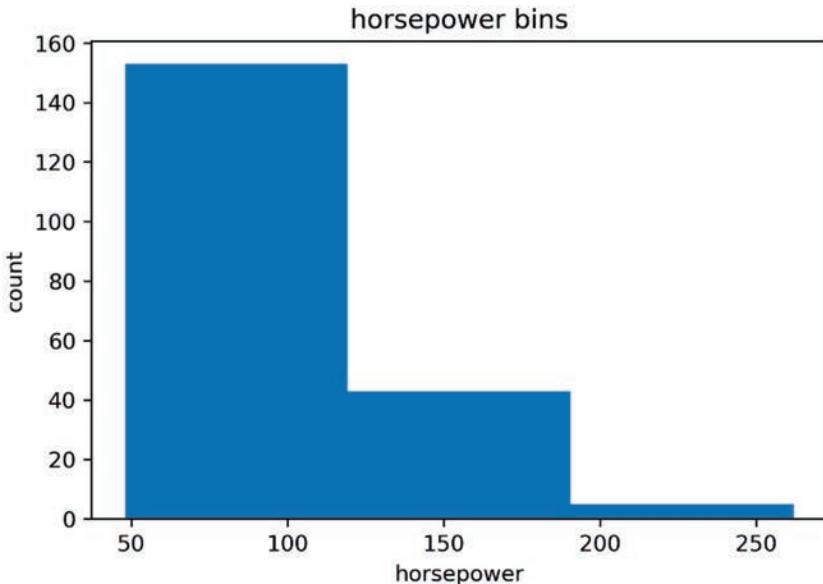


Figure 1-3. Histogram of horsepower in three continuous range bins

Example: Fuel Type as an Indicator Variable

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`.

Here, we set the indicator variables and assign it to the data frame `dummy_variable_1`:

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

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

Next, we change the column names for clarity:

```
dummy_variable_1.rename(columns={'gas': 'fuel-type-gas',
'diesel': 'fuel-type-diesel'}, inplace=True)
dummy_variable_1.head()
```

	fuel-type-diesel	fuel-type-gas
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1

5 rows × 29 columns

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

```
# 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)
df.head()
```

The last two columns are now the indicator variable representation of the fuel-type variable. They're all 0s and 1s now. Similar to before, create an indicator variable for the column **aspiration**.

```
# 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()
```

	aspiration-std	aspiration-turbo
0	1	0
1	1	0
2	1	0
3	1	0
4	1	0

Next, let's merge the new dataframe to the original dataframe, then drop the column **aspiration**.

Finally, we save the new csv to use for the next part of our work, exploratory data analysis (EDA).

```
df.to_csv('clean_df.csv')
```

2. Exploratory Data Analysis

Objectives

In this chapter, we'll introduce some ways to explore our dataset efficiently with Pandas and Matplotlib so that we can start modeling our data:

- First, we'll import data, which is the first step needed to complete before we can start our analysis.
- Then, we'll get some basic descriptive statistics of our data, and check the first and last rows of our DataFrame.
- After gathering some information on our data, we'll take a deeper look at it by querying or indexing the data, which can help us test some of the basic hypotheses that you might have about the data.
- After this brief inspection of the data, we'll examine some features that may influence our feature selection later on.
- Next, we'll look at the challenges that our data can pose, such as missing values or outliers, and, of course, how you can handle those challenges.
- Finally, we'll discover patterns in our data, by either visualizing it with the Python data visualization packages Matplotlib and by using specific functions to compute the correlation between attributes.

We use a dataset comprised of Used Car prediction variables and the response variable, price. This will help us answer the question: What characteristics have the most impact on used car prices?

First, we'll import libraries we know we need.

```
import pandas as pd  
import numpy as np
```

Import Data

Now, we'll load the data and store it in dataframe `df`. This dataset is hosted in my Jupyter | Data repository on GitHub

```
path='https://raw.githubusercontent.com/stricje1/Data/master/automobileEDA.csv'  
df = pd.read_csv(path)  
df.head()
```

	Symbolic	Normalized-losses	make...	price	city-L/100km
0	3	122	alfa-romero...	13495.0	11.19
1	3	122	alfa-romero...	16500.0	11.19
2	1	122	alfa-romero...	16500.0	12.37
3	2	164	audi...	13950.0	9.79
4	2	164	audi...	17450.0	13.06

5 rows × 29 columns

Analyzing Feature Using Visualization

To install Seaborn we use `pip`, the Python package manager.

```
%capture
! pip install seaborn
```

Import visualization packages `Matplotlib` and `Seaborn`. Don't forget about `%matplotlib inline` to plot in a Jupyter notebook.

```
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.

```
# List the data types for each column
print(df.dtypes)
```

<code>symboling</code>	<code>int64</code>
<code>normalized-losses</code>	<code>int64</code>
<code>make</code>	<code>object</code>
<code>aspiration</code>	<code>object</code>
<code>num-of-doors</code>	<code>object</code>
<code>body-style</code>	<code>object</code>
<code>drive-wheels</code>	<code>object</code>
<code>engine-location</code>	<code>object</code>
<code>wheel-base</code>	<code>float64</code>
<code>length</code>	<code>float64</code>
<code>width</code>	<code>float64</code>
<code>height</code>	<code>float64</code>

symboling	int64
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

Suppose we want to find the data type of the column "peak-rpm". We merely look for "peak-rpm" in the left column and read to the right getting **float64**. We can calculate the correlation between variables of type "int64" or "float64" using the method **corr()**:

```
df.corr ()
```

	Normal- Symboling	Normalized- losses	... city- L/ 100km	diesel	gas
Symboling	1.000	0.466 ...	0.066	-0.20	0.20
Normalized-losses	0.466	1.000 ...	0.239	-0.10	0.10
wheelbase	-0.536	-0.057 ...	0.476	0.31	-0.31
length	-0.365	0.019 ...	0.657	0.21	-0.21
width	-0.242	0.087 ...	0.673	0.24	-0.24
height	-0.550	-0.374 ...	0.004	0.28	-0.28
curb-weight	-0.233	0.099 ...	0.785	0.22	-0.22
engine-size	-0.111	0.112 ...	0.745	0.07	-0.07
bore	-0.140	-0.030 ...	0.555	0.05	-0.05
stroke	-0.008	0.056 ...	0.037	0.24	-0.24
Compression-ratio	-0.182	-0.115 ...	-0.299	0.99	-0.99
Horsepower	0.076	0.217 ...	0.889	-0.17	0.18
peak-rpm	0.280	0.240 ...	0.116	-0.48	0.48
city-mpg	-0.036	-0.225 ...	-0.950	0.27	-0.27

	Symbol	Normalized losses	...	L/100km	diesel	gas
highway-mpg	0.036	-0.182	...	-0.930	0.20	-0.20
price	-0.082	0.134	...	0.790	0.11	-0.11
city-L/100km	0.066	0.239	...	1.000	-0.24	0.24
diesel	-0.197	-0.102	...	-0.241	1.00	-1.00
gas	0.197	0.102	...	0.241	-1.00	1.00

The diagonal elements are always one; we will study correlation more precisely **Pearson correlation** in-depth at the end of the notebook.

Now we want to find the correlation between the following columns: **bore**, **stroke**, **compression-ratio**, and **horsepower**. So, we can select those columns using the following.

```
df[['bore', 'stroke', 'compression-ratio',
   'horsepower']].corr()
```

	bore	stroke	compression-ratio	Horse-power
bore	1.0000	-0.055	0.001263	0.56694
stroke	-0.0554	1.000	0.187923	0.09846
compression-ratio	0.0013	0.1879	1.000000	-0.21451
horsepower	0.5669	0.0985	-0.214514	1.00000

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

Let's construct the scatterplot of **engine-size** and **price** in **Figure 2-1**.

```
# Engine size as potential predictor variable of price
sns.set(rc={"figure.dpi":300, 'savefig.dpi':300})
sns.regplot(x="engine-size", y="price", data=df)
plt.ylim(0,)
plt.savefig('EDA01.png', bbox_inches='tight', dpi=300)
```

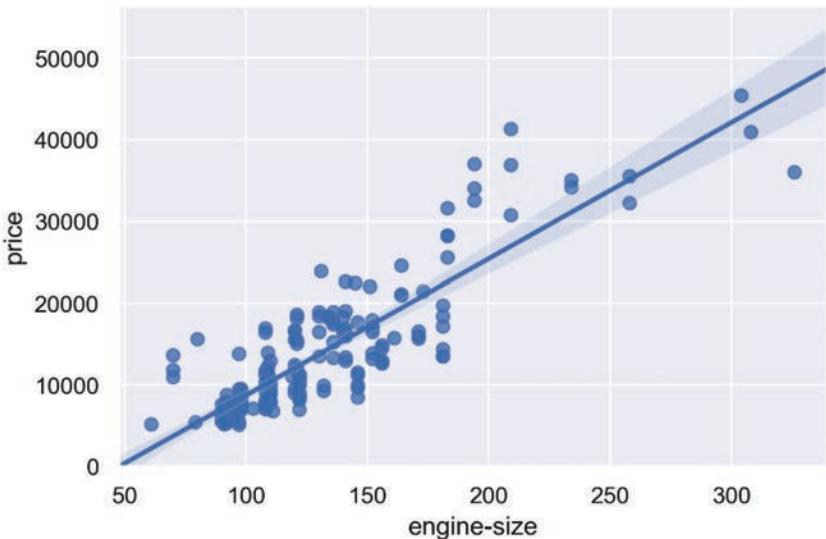


Figure 2-1. The scatterplot with fitted line shows a positive linear relationship between engine size and price of an automobile.

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.

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

	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 **Figure 2-2**.

```

sns.set(rc={"figure.dpi":300, 'savefig.dpi':300})
sns.regplot(x="highway-mpg", y="price", data=df)
plt.savefig('EDA02.png', bbox_inches='tight', dpi=300)

```

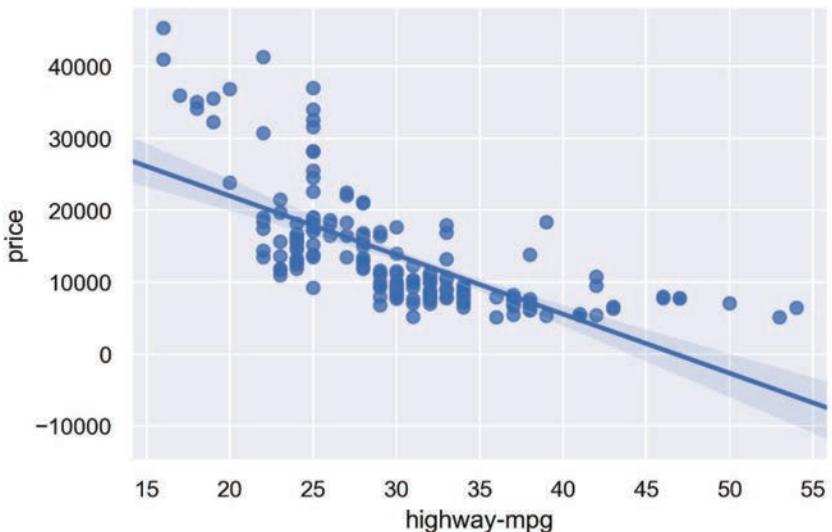


Figure 2-2. Here the scatterplot and fitted line show a negative linear relationship between highway-mpg and price of automobiles.

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.

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

	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 **Figure 2-3**.

```

sns.set(rc={"figure.dpi":300, 'savefig.dpi':300})
sns.regplot (x="peak-rpm", y="price", data=df)
plt.savefig('EDA03.png', bbox_inches='tight', dpi=300)

```

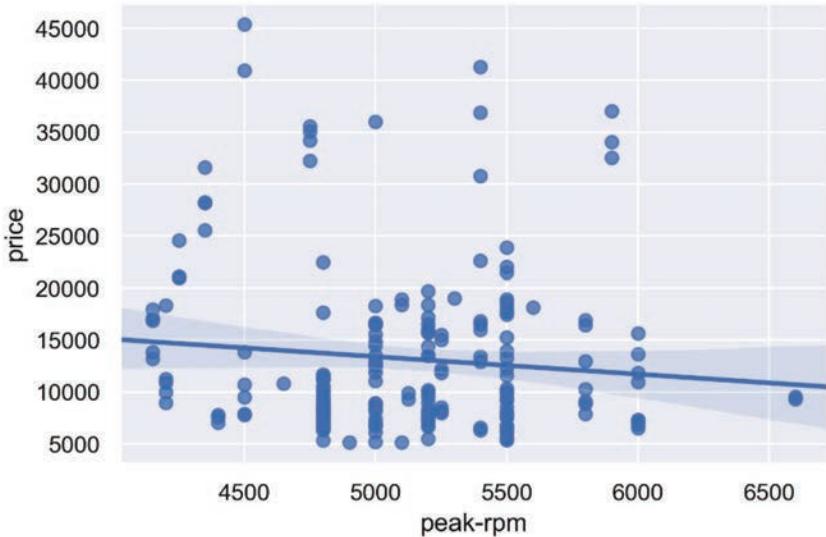


Figure 2-3. This scatterplot shows a very weak linear relationship between peak-rpm and price of automobiles.

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.

```
df[ [ 'peak-rpm' , 'price' ] ].corr ()
```

	peak-rpm	price
peak-rpm	1.000000	-0.101616
price	-0.101616	1.000000

Now, we want to find the correlation between `x=stroke` and `y=price`. So, we want to select those columns, using the following syntax: `df[["stroke", "price"]]`.

Now, we'll write our code below and to execute it to explore the correlation between stroke and sales price, displayed in **Figure 2-4**.

```
df[["stroke", "price"]].corr()
```

	stroke	price
stroke	1.00000	0.08231
price	0.08231	1.00000

Given the correlation results between `price` and `stroke`, can we expect a linear relationship? We can answer this using the function `regplot()`.

There is a weak correlation between the variable `stroke` and `price`, as such regression will not work well. We demonstrate this using the `regplot` function, revealed in **Figure 2-4**.

```
sns.set(rc={"figure.dpi":300, 'savefig.dpi':300})  
sns.regplot(x="stroke", y="price", data=df)  
plt.savefig('EDA04.png', bbox_inches='tight', dpi=300)
```

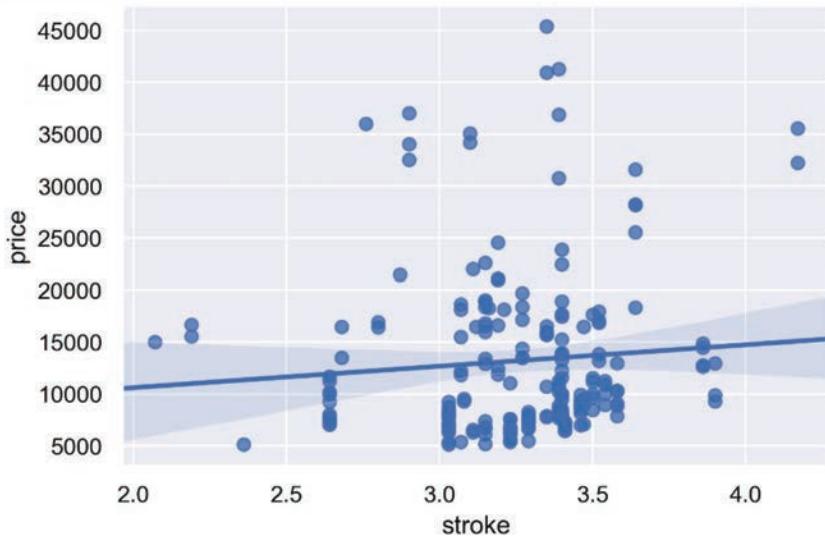


Figure 2-4. This scatterplot reveals a weak linear relationship between `stroke` and `price` of automobiles.

Categorical Variables

Categorical variables 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**.

```
sns.set(rc={"figure.dpi":300, 'savefig.dpi':300})  
sns.boxplot(x="body-style", y="price", data=df)  
plt.savefig('EDA05.png', bbox_inches='tight', dpi=300)
```

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.

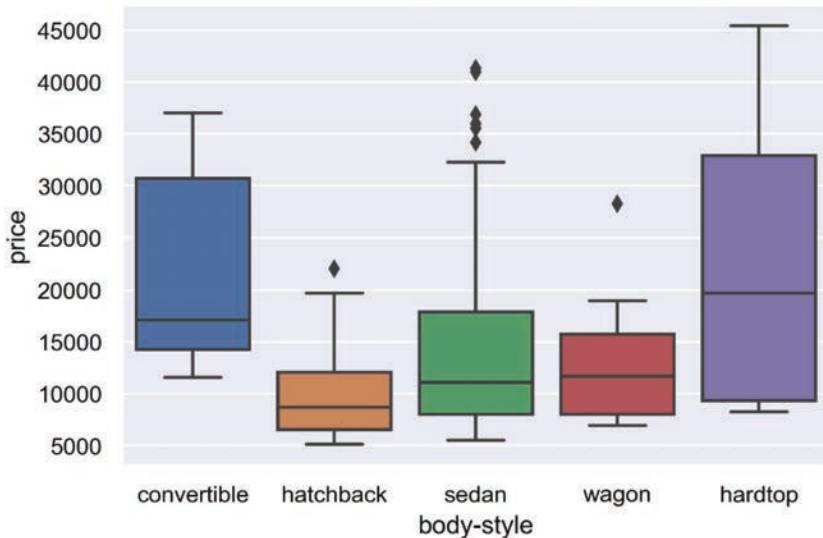


Figure 2-5. This boxplot shows the relationship of the categorical variable "body-style" in relationship to each other.

Let's examine automobiles' **engine-location** and **price**, having the values **front** and **rear**:

```
sns.set(rc={"figure.dpi":300, 'savefig.dpi':300})  
sns.boxplot(x="engine-location", y="price", data=df)  
plt.savefig('EDA06.png', bbox_inches='tight', dpi=300)
```

Figure 2-6 shows 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.

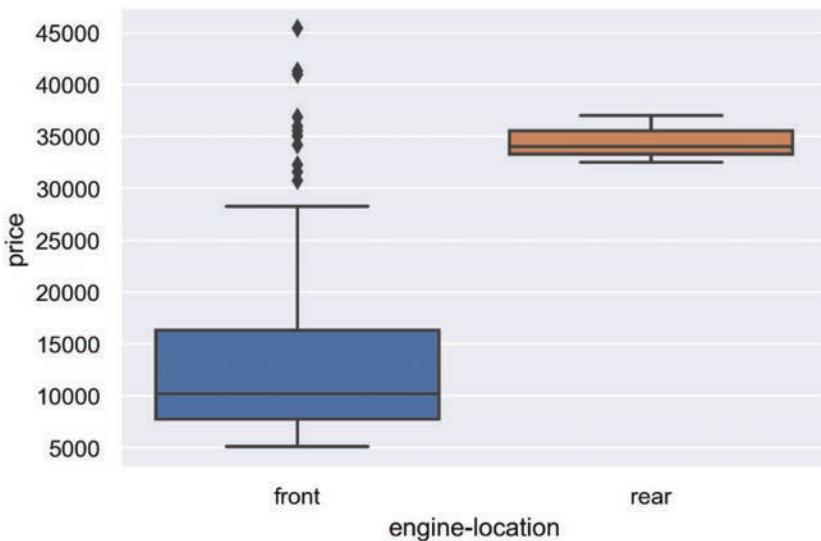


Figure 2-6. This boxplot reveals the relationship between the two levels of engine location on the price of automobiles.

Let's examine `drive-wheels` and `price`.

```
sns.set(rc={"figure.dpi":300, 'savefig.dpi':300})
sns.boxplot(x="drive-wheels", y="price", data=df)
plt.savefig('EDA07.png', bbox_inches='tight', dpi=300)
```

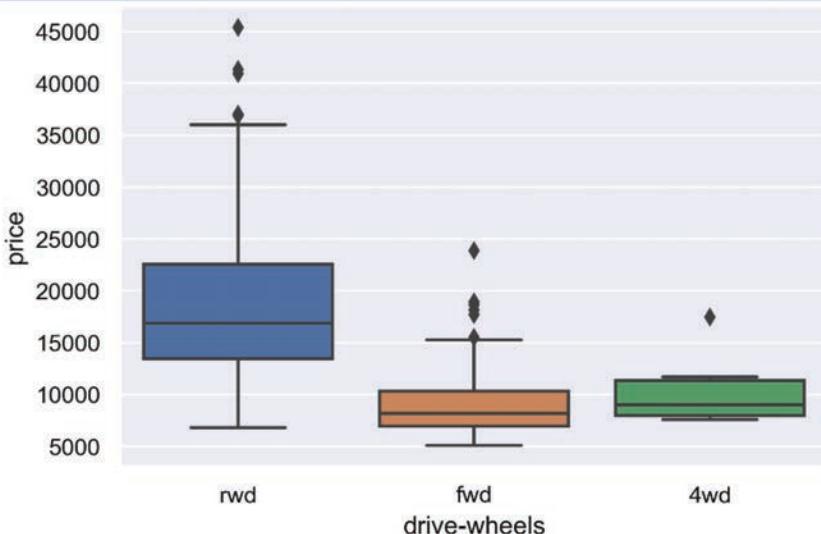


Figure 2-7. This boxplot displays the relationship between three levels of drive-wheels and the prices of automobiles.

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.

Descriptive Statistical Analysis

Let's first take a look at the variables by utilizing a description method.

The `dataframe.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:

```
df.describe ()
```

	normalized-	city-			
	symboling	losses	price	L/100km	diesel
count	201.000	201.00	201.00	201.000	201.00
mean	0.841	122.00	13207.13	9.944	0.0995
std	1.255	31.99	7947.07	2.535	0.3001
min	-2.000	65.00	5118.00	4.796	0.0000
25%	0.000	101.00	7775.00	7.833	0.0000
50%	1.000	122.00	10295.00	9.792	0.0000
75%	2.000	137.00	16500.00	12.368	0.0000
max	3.000	256.00	45400.00	18.077	1.0000

The default setting of `describe` skips variables of type object. We can apply the method `describe` on the variables of type 'object' as follows:

```
df.describe(include=[ 'object' ])
```

	make	aspira-tion	num-of-cylinders	fuel-system	horse-power-binned
count	201	201 . . .	201	201	200
unique	22	2 . . .	7	8	3
top	toyota	std . . .	four	mpfi	Low
freq	32	165 . . .	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']]`.

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

```
fwd    118
rwd     75
4wd      8
Name: drive-wheels, dtype: int64
```

We can convert the series to a dataframe as follows:

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

drive-wheels
fwd
rwd
4wd

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`.

```
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	
fwd	118
rwd	75
4wd	8

Now let's rename the index to `drive-wheels`:

```
drive_wheels_counts.index.name = 'drive-wheels'
drive_wheels_counts
```

value_counts	
drive-wheels	
fwd	118
rwd	75
4wd	8

We can repeat the above process for the variable `engine-location`.

```
# 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.head(10)
```

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.

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.

```
df['drive-wheels'].unique ()  
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`.

```
df_group_one=df[['drive-wheels', 'body-style', 'price']]
```

We can then calculate the average price for each of the different categories of data.

```
# Grouping results  
df_group_one = df_group_one.groupby(['drive-wheels'],as_index=False).mean()  
df_group_one
```

	drive-wheels	price
0	4wd	10241.00
1	fwd	9244.78
2	rwd	19757.61

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`.

```
# 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
```

	drive-wheels	body-style	price
0	4wd	hatchback	7603.00
1	4wd	sedan	12647.33
2	4wd	wagon	9095.75
3	fwd	convertible	11595.00
4	fwd	hardtop	8249.00
5	fwd	hatchback	8396.39
6	fwd	sedan	9811.80
7	fwd	wagon	9997.33
8	rwd	convertible	23949.60
9	rwd	hardtop	24202.71
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:

```
grouped_pivot = grouped_test1.pivot(index =
    'drive-wheels', columns='body-style')
grouped_pivot
```

	price			
body-style	convertible	hardtop	hatchback	sedan
drive-wheels				
4wd	NaN	NaN	7603.00	12647.33
fwd	11595.00	8249.00	8396.39	9811.80
rwd	23949.60	24202.71	14337.78	21711.83

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 book on its own.

```
# Fill missing values with 0
grouped_pivot = grouped_pivot.fillna(0)
grouped_pivot
```

price					
body-style	convertible	hardtop	hatchback	sedan	
drive-wheels					
4wd	0.00	0.00	7603.00	12647.33	
fwd	11595.00	8249.00	8396.39	9811.80	
rwd	23949.60	24202.71	14337.78	21711.83	

Now, we use the `groupby` function to find the average `price` of each car based on `body-style`.

```
# Grouping results
df_gptest2 = df[['body-style', 'price']]
grouped_test_bodystyle = df_gptest2.groupby (
    ['body-style'], as_index= False).mean ()
grouped_test_bodystyle
```

	body-style	price
0	convertible	21890.50
1	hardtop	22208.50
2	hatchback	9957.44
3	sedan	14459.76
4	wagon	12371.96

If you did not import `pyplot`, let's do it again.

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

Variables: Drive Wheels and Body Style vs. Price

Let's use the **heat map** in **Figure 2-8** to visualize the relationship between `body-style` vs `price`. We'll talk more about heat maps (and their application with confusion matrices) in later chapters.

```
# Use the grouped results
sns.set(rc={"figure.dpi":300, 'savefig.dpi':300})
plt.pcolor(grouped_pivot, cmap='RdBu')
plt.colorbar()
plt.show()
plt.savefig('EDA08.png', bbox_inches='tight', dpi=300)
```

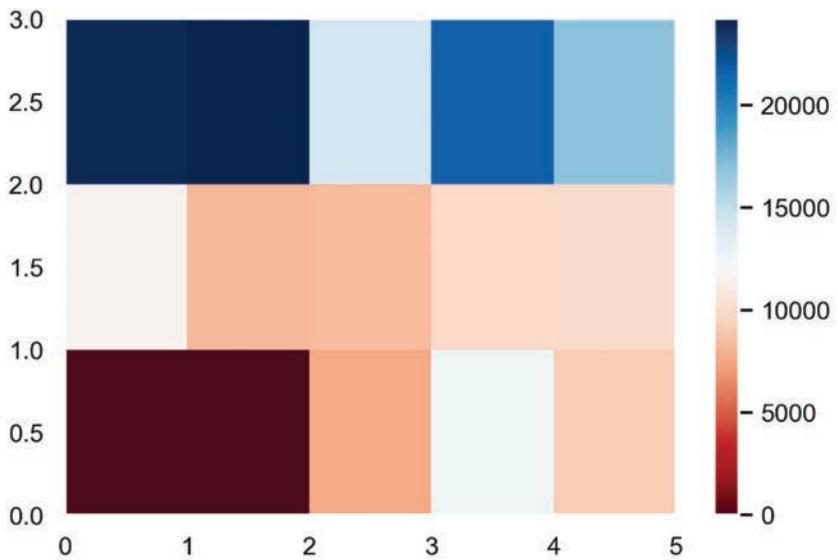


Figure 2-8. This heatmap shows the relationship between drive-wheel, body-style and price

The heatmap plots the target variable (`price`) proportional to color 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 and show the plot in **Figure 2-9**.

```

sns.set(rc={"figure.dpi":300, 'savefig.dpi':300})
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()
plt.savefig('EDA09.png', bbox_inches='tight', dpi=300)

```

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.

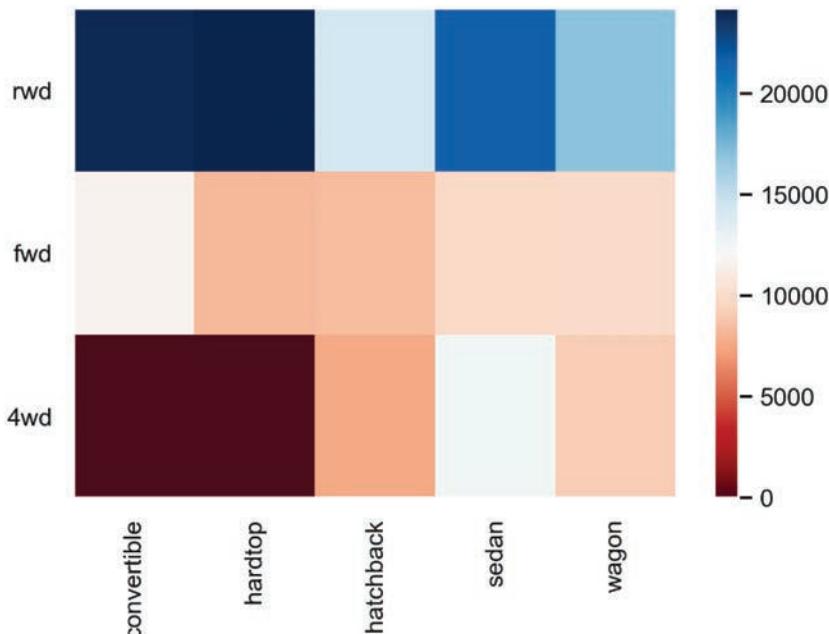


Figure 2-9. Same heat map as in , except the variable levels/categories are shown on the axes.

<Figure size 1800x1200 with 0 Axes>¹

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?

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 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 'int64' or 'float64' variables.

```
df.corr()
```

¹ The output shows the figure dimension, which corresponds to dpi = 300.

	Symbo- ling	Normal-ized- losses	.	.	diesel	gas
symboling	1	0.4663	.	.	-0.196	0.196
normalized-losses	0.466		1	.	-0.101	0.101
wheel-base	-0.536	-0.0567	.	.	0.307	-0.307
length	-0.365	0.0194	.	.	0.211	-0.211
width	-0.242	0.0868	.	.	0.244	-0.244
height	-0.550	-0.3737	.	.	0.281	-0.281
curb-weight	-0.233	0.0994	.	.	0.221	-0.221
engine-size	-0.110	0.1124	.	.	0.070	-0.070
bore	-0.140	-0.0299	.	.	0.054	-0.054
stroke	-0.008	0.0556	.	.	0.241	-0.241
compression-ratio	-0.182	-0.1147	.	.	0.985	-0.985
horsepower	0.075	0.2173	.	.	-0.169	0.169
peak-rpm	0.279	0.2395	.	.	-0.475	0.475
city-mpg	-0.035	-0.2250	.	.	0.265	-0.265
highway-mpg	0.036	-0.1819	.	.	0.198	-0.198
price	-0.082	0.1340	.	.	0.110	-0.110
city-L/100km	0.066	0.2386	.	.	-0.241	0.241
diesel	-0.196	-0.1015	.	.	1	-1
gas	0.196	0.1015	.	.	-1	1

Sometimes we would like to know the significant of the correlation estimate.

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 < 0.001 : we say there is strong evidence that the correlation is significant.
- the p-value is < 0.05 : there is moderate evidence that the correlation is significant.
- the p-value is < 0.1 : there is weak evidence that the correlation is significant.
- 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.

```
from scipy import stats
```

Wheel-Base vs. Price

Let's calculate the Pearson Correlation Coefficient and P-value of `wheel-base` and `price`.

```
pearson_coef, p_value = stats.pearsonr (df['wheel-base'],
                                         df['price'])
print("The Pearson Correlation Coefficient is",
      pearson_coef, " with a P-value of P =", p_value)
```

The Pearson Correlation Coefficient is 0.5846418222655081
with a P-value of P = 8.076488270732989e-20

Conclusion:

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

Horsepower vs. Price

Let's calculate the Pearson Correlation Coefficient and P-value of `horsepower` and `price`.

```
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 0.809574567003656
with a P-value of P = 6.369057428259557e-48

Conclusion:

Since the p-value is < 0.001, the correlation between horsepower and price is statistically significant, and the linear relationship is quite strong (~0.809, close to 1).

Length vs. Price

Let's calculate the Pearson Correlation Coefficient and P-value of `length` and `price`.

```
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 0.690628380448364
with a P-value of P = 8.016477466158986e-30

Conclusion:

Since the p-value is < 0.001, the correlation between `length` and `price` is statistically significant, and the linear relationship is moderately strong (~0.691).

Width vs. Price

Let's calculate the Pearson Correlation Coefficient and P-value of `width` and `price`:

```
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 0.7512653440522674
with a P-value of P = 9.200335510481516e-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`:

```
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 0.8344145257702846
with a P-value of P = 2.1895772388936914e-53

Conclusion:

Since the p-value is < 0.001 , the correlation between `curb-weight` and `price` is statistically significant, and the linear relationship is quite strong (~ 0.834).

Engine-Size vs. Price

Let's calculate the Pearson Correlation Coefficient and P-value of `engine-size` and `price`:

```
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 < 0.001 , the correlation between `engine-size` and `price` is statistically significant, and the linear relationship is very strong (~ 0.872).

Bore vs. Price

Let's calculate the Pearson Correlation Coefficient and P-value of `bore` and `price`:

```
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 0.5431553832626602
with a P-value of P = 8.049189483935489e-17

Conclusion:

Since the p-value is < 0.001 , the correlation between `bore` and `price` is statistically significant, but the linear relationship is only moderate (~ 0.521).

We can relate the process for each `city-mpg` and `highway-mpg`:

City-mpg vs. Price

```
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 -0.6865710067844677
with a P-value of P = 2.321132065567674e-29

Conclusion:

Since the p-value is < 0.001, the correlation between **city-mpg** and **price** is statistically significant, and the coefficient of about -0.687 shows that the relationship is negative and moderately strong.

Highway-mpg vs. Price

```
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 -0.7046922650589529
with a P-value of P = 1.7495471144477352e-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.

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 beforehand.

To see if different types of `drive-wheels` impact `price`, we group the data.

```
grouped_test2 = df_gptest[['drive-wheels',
                           'price']].groupby(['drive-wheels'])
grouped_test2.head(2)
```

	drive-wheels	price
0	rwd	13495.0
1	rwd	16500.0
3	fwd	13950.0
4	4wd	17450.0
5	fwd	15250.0
136	4wd	7603.0

```
df_gptest
```

	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`.

```
grouped_test2.get_group('4wd')['price']
```

```
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**.

```
# 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

```
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

```
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

```
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.

3. Model Development

Objectives

After completing this chapter, you will be able to:

- Develop some linear models of our data
- Develop prediction models based on our linear models

In this chapter, 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 chapter are:

- 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

Let's import these libraries (we'll always need these at a minimum):

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

Load the data and store it in dataframe df:

This dataset is hosted in my GitHub Jupyter | Data repository:
<https://github.com/stricje1/Data>

```
# Path of data  
path='https://raw.githubusercontent.com/stricje1/Data/master/automobileEDA.csv'  
df = pd.read_csv(path)  
df.head()
```

	make	doors	body-style	...	price	city-L/100km	Horse-power-binned
0	alfa-romero	two	convert	...	13495.0	11.1905	Medium
1	alfa-romero	two	convert	...	16500.0	11.1905	Medium
2	alfa-romero	two	hatchback	...	16500.0	12.3684	Medium
3	audi	four	sedan	...	13950.0	9.7917	Medium
4	audi	four	sedan	...	17450.0	13.0556	Medium

5 rows × 29 columns

Linear and Multiple Linear Regression

Linear Regression

One example of a Data Model that we will be using is a simple linear regression model.

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 (Y) (the one we want to predict)

The result of Linear Regression is a **linear function** that predicts the response (dependent) variable as a function of the predictor (independent) variable.

Y : Response Variable

X : Predictor Variables

Linear Function

$$\hat{Y} = a + bX$$

- 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:

```
from sklearn.linear_model import LinearRegression
```

Now, let's create the linear regression object using the linear model function, `lm`.

```
lm = LinearRegression()
lm
```

```
LinearRegression(copy_X=True, fit_intercept=True,
n_jobs=None, normalize=False)
```

Example: Predicting Car Prices

How could the variable, `highway-mpg`, help us predict car prices?

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.

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

Using the `fit` function, we fit the linear model using `highway-mpg`:

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

```
LinearRegression(copy_X=True, fit_intercept=True,
n_jobs=None, normalize=False)
```

Using the `predict` function, we can output a forecast.

```
Yhat = lm.predict(X)
Yhat[0:5]
```

```
array([16236.50464347, 16236.50464347, 17058.23802179,
13771.3045085, 20345.17153508])
```

What is the value of the intercept (a)?

```
lm.intercept_
38423.3058581574
```

What is the value of the slope (b)?

```
lm.coef_
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:

$$\hat{Y} = a + bX$$

Plugging in the actual values we get:

$$\text{Price} = 38423.31 - 821.73 \text{ highway-mpg}$$

Now we'll create a linear regression object called `lm1`.

```
lm1 = LinearRegression()
lm1

LinearRegression(copy_X=True, fit_intercept=True,
n_jobs=None, normalize=False)
```

Let's train the model using `engine-size` as the independent variable and `price` as the dependent variable.

```
lm1.fit(df[['engine-size']], df[['price']])
lm1
```

```
LinearRegression(copy_X=True, fit_intercept=True,
n_jobs=None, normalize=False)
```

Next, let's find the slope and intercept of the model.

Slope

```
# Slope
lm1.coef_
array([[166.86001569]])
```

And now we'll find the intercept.

Intercept

```
# Intercept
lm1.intercept_
```

```
array([-7963.33890628])
```

What is the equation of the predicted line? We can use x and \hat{y} or engine-size and price.

```
# Using X and Y  
X = df[['engine-size']]  
Yhat = -7963.34 + 166.86*X  
# Using the variable (column) names  
Price = -7963.34 + 166.86*df[['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

X₁: Predictor Variable 1

X₂: Predictor Variable 2

X₃: Predictor Variable 3

X₄: Predictor Variable 4

a: intercept

b₁: coefficients of Variable 1

b₂: coefficients of Variable 2

b₃: coefficients of Variable 3

b₄: coefficients of Variable 4

The equation is given by:

$$\hat{Y} = a + b_1X_1 + b_2X_2 + b_3X_3 + b_4X_4$$

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.

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

Fit the linear model using the four above-mentioned variables.

```
lm.fit(Z, df['price'])  
LinearRegression(copy_X=True, fit_intercept=True,  
n_jobs=None, normalize=False)
```

What is the value of the intercept(a)?

```
lm.intercept_  
-15806.624626329198
```

What are the values of the coefficients (b_1, b_2, b_3, b_4)?

```
lm.coef_  
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:

$$\hat{Y} = a + b_1X_1 + b_2X_2 + b_3X_3 + b_4X_4$$

What is the linear function we get in this example?

```
Price = -15678.7426280615 + 52.65851272 horsepower  
+ 4.69878948 curb-weight + 81.95906216 engine-size  
+ 33.58258185 highway-mpg
```

Now, we 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.

```
lm2 = LinearRegression()
lm2.fit(df[['normalized-losses' , 'highway-mpg']],
        df['price'])

LinearRegression(copy_X=True, fit_intercept=True,
                 n_jobs=None, normalize=False)
```

Find the coefficient of the model.

```
lm2.coef_
array([ 1.49789586, -820.45434016])
```

$$\hat{Y} = a + b_1X_1 + b_2X_2 + b_3X_3 + b_4X_4$$

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.

First, let's import the visualization package, *seaborn*:

```
# 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**. We display the results in **Figure 3-1**.

```
plt.rcParams['figure.dpi'] = 200
plt.figure(figsize=(width, height))
sns.regplot(x = "highway-mpg", y = "price", data = df)
```

```
plt.ylim(0,)  
plt.savefig('mod1.png', bbox_inches='tight', dpi=300)
```

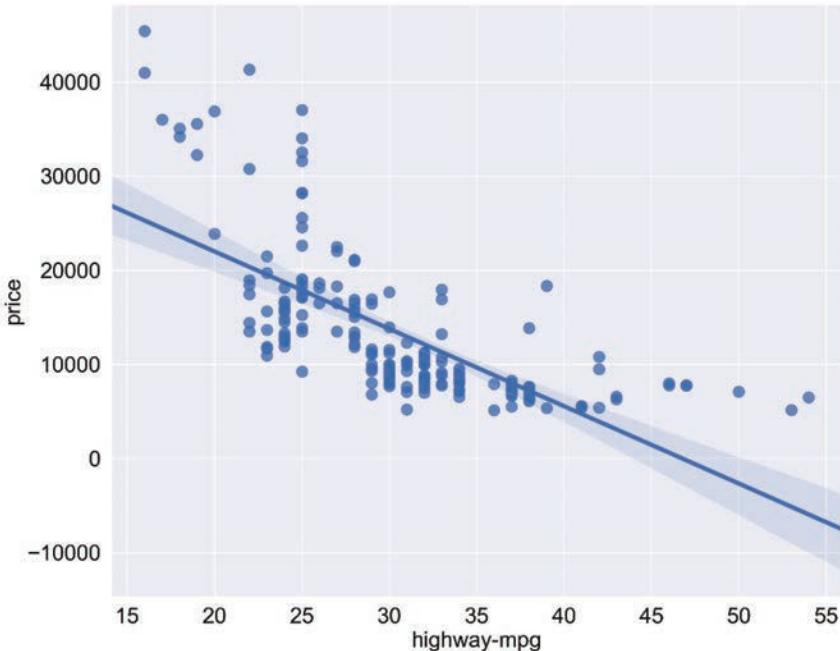


Figure 3-1. The scatter plot of highway-mpg and price with a fitted linear model and confidence interval.

We can see from this plot that `price` is negatively correlated to `highway-mpg` since the regression slope is negative. Keep in mind when looking at a regression plot, 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`.

```
plt.figure(figsize=(width, height))  
sns.regplot(x="peak-rpm", y="price", data=df)  
plt.ylim(0,)  
plt.savefig('mod2.png', bbox_inches='tight', dpi=300)
```

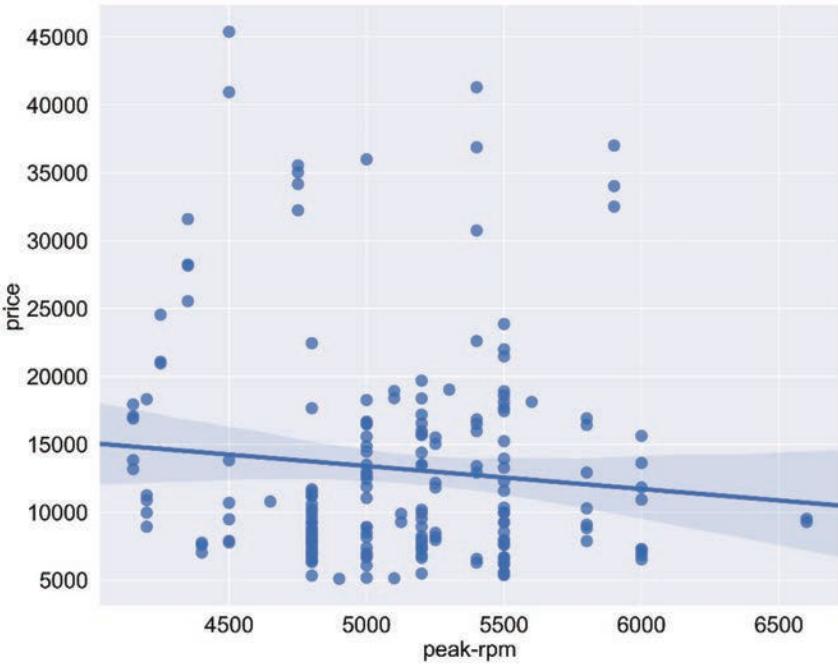


Figure 3-2. The scatter plot of peak-rpm and price with a fitted linear mode and confidence interval.

Comparing the regression plot of `peak-rpm` and `highway-mpg`, displayed in **Figure 3-2**, 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 `peak-rpm` increases.

Given the regression plots above, is `peak-rpm` or `highway-mpg` more strongly correlated with `price`? Use the method `.corr()` to verify your answer.

```
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

The variable `highway-mpg` has a stronger correlation with `price`, it

is approximate -0.704692 compared to `peak-rpm` which is approximate -0.101616. We can verify the correlation 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. When we decide on how well the model fits the data, we perform so-called **model diagnostics**. Residual analysis is one of the primary means of performing model diagnostics, and we start with a `residplot` from the `Seaborn` package and we plot the results in **Figure 3-3**.

```
sns.set(rc={"figure.dpi":300, 'savefig.dpi':300})
plt.figure(figsize=(width, height))
res_plot = sns.residplot(df['highway-mpg'], df['price'])
fig = res_plot.get_figure()
plt.show()
#sns.savefig('mod3.png', bbox_inches='tight', dpi=300)
fig.savefig('mod3.png', dpi=300)
```

We did not mention this earlier, but notice that we are saving our

plots as high-resolution PNG images. The parameter `dpi` stands for dots-per-inch.

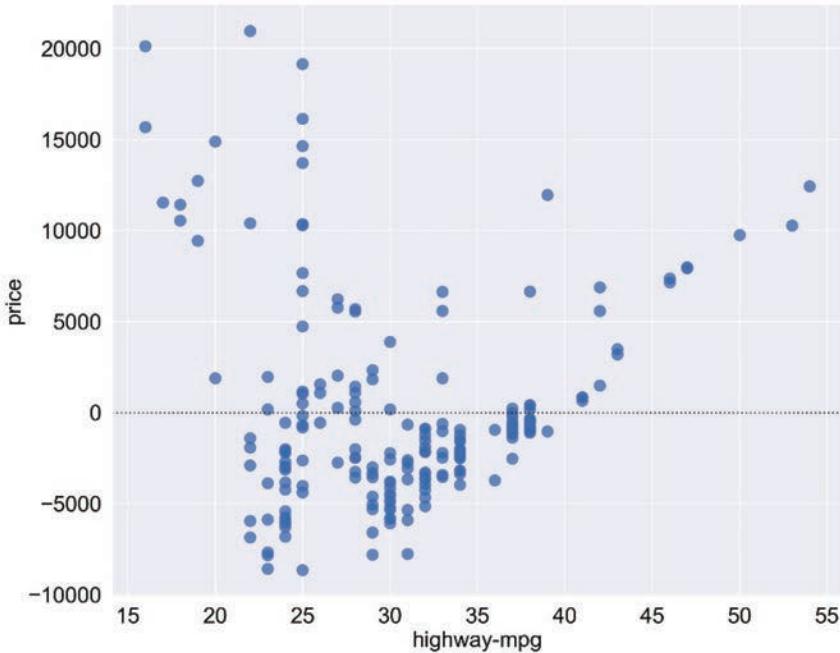


Figure 3-3. The residual plot for the price fitted by the linear model using highway mpg as the single predictor variable.

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 using the `predict` function.

```
Y_hat = lm.predict (Z)
```

Now let's use seaborn's `distplot` function to construct the plot and displaying it in **Figure 3-4**. Here we do more than basic plotting. We also add some aesthetics to enhance the information provided by the plot. Later, we'll discuss "underdoing" and "overdoing" plots. Either can confuse or distract the audience the we intend to inform.

```
sns.set(rc={"figure.dpi":300, 'savefig.dpi':300})  
  
# Set the default text font size  
plt.rc('font', size = 24)  
# Set the axes title font size  
plt.rc('axes', titlesize = 20)  
# Set the axes Labels font size  
plt.rc('axes', labelsize = 20)  
# Set the font size for x tick Labels  
plt.rc('xtick', labelsize = 20)  
# Set the font size for y tick Labels  
plt.rc('ytick', labelsize = 20)  
# Set the legend font size  
plt.rc('legend', fontsize = 22)  
  
ax1 = sns.distplot(df['price'], hist=False, color="r",  
                   label="Actual Value", kde_kws=dict(linewidth=5))  
dist_plot = sns.distplot(Y_hat, hist=False, color="b",  
                        label="Fitted Values", kde_kws=dict(linewidth=5),  
                        ax=ax1)  
fig = dist_plot.get_figure()  
  
plt.title('Actual vs Fitted Values for Price')  
plt.xlabel('Price (in dollars)')  
plt.ylabel('Proportion of Cars')  
  
plt.show()  
fig.savefig('mod4.png', dpi=300)
```

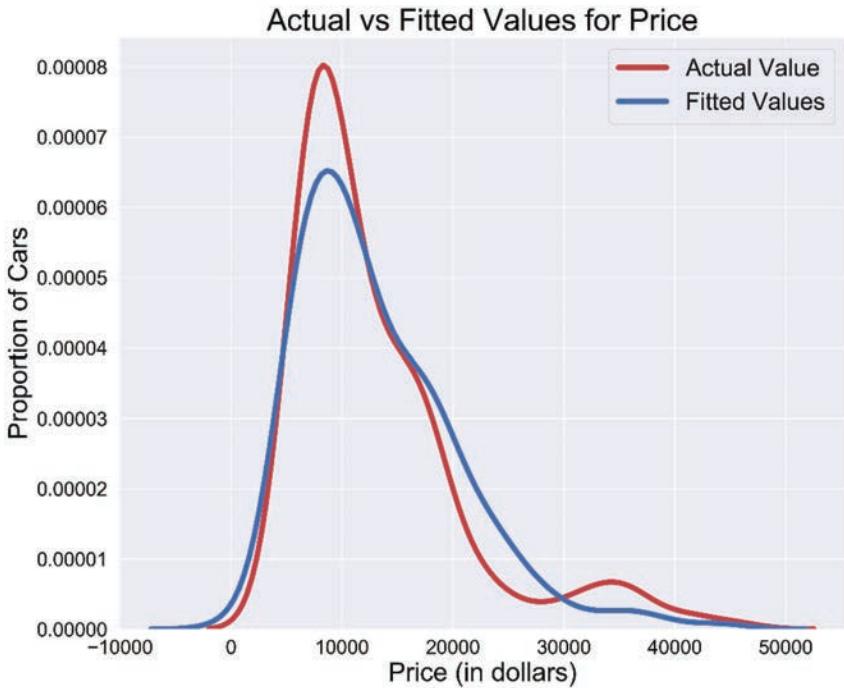


Figure 3-4. The distribution plot for the fitted and actual values of car price.

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.

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

$$\hat{Y} = a + b_1X + b_2X^2$$

Cubic - 3rd Order

$$\hat{Y} = a + b_1X + b_2X^2 + b_3X^3$$

Higher-Order:

$$Y = a + b_1X + b_2X^2 + b_3X^3 + \cdots + b_nX^n$$

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 user-defined function to plot the data.

```
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, '-.', linewidth=2, markersize=12)
    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')

    # Set the default text font sizes
    plt.rc('font', size = 24)
    # Set the title label font size
    plt.rc('axes', titlesize = 20)
    # Set the axes Labels font size
    plt.rc('axes', labelsize = 20)
    # Set the font size for x tick Labels
    plt.rc('xtick', labelsize = 20)
    # Set the font size for y tick Labels
    plt.rc('ytick', labelsize = 20)
    # Set the Legend font size

    plt.show()
    plt.close()
    fig.savefig('mod5.png', bbox_inches='tight', dpi=300)
```

Let's get the variables we need:

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

Let's fit the polynomial using the function `polyfit` in **Figure 3-5**, then use the function `poly1d` to display the polynomial function.

```
# 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 using our `PlotPolly` definition.

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

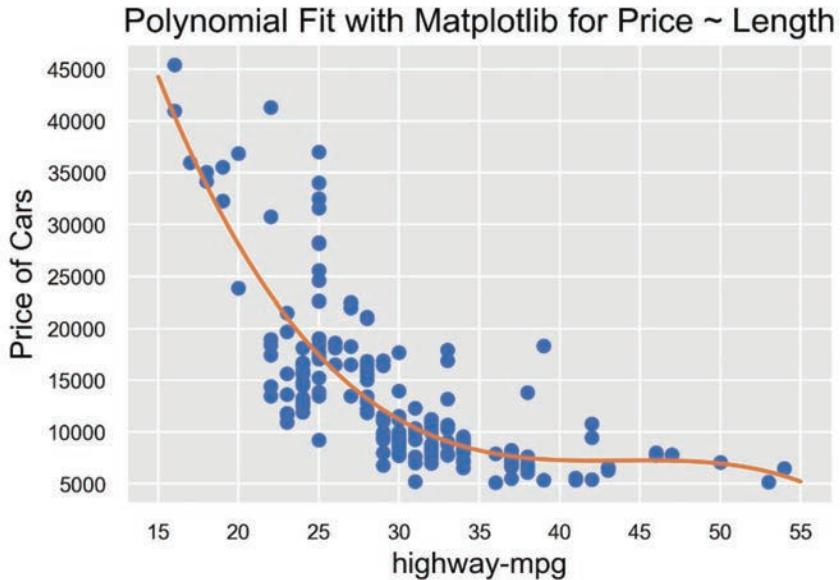


Figure 3-5. Polynomial regression model plot of price based on car length.

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

Now, we create 11th-order polynomial model with the variables x and y from above, a show it in **Figure 3-6**.

```
# 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')
```

$$\begin{aligned} & 11 & 10 & 9 & 8 & 7 \\ -1.243e-08x + 4.722e-06x^2 - 0.0008028x^3 + 0.08056x^4 - 5.297x^5 \\ & 6 & 5 & 4 & 3 & 2 \\ + 239.5x^6 - 7588x^7 + 1.684e+05x^8 - 2.565e+06x^9 + 2.551e+07x^{10} - \\ 1.491e+08 & x^{11} + 3.879e+08 \end{aligned}$$

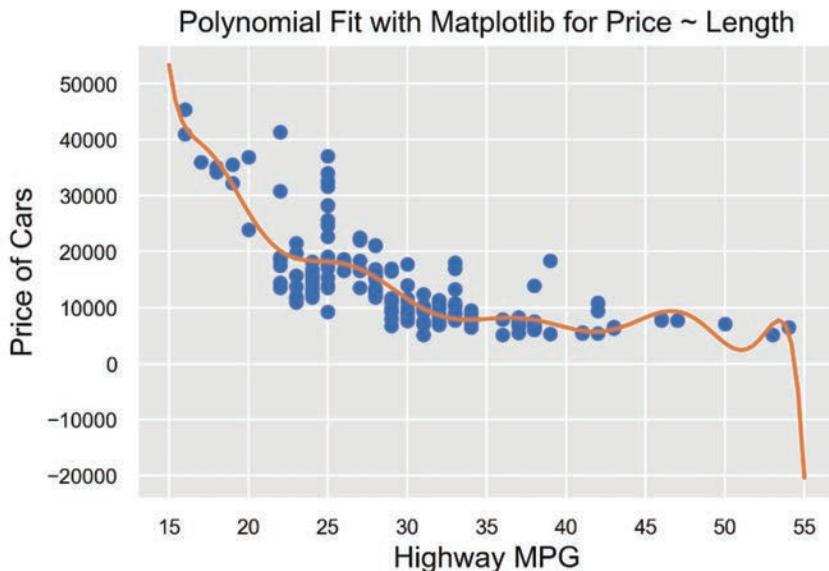


Figure 3-6. The 11th-order polynomial regression model plot of price based on highway-mpg

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:

$$\hat{Y} = a + b_1X_1 + b_2X_2 + b_3X_1X_2 + b_4X_1^2 + b_5X_2^2$$

We can perform a polynomial transform on multiple features. First,

we import the module:

```
from sklearn.preprocessing import PolynomialFeatures
```

We create a `PolynomialFeatures` object of degree 2:

```
Pr = PolynomialFeatures(degree=2)
pr
PolynomialFeatures(degree = 2, include_bias = True,
interaction_only=False,
order='C')
Z_pr = pr.fit_transform (Z)
```

In the original data, there are 201 samples and 4 features.

```
Z.shape
```

```
(201, 4)
```

After the transformation, there are 201 samples and 15 features.

```
Z_pr.shape
```

```
(201, 15)
```

Pipeline

Data Pipelines simplify the steps of processing the data. We use the module `Pipeline` and the `pipeline` function to create a pipeline. We also use `StandardScaler` as a step in our pipeline. `StandardScaler` standardizes features by removing the mean and scaling to unit variance.

```
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.

```
Input = [('scale', StandardScaler()), ('polynomial',
PolynomialFeatures(include_bias = False)),
('model', LinearRegression())]
```

We input the list as an argument to the pipeline constructor:

```
pipe = pipeline(Input)
pipe
```

```
Pipeline(memory=None,
      steps= [('scale',
                StandardScaler(copy=True, with_mean=True,
with_std=True)),
          ('polynomial', PolynomialFeatures (degree=2,
include_bias=False,
interaction_only=False, order='C')),
          ('model', LinearRegression(copy_X=True,
fit_intercept=True, n_jobs=None,
normalize=False))],
      verbose=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.

```
Z = Z.astype (float)
pipe.fit(Z, y)

Pipeline(memory=None,
      steps=[('scale',
                StandardScaler(copy=True, with_mean=True,
with_std=True)),
          ('polynomial', PolynomialFeatures (degree=2,
include_bias=False,
interaction_only=False, order='C')),
          ('model', LinearRegression(copy_X=True,
fit_intercept=True, n_jobs=None,
normalize=False))],
      verbose=False)
```

Similarly, we can normalize the data, perform a transform and produce a prediction simultaneously.

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

array([13102.74784201, 13102.74784201, 18225.54572197,
10390.29636555])
```

Create a pipeline that standardizes the data, then produce a prediction using a linear regression model using the features Z and target y.

```

Input=[('scale', StandardScaler()), ('model',
    LinearRegression())]
pipe=Pipeline(Input)
pipe.fit(Z, y)
ypipe=pipe.predict (Z)
ypipe[0:10]

array([13699.11161184, 13699.11161184, 19051.65470233,
10620.36193015, 15521.31420211, 13869.66673213,
15456.16196732, 15974.00907672, 17612.35917161,
10722.32509097])

```

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^2 or 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. It represents another diagnostic measure of model fit.

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 :

```
# 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`.

Now, we'll calculate the MSE. First, we can predict the output i.e., `yhat` using the `predict` method, where `X` is the input variable:

```
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]

Now, let's import the function `mean_squared_error` from the module `metrics`.

```
from sklearn.metrics import mean_squared_error
```

Next, we can compare the predicted results with the actual results.

```
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 :

```
# 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:

```
Y_predict_multifit = lm.predict (Z)
```

We compare the predicted results with the actual results:

```
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.

```
from sklearn.metrics import r2_score
```

We apply the function to get the value of R^2 :

```
r_squared = r2_score(y, p(x))  
print('The R-square value is: ', r_squared)
```

The R-square value is: 0.6741946663906522

We can say that ~67.419 % of the variation of price is explained by this polynomial fit.

MSE

We can also calculate the MSE:

```
mean_squared_error(df['price'], p(x))  
20474146.426361203
```

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. Let's import `pyplot` for plotting; we will also be using some functions from

numpy.

```
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline
```

Create a new input:

```
new_input = np.arange(1, 100, 1).reshape(-1, 1)
```

Fit the model:

```
lm.fit(X, Y)
lm
LinearRegression(copy_X = True, fit_intercept = True,
n_jobs = None, normalize = False)
```

Produce a prediction:

```
Yhat = lm.predict(new_input)
yhat[0:5]
array([37601.57247984, 36779.83910151, 35958.10572319,
35136.37234487, 34314.63896655])
```

We now plot the predicted data in **Figure 3-7**.

```
plt.figure(figsize=(10, 6))
plt.plot(new_input, yhat)
plt.show()
plt.savefig('mod7.png', dpi=300)
```

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.

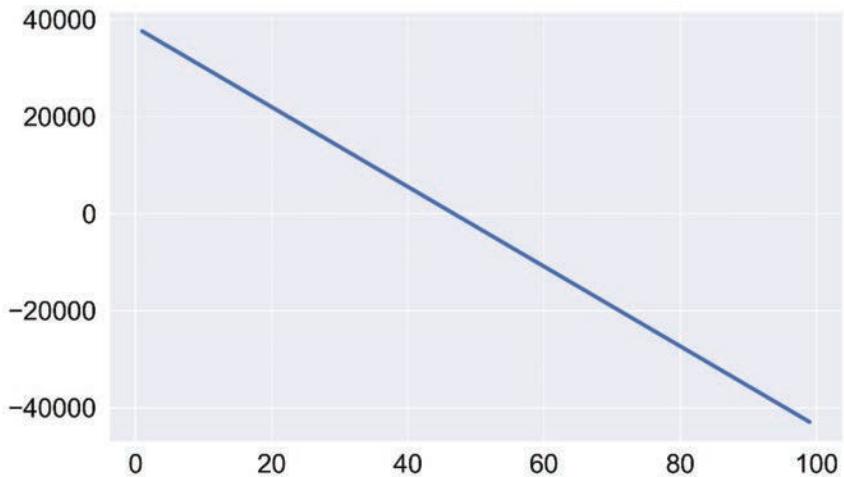


Figure 3-7. Predicted data for the car price model.

What is a good MSE value?

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: 3.16×10^7

Multiple Linear Regression: Using Horsepower, Curb-weight, Engine-size, and Highway-mpg as Predictor Variables of Price.

- R-squared: 0.80896354913783497
- MSE: 1.2×10^7

Polynomial Fit: Using Highway-mpg as a Predictor Variable of Price.

- R-squared: 0.6741946663906514
- MSE: 2.05×10^7

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^2 .

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 3.16×10^7 while MLR has an MSE of 1.2×10^7 . 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 (~ 0.497) is very small compared to the R-squared for the MLR (~ 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.

4. Model Evaluation and Refinement

Objectives

In this chapter, we will discover how to:

- Evaluate models for their predictive value
- Refine our prediction models for improved forecasts
- Examine models for over-fitting and under-fitting
- Perform model selection
- Perform Ridge Regression
- Perform Grid Searches

This dataset is hosted on my GitHub Jupyter | Data repository:
<https://github.com/stricje1/Data>

```
import pandas as pd
import numpy as np
# Import clean data
path =
'https://raw.githubusercontent.com/stricje1/jupyter/main/d
ata/chapter_4_auto.csv'
df = pd.read_csv(path)
df.to_csv('chapter_4_auto.csv')
```

First, let's only use numeric data:

```
df = df._get_numeric_data()
df.head()
```

	Un-named: 0	named: 0.1	...	city-mpg	High-way-mpg	price	city-L/100km
0	0	0	...	21	27	13495.0	11.1905
1	1	1	...	21	27	16500.0	11.1905
2	2	2	...	19	26	16500.0	12.3684
3	3	3	...	24	30	13950.0	9.7917
4	4	4	...	18	22	17450.0	13.0556

5 rows × 21 columns

In this chapter, we'll use two user-defined function for some of our plots, so we'll define them upfront.

```
%capture
! pip install ipywidgets
```

Ipywidgets are interactive HTML widgets for Jupyter notebooks. Notebooks come alive when interactive widgets are used. We can gain control of our data and can visualize changes in the data. Although interaction cannot occur in the book, you can download all the Jupyter notebooks represent in this book from my Jupyter GitHub repository. So far in this book we have used a few user-defined functions. Notice that these read just like ordinary Python code.

```
from ipywidgets import interact, interactive, fixed,  
interact_manual
```

Functions for Plotting

```
def DistributionPlot(RedFunction, BlueFunction, RedName,  
Blue Name, Title, Fig_Name):  
    # Preliminary figure setup for resolution, size, etc.  
    width = 8  
    height = 6  
    sns.set(rc={"figure.dpi":300, 'savefig.dpi':300})  
    plt.figure(figsize=(width, height))  
    # plot functions for multiple plots ax1 & dist_plot  
    ax1 = sns.distplot(RedFunction, hist=False, color="r",  
        label=RedName, kde_kws=dict(linewidth=5))  
    dist_plot = sns.distplot(BlueFunction, hist = False,  
        color="b", label = BlueName, kde_kws =  
        dict(linewidth=5), ax=ax1)  
    # Store plot object in fig for saving to Local drive  
    fig = dist_plot.get_figure()  
    # Set the default text font size for plots & Legend  
    plt.rc('font', size=24)  
    plt.rc('legend', fontsize=20)  
    # Set the plot title and x- and y-axes Labels  
    plt.title>Title)  
    plt.xlabel('Price (in dollars)')  
    plt.ylabel('Proportion of Cars')  
    # Show the plot in non-interactive mode  
    plt.show()  
    plt.close()  
    # Save figures to current directory  
    fig.savefig('Fig_Name', bbox_inches='tight',  
        dpi=300)
```

We defined a similar version of PollyPlot in the previous chapter, but we need to use this one here (there are significant differences).

```
def PollyPlot(xtrain, xtest, y_train, y_test, lr,
              poly_transform, fig_name):
    # Preliminary figure setup for plot size
    width = 12
    height = 10
    fig = plt.figure(figsize=(width, height))
    # PollyPlot terms:
    # Training data: x_train and y_train
    # Testing data: x_test and y_test
    # lr: linear regression object
    # Poly_transform: polynomial transformation object
    # Return evenly spaced values within a given interval
    xmax=max([xtrain.values.max(), xtest.values.max()])
    xmin=min([xtrain.values.min(), xtest.values.min()])
    x=np.arange(xmin, xmax, 0.1)
    # Three plotting functions:
    plt.plot(xtrain, y_train, 'ro', label='Training Data')
    ax2 = plt.plot(xtest, y_test, 'go', label='Test Data')
    poly=plt.plot(x, lr.predict(
        poly_transform.fit_transform(x.reshape(-1, 1))),
        label='Predicted Function', linewidth=4)
    # Plot figure aesthetics
    plt.ylim([-10000, 60000])
    plt.ylabel('Price')
    plt.legend()
    # Set the default text font size
    plt.rc('font', size=20)
    plt.rc('legend', fontsize=18)
    # Show the plot in non-interactive mode
    plt.show()
    plt.close()
    # Saves figure named 'fig_name' as PNG format
    fig.savefig('fig_name', bbox_inches='tight',
                dpi=300)
```

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

Here we separate the `price` variable in a dataframe, `y_data`.

```
y_data = df['price']
```

We drop `price` from the other variables in a dataframe, `x_data`:

```
X_data=df.drop ('price', axis=1)
```

Now, we randomly split our data `x` and `y` data into training and testing data. Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called **overfitting**. To avoid it, it is common practice when performing a supervised experiment to hold out part of the available data as a test set `X_test, y_test`. Note that the word “experiment” is not intended to denote Design of Experiments (DOE), because even in commercial settings training a model usually starts out experimentally. **Figure 4-1** is a flowchart of typical cross validation workflow in model training. The best parameters can be determined by grid search techniques, which we’ll discuss later as **hyperparameter-tuning**.

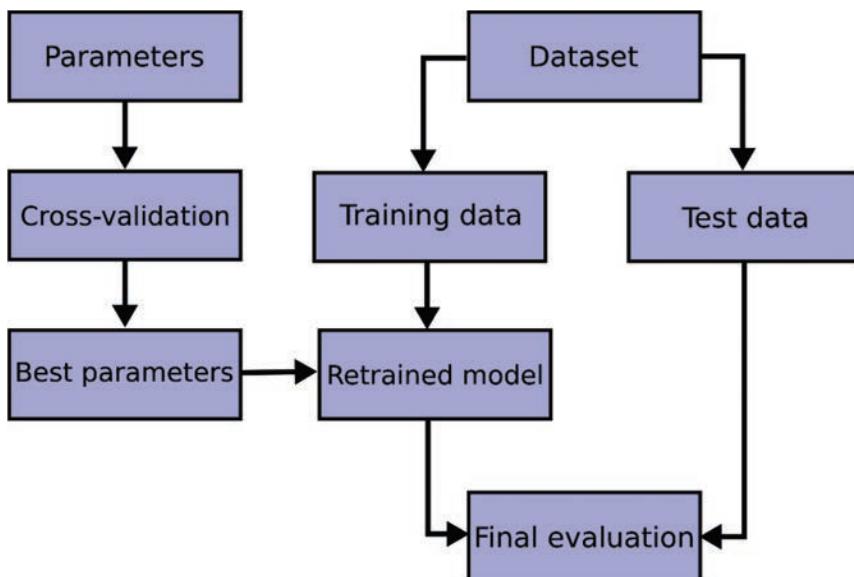


Figure 4-1. This is a flowchart to guide partitioning data for training, cross-validation, and testing a model (Source: Scikit-Learn, 3.1. Cross-validation).

In **Figure 4-1**, the test set on the right is sometimes referred to as the **validation set**, used to “validate” a model. We’ll partition our data using the function `train_test_split`. Our objective is to have training and testing data to train and cross-validate our models.

```
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
```

How much training and testing data is enough? The `test_size` parameter sets the proportion of data that is split into the testing set. Above, the testing set is 10% of the total dataset, but is that enough data for testing (validation)? To answer our question, we’ll perform another experiment (later, we’ll discuss an approach when we have very little data).

Here we’ll use the function `train_test_split` to split up the dataset such that 40% of the data samples will be utilized for testing. Let’s 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,
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
```

Next, let’s import the `LinearRegression` function from the module `linear_model`.

```
from sklearn.linear_model import LinearRegression
```

Here, we create a Linear Regression object.

```
lre = LinearRegression ()
```

Fist, we'll fit a 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)  
0.3635875575078824
```

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)  
0.6619724197515103
```

Let's 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, random_state=0)  
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 0.29118394447560286

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([20254142.84026705, 43745493.2650517 ,  
12539630.34014931, 17561927.72247591])
```

Now, let's calculate the average R^2 using two folds, then find the average R^2 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])
```

Now, let's perform a 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.

```
%matplotlib inline
import seaborn as sns
```

Let's examine the distribution of the predicted values of the training data as it appears in **Figure 4-2**.

```
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)
```

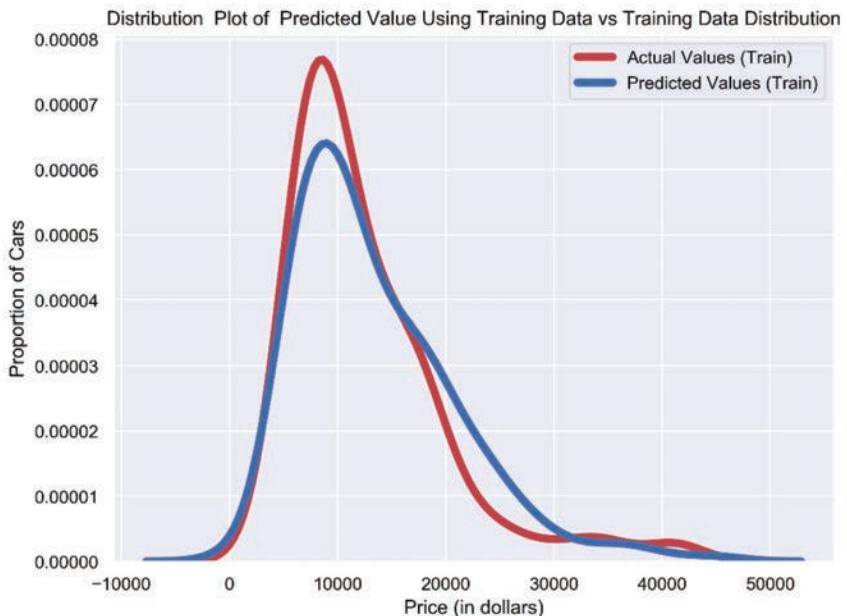


Figure 4-2. The plot of predicted values using the training data compared to the actual values of the training data.

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 as seen in **Figure 4-3**.

```
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)
```

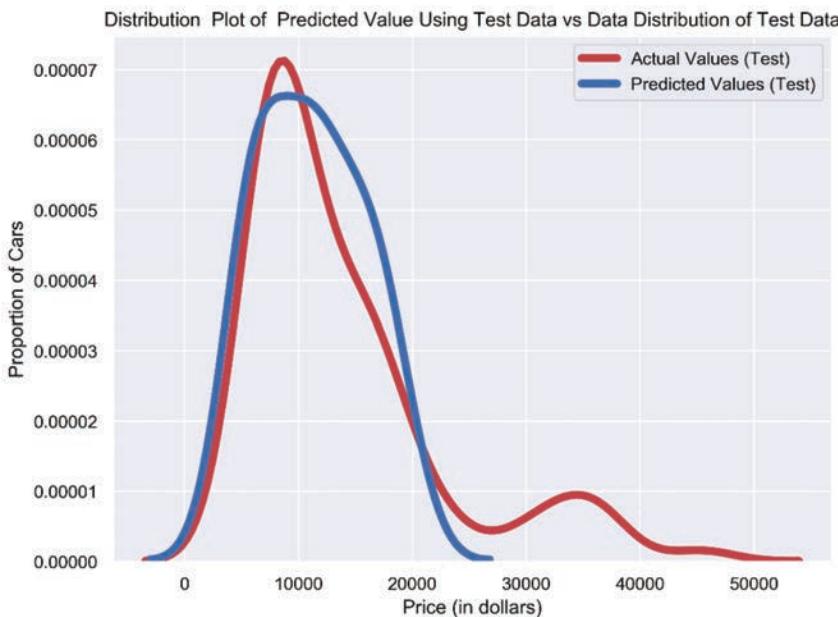


Figure 4-3. The plot of predicted value using the test data compared to the actual values of the test data.

Comparing **Figure 4-2** and **Figure 4-3**, it is evident that the distribution of the test data in Figure 1 is much better at fitting the data. This difference in **Figure 4-3** 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 analyzing 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']])
```

```
PolynomialFeatures(degree=5, include_bias=True, ...)
```

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, ...)
```

We can see the output of our model using the method predict. We assign the values to \hat{y} .

```
yhat = poly.predict(x_test_pr)  
yhat[0:5]
```

```
array([ 6728.77547414,  7308.09791963, 12213.83954918,  
18893.06169793, 19995.73194318])
```

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: [ 6728.77547414  7308.09791963  
12213.83954918 18893.06169793]  
True values: [ 6295. 10698. 13860. 13499.]
```

We will use the function PollyPlot that we defined at the beginning of the chapter to display the training data, testing data, and the predicted function in **Figure 4-4**.

```
PollyPlot (x_train[['horsepower']],  
x_test[['horsepower']], y_train, y_test, poly, pr)
```

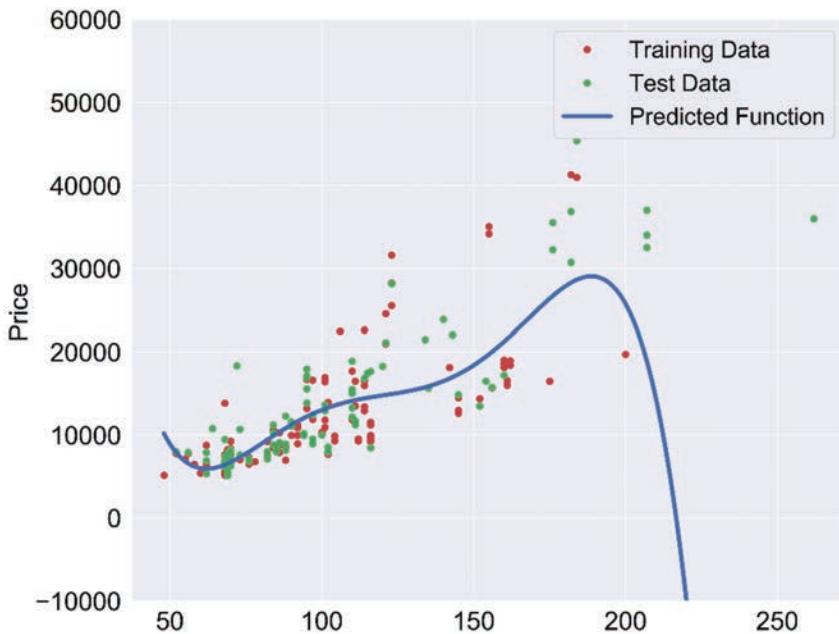


Figure 4-4. A polynomial regression model where red dots represent training data, green dots represent test data, and the blue line represents the model prediction.

We see that the estimated function appears to track the data but around 200 horsepower, the function begins to diverge from the data points.

Now, we'll calculate R^2 of the training data.

```
poly.score(x_train_pr, y_train)
0.5567716899771691
```

Next, we'll calculate R^2 of the test data:

```
poly.score (x_test_pr, y_test)
-29.87184147296421
```

We see the R^2 for the training data is 0.5567 while the R^2 on the test data was -29.87. The lower the R^2 the worse the model. A negative R^2 is a sign of overfitting.

Let's see how the R^2 changes on the test data for different order polynomials and then plot the results in **Figure 4-5**.

```

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))
width = 8
height = 6
fig = plt.figure(figsize= (width, height))
plt.plot(order, Rsqu_test, linewidth = 4)
plt.xlabel('order')
plt.ylabel('R^2')
plt.title('R^2 Using Test Data')
plt.text(3, 0.75, 'Maximum R^2')
fig.savefig('xFig_01.png', bbox_inches='tight', dpi=300)

```

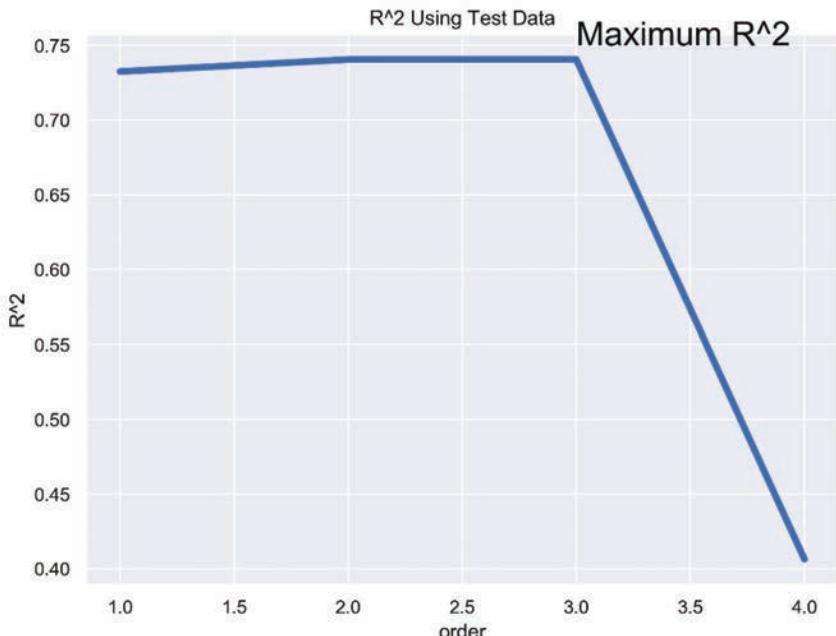


Figure 4-5. The plot shows the change in R^2 with changes in the orders of the polynomials.

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):
    width = 8
    height = 6
    plt.figure(figsize=(width, height))
    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)
    fig = PollyPlot(x_train[['horsepower']],
                    x_test[['horsepower']], y_train, y_test, poly, pr)

    fig.savefig('xFig_01.png', bbox_inches='tight', dpi=300)
```

The following interface allows you to experiment with different polynomial orders and different amounts of data. The plot for the parameters polynomial order = 0, 6, and 1, with the defined `test_data` is shown in **Figure 4-6**.

```
interact(f, order=(0, 6, 1), test_data=(0.05, 0.95, 0.05))
```

Earlier, we mentioned “overdoing” and “underdoing” figures, at least for presentation. Overdoing, most often occurs, and is characterize by multi-color or gradient backgrounds, muti-color grid lines, multi-color labels and so on. Unless you are using colors to bring out an import aspect of the figure, you should be careful of overdoing it.

In contrast, underdoing may be very unappealing, for example, no axes labels, no titles, no legend, no grid-lines, and so on.

Finding the right blend of aesthetics, is an art and depends heavily on who your target audience is.

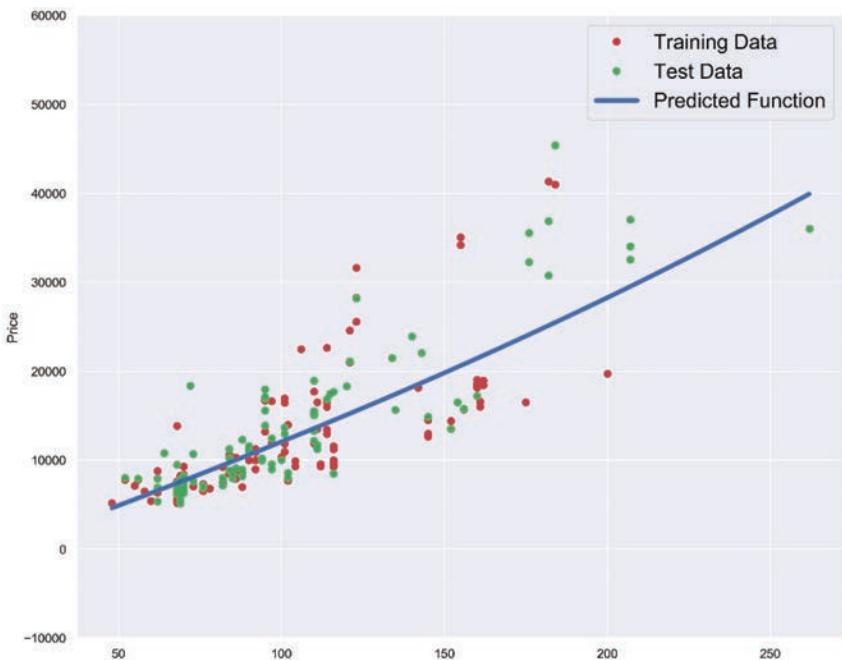


Figure 4-6. The plot shows the predictive function for various polynomial orders and

We can perform polynomial transformations with more than one feature. Create a [PolynomialFeatures](#) object `pr1` of degree two.

```
pr1 = PolynomialFeatures(degree=2)
```

Next, let's transform the training and testing samples for the features horsepower, curb-weight, engine-size, and highway-mpg. Hint: We'll use the method [fit_transform](#).

[fit_transform\(\)](#) is used on the training data so that we can scale it, and to learn the scaling parameters of that data. It combines the functionality of [fit\(\)](#) and [transform\(\)](#). It is used for calculating the initial parameters on the training data and later saves them as internal objects state. This method calculates the parameters μ (mean) and σ (standard deviation) and saves them as internal objects.

We use [transform\(\)](#) for the initial calculated mu and sigma values and return modified training data as output. These are calculated weights

for the training data. We then use the calculated weights on the test data to make the predictions. Doing this presents so-called “data leakage.” So, `fit_transform()` is the general procedure to scale the data when building a machine learning model. So that the model is not biased to a specific feature and prevents our model to learn the trends of our test data at the same time.

So, let's now implement this in our Jupyter notebook:

```
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']])
```

How many dimensions does the new feature have? We can use the attribute shape to address this question.

```
train_shapep = x_train_pr1.
print('There are ', train_shapep, 'features')
```

There are (110, 15) features

Now, let's 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)
```

Let's 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, and display it in **Figure 4-7**.

```
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)
```

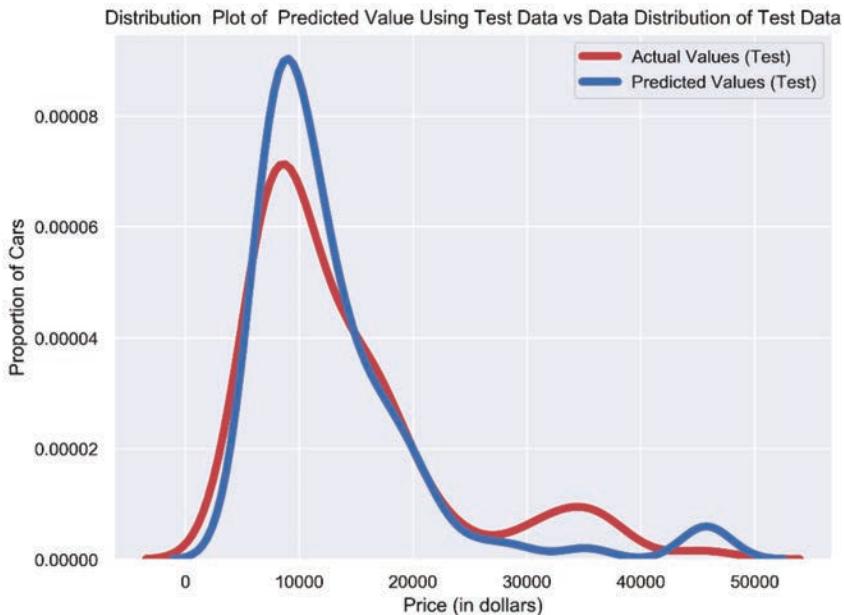


Figure 4-7. The distribution plot of the predicted test outputs vs. the actual test data.

Using the distribution plot above, we'll describe 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: Cross-Validation

K-Folds Cross Validation:

K-Folds technique is a popular and easy to understand, it generally results in a less biased model compare to other methods. Because it ensures that every observation from the original dataset has the chance of appearing in training and test set. This is one among the best approach if we have a limited input data. This method follows the below steps, using **Figure 4-8**.

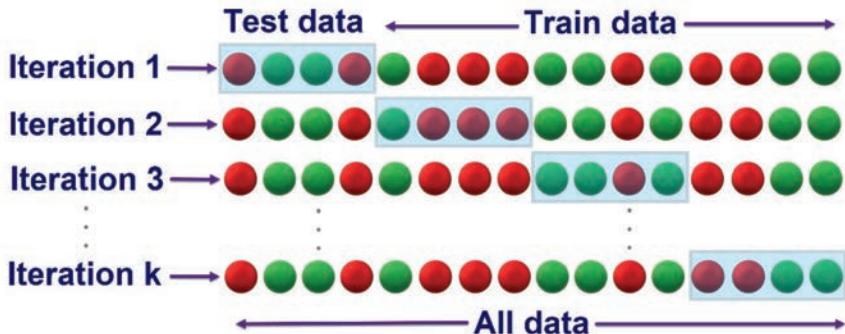


Figure 4-8. K-Fold Cross Validation process diagram, where we take a different sample from the same data for k iterations.

1. Split the entire data randomly into K folds (value of K shouldn't be too small or too high, ideally, we choose 5 to 10 depending on the data size). The higher value of K leads to less biased model (but large variance might lead to over-fit), whereas the lower value of K is similar to the train-test split approach we saw before.
2. Then fit the model using the $K - 1$ (K minus 1) folds and validate the model using the remaining K th fold. Note down the scores/errors.
3. Repeat this process until every K -fold serve as the test set. Then take the average of your recorded scores. That will be the performance metric for the model.

We can write a logic manually to perform this or we can use the built in `cross_val_score/cross_val_predict` functions of `scikit-learn` (`cross_val_score` returns score of each test folds, and `cross_val_predict` returns the predicted score for each observation in the input dataset when it was part of the test set) from the `scikit_learn` library.

If the estimator (model) is a Classifier and `y` (target variable) is either binary/multiclass, then `StratifiedKFold` technique is used by default. In all other cases `K-Fold` technique is used as a default to split and train the model.

We could do this for the `ploy1` model we just built using the

`PolynomialFeatures` data, `X_train_pr1` and `X_text_pr1`, and three folds:

```
from sklearn.model_selection import cross_val_score
cv_r2_scores = cross_val_score(poly1, X_train_pr1,
                                y_train, cv=3)
print(cv_r2_scores_rf)
print("Mean 5-fold R Squared
      {}".format(np.mean (cv_r2_scores_rf)))
[0.71090232 0.76046369 0.73245767]
Mean 5-fold R Squared 0.7346078963962787
```

So, taking the average of a 3-fold R^2 , we get 0.7346.

Part 4: 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', '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.

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

Let's create a Ridge regression object, setting the regularization parameter (α) 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, we can obtain a prediction.

```
yhat = RigeModel.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  9636.24891471 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.

```
from tqdm import tqdm

Rsqu_test = []
Rsqu_train = []
dummy1 = []
Alpha = 10 * np.array (range(0,1000))
pbar = tqdm(Alpha)

for alpha in pbar:
    RigeModel = Ridge(alpha=alpha)
    RigeModel.fit(x_train_pr, y_train)
    test_score, train_score = RigeModel.score (x_test_pr,
y_test), RigeModel.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 [00:02<00:00, 342.35it/s, Test
Score=0.564, Train Score=0.859]
```

We plot out the value of R^2 for different alphas in **Figure 4-9**:

```
width = 8
height = 6
fig = plt.figure(figsize = (width, height))
```

```

plt.plot(Alpha, Rsqu_test, label='validation data',
        linewidth=4)
plt.plot(Alpha, Rsqu_train, 'r', label='training Data',
        linewidth=4)
plt.rc('font', size=18)
plt.xlabel('alpha')
plt.ylabel('R^2')
plt.legend()
fig.savefig('xFig_01.png', bbox_inches='tight', dpi=300)

```

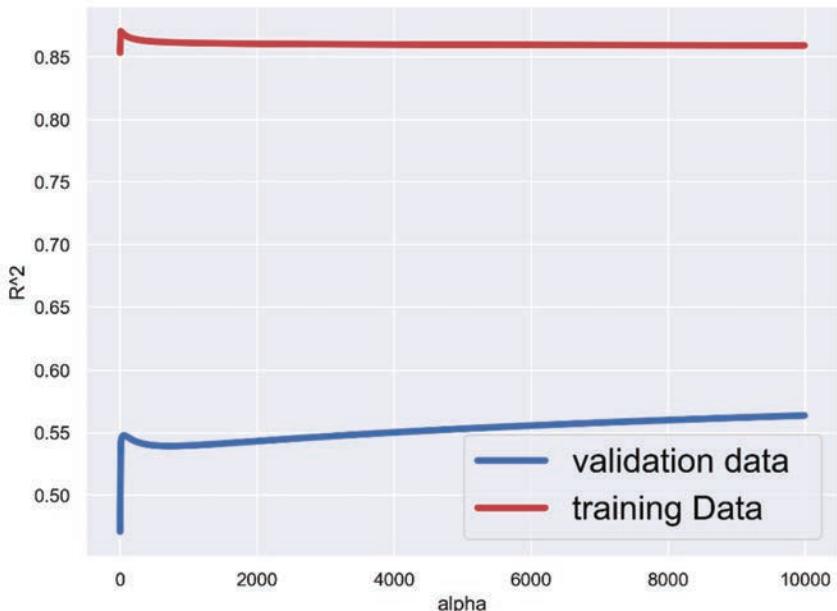


Figure 4-9. The blue line represents the R^2 of the validation data, and the red line represents the R^2 of the training data. The x-axis represents the different values of α .

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 4-9** represents the R^2 of the training data. As α increases the R^2 decreases. Therefore, as α increases, the model performs worse on the training data

The blue line represents the R^2 on the validation data. As the value

for α increases, the R^2 increases and converges at a point.

Next, we'll 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 α should be set to 10.

```
RidgeModel = Ridge (alpha=10)
RidgeModel.fit (x_train_pr, y_train)
RidgeModel.score (x_test_pr, y_test)
```

```
0.541857644020735
```

Part 5: Grid Search

The term α is a **hyperparameter**. **Sklearn** has the class **GridSearchCV** to make the process of finding the best hyperparameter simpler. So, let's import **GridSearchCV** from the module **model_selection**.

```
from sklearn.model_selection import GridSearchCV
```

Here, we create a dictionary of parameter values.

```
parameters1= [{'alpha': [0.001,0.1,1, 10, 100, 1000,
10000, 100000, 1000000]}]
parameters1
[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 1000000]}]
```

Now, we 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)
```

Next, we create a ridge grid search object, using **GridSearchCV**.

```
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. Next, we 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, random_state=None,
            solver='auto', tol=0.001), 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=False, 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:

```
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)
```

```
0.8411649831036149
```

Finally, we 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)
```


5. Data Visualization

Objectives

After completing this chapter, you will be able to:

- Exploring Datasets with *pandas*
- Indexing and Selection data using pandas
- Create data visualizations using the standard Python visualization library
- Use various Python the Matplotlib libraries for visualization

Introduction

The aim of this chapter is to introduce you to data visualization with Python as concrete and as consistent as possible. Speaking of consistency, because there is no *best* data visualization library available for Python, we have to introduce different libraries and show their benefits when we are discussing new visualization concepts. Doing so, we hope to make beginning data scientists well-rounded with visualization libraries and concepts so that they are able to judge and decide on the best visualization technique and tool for a given problem *and* audience.

The Data

In this chapter, we'll work with the *Pandas* package and the dataset, Immigration to Canada from 1980 to 2013.

Part 1: Exploring Datasets with *pandas*

Pandas is an essential data analysis toolkit for Python. From their website (<http://pandas.pydata.org/>):

"pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python."

The course heavily relies on *pandas* for data wrangling, analysis, and

visualization. We encourage you to spend some time and familiarize yourself with the *pandas* API Reference: <http://pandas.pydata.org/pandas-docs/stable/api.html>.

The Dataset: Immigration to Canada from 1980 to 2013

Dataset Source: [International migration flows to and from selected countries - The 2015 revision](#).

The dataset contains annual data on the flows of international immigrants as recorded by the countries of destination. The data presents both inflows and outflows according to the place of birth, citizenship or place of previous / next residence both for foreigners and nationals. The current version presents data pertaining to 45 countries.

In this chapter, we will focus on the Canadian immigration data. **Figure 5-1** shows a partial screenshot of the *Excel* dataset. Notice that the data start after row 20, with the header in row 21.



United Nations
Population Division
Department of Economic and Social Affairs

International Migration Flows to and from Selected Countries: The 2015 Revision

POP/DB/MIG/Flow/Rev.2015
December 2015 - Copyright © 2015 by United Nations. All rights reserved
Suggested citation: United Nations, Department of Economic and Social Affairs, Population Division (2015).
International Migration Flows to and from Selected Countries: The 2015 Revision. (United Nations database, POP/DB/MIG/Flow/Rev.2015).

Reporting country: Canada
Criterion: Citizenship

Classification		Origin/Destination	Major area		Region		Development region	
Type	Coverage	OdName	AREA	AreaName	REG	RegName	DEV	DevName
Immigrants	Foreigners	Afghanistan	935	Asia	5501	Southern Asia	902	Developing regions
Immigrants	Foreigners	Albania	908	Europe	925	Southern Europe	901	Developed regions
Immigrants	Foreigners	Algeria	903	Africa	912	Northern Africa	902	Developing regions
Immigrants	Foreigners	American Samoa	909	Oceania	957	Polynesia	902	Developing regions
Immigrants	Foreigners	Andorra	908	Europe	925	Southern Europe	901	Developed regions
Immigrants	Foreigners	Angola	903	Africa	911	Middle Africa	902	Developing regions
Immigrants	Foreigners	Antigua and Barbuda	904	Latin America	915	Caribbean	902	Developing regions

Figure 5-1. Partial view of the Excel workbook with Canadian immigration .

pandas Basics

The first thing we'll do is import two key data analysis modules: *pandas* and *numpy*.

```
# Useful for many scientific computing in Python
import numpy as np
# Primary data structure library
import pandas as pd
```

Let's download and import our primary Canadian Immigration dataset using `pandas's read_excel()` method. Normally, before we can do that, we would need to download a module which `pandas` requires reading in Excel files. This package was `openpyxl` (formerly `xlrd`). If this package is not installed on your system, you would need to run the following line of code to install the `openpyxl` module:

```
! pip install openpyxl
```

Now we are ready to read in our data.

```
df_can = pd.read_excel(
    'https://github.com/stricje1/jupyter/blob/main/data/Canada
.xlsx?raw=true', sheet_name='Canada by Citizenship',
skiprows=range(20), skipfooter=2)
print('Data read into a pandas dataframe!')
```

Data read into a pandas dataframe!

Let's view the top 5 rows of the dataset using the `head()` function. You can specify the number of rows you'd like to see as follows by entering an integer inside the parentheses, like `df_can.head(10)`.

```
df_can.head()
```

Type	Coverage	REG	RegName	1980	...	2013	
0	Immigrants	Foreigners	5501	Southern Asia	16	...	2004
1	Immigrants	Foreigners	925	Southern Europe	1	...	603
2	Immigrants	Foreigners	912	Northern Africa	80	...	4331
3	Immigrants	Foreigners	957	Polynesia	0	...	0
4	Immigrants	Foreigners	925	Southern Europe	0	...	1

5 rows × 43 columns

We can also view the bottom 5 rows of the dataset using the `tail()` function.

```
df_can.tail()
```

Type	Coverage	REG	RegName	1980	...	2013	
190	Immigrants	Foreigners	920	South-Eastern Asia	1191	...	2112
191	Immigrants	Foreigners	912	Northern Africa	0	...	0

Type	Coverage	REG	RegName	1980	... 2013
192	Immigrants	Foreigners	922	Western Asia	1 ... 217
193	Immigrants	Foreigners	910	Eastern Africa	11 ... 59
194	Immigrants	Foreigners	910	Eastern Africa	72 ... 407

5 rows × 43 columns

When analyzing a dataset, it's always a good idea to start by getting basic information about your dataframe. We can do this by using the `info()` method.

This method can be used to get a short summary of the dataframe.

```
df_can.info(verbose=False)
```

RangeIndex: 195 entries, 0 to 194
 Columns: 43 entries, Type to 2013
 dtypes: int64(37), object(6)
 memory usage: 65.6+ KB

To get the list of column headers we can call upon the data frame's **columns instance variable**.

```
df_can.columns
```

Index(['Type', 'Coverage', 'OdName', 'AREA',
 'AreaName', 'REG', 'RegName', 'DEV', 'DevName', 1980,
 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989,
 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998,
 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007,
 2008, 2009, 2010, 2011, 2012, 2013],
 dtype='object')

Similarly, to get the list of indices we use the `.index` instance variables.

```
df_can.index
```

RangeIndex(start=0, stop=195, step=1)

```
print(type(df_can.columns))
print(type(df_can.index))
```

To get the index and columns as lists, we can use the `tolist()` method.

```
df_can.columns.tolist()
```

```
[ 'Type', 1986, 2000,
  'Coverage', 1987, 2001,
  'OdName', 1988, 2002,
  'AREA', 1989, 2003,
  'AreaName', 1990, 2004,
  'REG', 1991, 2005,
  'RegName', 1992, 2006,
  'DEV', 1993, 2007,
  'DevName', 1994, 2008,
  1980, 1995, 2009,
  1981, 1996, 2010,
  1982, 1997, 2011,
  1983, 1998, 2012,
  1984, 1999, 2013]
  1985,
```

```
df_can.index.tolist()
```

```
[0, 25, 50, 75, 100, 125, 150, 175,
 1, 26, 51, 76, 101, 126, 151, 176,
 2, 27, 52, 77, 102, 127, 152, 177,
 3, 28, 53, 78, 103, 128, 153, 178,
 4, 29, 54, 79, 104, 129, 154, 179,
 5, 30, 55, 80, 105, 130, 155, 180,
 ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮
19, 44, 69, 94, 119, 144, 169, 194]
20, 45, 70, 95, 120, 145, 170,
21, 46, 71, 96, 121, 146, 171,
22, 47, 72, 97, 122, 147, 172,
23, 48, 73, 98, 123, 148, 173,
24, 49, 74, 99, 124, 149, 174,
```

```
print(type(df_can.columns.tolist()))
print(type(df_can.index.tolist()))
```

To view the dimensions of the dataframe, we use the `shape` instance variable of it.

```
# size of dataframe (rows, columns)
df_can.shape
```

(195, 43)

Note: The main types stored in `pandas` objects are `float`, `int`, `bool`, `datetime64[ns]`, `datetime64[ns, tz]`, `timedelta[ns]`, `category`, and `object` (string). In addition, these dtypes have item sizes, e.g., `int64` and `int32`.

Let's clean the data set to remove a few unnecessary columns. We can use `pandas drop()` method as follows:

```
# In pandas, axis=0 represents rows (default) and axis=1  
# represents columns.  
df_can.drop(['AREA', 'REG', 'DEV', 'Type', 'Coverage'],  
axis=1, inplace=True)  
df_can.head(2)
```

	OdName	AreaName	RegName	DevName	1980 ...	2013
0	Afghanistan	Asia	Southern Asia	Developing regions	16 ... 2004	
1	Albania	Europe	Southern Europe	Developed regions	1 ... 603	

2 rows × 38 columns

Let's rename the columns so that they make sense. We can use `rename()` method by passing in a dictionary of old and new names as follows:

```
df_can.rename(columns={'OdName': 'Country',  
'AreaName': 'Continent', 'RegName': 'Region'},  
inplace=True)  
df_can.columns  
Index(['Country', 'Continent', 'Region', 'DevName', 1980,  
1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989,  
1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998,  
1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007,  
2008, 2009, 2010, 2011, 2012, 2013], dtype='object')
```

We will also add a `Total` column that sums up the total immigrants by country over the entire period 1980 - 2013, as follows:

```
df_can['Total'] = df_can.sum(axis=1)
```

We can check to see how many null objects we have in the dataset as follows:

```
df_can.isnull().sum()
```

Country	0	1995	0
Continent	0	1996	0
Region	0	1997	0
DevName	0	1998	0
1980	0	1999	0
1981	0	2000	0
1982	0	2001	0
1983	0	2002	0
1984	0	2003	0
1985	0	2004	0
1986	0	2005	0
1987	0	2006	0
1988	0	2007	0
1989	0	2008	0
1990	0	2009	0
1991	0	2010	0
1992	0	2011	0
1993	0	2012	0
1994	0	2013	0
		Total	0

Finally, let's view a quick summary of each column in our dataframe using the `describe()` method.

```
df_can.describe()
```

	1980	1981 ...	2012	2013	Total
count	195.000	195.000 ...	195.000	195.000	195.000
mean	508.394	566.989 ...	1313.958	1320.702	32867.451
std	1949.588	2152.643 ...	4247.555	4237.951	91785.498
min	0.000	0.000 ...	0.00	0.000	1.000000
25%	0.000	0.000 ...	42.50	45.000	952.000
50%	13.000	10.000 ...	233.00	213.000	5018.000
75%	251.500	295.500 ...	783.00	796.000	22239.500
max	22045.00	24796.00 ...	34315.00	34129.000	691904.000

8 rows × 35 columns

pandas Intermediate: Indexing and Selection (slicing)

Select Column

There are two ways to filter on a column name:

Method 1: Quick and easy, but only works if the column name does NOT have spaces or special characters.

```
df.column_name # returns series
```

Method 2: More robust, and can filter on multiple columns.

```
df['column'] # returns series  
df[['column 1', 'column 2']] # returns dataframe
```

Example: Let's try filtering on the list of countries ('Country').

```
df_can.Country # returns a series
```

```
0      Afghanistan  
1          Albania  
2          Algeria  
3   American Samoa  
4          Andorra  
...  
191     Western Sahara  
192          Yemen  
193          Zambia  
194        Zimbabwe  
Name: Country, Length: 195, dtype: object
```

Let's try filtering on the list of countries (`Country`) and the data for years: 1980 - 1985.

```
df_can[['Country', 1980, 1981, 1982, 1983, 1984, 1985]] #  
returns a dataframe
```

Notice that `Country` is string, and the years are integers. For the sake of consistency, we will convert all column names to string later on.

	Country	1980	1981	1982	1983	1984	1985
0	Afghanistan	16	39	39	47	71	340
1	Albania	1	0	0	0	0	0
2	Algeria	80	67	71	69	63	44
3	American Samoa	0	1	0	0	0	0
4	Andorra	0	0	0	0	0	0
...
190	Viet Nam	1191	1829	2162	3404	7583	5907
191	Western Sahara	0	0	0	0	0	0
192	Yemen	1	2	1	6	0	18
193	Zambia	11	17	11	7	16	9
194	Zimbabwe	72	114	102	44	32	29

195 rows × 7 columns

Select Row

There are two main ways to select rows:

```
df.loc[label] # filters by the labels of the index/column  
df.iloc[index] # filters by the positions of the  
index/column
```

Before we proceed, notice that the default index of the dataset is a numeric range from 0 to 194. This makes it very difficult to do a query by a specific country. For example, to search for data on Japan, we need to know the corresponding index value.

This can be fixed very easily by setting the `Country` column as the index using `set_index()` method.

```
df_can.set_index('Country', inplace=True)  
df_can.head(3)
```

	Continent	Region	1980	1981	...	2013	Total
Country							
Afghanistan	Asia	Southern Asia	16	39	...	2004	58639
Albania	Europe	Southern Europe	1	0	...	603	15699
Algeria	Africa	Northern Africa	80	67	...	4331	69439

3 rows × 38 columns

Note that the opposite of set is reset. So, to reset the index, we can use `df_can.reset_index()`. To remove the index is optional.

There are two main options to achieve the selection and indexing activities in Pandas, which can be confusing. The two selection cases and methods are position-based selections (i.e., rows, columns, etc.)

1. Selecting data by row numbers (`.iloc`)
2. Selecting data by label or by a conditional statement (`.loc`)

```
# Optional: to remove the name of the index  
df_can.index.name = None
```

Example: Let's view the number of immigrants from Japan (row 87) for the following scenarios:

1. The full row data (all columns)
2. For year 2013
3. For years 1980 to 1985

```
# 1. the full row data (all columns)
df_can.loc['Japan']
```

```
Continent           Asia
Region            Eastern Asia
DevName          Developed regions
1980                  701
1981                  756
1982                  598
1983                  309
:                   :
2011                 1265
2012                 1214
2013                  982
Total                27707
Name: Japan, dtype: object
```

```
# Alternate methods
df_can.iloc[87] # Japan is row 87
```

```
Continent           Asia
Region            Eastern Asia
DevName          Developed regions
1980                  701
1981                  756
1982                  598
1983                  309
:                   :
2011                 1265
2012                 1214
2013                  982
Total                27707
Name: Japan, dtype: object
```

```
df_can[df_can.index == 'Japan']
```

	Continent	Region	DevName	1980	...	2013	Total
Japan	Asia	Eastern	Developed	701	...	982	27707
		Asia	regions				

1 rows x 38 columns

```
# 2. for year 2013
df_can.loc['Japan', 2013]
```

982

```
# Alternate method  
# 2013 is the last column with a positional index of 36  
df_can.iloc[87, 36]  
982  
  
# 3. for years 1980 to 1985  
df_can.loc ['Japan', [1980, 1981, 1982, 1983, 1984,  
1985]]
```

```
1980    701  
1981    756  
1982    598  
1983    309  
1984    246  
1984    246  
Name: Japan, dtype: object
```

```
# Alternative method  
df_can.iloc [87, [3, 4, 5, 6, 7, 8]]
```

```
1980    701  
1981    756  
1982    598  
1983    309  
1984    246  
1985    198  
Name: Japan, dtype: object
```

Column names that are integers (such as the years) might introduce some confusion. For example, when we are referencing the year 2013, one might confuse that when the 2013th positional index.

To avoid this ambiguity, let's convert the column names into strings: '1980' to '2013'.

```
df_can.columns = list(map(str, df_can.columns))  
# [print (type(x)) for x in df_can.columns.values] <--  
# uncomment to check type of column headers
```

Since we converted the years to string, let's declare a variable that will allow us to easily call upon the full range of years:

```
# Useful for plotting Later on  
years = list(map(str, range(1980, 2014)))  
years
```

```
[ '1980',          '1989',          '1998',          '2007',
  '1981',          '1990',          '1999',          '2008',
  '1982',          '1991',          '2000',          '2009',
  '1983',          '1992',          '2001',          '2010',
  '1984',          '1993',          '2002',          '2011',
  '1985',          '1994',          '2003',          '2012',
  '1986',          '1995',          '2004',          '2013'],
  '1987',          '1996',          '2005',          '',
  '1988',          '1997',          '2006',          '' ]
```

Filtering based on a criteria

To filter the dataframe based on a condition, we simply pass the condition as a Boolean vector. For example, let's filter the dataframe to show the data on Asian countries ([AreaName = Asia](#)).

```
# 1. create the condition Boolean series
condition = df_can['Continent'] == 'Asia'
print(condition)
```

```
Afghanistan      True
Albania          False
Algeria          False
American Samoa  False
Andorra          False
:
Viet Nam         True
Western Sahara  False
Yemen             True
Zambia            False
Zimbabwe         False
Name: Continent, Length: 195, dtype: bool
```

```
# 2. pass this condition into the DataFrame
df_can[condition]
```

Country	Continent	Region	1980	...	2013	Total
Afghanistan	Asia	Southern Asia	16	...	2004	58639
Armenia	Asia	Western Asia	0	...	207	3310
Azerbaijan	Asia	Western Asia	0	...	57	2649
Bahrain	Asia	Western Asia	0	...	32	475
Bangladesh	Asia	Southern Asia	83	...	3789	65568
Bhutan	Asia	Southern Asia	0	...	487	5876

Country	Continent	Region	1980	...	2013	Total
Brunei Darussalam	Asia	South-Eastern Asia	79	...	6	600
Cambodia	Asia	South-Eastern Asia	12	...	288	6538
China	Asia	Eastern Asia	5123	...	34129	659962
:	:	:	:	:	:	:
Saudi Arabia	Asia	Western Asia	0	...	267	3425
Singapore	Asia	South-Eastern Asia	241	...	141	14579
Sri Lanka	Asia	Southern Asia	185	...	2394	148358
State of Palestine	Asia	Western Asia	0	...	462	6512
Syrian Arab Republic	Asia	Western Asia	315	...	1009	31485
Tajikistan	Asia	Central Asia	0	...	39	503
Thailand	Asia	South-Eastern Asia	56	...	400	9174
Turkey	Asia	Western Asia	481	...	729	31781
Turkmenistan	Asia	Central Asia	0	...	14	310
United Arab Emirates	Asia	Western Asia	0	...	46	836
Uzbekistan	Asia	Central Asia	0	...	167	3368
Viet Nam	Asia	South-Eastern Asia	1191	...	2112	97146
Yemen	Asia	Western Asia	1	...	217	2985

49 rows × 38 columns

We can pass multiple criteria in the same line. Let's filter for `AreaName = Asia` and `RegName = Southern Asia`. Note: When using 'and' and 'or' operators, `pandas` requires we use `&` and `|` instead of `and` and `or`. Don't forget to enclose the two conditions in parentheses.

```
df_can[(df_can['Continent']=='Asia') &
(df_can['Region']=='Southern Asia')]
```

Country	Continent	Region	1980	...	2013	Total
Afghanistan	Asia	Southern Asia	16	...	2004	58639
Bangladesh	Asia	Southern Asia	83	...	3789	65568
Bhutan	Asia	Southern Asia	0	...	487	5876
India	Asia	Southern Asia	8880	...	33087	691904

Country	Continent	Region	1980	...	2013	Total
Iran (Islamic Republic of)	Asia	Southern Asia	1172	...	11291	175923
Maldives	Asia	Southern Asia	0	...	1	30
Nepal	Asia	Southern Asia	1	...	1308	10222
Pakistan	Asia	Southern Asia	978	...	12603	241600
Sri Lanka	Asia	Southern Asia	185	...	2394	148358

9 rows × 38 columns

Before we proceed, let's review the changes we have made to our dataframe. So, there are now 38 columns. Originally, we had 43 columns. We dropped `AREA`, `REG`, `DEV`, `Type`, and `Coverage`. We changed `OdName` to `Country`, `AreaName` to `Continent`, and `RegName` to `Region`. We now have 9 rows and there were originally 195 rows. We filtered row to give `Continent = Asia` and `Region = Southern Asia`.

```
print('data dimensions:', df_can.shape)
print(df_can.columns)
df_can.head(2)
```

```
data dimensions: (195, 38)
Index(['Continent', 'Region', 'DevName', '1980', '1981',
       '1982', '1983', '1984', '1985', '1986', '1987', '1988',
       '1989', '1990', '1991', '1992', '1993', '1994', '1995',
       '1996', '1997', '1998', '1999', '2000', '2001', '2002',
       '2003', '2004', '2005', '2006', '2007', '2008', '2009',
       '2010', '2011', '2012', '2013', 'Total'],
      dtype='object')
```

	Continent	Region	1980	1981	...	2013	Total
Afghanistan	Asia	Southern Asia	16	39	...	2004	58639
Albania	Europe	Southern Europe	1	0	...	603	15699

2 rows × 38 columns

Part 2: Visualizing Data using Matplotlib

Matplotlib: Standard Python Visualization Library

The primary plotting library we will explore in the chapter is **Matplotlib**. According to their website:

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell, the jupyter notebook, web application servers, and four graphical user interface toolkits.

If you are aspiring to create impactful visualization with python, Matplotlib is an essential tool to have at your disposal.

Matplotlib.Pyplot

One of the core aspects of **Matplotlib** is **matplotlib.pyplot**. It is Matplotlib's **scripting layer** which we studied in details in the videos about **Matplotlib**. Recall that it is a collection of command style functions that make **Matplotlib** work like **MATLAB**. Each **pyplot** function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc. In this chapter, we will work with the scripting layer to learn how to generate line plots. In future labs, we will get to work with the **artist layer** as well to experiment firsthand how it differs from the scripting layer.

Let's start by importing **Matplotlib** and **matplotlib.pyplot** as follows:

```
# we are using the inline backend  
%matplotlib inline  
  
import matplotlib as mpl  
import matplotlib.pyplot as plt
```

Check to see if Matplotlib is loaded with version 2.0.0 or greater.

```
print('Matplotlib version: ', mpl.__version__) # >= 2.0.0  
Matplotlib version: 3.1.1
```

We can also apply a style to Matplotlib.

```
print(plt.style.available)
mpl.style.use(['ggplot']) # for ggplot-like style
['bmh',      'classic',      'dark_background',      'fast',
 'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn-bright',
 'seaborn-colorblind', 'seaborn-dark-palette', 'seaborn-dark',
 'seaborn-darkgrid', 'seaborn-deep', 'seaborn-muted',
 'seaborn-notebook', 'seaborn-paper', 'seaborn-pastel',
 'seaborn-poster', 'seaborn-talk', 'seaborn-ticks',
 'seaborn-white', 'seaborn-whitegrid', 'seaborn',
 'Solarize_Light2', 'tableau-colorblind10',
 '_classic_test']
```

Plotting in pandas

Fortunately, [pandas](#) has a built-in implementation of [Matplotlib](#) that we can use. Plotting in [pandas](#) is as simple as appending a `.plot()` method to a series or dataframe.

Documentation:

- Plotting with Series
- Plotting with Dataframes

Line Pots (Series/Dataframe)

What is a line plot and why use it?

A line chart or line plot is a type of plot which displays information as a series of data points called **markers** connected by straight line segments. It is a basic type of chart common in many fields. Use line plot when you have a continuous data set. These are best suited for trend-based visualizations of data over a period of time.

Let's start with a case study:

In 2010, Haiti suffered a catastrophic magnitude 7.0 earthquake. The quake caused widespread devastation and loss of life and over three million people were affected by this natural disaster. As part of Canada's humanitarian effort, the Government of Canada stepped up its effort in accepting refugees from Haiti. We can quickly visualize this effort using a Line plot:

Question: Plot a line graph of immigration from Haiti using `df.plot()`.

First, we will extract the data series for Haiti.

```
haiti = df_can.loc['Haiti', years] # passing in years  
1980 - 2013 to exclude the 'total' column  
haiti.head()
```

```
1980    1666  
1981    3692  
1982    3498  
1983    2860  
1984    1418  
Name: Haiti, dtype: object
```

Next, we will plot a line plot by appending `.plot()` to the `haiti` dataframe as shown in **Figure 5-2**.

```
haiti.plot()
```

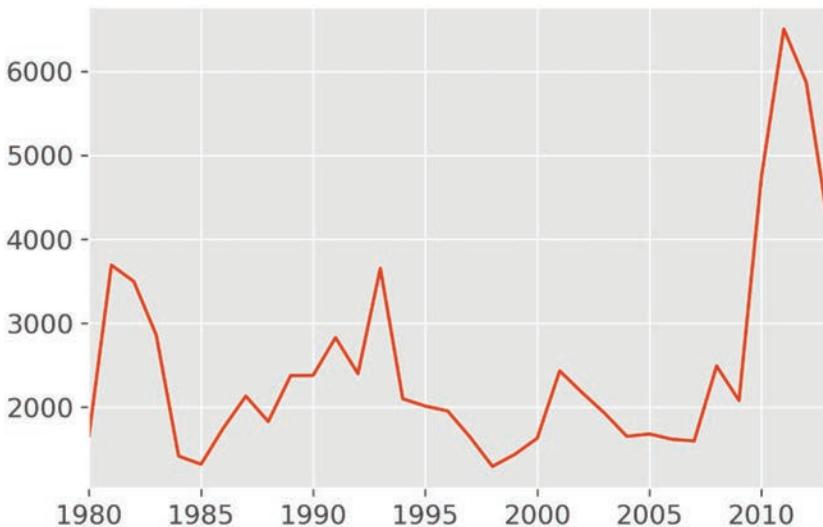


Figure 5-2. Immigration to Canada from Haiti, 1980 – 2013

`pandas` automatically populated the x-axis with the index values (years), and the y-axis with the column values (population). However, notice how the years were not displayed because they are of type *string*. Therefore, let's change the type of the index values to

integer for plotting.

Also, let's label the x and y axis using `plt.title()`, `plt.ylabel()`, and `plt.xlabel()` as follows and displayed in **Figure 5-3**.

```
plt.rcParams['figure.dpi'] = 300
haiti.index = haiti.index.map(int) # Let's change the
# index values of Haiti to type integer for plotting
haiti.plot(kind='line')

plt.title('Immigration from Haiti')
plt.ylabel('Number of immigrants')
plt.xlabel('Years')

plt.show() # need this line to show updates made to the
# figure
plt.savefig('VIS_02.png', bbox_inches='tight', dpi=300)
```

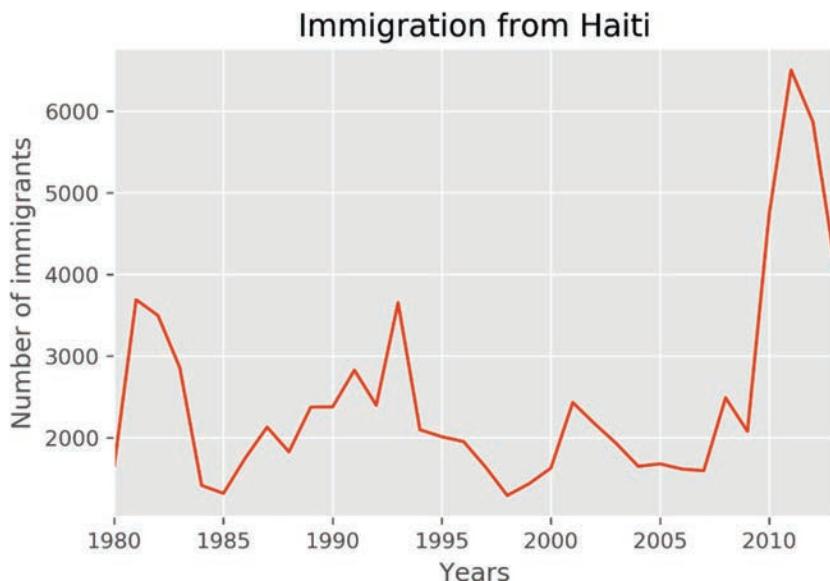


Figure 5-3. This plot is the same as the plot in **Figure 5-2** with title and labels added. This figure also increases the resolution using 300 dpi.

We can clearly notice how number of immigrants from Haiti spiked up from 2010 as Canada stepped up its efforts to accept refugees from Haiti. Let's annotate this spike in the plot by using the

`plt.text()` method, as shown in **Figure 5-4**.

```
plt.rcParams ['figure.dpi'] = 300
haiti.plot(kind='line')

plt.title('Immigration from Haiti')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

# Annotate the 2010 Earthquake.
# Syntax: plt.text(x, y, Label)
plt.text(2000, 6000, '2010 Earthquake') # see note below

plt.show()
```

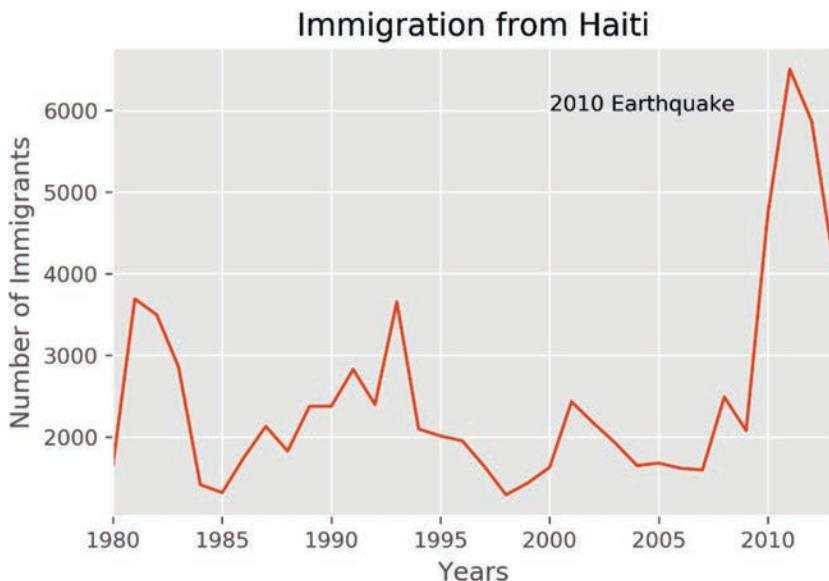


Figure 5-4. This plot is the same as the plot in **Figure 5-3** except the earthquake event label has been added.

With just a few lines of code, you were able to quickly identify and visualize the spike in immigration!

Quick note on x and y values in `plt.text(x, y, label)`:

With just a few lines of code, you were able to quickly identify and visualize the spike in immigration!

Quick note on x and y values in plt.text(x, y, label):

Since the x-axis (years) is type 'integer', we specified x as a year. The y axis (number of immigrants) is type 'integer', so we can just specify the value y = 6000.

```
plt.text(2000, 6000, '2010 Earthquake') # years stored as  
type int
```

If the years were stored as type 'string', we would need to specify x as the index position of the year. For example, 20th index is year 2000 since it is the 20th year with a base year of 1980.

```
plt.text(20, 6000, '2010 Earthquake') # years stored as  
type int
```

We will cover advanced annotation methods in later modules.

We can easily add more countries to line plot to make meaningful comparisons immigration from different countries.

Example: Let's compare the number of immigrants from India and China from 1980 to 2013.

Step 1: Get the data set for China and India, and display the dataframe.

```
df_CI = df_can.loc[['India', 'China'], years]  
df_CI
```

	1980	1981	1982	1983	...	2010	2011	2012	2013
India	8880	8670	8147	7338	...	34235	27509	30933	33087
China	5123	6682	3308	1863	...	30391	28502	33024	34129

2 rows x 34 columns

Step 2: Plot graph. We will explicitly specify line plot by passing in the `kind` parameter to the `plot()` function, and display the plot in **Figure 5-5**.

```
plt.rcParams['figure.dpi'] = 300  
df_CI.plot(kind='line')
```

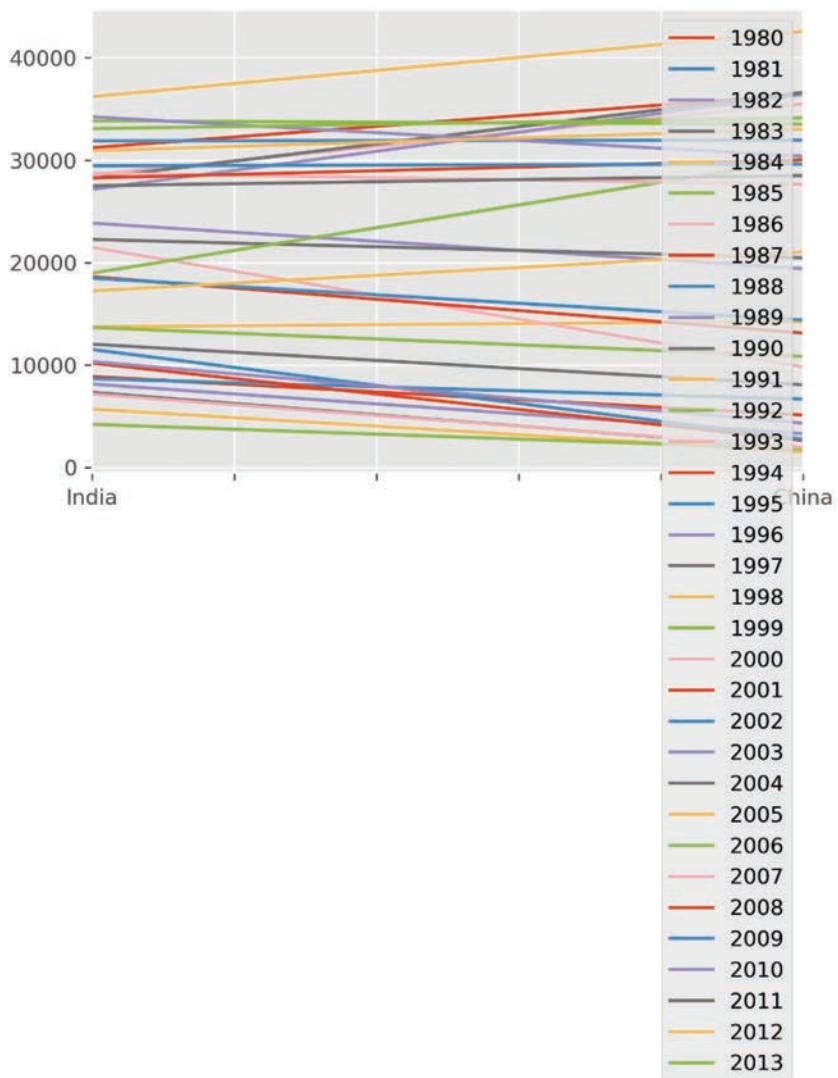


Figure 5-5. The line plot showing shows year as the columns resulting in something difficult to interpret and not what we wanted.

That doesn't look right...

Recall that **pandas** plots the indices on the x-axis and the columns as individual lines on the y-axis. Since `df_CI` is a dataframe with the country as the index and years as the columns, we must first transpose the dataframe using `transpose()` method to swap the

row and columns.

```
df_CI = df_CI.transpose()  
df_CI.head()
```

	India	China
1980	8880	5123
1981	8670	6682
1982	8147	3308
1983	7338	1863
1984	5704	1527

pandas will automatically graph the two countries on the same graph. Here we plot the new transposed dataframe, making sure to add a title to the plot and label the axes, as shown in **Figure 5-6**.

```
df_CI.plot(kind='line')  
  
plt.title('Immigration from China and India')  
plt.ylabel('Number of Immigrants')  
plt.xlabel('Years')  
# syntax: plt.text(x, y, Label)  
plt.show()
```

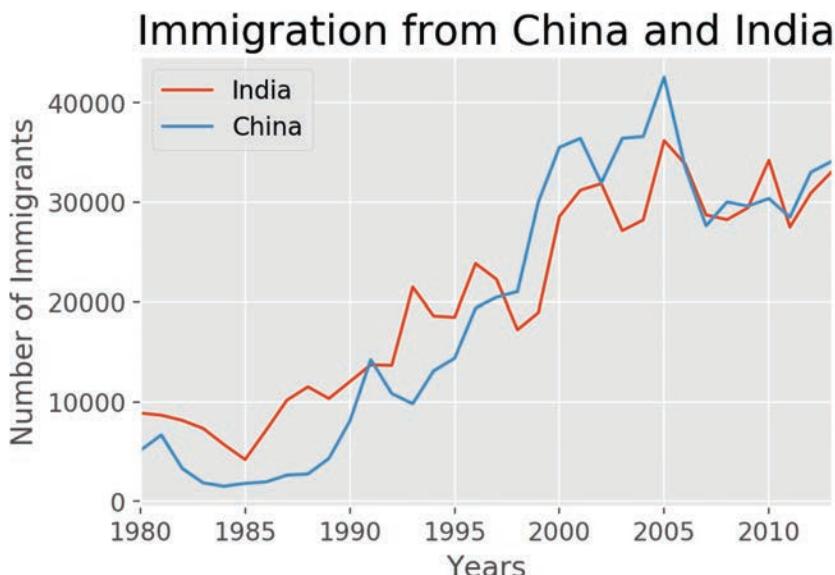


Figure 5-6. Line plot showing immigration to Canada from India and China from 1980 to 2013.

From the plot in **Figure 5-6**, we can observe that the China and India have very similar immigration trends through the years.

Note: How come we didn't need to transpose Haiti's dataframe before plotting (like we did for `df_CI`)?

That's because `haiti` is a series as opposed to a dataframe, and has the years as its indices as shown below.

```
print(type(haiti))
print(haiti.head(5))

class                                     'pandas.core.series.Series'
1980                                         1666
1981                                         3692
1982                                         3498
1983                                         2860
1984                                         1418
Name: Haiti, dtype: int64
```

A line plot is a handy tool to display several dependent variables against one independent variable. However, it is recommended that no more than 5-10 lines on a single graph; any more than that and it becomes difficult to interpret.

Example: Compare the trend of top 5 countries that contributed the most to immigration to Canada.

Step 1: Get the dataset. Recall that we created a Total column that calculates cumulative immigration by country. We will sort on this column to get our top 5 countries using pandas `sort_values()` method.

```
inplace = True # This parameter saves the changes to the
               # original df_can dataframe

df_can.sort_values(by='Total', ascending=False, axis=0,
                   inplace=True)

# get the top 5 entries
df_top5 = df_can.head(5)
# transpose the dataframe
df_top5 = df_top5[years].transpose()

print(df_top5)
```

Step 2: Plot the dataframe. To make the plot more readable, we will change the size using the `figsize` parameter.

```
plt.rcParams ['figure.dpi'] = 300

df_top5.index = df_top5.index.map(int) # Changes the index
                                         # values of df_top5 to type integer for plotting
df_top5.plot(kind='line', figsize= (14, 8), linewidth=3)
# pass a tuple (x, y) size

# Set the axes title font size
plt.rc('font', size = 16)
plt.rc('axes', titlesize = 24)
plt.rc('axes', labelsize = 18)
plt.rc('xtick', labelsize = 18)
plt.rc('ytick', labelsize = 18)
plt.rc('legend', fontsize = 12)
plt.title('Immigration Trend of Top 5 Countries')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

plt.show()
plt.savefig('VIS_03.png', bbox_inches='tight')
```

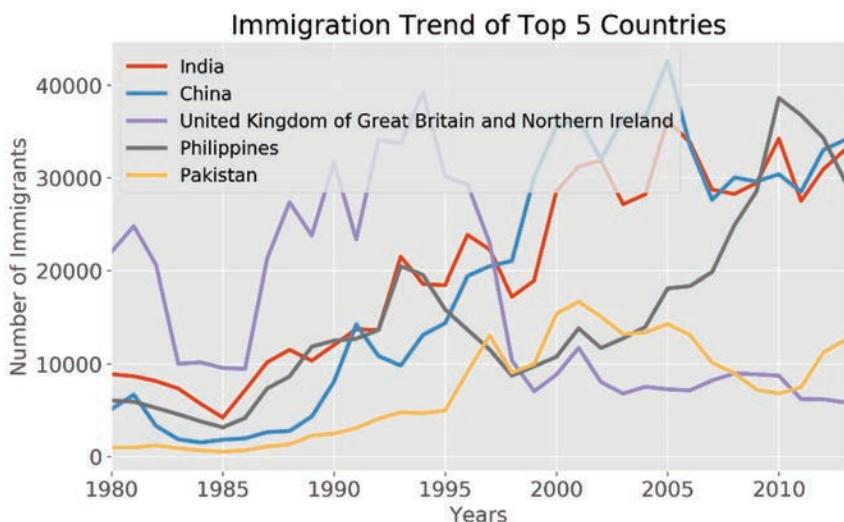


Figure 5-7. Line plot of top 5 countries immigrating to Canada from 1980-2013

Other Plots

Now we have learned how to wrangle data with **Python** and create a line plot with **Matplotlib**. There are many other plotting styles available other than the default Line plot, all of which can be accessed by passing `kind` keyword to `plot()`. The full list of available plots are as follows:

- `bar` for vertical bar plots
- `barh` for horizontal bar plots
- `hist` for histogram
- `box` for boxplot
- `kde` or `density` for density plots
- `area` for area plots
- `pie` for pie plots
- `scatter` for scatter plots
- `hexbin` for hexbin plot

6. Area Plots, Histograms, & Bar Plots

Objectives

In this chapter, we will discover how to:

- Explore datasets with **pandas**
- Download and preprocess data
- Visualizing data using Matplotlib
- Generate area plots
- Create histograms
- Create bar charts

Exploring Data with Pandas & Matplotlib

Toolkits:

The course heavily relies on **pandas** and **Numpy** for data wrangling, analysis, and visualization. The primary plotting library that we are exploring in the course is Matplotlib.

Dataset:

Immigration to Canada from 1980 to 2013 - [International migration flows to and from selected countries - The 2015 revision](#) from United Nation's website.

The dataset contains annual data on the flows of international migrants as recorded by the countries of destination. The data presents both inflows and outflows according to the place of birth, citizenship or place of previous / next residence both for foreigners and nationals. For this Chapter, we will focus on the Canadian Immigration data.

Downloading and Prepping Data

Import Primary Modules. The first thing we'll do is import two key data analysis modules: pandas and numpy.

```
import numpy as np # useful for scientific computing
import pandas as pd # primary data structure Library
```

Let's download and import our primary Canadian Immigration dataset using [pandas's read_excel \(\)](#) method. Normally, before we can do that, we would need to download a module which [pandas](#) requires reading in *Excel* files. This module was [openpyxl](#) (formerly [xlrd](#)). I pre-installed for this module on *Github*, so you would not have to worry about that. Otherwise, you would need to run the following line of code to install the [openpyxl](#) module:

```
! pip3 install openpyxl
```

Download the dataset and read it into a *pandas* dataframe.

```
df_can = pd.read_excel(
    'https://github.com/stricje1/jupyter/blob/main/data/Canada
.xlsx?raw=true', sheet_name='Canada by Citizenship',
skiprows=range(20), skipfooter=2)
print('Data read into a pandas dataframe!')
```

Data downloaded and read into a dataframe!

Let's take a look at the first five items in our dataset.

```
df_can.head()
```

Type	Coverage	REG	RegName	1980 ...	2013
0	Immigrants Foreigners	5501	Southern Asia	16	...
1	Immigrants Foreigners	925	Southern Europe	1	...
2	Immigrants Foreigners	912	Northern Africa	80	...
3	Immigrants Foreigners	957	Polynesia	0	...
4	Immigrants Foreigners	925	Southern Europe	0	...

5 rows × 43 columns

We can also view the bottom 5 rows of the dataset using the [tail\(\)](#) function.

Let's find out how many entries there are in our dataset.

```
# Print the dimensions of the dataframe  
print(df_can.shape)
```

(195, 43)

Clean up data. We will make some modifications to the original dataset to make it easier to create our visualizations. Refer to *Introduction to Matplotlib and Line Plots* Chapter for the rational and detailed description of the changes.

1. Clean up the dataset to remove columns that are not informative to us for visualization (e.g., Type, AREA, REG).

```
df_can.drop(['AREA', 'REG', 'DEV', 'Type', 'Coverage'],  
axis=1, inplace=True)
```

```
# let's view the first five elements and see how the  
dataframe was changed  
df_can.head()
```

	OdName	AreaName	RegName	1980	... 2013
0	Afghanistan	Asia	Southern Asia	16	... 2004
1	Albania	Europe	Southern Europe	1	... 603
2	Algeria	Africa	Northern Africa	80	... 4331
3	America Samoa	Oceania	Polynesia	0	... 0
4	Andorra	Europe	Southern Europe	0	... 0

5 rows × 38 columns

Notice how the columns `Type`, `Coverage`, `AREA`, `REG`, and `DEV` got removed from the dataframe.

2. Rename some of the columns so that they make sense.

```
df_can.rename(columns={'OdName': 'Country',  
'AreaName': 'Continent', 'RegName': 'Region'}, inplace=True)  
# Let's view the first five elements and see how the  
dataframe was changed  
df_can.head()
```

Country	Continent	Region	1980	... 2013
0 Afghanistan	Asia	Southern Asia	16	... 2004
1 Albania	Europe	Southern Europe	1	... 603
2 Algeria	Africa	Northern Africa	80	... 4331
3 America Samoa	Oceania	Polynesia	0	... 0
4 Andorra	Europe	Southern Europe	0	... 0

5 rows × 38 columns

4. For consistency, ensure that all column labels of type string.

Let's examine the types of the column labels

```
all(isinstance(column, str) for column in df_can.columns)
False
```

Notice how the above line of code returned *False* when we tested if all the column labels are of type **string**. So, let's change them all to **string** type.

```
df_can.columns = list(map(str, df_can.columns))
# Let's check the column labels types now
all(isinstance(column, str) for column in df_can.columns)
True
```

Country	Continent	Region	1980	... 2013
0 Afghanistan	Asia	Southern Asia	16	... 2004
1 Albania	Europe	Southern Europe	1	... 603
2 Algeria	Africa	Northern Africa	80	... 4331
3 America Samoa	Oceania	Polynesia	0	... 0
4 Andorra	Europe	Southern Europe	0	... 0

5 rows × 37 columns

Notice now the country names now serve as indices.

5. Add total column.

```
df_can['Total'] = df_can.sum(axis=1)
# Let's view the first five elements and see how the
# dataframe was changed
df_can.head()
```

Country	Continent	Region	1980	...	Total
0 Afghanistan	Asia	Southern Asia	16	...	58639
1 Albania	Europe	Southern Europe	1	...	15699
2 Algeria	Africa	Northern Africa	80	...	69439
3 America Samoa	Oceania	Polynesia	0	...	6
4 Andorra	Europe	Southern Europe	0	...	15

5 rows × 38 columns

Now the dataframe has an extra column that presents the total number of immigrants from each country in the dataset from 1980 - 2013. So, if we print the dimension of the data, we get:

```
print('data dimensions:', df_can.shape)
data dimensions: (195, 38)
```

So now our dataframe has 38 columns instead of 37 columns that we had before.

Finally, let's create a list of years from 1980 – 2013. This will come in handy when we start plotting the data.

```
years = list(map(str, range(1980, 2014)))
years
['1980', '1981', '1982', '1983', '1984', '1985', '1986', '1987', '1988', '1989', '1990', '1991', '1992', '1993', '1994', '1995', '1996', '1997', '1998', '1999', '2000', '2001', '2002', '2003', '2004', '2005', '2006', '2007', '2008', '2009', '2010', '2011', '2012', '2013']
```

Visualizing Data using Matplotlib

First, let's import the matplotlib library. We'll use the inline backend to generate the plots within the browser.

```
%matplotlib inline

import matplotlib as mpl
import matplotlib.pyplot as plt

mpl.style.use('ggplot') # optional: for ggplot-like style

# Check for latest version of Matplotlib
print('Matplotlib version: ', mpl.__version__) # >= 2.0.0
```

Matplotlib version: 3.1.1

Area Plots

In the last module, we created a line plot that visualized the top 5 countries that contributed the most immigrants to Canada from 1980 to 2013. With a little modification to the code, we can visualize this plot as a cumulative plot, also known as a **Stacked Line Plot** or **Area plot**.

```
df_can.sort_values(['Total'], ascending=False, axis=0,
                  inplace=True)

# Get the top 5 entries
df_top5 = df_can.head()

# Transpose the dataframe
df_top5 = df_top5[years].transpose()

df_top5.head()
```

Country	United Kingdom of Great Britain and Northern Ireland					Philippines	Pakistan
	India	China		Northern Ireland			
1980	8880	5123		22045		6051	978
1981	8670	6682		24796		5921	972
1982	8147	3308		20620		5249	1201
1983	7338	1863		10015		4562	900
1984	5704	1527		10170		3801	668

An area chart or area graph displays graphically quantitative data. It is based on the line chart. The area between axis and line are commonly emphasized with colors, textures and hatchings. Area plots are stacked by default. And to produce a stacked area plot, each column must be either all positive or all negative values (any `NaN`, i.e., not-a-number, values will default to 0).

- Show or compare a quantitative progression over time.
- Represent cumulated totals using numbers or percentages over time.
- Visualize part-to-whole relationships, helping show how each category contributes to the cumulative total.
- To show distribution of categories as parts of a whole, where the cumulative total is unimportant.

To produce an unstacked plot, set parameter `stacked` to value `False`. Let's change the index values of `df_top5` to type integer for plotting, as displayed in **Figure 6-1**.

```
plt.rcParams ['figure.dpi'] = 300
df_top5.index = df_top5.index.map(int)
df_top5.plot(kind = 'area',
              stacked = False,
              figsize = (20, 10)) # pass a tuple(x, y) size

# Set the axes title font size
plt.rc('font', size = 18)
plt.rc('axes', titlesize = 28)
plt.rc('axes', labelsize = 24)
plt.rc('xtick', labelsize = 24)
plt.rc('ytick', labelsize = 24)
plt.rc('legend', fontsize = 20)

plt.title('Immigration Trend of Top 5 Countries')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

plt.show()
plt.savefig('MAP_01.png', dpi = 300)
```

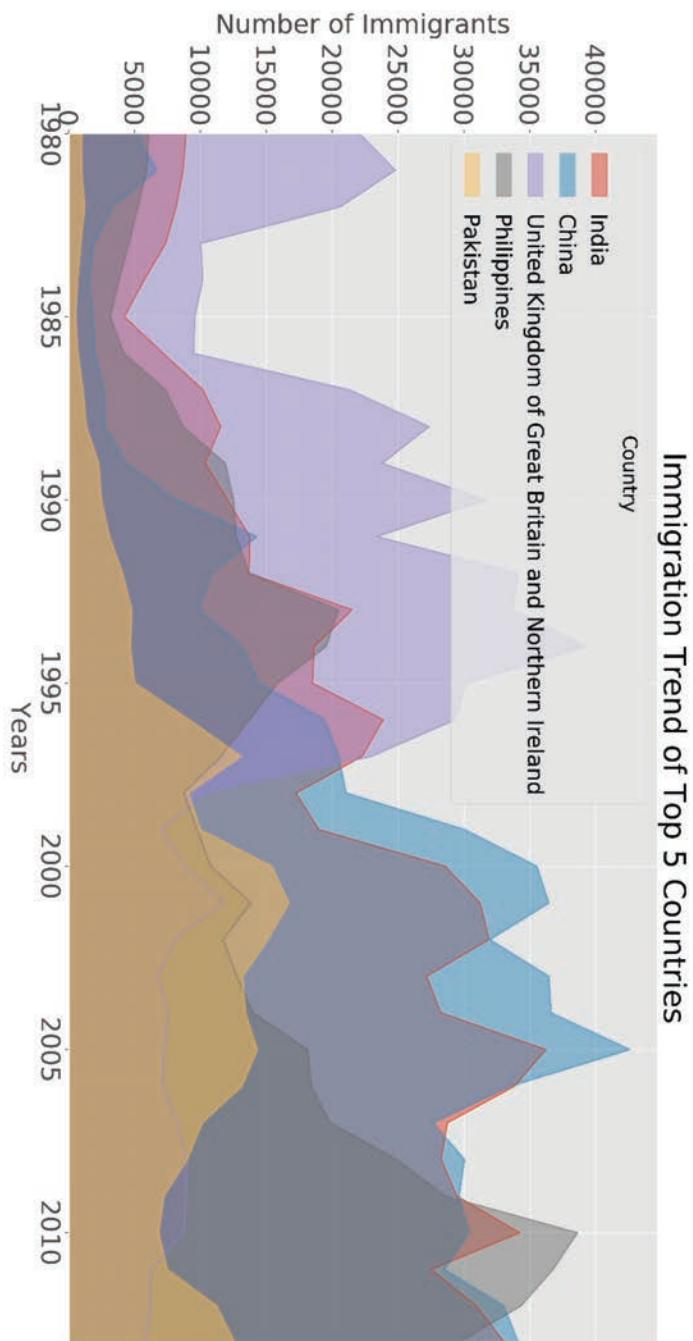


Figure 6-1. An area chart of top-5 countries immigrating to Canada

The unstacked plot has a default transparency (alpha value) at 0.5. We can modify this value by passing in the `alpha` parameter. The main purpose of transparency is to illustrate the areas in the scatterplot with higher densities of data. If we want to make the graph plot more transparent, then we can make alpha less than 0.5, such as 0.25, or to make it less transparent, then we can set alpha greater than 0.5, up to 1. With alpha = 1, it solidifies the graph plot, making it less transparent and thicker and denser, so to speak. The plot defined here is depicted in **Figure 6-2**.

```
plt.rcParams ['figure.dpi'] = 300
df_top5.plot(kind = 'area',
              alpha = 0.25, # 0-1, default value alpha=0.5
              stacked = False,
              figsize = (20, 10))

# Set the axes title font size
plt.rc('font', size = 18)
plt.rc('axes', titlesize = 28)
plt.rc('axes', labelsize = 24)
plt.rc('xtick', labelsize = 24)
plt.rc('ytick', labelsize = 24)
plt.rc('legend', fontsize = 20)

plt.title('Immigration Trend of Top 5 Countries')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

plt.show()
```

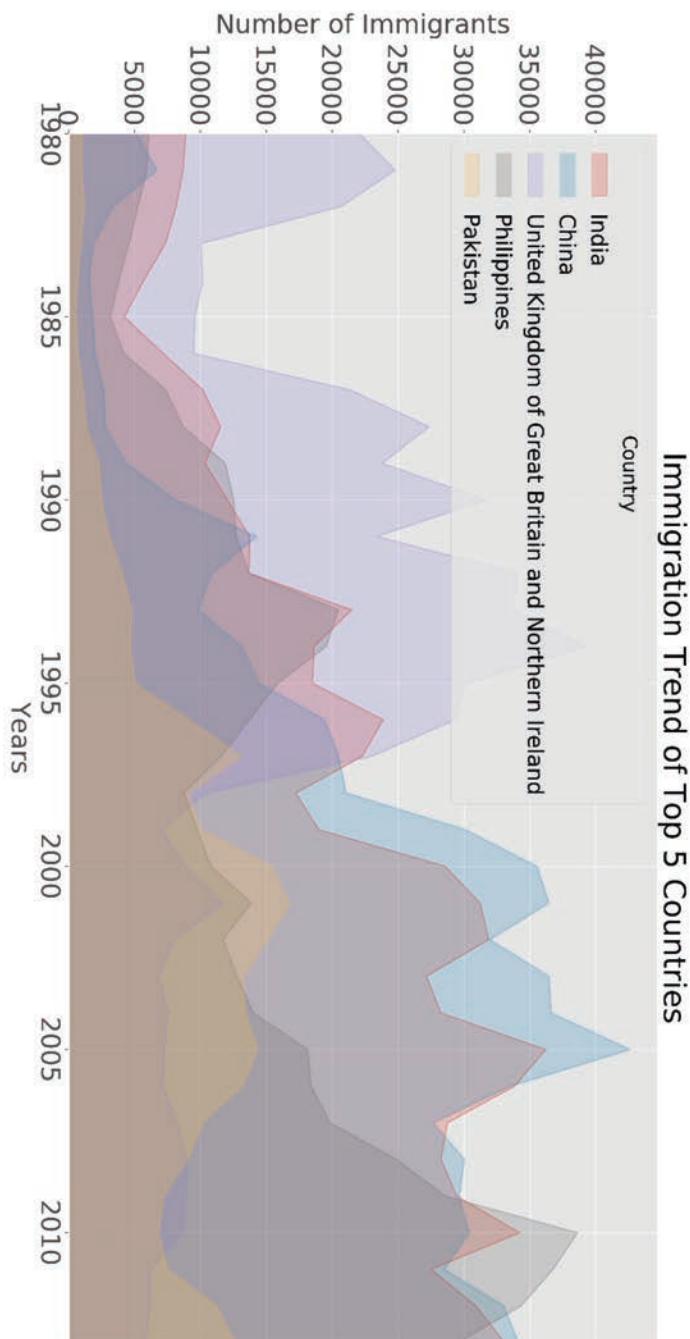


Figure 6-2. Same plot as in **Figure 6-1** except for increased transparency

Two types of plotting

As we discussed in the video lectures, there are two styles/options of plotting with **matplotlib**, plotting using the **Artist** layer and plotting using the scripting layer.

Option 1: The scripting layer (procedural method) – using `matplotlib.pyplot` as `plt`.

You can use `plt` i.e., `matplotlib.pyplot` and add more elements by calling different methods procedurally; for example, `plt.title(...)` to add title or `plt.xlabel(...)` to add label to the x-axis.

```
# Option 1: This is what we have been using so far
df_top5.plot(kind='area', alpha=0.35, figsize=(20, 10))
plt.title('Immigration trend of top 5 countries')
plt.ylabel('Number of immigrants')
plt.xlabel('Years')
```

Option 2: The artist layer (Object oriented method) - using an `Axes` instance from **Matplotlib** (preferred)

You can use an `Axes` instance of your current plot and store it in a variable (e.g., `ax`). You can add more elements by calling methods with a little change in syntax (by adding `set_` to the previous methods). For example, use `ax.set_title()` instead of `plt.title()` to add title, or `ax.set_xlabel()` instead of `plt.xlabel()` to add label to the x-axis.

This option sometimes is more transparent and flexible to use for advanced plots (in particular when having multiple plots, as you will see later).

In this course, we will stick to the **scripting layer**, except for some advanced visualizations where we will need to use the **artist layer** to manipulate advanced aspects of the plots. The plot we define here is shown in .

```
# Option 2: preferred option with more flexibility
plt.rcParams['figure.dpi'] = 300
ax = df_top5.plot(kind='area', alpha=0.35,
                   figsize=(20, 10))
```

```

# Set the axes title font size
plt.rc('font', size = 18)
plt.rc('axes', titlesize = 28)
plt.rc('axes', labelsize = 24)
plt.rc('xtick', labelsize = 24)
plt.rc('ytick', labelsize = 24)
plt.rc('legend', fontsize = 20)
ax.set_title('Immigration Trend of Top 5 Countries')
ax.set_ylabel('Number of Immigrants')
ax.set_xlabel('Years')
plt.savefig('MAP_03.png', bbox_inches='tight', dpi=240)

```

Plotting Parameters

As an aside, we can define plotting parameter for an entire workbook as follows:

```

def set_pub():
    plt.rc('font', weight='bold') # bold fonts
    plt.rc('tick', labelsize=15) # tick Labels bigger
    plt.rc('lines', lw=1, color='k') # thicker black Lines
    plt.rc('grid', c='0.5', ls='-', lw=0.5) # solid gray
                                         grid Lines
    plt.rc('savefig', dpi=300) # higher res outputs
    plt.rc('font', size=18) # default font size
    plt.rc('axes', titlesize=24) # plot title Label size
    plt.rc('axes', labelsize=16) # x and y Label size
    plt.rc('xtick', labelsize=14) # x tick mark Label size
    plt.rc('ytick', labelsize=14) # y tick mark Label size
    plt.rc('legend', fontsize=14) # Legend font size

```

We used these plotting parameters for this notebook, but we are able to either override them using other values (as we did above) when calling `set_pub()`, or return to default values, using `plt.rcdefaults()`.

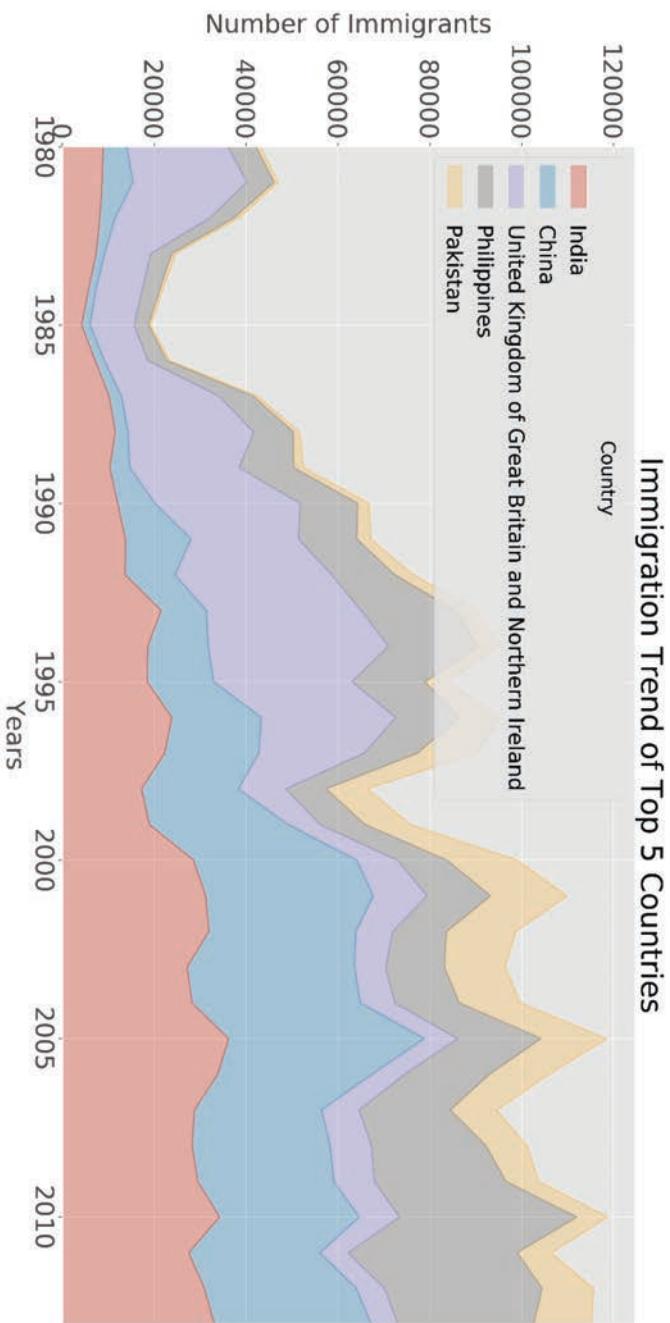


Figure 6-3. An artist layer method for the same plot in [Figure 6-2](#)

Example: Use the scripting layer to create a stacked area plot of the 5 countries that contributed the least to immigration to Canada **from** 1980 to 2013. Use a transparency value of 0.45.

So, now we define the plot and display the result in **Figure 6-4**.

```
# Get the 5 countries with the Least contribution
plt.rcParams ['figure.dpi'] = 300
df_least5 = df_can.tail(5)

# Transpose the dataframe
df_least5 = df_least5[years].transpose()
df_least5.head()

df_least5.index = df_least5.index.map(int) # change the
index values of df_least5 to type integer for plotting
df_least5.plot(kind='area', alpha=0.45, figsize= (20, 10))

# Set the axes title font size
plt.rc('font', size = 18)
plt.rc('axes', titlesize = 28)
plt.rc('axes', labelsize = 24)
plt.rc('xtick', labelsize = 24)
plt.rc('ytick', labelsize = 24)
plt.rc('legend', fontsize = 20)

plt.title('Immigration Trend of 5 Countries with
Least Contribution to Immigration')
plt.ylabel('Number of Immigrants')
plt.xlabel('Years')

plt.show()
plt.savefig('MAP_04.png', bbox_inches='tight', dpi=300)
```

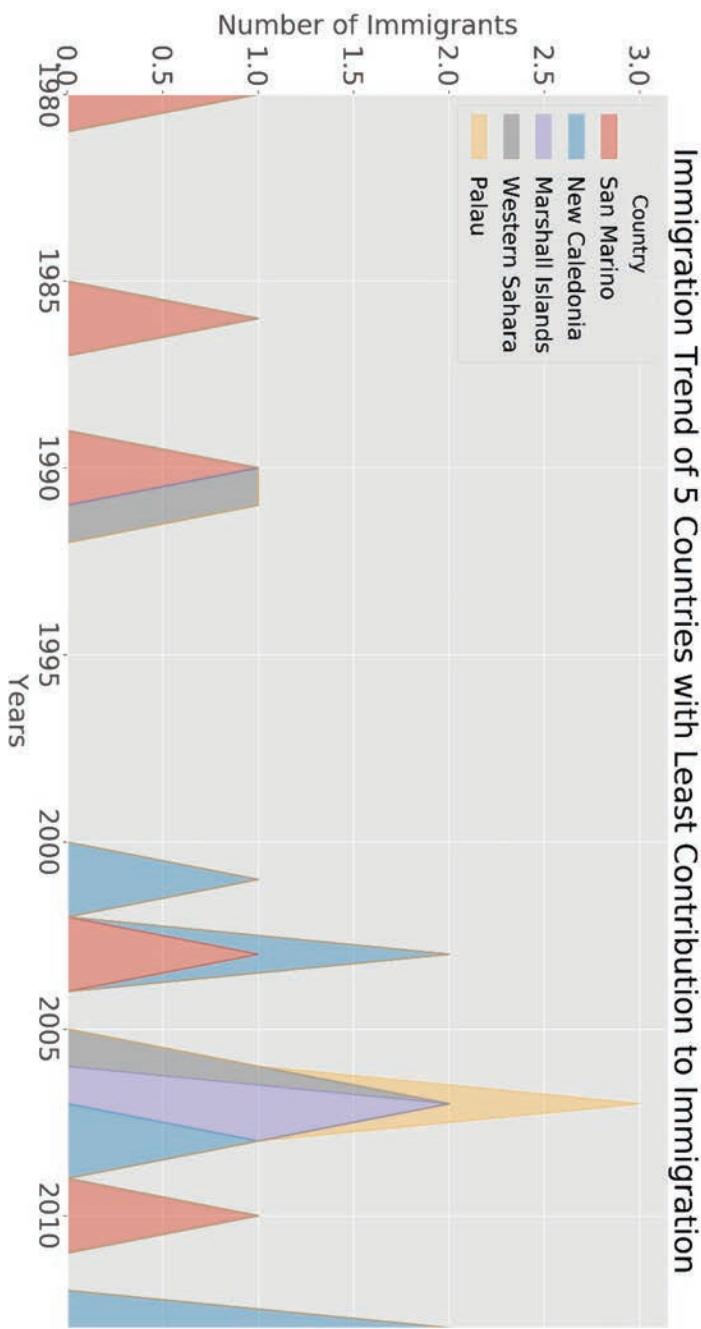


Figure 6-4. Stacked area plot of the countries with the least immigration to Canada

Example: Use the artist layer to create an unstacked area plot of the 5 countries that contributed the least to immigration to Canada **from** 1980 to 2013. Use a transparency value of 0.55.

We define our plot here to be displayed in **Figure 6-5**.

```
# Get the 5 countries with the Least contribution
plt.rcParams ['figure.dpi'] = 300
df_least5 = df_can.tail(5)

# Transpose the dataframe
df_least5 = df_least5[years].transpose()
df_least5.head()
df_least5.index = df_least5.index.map(int) # Let's change
# the index values of df_least5 to type integer for plotting

ax = df_least5.plot(kind = 'area', alpha = 0.55,
                     stacked = False, figsize = (20, 10))

# Set the axes title font size
plt.rc('font', size = 18)
plt.rc('axes', titlesize = 28)
plt.rc('axes', labelsize = 24)
plt.rc('xtick', labelsize = 24)
plt.rc('ytick', labelsize = 24)
plt.rc('legend', fontsize = 20)

ax.set_title('Immigration Trend of 5 Countries with
             Least Contribution to Immigration')
ax.set_ylabel('Number of Immigrants')
ax.set_xlabel('Years')

plt.savefig('MAP_05.png', bbox_inches='tight', dpi=300)
```

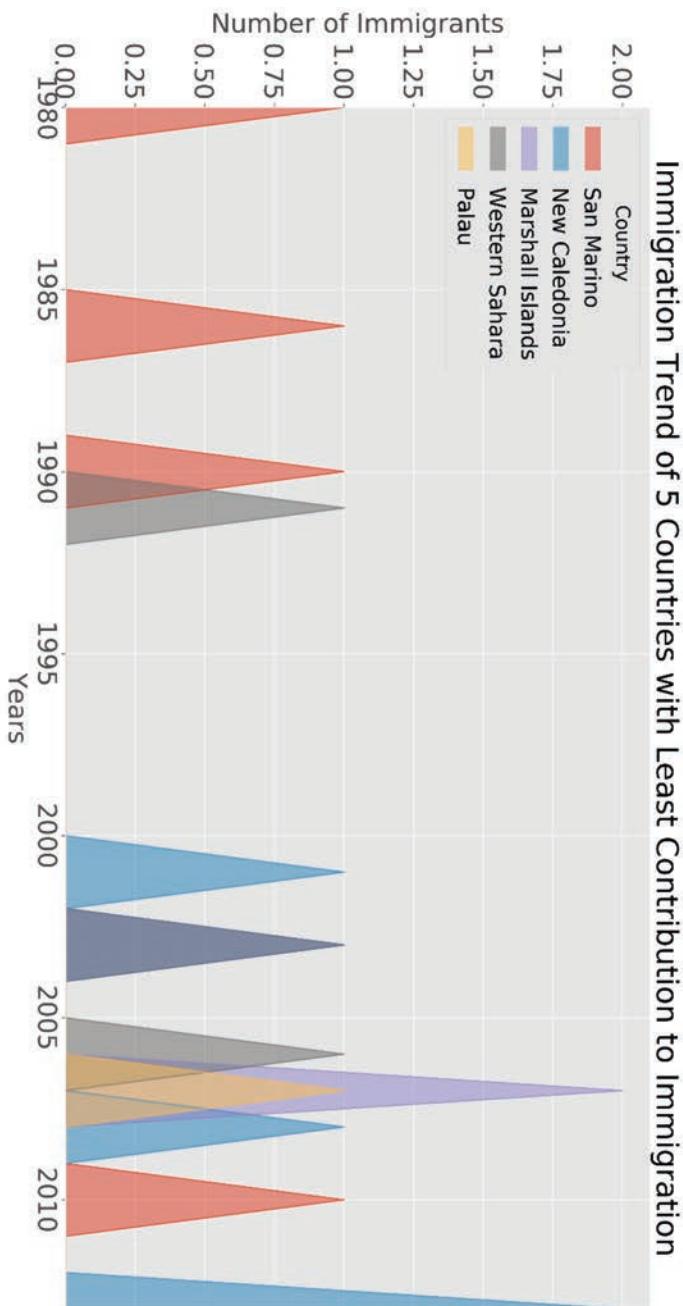


Figure 6-5. An unstacked area plot of countries with the least immigration to Canada

Histograms

A histogram is a way of representing the *frequency* distribution of numeric dataset. The way it works is it partitions the x-axis into *bins*, assigns each data point in our dataset to a bin, and then counts the number of data points that have been assigned to each bin. So, the y-axis is the frequency or the number of data points in each bin. Note that we can change the bin size and usually one needs to tweak it so that the distribution is displayed nicely.

Example: Let's find the frequency distribution of the number (population) of new immigrants from the various countries to Canada in 2013. Before we proceed with creating the histogram plot, let's first examine the data split into intervals. To do this, we will use [Numpy's](#) histogram method to get the bin ranges and frequency counts as follows:

```
# Let's quickly view the 2013 data
df_can['2013'].head()
```

```
Country
India                      33087
China                      34129
United Kingdom of Great Britain and Northern Ireland      5827
Philippines                  29544
Pakistan                     12603
Name: 2013, dtype: int64
```

```
# np.histogram returns 2 values
count, bin_edges = np.histogram(df_can['2013'])
print(count) # frequency count
print(bin_edges) # bin ranges, default = 10 bins
```

```
[178    11     1     2     0     0     0     0     0
 [        0.  3412.9  6825.8 10238.7 13651.6 17064.5 20477.4
   1       2 ]
23890.3 27303.2
30716.1 34129. ]
```

```
from IPython.display import Image
from IPython.core.display import HTML
```

By default, the histogram method breaks up the dataset into 10 bins. **Figure 6-6** summarizes the bin ranges and the frequency distribution of immigration in 2013. We can see that in 2013:

- 178 countries contributed between 0 to 3412.9 immigrants
- 11 countries contributed between 3412.9 to 6825.8 immigrants
- 1 country contributed between 6285.8 to 10238.7 immigrants, and so on...

	Bin 1	Bin 2	Bin 3	Bin 4	Bin 5
Range	0. to 3412.9	3412.9 to 6825.8	6825.8 to 10238.7	10238.7 to 13651.6	13651.6 to 17064.5
Frequency	178	11	1	2	0
	Bin 6	Bin 7	Bin 8	Bin 9	Bin 10
	17064.5 to 20477.4	20477.4 to 23890.3	23890.3 to 27303.2	27303.2 to 30716.1	30716.1 to 34129.
	0	0	0	1	2

Figure 6-6. Frequency distribution of new immigrant to Canada by 10 bins

We can easily graph this distribution by passing `kind='hist'` to `plot()`. So, we define the plot below and present it in **Figure 6-7**.

```
plt.rcParams ['figure.dpi'] = 300
df_can['2013'].plot(kind='hist', figsize=(8, 5))
# Set the axes title font size
plt.rc('font', size = 16)
plt.rc('axes', titlesize = 18)
plt.rc('axes', labelsize = 14)
plt.rc('xtick', labelsize = 12)
plt.rc('ytick', labelsize = 14)
# Add a title to the histogram
plt.title('Histogram of Immigration from 195 Countries in 2013')
# Add y-label
plt.ylabel('Number of Countries')
# Add x-label
plt.xlabel('Number of Immigrants')
plt.show()
```

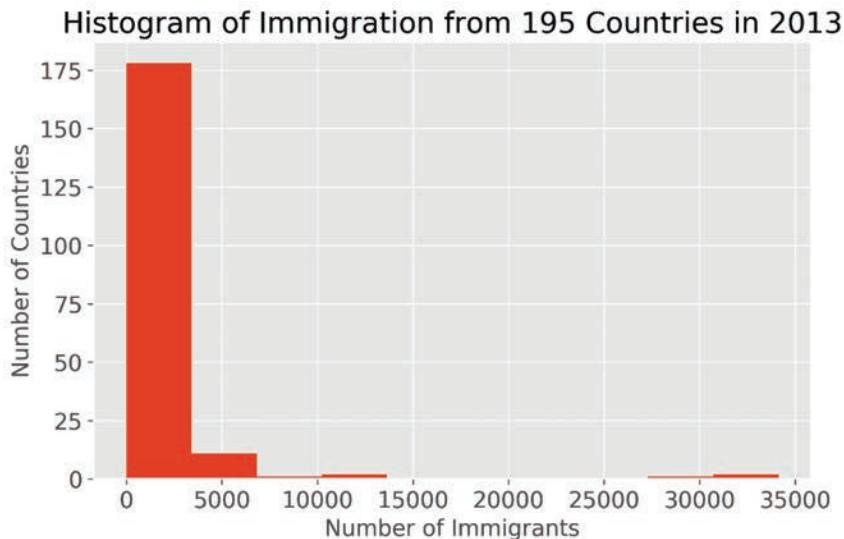


Figure 6-7. Histogram of population range of immigrant by country count.

In the above plot, the x-axis represents the population range of immigrants in intervals of 3412.9. The y-axis represents the number of countries that contributed to the aforementioned population.

Notice that the x-axis labels do not match with the bin size. This can be fixed by passing in a `xticks` keyword that contains the list of the bin sizes, as follows: We show the plot in **Figure 6-8**.

```
# 'bin_edges' is a list of bin intervals
count, bin_edges = np.histogram (df_can['2013'])

df_can['2013'].plot(kind='hist', figsize=(8, 5),
xticks=bin_edges)
# Set the axes title font size
plt.rc('axes', titlesize = 18)
plt.rc('axes', labelsize = 14)
plt.rc('xtick', labelsize = 9)
plt.rc('ytick', labelsize = 14)
plt.title('Histogram of Immigration from 195 countries in
2013') # add a title to the histogram
plt.ylabel('Number of Countries') # add y-label
plt.xlabel('Number of Immigrants') # add x-label
plt.show()
```

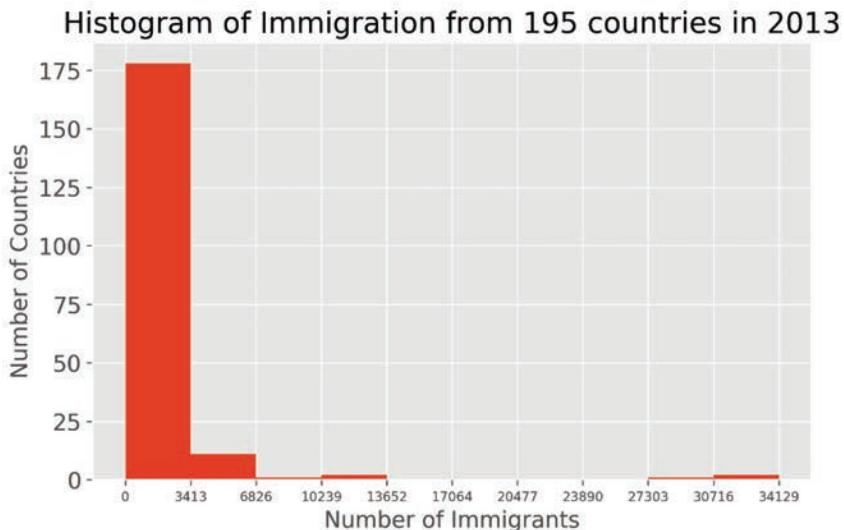


Figure 6-8. The same histogram as in **Figure 6-7** but with corrected x-axis.

We could use `df_can['2013'].plot.hist()`, instead. In fact, throughout this chapter, using `data.plot(kind='type_plot')` is equivalent to `data.plot.type_plot(...)`. That is, passing the type of the plot as argument or method behaves the same. We can also plot multiple histograms on the same plot. For example, let's try to answer the following questions using a histogram.

Example: We'll find the immigration distribution for Denmark, Norway, and Sweden for years 1980 – 2013 and present the plot it in **Figure 6-9**.

```
# Let's quickly view the dataset
df_can.loc[['Denmark', 'Norway', 'Sweden'], years]
```

Country	1980	1981	1982	1983	...	2010	2011	2012	2013
Denmark	272	293	299	106	...	92	93	94	81
Norway	116	77	106	51	...	46	49	53	59
Sweden	281	308	222	176	...	159	134	140	140

3 rows × 34 columns

```
# Generate histogram
df_can.loc[['Denmark', 'Norway', 'Sweden'],
years].plot.hist()
```

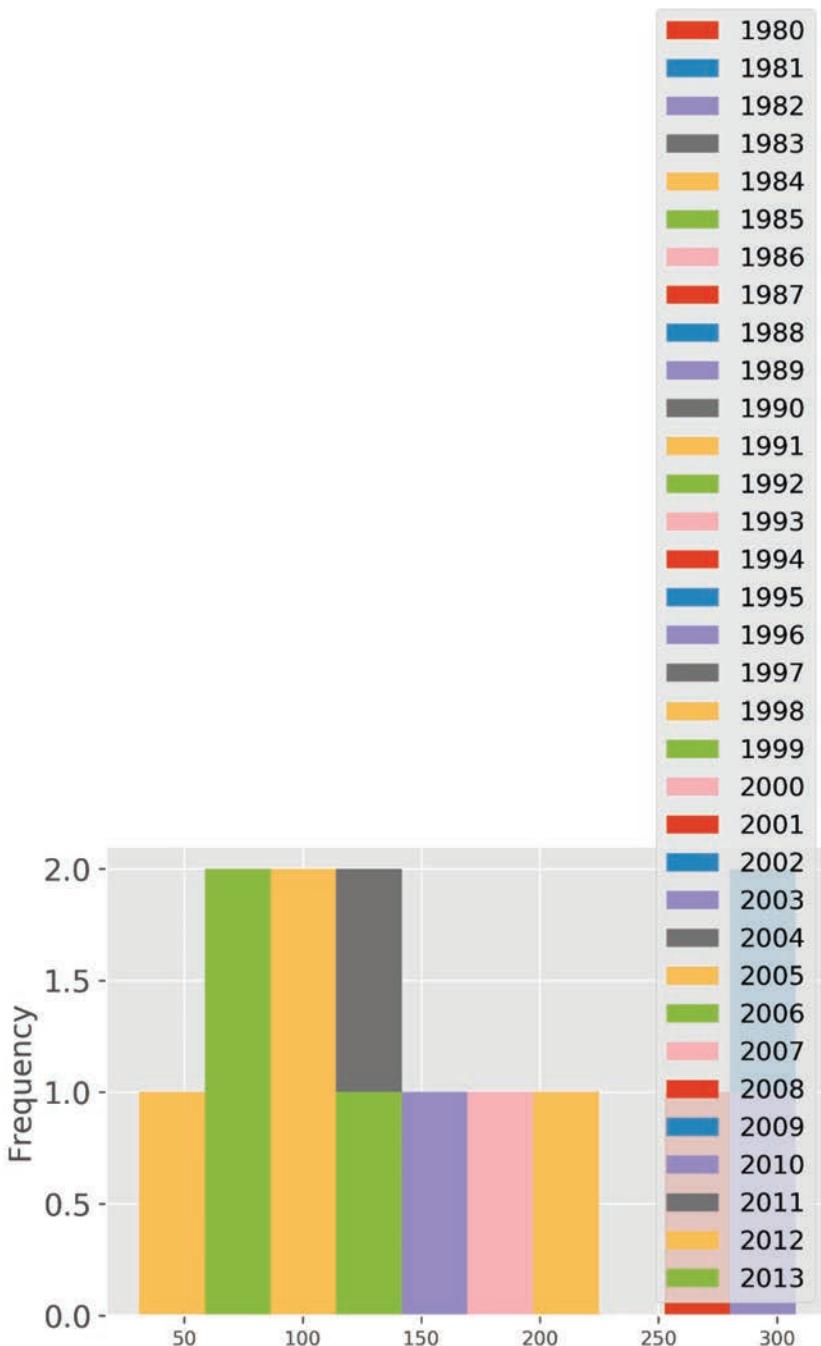


Figure 6-9. Example of how a plot can go wrongs with untransformed data.

That does not look right!

Don't worry, you'll often come across situations like this when creating plots. The solution often lies in how the underlying dataset is structured.

Instead of plotting the population frequency distribution of the population for the 3 countries, *pandas* instead plotted the population frequency distribution for the years. This can be easily fixed by first transposing the dataset, and then plotting as shown below.

```
# Transpose dataframe
df_t = df_can.loc[['Denmark', 'Norway', 'Sweden'],
years].transpose()
df_t.head()
```

Country	Denmark	Norway	Sweden
1980	272	116	281
1981	293	77	308
1982	299	106	222
1983	106	51	176
1984	93	31	128

Here we define the plot and display it in **Figure 6-10**.

```
# Generate histogram
plt.rcParams['figure.dpi'] = 300
df_t.plot(kind='hist', figsize=(12, 8))
# Set the axes title font size
plt.rc('font', size = 16)
plt.rc('axes', titlesize = 16)
#plt.rc('axes', labelsize = 20)
#plt.rc('xtick', labelsize = 14)
#plt.rc('ytick', labelsize = 18)
plt.title('Histogram of Immigration from Denmark, Norway, and Sweden from 1980 - 2013')
plt.ylabel('Number of Years')
plt.xlabel('Number of Immigrants')
plt.show()
```

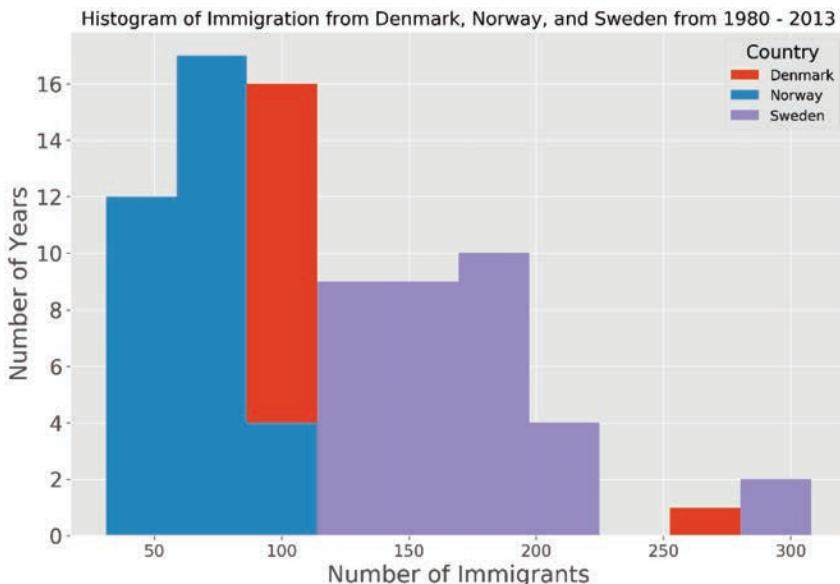


Figure 6-10. Scandinavian countries immigrating to Canada

Let's make a few modifications to improve the impact and aesthetics of the previous plot and show it in **Figure 6-11**:

- increase the bin size to 15 by passing in bins parameter;
- set transparency to 60% by passing in alpha parameter;
- label the x-axis by passing in x-label parameter;
- change the colors of the plots by passing in color parameter.

```
# Let's get the x-tick values
plt.rcParams['figure.dpi'] = 300
count, bin_edges = np.histogram (df_t, 15)

# un-stacked histogram
df_t.plot(kind ='hist',
           figsize=(10, 6),
           bins=15,
           alpha=0.6,
           xticks=bin_edges,
           color=[ 'coral', 'darkslateblue',
                   'mediumseagreen']
         )
```

```

plt.rc('font', size = 16)
plt.rc('axes', titlesize = 16)
plt.rc('axes', labelsize = 16)
plt.rc('xtick', labelsize = 10)
plt.title('Histogram of Immigration from Denmark, Norway,
           and Sweden from 1980 - 2013')
plt.ylabel('Number of Years')
plt.xlabel('Number of Immigrants')

plt.show()

```

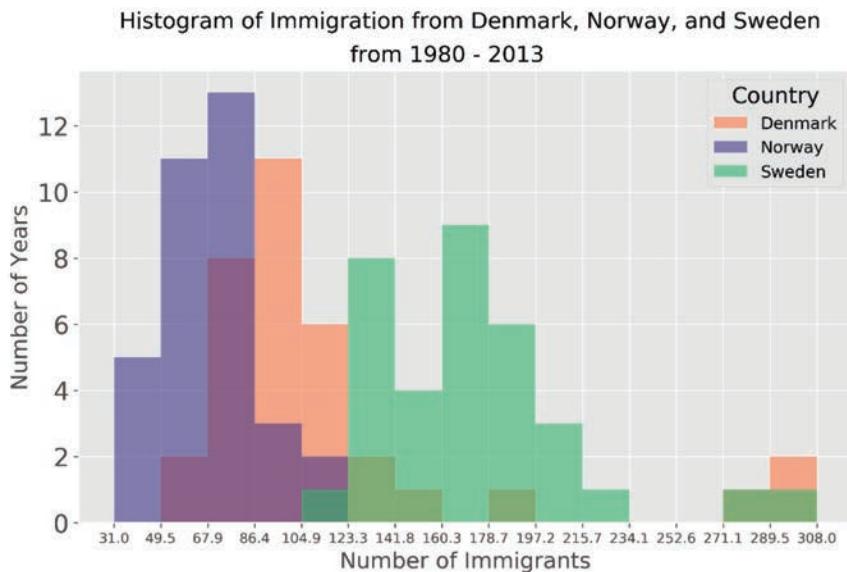


Figure 6-11. Scandinavian-country immigration to Canada with more bins

Tip: For a full listing of colors available in **Matplotlib**, run the following code in your python shell:

```

import matplotlib
for name, hex in matplotlib.colors.cnames.items():
    print(name, hex)

```

If we do not want the plots to overlap each other, we can stack them using the `stacked` parameter. Let's also adjust the min and max x-axis labels to remove the extra gap on the edges of the plot. We can pass a tuple (min, max) using the `xlim` parameter, as shown in **Figure 6-12**.

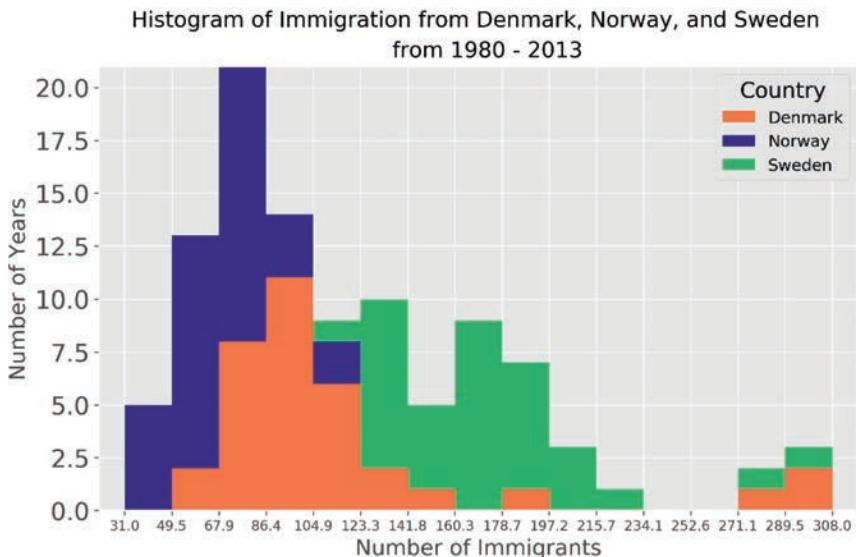


Figure 6-12. Plot of Scandinavian-county immigration to Canada us the stacked parameter

Example: Use the scripting layer to display the immigration distribution for Greece, Albania, and Bulgaria for years 1980 – 2013. Use an overlapping plot with 15 bins and a transparency value of 0.35. We define the plot here and show it in **Figure 6-13**

```
plt.rcParams ['figure.dpi'] = 300
# Create a dataframe of the countries of interest (cof)
df_cof = df_can.loc[['Greece', 'Albania', 'Bulgaria'],
years]
# Transpose the dataframe
df_cof = df_cof.transpose()
# Let's get the x-tick values
count, bin_edges = np.histogram (df_cof, 15)

# Un-stacked Histogram
df_cof.plot(kind ='hist',
            figsize=(10, 6),
            bins=15,
            alpha=0.35,
            xticks=bin_edges,
            color=[ 'coral', 'darkslateblue',
            'mediumseagreen'])
```

```

plt.rc('font', size = 16)
plt.rc('axes', titlesize = 16)
plt.rc('axes', labelsize = 16)
plt.rc('xtick', labelsize = 10)
plt.rc('ytick', labelsize = 14)
plt.rc('legend', fontsize = 16)

plt.title('Histogram of Immigration from Greece, Albania, and Bulgaria from 1980 - 2013')
plt.ylabel('Number of Years')
plt.xlabel('Number of Immigrants')

plt.show()

```

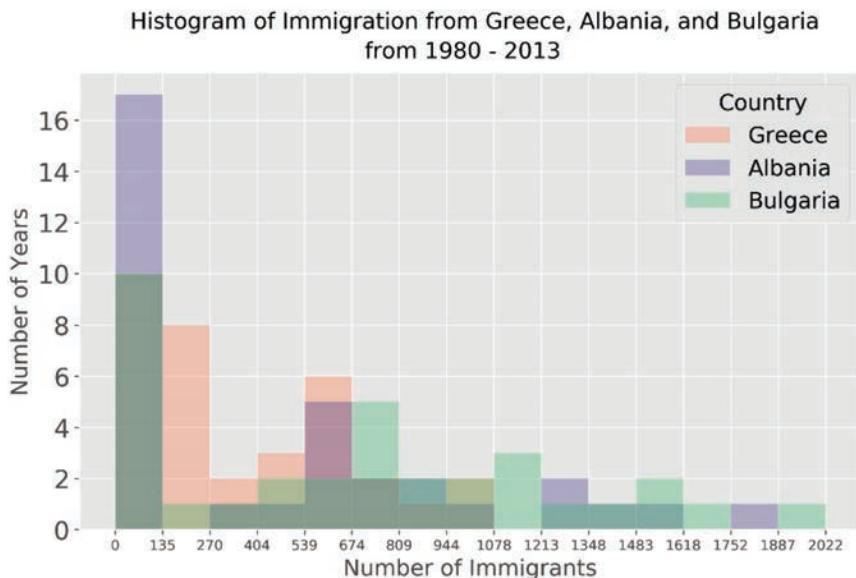


Figure 6-13. Immigration to Canada by Greece, Albania, and Bulgaria

Bar Charts (Dataframe)

A bar plot is a way of representing data where the *length* of the bars represents the magnitude/size of the feature/variable. Bar graphs usually represent numerical and categorical variables grouped in intervals. To create a bar plot, we can pass one of two arguments via kind parameter in `plot()`:

- `kind=bar` creates a *vertical* bar plot
- `kind=barh` creates a *horizontal* bar plot

Vertical bar plot

In vertical bar graphs, the x-axis is used for labelling, and the length of bars on the y-axis corresponds to the magnitude of the variable being measured. Vertical bar graphs are particularly useful in analyzing time series data. One disadvantage is that they lack space for text labelling at the foot of each bar.

Let's start off by analyzing the effect of Iceland's Financial Crisis:

The 2008 - 2011 Icelandic Financial Crisis was a major economic and political event in Iceland. Relative to the size of its economy, Iceland's systemic banking collapse was the largest experienced by any country in economic history. The crisis led to a severe economic depression in 2008 - 2011 and significant political unrest.

Example: Let's compare the number of Icelandic immigrants (`country = Iceland`) to Canada from year 1980 to 2013. We define the plot below and display it in **Figure 6-14**.

Step 1: get the data

```
df_iceland = df_can.loc ['Iceland', years]
df_iceland.head()

1980      17
1981      33
1982      10
1983       9
1984      13
Name: Iceland, dtype: object
```

Step 2: plot data

```
plt.rcParams['figure.dpi'] = 300
df_iceland.plot(kind='bar', figsize= (10, 6))

plt.rc('font', size = 16)
plt.rc('axes', titlesize = 20)
plt.rc('axes', labelsize = 16)
plt.rc('xtick', labelsize = 10)
plt.rc('xtick', labelsize = 14)
```

```

plt.rc('legend', fontsize = 16)
plt.xlabel('Year') # add x-label to the plot
plt.ylabel('Number of immigrants') # add y-label to plot
plt.title('Icelandic immigrants to Canada from 1980 to 2013') # add title to the plot

plt.show()

```

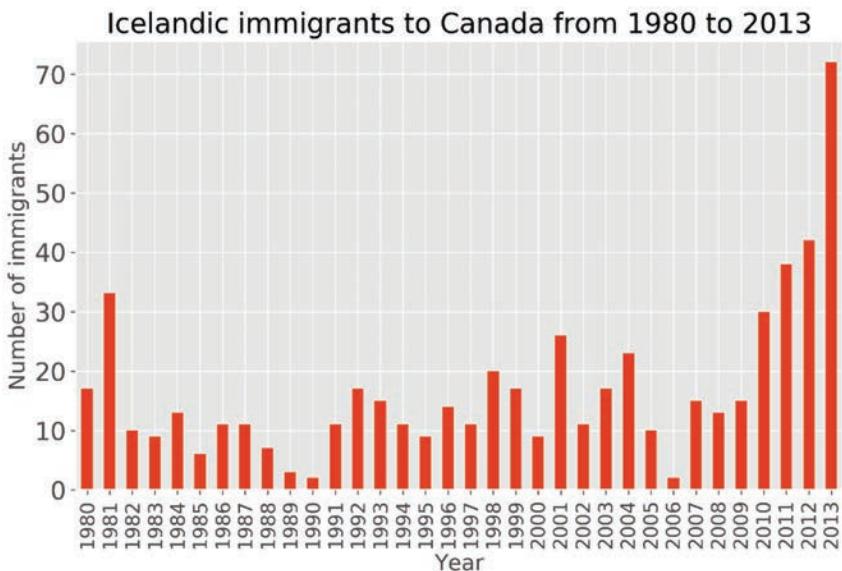


Figure 6-14. Bar chart of Icelandic immigrants to Canada from 1980-2013

The bar plot above shows the total number of immigrants broken down by each year. We can clearly see the impact of the financial crisis; the number of immigrants to Canada started increasing rapidly after 2008. In our next plot, we'll use the `annotate` function to draw an arrow marking the crisis effect, as seen in **Figure 6-15**.

Let's annotate this on the plot using the `annotate` method of the **scripting layer** or the **pyplot interface**. We will pass in the following parameters:

- `s`: str, the text of annotation.
- `xy`: Tuple specifying the (x, y) point to annotate (in this case, end point of arrow).

- **xytext**: Tuple specifying the (x, y) point to place the text (in this case, start point of arrow).
- **xycoords**: The coordinate system that xy is given in – 'data' uses the coordinate system of the object being annotated (default).
- **arrowprops**: Takes a dictionary of properties to draw the arrow:
 - **arrowstyle**: Specifies the arrow style, → , is standard arrow.
 - **connectionstyle**: Specifies the connection type. arc3 is a straight line.
 - **color**: Specifies color of arrow.
 - **lw**: Specifies the line width.

I encourage you to read the **Matplotlib** documentation for more details on annotations:

```
plt.rcParams ['figure.dpi'] = 300

# rotate xticks(labelled points on x-axis) by 90 degrees

df_iceland.plot(kind='bar', figsize=(10, 6), rot=90)

plt.xlabel('Year')
plt.ylabel('Number of Immigrants')
plt.title('Icelandic Immigrants to Canada from 1980 to
           2013')

plt.rc('font', size = 16)
plt.rc('axes', titlesize = 20)
plt.rc('axes', labelsize = 16)
plt.rc('xtick', labelsize = 10)
plt.rc('ytick', labelsize = 14)
plt.rc('legend', fontsize = 16)

# Annotate arrow
plt.annotate('', # s: str. Leaves it blank for no text
             xy=(32, 70), # place head of the arrow at point
                           (year 2012 , pop 70)
```

```

xytext=(28, 20), # place base of the arrow at
                  point (year 2008 , pop 20)
xycoords='data', # will use the coordinate system
                  of the object being annotated
arrowprops=dict(arrowstyle='->',
               connectionstyle='arc3', color='blue', lw=2)
)
plt.show()

```

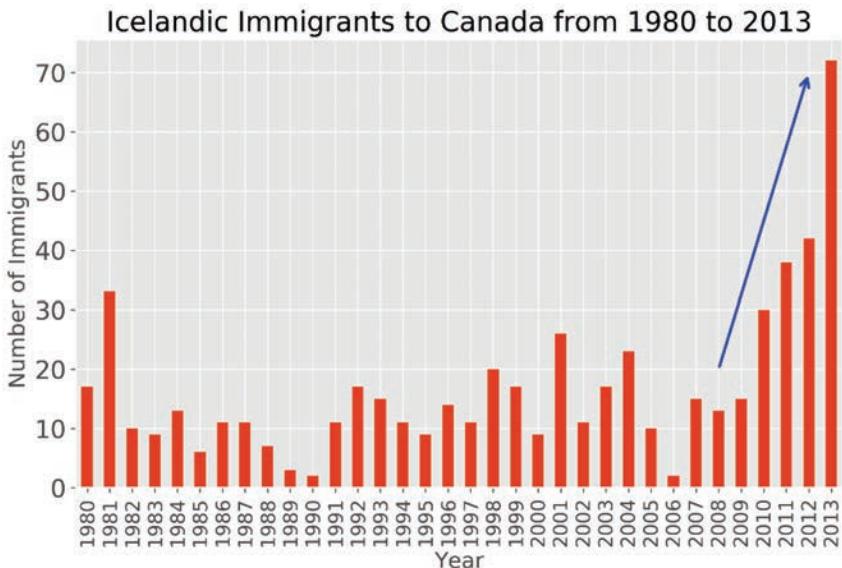


Figure 6-15. Bar chart of Icelandic immigration to Canada annotated with an upward arrow signifying the economic crisis of 2008 and its effect.

Let's also annotate a text to go over the arrow. We will pass in the following additional parameters, and show the plot in **Figure 6-16**:

- **rotation**: rotation angle of text in degrees (counter clockwise)
- **va**: vertical alignment of text ['center' | 'top' | 'bottom' | 'baseline']
- **ha**: horizontal alignment of text ['center' | 'right' | 'left']

```

plt.rcParams['figure.dpi'] = 300
df_iceland.plot(kind='bar', figsize=(10, 6), rot=90)

```

```

plt.xlabel('Year')
plt.ylabel('Number of Immigrants')
plt.title('Icelandic Immigrants to Canada from 1980 to
2013')
plt.rc('font', size = 16)
plt.rc('axes', titlesize = 18)
plt.rc('axes', labelsize = 16)
plt.rc('xtick', labelsize = 10)
plt.rc('xtick', labelsize = 14)
plt.rc('legend', fontsize = 16)

# Annotate arrow
plt.annotate('', # s: str. Leaves it blank for no text
            xy=(32, 70), # head of arrow at (year 2012 , pop 70)
            xytext=(28, 20), # base of arrow at (year 2008 , pop
20)
            xycoords='data', # use the coordinate system of the
object being annotated

            arrowprops=dict(arrowstyle='->',
connectionstyle='arc3', color='blue', lw=2)
            )
# Annotate Text
plt.annotate('2008-2011 Financial Crisis', # display text
            xy=(28, 30), # start the text at at point (year 2008
, pop 30)

            rotation=72.5, # based on trial and error to match
the arrow

            va='bottom', # want the text to be vertically
'bottom' aligned

            ha='left', # want the text to be horizontally 'left'
aligned.
            )

plt.show()

```

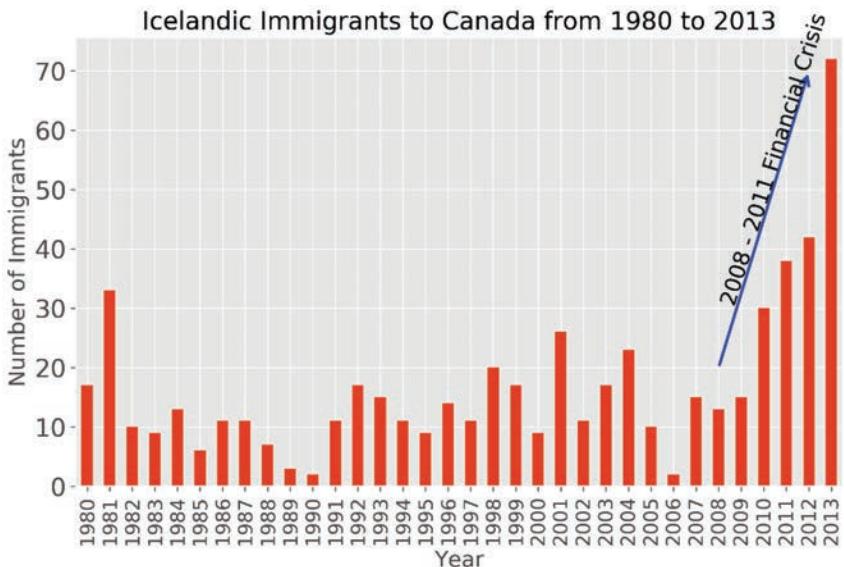


Figure 6-16. Bar chart of Icelandic immigration to Canada annotated with an upward arrow and label showing the economic crisis of 2008's effect.

Horizontal Bar Plot

Sometimes it is more practical to represent the data horizontally, especially if you need more room for labelling the bars. In horizontal bar graphs, the y-axis is used for labelling, and the length of bars on the x-axis corresponds to the magnitude of the variable being measured. As you will see, there is more room on the y-axis to label categorical variables.

Example: Using the scripting layer and the `df_can` dataset, create a *horizontal* bar plot showing the *total* number of immigrants to Canada from the top 15 countries, for the period 1980 – 2013. Label each country with the total immigrant count.

Step 1: Get the data pertaining to the top 15 countries.

```
# sort dataframe on 'Total' column (descending)
df_can.sort_values(by='Total', ascending=True,
inplace=True)
# get top 15 countries
df_top15 = df_can['Total'].tail(15)
df_top15
```

Country	
Romania	93585
Viet Nam	97146
Jamaica	106431
France	109091
Lebanon	115359
Poland	139241
Republic of Korea	142581
Sri Lanka	148358
Iran (Islamic Republic of)	175923
United States of America	241122
Pakistan	241600
Philippines	511391
United Kingdom of Great Britain and Northern Ireland	551500
China	659962
India	691904

Name: Total, dtype: int64

Step 2: Plot data:

1. Use kind = `barh` to generate a bar chart with horizontal bars.
2. Make sure to choose a good size for the plot and to label your axes and to give the plot a title.
3. Loop through the countries and annotate the immigrant population using the `annotate` function of the scripting interface.

Here we define the plot and show it in **Figure 6-17**.

```
# Generate plot
plt.rcParams['figure.dpi'] = 300
df_top15.plot(kind='barh', figsize= (12, 12),
               color='steelblue')
plt.xlabel('Number of Immigrants')
plt.title('Top 15 Countries Contributing to the  
Immigration to Canada between 1980 - 2013')
plt.rc('font', size = 16)
plt.rc('axes', titlesize = 18)
plt.rc('axes', labelsize = 16)
plt.rc('xtick', labelsize = 10)
plt.rc('xtick', labelsize = 14)
plt.rc('legend', fontsize = 16)
```

```

# Annotate value Labels to each country
for index, value in enumerate(df_top15):
    label = format(int(value), ',') # format int with commas

    # Place text at the end of bar (subtracting 47000 from x, and 0.1 from y to make it fit within the bar)
    plt.annotate(label, xy=(value - 47000, index - 0.10),
                 color='magenta')

plt.show()

```

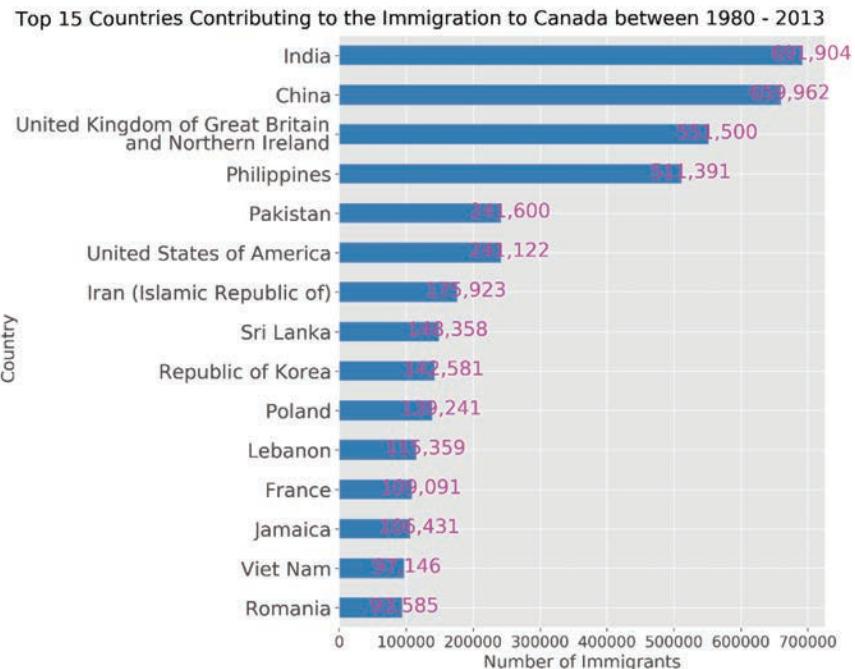


Figure 6-17. A horizontal bar chart of the top-15 countries immigrating to Canada, from 1980 – 2013.

7. Pie Charts, Box Plots, Scatter Plots, and Bubble Plots

Objectives

In this chapter, we will discover how to:

- Further explore our datasets with **pandas** functions
- Further explore our data using additional **Matplotlib** library functions
- Create pie charts, box plots, scatter plots and bubble charts

Exploring Datasets with *pandas* and *Matplotlib*

Toolkits: The course heavily relies on **Pandas** and **Numpy** for data wrangling, analysis, and visualization. The primary plotting library we will explore in the course is **Matplotlib**.

Dataset: Immigration to Canada from 1980 to 2013 - [International migration flows to and from selected countries - The 2015 revision](#) from United Nation's website.

The dataset here is the same we've been using, with annual data on the flows of international migrants as recorded by the countries of destination. Once again, we will focus on the Canadian Immigration data.

Downloading and Prepping Data

Import Primary Modules. The first thing we'll do is import two key data analysis modules: pandas and numpy.

```
import numpy as np # useful for scientific computing
import pandas as pd # primary data structure library
```

Let's download and import our primary Canadian Immigration dataset using **pandas's read_excel ()** method. Normally, before we can do that, we would need to download a module which **pandas** requires reading in *Excel* files. This module was **openpyxl** (formerly

`xlrd`). I pre-installed for this module on *GithHub*, so you would not have to worry about that. Otherwise, you would need to run the following line of code to install the `openpyxl` module:

```
! pip3 install openpyxl
```

Download the dataset and read it into a *pandas* dataframe.

```
df_can = pd.read_excel(  
    'https://github.com/stricje1/jupyter/blob/main/data/Canada  
.xlsx?raw=true', sheet_name='Canada by Citizenship',  
    skiprows=range(20), skipfooter=2)  
print('Data read into a pandas dataframe!')
```

Data downloaded and read into a dataframe!

Let's take a look at the first five items in our dataset.

```
df_can.head()
```

Type	Coverage	REG	RegName	1980	...	2013
0	Immigrants Foreigners	5501	Southern Asia	16	...	2004
1	Immigrants Foreigners	925	Southern Europe	1	...	603
2	Immigrants Foreigners	912	Northern Africa	80	...	4331
3	Immigrants Foreigners	957	Polynesia	0	...	0
4	Immigrants Foreigners	925	Southern Europe	0	...	1

5 rows × 43 columns

We can also view the bottom 5 rows of the dataset using the `tail()` function.

Let's find out how many entries there are in our dataset.

```
# Print the dimensions of the dataframe  
print(df_can.shape)
```

(195, 43)

Clean up Data

We will make some modifications to the original dataset to make it easier to create our visualizations. Refer to the chapter, *Introduction to Matplotlib and Line Plots and Area Plots, Histograms, and Bar Plots* for a detailed description of this preprocessing. So, let's clean up the dataset to remove unnecessary columns (e.g., REG)

```
df_can.drop(['AREA', 'REG', 'DEV', 'Type', 'Coverage'],
            axis=1, inplace=True)
```

Now, let's rename the columns so that they make sense

```
df_can.rename(columns={'OdName':'Country',
                      'AreaName':'Continent','RegName':'Region'}, inplace=True)
```

For sake of consistency, let's also make all column labels of type string

```
df_can.columns = list(map(str, df_can.columns))
```

Now we'll set the country name as index - useful for quickly looking up countries using .loc method and add total column.

```
df_can.set_index('Country', inplace=True)
df_can['Total'] = df_can.sum(axis=1)
```

We'll also setup the years that we will be using in this chapter - useful for plotting later on.

```
years = list(map(str, range(1980, 2014)))
print('data dimensions:', df_can.shape)
data dimensions: (195, 38)
```

Visualizing Data using Matplotlib

Import Matplotlib.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

```
%matplotlib inline

mpl.style.use('ggplot') # optional: for ggplot-like style

# check for latest version of Matplotlib
print('Matplotlib version: ', mpl.__version__) # >= 2.0.0
```

Matplotlib version: 3.1.1

Set Notebook Plotting Parameters

Near the end of Chapter 7, we showed how to define our custom plotting parameters for use throughout all our plotting operations, unless we override them or reset the **Matplotlib** default values.

```
def set_pub():
    # plt.rc('font', weight='bold') # bold fonts
    # plt.rc('tick', labelsize=15) # bigger tick labels
    plt.rcParams ['font.family'] = 'sans-serif'
    plt.rcParams['font.sans-serif'] # available fonts -
        see tip below
    plt.rc('lines', lw=1, color='k') # thicker black lines
    plt.rc('grid', c='0.5', ls='-', lw=0.5) # solid gray
        grid lines
    plt.rc('savefig', dpi=200)      # higher res outputs
    plt.rc('font', size = 18) # default font size
    plt.rc('axes', titlesize = 24) # plot title label size
    plt.rc('axes', labelsize = 18) # x and y label size
    plt.rc('axes', labelcolor='steelblue')
    plt.rc('xtick', labelsize = 18) # x tick label size
    plt.rc('ytick', labelsize = 18) # y tick label size
    plt.rc('legend', fontsize = 12) # legend font size
    plt.rc('axes', axisbelow=True) # tickmarks below axis
    plt.rc('axes', grid=True) # grid if true
    plt.rc('axes', facecolor='whitesmoke') # background
    plt.rc('grid', color='gray') # grid-line color
    plt.rc('grid', linestyle=':') # grid line style
    plt.rc('grid', linewidth=0.5) # grid-line thickness
```

Tip: We can list the fonts in the san-serif family by executing the following cell:

```
plt.rcParams['font.sans-serif']  
['DejaVu Sans',  
 'Bitstream Vera Sans',  
 'Computer Modern Sans Serif',  
 'Lucida Grande',  
 'Verdana',  
 'Geneva',  
 'Lucid',  
 'Arial',  
 'Helvetica',  
 'Avant Garde',  
 'sans-serif']
```

Pie Charts

A pie chart is a circular graphic that displays numeric proportions by dividing a circle (or pie) into proportional slices. You are most likely already familiar with pie charts as it is widely used in business and media. We can create pie charts in Matplotlib by passing in the `kind=pie` keyword.

Let's use a pie chart to explore the proportion (percentage) of new immigrants grouped by continents for the entire time period from 1980 to 2013.

Step 1: Gather data.

We will use `pandas.groupby` method to summarize the immigration data by Continent. The general process of `groupby` involves the following steps and depicted in **Figure 7-1**:

1. **Split:** Splitting the data into groups based on some criteria.
2. **Apply:** Applying a function to each group independently:
`.sum()`, `.count()`, `.mean ()`, `.std()`, `.aggregate()`,
`.apply()`, etc.
3. **Combine:** Combining the results into a data structure.

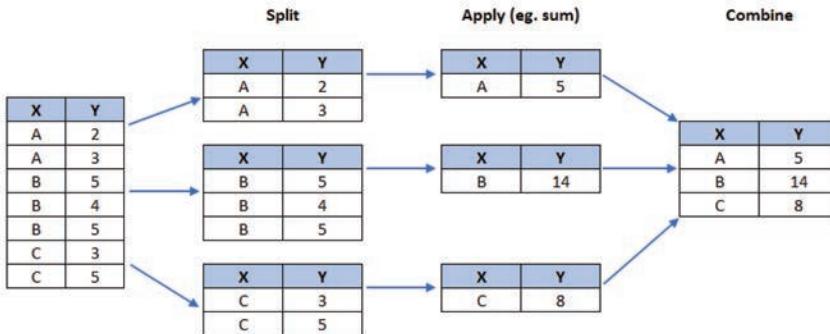


Figure 7-1. This shows the flow of data from the original set, through the split, applying a sum function, for example, and combining the outcome (IBM, 2022).

Here we'll group countries by continents and apply `sum()` function. Note that the output of the `groupby` method is a 'groupby' object. Also, we cannot use the object further until we apply a function, like `sum()`.

```
df_continents = df_can.groupby ('Continent', axis=0).sum()
print(type(df_can.groupby('Continent', axis=0)))
df_continents.head()
```

```
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>
      1980   1981   1982 ...   2012   2013   Total
Continent
Africa    3951   4363   3819 ...  38083  38543  618948
Asia     31025  34314  30214 ... 152218 155075 3317794
Europe    39760  44802  42720 ...  29177  28691 1410947
Latin America and
the Caribbean  13081  15215  16769 ...  27173  24950  765148
Northern America  9378  10030  9074 ...   7892   8503  241142
5 rows × 35 columns
```

Step 2: Plot the data. We will pass in `kind='pie'` keyword, along with the following additional parameters and produce a pie chart in **Figure 7-2**:

- `autopct` - is a string or function used to label the wedges with

their numeric value. The label will be placed inside the wedge. If it is a format string, the label will be `fmt%pct`.

- `startangle` - rotates the start of the pie chart by angle degrees counterclockwise from the x-axis.
- `shadow` - Draws a shadow beneath the pie (to give a 3D feel).

```
# Autopct create %, start angle represent starting point
set_pub()
fig, ax = plt.subplots()
ax1 = fig.add_subplot()
ax = df_continents['Total'].plot(kind='pie',
                                  figsize=(8, 10),
                                  autopct='%1.1f%%', # add in percentages
                                  startangle=90, # start angle 90° (Africa)
                                  shadow=True, # add shadow
                                  ax=ax1)
plt.title('Immigration to Canada by Continent
           [1980 - 2013]')
ax.axis('equal') # Sets the pie chart to look like a
circle.

plt.show()
fig.savefig('myfig20', dpi=300)
```

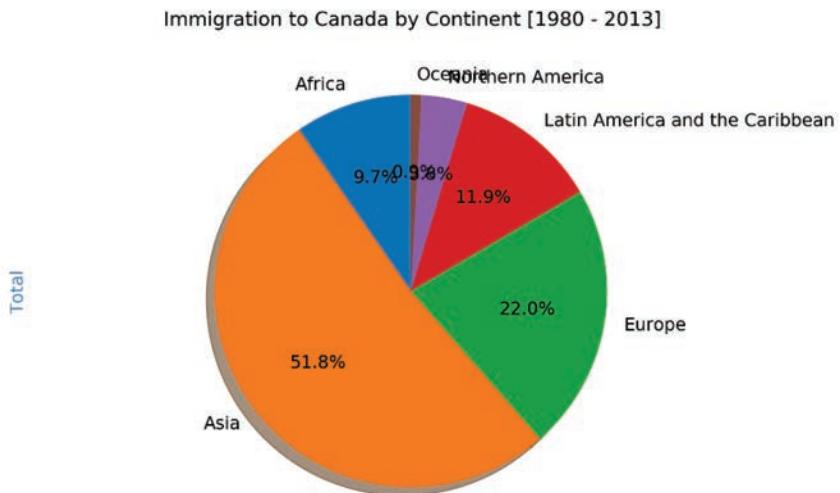


Figure 7-2. A pie chart representing the five countries of interest, but is not the output we want (small fonts, overlapping labels, bad color combinations).

The above visual is not very clear, the numbers and text overlap in some instances. Let's make a few modifications to improve the visuals:

- Remove the text labels on the pie chart by passing in legend and add it as a separate legend using `plt.legend()`.
- Push out the percentages to sit just outside the pie chart by passing in `pctdistance` parameter.
- Pass in a custom set of colors for continents by passing in colors parameter.
- **Explode** the pie chart to emphasize the lowest three continents (Africa, North America, and Latin America and Caribbean) by passing in explode parameter.

```
fig, ax = plt.subplots()

colors_list = ['gold', 'yellowgreen', 'lightcoral',
               'lightskyblue', 'lightgreen', 'pink'] # slice colors
explode_list = [0.1, 0, 0, 0, 0.1, 0.1] # ratio for each
                                         continent with which to offset each wedge.

ax = df_continents['Total'].plot(kind='pie',
                                  figsize= (14, 8),
                                  autopct='%1.1f%%',
                                  startangle=90,
                                  shadow=True,
                                  labels=None,      # turn off labels on pie chart
                                  pctdistance=1.2,   # ratio between the center of
                                         each pie slice and start of the text generated by autopct
                                         colors=colors_list, # add custom colors
                                         explode=explode_list # 'explode' 3 Low % continents
                                         )

# Scale the title up by 12% to match pctdistance
plt.title('Immigration to Canada by Continent [1980 - 2013]', y=1.1)
plt.axis('equal')

plt.rc('font', size = 24) # default font size
```

```

plt.rc('axes', titlesize = 24) # plot title Label size
plt.rc('axes', labelsize = 24) # x and y Label size
plt.rc('legend', fontsize = 16) # legend font size
# Add Legend
plt.legend(labels=df_continents.index, loc='upper left')

plt.show()
fig.savefig('myfig21', dpi=150)

```

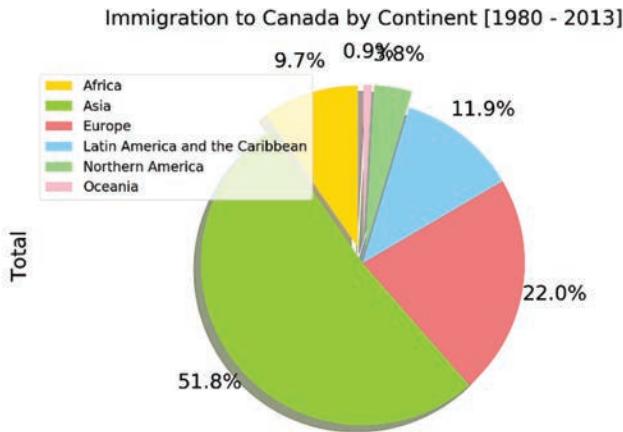


Figure 7-3. Pie with of our countries of interest with exploded slices.

The pie chart shown in **Figure 7-3** is much better in terms of clarity, such as easily distinguishable colors and a legend. However, the values for percentages for Oceania and Northern America are still overlaid. There are two ways that we can fix this: (1) Change the orientation of the slices, i.e., rotate the pie; and (2) offset or explode one of the overlaid slices (actually the % labels for the overlaid slices).

Method 1: Rotate the pie using the `startangle` parameter. It was set to 90° for **Figure 7-3**, do we want to try 15° and see if that makes a difference the code to do this sets `startangle=15`. The output of the code is shown **Figure 7-4**. Another parameter we changed is the resolution of the saved PNG image file. It was previously set to 150 and we changed to `dpi=200` (recall that `dpi` is dots-per-inch). We do this for printing purposes. If you are showing a chart in a PowerPoint

presentation online, for example, then 150 dpi is fine, but where we want print quality output, like you see in the book.

```
fig, ax = plt.subplots()

colors_list = ['gold', 'yellowgreen', 'lightcoral',
'lightskyblue', 'lightgreen', 'pink']
explode_list = [0.1, 0, 0, 0, 0.1, 0.1] # ratio for each
continent with which to offset each wedge

ax = df_continents['Total'].plot(kind='pie',
    figsize=(10, 8),
    autopct='%1.1f%%',
    startangle=15,
    shadow=True,
    labels=None, # turn off labels on pie chart
    pctdistance=1.2, # ratio between the center of
each pie slice and start of the text generated by autopct
    colors=colors_list, # add custom colors
    explode=explode_list # 'explode' 3 low % continents
)

plt.rc('font', size = 24) # default font size
plt.rc('axes', titlesize = 24) # plot title label size
plt.rc('axes', labelsize = 0) # x and y label size
plt.rc('legend', fontsize = 18) # legend font size
plt.axis('equal')
ax.xaxis.set_label_position('top')
ax.set_xlabel('')
ax.set_title('Immigration to Canada by Continent [1980 - 2013]')

# Add Legend
plt.legend(labels=df_continents.index,
    fancybox=True, # rounded corners, for example
    framealpha=0.7, # transparency setting
    bbox_to_anchor= (0.50, 0.45),) # position setting

plt.show()
fig.savefig('myfig22', dpi=200)
```

The output in **Figure 7-4** looks pretty good. Another parameter we

experimented with is the legend, by changing its location using values relative to the center of the pie, instead of ‘upper left’ or ‘lower right’. We also changed the transparency, making it less transparent, by increasing the value of `framealpha`, which we introduced in the previous chapter.

We also changed the figure size, making the one in **Figure 7-5** much smaller than one in **Figure 7-4**, which was 14×8 . That may seem counter-intuitive, but with **Figure 7-5** we filled more white-space by increasing the resolution and setting the pie in a smaller space, thus minimizing the white-space while maximizing the “pie-space.” And, that easier than making apple pie.

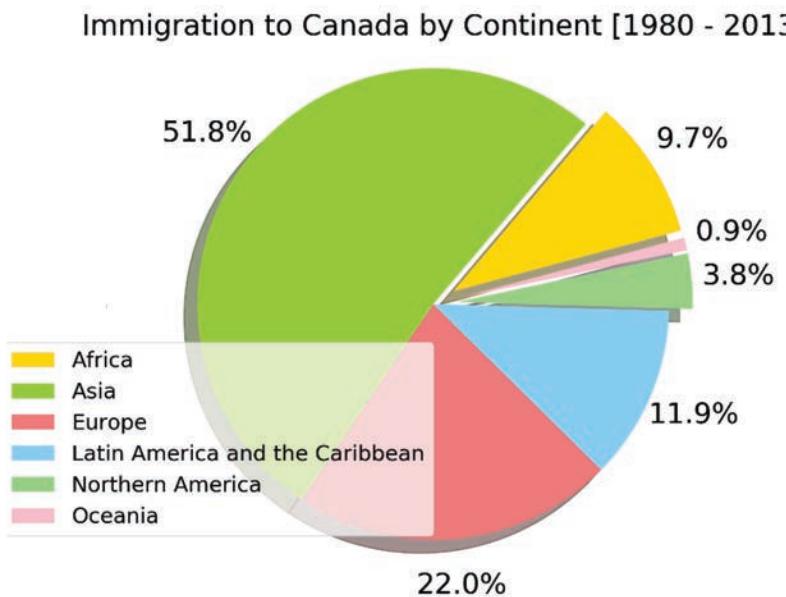


Figure 7-4. Pie chart of our countries of interest, with explode slices and rotated clockwise from a 95° startangle to a 15° startangle.

Method 2: Add more offset to tone of the troublesome exploded slices. Let’s use the pie chart from **Figure 7-2** as a basis for the new pie chart.

You might need to play with the parameter values in order to fix any

overlapping slice values. So, let's see what the code that produces **Figure 7-5** is doing. First, we'll look again at the `explode_list` function. Once again, the values indicate that we offset (exploded) the Oceania slice more than the others, from the pie. We may need to manipulate this a little more because the percentage for Oceania and North America are still close together and changing the font size (if we needed to do that) could cause overlap.

Another this to notice if we compare this pie chart in **Figure 7-5** with **Figure 7-2** has the color scheme are the same but exploding Oceania makes the brown color most distinguishable. Moreover, we use the same startangle, which was 90° .

The last thing we want to point out in **Figure 7-5** is the location of the legend, using `loc='best'`, which allows the algorithm to choose the best location for the legend.

```
fig, ax = plt.subplots()
    plt.rcParams.update({'font.family': 'Tahoma'})
    explode_list = [0.0, 0, 0, 0.1, 0.1, 0.2] # ratios for
        each continent with which to offset each wedge.

    df_continents['2013'].plot(kind='pie', # the plot
        figsize=(15, 8),
        autopct='%1.1f%%',
        startangle=90,
        shadow=True,
        labels=None,          # turn off labels on pie chart
        pctdistance=1.16,    # position of text label
        explode=explode_list # explode 3 low continents
    )

plt.title('Immigration to Canada by Continent in 2013',
    y=1.075) # Title label and position
plt.rc('font', size = 20) # default font size
plt.rc('axes', titlesize = 24) # plot title label size
plt.rc('axes', labelsize = 0) # x and y label size

# Add Legend
plt.legend(labels=df_continents.index, loc='best')
plt.rc('legend', fontsize = 16) # Legend font size
```

```
# show plot
plt.show()
fig.savefig('myfig23', dpi=150)
```

Now, that makes a nice pie chart than our **Figure 7-2** chart. Another feature to notice is the tile font is not only larger, it is a different font, namely Tahoma. We did this using a font update:

```
plt.rcParams.update({'font.family': 'Tahoma'})
```

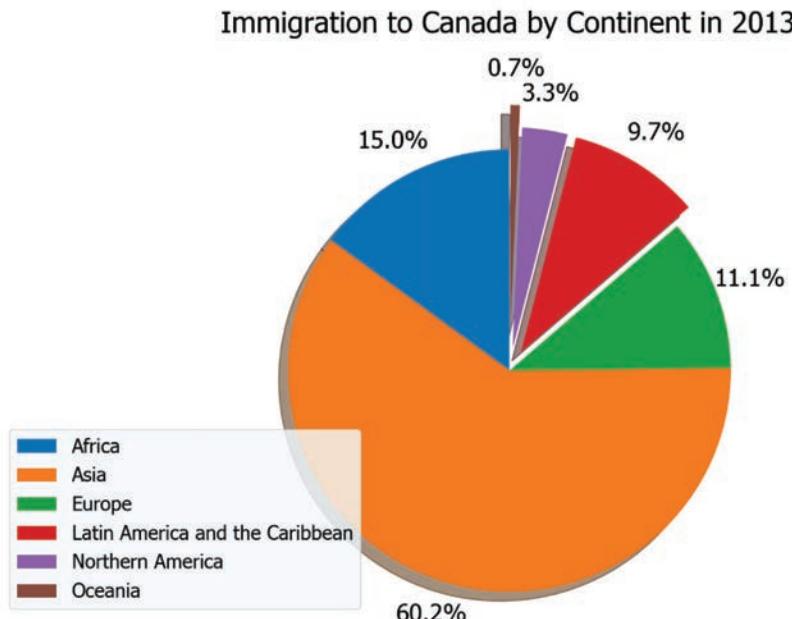


Figure 7-5. Pie chart as in **Figure 7-2** but with offset exploded slices, a legend, and a higher resolution image.

Box Plots

A box plot is a way of statistically representing the distribution of the data through five main dimensions below and as shown in **Figure 7-6**:

- **Minimum:** The smallest number in the dataset excluding the outliers.
- **First quartile:** Middle number between the minimum and the median.

- **Second quartile (Median):** Middle number of the (sorted) dataset.
- **Third quartile:** Middle number between median and maximum.
- **Maximum:** The largest number in the dataset excluding the outliers.

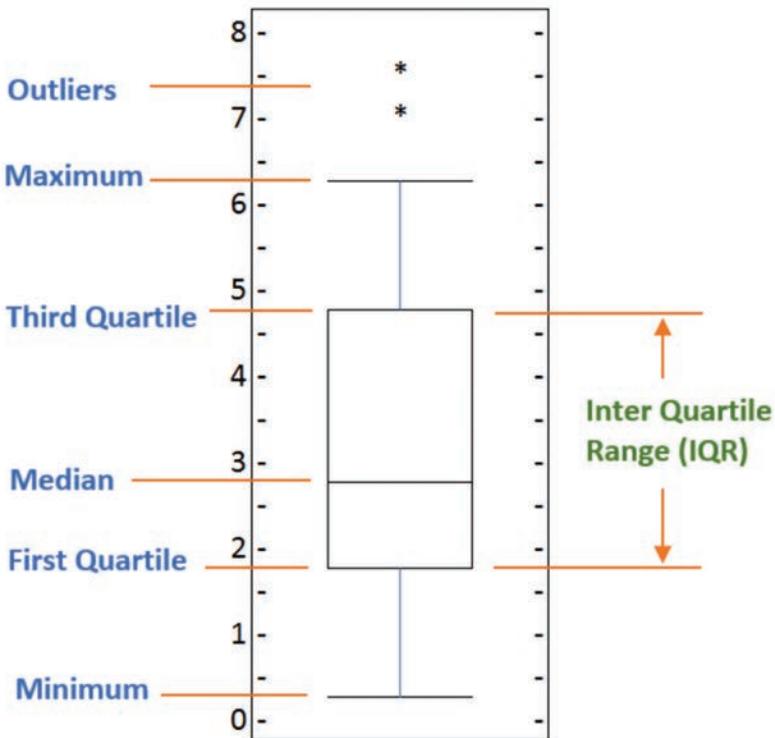


Figure 7-6. Box plot diagram showing its features we respective labels (IBM, 2022).

To make a boxplot, we can use `kind=box` in `plot` method invoked on a `pandas` series or dataframe.

Example: Let's plot the box plot for the Japanese immigrants between 1980 - 2013.

Step 1: Get the subset of the dataset. Even though we are extracting the data for just one country, we will obtain it as a dataframe. This will help us with calling the `dataframe.describe()` method to

view the percentiles. To get a dataframe, we place extra square brackets around '[Japan](#)'.

```
df_japan = df_can.loc [[ 'Japan'], years].transpose()  
df_japan.head()
```

Country	Japan
1980	701
1981	756
1982	598
1983	309
1984	246

Step 2: We now generate the plot by passing in `kind='box'`, and we show the plot in **Error! Reference source not found..**

```
set_pub()  
  
fig, ax = plt.subplots()  
df_japan.plot(kind='box', figsize= (8, 6))  
  
plt.title('Box plot of Japanese Immigrants from  
1980-2013')  
plt.ylabel('Number of Immigrants')  
  
plt.show()  
fig.savefig('myfig24', dpi=120)
```

Something must have gone wrong. We seem to have a “blank” plot, or do we? What we really have is an empty figure and a plot, but the plot did not get set into the figure. In the artists layer, we generally have three layers:

the `matplotlib.backend_bases.FigureCanvas` is the area onto which the figure is drawn

1. the `matplotlib.backend_bases.Renderer` is the object which knows how to draw on the `Figure Canvas`
2. and the `matplotlib.artist.Artist` is the object that knows how to use a renderer to paint onto the canvas.

The `FigureCanvas` and `Renderer` handle all the details of talking to

user interface toolkits like [wxPython](#) or drawing languages like PostScript®, and the Artist handles all the high-level constructs like representing and laying out the figure, text, and lines. The typical user will spend 95% of their time working with the Artists.

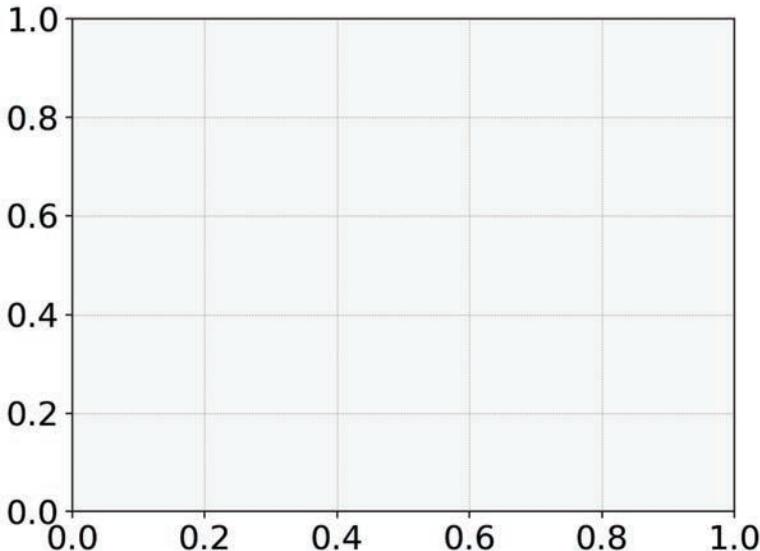


Figure 7-7. A box plot with no output presents could be a problem

There are two types of Artists:

1. **Primitives.** The primitives represent the standard graphical objects we want to paint onto our canvas: [Line2D](#), [Rectangle](#), [Text](#), [AxesImage](#), etc., and
2. **Containers.** The containers are places to put them ([Axis](#), [Axes](#) and [Figure](#)). The standard use is to create a [Figure](#) instance, use the [Figure](#) to create one or more [Axes](#) or [Subplot](#) instances, and use the [Axes](#) instance helper methods to create the primitives.

In the example below, we create a [Figure](#) instance using `matplotlib.pyplot.figure()`, which is a convenience method for instantiating Figure instances and connecting them with your user interface or drawing toolkit [FigureCanvas](#). As we will discuss below, this is not necessary -- you can work directly with PostScript,

PDF Gtk+, or wxPython `FigureCanvas` instances, instantiate your `Figures` directly and connect them yourselves—but since we are focusing here on the **Artist API**, we'll let `pyplot` handle some of those details for us:

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(2, 1, 1) # two rows, one column,
# first plot
```

The `Axes` is probably the most important class in the Matplotlib API, and the one you will be working with most of the time. This is because the `Axes` is the plotting area into which most of the objects go, and the `Axes` has many special helper methods (`plot()`, `text()`, `hist()`, `imshow()`) to create the most common graphics primitives (`Line2D`, `Text`, `Rectangle`, `AxesImage`, respectively). These helper methods will take your data (e.g., numpy arrays and strings) and create primitive `Artist` instances as needed (e.g., `Line2D`), add them to the relevant containers, and draw them when requested. Most of you are probably familiar with the `Subplot`, which is just a special case of an `Axes` that lives on a regular row by columns grid of `Subplot` instances. If you want to create an `Axes` at an arbitrary location, simply use the `add_axes()` method which takes a list of `[left, bottom, width, height]` values in 0-1 relative figure coordinates:

```
fig2 = plt.figure()
ax2 = fig2.add_axes([0.15, 0.1, 0.7, 0.3])
```

Continuing with our example:

```
import numpy as np
t = np.arange (0.0, 1.0, 0.01)
s = np.sin(2*np.pi*t)
line, = ax.plot(t, s, color='blue', lw=2)
```

In this example, `ax` is the `Axes` instance created by the `fig.add_subplot` call above (remember `Subplot` is just a subclass of `Axes`) and when you call `ax.plot`, it creates a `Line2D` instance and adds it to the `Axes.lines` list. In the interactive `Jupyter` session below, you can see that the `Axes.lines` list is length one and

contains the same line that was returned by the `line, = ax.plot...` call:

```
ax.lines[0]
```

```
<matplotlib.lines.Line2D at 0x19a95710>
```

```
line
```

If you make subsequent calls to `ax.plot` (and the hold state is "on" which is the default) then additional lines will be added to the list. You can remove lines later simply by calling the list methods; either of these will work:

```
del ax.lines[0]
ax.lines.remove(line) # one or the other, not both!
```

The `Axes` also has helper methods to configure and decorate the x-axis and y-axis tick, tick labels and axis labels:

```
xtext = ax.set_xlabel('my xdata') # returns a Text
instance
ytext = ax.set_ylabel('my ydata')
```

When you call `ax.set_xlabel`, it passes the information on the Text instance of the `XAxis`. Each `Axes` instance contains an `XAxis` and a `YAxis` instance, which handle the layout and drawing of the ticks, tick labels and axis labels.

Customizing your objects

Every element in the figure is represented by an **Artist object**, and each has an extensive list of properties to configure its appearance. The figure itself contains a `Rectangle` exactly the size of the figure, which you can use to set the background color and transparency of the figures. Likewise, each Axes bounding box (the standard white box with black edges in the typical Matplotlib plot, has a Rectangle instance that determines the color, transparency, and other properties of the Axes. These instances are stored as **member variables** `Figure.patch` and `Axes.patch` ("Patch" is a name inherited from MATLAB, and is a 2D "patch" of color on the figure, e.g., rectangles, circles and polygons). Every Matplotlib `Artist` has the following properties

Property	Description
alpha	The transparency - a scalar from 0-1
animated	A Boolean that is used to facilitate animated drawing
axes	The Axes that the Artist lives in, possibly None
clip_box	The bounding box that clips the Artist
clip_on	Whether clipping is enabled
clip_path	The path the artist is clipped to
contains	A picking function to test whether the artist contains the pick point
figure	The figure instance the artist lives in, possibly None
label	A text label (e.g., for auto-labeling)
picker	A python object that controls object picking
transform	The transformation
visible	A Boolean whether the artist should be drawn
zorder	A number which determines the drawing order
rasterized	Boolean; Turns vectors into raster graphics (for compression & EPS transparency)

Object containers

Now that we know how to inspect and set the properties of a given object we want to configure, we need to know how to get at that object. As mentioned in the introduction, there are two kinds of objects: primitives and containers. The primitives are usually the things you want to configure (the font of a `Text` instance, the width of a `Line2D`) although the containers also have some properties as well -- for example the Axes Artist is a container that contains many of the primitives in your plot, but it also has properties like the `xscale` to control whether the `xaxis` is 'linear' or 'log'. In this section we'll review where the various container objects store the `Artists` that you want to get at.

Figure container

A top-level container `Artist` is the `matplotlib.figure.Figure`, and it contains everything in the figure. The background of the figure is a `Rectangle` which is stored in `Figure.patch`. As you add subplots (`add_subplot()`) and axes (`add_axes()`) to the figure these will be appended to the `Figure.axes`. These are also returned

by the methods that create them:

```
fig = plt.figure()
ax1 = fig.add_subplot(211)
ax2 = fig.add_axes([0.1, 0.1, 0.7, 0.3])
ax1
print(fig.axes)
```

Because the figure maintains the concept of the "current Axes" (see `Figure.gca` and `Figure.sca`) to support the pylab/pyplot state machine, you should not insert or remove Axes directly from the Axes list, but rather use the `add_subplot()` and `add_axes()` methods to insert, and the `Axes.remove` method to delete. You are free however, to iterate over the list of `Axes` or index into it to get access to `Axes` instances you want to customize. Here is an example which turns all the Axes grids on:

```
for ax in fig.axes:
    ax.grid(True)
```

The figure also has its own `images`, `lines`, `patches` and `text` attributes, which you can use to add primitives directly. When doing so, the default coordinate system for the `Figure` will simply be in pixels (which is not usually what you want). If you instead use Figure-level methods to add Artists (e.g., using `Figure.text` to add text), then the default coordinate system will be "figure coordinates" where (0, 0) is the bottom-left of the figure and (1, 1) is the top-right of the figure.

As with all `Artists`, you can control this coordinate system by setting the `transform` property. You can explicitly use figure coordinates by setting the `Artist` transform to `fig.transFigure`, and we demonstrate this below, showing the output in **Figure 7-8**.

```
import matplotlib.lines as line
fig = plt.figure()

l1 = line.Line2D ([0, 1], [0, 1], transform =
                 fig.transFigure, figure = fig)
l2 = line.Line2D([0, 1], [1, 0], transform =
                 fig.transFigure, figure = fig)
```

```
fig.lines.extend([l1, l2])
plt.show()
```

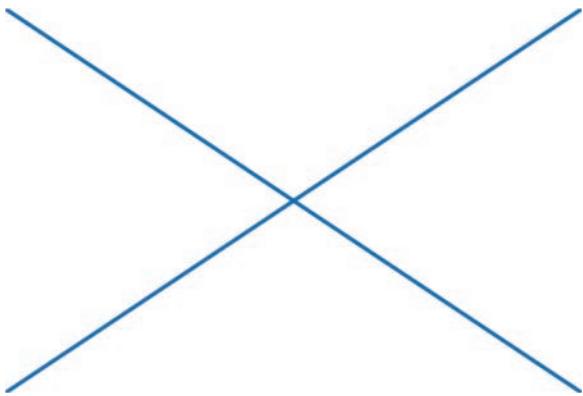


Figure 7-8. Unlike “Hello World,” X marks the spot...

Axes container

The `matplotlib.axes.Axes` is the center of the Matplotlib universe -- it contains the vast majority of all the Artists used in a figure with many helper methods to create and add these `Artists` to itself, as well as helper methods to access and customize the `Artists` it contains. Like the `Figure`, it contains a Patch `patch` which is a `Rectangle` for Cartesian coordinates and a `Circle` for polar coordinates; this patch determines the shape, background and border of the plotting region:

```
ax = fig.add_subplot()
rect = ax.patch # a Rectangle instance
rect.set_facecolor('green')
```

When you call a plotting method, e.g., the canonical `plot()` and pass in arrays or lists of values, the method will create a `matplotlib.lines.Line2D()` instance, update the line with all the `Line2D` properties passed as keyword arguments, add the line to the `Axes.lines` container, and returns it to you:

```
x, y = np.random.rand(2, 100)

line, = ax.plot(x, y, '-.', color='blue', linewidth=2)
```

`plot` returns a list of lines because you can pass in multiple x, y pairs to plot, and we are unpacking the first element of the length one list into the line variable. The line has been added to the `Axes.lines` list:

```
print(ax.lines)
```

Similarly, methods that create patches, like `bar()` creates a list of rectangles, will add the patches to the `Axes.patches` list:

```
n, bins, rectangles = ax.hist(np.random.randn(1000), 50)
rectangles
print(len(ax.patches))
```

50

You should not add objects directly to the `Axes.lines` or `Axes.patches` lists unless you know exactly what you are doing, because the `Axes` needs to do a few things when it creates and adds an object. It sets the figure and axes property of the `Artist`, as well as the default `Axes` transformation (unless a transformation is set). It also inspects the data contained in the `Artist` to update the data structures controlling auto-scaling, so that the view limits can be adjusted to contain the plotted data. You can, nonetheless, create objects yourself and add them directly to the `Axes` using helper methods like `add_line()` and `add_patch()`. Here is an annotated interactive session illustrating what is going on:

There are many, many `Axes` helper methods for creating primitive `Artists` and adding them to their respective containers. The table below summarizes a small sampling of them, the kinds of `Artist` they create, and where they store them

Axes helper method	Artist	Container
annotate - text annotations	Annotation	ax.texts
bar - bar charts	Rectangle	ax.patches
errorbar - error bar plots	Line2D and Rectangle	ax.lines and ax.patches
fill - shared area	Polygon	ax.patches
hist - histograms	Rectangle	ax.patches
imshow - image data	AxesImage	ax.images
legend - Axes legends	Legend	ax.legend

Axes helper method	Artist	Container
plot - xy plots	Line2D	ax.lines
scatter - scatter charts	PolyCollection	ax.collections
text - text	Text	ax.texts

In addition to all of these `Artists`, the `Axes` contains two important Artist containers: the `XAxis` and `YAxis`, which handle the drawing of the ticks and labels. These are stored as instance variables `xaxis` and `yaxis`. The `XAxis` and `YAxis` containers will be detailed below, but note that the `Axes` contains many helper methods which forward calls on to the `Axis` instances so you often do not need to work with them directly unless you want to. For example, you can set the font color of the `XAxis` ticklabels using the `Axes` helper method:

Axis containers

The `matplotlib.axis.Axis` instances handle the drawing of the tick lines, the grid lines, the tick labels and the axis label. You can configure the left and right ticks separately for the y-axis, and the upper and lower ticks separately for the x-axis. The `Axis` also stores the data and view intervals used in auto-scaling, panning and zooming, as well as the Locator and Formatter instances which control where the ticks are placed and how they are represented as strings.

Each `Axis` object contains a `label` attribute (this is what `pyplot` modifies in calls to `xlabel` and `ylabel`) as well as a list of major and minor ticks. The ticks are `axis.XTick` and `axis.YTick` instances, which contain the actual line and text primitives that render the ticks and ticklabels. Because the ticks are dynamically created as needed (e.g., when panning and zooming), you should access the lists of major and minor ticks through their accessor methods `axis.Axis.get_major_ticks` and `axis.Axis.get_minor_ticks`. Although the ticks contain all the primitives and will be covered below, `Axis` instances have accessor methods that return the tick lines, tick labels, tick locations etc.:

```
fig, ax = plt.subplots()
axis = ax.xaxis
axis.get_ticklocs()
```

Axis accessor method	Description
get_scale	The scale of the Axis, e.g., 'log' or 'linear'
get_view_interval	The interval instance of the Axis view limits
get_data_interval	The interval instance of the Axis data limits
get_gridlines	A list of grid lines for the Axis
get_label	The Axis label - a Text instance
get_offset_text	The Axis offset text - a Text instance
get_ticklabels	A list of Text instances - keyword minor=True False
get_ticklines	A list of Line2D instances - keyword minor=True False
get_ticklocs	A list of Tick locations - keyword minor=True False
get_major_locator	The ticker.Locator instance for major ticks
get_major_formatter	The ticker.Formatter instance for major ticks
get_minor_locator	The ticker.Locator instance for minor ticks
get_minor_formatter	The ticker.Formatter instance for minor ticks
get_major_ticks	A list of Tick instances for major ticks
get_minor_ticks	A list of Tick instances for minor ticks
grid	Turn the grid on or off for the major or minor ticks

Tick containers

The `matplotlib.axis.Tick` is the final container object in our descent from the `Figure` to the `Axes` to the `Axis` to the `Tick`. The `Tick` contains the tick and grid line instances, as well as the label instances for the upper and lower ticks. Each of these is accessible directly as an attribute of the `Tick`.

Tick attribute	Description
tick1line	A Line2D instance
tick2line	A Line2D instance
gridline	A Line2D instance
label1	A Text instance
label2	A Text instance

So, not knowing how the details of the Artists layer works, we got an empty figure in `Figure 7-7`

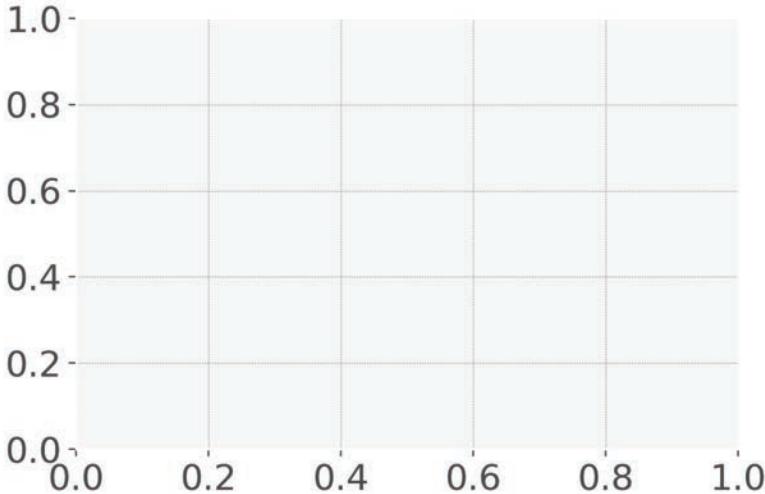


Figure 7-9. Plot showing the figure and axes without the box-plot, which is like **Figure 7-7**.

So, let's look at our incorrect code and compare it to a correct version.

```
fig, ax = plt.subplots()
df_japan.plot(kind='box', figsize= (8, 6))
plt.title('Box plot of Japanese Immigrants from
           1980-2013')
plt.ylabel('Number of Immigrants')
plt.show()
fig.savefig('myfig24', dpi=120)
```

Let's highlight the differences.

```
fig, ax1 = plt.subplots()
ax = df_japan.plot(kind='box', figsize= (8, 6), ax=ax1)
plt.title('Box plot of Japanese Immigrants\n from 1980 -
           2013')
plt.ylabel('Number of Immigrants')
plt.show()
fig.savefig('myfig25', dpi=120)
```

What may seem like a trivial change in the code produces the plt that

was intended with **Figure 7-7**.

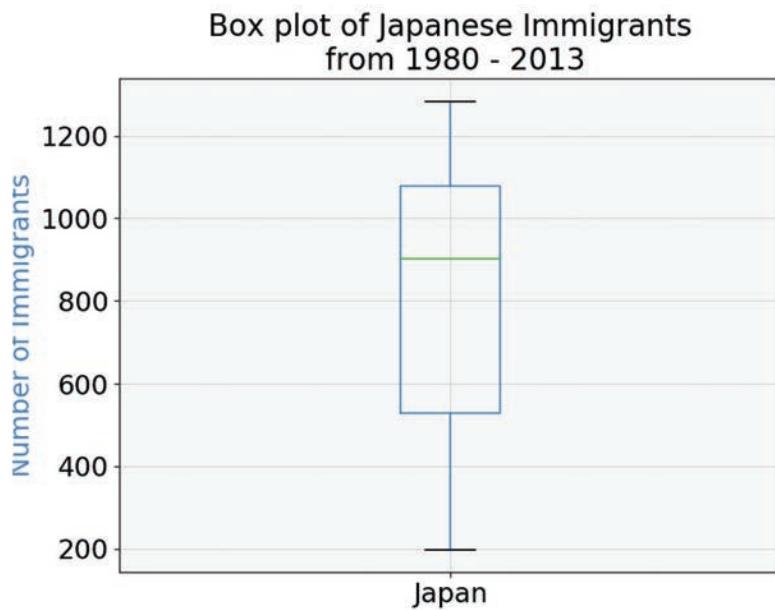


Figure 7-10. Boxplot like what we expected in **Figure 7-7** showing the characteristics of the distribution of Japanese immigrants in Canada from 1989-2013.

We can immediately make a few key observations from the plot above:

1. The minimum number of immigrants is around 200 (min), maximum number is around 1300 (max), and median number of immigrants is around 900 (median).
2. 25% of the years for period 1980 - 2013 had an annual immigrant count of ~500 or fewer (First quartile).
3. 75% of the years for period 1980 - 2013 had an annual immigrant count of ~1100 or fewer (Third quartile).

We can view the actual numbers by calling the `describe()` method on the dataframe.

```
df_japan.describe ()
```

Country	Japan
count	34.000000
mean	814.911765
std	337.219771
min	198.000000
25%	529.000000
50%	902.000000
75%	1079.000000
max	1284.000000

One of the key benefits of box plots is comparing the distribution of multiple datasets. In one of the previous labs, we observed that China and India had very similar immigration trends. Let's analyze these two countries further using box plots.

Example: Compare the distribution of the number of new immigrants from India and China for the period 1980 - 2013.

Step 1: Get the dataset for China and India and call the dataframe `df_CI`.

```
df_CI = df_can.loc [[ 'China', 'India'], years].transpose()
df_CI.head()
```

Country	China	India
1980	5123	8880
1981	6682	8670
1982	3308	8147
1983	1863	7338
1984	1527	5704

Let's view the percentiles associated with both countries using the `describe()` method. We'll generate boxplots to show and compare the percentiles in

```
df_CI.describe()
```

Country	China	India
count	34.000000	34.000000
mean	19410.647059	20350.117647
std	13568.230790	10007.342579
min	1527.000000	4211.000000
25%	5512.750000	10637.750000

Country	China	India
50%	19945.000000	20235.000000
75%	31568.500000	28699.500000
max	42584.000000	36210.000000

Step 2: Plot data.

```
df_CI.plot(kind= 'box', figsize = (10, 7))
plt.title('Box plots of Immigrants from China and India
(1980 - 2013)')
plt.ylabel('Number of Immigrants')
plt.rc('grid', color = 'deepskyblue')
plt.rc('grid', linestyle = ':')
plt.rc('grid', linewidth = 0.5)
plt.rc('axes', facecolor = 'snow')
plt.show()
fig.savefig('myfig26', dpi=120)
```

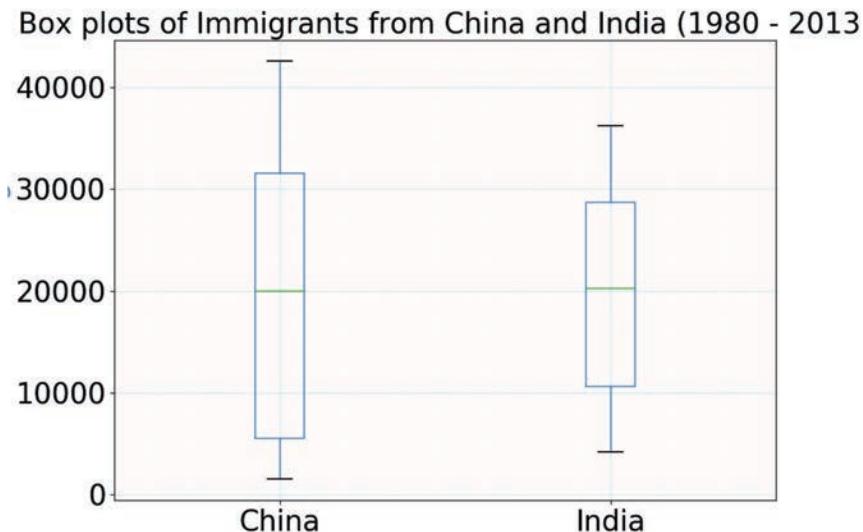


Figure 7-11. Boxplot showing the percentiles of China and India immigration to Canada from 1980-2013.

Now we'll make the same boxplots in **Figure 7-11** but using the plotting parameters we defined at the beginning of the chapter. In **Figure 7-12** our plot has a light gray `facecolor` while the `facecolor` in **Figure 7-11** is '`snow`'. The other difference are more

subtle; see if you can find them.

```
set_pub()  
fig, ax1 = plt.subplots()  
ax = df_CI.plot(kind='box', figsize= (10, 7), ax=ax1)  
  
plt.title('Box plots of Immigrants from China and India  
(1980 - 2013)')  
plt.ylabel('Number of Immigrants')  
plt.show()  
fig.savefig('myfig27', dpi=120)
```

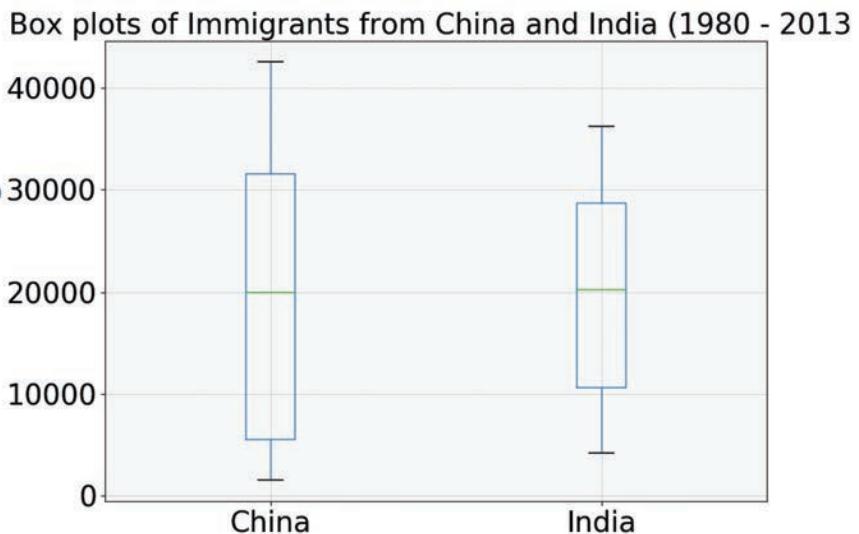


Figure 7-12. Boxplots showing the percentile of Chinese and Indian immigrants to Canada between 1980 -2013, with set_pub() parameters.

We can observe that, while both countries have around the same median immigrant population (~20,000), China's immigrant population range is more spread out than India's. The maximum population from India for any year (36,210) is around 15% lower than the maximum population from China (42,584).

If you prefer to create horizontal box plots, you can pass the `vert (verticle)` parameter in the `plot` function and assign it to `False`. You can also specify a different color in case you are not a big fan of the default `red` color.

```

# Horizontal box plots
plt.rcdefaults()
fig, ax1 = plt.subplots()
ax = df_CI.plot(kind='box', figsize=(10, 7), color='blue',
vert=False, ax=ax1)

plt.title('Box plots of Immigrants from China and India
(1980 - 2013)')
plt.xlabel('Number of Immigrants')

plt.show()
fig.savefig('myfig28', dpi=120)

```

We plot the horizontal boxplots in **Figure 7-13**, and make note that it is very difficult to read, since the font size is very small. But, we made one of those subtle changes in this figure by resetting the notebook to use the default Matplotlib plotting parameters, using `plt.redefaults()`.

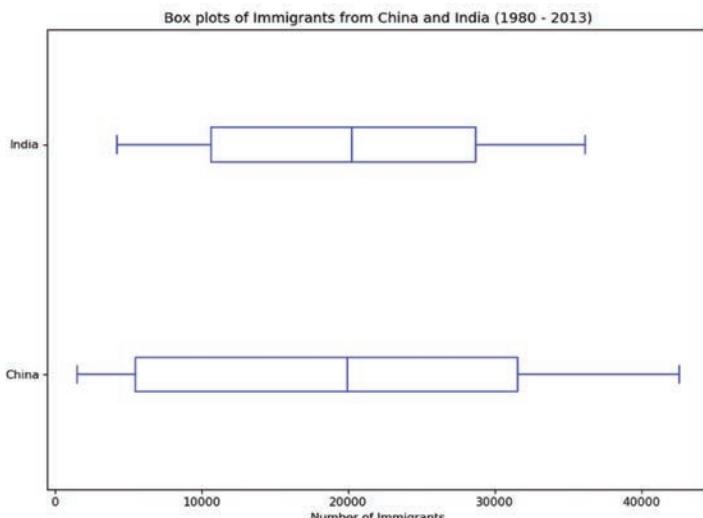


Figure 7-13. A difficult-to-read horizontal boxplot with the same information as **Figure 7-12**

Now, we'll represent the same data as in **Figure 7-13**, but with better parameters as demonstrated in **Figure 7-14**. Of note, we made the font size larger and used the 'fantasy' font. We also thickened the lines and used different colors for each of its parts.

```

fig, ax1 = plt.subplots()
color = dict(boxes='dodgerblue', whiskers='blueviolet',
medians='crimson', caps='teal') # Set boxplot parts colors

ax = df_CI.plot.box ( figsize= (10, 6), color=color,
vert=False, # Make a horizontal boxplot
whiskerprops = dict(linestyle='-', linewidth=2.0,
color='pink'), # whicker settings
boxprops = dict(linewidth=3), # box settings
medianprops = dict(linewidth=3), # median settings
capprops = dict(linewidth=3), ax=ax1) #cap settings

plt.rcParams.update({'font.family':'fantasy'})
plt.title('Box plots of Immigrants from China and India
(1980 - 2013)', fontsize=20)
plt.xlabel('Number of Immigrants', weight='bold',
fontsize=20)
plt.rc('grid', color='steelblue')
plt.rc('grid', linestyle=':')
plt.rc('grid', linewidth=1)
plt.rc('axes', facecolor='aliceblue')

plt.show()
fig.savefig('myfig29', dpi=120)

```

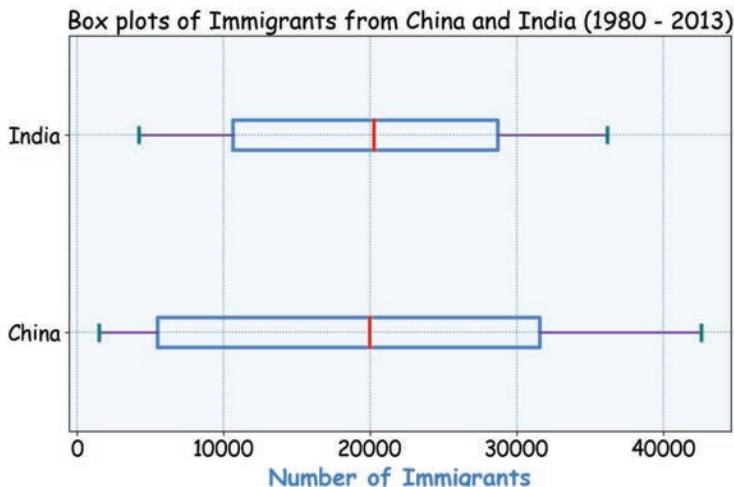


Figure 7-14. Horizontal boxplot of Chinese and Indian immigrants to Canada from 1980-2013.

Subplots

Often times we might want to plot multiple plots within the same figure. For example, we might want to perform a side by side comparison of the box plot with the line plot of China and India's immigration.

To visualize multiple plots together, we can create a `figure` (overall canvas) and divide it into `subplots`, each containing a plot, like in **Figure 7-15**. With `subplots`, we usually work with the `Artist` layer instead of the `scripting` layer.

Typical syntax is :

```
fig = plt.figure() to create a figure  
ax = fig.add_subplot(nrows, ncols, plot_number) to  
create the subplots
```

Where

- `nrows` and `ncols` are used to notionally split the figure into $(\text{nrows} * \text{ncols})$ sub-axes,
- `plot_number` is used to identify the particular subplot that this function is to create within the notional grid. `plot_number` starts at 1, increments across rows first and has a maximum of `nrows * ncols` as shown below.

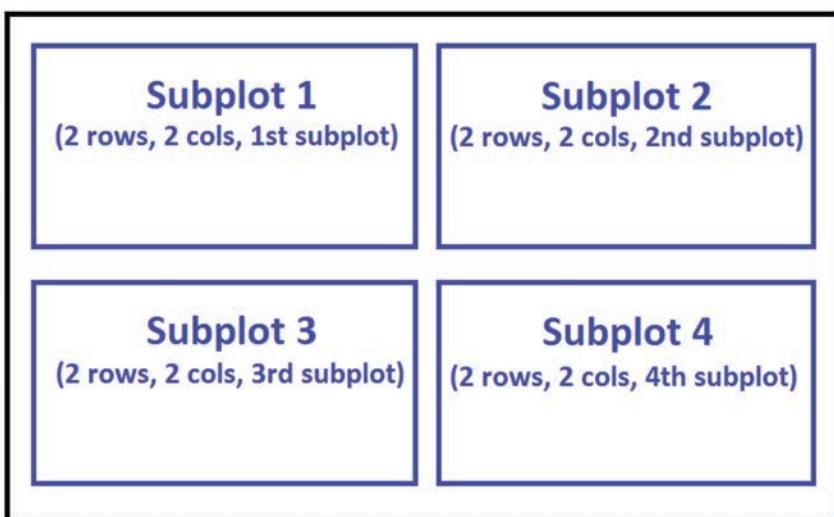


Figure 7-15. Picture showing a 2×2 plot grid of subplots.

Example: Stacked subplots

```
fig, axs = plt.subplots(2, 2)
plt.figure(dpi=200)
axs[0, 0].plot(x, y)
axs[0, 0].set_title('Axis [0, 0]')
axs[0, 1].plot(x, y, 'tab:orange')
axs[0, 1].set_title('Axis [0, 1]')
axs[1, 0].plot(x, -y, 'tab:green')
axs[1, 0].set_title('Axis [1, 0]')
axs[1, 1].plot(x, -y, 'tab:red')
axs[1, 1].set_title('Axis [1, 1]')

for ax in axs.flat:
    ax.set(xlabel='x-label', ylabel='y-label')

# Hide x labels and tick labels for top plots and y ticks
# for right plots.
for ax in axs.flat:
    ax.label_outer()
```

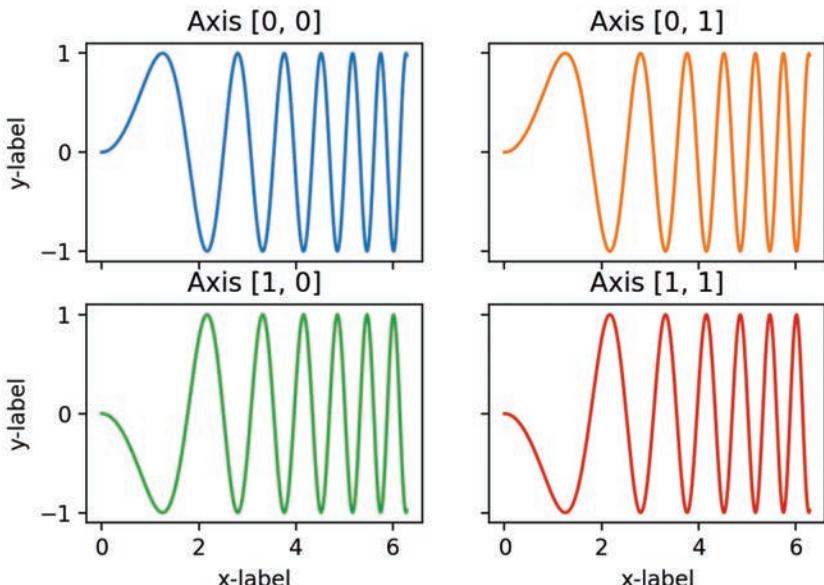


Figure 7-16. Staked 2×2 subplots (Hunter, Dale, Firing, & Droettboom, 2012)

We can then specify which `subplot` to place each plot by passing in

the `ax` parameter in `plot()` method as follows and is shown in **Figure 7-16**.

```
set_pub()
fig = plt.figure() # create figure

ax0 = fig.add_subplot(2, 1, 1) # add subplot 1 (2nd row, 1
columns, first plot)

ax1 = fig.add_subplot(2, 1, 2) # add subplot 2 (2nd row, 1
column, second plot). See tip below**

# Subplot 1: Box plot
df_CI.plot(kind='box',
            color='blue',
            vert=False,
            figsize= (20, 6),
            ax=ax0) # add to subplot 1

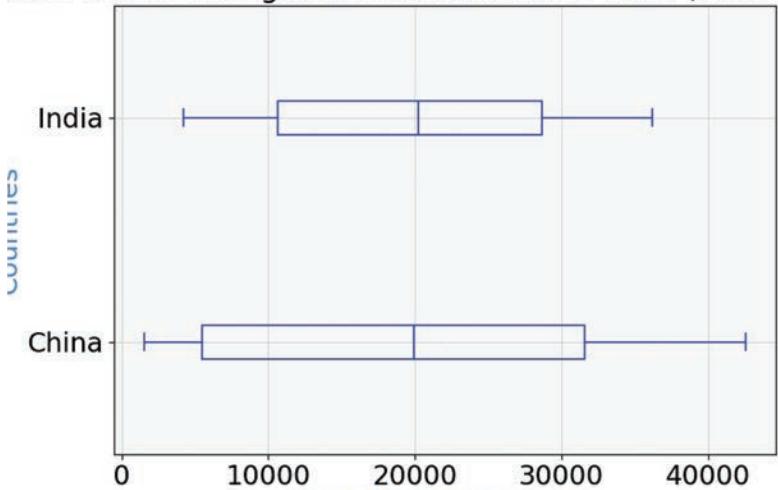
# Subplot 1 esthetics:
ax0.set_title('Box Plots of Immigrants from China and
India (1980 - 2013)')
ax0.set_xlabel('Number of Immigrants')
ax0.set_ylabel('Countries')

# Subplot 2: Line plot
df_CI.plot(kind='line',
            figsize= (8, 12),
            ax=ax1) # add to subplot 2

# Subplot 2 esthetics:
ax1.set_title ('Line Plots of Immigrants from China and
India (1980 - 2013)')
ax1.set_ylabel('Number of Immigrants')
ax1.set_xlabel('Years')

plt.show()
fig.savefig('myfig30', dpi=120)
```

Box Plots of Immigrants from China and India (1980 - 201)



Line Plots of Immigrants from China and India (1980 - 201)



Figure 7-17. Two plots arranged in two rows and one column.

Oh no! What happened? So, the top subplot is partially overlaid on the second subplot. We need to add a buffer space between the two subplots, which is easily done using:

```
fig.tight_layout(pad=0.75).
```

```

set_pub()
fig = plt.figure() # create figure

fig.tight_layout(pad=0.75) # add space between plots

ax0 = fig.add_subplot(2, 1, 1) # add subplot 1 (2nd row, 1
# columns, first plot)

ax1 = fig.add_subplot(2, 1, 2) # add subplot 2 (2nd row, 1
# column, second plot). See tip below**

# Subplot 1: Box plot
df_CI.plot(kind='box',
            color='blue',
            vert=False,
            figsize= (20, 6),
            ax=ax0) # add to subplot 1

# Subplot 1 esthetics:
ax0.set_title('Box Plots of Immigrants from China and
India (1980 - 2013)')
ax0.set_xlabel('Number of Immigrants')
ax0.set_ylabel('Countries')

# Subplot 2: Line plot
df_CI.plot(kind='line',
            figsize= (8, 12),
            ax=ax1) # add to subplot 2

# Subplot 2 esthetics:
ax1.set_title ('Line Plots of Immigrants from China and
India (1980 - 2013)')
ax1.set_ylabel('Number of Immigrants')
ax1.set_xlabel('Years')

plt.show()

```

The result of this code is shown in **Figure 7-17**. This time, the two plots are separated adequately by a buffer.

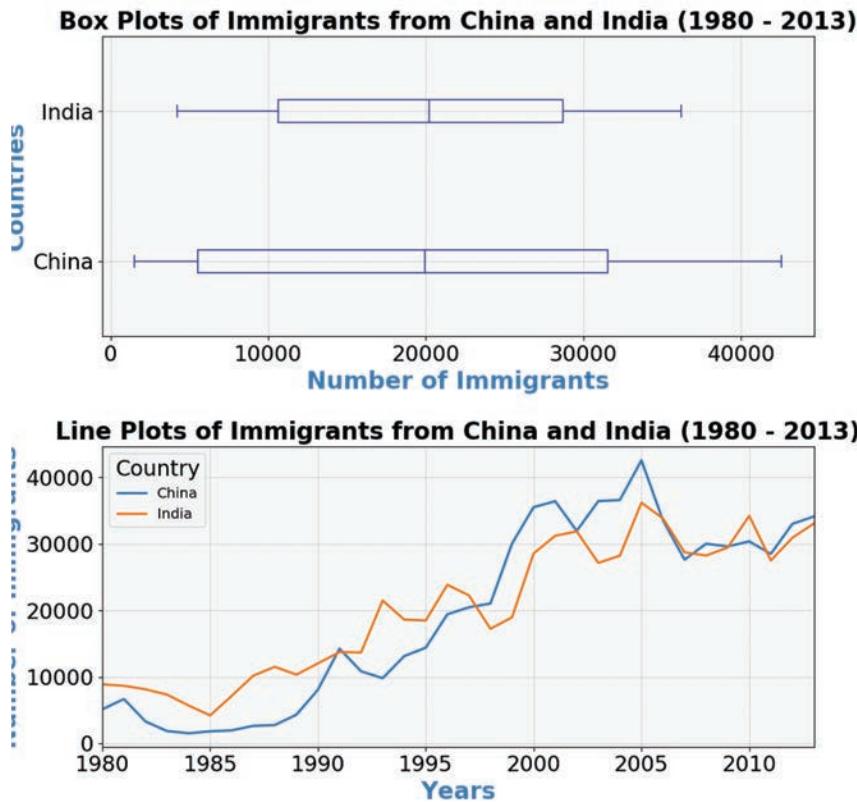


Figure 7-18. Two plots arranged in two rows and one column with a buffer space between the top and bottom plots.

Tip regarding subplot convention

In the case when `nrows`, `ncols`, and `plot_number` are all less than 10, a convenience exists such that a 3-digit number can be given instead, where the hundreds represent `nrows`, the tens represent `ncols` and the units represent `plot_number`. For instance,

```
subplot(211) == subplot(2, 1, 1)
```

produces a subaxes in a `figure` which represents the top plot (i.e. the first) in a 2 rows by 1 column notional grid (no grid actually exists, but conceptually this is how the returned `subplot` has been positioned).

Let's try something a little more advanced.

Previously we identified the top 15 countries based on total immigration from 1980 - 2013.

Example: Create a box plot to visualize the distribution of the top 15 countries (based on total immigration) grouped by the *decades* 1980s, 1990s, and 2000s.

Step 1: Get the dataset. Get the top 15 countries based on Total immigrant population. Name the dataframe `df_top15`.

```
df_top15 = df_can.sort_values(['Total'], ascending=False,
axis=0).head(15)
df_top15
```

	Continent	1980	...	2013	Total
Country					
India	Asia	8880	...	33087	691904
China	Asia	5123	...	34129	659962
United Kingdom of Great Britain and Northern Ireland	Europe	22045	...	5827	551500
Philippines	Asia	6051	...	29544	511391
Pakistan	Asia	978	...	12603	241600
United States of America	Northern America	9378	...	8501	241122
Iran (Islamic Republic of)	Asia	1172	...	11291	175923
Sri Lanka	Asia	185	...	2394	148358
Republic of Korea	Asia	1011	...	4509	142581
Poland	Europe	863	...	852	139241
Lebanon	Asia	1409	...	2172	115359
France	Europe	1729	...	5623	109091
Jamaica	Latin America and the Caribbean	3198	...	2479	106431
Viet Nam	Asia	1191	...	2112	97146
Romania	Europe	375	...	1512	93585

15 rows × 38 columns

Step 2: Create a new dataframe which contains the aggregate for each decade. One way to do that:

1. Create a list of all years in decades 80's, 90's, and 00's.

2. Slice the original dataframe `df_can` to create a series for each decade and sum across all years for each country.
3. Merge the three series into a new data frame. Call your dataframe `new_df`.

```
# create a list of all years in decades 80's, 90's, and 00's
years_80s = list(map(str, range(1980, 1990)))
years_90s = list(map(str, range(1990, 2000)))
years_00s = list(map(str, range(2000, 2010)))

# Slice the original dataframe df_can to create a series for each decade
df_80s = df_top15.loc[:, years_80s].sum(axis=1)
df_90s = df_top15.loc[:, years_90s].sum(axis=1)
df_00s = df_top15.loc[:, years_00s].sum(axis=1)

# Merge the three series into a new data frame
new_df = pd.DataFrame ({'1980s': df_80s, '1990s': df_90s,
'2000s':df_00s})

# Display dataframe
new_df.head()
```

	1980s	1990s	2000s
Country			
India	82154	180395	303591
China	32003	161528	340385
United Kingdom of Great Britain and Northern Ireland	179171	261966	83413
Philippines	60764	138482	172904
Pakistan	10591	65302	127598

Let's learn more about the statistics associated with the dataframe using the `describe()` method.

```
new_df.describe ()
```

	1980s	1990s	2000s
count	15.000000	15.000000	15.000000
mean	44418.333333	85594.666667	97471.533333
std	44190.676455	68237.560246	100583.204205
min	7613.000000	30028.000000	13629.000000
25%	16698.000000	39259.000000	36101.500000
50%	30638.000000	56915.000000	65794.000000
75%	59183.000000	104451.500000	105505.500000
max	179171.000000	261966.000000	340385.000000

Step 3: Plot the box plots.

```
fig, ax1 = plt.subplots()

plt.rcParams.update({'font.family':'Arial'})

ax = new_df.plot(kind='box',
                  figsize= (10, 6),
                  whiskerprops = dict(linewidth=2.0),
                  boxprops = dict(linewidth=3),
                  medianprops = dict(linewidth=3),
                  capprops = dict(linewidth=3),
                  flierprops=dict(marker='o', markerfacecolor = 'red',
                                  markersize  =12), ax=ax1)

plt.title('Immigration from top 15 countries\n for decades
80s, 90s and 2000s', fontsize=24, weight='bold',
color='#2138ab')
plt.rc('xtick', labelsize = 18)
plt.rc('ytick', labelsize = 18)
ax.xaxis.grid(linestyle=':', linewidth=1,
color='steelblue')

plt.show()
fig.savefig('myfig31', dpi=200)
```

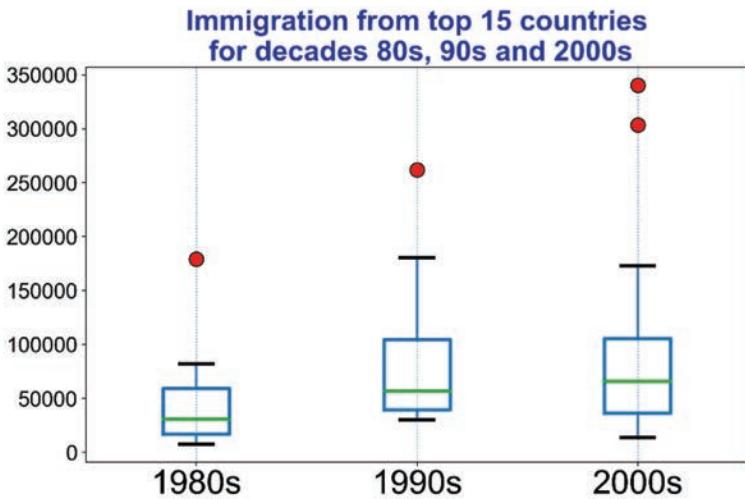


Figure 7-19. Boxplots of immigration to Canada by decades for three decades, by the top-five countries previously identified.

Note how the box plot differs from the summary table created. The box plot scans the data and identifies the outliers. In order to be an outlier, the data value must be:

- larger than Q3 by at least 1.5 times the interquartile range (IQR), or,
- smaller than Q1 by at least 1.5 times the IQR.

Let's look at decade 2000s as an example:

- Q1 (25%) = 36,101.5
- Q3 (75%) = 105,505.5
- IQR = Q3 - Q1 = 69,404

Using the definition of outlier, any value that is greater than Q3 by 1.5 times IQR will be flagged as outlier.

$$\text{Outlier} > 105,505.5 + (1.5 * 69,404)$$

$$\text{Outlier} > 209,611.5$$

```
new_df=new_df.reset_index ()
new_df[new_df['2000s']> 209611.5]
```

	Country	1980s	1990s	2000s
0	India	82154	180395	303591
1	China	32003	161528	340385

China and India are both considered as outliers since their population for the decade exceeds 209,611.5.

The box plot is an advanced visualization tool, and there are many options and customizations that exceed the scope of this lab. Please refer to Matplotlib documentation on box plots for more information.

Scatter Plots

A scatter plot (2D) is a useful method of comparing variables against each other. Scatter plots look similar to line plots in that they both map independent and dependent variables on a 2D graph. While the data points are connected together by a line in a line plot, they are not connected in a scatter plot. The data in a scatter plot is considered to express a trend. With further analysis using tools like regression, we can mathematically calculate this relationship and use it to predict trends outside the dataset.

Let's start by exploring the following:

Using a scatter plot, let's visualize the trend of total immigration to Canada (all countries combined) for the years 1980 - 2013.

Step 1: Get the dataset. Since we are expecting to use the relationship between years and total population, we will convert years to int type. We can use the `sum()` method to get the total population per year

```
df_tot = pd.DataFrame (df_can[years].sum(axis=0))

# Change the years to type int (useful for regression later on)
df_tot.index = map(int, df_tot.index)

# Reset the index to put in back in as a column in the df_tot dataframe
```

```

df_tot.reset_index (inplace = True)

# Rename columns
df_tot.columns = ['year', 'total']

# View the final dataframe
df_tot.head()

```

	year	total
0	1980	99137
1	1981	110563
2	1982	104271
3	1983	75550
4	1984	73417

Step 2: Plot the data. In Matplotlib, we can create a scatter plot set by passing in `kind='scatter'` as plot argument. We will also need to pass in `x` and `y` keywords to specify the columns that go on the x- and the y-axis. We show the scatterplot in **Figure 7-19**.

```

fig, ax1 = plt.subplots()

ax = df_tot.plot(kind='scatter', x='year', y='total',
                  figsize= (10, 6), color='darkblue', s = 50, ax=ax1)

plt.title('Total Immigration to Canada from 1980 - 2013',
          fontsize=24, weight='bold', color='#2138ab')
plt.xlabel('Year', fontsize = 18)
plt.ylabel('Number of Immigrants', fontsize = 18)
plt.rc('xtick', labelsize = 18)
plt.rc('ytick', labelsize = 18)
ax.xaxis.grid(linestyle=':', linewidth=1,
              color='steelblue')
ax.yaxis.grid(linestyle=':', linewidth=1,
              color='steelblue')

plt.show()
fig.savefig('myfig32', dpi=200)

```

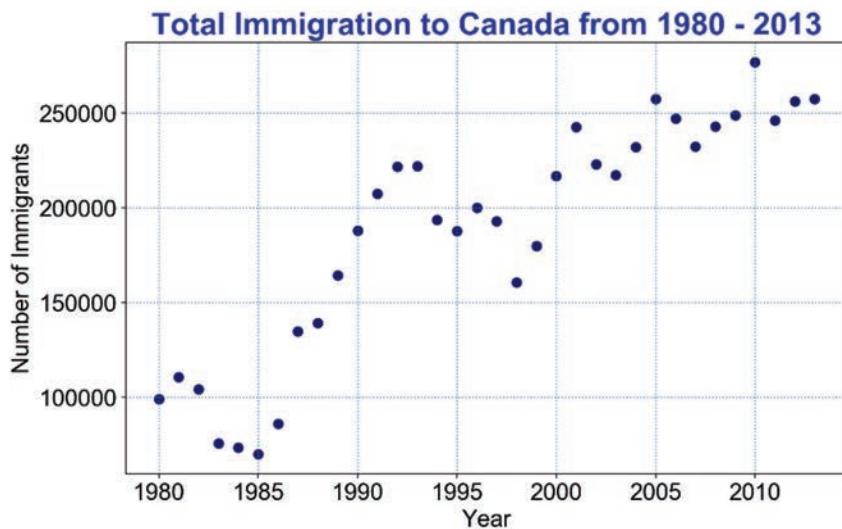


Figure 7-20. Scatterplot of total immigration to Canada from 1980-2013

Notice how the scatter plot does not connect the data points together. We can clearly observe an upward trend in the data: as the years go by, the total number of immigrants increases. We can mathematically analyze this upward trend using a regression line (line of best fit).

So, let's try to plot a linear line of best fit, and use it to predict the number of immigrants in 2015.

Step 1: Get the equation of line of best fit. We will use **Numpy's** **polyfit()** method by passing in the following:

- **x**: x-coordinates of the data.
- **y**: y-coordinates of the data.
- **deg**: Degree of fitting polynomial. 1 = linear, 2 = quadratic, and so on.

```
x = df_tot['year']      # year on x-axis
y = df_tot['total']     # total on y-axis
```

```
fit = np.polyfit (x, y, deg=1)
fit
```

```
array([ 5.56709228e+03, -1.09261952e+07])
```

The output is an array with the polynomial coefficients, highest

powers first. Since we are plotting a linear regression $y = a x + b$, our output has 2 elements [5.56709228e+03, -1.09261952e+07] with the the slope in position 0 and intercept in position 1.

Step 2: Plot the regression line on the scatter plot. We show the plot in **Figure 7-20**.

```
fig, ax1 = plt.subplots()

ax = df_tot.plot(kind='scatter', x='year', y='total',
                  figsize= (10, 6), color='darkblue', s = 50, ax=ax1)

plt.title('Total Immigration to Canada from 1980 - 2013',
          fontsize=24, weight='bold', color='#2138ab')
plt.xlabel('Year', fontsize = 24)
plt.ylabel('Number of Immigrants', fontsize=24)
plt.rc('xtick', labelsize = 18)
plt.rc('ytick', labelsize = 18)

ax.xaxis.grid(linestyle = ':', linewidth=1,
              color='steelblue')
ax.yaxis.grid(linestyle = ':', linewidth=1,
              color='steelblue')

# Plot Line of best fit
plt.plot(x, fit [0] * x + fit[1], color='red') # recall
    that x is the Years
plt.annotate('y={0:.0f} x + {1:.0f}'.format(fit[0],
                                             fit[1]), xy = (2000, 150000))

plt.show()
fig.savefig('myfig33', dpi=200)

# Print out the line of best fit
'No. Immigrants = {0:.0f} * Year + {1:.0f}'.format(fit[0],
                                                    fit[1])
```

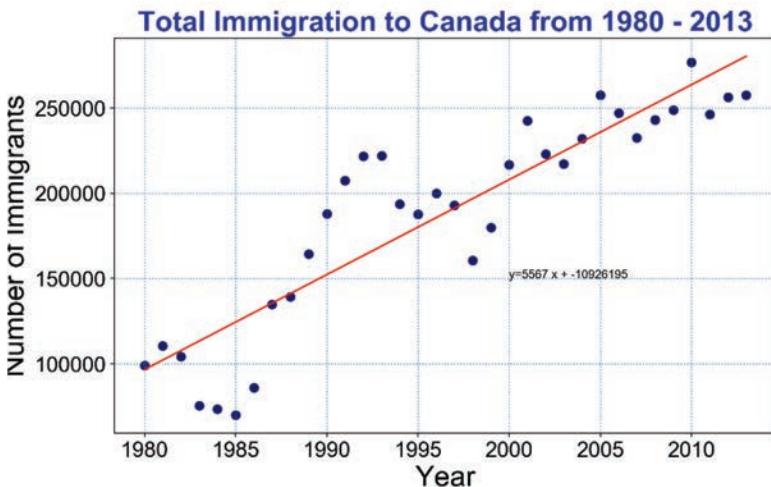


Figure 7-21. Scatterplot of total immigration to Canada from 1980-2013 with a fitted line.

'No. Immigrants = 5567 * Year + -10926195'

Using the equation of line of best fit, we can estimate the number of immigrants in 2015:

$$\text{No. Immigrants} = 5567 * \text{Year} - 10926195$$

$$\text{No. Immigrants} = 5567 * 2015 - 10926195$$

$$\text{No. Immigrants} = 291,310$$

When compared to the actual from Citizenship and Immigration Canada's (CIC) 2016 Annual Report, we see that Canada accepted 271,845 immigrants in 2015. Our estimated value of 291,310 is within 7% of the actual number, which is pretty good considering our original data came from United Nations (and might differ slightly from CIC data).

As a side note, we can observe that immigration took a dip around 1993 - 1997. Further analysis into the topic revealed that in 1993 Canada introduced Bill C-86 which introduced revisions to the refugee determination system, mostly restrictive. Further amendments to the Immigration Regulations cancelled the sponsorship required for "assisted relatives" and reduced the points awarded to them, making it more difficult for family members (other than nuclear family) to immigrate to Canada. These restrictive

measures had a direct impact on the immigration numbers for the next several years.

Example: Create a scatter plot of the total immigration from Denmark, Norway, and Sweden to Canada from 1980 to 2013?

Step 1: Get the data:

1. Create a dataframe that consists of the numbers associated with Denmark, Norway, and Sweden only. Name it `df_countries`.
2. Sum the immigration numbers across all three countries for each year and turn the result into a dataframe. Name this new dataframe `df_total`.
3. Reset the index in place.
4. Rename the columns to `year` and `total`.
5. Display the resulting dataframe.

```
# Create df_countries dataframe
df_countries = df_can.loc[['Denmark', 'Norway', 'Sweden'],
                           years].transpose()

# Create df_total by summing across three countries for
# each year
df_total = pd.DataFrame (df_countries.sum(axis=1))

# Reset index in place
df_total.reset_index (inplace=True)

# Rename columns
df_total.columns = ['year', 'total']

# Change column year from string to int to create scatter
# plot
df_total['year'] = df_total['year'].astype (int)

# Show resulting dataframe
df_total.head()
```

year	total
0	1980669
1	1981678
2	1982627
3	1983333
4	1984252

Step 2: Generate the scatter plot by plotting the total versus year in `df_total`. This plot is shown in **Figure 7-21**.

```
# Generate scatter plot
fig, ax1 = plt.subplots()

ax = df_total.plot(kind='scatter',
                    x='year',
                    y='total',
                    figsize= (10, 6),
                    color='dodgerblue',
                    s = 50,
                    ax=ax1)

# Add title and label to axes
plt.title('Immigration from Denmark, Norway, and Sweden\n
           to Canada from 1980 - 2013', fontsize=24,
           weight='bold', color='dodgerblue')

plt.xlabel('Year', fontsize=4)
plt.ylabel('Number of Immigrants', fontsize=24)
plt.rc('xtick', labelsize = 18)
plt.rc('ytick', labelsize = 18)

ax.xaxis.grid(linestyle=':', linewidth=1,
              color='steelblue')
ax.yaxis.grid(linestyle=':', linewidth=1,
              color='steelblue')

plt.show()
fig.savefig('myfig34', dpi=200)
```

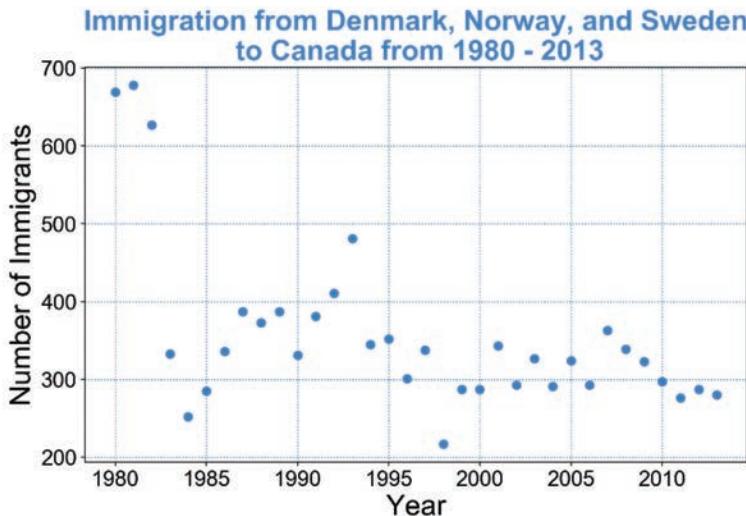


Figure 7-22. Scatter plot of the total immigration from Denmark, Norway, and Sweden to Canada from 1980 to 2013

1. Bubble Plots

A bubble plot is a variation of the scatter plot that displays three dimensions of data (x, y, z). The data points are replaced with bubbles, and the size of the bubble is determined by the third variable z, also known as the weight. In `matplotlib`, we can pass in an array or scalar to the parameter s to `plot()`, that contains the weight of each point.

Let's start by analyzing the effect of Argentina's great depression.

Argentina suffered a great depression from 1998 to 2002, which caused widespread unemployment, riots, the fall of the government, and a default on the country's foreign debt. In terms of income, over 50% of Argentines were poor, and seven out of ten Argentine children were poor at the depth of the crisis in 2002.

Let's analyze the effect of this crisis, and compare Argentina's immigration to that of its neighbor Brazil. Let's do that using a bubble plot of immigration from Brazil and Argentina for the years 1980 - 2013. We will set the weights for the bubble as the *normalized* value of the population for each year.

Step 1: Get the data for Brazil and Argentina. Like in the previous example, we will convert the Years to type int and include it in the dataframe.

```
# Transposed dataframe
df_can_t = df_can[years].transpose()

# Cast the Years (the index) to type int
df_can_t.index = map(int, df_can_t.index)

# Let's Label the index. This will automatically be the
# column name when we reset the index
df_can_t.index.name = 'Year'

# Reset index to bring the Year in as a column
df_can_t.reset_index(inplace=True)

# View the changes
df_can_t.head()
```

Country	Year	Afghanistan	...	Yemen	Zambia	Zimbabwe
0	1980	16	...	1	11	72
1	1981	39	...	2	17	114
2	1982	39	...	1	11	102
3	1983	47	...	6	7	44
4	1984	71	...	0	16	32

5 rows × 196 columns

Step 2: Create the normalized weights.

There are several methods of normalizations in statistics, each with its own use. In this case, we will use feature scaling to bring all values into the range [0, 1]. The general formula is:

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

where X is the original value, X' is the corresponding normalized value. The formula sets the max value in the dataset to 1, and sets the

min value to 0. The rest of the data points are scaled to a value between 0 – 1 accordingly.

```
# Normalize Brazil data
norm_brazil = (df_can_t['Brazil'] -
df_can_t['Brazil'].min()) / (df_can_t['Brazil'].max() -
df_can_t['Brazil'].min())

# Normalize Argentina data
norm_argentina = (df_can_t['Argentina'] -
df_can_t['Argentina'].min()) / (df_can_t['Argentina'].max() -
df_can_t['Argentina'].min())
```

Step 3: Plot the data.

- To plot two different scatter plots in one plot, we can include the axes one plot into the other by passing it via the ax parameter. We show the finished plot in **Figure 7-22**.
- We will also pass in the weights using the s parameter. Given that the normalized weights are between 0-1, they won't be visible on the plot. Therefore, we will:
 - multiply weights by 2000 to scale it up on the graph, and,
 - add 10 to compensate for the min value (which has a 0 weight and therefore scale with $\times 2000$).

```
# Brazil
fig, ax = plt.subplots()
ax0 = df_can_t.plot(kind='scatter',
                     x='Year',
                     y='Brazil',
                     figsize= (14, 8),
                     alpha=0.5, # transparency
                     color='green',
                     s=norm_brazil * 2000 + 10, # pass in weights
                     xlim = (1975, 2015),
                     ax=ax)

# Argentina
ax1 = df_can_t.plot(kind='scatter',
                     x='Year',
```

```

y='Argentina',
alpha=0.5,
color='blue',
s=norm_argentina * 2000 + 10,
ax=ax0
)

ax0.set_xlabel('Year', fontsize=24)
ax0.set_ylabel('Number of Immigrants', fontsize=24)
ax0.set_title('Immigration from Brazil and Argentina from 1980 to 2013', fontsize=30)
ax0.legend(['Brazil', 'Argentina'], loc='upper left',
fontsize='x-large')

fig.savefig('myfig35', dpi=200)

```

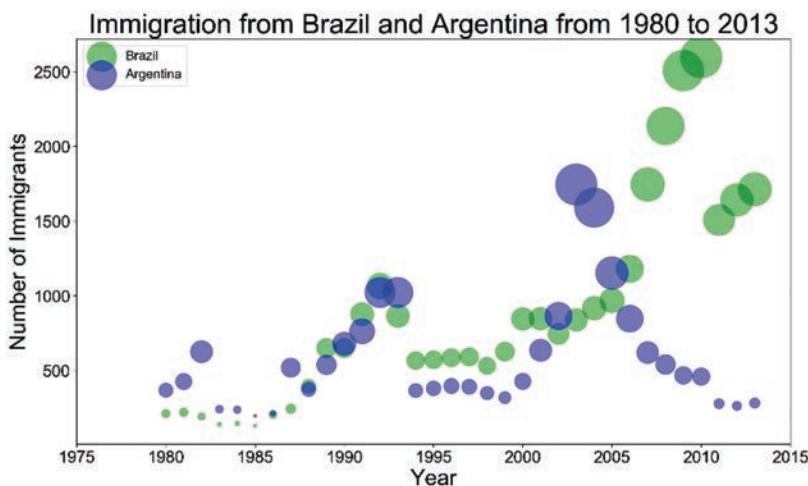


Figure 7-23. Bubble plot of Brazilian and Argentine immigration to Canada from 1980-2013

The size of the bubble corresponds to the magnitude of immigrating population for that year, compared to the 1980 - 2013 data. The larger the bubble is, the more immigrants are in that year.

From the plot above, we can see a corresponding increase in immigration from Argentina during the 1998 - 2002 great

depression. We can also observe a similar spike around 1985 to 1993. In fact, Argentina had suffered a great depression from 1974 to 1990, just before the onset of 1998 - 2002 great depression.

On a similar note, Brazil suffered the *Samba Effect* where the Brazilian real (currency) dropped nearly 35% in 1999. There was a fear of a South American financial crisis as many South American countries were heavily dependent on industrial exports from Brazil. The Brazilian government subsequently adopted an austerity program, and the economy slowly recovered over the years, culminating in a surge in 2010. The immigration data reflect these events.

Question: Previously in this lab, we created box plots to compare immigration from China and India to Canada. Create bubble plots of immigration from China and India to visualize any differences with time from 1980 to 2013. You can use `df_can_t` that we defined and used in the previous example.

Step 1: Normalize the data pertaining to China and India.

```
# Normalized Chinese data
norm_china = (df_can_t['China'] - df_can_t['China'].min()) / (df_can_t['China'].max() - df_can_t['China'].min())

# Normalized Indian data
norm_india = (df_can_t['India'] - df_can_t['India'].min()) / (df_can_t['India'].max() - df_can_t['India'].min())
```

Step 2: Generate the bubble plots shown in **Figure 7-23**.

```
# China
fig, ax = plt.subplots()
ax0 = df_can_t.plot(kind='scatter',
                     x='Year',
                     y='China',
                     figsize= (14, 8),
                     alpha=0.5, # transparency
                     color='green',
                     s=norm_china * 2000 + 10, # pass in weights
                     xlim= (1975, 2015),
                     ax=ax
                    )
```

```

# India
ax1 = df_can_t.plot(kind='scatter',
                     x='Year',
                     y='India',
                     alpha=0.5,
                     color='blue',
                     s=norm_india * 2000 + 10,
                     ax = ax0
                    )

ax0.set_xlabel('Year', fontsize=24)
ax0.set_ylabel('Number of Immigrants', fontsize=24)
ax0.set_title('Immigration from China and India from 1980 - 2013', fontsize=36)
ax0.legend(['China', 'India'], loc='upper left',
           fontsize='x-large')

fig.savefig('myfig36', dpi=200)

```

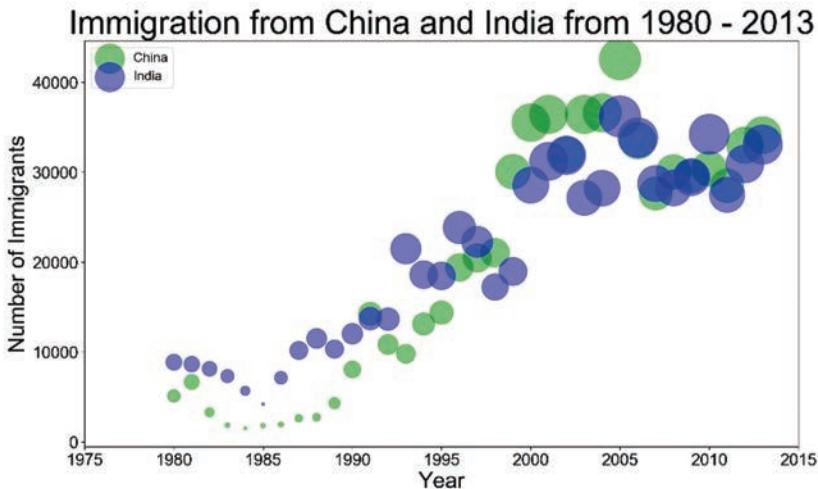


Figure 7-24. Scatterplot of the normalized data pertaining to China and India

8. Waffle Charts, Word Clouds, and Regression Plots

Objectives

After completing this chapter, you will be able to:

- Downloading and preprocessing data
- Further explore datasets with **pandas**
- Further explore visualizing data using **Matplotlib**
- Create waffle charts, word clouds, and regression plots

Exploring Datasets with **pandas** and **Matplotlib**

Toolkits: The course heavily relies on **Pandas** and **Numpy** for data wrangling, analysis, and visualization. The primary plotting library we will explore in the course is **Matplotlib**.

Dataset: Immigration to Canada from 1980 to 2013 - [International migration flows to and from selected countries - The 2015 revision](#) from United Nation's website.

The dataset here is the same we've been using, with annual data on the flows of international migrants as recorded by the countries of destination. Once again, we will focus on the Canadian Immigration data.

Downloading and Prepping Data

Import Primary Modules. The first thing we'll do is import two key data analysis modules: pandas and numpy.

```
import numpy as np # useful for scientific computing
import pandas as pd # primary data structure Library
```

Let's download and import our primary Canadian Immigration dataset using **pandas's** `read_excel ()` method. Normally, before we can do that, we would need to download a module which **pandas** requires reading in *Excel* files. This module was **openpyxl** (formerly **xlrd**). I pre-installed for this module on *Github*, so you would not

have to worry about that. Otherwise, you would need to run the following line of code to install the `openpyxl` module:

```
! pip3 install openpyxl
```

Download the dataset and read it into a *pandas* dataframe.

```
df_can = pd.read_excel('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DV0101EN-SkillsNetwork/Data%20Files/Canada.xlsx',
                       sheet_name='Canada by Citizenship',
                       skiprows=range(20),
                       skipfooter=2)
print('Data downloaded and read into a dataframe!')
```

Data downloaded and read into a dataframe!

Let's take a look at the first five items in our dataset.

```
df_can.head()
```

Type	Coverage	REG	RegName	1980	...	2013	
0	Immigrants	Foreigners	5501	Southern Asia	16	...	2004
1	Immigrants	Foreigners	925	Southern Europe	1	...	603
2	Immigrants	Foreigners	912	Northern Africa	80	...	4331
3	Immigrants	Foreigners	957	Polynesia	0	...	0
4	Immigrants	Foreigners	925	Southern Europe	0	...	1

5 rows × 43 columns

We can also view the bottom 5 rows of the dataset using the `tail()` function.

Let's find out how many entries there are in our dataset.

```
# Print the dimensions of the dataframe
print(df_can.shape)
```

(195, 43)

Clean up Data

We will make some modifications to the original dataset to make it

easier to create our visualizations. Refer to the chapter, *Introduction to Matplotlib and Line Plots and Area Plots, Histograms, and Bar Plots* for a detailed description of this preprocessing. So, let's clean up the dataset to remove unnecessary columns (e.g., REG)

```
df_can.drop ([ 'AREA', 'REG', 'DEV', 'Type', 'Coverage' ],  
axis=1, inplace=True)
```

Now, let's rename the columns so that they make sense

```
df_can.rename (columns={ 'OdName': 'Country', 'AreaName':  
'Continent', 'RegName': 'Region'}, inplace=True)
```

For sake of consistency, let's also make all column labels of type string

```
df_can.columns = list(map(str, df_can.columns))
```

Now we'll set the country name as index - useful for quickly looking up countries using .loc method and add total column.

```
df_can.set_index('Country', inplace=True)  
df_can[ 'Total' ] = df_can.sum(axis=1)
```

We'll also setup the years that we will be using in this chapter - useful for plotting later on.

```
years = list(map(str, range(1980, 2014)))  
print('data dimensions:', df_can.shape)  
data dimensions: (195, 38)
```

Visualizing Data using Matplotlib

Import Matplotlib.

```
import matplotlib as mpl  
import matplotlib.pyplot as plt  
%matplotlib inline  
  
mpl.style.use('ggplot') # optional: for ggplot-like style  
# check for latest version of Matplotlib  
print('Matplotlib version: ', mpl.__version__) # >= 2.0.0
```

Matplotlib version: 3.1.1

Set Notebook Plotting Parameters

Near the end of Chapter 7, we showed how to define our custom plotting parameters for use throughout all our plotting operations, unless we override them or reset the **Matplotlib** default values.

```
def set_pub():
    # plt.rc('font', weight='bold') # bold fonts
    # plt.rc('tick', labelsize=15) # bigger tick Labels
    plt.rcParams ['font.family'] = 'sans-serif'
    plt.rcParams['font.sans-serif'] # available fonts -
        see tip below
    plt.rc('lines', lw=1, color='k') # thicker black Lines
    plt.rc('grid', c='0.5', ls='-', lw=0.5) # solid gray
        grid Lines
    plt.rc('savefig', dpi=200)      # higher res outputs
    plt.rc('font', size = 18) # default font size
    plt.rc('axes', titlesize = 24) # plot title Label size
    plt.rc('axes', labelsize = 18) # x and y Label size
    plt.rc('axes', labelcolor='steelblue')
    plt.rc('xtick', labelsize = 18) # x tick Label size
    plt.rc('ytick', labelsize = 18) # y tick Label size
    plt.rc('legend', fontsize = 12) # legend font size
    plt.rc('axes', axisbelow=True) # tickmarks below axis
    plt.rc('axes', grid=True) # grid if true
    plt.rc('axes', facecolor='whitesmoke') # background
    plt.rc('grid', color='gray') # grid-Line color
    plt.rc('grid', linestyle':') # grid Line style
    plt.rc('grid', linewidth=0.5) # grid-Line thickness
```

Tip: We can list the fonts in the san-serif family by executing the following cell:

```
plt.rcParams['font.sans-serif']
```

Waffle Charts

A **waffle chart** is an interesting visualization that is normally created to display progress toward goals. It is commonly an effective option when you are trying to add interesting visualization features to a visual that consists mainly of cells, such as an *Excel* dashboard.

Let's revisit the previous case study about Denmark, Norway, and Sweden.

```
# Let's create a new dataframe for these three countries
df_dsn = df_can.loc [['Denmark', 'Norway', 'Sweden'], :]
# Let's take a look at our dataframe
df_dsn
```

Country	Continent	Region	DevName	1981 ... 2013 Total
Denmark	Europe	Northern Europe	Developed regions	272 ... 81 3901
Norway	Europe	Northern Europe	Developed regions	116 ... 59 2327
Sweden	Europe	Northern Europe	Developed regions	281 ... 140 5866

3 rows × 38 columns

Unfortunately, unlike R, `waffle` charts are not built into any of the `Python` visualization libraries. Therefore, we will learn how to create them from scratch.

Step 1. The first step into creating a waffle chart is determining the proportion of each category with respect to the total.

```
# Compute the proportion of each category with respect to
# the total
total_values = df_dsn['Total'].sum()
category_proportions = df_dsn['Total'] / total_values
```

```
# Print out proportions
pd.DataFrame ({"Category Proportion": 
category_proportions})
```

Category Proportion	
Country	
Denmark	0.322557
Norway	0.192409
Sweden	0.485034

Step 2. The second step is defining the overall size of the `waffle` chart.

```
width = 40 # width of chart  
height = 10 # height of chart  
  
total_num_tiles = width * height # total number of tiles  
  
print(f'Total number of tiles is {total_num_tiles}.')  
Total number of tiles is 400.
```

Step 3. The third step is using the proportion of each category to determine its respective number of tiles

```
# Compute the number of tiles for each category  
tiles_per_category = (category_proportions *  
total_num_tiles).round().astype (int)  
  
# Print out number of tiles per category  
pd.DataFrame({"Number of tiles": tiles_per_category})
```

Number of tiles	
Country	
Denmark	129
Norway	77
Sweden	194

Based on the calculated proportions, Denmark will occupy 129 tiles of the `waffle` chart, Norway will occupy 77 tiles, and Sweden will occupy 194 tiles.

Step 4. The fourth step is creating a matrix that resembles the `waffle` chart and populating it.

```
# Initialize the waffle chart as an empty matrix  
waffle_chart = np.zeros((height, width), dtype = np.uint)  
  
# Define indices to loop through waffle chart  
category_index = 0  
tile_index = 0
```

```
# Populate the waffle chart
for col in range(width):
    for row in range(height):
        tile_index += 1

# If the number of tiles populated for the current
# category is equal to its corresponding allocated tiles...
    if tile_index >
        sum(tiles_per_category[0:category_index]):
            # ...proceed to the next category
            category_index += 1

# Set the class value to an integer, which increases with
# class
        waffle_chart[row, col] = category_index
print ('Waffle chart populated!')
```

Waffle chart populated!

Let's take a peek at how the matrix looks like.

waffle_chart

As expected, the matrix consists of three categories and the total number of each category's instances matches the total number of tiles allocated to each category.

Step 5. Map the [waffle](#) chart matrix into a visual. The following code produces the outcome in [Figure 8-1](#).

```
# Instantiate a new figure object
set_pub()
fig = plt.figure()
```

```
# Use matshow to display the waffle chart
colormap = plt.cm.cool
plt.matshow(waffle_chart, cmap=colormap)
plt.colorbar()
plt.rcParams['grid', color='white']
plt.show()
```

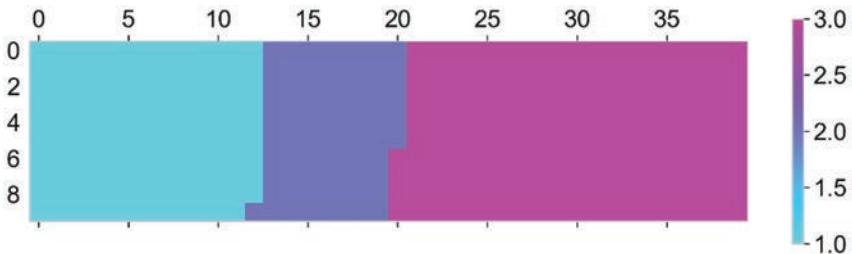


Figure 8-1. Waffle chart representing the proportion of immigration by Denmark, Norway, and Sweden to Canada

The code below produces the `waffle` chart with add grid-lines seen in **Figure 8-2**.

```
# Instantiate a new figure object
set_pub()
fig = plt.figure()
# Use matshow to display the waffle chart
colormap = plt.cm.Blues
plt.matshow(waffle_chart, cmap=colormap)
plt.colorbar()
plt.show()
```

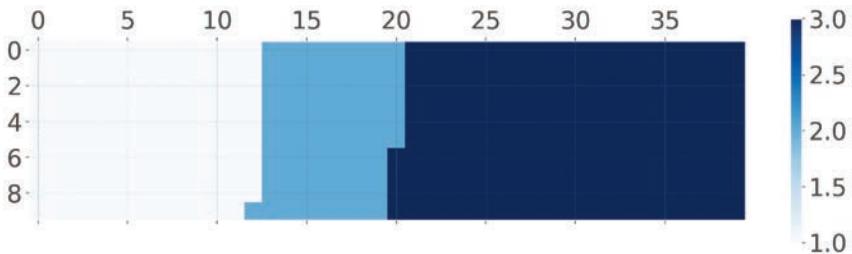


Figure 8-2. Waffle chart as in **Figure 8-1** with a colormap change and grid-lines true.

Step 6. Beautify the chart. The waffle chart with grid squares appears more as a waffle, perhaps with maple syrup. The chart is depicted in **Figure 8-3**.

```
# Instantiate a new figure object
set_pub()
fig = plt.figure()

# Use matshow to display the waffle chart
colormap = plt.cm.cool
plt.matshow(waffle_chart, cmap=colormap)
plt.colorbar()

# Get the axis
ax = plt.gca ()

# Set minor ticks
ax.set_xticks(np.arange (-0.5, (width), 1), minor=True)
ax.set_yticks(np.arange(-0.5, (height), 1), minor=True)

# Add gridlines based on minor ticks
ax.grid(which='minor', color='w', linestyle='-' ,
 linewidth=2)

plt.xticks([])
plt.yticks([])
plt.show()
```

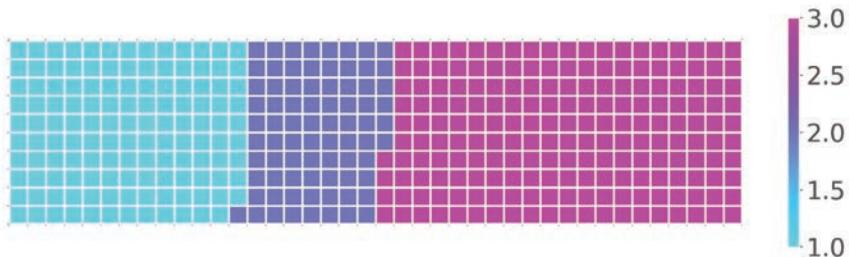


Figure 8-3. Waffle chart with grid-squares.

Step 7. Create a legend and add it to chart as seen in **Figure 8-4**.

```

# Instantiate a new figure object
set_pub()
fig = plt.figure()

# Use matshow to display the waffle chart
colormap = plt.cm.Blues
plt.matshow(waffle_chart, cmap=colormap)
plt.colorbar()

# Get the axis
ax = plt.gca ()

# Set minor ticks
ax.set_xticks(np.arange(-0.5, (width), 1), minor=True)
ax.set_yticks(np.arange(-0.5, (height), 1), minor=True)

# Add gridlines based on minor ticks
ax.grid(which='minor', color='steelblue', linestyle='-', linewidth=2)
plt.xticks([])
plt.yticks([])

# Compute cumulative sum of individual categories to match
# color schemes between chart and legend
values_cumsum = np.cumsum(df_dsn['Total'])
total_values = values_cumsum[len(values_cumsum) - 1]

# Create legend
legend_handles = []

for i, category in enumerate(df_dsn.index.values):
    label_str = category + ' (' + str(df_dsn['Total'][i])
    + ')'
    color_val =
        colormap(float(values_cumsum[i])/total_values)
    legend_handles.append(mpatches.Patch(color=color_val,
                                         label=label_str))

# Add Legend to chart
plt.legend(handles=legend_handles,
           loc='lower center',

```

```

        ncol=len(df_dsn.index.values),
        bbox_to_anchor= (0., -0.2, 0.95, .1)
    )
plt.show()

fig.savefig('myfig03', dpi=200)

```

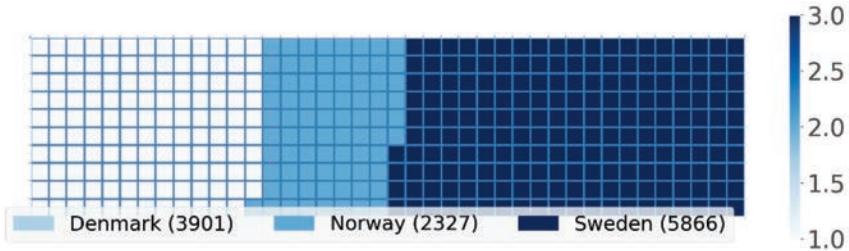


Figure 8-4. Repeat of the previous waffle chart with a colormap change and legend.

And there you go! What a good-looking *delicious* waffle chart, don't you think?

Now it would very inefficient to repeat these seven steps every time we wish to create a waffle chart. So, let's combine all seven steps into one function called *create_waffle_chart*. This function would take the following parameters as input:

1. **categories**: Unique categories or classes in dataframe.
2. **values**: Values corresponding to categories or classes.
3. **height**: Defined height of waffle chart.
4. **width**: Defined width of waffle chart.
5. **colormap**: Colormap class
6. **value_sign**: In order to make our function more generalizable, we will add this parameter to address signs that could be associated with a value such as %, \$, and so on. **value_sign** has a default value of empty string.

```

def create_waffle_chart(categories, values, height, width,
                        colormap, value_sign=''):

```

```

# Compute the proportion of each category with respect to
# the total
total_values = sum(values)
category_proportions = [(float(value) / total_values) for
value in values]

# Compute the total number of tiles
total_num_tiles = width * height # total number of tiles
print ('Total number of tiles is', total_num_tiles)

# Compute the number of tiles for each category
tiles_per_category = [round(proportion * total_num_tiles)
for proportion in category_proportions]

# Print out number of tiles per category
for i, tiles in enumerate(tiles_per_category):
    print (df_dsn.index.values[i] + ': ' + str(tiles))

# Initialize the waffle chart as an empty matrix
waffle_chart = np.zeros((height, width))

# Define indices to loop through waffle chart
category_index = 0
tile_index = 0

# Populate the waffle chart
for col in range(width):
    for row in range(height):
        tile_index += 1
        # If the number of tiles populated for the current
        # category
        # Is equal to its corresponding allocated tiles...
        if tile_index >
            sum(tiles_per_category[0:category_index]):
                # ...proceed to the next category
                category_index += 1

        # Set the class value to an integer, which
        # increases with class
        waffle_chart[row, col] = category_index

```

```

# Instantiate a new figure object
fig = plt.figure()

# Use matshow to display the waffle chart
colormap = plt.cm.coolwarm
plt.matshow(waffle_chart, cmap=colormap)
plt.colorbar()

# Get the axis
ax = plt.gca ()

# Set minor ticks
ax.set_xticks(np.arange (-0.5, (width), 1), minor=True)
ax.set_yticks(np.arange(-0.5, (height), 1), minor=True)

# Add gridlines based on minor ticks
ax.grid(which='minor', color='w', linestyle='-', linewidth=2)

plt.xticks([])
plt.yticks([])

# Compute cumulative sum of individual categories to match
# color schemes between chart and legend
values_cumsum = np.cumsum(values)
    total_values = values_cumsum[len(values_cumsum) - 1]

# Create legend
legend_handles = []
    for i, category in enumerate(categories):
        if value_sign == '%':
            label_str = category + ' (' + str(values[i]) +
                value_sign + ')'
        else:
            label_str = category + ' (' + value_sign +
                str(values[i]) + ')'
        color_val =
colormap(float(values_cumsum[i])/total_values)

legend_handles.append(mpatches.Patch(color=color_val,
label=label_str))

```

```

# Add Legend to chart
plt.legend(
    handles=legend_handles,
    loc='lower center',
    ncol=len(categories),
    bbox_to_anchor=(0., -0.2, 0.95, .1)
)
plt.show()

```

Now to create a waffle chart, all we have to do is call the function `create_waffle_chart`. Let's define the input parameters:

```

width = 40 # width of chart
height = 10 # height of chart

categories = df_dsn.index.values # categories
values = df_dsn['Total'] # corresponding values of
# categories

colormap = plt.cm.cool # color map class

fig.savefig('myfig04', dpi=200)

```

And now let's call our function to create the waffle chart in shown **Figure 8-5**.

```
create_waffle_chart(categories, values, height, width,
colormap)
```

Total number of tiles is 400

Denmark: 129

Norway: 77

Sweden: 194

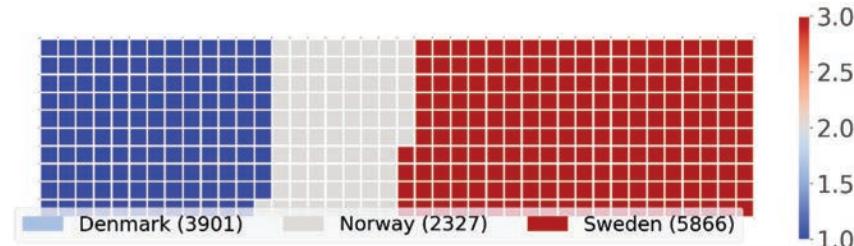


Figure 8-5. Waffle chart resulting for the newly defined algorithm

There seems to be a new Python package for generating waffle charts called PyWaffle, but it looks like the repository is still being built. But feel free to check it out and play with it.

Word Clouds

Word clouds (also known as text clouds or tag clouds) work in a simple way: the more a specific word appears in a source of textual data (such as a speech, blog post, or database), the bigger and bolder it appears in the word cloud.

Luckily, a Python package already exists in Python for generating word clouds. The package, called `wordcloud` was developed by **Andreas Mueller**. You can learn more about the package by following this [link](#).

Let's use this package to learn how to generate a word cloud for a given text document.

First, let's install the package.

```
# Install wordCloud
! pip install wordcloud

# Import package and its set of stopwords
from wordcloud import WordCloud, STOPWORDS

print ('Wordcloud is installed and imported!')
```

Word clouds are commonly used to perform high-level analysis and visualization of text data. Accordingly, let's digress from the immigration dataset and work with an example that involves analyzing text data. Let's try to analyze a short novel written by **Lewis Carroll** titled *Alice's Adventures in Wonderland*. Let's go ahead and download a `.txt` file of the novel.

```
import urllib

# Open the file and read it into a variable alice_novel
alice_novel = urllib.request.urlopen('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DV0101EN-SkillsNetwork/Data%20Files/alice_novel.txt').read().decode("utf-8")
```

Next, let's use the stopwords that we imported from word_cloud. We use the function `set` to remove any redundant stopwords.

```
stopwords = set(STOPWORDS)
```

Create a word cloud object and generate a word cloud. For simplicity, let's generate a word cloud using only the first 2000 words in the novel.

```
# Instantiate a word cloud object
alice_wc = WordCloud(
    background_color='white',
    max_words=2000,
    stopwords=stopwords
)
# Generate the word cloud
alice_wc.generate(alice_novel)
```

Awesome! Now that the word cloud is created, let's visualize it in **Figure 8-6**.

```
# Display the word cloud
plt.imshow(alice_wc, interpolation='bilinear')
plt.axis('off')
plt.show()
```

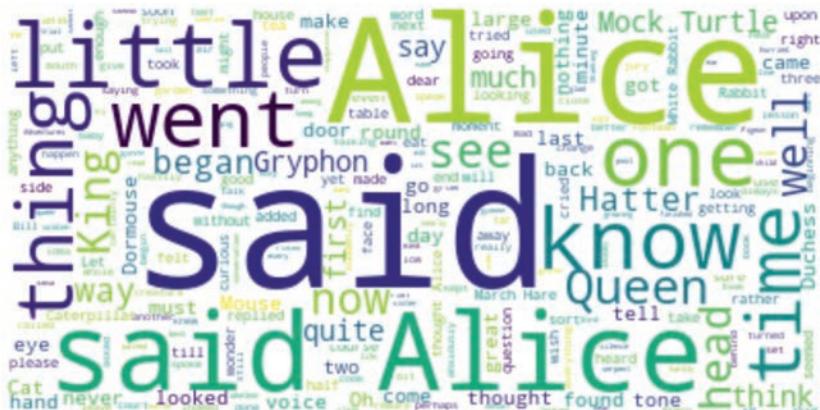


Figure 8-6. Word cloud of Lewis Carroll's *Alice in Wonderland*

Interesting! So, in the first 2000 words in the novel, the most common words are **Alice**, **said**, **little**, **Queen**, and so on. Let's resize the cloud so that we can see the less frequent words a little better, as shown in **Figure 8-7**.

```
# Display the cloud
plt.imshow(alice_wc, interpolation='bilinear')
plt.axis('off')
plt.show()
```



Figure 8-7. Word cloud of Lewis Carroll's *Alice in Wonderland* with more words

Much better! However, **said** isn't really an informative word. So, let's add it to our stopwords and re-generate the cloud in **Figure 8-8**.

```
stopwords.add('said') # add the words said to stopwords
# re-generate the word cloud
alice_wc.generate(alice_novel)
# Display the cloud
fig = plt.figure(figsize= (14, 18))
plt.imshow(alice_wc, interpolation='bilinear')
plt.axis('off')
plt.show()
```



Figure 8-8. Word cloud of Lewis Carroll's Alice in Wonderland with stopwords removed

Excellent! This looks really interesting! Another cool thing you can implement with the `word_cloud` package is superimposing the words onto a mask of any shape. Let's use a mask of Alice and her rabbit. We already created the mask for you, so let's go ahead and download it and call it `alice_mask.png` shown in **Figure 8-9**

```
# Save mask to alice_mask
alice_mask = np.array
(Image.open(urllib.request.urlopen('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDriverSkillsNetwork-DV0101EN-SkillsNetwork/labs/Module%204/images/alice_mask.png')))
```

Let's take a look at how the mask looks like.

```
plt.imshow(alice_mask, cmap=plt.cm.gray, interpolation='bilinear')
plt.axis('off')
plt.show()
```



Figure 8-9. Mask of Alice and the rabbit

Shaping the word cloud according to the mask is straightforward using `word_cloud` package. For simplicity, we will continue using the first 2000 words in the novel. This is displayed in **Figure 8-10**.

```
# Instantiate a word cloud object
alice_wc = WordCloud(background_color='white',
max_words=2000, mask=alice_mask, stopwords=stopwords)

# Generate the word cloud
alice_wc.generate(alice_novel)

# Display the word cloud
fig = plt.figure(figsize= (14, 18))

plt.imshow(alice_wc, interpolation='bilinear')
plt.axis('off')
plt.show()
```

Really impressive!

Unfortunately, our immigration data does not have any text data, but where there is a will there is a way. Let's generate sample text data from our immigration dataset, say text data of 90 words.

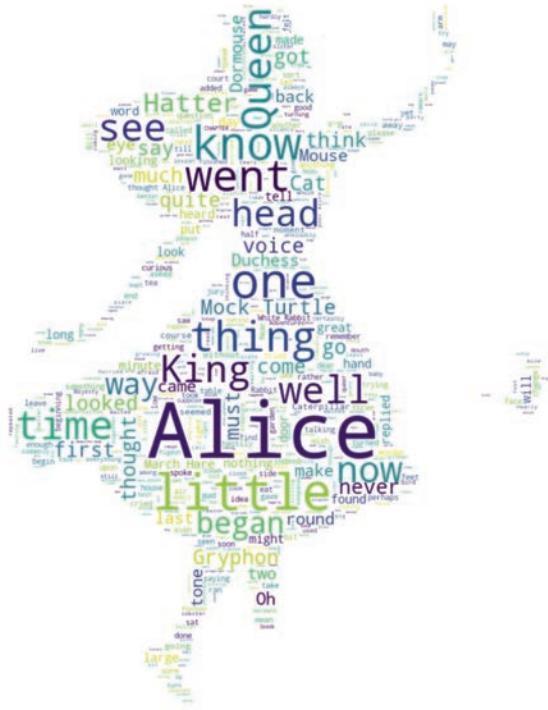


Figure 8-10. Word cloud according to the mask of Alice and the rabbit

Let's recall how our data looks like.

df_can.head()						
	Continent	Region	DevName	1980 ...	2013	Total
Country						
Afghanistan	Asia	Southern Asia	Developing regions	16	...	200458639
Albania	Europe	Southern Europe	Developed regions	1	... 603	15699
Algeria	Africa	Northern Africa	Developing regions	80	... 433169439	
American Samoa	Oceania	Polynesia	Developing regions	0	... 0	6
Andorra	Europe	Southern Europe	Developed regions	0	... 1	15

5 rows × 38 columns

And what was the total immigration from 1980 to 2013?

```
total_immigration = df_can['Total'].sum()  
total_immigration
```

6409153

Using countries with single-word names, let's duplicate each country's name based on how much they contribute to the total immigration.

```
max_words = 90  
word_string = ''  
  
for country in df_can.index.values:  
    # Check if country's name is a single-word name  
    if country.count(" ") == 0:  
        repeat_num_times = int(df_can.loc[country,  
'Total'] / total_immigration * max_words)  
        word_string = word_string + ((country + ' ') *  
repeat_num_times)  
  
# Display the generated text  
word_string
```

'China China China China China China China China
Colombia Egypt France Guyana Haiti India India India India
India India India India India Jamaica Lebanon Morocco
Pakistan Pakistan Pakistan Philippines Philippines
Philippines Philippines Philippines Philippines
Philippines Poland Portugal Romania '

We are not dealing with any stopwords here, so there is no need to pass them when creating the word cloud.

```
# Create the word cloud  
wordcloud =  
WordCloud(background_color='white').generate(word_string)  
print('Word cloud created!')
```

Word cloud created!

We now display it in **Figure 8-11**.

```
# Display the cloud  
plt.imshow(wordcloud, interpolation='bilinear')  
plt.axis('off')  
plt.show()
```

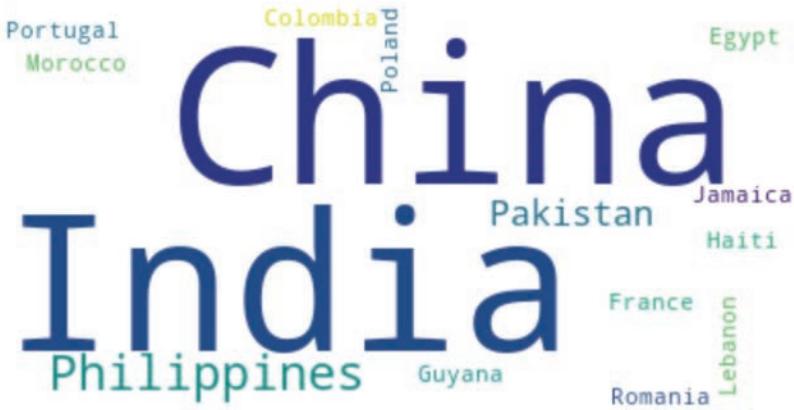


Figure 8-11. Word cloud using countries with single-word names, let's duplicate each country's name based on how much they contribute to the total immigration.

According to the above word cloud, it looks like the majority of the people who immigrated came from one of 15 countries that are displayed by the word cloud. One cool visual that you could build, is perhaps using the map of Canada and a mask and superimposing the word cloud on top of the map of Canada. That would be an interesting visual to build!

Regression Plots

Seaborn is a Python visualization library based on matplotlib. It provides a high-level interface for drawing attractive statistical graphics. You can learn more about *seaborn* and *seaborn* regression plots at <https://seaborn.pydata.org/>.

In this chapter *Pie Charts, Box Plots, Scatter Plots, and Bubble Plots*, we learned how to create a scatter plot and then fit a regression line. It took ~20 lines of code to create the scatter plot along with the regression fit. In this final section, we will explore *seaborn* and see

how efficient it is to create regression lines and fits using this library!

Let's first install *seaborn*

```
# Install seaborn
! pip install seaborn

# Import Library
import seaborn as sns

print('Seaborn installed and imported!')
```

Create a new dataframe that stores that total number of landed immigrants to Canada per year from 1980 to 2013. Create a new dataframe that stores that total number of landed immigrants to Canada per year from 1980 to 2013.

```
# We can use the sum() method to get the total population
per year

df_tot = pd.DataFrame (df_can[years].sum(axis=0))

# Change the years to type float (useful for regression
# later on)
df_tot.index = map(float, df_tot.index)

# Reset the index to put in back in as a column in the
# df_tot dataframe
df_tot.reset_index (inplace=True)

# Rename columns
df_tot.columns = ['year', 'total']

# View the final dataframe
df_tot.head()
```

	year	total
0	1980.0	99137
1	1981.0	110563
2	1982.0	104271
3	1983.0	75550
4	1984.0	73417

With *seaborn*, generating a regression plot is as simple as calling the **regplot** function. We view the result in **Figure 8-12**.

```
set_pub()
sns.set(rc={"figure.dpi":300, 'savefig.dpi':300})
plt.figure(figsize=(width, height))
plt.figure(figsize=(width, height))
sns.regplot(x='year', y='total', data=df_tot)
plt.savefig('myfig11', dpi=200)
```

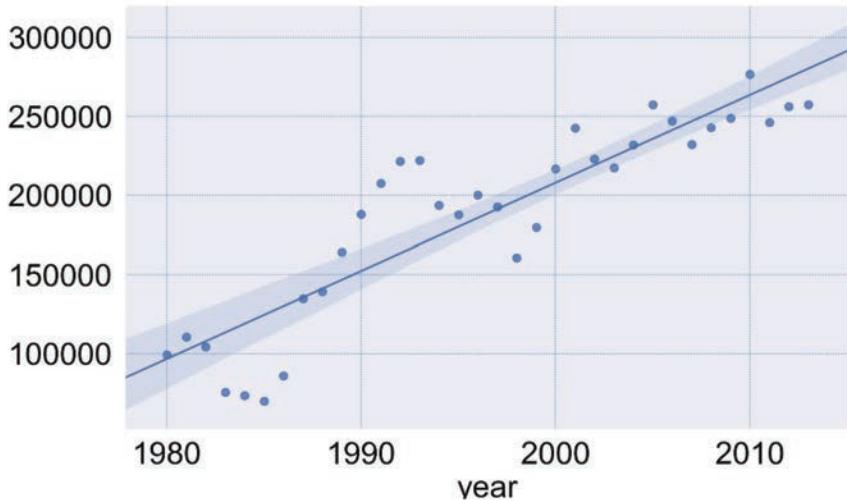


Figure 8-12. Regression plot of the total number of landed immigrants to Canada per year from 1980 to 2013

We can also customize the color of the scatter plot and regression line. So, let's change the color to green as seen in **Figure 8-13**.

```
sns.set(rc={"figure.dpi":300, 'savefig.dpi':300})
sns.regplot(x='year', y='total', data=df_tot, color='green')
plt.show()
```

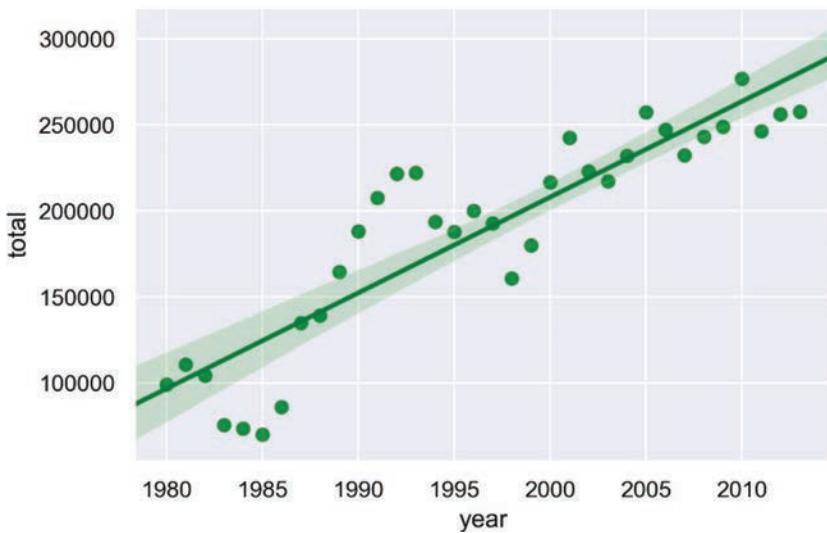


Figure 8-13. Subsequent regression, with color change, of the total number of landed immigrants to Canada per year from 1980 to 2013

Now let's change the markers from circles to crosses. This plot is shown in **Figure 8-14**.

```
ax = sns.regplot(x='year', y='total', data=df_tot,
color='green', marker='+')
plt.show()
```

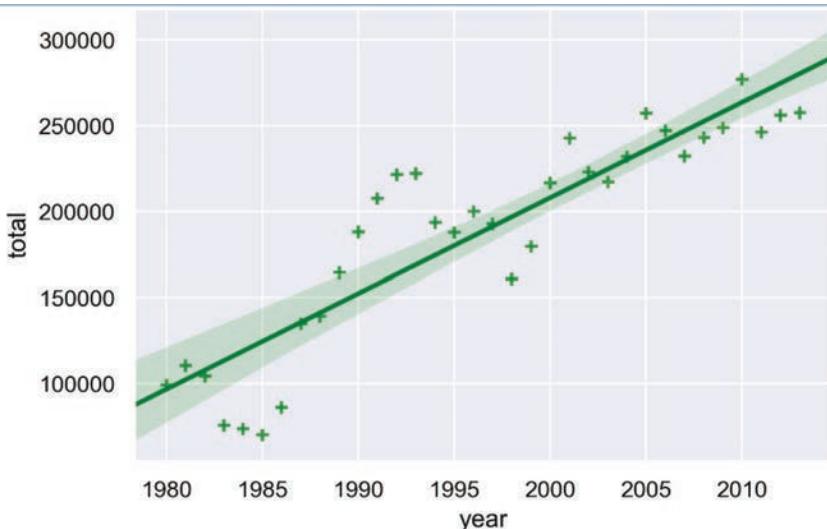


Figure 8-14. Subsequent regression, with marker-shape change of the total number of landed immigrants to Canada per year from 1980 to 2013

Let's blow up the plot a little so that it is more appealing to the sight. The result is shown in **Figure 8-15**.

```
plt.figure(figsize=(15, 10))
sns.regplot(x='year', y='total', data=df_tot,
color='green', marker='+')
plt.show()
fig.savefig('myfig14', dpi=200)
```

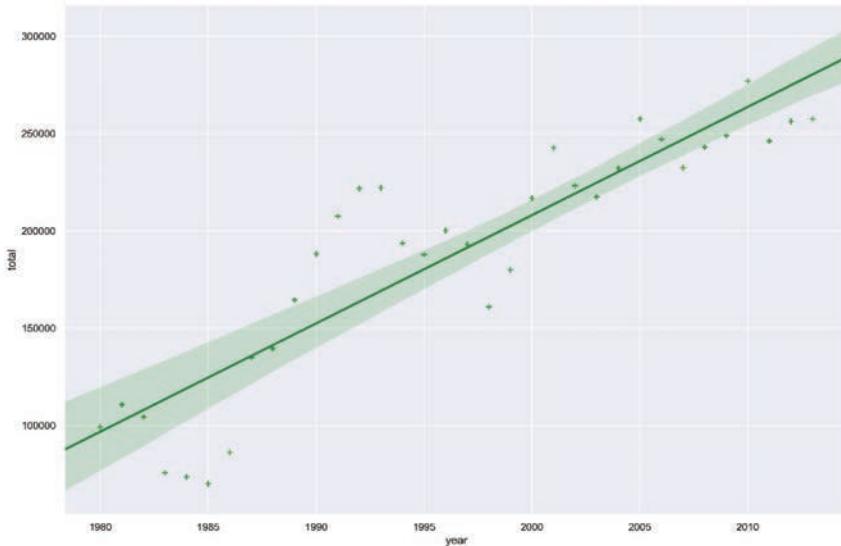


Figure 8-15. Subsequent regression plot, with size change, of the total number of landed immigrants to Canada per year from 1980 to 2013

And let's increase the size of markers so they match the new size of the figure, and add a title and x- and y-labels. The result is shown in **Figure 8-16**.

```
plt.figure(figsize= (15, 10))
ax = sns.regplot(x='year', y='total', data=df_tot,
color='green', marker='+', scatter_kws={'s': 200})

ax.set(xlabel='Year', ylabel='Total Immigration') # add x-
# and y-labels
ax.set_title('Total Immigration to Canada from 1980 - 2013') # add title
plt.show()
fig.savefig('myfig15', dpi=200)
```

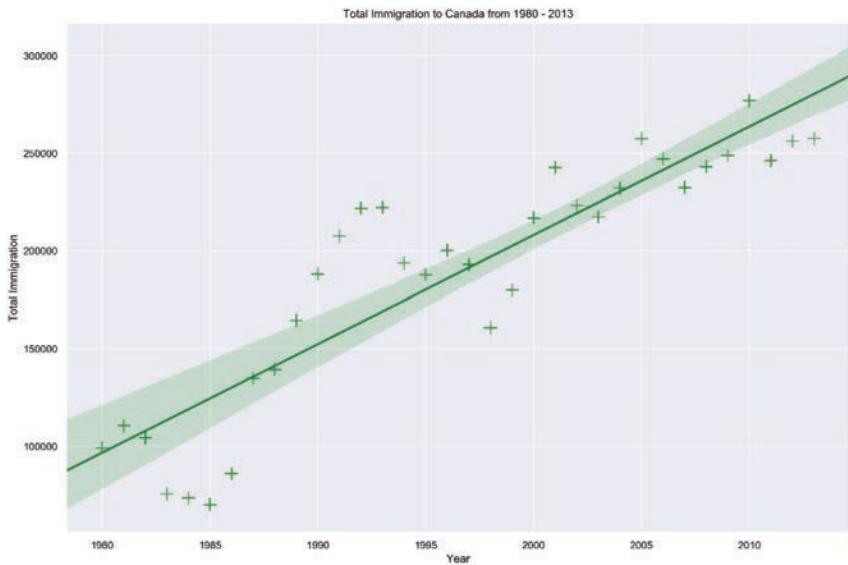


Figure 8-16. Subsequent regression plot, with size change and marker size change, of the total number of landed immigrants to Canada per year from 1980 to 2013

Finally, we'll increase the font size of the tickmark labels, the title, and the x- and y-labels so they don't feel barely discernable! The result is shown in **Figure 8-17**. Note that the effect can also be realized by decreasing the dpi, say from 200 to 120, but we seldom want to decrease the resolution of a plot.

```
plt.figure(figsize=(15, 10))

sns.set(font_scale=1.5)

ax = sns.regressionplot(x='year', y='total', data=df_tot,
color='green', marker='+', scatter_kws={'s': 200})
ax.set(xlabel='Year', ylabel='Total Immigration')
ax.set_title('Total Immigration to Canada from 1980 - 2013')

plt.show()
fig.savefig('myfig16', dpi=200)
```

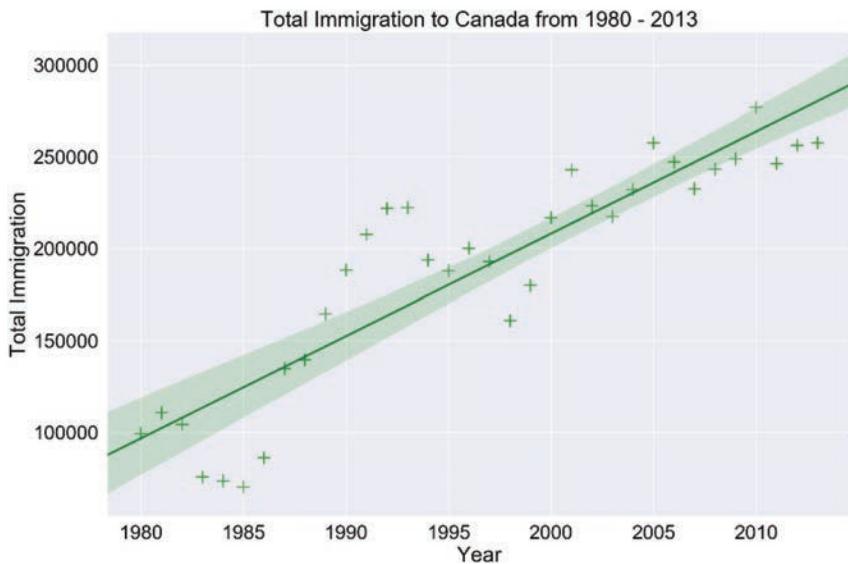


Figure 8-17. Subsequent regression plot, with size change and marker size change, and overall font size change, of the total number of landed immigrants to Canada per year from 1980 to 2013

Amazing! A complete scatter plot with a regression fit with 5 lines of code only. Isn't this really amazing?

If you are not a big fan of the purple background, you can easily change the style to a white plain background. The result is shown in **Figure 8-18**.

```
plt.figure(figsize=(15, 10))

sns.set(font_scale=2.5)
sns.set_style('ticks') # change background to white
background

ax = sns.regplot (x='year', y='total', data=df_tot,
color='green', marker='+', scatter_kws={'s': 200})
ax.set(xlabel='Year', ylabel='Total Immigration')
ax.set_title('Total Immigration to Canada from 1980-2013')

plt.show()
fig.savefig('myfig17', dpi=200)
```

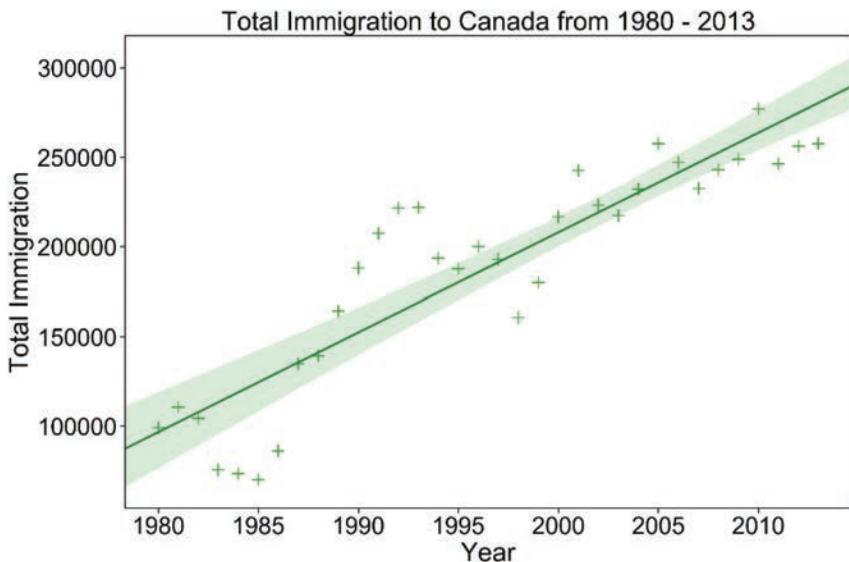


Figure 8-18. Subsequent regression plot, with background change, of the total number of landed immigrants to Canada per year from 1980 to 2013

We can also take our white background and add gridlines. The result is shown in **Figure 8-19**.

```
plt.figure(figsize= (15, 10))

sns.set(font_scale=1.5)
sns.set_style('whitegrid')

ax = sns.regplot(x='year', y='total', data=df_tot,
color='green', marker='+', scatter_kws={'s': 200})
ax.set(xlabel='Year', ylabel='Total Immigration')
ax.set_title('Total Immigration to Canada from 1980 - 2013')

plt.show()
fig.savefig('myfig18', dpi=200)
```

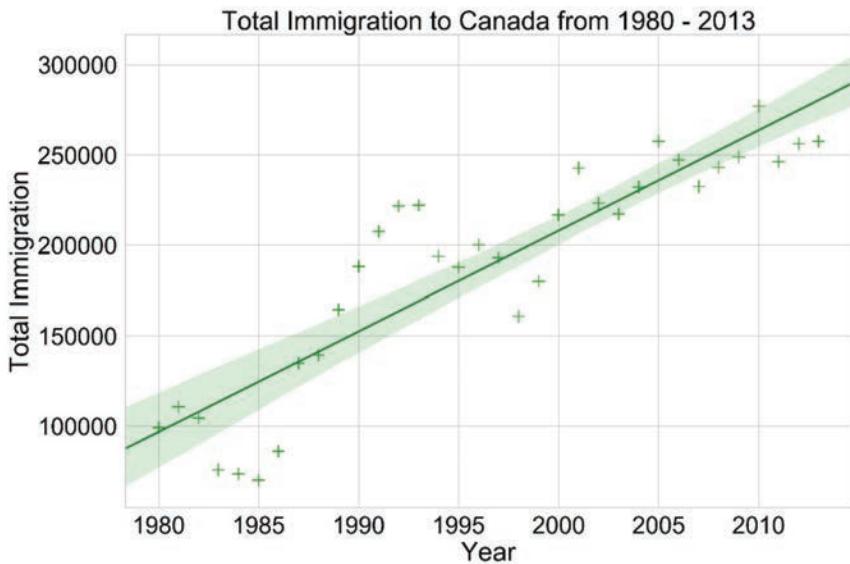


Figure 8-19. Subsequent regression plot, with grid-lines added, of the total number of landed immigrants to Canada per year from 1980 to 2013

Example: Use seaborn to create a scatter plot with a regression line to visualize the total immigration from Denmark, Sweden, and Norway to Canada from 1980 to 2013. The result is shown in **Figure 8-20**.

```
# Create df_countries dataframe
df_countries = df_can.loc[['Denmark', 'Norway',
'Sweden'], years].transpose()

# Create df_total by summing across three countries for
each year
df_total = pd.DataFrame (df_countries.sum(axis=1))

# Reset index in place
df_total.reset_index (inplace=True)

# Rename columns
df_total.columns = ['year', 'total']

# Change column year from string to int to create scatter
plot
df_total['year'] = df_total['year'].astype (int)
```

```

# Define figure size
plt.figure(figsize= (15, 10))

# Define background style and font size
sns.set(font_scale=2.5)
sns.set_style('whitegrid')

# Generate plot and add title and axes Labels
ax = sns.regplot (x='year', = 'total', data=df_total,
color='green', marker='+', scatter_kws={'s': 200})
ax.set(xlabel='Year', ylabel='Total Immigration')
ax.set_title('Total Immigration from Denmark, Sweden, and
Norway\n to Canada from 1980 - 2013')

```

Text(0.5, 1.0, 'Total Immigration from Denmark, Sweden,
and Norway\n to Canada from 1980 - 2013')

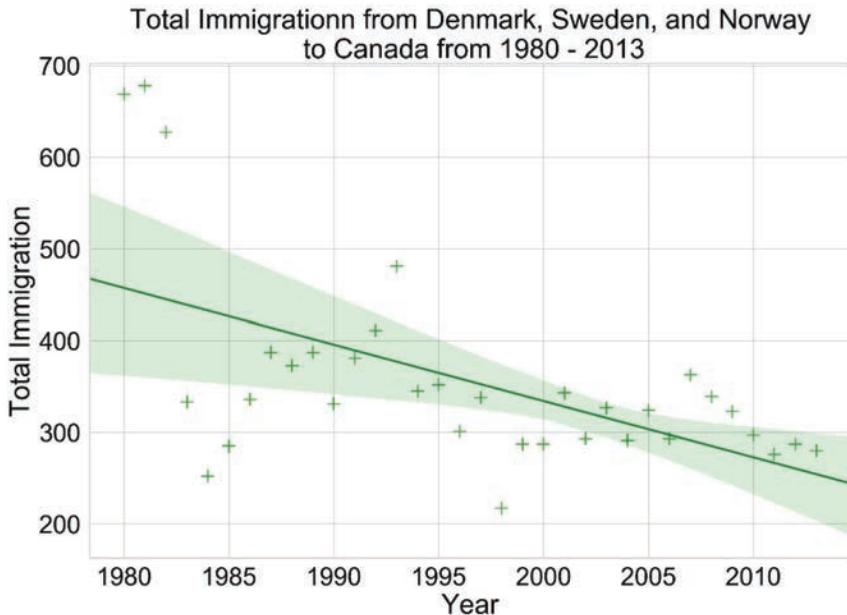


Figure 8-20. Scatter plot with a regression line to visualize the total immigration from Denmark, Sweden, and Norway to Canada from 1980 to 2013

9. Simple Linear Regression

Objectives

After completing this chapter, you will be able to:

- Use scikit-learn to implement simple Linear Regression
- Create a model, train it, test it and use the model

Importing Needed packages

```
import matplotlib.pyplot as plt  
import pandas as pd  
import pylab as pl  
import numpy as np  
%matplotlib inline
```

Downloading Data

To download the data, we will use the raw path to my Jupyter repository on [GitHub](#). The path is given below. If you want to download the .csv file, then click on or enter the url in your browser:
<https://raw.githubusercontent.com/stricje1/jupyter/main/data/MY2020Fuel Consumption Ratings.csv>

```
path = "  
https://raw.githubusercontent.com/stricje1/jupyter/main/da  
ta/MY2020%20Fuel%20Consumption%20Ratings.csv"
```

Understanding the Data

We have downloaded a fuel consumption dataset, [MY2020 Fuel Consumption Ratings.csv](#), which contains model-specific fuel consumption ratings and estimated carbon dioxide emissions for new light-duty vehicles for retail sale in Canada. [Dataset source](#):
<https://open.canada.ca/data/en/dataset/98f1a129-f628-4ce4-b24d-6f16bf24dd64>

- **MODELYEAR** e.g., 2014
- **MAKE** e.g., Acura
- **MODEL** e.g., ILX
- **VEHICLE CLASS** e.g., SUV

- **ENGINE SIZE** e.g., 4.7
- **CYLINDERS** e.g., 6
- **TRANSMISSION** e.g., A6
- **FUEL CONSUMPTION in CITY(L/100 km)** e.g., 9.9
- **FUEL CONSUMPTION in HWY (L/100 km)** e.g., 8.9
- **FUEL CONSUMPTION COMB (L/100 km)** e.g., 9.2
- **CO2 EMISSIONS (g/km)** e.g., 182 --> low --> 0

Reading the in the Data

As usual, we read the data into our **Jupyter notebook** using **pandas read_csv()** function:

```
df = pd.read_csv(path)
df.head()
```

							FUELCONS
	MODEL	YEAR	MAKE	VEHICLECL	ENGINE	UMPTION	CO2EMI
				MODEL	ASS	SIZE	... COMB_MPG SSIONS
0	2014	ACURA		ILX	COMPACT	2.0	... 33 196
1	2014	ACURA		ILX	COMPACT	2.4	... 29 221
2	2014	ACURA	ILX	HYBRID	COMPACT	1.5	... 48 136
3	2014	ACURA	MDX	4WD SUV-SMALL		3.5	... 25 255
4	2014	ACURA	RDX	AWD SUV-SMALL		3.5	... 27 244

Data Exploration

Let's first have a descriptive exploration on our data.

```
# Summarize the data
df.describe()
```

					FUELCON		FUELCONS
	MODEL	YE	ENGINES	CYLINDE	SUMPTIO	UMPTION_	CO2EMISS
	AR	IZE	RS	N_CITY	... COMB_MPG		IONS
count	1067.0	1067.00	1067.00	1067.00	...	1067.00	1067.00
mean	2014.0	3.35	5.80	13.30	...	26.44	256.23
std	0.0	1.42	1.80	4.10	...	7.47	63.37
min	2014.0	1.00	3.00	4.60	...	11.00	108.00
25%	2014.0	2.00	4.00	10.25	...	21.00	207.00
50%	2014.0	3.40	6.00	12.60	...	26.00	251.00
75%	2014.0	4.30	8.00	15.55	...	31.00	294.00
max	2014.0	8.40	12.00	30.20	...	60.00	488.00

Let's select some features to explore more.

```
cdf =  
df[['ENGINESIZE', 'CYLINDERS', 'FUELCONSUMPTION_COMB', 'CO2Emissions']]  
cdf.head(9)
```

			FUELCONSUMPTION_C	
	ENGINESIZE	CYLINDERS	OMB	CO2EMISSIONS
0	2.0	4	8.5	196
1	2.4	4	9.6	221
2	1.5	4	5.9	136
3	3.5	6	11.1	255
4	3.5	6	10.6	244
5	3.5	6	10.0	230
6	3.5	6	10.1	232
7	3.7	6	11.1	255
8	3.7	6	11.6	267

We plot each of these features using the `hist()` function and display the results in **Figure 9-1**:

```
fig, ax1 = plt.subplots()  
  
# Designate multiple plots by feature name  
viz =  
cdf[['CYLINDERS', 'ENGINESIZE', 'CO2EMISSIONS', 'FUELCONSUMPTION_COMB']]  
  
# Plot multiple histograms in grids  
ax = viz.hist (figsize=(6,10), ax=ax1)  
  
# Default font size  
plt.rc('font', size=12)  
  
# Picture size  
ax = plt.figure(dpi=200)  
  
# Buffer between Labels  
fig.tight_layout()  
plt.show()  
  
fig.savefig('outfig_01', dpi=200)
```

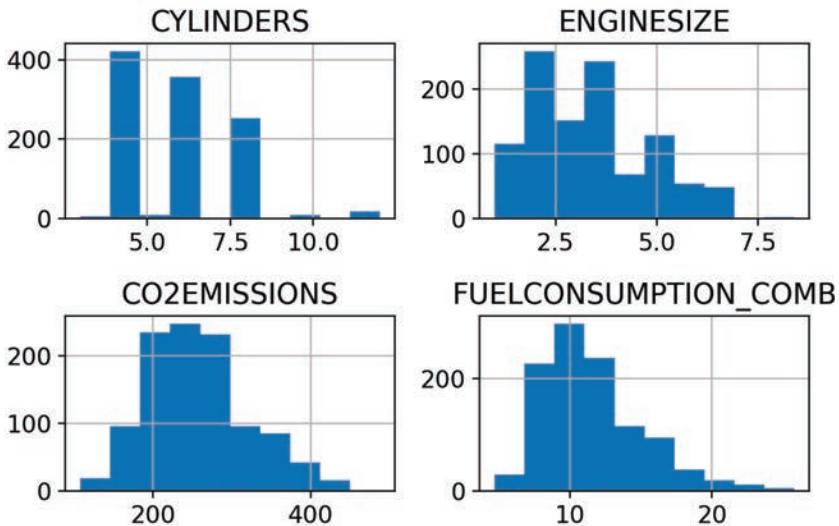


Figure 9-1. Histograms for our four chosen features from the emissions data.

Now, let's plot each of these features against the Emission, to see how linear their relationship is. The plots appear in **Figure 9-2**.

```
# Set the inline plot resolution
plt.rcParams ['figure.dpi'] = 200
fig=plt.figure()

plt.scatter(cdf.FUELCONSUMPTION_COMB, cdf.CO2EMISSIONS,
color='blue')

# plt.figure(figsize=(8, 10)), over-ridden by dpi setting
# User-defined default font size
plt.rc ('font', size=18)

#Figure labels and padding
plt.xlabel("FUELCONSUMPTION_COMB")
plt.ylabel("Emission")
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)

# Display and save the figure
plt.show()
fig.savefig('outfig_02')
```

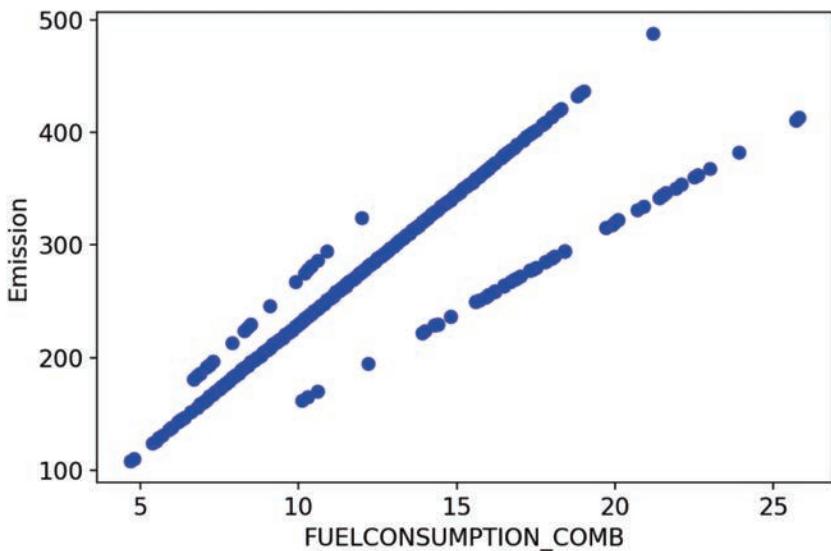


Figure 9-2. Scatter of the combined city and highway mileage vs. CO^2 emissions.

Now, let's look at engine size vs. CO^2 emissions with a scatterplot, shown in **Figure 9-3**.

```
fig = plt.figure()

# Scatterplot defined for engine size & CO2 emissions
plt.scatter(cdf.ENGINESIZE, cdf.CO2EMISSIONS,
            color='blue')

# Set labels and label/figure padding
plt.xlabel("Engine size")
plt.ylabel("Emission")
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)

# Display and save the figure as a PNG file (default)
plt.show()
fig.savefig('outfig_03', dpi=200)
```

If you ever wonder what directory your files are being saved in, query **Jupyter Notebook** or **JupyterLab** for the working directory:

```
os.getcwd()
```

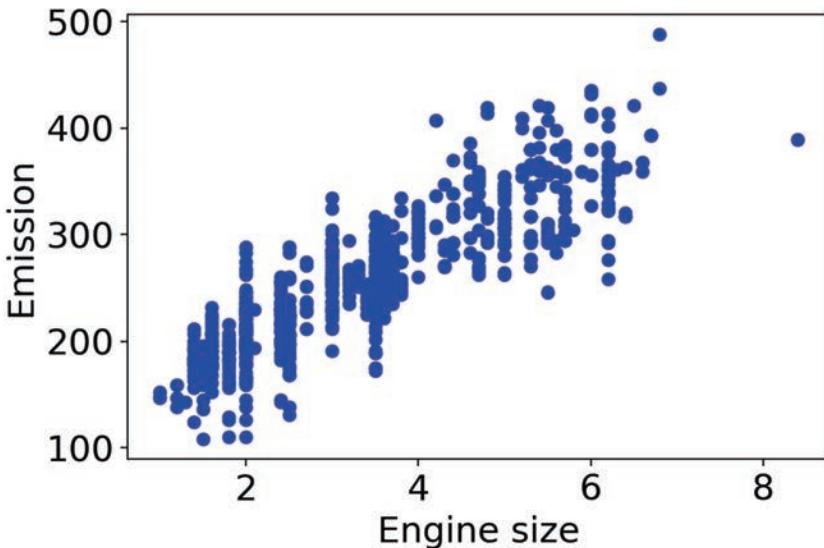


Figure 9-3. Scatter of the combined city and engine size vs. CO₂ emissions.

Practice

Plot CYLINDER vs the Emission, to see how linear is their relationship is. This plot is displayed in **Figure 9-4**.

```
plt.scatter(cdf.CYLINDERS, cdf.CO2EMISSIONS,  
           color='blue')  
plt.xlabel("Cylinders")  
plt.ylabel("Emission")  
plt.show()  
fig.savefig('outfig_04', dpi=200)
```

When you save a **Jupyter Notebook** as **Markdown**, it generates a ZIP file with the markdown file (.md), and all of the output files, like plots. To get high resolution pictures (the output), be sure to set your image resolution to 200 – 600 dpi. For a 6" × 9" trade book, I use between 200 and 400 dpi when publishing. For blogging, the images can be between 96 to 120 dpi, most of the time. The resolution depends on the graphic you're saving.

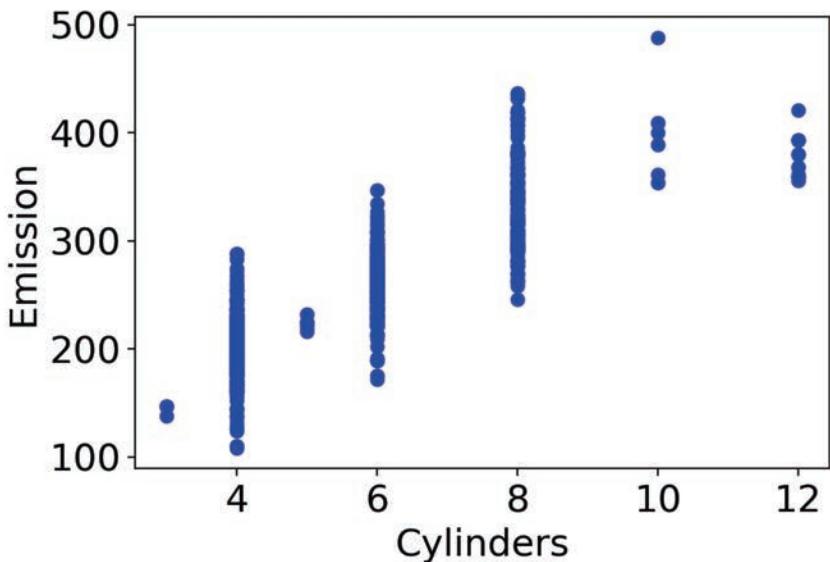


Figure 9-4. Scatter of the combined city and number of cylinders vs. CO₂ emissions.

Creating train and test dataset

Train/Test Split involves splitting the dataset into training and testing sets that are mutually exclusive. After which, you train with the training set and test with the testing set. This will provide a more accurate evaluation on out-of-sample accuracy because the testing dataset is not part of the dataset that have been used to train the model. Therefore, it gives us a better understanding of how well our model generalizes on new data.

This means that we know the outcome of each data point in the testing dataset, making it great to test with! Since this data has not been used to train the model, the model has no knowledge of the outcome of these data points. So, in essence, it is truly an out-of-sample testing.

Let's split our dataset into train and test sets. 80% of the entire dataset will be used for training and 20% for testing. We create a mask to select random rows using `np.random.rand()` function:

```
msk = np.random.rand(len(df)) < 0.8
train = cdf[msk]
test = cdf[~msk]
```

Simple Regression Model

Linear Regression fits a linear model with coefficients $B = (B_1, \dots, B_n)$ to minimize the '**residual sum of squares**' between the actual value y in the dataset, and the predicted value \hat{y} using linear approximation.

Train data distribution

So, let's take a look at a visual for `ENGINESIZE` versus `CO2EMISSIONS`, which we show as a scatterplot in **Figure 9-5**.

```
plt.scatter(train.ENGINESIZE, train.CO2EMISSIONS,
color='blue')
plt.xlabel('Engine size')
plt.ylabel('Emission')
plt.show()
fig.savefig('outfig_05', dpi=200)
```

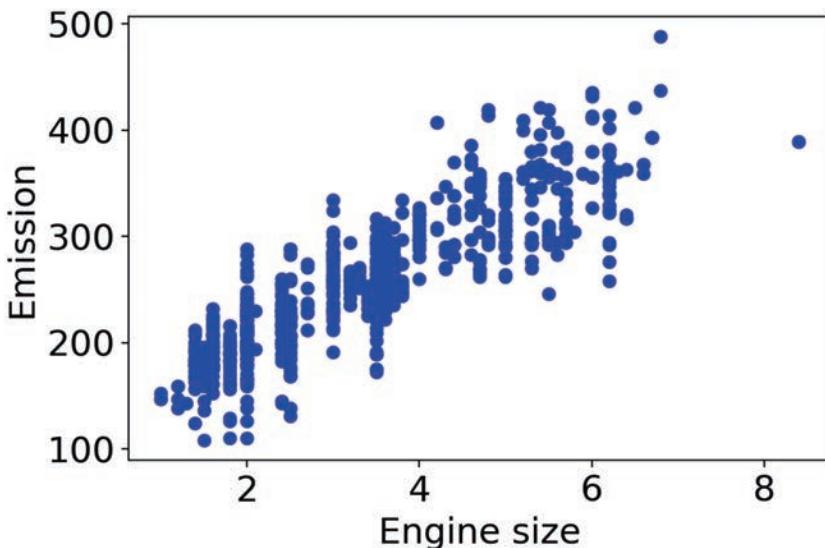


Figure 9-5. Scatterplot of Engine Size vs. CO² Emissions

Modeling

Using sklearn package to model data.

```
from sklearn import linear_model
regr = linear_model.LinearRegression()
train_x = np.asarray(train[['ENGINESIZE']])
train_y = np.asarray(train[['CO2EMISSIONS']])

regr.fit (train_x, train_y)

# The coefficients
print ('Coefficients: ', regr.coef_)
print ('Intercept: ',regr.intercept_)
```

Coefficients: [[38.82972164]]

Intercept: [125.78534598]

As mentioned before, **Coefficient** and **Intercept** in the simple linear regression, are the parameters of the fit line. Given that it is a simple linear regression, with only 2 parameters, and knowing that the parameters are the intercept and slope of the line, sklearn can estimate them directly from our data. Notice that all of the data must be available to traverse and calculate the parameters.

Plot outputs

We can plot the fit line over the data in **Figure 9-6**:

```
#fig = plt.subplots()
plt.scatter(train.ENGINESIZE, train.CO2EMISSIONS,
            color='blue')
plt.plot(train_x, regr.coef_[0][0]*train_x +
         regr.intercept_[0], '-r')

plt.xlabel('Engine size')
plt.ylabel('Emission')
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)

plt.savefig('outfig_06', dpi=200)
```

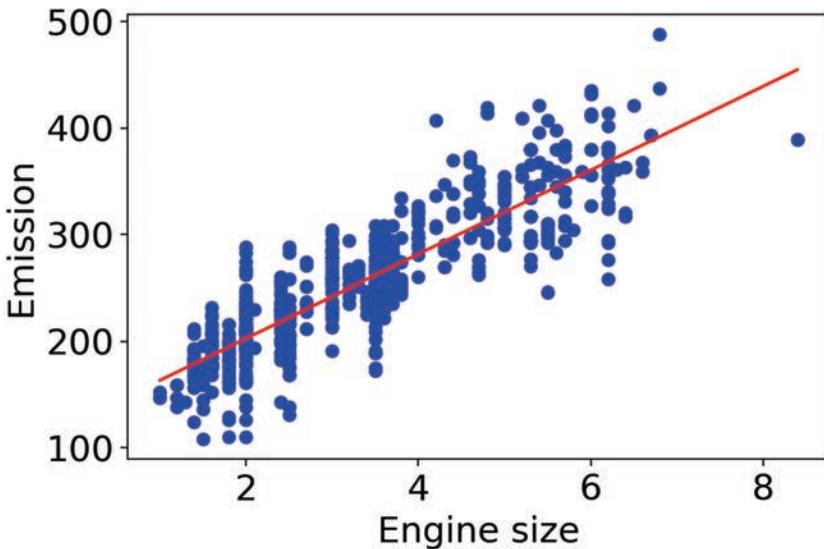


Figure 9-6. Scatterplot of engine size versus CO^2 emissions with a fitted line

Evaluation

We compare the actual values and predicted values to calculate the accuracy of a regression model. Evaluation metrics provide a key role in the development of a model, as it provides insight to areas that require improvement.

There are different model evaluation metrics, let's use MSE here to calculate the accuracy of our model based on the test set:

- Mean Absolute Error: It is the mean of the absolute value of the errors. This is the easiest of the metrics to understand since it's just average error.
- Mean Squared Error (MSE): Mean Squared Error (MSE) is the mean of the squared error. It's more popular than Mean Absolute Error because the focus is geared more towards large errors. This is due to the squared term exponentially increasing larger errors in comparison to smaller ones.
- Root Mean Squared Error (RMSE).
- R-squared is not an error, but rather a popular metric to measure the performance of your regression model. It represents how

close the data points are to the fitted regression line. The higher the R-squared value, the better the model fits your data. The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse).

```
from sklearn.metrics import r2_score

test_x = np.asarray(test[['ENGINESIZE']])
test_y = np.asarray(test[['CO2EMISSIONS']])
test_y_ = regr.predict (test_x)

print('Mean absolute error: %.2f' %
np.mean(np.absolute(test_y_ - test_y)))
print('Residual sum of squares (MSE): %.2f' % np.mean
((test_y_ - test_y) ** 2))
print('R2-score: %.2f' % r2_score (test_y , test_y_) )
```

Mean absolute error: 24.26
Residual sum of squares (MSE): 1062.26
R2-score: 0.75

Example

Let's see what the evaluation metrics are if we trained a regression model using the `FUELCONSUMPTION_COMB` feature.

Start by selecting `FUELCONSUMPTION_COMB` as the `train_x` data from the `train` dataframe, then select `FUELCONSUMPTION_COMB` as the `test_x` data from the `test` dataframe

```
train_x = train[['FUELCONSUMPTION_COMB']]
test_x = test[['FUELCONSUMPTION_COMB']]
```

Now we can train a Logistic Regression Model using the `train_x` we created and the `train_y` created previously.

```
regr = linear_model.LinearRegression ( )
regr.fit (train_x, train_y)
LinearRegression()
```

Find the predictions using the model's `predict` function and the

`test_x` data.

```
predictions = regr.predict (test_x)
```

Finally use the `predictions` and the `test_y` data and find the Mean Absolute Error value using the `np.absolute` and `np.mean` function like done previously

```
print('Mean absolute error: %.2f' %  
np.mean(np.absolute(predictions - test_y)))  
print('Residual sum of squares (MSE): %.2f' %  
np.mean((predictions - test_y) ** 2))  
print('R2-score: %.2f' % r2_score (test_y,  
predictions))
```

```
Mean absolute error: 20.33  
Residual sum of squares (MSE): 892.18  
R2-score: 0.79
```

We can see that the MAE is much worse than it is when we train using `ENGINESIZE`.

10. Multiple Linear Regression

Objectives

After completing this chapter, you will be able to:

- Use scikit-learn to implement Multiple Linear Regression
- Create a model, train it, test it and use the model

Importing Needed packages

```
import matplotlib.pyplot as plt  
import pandas as pd  
import pylab as pl  
import numpy as np  
%matplotlib inline
```

Downloading Data

To download the data, we will use the raw path to my Jupyter repository on **GitHub**. The path is given below. If you want to download the .csv file, then click on or enter the **url** in your browser:

<https://github.com/stricje1/jupyter/blob/main/data/MY2020%20Fuel%20Consumption%20Ratings.csv>

```
path =  
"https://raw.githubusercontent.com/stricje1/jupyter/main/d  
ata/MY2020%20Fuel%20Consumption%20Ratings.csv"
```

Understanding the Data

We have downloaded a fuel consumption dataset, **MY2020 Fuel Consumption Ratings.csv**, which contains model-specific fuel consumption ratings and estimated carbon dioxide emissions for new light-duty vehicles for retail sale in Canada.

Dataset source: <https://open.canada.ca/data/en/dataset/98f1a129-f628-4ce4-b24d-6f16bf24dd64>

- **MODELYEAR** e.g., 2014
- **MAKE** e.g., Acura
- **MODEL** e.g., ILX
- **VEHICLE CLASS** e.g., SUV

- **ENGINE SIZE** e.g., 4.7
- **CYLINDERS** e.g., 6
- **TRANSMISSION** e.g., A6
- **FUEL CONSUMPTION in CITY(L/100 km)** e.g., 9.9
- **FUEL CONSUMPTION in HWY (L/100 km)** e.g., 8.9
- **FUEL CONSUMPTION COMB (L/100 km)** e.g., 9.2
- **CO2 EMISSIONS (g/km)** e.g., 182 --> low --> 0

Reading the in the Data

As usual, we read the data into our **Jupyter notebook** using **pandas read_csv()** function:

```
df = pd.read_csv(path)
df.head()
```

	MODELYEAR	MAKE	MODEL	VEHICLECLASS
0	2020	Acura ILX		Compact
1	2020	Acura MDX SH-AWD		SUV: Small
2	2020	Acura MDX SH-AWD A-SPEC		SUV: Small
3	2020	Acura MDX Hybrid AWD		SUV: Small
4	2020	Acura RDX AWD		SUV: Small
5	2020	Acura RDX AWD A-SPEC		SUV: Small

	ENGINESIZE	CYLINDERS	TRANSMISSION	FUELTYPE	FUELCONSUMPTION_CITY
0	2.4	4	AM8	Z	9.9
1	3.5	6	AS9	Z	12.3
2	3.5	6	AS9	Z	12.2
3	3	6	AM7	Z	9.1
4	2	4	AS10	Z	11
5	2	4	AS10	Z	11.3

FUELCONSUMPTION_HWY	FUELCONSUMPTION_COMB	FUELCONSUMPTION_Comb
7	8.6	33
9.2	10.9	26

	FUELCONSUMPTION_HWY	FUELCONSUMPTION_COMB	FUELCONSUMPTION_COMB
	B	B	_MPG
2	9.5	11	26
3	9	9	31
4	8.6	9.9	29
5	9.1	10.3	27

	CO2EMISSIONS	Rating
0	199	3
1	254	3
2	258	3
3	210	3
4	232	6
5	241	6

Let's select some features that we want to use for regression.

```
df.shape
```

```
(967, 14)
```

```
cdf =
df[['ENGINESIZE', 'CYLINDERS', 'FUELCONSUMPTION_CITY', 'FUELCONSUMPTION_HWY', 'FUELCONSUMPTION_COMB', 'CO2EMISSIONS']]
cdf.head(9)
```

	ENGINESIZE	CYLINDERS	FUELCONSUMPTION_CITY	FUELCONSUMPTION_HWY	FUELCONSUMPTION_COMB	CO2EMISSIONS
0	2.4	4	9.9	7.0	8.6	199
1	3.5	6	12.3	9.2	10.9	254
2	3.5	6	12.2	9.5	11.0	258
3	3.0	6	9.1	9.0	9.0	210
4	2.0	4	11.0	8.6	9.9	232
5	2.0	4	11.3	9.1	10.3	241
6	3.5	6	8.4	8.2	8.4	196
7	2.4	4	10.2	7.4	8.9	209
8	3.5	6	11.4	7.7	9.8	228

Let's plot Emission values with respect to Engine size and display it in **Figure 10-1**:

```

fig=plt.figure()
plt.scatter(cdf.ENGINESIZE, cdf.CO2EMISSIONS,
color='blue')
plt.xlabel("Engine size")
plt.ylabel("Emission")
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
plt.show()
fig.savefig('outfig_07', dpi=200)

```

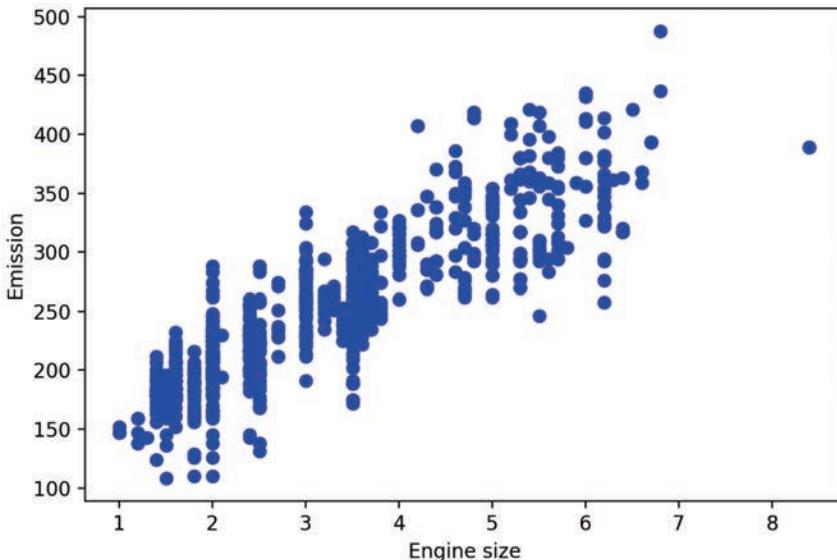


Figure 10-1. Scatterplot of engine size versus CO₂ emissions

Creating train and test dataset

Train/Test Split involves splitting the dataset into training and testing sets respectively, which are mutually exclusive. After which, you train with the training set and test with the testing set. This will provide a more accurate evaluation on out-of-sample accuracy because the testing dataset is not part of the dataset that have been used to train the model. Therefore, it gives us a better understanding of how well our model generalizes on new data.

We know the outcome of each data point in the testing dataset, making it great to test with! Since this data has not been used to train the model, the model has no knowledge of the outcome of these data points. So, in essence, it is an out-of-sample testing (Schafer, 1997).

Let's split our dataset into train and test sets. Around 80% of the entire dataset will be used for training and 20% for testing. We create a mask to select random rows using the `np.random.rand()` function:

```
msk = np.random.rand(len(df)) < 0.8
train = cdf[msk]
test = cdf[~msk]
print('The shape of the training set is: ', train.shape)
print('The shape of the testing set is: ', test.shape)
```

The shape of the training set is: (746, 6)

The shape of the testing set is: (221, 6)

Train data distribution

We'll now look at the data for the train subset we just defined. Well use a scatterplot as shown in **Figure 10-2**.

```
fig=plt.figure()
plt.scatter(train.ENGINESIZE, train.CO2EMISSIONS,
color='blue')
plt.xlabel("Engine size")
plt.ylabel("Emission")
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
plt.show()
```

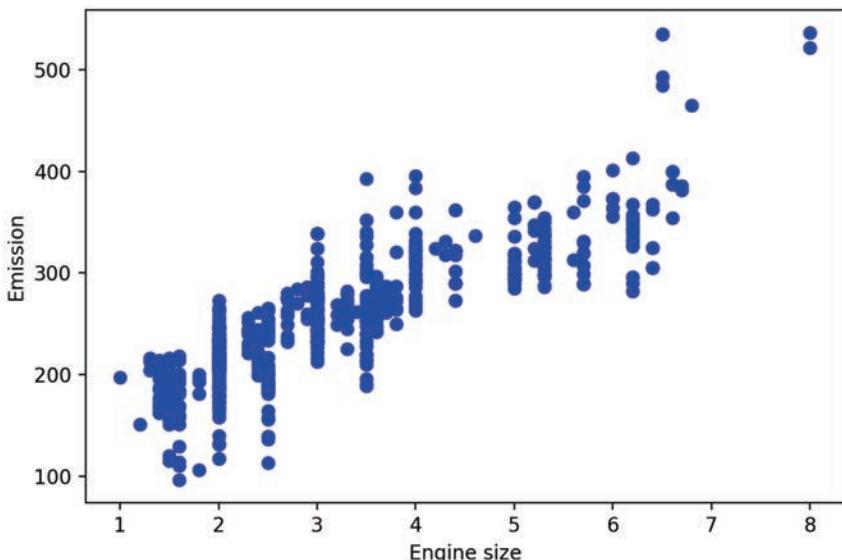


Figure 10-2. Scatterplot of train data distribution for engine size versus CO₂

Multiple Regression Model

In reality, there are multiple variables that impact the Scatterplot of engine size versus CO^2 emissions emission. When more than one independent variable is present, the process is called multiple linear regression. An example of multiple linear regression is predicting CO^2 emission using the features `FUELCONSUMPTION_COMB`, `EngineSize` and `Cylinders` of cars. The good thing here is that multiple linear regression model is the extension of the simple linear regression model.

```
from sklearn import linear_model
regr = linear_model.LinearRegression()
x =
np.asarray(train[['ENGINESIZE', 'CYLINDERS', 'FUELCONSUMPTION_COMB']])
y = np.asarray(train[['CO2EMISSIONS']])
regr.fit (x, y)
# The coefficients
print ('Coefficients: ', regr.coef_)
```

Coefficients: [[1.24581524 5.40360434 17.07786719]]

As mentioned before, **Coefficient** and **Intercept** are the parameters of the fitted line. Given that it is a multiple linear regression model with 3 parameters and that the parameters are the intercept and coefficients of the hyperplane, sklearn can estimate them from our data. Scikit-learn uses plain *Ordinary Least Squares* method to solve this problem.

Ordinary Least Squares (OLS)

Ordinary Least Squares (OLS) is a method for estimating the unknown parameters in a linear regression model. OLS chooses the parameters of a linear function of a set of explanatory variables by minimizing the sum of the squares of the differences between the target dependent variable and those predicted by the linear function. In other words, it tries to minimizes the sum of squared errors (SSE) or mean squared error (MSE) between the target variable (y) and our predicted output (\hat{y}) over all samples in the dataset.

OLS can find the best parameters using of the following methods:

- Solving the model parameters analytically using closed-form equations
- Using an optimization algorithm (Gradient Descent, Stochastic Gradient Descent, Newton's Method, etc.)

Prediction

Here, we use `y_hat` for \hat{y} (or `y_pred`). The residual sum of squares (RSS) is a statistical technique used to measure the amount of variance in a data set that is not explained by a regression model itself. Instead, it estimates the variance in the residuals, or error term. The Residual Sum of Squares or RSS is defined as:

$$RSS = \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

Intuitively, the formula says, take the predicted values, \hat{y} , and measure the distance between them and the true values, y , then square the distances, and add all of these square distances. These distances are the residuals, so this gives us the sum of square distances or residuals.

```
y_hat= regr.predict
(test[['ENGINESIZE','CYLINDERS','FUELCONSUMPTION_COMB']])
x =
np.asarray(test[['ENGINESIZE','CYLINDERS','FUELCONSUMPTION_COMB']])
y = np.asarray(test[['CO2EMISSIONS']])
print("Residual sum of squares: %.2f"
      % np.mean ((y_hat - y) ** 2))

# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % regr.score(x, y))
```

Residual sum of squares: 239.68

Variance score: 0.93

Variance Regression Score Explained

Let \hat{y} be the estimated target output, y the corresponding (correct) target output, and Var be the Variance (the square of the standard deviation). Then the explained variance is estimated as follows:

$$\text{explainedVariance}(y, \hat{y}) = 1 - \frac{\text{Var } y - \hat{y}}{\text{Var } y}$$

The best possible score is 1.0, the lower values are worse.

Example

Let's try to use a multiple linear regression with the same dataset, but this time use `FUELCONSUMPTION_CITY` and `FUELCONSUMPTION_HWY` instead of `FUELCONSUMPTION_COMB`. Does it result in better accuracy?

```
# Fit a model
regr = linear_model.LinearRegression()

x = np.asarray(train[['ENGINESIZE', 'CYLINDERS',
                      'FUELCONSUMPTION_CITY', 'FUELCONSUMPTION_HWY']])

y = np.asarray(train[['CO2EMISSIONS']])

regr.fit (x, y)
print ('Coefficients: ', regr.coef_)
```

Coefficients: [[1.01171836 5.11900019 10.52471693
6.13968126]]

```
# Make predictions and score the model
y_= regr.predict(test[['ENGINESIZE', 'CYLINDERS',
                      'FUELCONSUMPTION_CITY', 'FUELCONSUMPTION_HWY']])

x = np.asarray(test[['ENGINESIZE', 'CYLINDERS',
                      'FUELCONSUMPTION_CITY', 'FUELCONSUMPTION_HWY']])

y = np.asarray(test[['CO2EMISSIONS']])

print("Residual sum of squares: %.2f" % np.mean((y_ - y)** 2))

print('Variance score: %.2f' % regr.score(x, y))
```

Residual sum of squares: 242.29
Variance score: 0.93

Here, we can see that the metrics are nearly the same as they are for the previous model.

11. Polynomial Regression

Objectives

After completing this chapter, you will be able to:

- Use scikit-learn to implement Polynomial Regression
- Create a model, train it, test it and use the model

Table of contents

1. Downloading Data
2. Polynomial regression
3. Evaluation
4. Practice

Importing Needed packages

```
import matplotlib.pyplot as plt  
import pandas as pd  
import pylab as pl  
import numpy as np  
%matplotlib inline
```

Downloading Data

To download the data, we will use the raw path to my Jupyter repository on **GitHub**. The path is given below. If you want to download the .csv file, then click on or enter the **url** in your browser:
<https://github.com/stricje1/jupyter/blob/main/data/MY2020%20Fuel%20Consumption%20Ratings.csv>

```
path = "  
https://raw.githubusercontent.com/stricje1/jupyter/main/da  
ta/MY2020%20Fuel%20Consumption%20Ratings.csv"
```

Understanding the Data

We have downloaded a fuel consumption dataset, [MY2020 Fuel Consumption Ratings.csv](#), which contains model-specific fuel consumption ratings and estimated carbon dioxide emissions for new light-duty vehicles for retail sale in Canada. [Dataset source](#):

<https://open.canada.ca/data/en/dataset/98f1a129-f628-4ce4-b24d-6f16bf24dd64>

- **MODELYEAR** e.g., 2014
- **MAKE** e.g., Acura
- **MODEL** e.g., ILX
- **VEHICLE CLASS** e.g., SUV
- **ENGINE SIZE** e.g., 4.7
- **CYLINDERS** e.g., 6
- **TRANSMISSION** e.g., A6
- **FUEL CONSUMPTION in CITY(L/100 km)** e.g., 9.9
- **FUEL CONSUMPTION in HWY (L/100 km)** e.g., 8.9
- **FUEL CONSUMPTION COMB (L/100 km)** e.g., 9.2
- **CO2 EMISSIONS (g/km)** e.g., 182 --> low --> 0

Reading the in the Data

As usual, we read the data into our **Jupyter notebook** using **pandas read_csv()** function:

```
df = pd.read_csv(path)
df.head()
```

							FUELCONSU MPTION_CO CO2EMIS	
	MODEL	YEAR	MAKE	MODEL	VEHICLECLASS	...	MB MPG	SIONS
0	2020	Acura		ILX	Compact	...	33	199
1	2020	Acura	MDX	SH-AWD	SUV: Small	...	26	254
2	2020	Acura	MDX	SH-AWD	SUV: Small	...	26	258
					A-SPEC			
3	2020	Acura	MDX	Hybrid AWD	SUV: Small	...	31	210
4	2020	Acura	RDX	AWD	SUV: Small	...	29	232

Let's select some features that we want to use for regression. We'll call the dataframe, **cdf**:

```
cdf =
df[['ENGINESIZE', 'CYLINDERS', 'FUELCONSUMPTION_COMB', 'CO2EM
ISSONS']]
cdf.head(9)
```

			FUELCONSUMPTION_CO	
	ENGINESIZE	CYLINDERS	MB	CO2EMISSIONS
0	2.4	4	8.6	199
1	3.5	6	10.9	254
2	3.5	6	11.0	258
3	3.0	6	9.0	210
4	2.0	4	9.9	232
5	2.0	4	10.3	241
6	3.5	6	8.4	196
7	2.4	4	8.9	209
8	3.5	6	9.8	228

Let's plot Emission values with respect to Engine size:

```
fig=plt.figure()
plt.scatter(cdf.ENGINESIZE, cdf.CO2EMISSIONS,
color='blue')
plt.rc('font', size=16)
plt.xlabel("Engine size")
plt.ylabel("Emission")
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
plt.show()
fig.savefig('outfig_09', dpi=200)
```

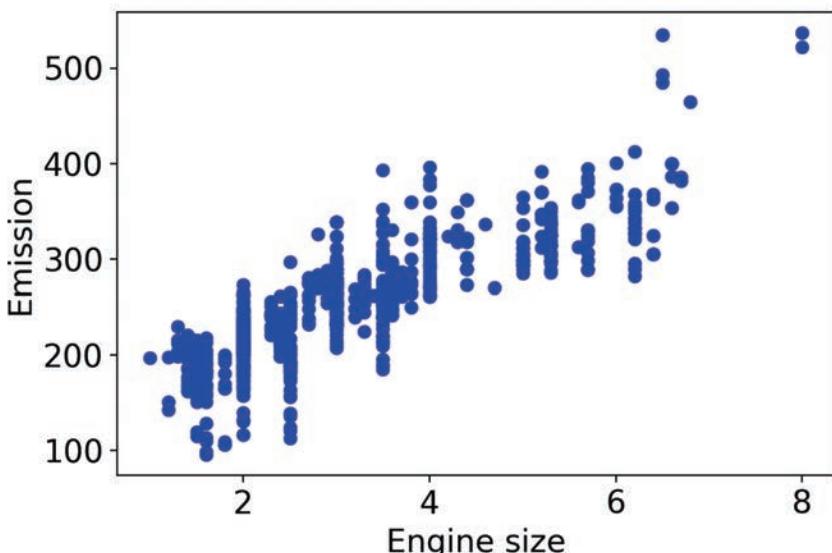


Figure 11-1. Scatterplot for Emissions vs. Engine Size from the fuel consumption data

Creating train and test dataset

Train/Test Split involves splitting the dataset into training and testing sets respectively, which are mutually exclusive. After which, you train with the training set and test with the testing set.

```
msk = np.random.rand(len(df)) < 0.8  
train = cdf[msk]  
test = cdf[~msk]
```

Polynomial regression

Sometimes, the trend of data is not really linear, and looks curvy. In this case we can use Polynomial regression methods. In fact, many different regressions exist that can be used to fit whatever the dataset looks like, such as quadratic, cubic, and so on, and it can go on and on to infinite degrees (Trab, 2019).

In essence, we can call all of these, polynomial regression, where the relationship between the independent variable x and the dependent variable y is modeled as an n th degree polynomial in x . Let's say you want to have a polynomial regression (let's make 2-degree polynomial):

$$y = b + \theta_1 x + \theta_2 x^2$$

Now, the question is: how we can fit our data on this equation while we have only x values, such as **Engine Size**? Well, we can create a few additional features: 1, x , and x^2 .

PolynomialFeatures() function in Scikit-learn library, drives a new feature set from the original feature set. That is, a matrix will be generated consisting of all polynomial combinations of the features with degree less than or equal to the specified degree. For example, let's say the original feature set has only one feature, *ENGINESIZE*. Now, if we select the degree of the polynomial to be 2, then it generates 3 features, *degree=0*, *degree=1* and *degree=2*:

```
from sklearn.preprocessing import PolynomialFeatures  
from sklearn import linear_model  
train_x = np.asarray(train[['ENGINESIZE']])  
train_y = np.asarray(train[['CO2EMISSIONS']])
```

```

test_x = np.asanyarray(test[['ENGINESIZE']])
test_y = np.asanyarray(test[['CO2EMISSIONS']])

poly = PolynomialFeatures(degree=2)
train_x_poly = poly.fit_transform (train_x)
train_x_poly

array([[ 1. ,  3.5 , 12.25],
       [ 1. ,  2. ,  4. ],
       [ 1. ,  2. ,  4. ],
       ...,
       [ 1. ,  2. ,  4. ],
       [ 1. ,  2. ,  4. ],
       [ 1. ,  2. ,  4. ]])

```

`fit_transform` takes our x values, and output a list of our data raised from power of 0 to power of 2 (since we set the degree of our polynomial to 2).

The equation and the sample example are displayed below.

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \rightarrow \begin{bmatrix} 1 & v_1 & v_1^2 \\ 1 & v_2 & v_2^2 \\ \vdots & \vdots & \vdots \\ 1 & v_n & v_n^2 \end{bmatrix}$$

$$\begin{bmatrix} 2. \\ 2.4 \\ 1.5 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2. & 4. \\ 1 & 2.4 & 5.76 \\ 1 & 1.5 & 2.25 \\ \vdots & \vdots & \vdots \end{bmatrix}$$

It looks like feature sets for multiple linear regression analysis, right? Yes, polynomial regression is a special case of linear regression, with the main idea of how do you select your features. Just consider replacing the x with x_1, x_1^2 with x_2 , and so on. Then the 2nd degree equation would be turn into:

$$y = b + \theta_1 x_1 + \theta_2 x_2$$

Now, we can deal with it as a 'linear regression' problem. Therefore, we consider this polynomial regression to be a special case of traditional multiple linear regression. So, you can use the same mechanism as linear regression to solve such problems.

so, we can use `LinearRegression()` function to solve it:

```

clf = linear_model.LinearRegression ()
train_y_ = clf.fit (train_x_poly, train_y)
# The coefficients
print ('Coefficients: ', clf.coef_)
print ('Intercept: ',clf.intercept_)

```

```

Coefficients: [[ 0.          53.92693813 -2.05613277]]
Intercept: [108.39429848]

```

As mentioned before, **Coefficient** and **Intercept**, are the parameters of the fit curvy line. Given that it is a typical multiple linear regression, with 3 parameters, and knowing that the parameters are the intercept and coefficients of hyperplane, sklearn has estimated them from our new set of feature sets. Let's plot it:

```

fig=plt.figure()
# Define the scatterplot
plt.scatter(train.ENGINESIZE, train.CO2EMISSIONS,
color='blue')

XX = np.arange (0.0, 10.0, 0.1) # Evenly spaced values
within a given interval

# Get the intercept and coefficient values for the
regressor
yy = clf.intercept_[0]+ clf.coef_[0][1]*XX+
      clf.coef_[0][2]*np.power(XX, 2)
# Plot fitted line colored red
plt.plot(XX, yy, '-r' )

# User defined default font size
plt.rc('font', size=16)
plt.xlabel("Engine size")
plt.ylabel("Emission")

# Set figure padding for below and above Labels
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
plt.show()
# Save the figure to the working directory as a PNG file
fig.savefig('outfig_10', dpi=200)

```

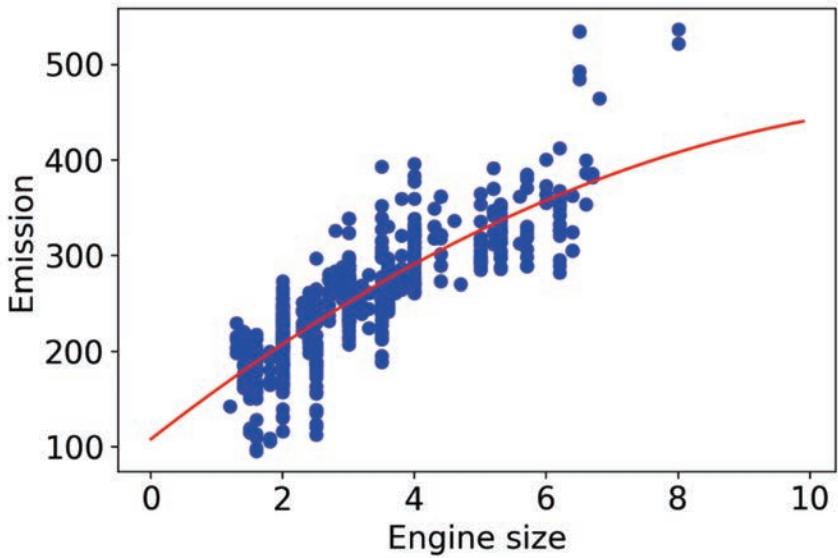


Figure 11-2. Scatterplot with fitted a second-order polynomial regression model.

Evaluation

Now, let's evaluate the regression model's fit to the emissions data, using *Mean Absolute Error* (MAE), *Residual Sum of Squares* (RSE), and R^2 -score. The outcome below shows that we have a pretty good fit using the second-order polynomial regression model.

```
from sklearn.metrics import r2_score

test_x_poly = poly.transform (test_x)
test_y_ = clf.predict (test_x_poly)

print("Mean absolute error: %.2f" % np.mean
(np.absolute(test_y_ - test_y)))
print("Residual sum of squares (MSE): %.2f" %
np.mean((test_y_ - test_y) ** 2))
print("R2-score: %.2f" % r2_score(test_y, test_y_))
```

```
Mean absolute error: 23.06
Residual sum of squares (MSE): 773.01
R2-score: 0.71
```

Example

Next, we try to use a polynomial regression with the dataset but this time with degree three (cubic). So, let's see if we can increase the accuracy using a third-degree polynomial regression model to fit the data. We'll get the model metrics and plot the fit in **Figure 11-3**.

```
poly3 = PolynomialFeatures (degree=3)
train_x_poly3 = poly3.fit_transform (train_x)
clf3 = linear_model.LinearRegression ()
train_y3_ = clf3.fit (train_x_poly3, train_y)

# The coefficients
print ('Coefficients: ', clf3.coef_)
print ('Intercept: ', clf3.intercept_)

fig=plt.figure()
plt.scatter(train.ENGINESIZE, train.CO2EMISSIONS,
            color='blue')
XX = np.arange(0.0, 10.0, 0.1)
yy = clf3.intercept_[0]+ clf3.coef_[0][1]*XX +
      clf3.coef_[0][2]*np.power(XX, 2) +
      clf3.coef_[0][3]*np.power(XX, 3)
plt.plot(XX, yy, '-r' )
plt.xlabel("Engine size")
plt.ylabel("Emission")
plt.tight_layout(pad=0.4, w_pad=0.5, h_pad=1.0)
plt.show()
fig.savefig('outfig_11', dpi==200)

test_x_poly3 = poly3.transform(test_x)
test_y3_ = clf3.predict(test_x_poly3)
print("Mean absolute error: %.2f" %
      np.mean(np.absolute(test_y3_ - test_y)))
print("Residual sum of squares (MSE): %.2f" %
      np.mean((test_y3_ - test_y) ** 2))
print("R2-score: %.2f" % r2_score (test_y,test_y3_ ) )
```

```
Coefficients:  [[  0.          139.40829448 -25.75651153
  1.97802964]]
Intercept:  [17.27314255]
```

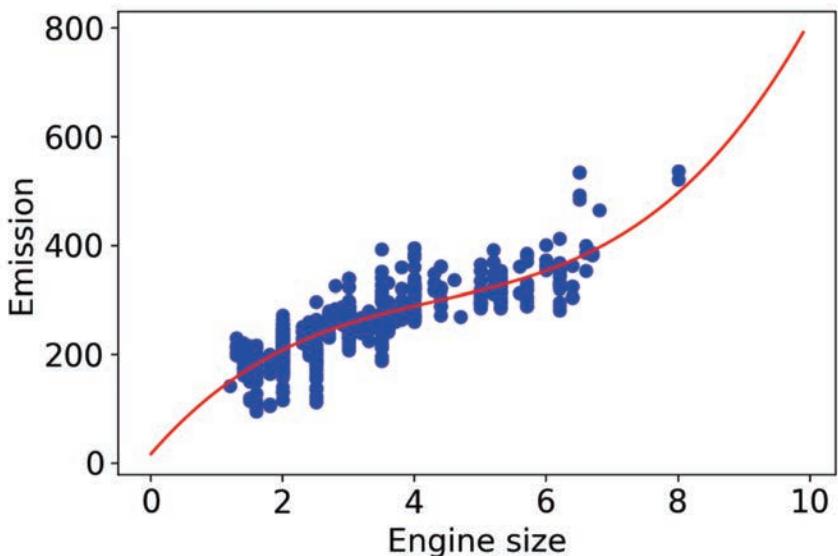


Figure 11-3. Emissions data base on engine size with a fitted third-order (cubic) polynomial model.

The model metrics shows that the third-order polynomial regression model does not perform significantly better than the second-order model.

Mean absolute error: 22.04

Residual sum of squares (MSE): 754.50

R2-score: 0.72

12. Non-Linear Regression Analysis

Objectives

After completing this chapter, you will be able to:

- Differentiate between linear and non-linear regression
- Use non-linear regression model in Python

If the data shows a curvy trend, then linear regression will not produce very accurate results when compared to a non-linear regression since linear regression presumes that the data is linear. Let's learn about nonlinear regressions and apply an example in python. In this notebook, we fit a non-linear model to the datapoints corresponding to China's GDP from 1960 to 2014.

Importing required libraries

```
import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline
```

Although linear regression can do a great job at modeling some datasets, it cannot be used for all datasets. First recall how linear regression, models a dataset. It models the linear relationship between a dependent variable y and the independent variables x . It has a simple equation, of degree 1, for example $y = 2x + 3$, as shown in **Figure 12-1**

```
x = np.arange (-5.0, 5.0, 0.1)  
  
# You can adjust the slope and intercept to verify the  
# changes in the graph  
y = 2*(x) + 3  
y_noise = 2 * np.random.normal(size=x.size)  
ydata = y + y_noise  
plt.figure(dpi=200)  
plt.plot(x, ydata, 'bo')  
plt.plot(x, y, 'r')  
plt.ylabel('Dependent Variable')  
plt.xlabel('Independent Variable')  
plt.show()
```

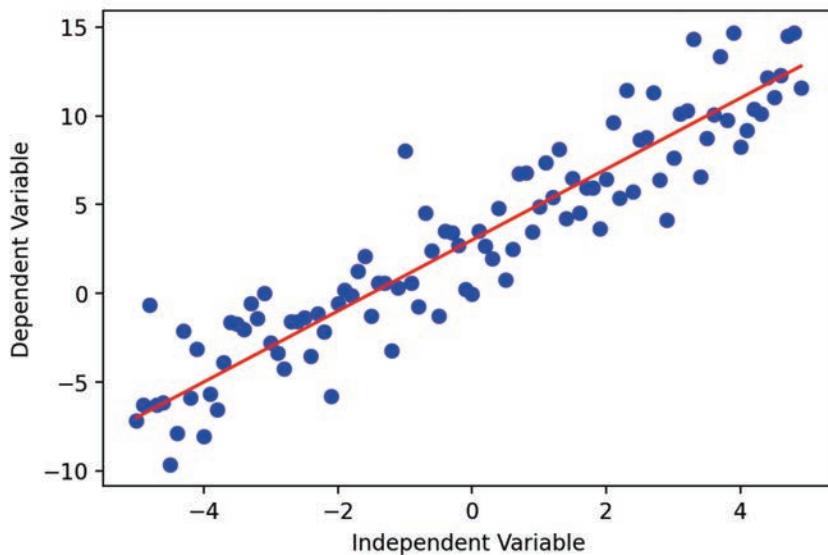


Figure 12-1. Linear regression $y=2x+3$ fit of the data

Non-linear regression is a method to model the non-linear relationship between the independent variables x and the dependent variable y . Essentially any relationship that is not linear can be termed as non-linear, and is usually represented by the polynomial of k degrees (maximum power of x). For example:

$$y = ax^3 + bx^2 + cx + d$$

Non-linear functions can have elements like exponentials, logarithms, fractions, and so on. For example:

$$y = \log(x)$$

We can have a function that's even more complicated such as :

$$y = \log(ax^3 + bx^2 + cx + d)$$

Let's take a look at a cubic function's graph in **Figure 12-2.**

```
x = np.arange(-5.0, 5.0, 0.1)
```

You can adjust the slope and intercept to verify the changes in the graph

```
y = 1*(x**3) + 1*(x**2) + 1*x + 3
```

```

y_noise = 20 * np.random.normal(size=x.size)
ydata = y + y_noise
plt.figure(dpi=200)
plt.plot(x, ydata, 'bo')
plt.plot(x, y, 'r')
plt.ylabel('Dependent Variable')
plt.xlabel('Independent Variable')
plt.show()

```

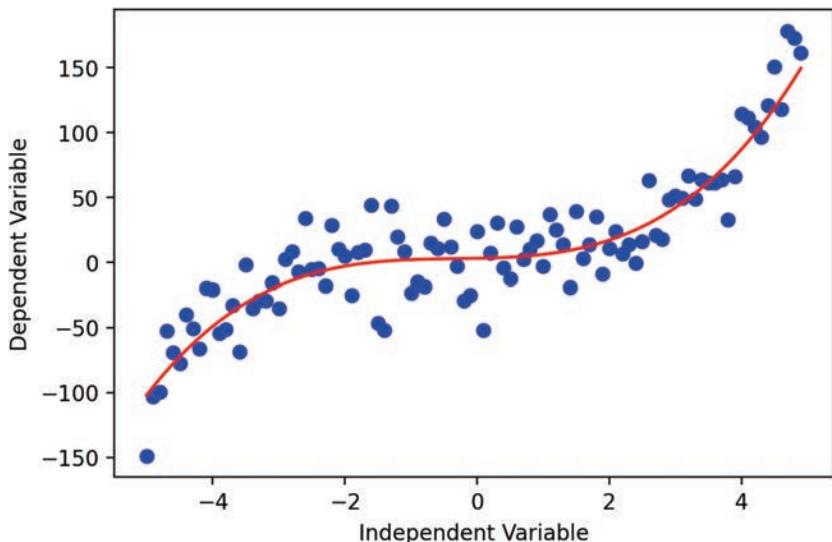


Figure 12-2. Non-linear regression (logarithmic) $y = \log(ax^3 + bx^2 + cx + d)$.

As you can see, this function has x^3 and x^2 as independent variables. Also, the graphic of this function is not a straight line over the 2D plane. So, this is a non-linear function.

Some other types of non-linear functions are:

Quadratic

The quadratic equation is best known for the U-shaped parabola. This can be visualized in **Figure 12-3**.

$$Y = X^2$$

```

x = np.arange(-5.0, 5.0, 0.1)

# You can adjust the slope and intercept to verify the
# changes in the graph

y = np.power(x,2)
y_noise = 2 * np.random.normal(size=x.size)
ydata = y + y_noise
plt.figure(dpi=200)
plt.plot(x, ydata, 'bo')
plt.plot(x, y, 'r')
plt.ylabel('Dependent Variable')
plt.xlabel('Independent Variable')
plt.show()

```

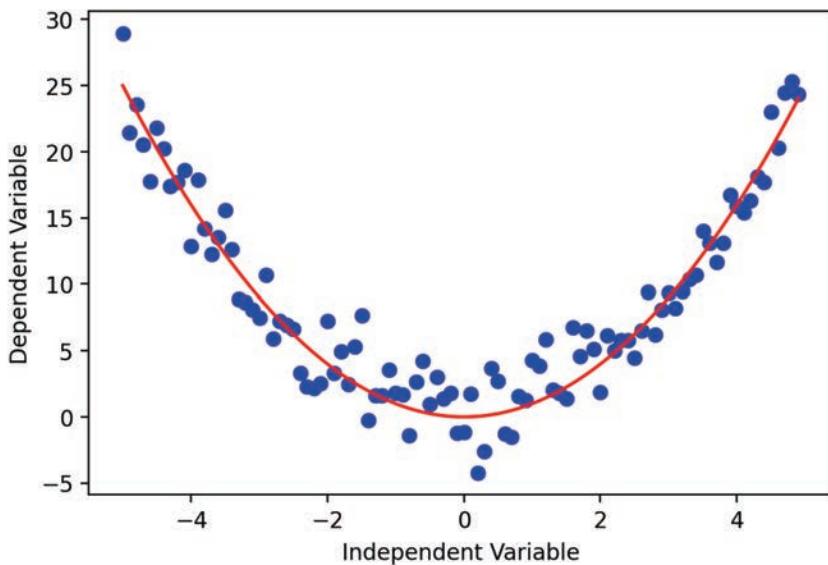


Figure 12-3. Non-linear regression (quadratic) $Y = X^2$.

Exponential

An exponential function with base c is defined by

$$Y = a + bc^x$$

where $b \neq 0$, $c > 0$, $c \neq 1$, and x is any real number. The base, c , is constant and the exponent, x , is a variable. This function is visualized

in **Figure 12-4**.

```
X = np.arange (-5.0, 5.0, 0.1)

# You can adjust the slope and intercept to verify the
# changes in the graph

Y= np.exp(X)
plt.figure(dpi=200)
plt.plot(X,Y)
plt.ylabel('Dependent Variable')
plt.xlabel('Independent Variable')
plt.show()
```

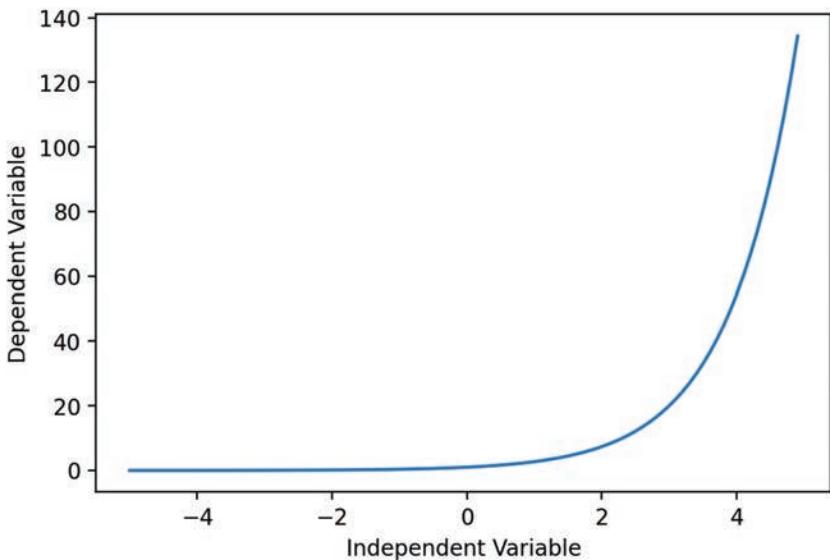


Figure 12-4. Non-linear regression (exponential) $Y = a + bc^x$.

Logarithmic

The response y is a result of applying the logarithmic map from the input x to the output y . It is one of the simplest forms of $\log()$: i.e.,

$$y = \log(x)$$

Consider that instead of x , we can use X , which can be a polynomial representation of the x values. In general form it would be written as

$$y = \log(X).$$

This function is visualized in **Figure 12-5**.

```
X = np.arange(-5.0, 5.0, 0.1)

Y = np.log(X)
plt.figure(dpi=200)
plt.plot(X,Y)
plt.ylabel('Dependent Variable')
plt.xlabel('Independent Variable')
plt.show()
```

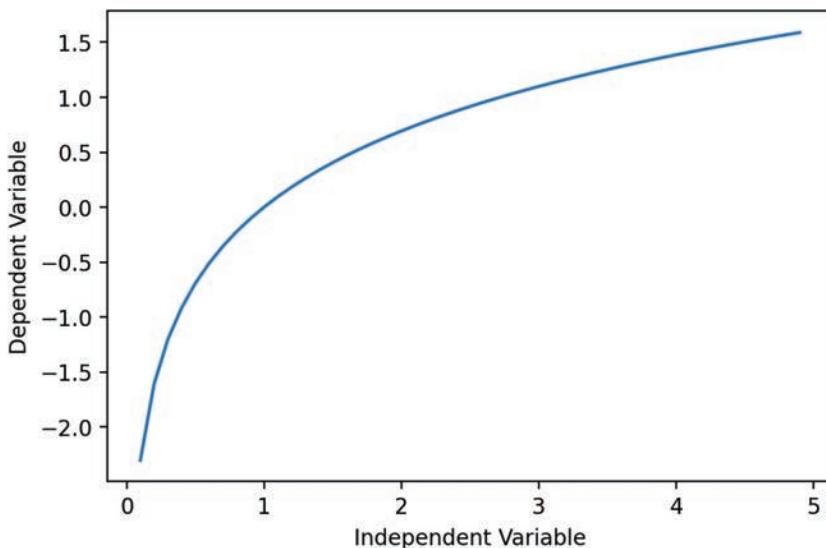


Figure 12-5. Non-linear regression(logarithmic) $y = \log(x)$.

Sigmoidal/Logistic

The sigmoid function shown below is visualized in **Figure 12-6**.

$$Y = a + \left(\frac{b}{1 + c^{x-d}} \right)$$

```
X = np.arange(-5.0, 5.0, 0.1)

Y = 1-4/(1 + np.power (3, X-2))
plt.figure(dpi=200)
```

```

plt.plot(X,Y)
plt.ylabel('Dependent Variable')
plt.xlabel('Independent Variable')
plt.show()

```

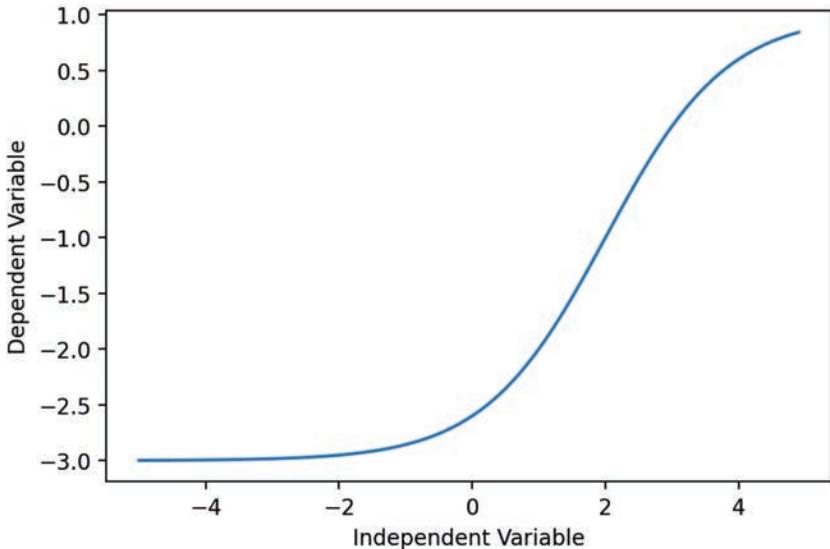


Figure 12-6. Non-linear regression (sigmoid) $Y = a + \left(\frac{b}{1+c^{x-d}} \right)$.

Non-Linear Regression example

For an example, we're going to try and fit a non-linear model to the datapoints corresponding to China's GDP from 1960 to 2014. We download a dataset with two columns, the first, a year between 1960 and 2014, the second, China's corresponding annual gross domestic income in US dollars for that year. So, let's download the data.

```

import numpy as np
import pandas as pd
path =
"https://raw.githubusercontent.com/stricje1/jupyter/main/d
ata/china_gdp.csv"

df = pd.read_csv(path)
df.head(10)

```

Year	Value
0	1960 5.918412e+10
1	1961 4.955705e+10
2	1962 4.668518e+10
3	1963 5.009730e+10
4	1964 5.906225e+10
5	1965 6.970915e+10
6	1966 7.587943e+10
7	1967 7.205703e+10
8	1968 6.999350e+10
9	1969 7.871882e+10

Plotting the Dataset

The datapoints in **Figure 12-7** look either a logistic or an exponential function. The growth starts off slow, then from 2005 on forward, the growth is very significant. And finally, it decelerates slightly in the 2010s.

```
plt.figure(dpi=200)

x_data, y_data = (df["Year"].values, df["Value"].values)
plt.plot(x_data, y_data, 'ro')

plt.ylabel('GDP')
plt.xlabel('Year')

plt.show()
plt.savefig('outfig_10', dpi=200)
```

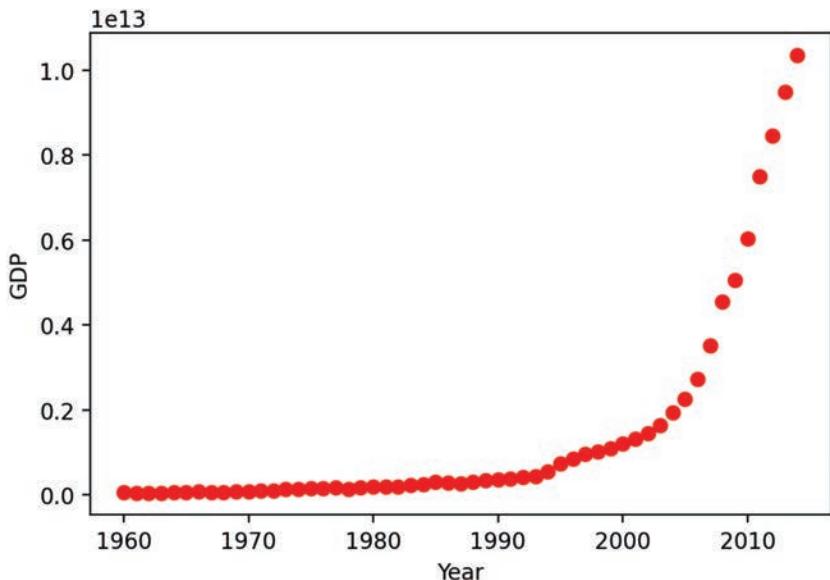


Figure 12-7. Non-linear data formed by Chinese GDP vs. Year.

Choosing a model

From an initial look at the plot, we determine that the logistic function could be a good approximation, since it has the property of starting with a slow growth, increasing growth in the middle, and then decreasing again at the end; as illustrated below and visualized in Figure 12-8.

```
plt.figure(dpi=200)

X = np.arange (-5.0, 5.0, 0.1)
Y = 1.0 / (1.0 + np.exp(-X))

plt.plot(X,Y)

plt.ylabel('Dependent Variable')
plt.xlabel('Independent Variable')

plt.show()
plt.savefig('outfig_10', dpi=200)
```

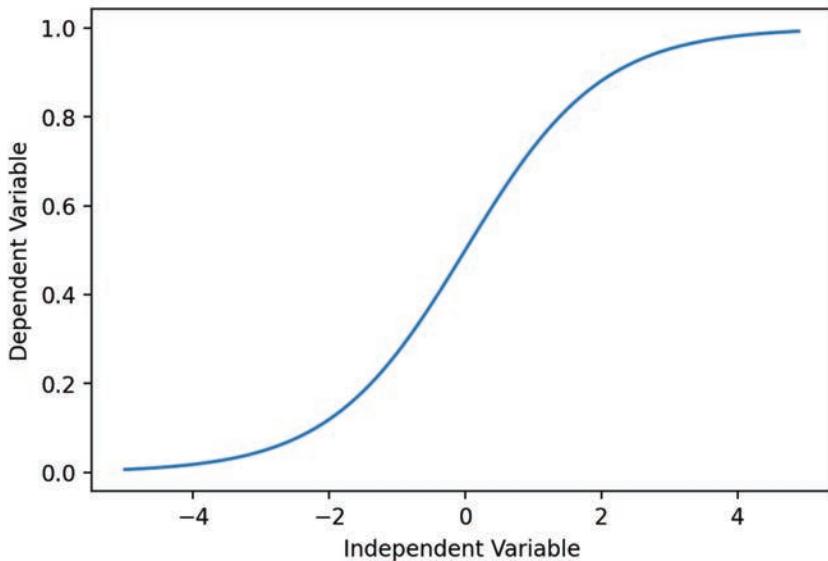


Figure 12-8. Non-linear regression (exponential) $Y = \frac{1.0}{(1.0+np.exp(-X))}$.

The formula for the logistic function is the following:

$$\hat{Y} = \frac{1}{1 + e^{(-\beta_1)(X-\beta_2)}}$$

β_1 : Controls the curve's steepness,

β_2 : Slides the curve on the x-axis.

Building The Model

Now, let's build our regression model and initialize its parameters.

```
def sigmoid(x, Beta_1, Beta_2):
    y = 1 / (1 + np.exp(-Beta_1*(x-Beta_2)))
    return y
```

Let's look at a sample sigmoid line that might fit with the data and plot it in **Figure 12-9**.

```
beta_1 = 0.10
beta_2 = 1990.0
```

```

# Logistic function
Y_pred = sigmoid(x_data, beta_1 , beta_2)

# Plot initial prediction against datapoints
plt.figure(dpi=200)
plt.plot(x_data, Y_pred*15000000000000.)
plt.plot(x_data, y_data, 'ro')

```

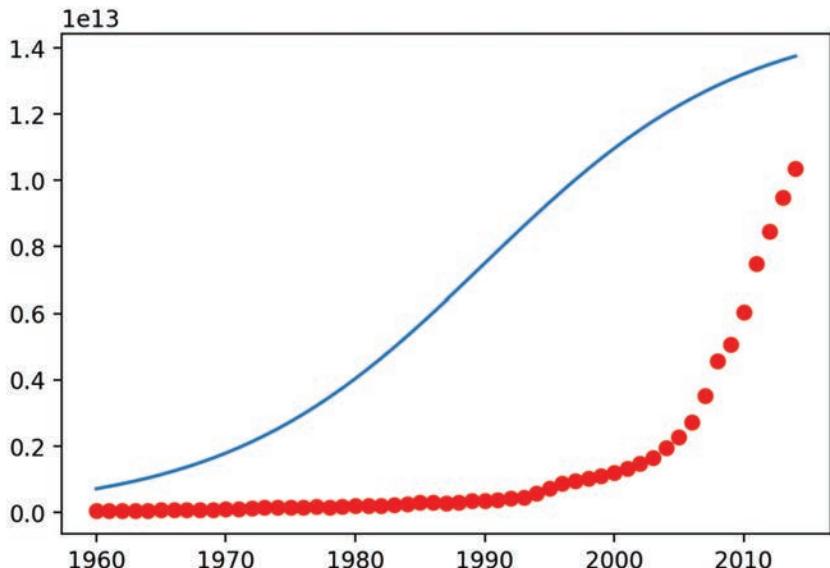


Figure 12-9. Non-linear regression (logistic) $\hat{Y} = \frac{1}{1+e^{(-\beta_1)(X-\beta_2)}}$.

Our task here is to find the best parameters for our model. Let's first normalize our x and y:

```

xdata = x_data/max(x_data)
ydata = y_data/max(y_data)

```

Finding the Best Fit

So, how we find the best parameters for our fit line? We can use **curve_fit** which uses non-linear least squares to fit our sigmoid function, to data. Optimize values for the parameters so that the sum of the squared residuals of `sigmoid(xdata, *popt)` - `ydata` is minimized.

*`popt` are our optimized parameters.

```
from scipy.optimize import curve_fit
popt, pcov = curve_fit(sigmoid, xdata, ydata)
# Print the final parameters
print(" beta_1 = %f, beta_2 = %f" % (popt[0], popt[1]))
beta_1 = 690.451712, beta_2 = 0.997207
```

Now we plot our resulting regression model in **Figure 12-10**.

```
x = np.linspace (1960, 2015, 55)
x = x/max(x)
plt.figure(dpi=200)
y = sigmoid(x, *popt)
plt.plot (xdata, ydata, 'ro', label = 'data')
plt.plot(x, y, linewidth = 3.0, label = 'fit')
plt.legend(loc='best')
plt.ylabel('GDP')
plt.xlabel('Year')
plt.show()
```

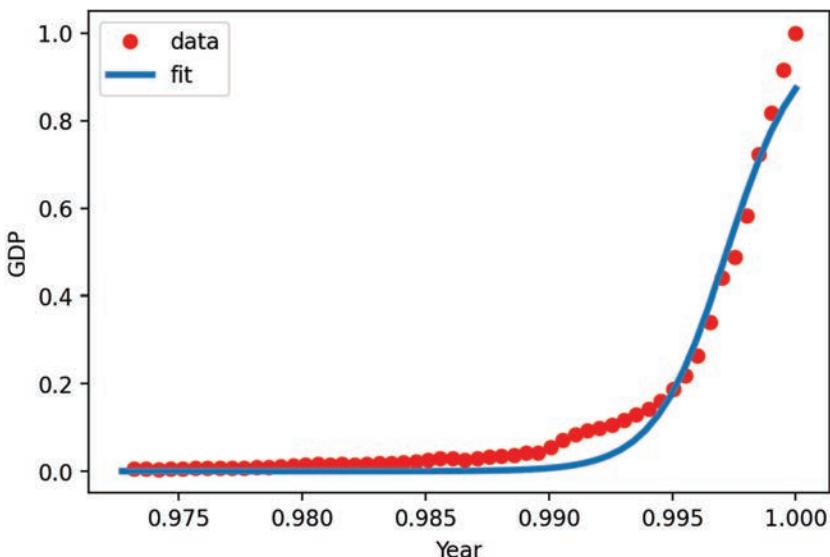


Figure 12-10. Non-linear regression (optimized parameters) `popt = [690.451712, 0.997207], pcov = [[1.52271494e+03, -2.88118935e-04], [-2.88118935e-04, 7.25961451e-09]]`

Example

Now we'll calculate the accuracy of our model. First, let's split data into train/test, then use the train data to fit the model, then predict using the test data, the evaluate the model with MAE, MSE, and R^2 .

```
msk = np.random.rand(len(df)) < 0.8
train_x = xdata[msk]
test_x = xdata[~msk]
train_y = ydata[msk]
test_y = ydata[~msk]

# Build the model using train set
popt, pcov = curve_fit(sigmoid, train_x, train_y)

# Predict using test set
y_hat = sigmoid(test_x, *popt)

# Evaluation
print("Mean absolute error: %.2f" %
np.mean(np.absolute(y_hat - test_y)))
print("Residual sum of squares (MSE): %.2f" % np.mean
((y_hat - test_y) ** 2))
from sklearn.metrics import r2_score
print("R2-score: %.2f" % r2_score(test_y, y_hat) )
```

```
Mean absolute error: 0.03
Residual sum of squares (MSE): 0.00
R2-score: 0.88
```


13. K-Nearest Neighbors

Objectives

After completing this chapter, you will be able to:

- Use K Nearest neighbors to classify data

In this Lab you will load a customer dataset, fit the data, and use K-Nearest Neighbors to predict a data point. But what is **K-Nearest Neighbors**?

K-Nearest Neighbors (KNN) is a supervised learning algorithm. Where the data is 'trained' with data points corresponding to their classification. To predict the class of a given data point, it takes into account the classes of the 'K' nearest data points and chooses the class in which the majority of the 'K' nearest data points belong to as the predicted class.

Figure 13-1 provides a visualization of the K-Nearest Neighbors (KNN) algorithm. (Navlani, 2018)

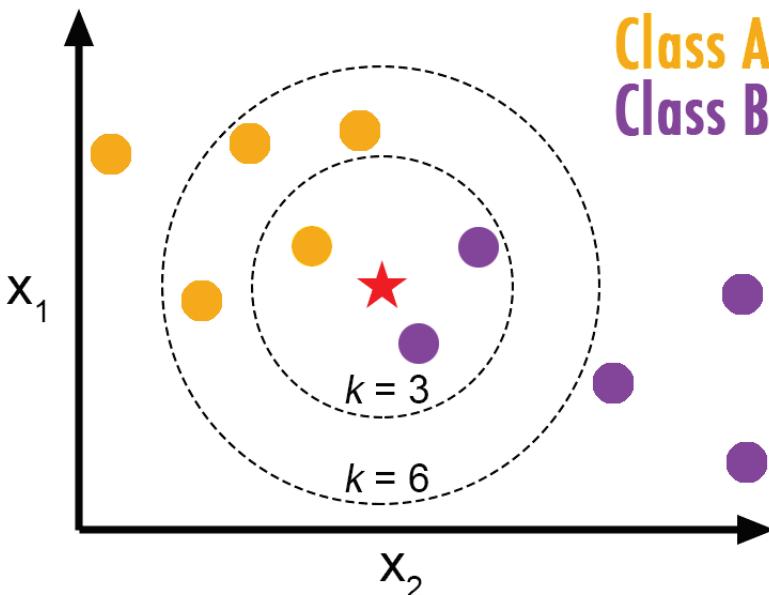


Figure 13-1. Depiction for KNN got two classes and two values of K .

In this case, we have data points of Class A and B. We want to predict what the star (test data point) is. If we consider a k value of 3 (3 nearest data points), we will obtain a prediction of Class B. Yet if we consider a k value of 6, we will obtain a prediction of Class A.

In this sense, it is important to consider the value of k. Hopefully from this diagram, you should get a sense of what the K-Nearest Neighbors algorithm is. It considers the 'K' Nearest Neighbors (data points) when it predicts the classification of the test point.

Table of contents

1. About the dataset
2. Data Visualization and Analysis
3. Classification

Let's load required libraries

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from sklearn import preprocessing
%matplotlib inline
```

About the dataset

Imagine a telecommunications provider has segmented its customer base by service usage patterns, categorizing the customers into four groups. If demographic data can be used to predict group membership, the company can customize offers for individual prospective customers. It is a **classification** problem. That is, given the dataset, with predefined labels, we need to build a model to be used to predict class of a new or unknown case. The example focuses on using demographic data, such as region, age, and marital, to predict usage patterns.

The target field, called custcat, has four possible values that correspond to the four customer groups, as follows: 1- Basic Service
2- E-Service 3- Plus Service 4- Total Service

Our objective is to build a classifier, to predict the class of unknown

cases. We will use a specific type of classification called K nearest neighbor.

Let's download the dataset from the url defined as `path` below.

```
path =  
"https://raw.githubusercontent.com/stricje1/jupyter/main/d  
ata/teleCust1000t.csv"
```

Did you know, when it comes to Machine Learning, you will likely be working with large datasets. Although we do not think about data storage here, it is a major and often expensive consideration.

So, we have a location for the data and we'll use it to load the data from a CSV File.

```
df = pd.read_csv(path)  
df.head()
```

	region	tenure	age	marital	address	income	e	gender	reside	custcat
0	2	13	44	1	9	64.0	4	...	0	2
1	3	11	33	1	7	136.0	5	...	0	6
2	3	68	52	1	24	116.0	1	...	1	2
3	2	33	33	0	12	33.0	2	...	1	1
4	2	23	30	1	9	30.0	1	...	0	4

Data Visualization and Analysis

Let's see how many of each class is in our data set:

```
df['custcat'].value_counts()
```

```
3    281  
1    266  
4    236  
2    217  
Name: custcat, dtype: int64
```

281 Plus Service, 266 Basic-service, 236 Total Service, and 217 E-Service customers

You can easily explore your data using visualization techniques, in

this case a histogram as seen in **Figure 13-2**. Recall that a histogram is a graphical view of a frequency distribution.

```
df.hist(column='income', bins=50)
```

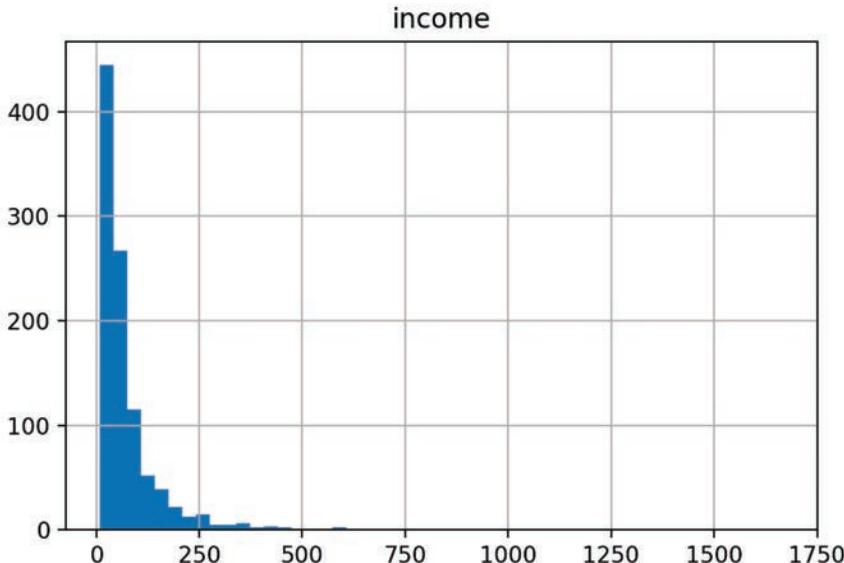


Figure 13-2. Histogram of income in 50 bins, with the left-most bin being the lowest income, then moving to the right for higher incomes.

Feature set

Recall that in machine learning and pattern recognition, a feature is an individual measurable property or characteristic of a phenomenon. Let's define feature sets, X, that are individual characteristics of income.:

```
df.columns  
Index(['region', 'tenure', 'age', 'marital', 'address',  
'income', 'ed',  
       'employ', 'retire', 'gender', 'reside', 'custcat'],  
      dtype='object')
```

To use **scikit-learn** library, we have to convert the **Pandas** data frame to a **Numpy** array:

```
X = df[['region', 'tenure', 'age', 'marital', 'address',
        'income', 'ed', 'employ', 'retire', 'gender', 'reside']]
.X.values #.astype(float)
X[0:5]
```

```
array([
 [2., 13., 44., 1., 9., 64., 4., 5., 0., 0., 2.],
 [3., 11., 33., 1., 7., 136., 5., 5., 0., 0., 6.],
 [3., 68., 52., 1., 24., 116., 1., 29., 0., 1., 2.],
 [2., 33., 33., 0., 12., 33., 2., 0., 0., 1., 1.],
 [2., 23., 30., 1., 9., 30., 1., 2., 0., 0., 4.]])
```

Let's see what values our labels possess.

```
y = df['custcat'].values
y[0:5]
```

```
array([1, 4, 3, 1, 3], dtype=int64)
```

Normalize Data

Data Standardization gives the data zero mean and unit variance, it is good practice, especially for algorithms such as KNN which is based on the distance of data points:

```
X = preprocessing.StandardScaler().fit
(X).transform(X.astype (float))
X[0:5]
```

```
array([
 [-0.026968, -1.055125,  0.184505,  1.010050,
 -0.253034, -0.126506,  1.087753, -0.594123, -0.222076,
 -1.034598, -0.230650],
 [ 1.198836, -1.148806, -0.691812,  1.010051, -0.451415,
  0.546449,  1.906227, -0.594122, -0.222076, -1.034598,
  2.556661],
 [ 1.198836,  1.521092,  0.821826,  1.010051,  1.234819,
  0.359517, -1.367671,  1.787528, -0.222076,  0.966559,
 -0.230650],
 [-0.026968, -0.118319, -0.691812, -0.990049,  0.044536,
 -0.416251, -0.549196, -1.090299, -0.222076,  0.966559,
 -0.927478],
 [-0.026968, -0.586721, -0.930808,  1.010051, -0.253034,
 -0.444291, -1.367671, -0.891829, -0.222076, -1.034598,
 1.163006]])
```

Train Test Split

Out of Sample Accuracy is the percentage of correct predictions that the model makes on data that the model has NOT been trained on. Doing a train and test on the same dataset will most likely have low out-of-sample accuracy, due to the likelihood of our model overfitting.

It is important that our models have a high, out-of-sample accuracy, because the purpose of any model, of course, is to make correct predictions on unknown data. So how can we improve out-of-sample accuracy? One way is to use an evaluation approach called **Train/Test Split**. Train/Test Split involves splitting the dataset into training and testing sets respectively, which are mutually exclusive. After which, you train with the training set and test with the testing set.

This will provide a more accurate evaluation on out-of-sample accuracy because the testing dataset is not part of the dataset that has been used to train the model. It is more realistic for the real-world problems.

```
from sklearn.model_selection import train_test_split  
X_train, X_test, y_train, y_test = train_test_split ( X,  
y, test_size=0.2, random_state=4)  
print ('Train set:', X_train.shape, y_train.shape)  
print ('Test set:', X_test.shape, y_test.shape)
```

```
Train set: (800, 11) (800,)  
Test set: (200, 11) (200,)
```

Classification – K nearest neighbor (KNN)

Import library

Classifier implementing the **k-nearest neighbor** vote.

```
from sklearn.neighbors import KNeighborsClassifier
```

Training

Let's start the algorithm with k=4 for now:

```
k = 4
#Train Model and Predict
neigh = KNeighborsClassifier(n_neighbors = k).fit(X_train,
y_train)
neigh
KNeighborsClassifier(n_neighbors=4)
```

Predicting

We can use the model to make predictions on the test set:

```
yhat = neigh.predict (X_test)
yhat[0:5]

array([1, 1, 3, 2, 4], dtype=int64)
```

Accuracy evaluation

In multilabel classification, **accuracy classification score** is a function that computes subset accuracy. This function is equal to the **jaccard_score** function. Essentially, it calculates how closely the actual labels and predicted labels are matched in the test set.

```
from sklearn import metrics
print("Train set Accuracy: ",
metrics.accuracy_score(y_train, neigh.predict(X_train)))
print("Test set Accuracy: ",
metrics.accuracy_score(y_test, yhat))
```

Train set Accuracy: 0.5475

Test set Accuracy: 0.32

Let's build the model again, but this time with k=6.

```
k = 6
neigh6 = KNeighborsClassifier(n_neighbors = k).fit
(X_train, y_train)
yhat6 = neigh6.predict(X_test)
print("Train set Accuracy: ",
metrics.accuracy_score(y_train, neigh6.predict(X_train)))
print("Test set Accuracy: ",
metrics.accuracy_score(y_test, yhat6))
```

Train set Accuracy: 0.51625

Test set Accuracy: 0.31

What about other K?

K in KNN, is the number of nearest neighbors to examine. It is supposed to be specified by the user. So, how can we choose right value for K? The general solution is to reserve a part of your data for testing the accuracy of the model. Then choose k =1, use the training part for modeling, and calculate the accuracy of prediction using all samples in your test set. Repeat this process, increasing the k, and see which k is the best for your model.

We can calculate the accuracy of KNN for different values of k.

```
Ks = 10
mean_acc = np.zeros((Ks-1))
std_acc = np.zeros((Ks-1))

for n in range(1,Ks):

    # Train Model and Predict
    neigh = KNeighborsClassifier(n_neighbors = n).fit(
        X_train, y_train)
    yhat=neigh.predict (X_test)
    mean_acc[n-1] = metrics.accuracy_score (y_test, yhat)

    std_acc[n-1]=np.std(yhat==y_test)/
                    np.sqrt(yhat.shape
[0])
mean_acc

array([0.3 , 0.29 , 0.315, 0.32 , 0.315, 0.31 , 0.335,
0.325, 0.34 ])
```

Now, we plot the model accuracy for a different number of neighbors in **Figure 13-3**.

```
plt.plot (range(1,Ks), mean_acc, 'g')
plt.fill_between(range(1,Ks), mean_acc - 1 * std_acc,
mean_acc + 1 * std_acc, alpha=0.10)
plt.fill_between(range(1,Ks),mean_acc - 3 * std_acc,
mean_acc + 3 * std_acc, alpha=0.10,color="green")
plt.legend(('Accuracy ', '+/- 1xstd','+/- 3xstd'))
plt.ylabel('Accuracy ')
plt.xlabel('Number of Neighbors (K)')
plt.tight_layout()
```

```
plt.show()
```

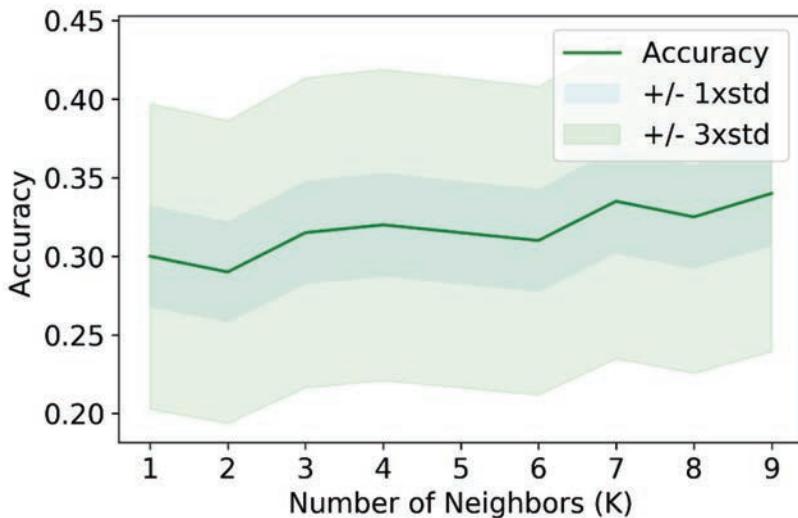


Figure 13-3. Accuracy plot for various k -values.

```
print( "The best accuracy was with", mean_acc.max(), "with  
k=", mean_acc.argmax()+1)
```

The best accuracy was with 0.34 with k= 9

14. Decision Trees

Objectives

After completing this chapter, you will be able to:

- Develop a classification model using Decision Tree Algorithm

In this chapter, you will learn a popular machine learning algorithm, Decision Trees. You will use this classification algorithm to build a model from the historical data of patients, and their response to different medications. Then you will use the trained decision tree to predict the class of an unknown patient, or to find a proper drug for a new patient.

Table of contents

1. About the dataset
2. Downloading the Data
3. Pre-processing
4. Setting up the Decision Tree
5. Modeling
6. Prediction
7. Evaluation
8. Visualization
9. Entropy

Import the Following Libraries:

- **numpy (as np)**
- **pandas**
- **DecisionTreeClassifier from sklearn.tree**

```
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
```

About the Dataset

We'll imagine that we are medical researchers compiling data for a study. we have collected data about a set of patients, all of whom suffered from the same illness. During their course of treatment, each patient responded to one of 5 medications, Drug A, Drug B, Drug C, Drug x and y.

Part of our job is to build a model to find out which drug might be appropriate for a future patient with the same illness. The features of this dataset are Age, Sex, Blood Pressure, and the Cholesterol of the patients, and the target is the drug that each patient responded to. It is a sample of multiclass classifier, and you can use the training part of the dataset to build a decision tree, and then use it to predict the class of an unknown patient, or to prescribe a drug to a new patient.

Downloading the Data

To download the data, we'll use my GitHub Jupyter | data repository.

```
path = "https://cf-courses-data.s3.us.cloud-object-
storage.appdomain.cloud/IBMDriverSkillsNetwork-
ML0101EN-SkillsNetwork/labs/Module%203/data/drug200.csv"
```

Did you know? When it comes to Machine Learning, you will likely be working with large datasets. As a business, where can you host your data?

Now, read the data using *pandas* dataframe:

```
my_data = pd.read_csv(path)
my_data[0:5]
```

	Age	Sex	BP	Cholesterol	Na_to_K	Drug
0	23	F	HIGH	HIGH	25.355	drugY
1	47	M	LOW	HIGH	13.093	drugC
2	47	M	LOW	HIGH	10.114	drugC
3	28	F	NORMAL	HIGH	7.798	drugX
4	61	F	LOW	HIGH	18.043	drugY

Let's find the size of data.

```
my_data.shape
(200, 6)
```

Pre-processing

```
Using my_data as the Drug.csv"
```

data read by *pandas*, declare the following variables:

- **X** as the **Feature Matrix** (data of my_data)

- **y** as the **response vector** (target)

Remove the column containing the target name since it doesn't contain numeric values.

```
X = my_data[['Age', 'Sex', 'BP', 'Cholesterol',
'Na_to_K']].values
X[0:5]
```

```
array([[23, 'F', 'HIGH', 'HIGH', 25.355],
       [47, 'M', 'LOW', 'HIGH', 13.093],
       [47, 'M', 'LOW', 'HIGH', 10.11399999999999],
       [28, 'F', 'NORMAL', 'HIGH', 7.79799999999999],
       [61, 'F', 'LOW', 'HIGH', 18.043]], dtype=object)
```

As you may figure out, some features in this dataset are categorical, such as **Sex** or **BP**. Unfortunately, **Sklearn** Decision Trees does not handle categorical variables. We can still convert these features to numerical values using `pandas.get_dummies()` to convert the categorical variable into dummy/indicator variables.

```
from sklearn import preprocessing
le_sex = preprocessing.LabelEncoder()
le_sex.fit(['F','M'])
X[:,1] = le_sex.transform (X[:,1])
```

```
le_BP = preprocessing.LabelEncoder()
le_BP.fit ([ 'LOW', 'NORMAL', 'HIGH'])
X[:,2] = le_BP.transform(X[:,2])
```

```
le_Chol = preprocessing.LabelEncoder()
le_Chol.fit([ 'NORMAL', 'HIGH'])
X[:,3] = le_Chol.transform (X[:,3])
```

```
X[0:5]
```

```
array([[23, 0, 0, 0, 25.355],
       [47, 1, 1, 0, 13.093],
       [47, 1, 1, 0, 10.11399999999999],
       [28, 0, 2, 0, 7.79799999999999],
       [61, 0, 1, 0, 18.043]], dtype=object)
```

Now we can fill the target variable.

```
y = my_data['Drug']
y[0:5]
```

0 drugY
1 drugC
2 drugC
3 drugX
4 drugY
Name: Drug, dtype: object

Setting up the Decision Tree

We will be using **train/test split** on our **decision tree**. Let's import `train_test_split` from `sklearn.cross_validation`.

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

Now `train_test_split` will return 4 different parameters. We will name them:

`X_trainset, X_testset, y_trainset, y_testset`

The `train_test_split` will need the parameters:

`X, y, test_size=0.3, and random_state=3.`

The `X` and `y` are the arrays required before the split, the `test_size` represents the ratio of the testing dataset, and the `random_state` ensures that we obtain the same splits.

```
X_trainset, X_testset, y_trainset, y_testset =
train_test_split (X, y, test_size=0.3, random_state=3)
```

Example

Print the shape of `X_trainset` and `y_trainset`. Ensure that the dimensions match.

```
print('Shape of X training set
{}'.format(X_trainset.shape), '&', 'Size of Y training set
{}'.format(y_trainset.shape))
```

Shape of X training set (140, 5) & Size of Y training set (140,)

Print the shape of `X_testset` and `y_testset`. Ensure that the

dimensions match.

```
print('Shape of X training set  
{0}'.format(X_testset.shape), '&', 'Size of Y training set  
{0}'.format(y_testset.shape))
```

Shape of X training set (60, 5) & Size of Y training set (60,)

Modeling

We will first create an instance of the `DecisionTreeClassifier` called `drugTree`.

Inside of the classifier, specify `criterion="entropy"` so we can see the information gain of each node.

```
drugTree = DecisionTreeClassifier(criterion="entropy",  
max_depth = 4)  
drugTree # it shows the default parameters
```

```
DecisionTreeClassifier(class_weight=None,  
criterion='entropy', max_depth=4,  
          max_features=None, max_leaf_nodes=None,  
          min_impurity_decrease=0.0, min_impurity_split=None,  
          min_samples_leaf=1, min_samples_split=2,  
          min_weight_fraction_leaf=0.0, presort=False,  
          random_state=None, splitter='best')
```

Next, we will fit the data with the training feature matrix `X_trainset` and training response vector `y_trainset`

```
drugTree.fit(X_trainset, y_trainset)
```

```
Decision  
min_impurity_decrease=0.0, min_impurity_split=None,  
      min_samples_leaf=1, min_samples_split=2,  
      min_weight_fraction_leaf=0.0, presort=False,  
      random_state=None, splitter='best')
```

Prediction

Let's make some **predictions** on the testing dataset and store it into a variable called `predTree`.

```
predTree = drugTree.predict (X_testset)
```

You can print out `predTree` and `y_testset` if you want to visually compare the predictions to the actual values.

```
print (predTree [0:5])
print (y_testset [0:5])

['drugY' 'drugX' 'drugX' 'drugX' 'drugX']
40      drugY
51      drugX
139     drugX
197     drugX
170     drugX
Name: Drug, dtype: object
```

Evaluation

Next, let's import `metrics` from `sklearn` and check the accuracy of our model.

```
from sklearn import metrics
import matplotlib.pyplot as plt
print("DecisionTrees's Accuracy: ",
metrics.accuracy_score(y_testset, predTree))
```

DecisionTrees's Accuracy: 0.9833333333333333

Accuracy classification score computes subset accuracy: the set of labels predicted for a sample must exactly match the corresponding set of labels in `y_true`.

In multilabel classification, the function returns the subset accuracy. If the entire set of predicted labels for a sample strictly matches with the true set of labels, then the subset accuracy is 1.0; otherwise, it is 0.0.

Visualization

Let's visualize the tree using a package called **PyDotPlus**. **PyDotPlus** is an improved version of the old `pydot` project that provides a Python Interface to Graphviz's Dot language

```
# Notice: You might need to uncomment and install the  
pydotplus and graphviz Libraries if you have not installed  
these before  
!conda install -c conda-forge pydotplus -y
```

```
Collecting package metadata (current_repodata.json):  
...working... done  
Solving environment: ...working... done
```

Above, we see that the algorithm has completed its task, so let's plot it to get a visualization. We'll do this with the **pydotplus** package and its function `graph_from_dot_data()`. This plot is displayed in **Figure 15-1**.

```
from io import StringIO  
import pydotplus  
import matplotlib.image as img  
from sklearn import tree  
%matplotlib inline  
  
dot_data = StringIO()  
filename = "drugtree.png"  
featureNames = my_data.columns[0:5]  
  
out = tree.export_graphviz(drugTree,  
    feature_names=featureNames, out_file=dot_data,  
    class_names= np.unique(y_trainset), filled=True,  
    special_characters=True, rotate=False)  
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())  
graph.write_png(filename)  
img = mpimg.imread(filename)  
  
plt.figure(figsize = (100, 200))  
plt.imshow(img, interpolation='nearest')
```

Entropy

Entropy is nothing but the **measure of disorder**. (You can think of it as a measure of purity as well. You'll see. I like disorder because it sounds cooler.)

The Mathematical formula for Entropy is as follows -

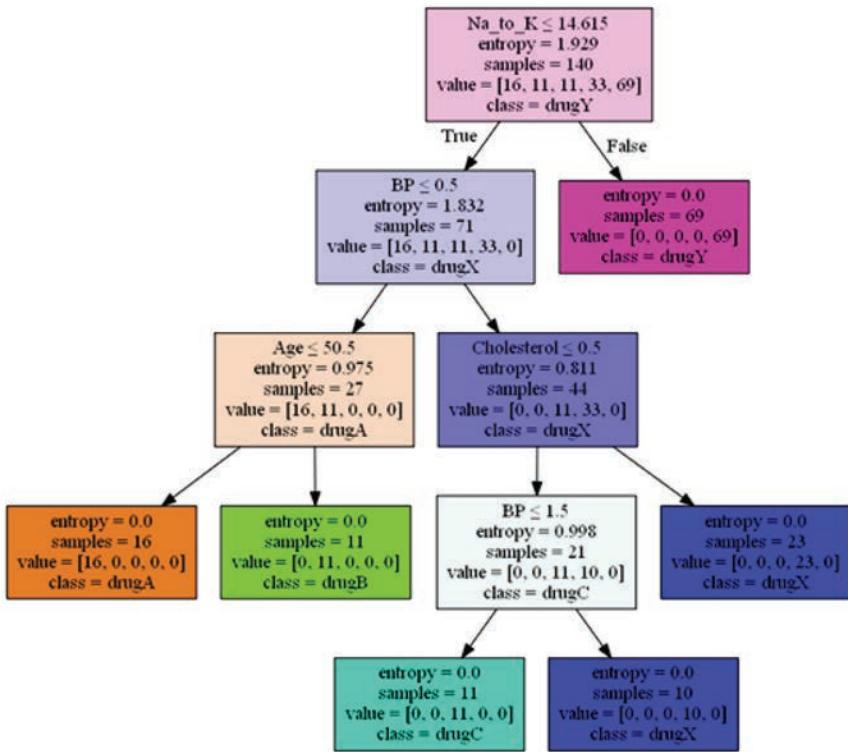


Figure 14-1. Classification tree for treatment medications.

$$E(S) = \sum_{i=1}^c -p_i \log_2 p_i$$

In the Entropy equation $E(S)$, P_i is simply the frequentist² probability of an element/class ‘i’ in our data. For simplicity’s sake let’s say we only have two classes , a positive class and a negative class. Therefore ‘i’ here could be either + or (-). So, if we had a total of 100 data points in our dataset, with 30 belonging to the positive class and 70

² According to the frequentist definition of probability, only events that are both random and repeatable, such as flipping of a coin or picking a card from a deck, have probabilities. These probabilities are equal to the long-term frequencies of such events occurring. The frequentist approach does not attach probabilities to any hypothesis or to any values that are fixed but not known (Mehta, 2019).

belonging to the negative class, then $P+$ would be 3/10 and $P-$ would be 7/10. Pretty straightforward.

If we calculate the entropy of the classes in this example using the formula above, we get:

$$-\frac{3}{10} \times \log_2\left(\frac{3}{10}\right) - \frac{7}{10} \times \log_2\left(\frac{7}{10}\right) \approx 0.88$$

The entropy here is approximately 0.88. This is considered a high entropy , a high level of disorder (meaning low level of purity). Entropy is measured between 0 and 1. (Depending on the number of classes in your dataset, entropy can be greater than 1 but it means the same thing, a very high level of disorder.

So, if you look in the bottom nodes of **Figure 15-1**, you'll see that all of the have an entropy equal to zero, that is high purity or no disorder.

15. Logistic Regression with Python

Objectives

After completing this chapter, you will be able to:

- Use scikit Logistic Regression to classify
- Understand confusion matrix

In this chapter, you will learn Logistic Regression, and then, you'll create a model for a telecommunication company, to predict when its customers will leave for a competitor, so that they can take some action to retain the customers.

Table of contents

1. About the dataset
2. Data pre-processing and selection
3. Modeling (Logistic Regression with Scikit-learn)
4. Evaluation
5. Practice

What is the difference between Linear and Logistic Regression?

While Linear Regression is suited for estimating continuous values (e.g., estimating house price), it is not the best tool for predicting the class of an observed data point. In order to estimate the class of a data point, we need some sort of guidance on what would be the **most probable class** for that data point. For this, we use **Logistic Regression**.

Recall linear regression:

As you know, **Linear regression** finds a function that relates a continuous dependent variable, y , to some predictors (independent variables x_1, x_2 , etc.). For example, simple linear regression assumes a function of the form:

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots$$

and finds the values of parameters $\theta_0, \theta_1, \theta_2$, etc, where the term θ_0 is the "intercept". It can be generally shown as:

$$\mathbf{h}\boldsymbol{\theta}(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{X}$$

Logistic Regression is a variation of Linear Regression, used when the observed dependent variable, y , is categorical. It produces a formula that predicts the probability of the class label as a function of the independent variables.

Logistic regression fits a special s-shaped curve by taking the linear regression function and transforming the numeric estimate into a probability with the following function, which is called the sigmoid function σ , as displayed in **Figure 16-1**:

$$\begin{aligned} h_{\theta}(x) &= \sigma(\theta^T X) \\ &= \frac{e^{\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots}}{1 + e^{\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots}} \end{aligned}$$

Or:

$$ProbabilityOfaClass_1 = P(Y = 1|X) = \sigma(\theta^T X) = \frac{e^{\theta^T X}}{1 + e^{\theta^T X}}$$

In this equation, $\theta^T X$ is the regression result (the sum of the variables weighted by the coefficients), exp is the exponential function and $\sigma(\theta^T X)$ is the sigmoid or logistic function, also called logistic curve. It is a common "S" shape (sigmoid curve).

So, briefly, Logistic Regression passes the input through the logistic/sigmoid but then treats the result as a probability:

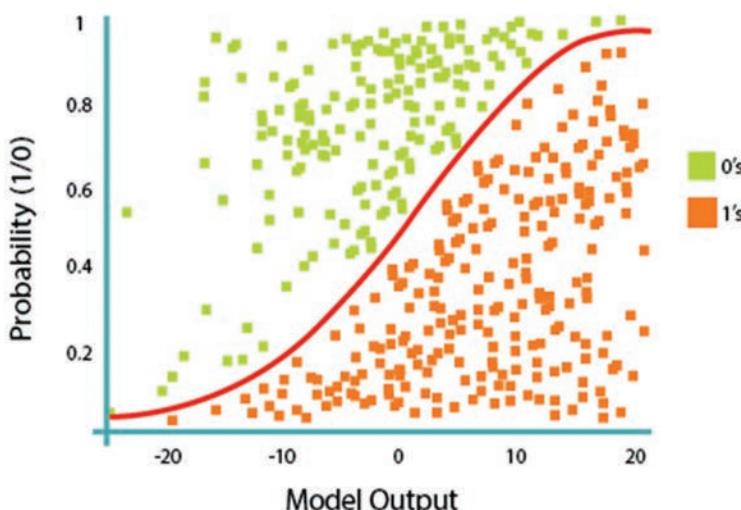


Figure 15-1. Depiction of the logistic or sigmoid curve, showing that as the model output increases, the probability of success increases accordingly.

The objective of the **Logistic Regression** algorithm, is to find the best parameters θ , for $h_{\theta}(x) = \sigma(\theta^T X)$, in such a way that the model best predicts the class of each case.

Customer churn with Logistic Regression

A telecommunications company is concerned about the number of customers leaving their land-line business for cable competitors. They need to understand who is leaving. Imagine that you are an analyst at this company and you have to find out who is leaving and why.

Let's first import required libraries:

```
import pandas as pd
import pylab as pl
import numpy as np
import scipy.optimize as opt
from sklearn import preprocessing
%matplotlib inline
import matplotlib.pyplot as plt
```

If *scikit-learn* is not installed on your system, the following code will detect it and install as necessary. If any of the `sklearn` function fails to operate, you may need to `uninstall` and `install scikit-learn`.

```
try:
    import sklearn
    print("module 'scikit-learn' is installed")
except ModuleNotFoundError:
    print("module 'scikit-learn' is not installed")
    !pip install scikit-learn==0.23.1
```

module 'scikit-learn' is installed

About the dataset

We will use a telecommunications dataset for predicting customer churn. This is a historical customer dataset where each row represents one customer. The data is relatively easy to understand, and you may uncover insights you can use immediately. Typically, it is less expensive to keep customers than acquire new ones, so the focus of this analysis is to predict the customers who will stay with

the company.

This data set provides information to help you predict what behavior will help you to retain customers. You can analyze all relevant customer data and develop focused customer retention programs.

The dataset includes information about:

- Customers who left within the last month – the column is called Churn
- Services that each customer has signed up for – phone, multiple lines, internet, online security, online backup, device protection, tech support, and streaming TV and movies
- Customer account information – how long they had been a customer, contract, payment method, paperless billing, monthly charges, and total charges
- Demographic info about customers – gender, age range, and if they have partners and dependents

Load the Telco Churn Data

Telco Churn is a hypothetical data file that concerns a telecommunications company's efforts to reduce turnover in its customer base. Each case corresponds to a separate customer and it records various demographic and service usage information. Before you can work with the data, you must use the URL to get the [ChurnData.csv](#). (Chen, 2020)

To download the data, we'll will use my GitHub Jupyter | Data repository.

```
path = "  
https://raw.githubusercontent.com/stricje1/jupyter/main/data/ChurnData.csv"
```

Did you know? When it comes to Machine Learning, you will likely be working with large datasets.

Load Data from CSV File

```
churn_df = pd.read_csv(path)  
churn_df.head()
```

	tenure	age	address	income	ed	employ	... custcat	churn	
0	11.0	33.0	7.0	136.0	5.0	5.0	...	4.0	1.0
1	33.0	33.0	12.0	33.0	2.0	0.0	...	1.0	1.0
2	23.0	30.0	9.0	30.0	1.0	2.0	...	3.0	0.0
3	38.0	35.0	5.0	76.0	2.0	10.0	...	4.0	0.0
4	7.0	35.0	14.0	80.0	2.0	15.0	...	3.0	0.0

5 rows × 28 columns

Data pre-processing and selection

Let's select some features for the modeling. Also, we change the target data type to be an integer, as it is a requirement by the skitlearn algorithm:

```
churn_df = churn_df[['tenure', 'age', 'address', 'income',
'ed', 'employ', 'equip', 'callcard', 'wireless', 'churn']]
churn_df['churn'] = churn_df['churn'].astype ('int')
churn_df.head()
```

	tenure	age	address	income	ed	... callcard	wireless	churn
0	11.0	33.0	7.0	136.0	5.0	... 1.0	1.0	1
1	33.0	33.0	12.0	33.0	2.0	... 0.0	0.0	1
2	23.0	30.0	9.0	30.0	1.0	... 0.0	0.0	0
3	38.0	35.0	5.0	76.0	2.0	... 1.0	1.0	0
4	7.0	35.0	14.0	80.0	2.0	... 1.0	0.0	0

Example

How many rows and columns are in this dataset in total? What are the names of columns?

```
churn_df.shape
```

```
(200, 10)
```

Let's define X, and y for our dataset:

```
X = np.asarray(churn_df[['tenure', 'age', 'address',
'income', 'ed', 'employ', 'equip']])
X[0:5]
```

```
array([[ 11.,   33.,    7.,  136.,     5.,     5.,     0.],
       [ 33.,   33.,   12.,   33.,     2.,     0.,     0.],
       [ 23.,   30.,    9.,   30.,     1.,     2.,     0.],
       [ 38.,   35.,    5.,   76.,     2.,    10.,    1.],
       [  7.,   35.,   14.,   80.,     2.,    15.,     0.]])
```

```
y = np.asarray(churn_df['churn'])
y[0:5]
```

```
array([1, 1, 0, 0, 0])
```

Also, we normalize the dataset:

```
from sklearn import preprocessing
X = preprocessing.StandardScaler().fit(X).transform(X)
X[0:5]
```

```
array([[-1.14, -0.63, -0.46,  0.48,  1.7 , -0.58, -0.86],
       [-0.12, -0.63,  0.03, -0.33, -0.64, -1.14, -0.86],
       [-0.58, -0.86, -0.26, -0.35, -1.42, -0.92, -0.86],
       [ 0.12, -0.47, -0.66,  0.01, -0.64, -0.03,  1.16],
       [-1.32, -0.47,  0.23,  0.04, -0.64,  0.53, -0.86]])
```

Train/Test dataset

We split our dataset into train and test set:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split (X,
y,
    test_size = 0.2, random_state = 4)
print ('Train set:', X_train.shape, y_train.shape)
print ('Test set:', X_test.shape, y_test.shape)
```

```
Train set: (160, 7) (160,)
Test set: (40, 7) (40,)
```

Modeling (Logistic Regression with Scikit-learn)

Let's build our model using **LogisticRegression** from the **Scikit-learn** package. This function implements logistic regression and can use different numerical optimizers to find parameters, including

'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga' solvers. You can find extensive information about the pros and cons of these optimizers if you search it in the internet.

The version of Logistic Regression in **Scikit-learn**, support regularization. Regularization is a technique used to solve the overfitting problem of machine learning models. **C** parameter indicates **inverse of regularization strength** which must be a positive float. Smaller values specify stronger regularization. Now let's fit our model with train set:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
LR = LogisticRegression(C=0.01,
                        solver='liblinear').fit (X_train, y_train)
LR
LogisticRegression(C=0.01, solver='liblinear')
```

Now we can predict using our test set:

```
yhat = LR.predict (X_test)
yhat
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0,
       1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
       0, 0, 1, 0, 0, 0])
```

predict_proba returns estimates for all classes, ordered by the label of classes. So, the first column is the probability of class 0, $P(Y=0|X)$, and second column is probability of class 1, $P(Y=1|X)$:

```
yhat_prob = LR.predict_proba(X_test)
yhat_prob
```

```
array([[0.54, 0.46],
       [0.61, 0.39],
       [0.56, 0.44],
       [0.63, 0.37],
       [0.56, 0.44],
       :
       [0.64, 0.36],
       [0.4 , 0.6 ],
       [0.52, 0.48],
       [0.66, 0.34],
       [0.51, 0.49]])
```

Evaluation

Jaccard index

Let's try the Jaccard index for accuracy evaluation. we can define `jaccard` as the size of the intersection divided by the size of the union of the two label sets. If the entire set of predicted labels for a sample strictly matches with the true set of labels, then the subset accuracy is 1.0; otherwise, it is 0.0.

```
from sklearn.metrics import jaccard_score  
jaccard_score(y_test, yhat, pos_label = 0)
```

0.7058823529411765

confusion matrix

Another way of looking at the accuracy of the classifier is to look at **confusion matrix**.

```
from sklearn.metrics import classification_report,  
    confusion_matrix  
import itertools  
  
# Confusion matrix definition  
  
def plot_confusion_matrix(cm, classes,  
                        normalize=False,  
                        title='Confusion matrix',  
                        cmap=plt.cm.Blues):  
    """  
    This function prints and plots the confusion matrix.  
    Normalization can be applied by setting  
    `normalize=True`.  
    """  
    if normalize:  
        cm = cm.astype ('float') / cm.sum(axis=1)  
        [:, np.newaxis]  
        print("Normalized confusion matrix")  
    else:  
        print('Confusion matrix, without normalization')  
  
    print(cm)
```

```

plt.imshow(cm, interpolation='nearest', cmap=cmap)
plt.title(title)
plt.colorbar()
plt.xticks(tick_marks, classes, rotation=45)
plt.yticks(tick_marks, classes)
tick_marks = np.arange(len(classes))

fmt = '.2f' if normalize else 'd'
thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]),
                               range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], fmt),
              horizontalalignment="center",
              color="white" if cm[i, j] > thresh else
              "black")

plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')

print(confusion_matrix(y_test, yhat, labels=[1,0]))
[[ 6  9]
 [ 1 24]]

```

Visualization of the confusion matrix as a heatmap is provided in **Figure 16-2**.

```

# Compute confusion matrix
cnf_matrix = confusion_matrix (y_test, yhat, labels=[1,0])
np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure(dpi=200)
plot_confusion_matrix(cnf_matrix,
                      classes=['churn=1', 'churn=0'],
                      normalize = False,
                      title='Confusion matrix')

```

Confusion matrix, without normalization
[[6 9]
 [1 24]]

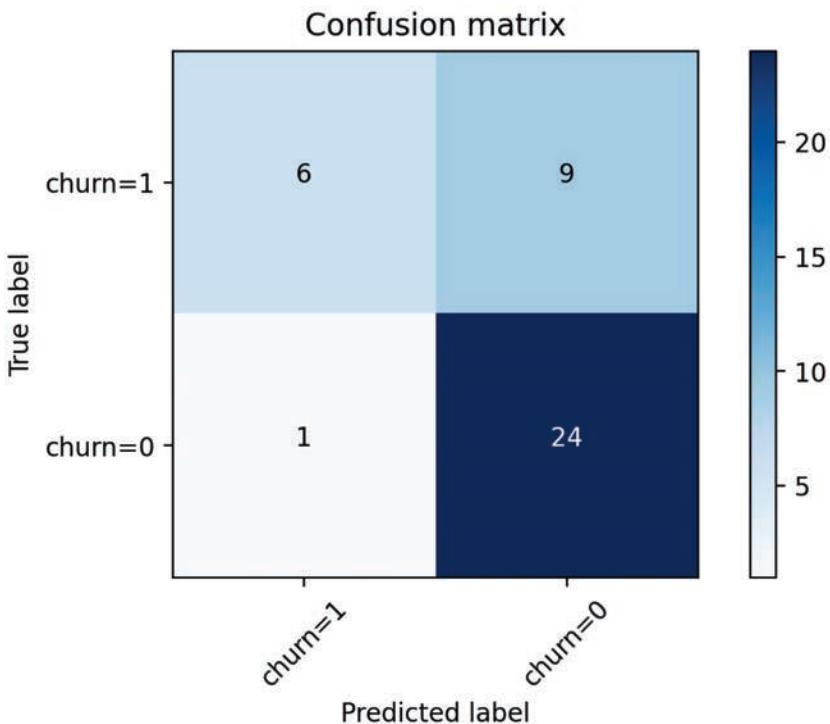


Figure 15-2. Confusion matrix showing the strength of the predictions, the darker the shade of blue, the better the prediction.

Let's look at first row. The first row is for customers whose actual churn value in the test set is 1. As you can calculate, out of 40 customers, the churn value of 15 of them is 1. Out of these 15 cases, the classifier correctly predicted 6 of them as 1, and 9 of them as 0.

This means, for 6 customers, the actual churn value was 1 in test set and classifier also correctly predicted those as 1. However, while the actual label of 9 customers was 1, the classifier predicted those as 0, which is not very good. We can consider it as the error of the model for first row.

What about the customers with churn value 0? Let's look at the second row. It looks like there were 25 customers whom their churn value was 0.

The classifier correctly predicted 24 of them as 0, and one of them wrongly as 1. So, it has done a good job in predicting the customers

with churn value 0. A good thing about the confusion matrix is that it shows the model's ability to correctly predict or separate the classes. In a specific case of the binary classifier, such as this example, we can interpret these numbers as the count of true positives, false positives, true negatives, and false negatives.

```
print(classification_report(y_test, yhat))
```

	precision	recall	f1-score	support
0	0.73	0.96	0.83	25
1	0.86	0.40	0.55	15
accuracy			0.75	40
macro avg	0.79	0.68	0.69	40
weighted avg	0.78	0.75	0.72	40

Based on the count of each section, we can calculate precision and recall of each label:

- **Precision** is a measure of the accuracy provided that a class label has been predicted. It is defined by:

$$precision = \frac{TP}{(TP + FP)}$$

- **Recall** is the true positive rate. It is defined as:

$$Recall = \frac{TP}{(TP + FN)}$$

So, we can calculate the precision and recall of each class.

F1 score: Now we are in the position to calculate the F1 scores for each label based on the precision and recall of that label.

The F1 score is the harmonic average of the precision and recall, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0. It is a good way to show that a classifier has a good value for both recall and precision.

Finally, we can tell the average accuracy for this classifier is the average of the F1-score for both labels, which is 0.72 in our case.

log loss

Now, let's try **log loss** for evaluation. In logistic regression, the output can be the probability of customer churn is yes (or equals to 1). This probability is a value between 0 and 1. Log loss(Logarithmic loss) measures the performance of a classifier where the predicted output is a probability value between 0 and 1.

```
from sklearn.metrics import log_loss  
log_loss(y_test, yhat_prob)
```

0.6017092478101187

Practice

Try to build Logistic Regression model again for the same dataset, but this time, use different `_solver_` and `_regularization_` values? What is new `_logLoss_` value?

```
LR2 = LogisticRegression (C=0.01,  
solver='sag').fit(X_train, y_train)  
yhat_prob2 = LR2.predict_proba (X_test)  
print ("LogLoss: : %.2f" % log_loss(y_test, yhat_prob2))
```

LogLoss: : 0.61

16. SVM (Support Vector Machines)

Objectives

After completing this chapter, you will be able to:

- Use scikit-learn to Support Vector Machine to classify

In this notebook, you will use SVM (Support Vector Machines) to build and train a model using human cell records, and classify cells to whether the samples are benign or malignant.

SVM works by mapping data to a high-dimensional feature space so that data points can be categorized, even when the data are not otherwise linearly separable. A separator between the categories is found, then the data is transformed in such a way that the separator could be drawn as a hyperplane. Following this, characteristics of new data can be used to predict the group to which a new record should belong.

Table of contents

1. Load the Cancer data
2. Modeling
3. Evaluation
4. Example

```
import pandas as pd
import pylab as pl
import numpy as np
import scipy.optimize as opt
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline
```

Load the Cancer data

The example is based on a dataset that is publicly available from the UCI Machine Learning Repository (Asuncion & Newman, 2007) [<http://mlearn.ics.uci.edu/MLRepository.html>]. The dataset consists of several hundred human cell sample records, each of which

contains the values of a set of cell characteristics. The fields in each record are:

Field name	Description
ID	Clump thickness
Clump	Clump thickness
UnifSize	Uniformity of cell size
UnifShape	Uniformity of cell shape
MargAdh	Marginal adhesion
SingEpiSize	Single epithelial cell size
BareNuc	Bare nuclei
BlandChrom	Bland chromatin
NormNucl	Normal nucleoli
Mit	Mitoses
Class	Benign or malignant

For the purposes of this example, we're using a dataset that has a relatively small number of predictors in each record. To download the data, we will use my GitHub *Jupyter / Data* repository.

```
path =  
"https://raw.githubusercontent.com/stricje1/jupyter/main/d  
ata/cell_samples.csv"
```

Load Data from CSV File

```
cell_df = pd.read_csv(path)  
cell_df.head()
```

ID	Clump	UnifSize	UnifShape	...	BareNuc	Mit	Class	
0	1000025	5	1	1	...	1	1	2
1	1002945	5	4	4	...	10	1	2
2	1015425	3	1	1	...	2	1	2
3	1016277	6	8	8	...	4	1	2
4	1017023	4	1	1	...	1	1	2

The ID field contains the patient identifiers. The characteristics of the

cell samples from each patient are contained in fields `Clump` (Clump Thickness) to `Mit` (Mitoses). The values are graded from 1 to 10, with 1 being the closest to benign.

The `Class` field contains the diagnosis, as confirmed by separate medical procedures, as to whether the samples are benign (value = 2) or malignant (value = 4).

Let's look at the distribution of the classes based on Clump thickness and Uniformity of cell size, as displayed in **Figure 17-1**.

```
ax = cell_df[cell_df['Class'] == 4][0:50].plot(kind='scatter', x='Clump', y='UnifSize', color='DarkBlue', label='malignant');  
cell_df[cell_df['Class'] == 2][0:50].plot(kind='scatter', x='Clump', y='UnifSize', color='Yellow', label='benign', ax=ax);  
plt.show()
```

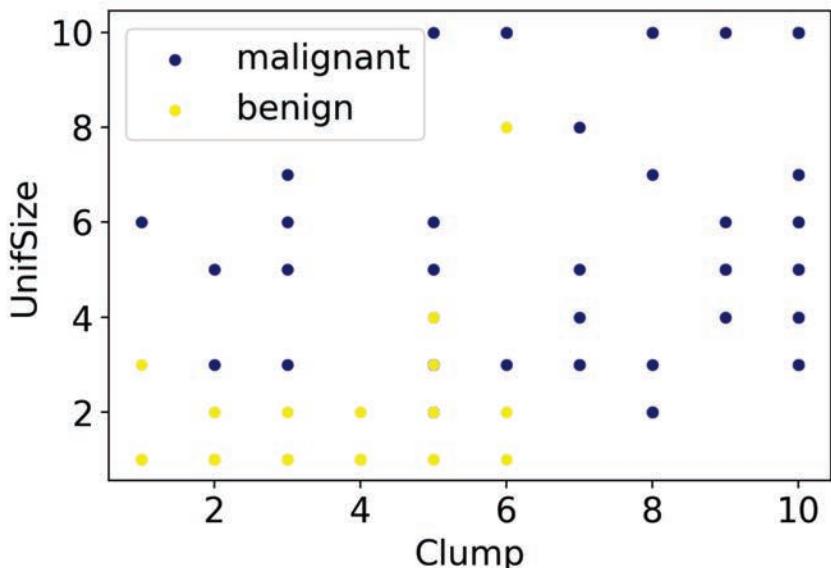


Figure 16-1. Distribution of cell classes based on uniformity and thickness, for benign and malignant cells.

Data pre-processing and selection

Let's first look at columns data types:

```
cell_df.dtypes
```

```
ID          int64
Clump       int64
UnifSize    int64
UnifShape   int64
MargAdh     int64
SingEpiSize int64
BareNuc     object
BlandChrom  int64
NormNucl    int64
Mit         int64
Class       int64
dtype: object
```

It looks like the **BareNuc** column includes some values that are not numerical. We can drop those rows:

```
cell_df = cell_df[pd.to_numeric(cell_df['BareNuc'],
                                 errors='coerce').notnull()]
cell_df['BareNuc'] = cell_df['BareNuc'].astype('int')
cell_df.dtypes
```

```
ID          int64
Clump       int64
UnifSize    int64
UnifShape   int64
MargAdh     int64
SingEpiSize int64
BareNuc     int32
BlandChrom  int64
NormNucl    int64
Mit         int64
Class       int64
dtype: object
```

```
feature_df = cell_df[['Clump', 'UnifSize', 'UnifShape',
                      'MargAdh', 'SingEpiSize', 'BareNuc', 'BlandChrom',
                      'NormNucl', 'Mit']]
X = np.asarray(feature_df)
X[0:5]
```

```
array([[ 5,  1,  1,  1,  2,  1,  3,  1,  1],
       [ 5,  4,  4,  5,  7, 10,  3,  2,  1],
       [ 3,  1,  1,  1,  2,  2,  3,  1,  1],
       [ 6,  8,  8,  1,  3,  4,  3,  7,  1],
       [ 4,  1,  1,  3,  2,  1,  3,  1,  1]], dtype=int64)
```

We want the model to predict the value of Class (that is, benign (=2) or malignant (=4)). As this field can have one of only two possible values, we need to change its measurement level to reflect this.

```
cell_df['Class'] = cell_df['Class'].astype ('int')
y = np.asarray(cell_df['Class'])
y [0:5]
array([2, 2, 2, 2, 2])
```

Train/Test dataset

We split our dataset into train and test set:

```
X_train, X_test, y_train, y_test = train_test_split ( x,
y,
    test_size=0.2, random_state=4)
print ('Train set:', X_train.shape, y_train.shape)
print ('Test set:', X_test.shape, y_test.shape)

Train set: (546, 9) (546,)
Test set: (137, 9) (137,)
```

Modeling (SVM with Scikit-learn)

The SVM algorithm offers a choice of kernel functions for performing its processing. Basically, mapping data into a higher dimensional space is called kernelling. The mathematical function used for the transformation is known as the kernel function, and can be of different types, such as:

1. 1.Linear
2. 2.Polynomial
3. 3.Radial basis function (RBF)
4. 4.Sigmoid

Each of these functions has its characteristics, its pros and cons, and its equation, but as there's no easy way of knowing which function performs best with any given dataset. We usually choose different functions in turn and compare the results. We'll use the default, **Radial Basis Function (RBF)** for this chapter.

An RBF is a real-valued function φ whose value depends only on the distance between the input and some fixed point, either the origin, so that $\varphi(x) = \hat{\varphi}(\|x\|)$ or some other fixed point c , called a center,

so that $\varphi(x) = \hat{\varphi}(\|x - c\|)$. Any function φ that satisfies the property $\varphi(x) = \hat{\varphi}(\|x\|)$ is a radial function. The distance is usually Euclidean distance, although other metrics are sometimes used.

```
from sklearn import svm
clf = svm.SVC(kernel='rbf')
clf.fit (X_train, y_train)
```

After being fitted, the model can then be used to predict new values:

```
yhat = clf.predict (X_test)
yhat [0:5]
array([2, 4, 2, 4, 2])
```

Evaluation

Classification Accuracy and its Limitations

Although we discussed this in the previous chapter, do so here from a different perspective. We have stated that classification accuracy is the ratio of correct predictions to total predictions made.

$$\text{classification accuracy} = \frac{\text{correct predictions}}{\text{total predictions}}$$

Classification accuracy can also easily be turned into a misclassification rate or error rate by inverting the value, such as:

$$\text{error rate} = \left(1 - \left(\frac{\text{correct predictions}}{\text{total predictions}}\right)\right) \times 100$$

Classification accuracy is a great place to start, but often encounters problems in practice. The main problem with classification accuracy is that it hides the detail you need to better understand the performance of your classification model. There are two examples where you are most likely to encounter this problem:

1. When your data has more than 2 classes. With 3 or more classes you may get a classification accuracy of 80%, but you don't know if that is because all classes are being predicted equally well or whether one or two classes are being neglected by the model.
2. When your data does not have an even number of classes. You

may achieve accuracy of 90% or more, but this is not a good score if 90 records for every 100 belong to one class and you can achieve this score by always predicting the most common class value.

Classification accuracy can hide the detail you need to diagnose the performance of your model.

What is a Confusion Matrix?

A confusion matrix is a summary of prediction results for a classification problem. The number of correct and incorrect predictions are summarized with count values and broken down by each class (Pedregosa, et al., 2011). This is the key to the confusion matrix.

The confusion matrix shows the ways in which your classification model is confused when it makes predictions. It gives us insight not only into the errors being made by your classifier but more importantly the types of errors that are being made. It is this breakdown that overcomes the limitation of using classification accuracy alone.

So, how do we interpret a confusion matrix? There are four possible outcomes for any classification problem, even when dealing with more than two classes (Haghghi, Jasemi, Hessab, & Zolanvari, 2018).

Outcome 1: True Positive:

Interpretation: You predicted positive and it's true.

Example: You predicted that a woman is pregnant and she actually is.

Outcome 2: True Negative:

Interpretation: You predicted negative and it's true.

Example: You predicted that a man is not pregnant and he actually is not.

Outcome 3: False Positive: (Type 1 Error)

Interpretation: You predicted positive and it's false.

Example: You predicted that a man is pregnant but he actually is

not.

Outcome 4: False Negative: (Type 2 Error)

Interpretation: You predicted negative and it's false.

Example: You predicted that a woman is not pregnant but she actually is.

Measuring Accuracy

We can use these results to evaluate model accuracy.

Measure 1: Recall (Strickland, 2020)

Using the confusion matrix as reference, recall is calculated by:

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

The above equation can be explained by saying, from all the positive classes, how many we predicted correctly. Recall should be high as possible.

Measure 2: Precision

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

The above equation can be explained by saying, from all the classes we have predicted as positive, how many are actually positive. Precision should be high as possible.

Measure 3: F-Score

From all the classes (positive and negative), how many of them we have predicted correctly gives use the F-score or accuracy. Accuracy should be high as possible.

$$F1\ Score = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}}$$

It is difficult to compare two models with low precision and high recall or vice versa. So, to make them comparable, we use F-Score. F-score helps to measure Recall and Precision at the same time (Strickland, 2020). It uses Harmonic Mean in place of Arithmetic Mean by punishing the extreme values more.

```

from sklearn.metrics import classification_report,
confusion_matrix
import itertools

def plot_confusion_matrix(cm, classes,
                        normalize=False,
                        title='Confusion matrix',
                        cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting normalize = True.
    """
    if normalize:
        cm = cm.astype ('float') / cm.sum(axis=1)
        [:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')
    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape [0]),
                                range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color= "white" if cm[i, j] >
thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

```

Compute confusion matrix

```
cnf_matrix = confusion_matrix(y_test, yhat, labels= [2,4])
np.set_printoptions(precision=2)
print (classification_report(y_test, yhat))
# Plot non-normalized confusion matrix
plt.figure(dpi=200)
plot_confusion_matrix(cnf_matrix,
classes=['Benign(2)', 'Malignant(4)'],normalize= False,
title='Confusion matrix')
```

	precision	recall	f1-score	support
2	1.00	0.94	0.97	90
4	0.90	1.00	0.95	47
accuracy			0.96	137
macro avg	0.95	0.97	0.96	137
weighted avg	0.97	0.96	0.96	137

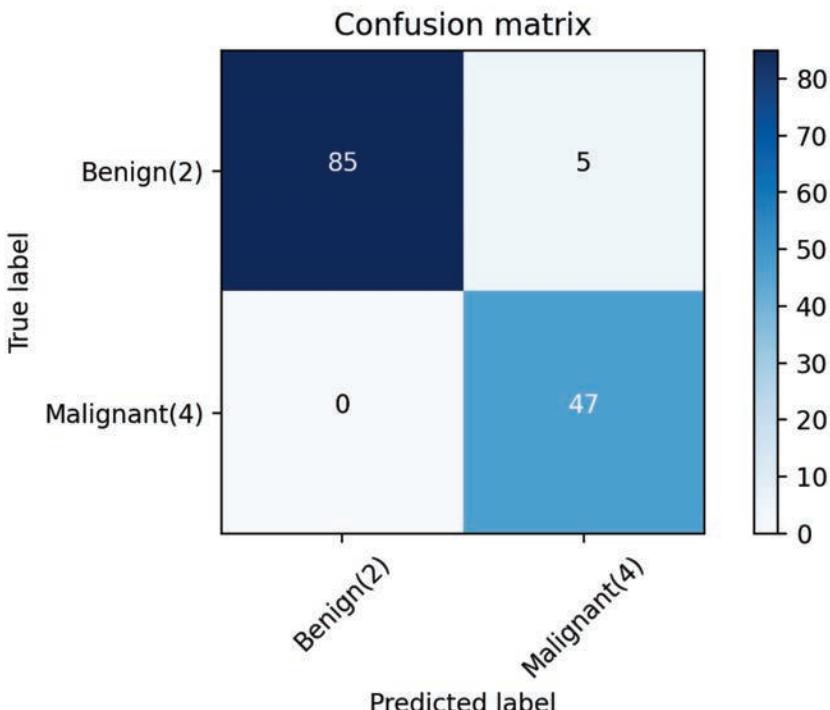


Figure 16-2. Confusion matrix for the SVM model, showing good predictive power, i.e., few false-positives (5) and no false-negatives.

You can also easily use the **f1_score** from sklearn library:

```
from sklearn.metrics import f1_score  
f1_score(y_test, yhat, average='weighted')  
0.9639038982104676
```

Let's try the jaccard index for accuracy:

```
from sklearn.metrics import jaccard_score  
jaccard_score(y_test, yhat, pos_label=2)  
0.9444444444444444
```

Example

Can you rebuild the model, but this time with a `_linear_` kernel? You can use `_kernel='linear'` option, when you define the svm. How the accuracy changes with the new kernel function?

```
clf2 = svm.SVC(kernel='linear')  
clf2.fit(X_train, y_train)  
yhat2 = clf2.predict (X_test)  
print("Avg F1-score: %.4f" % f1_score(y_test, yhat2,  
average='weighted'))  
print("Jaccard score: %.4f" % jaccard_score(y_test, yhat2,  
pos_label=2))  
  
Avg F1-score: 0.9639  
Jaccard score: 0.9444
```


17. K-Means Clustering

Objectives

After completing this chapter, you will be able to:

- Use scikit-learn's K-Means Clustering to cluster data

Introduction

There are many models for **clustering** out there. In this notebook, we will be presenting the model that is considered one of the simplest models amongst them. Despite its simplicity, the **K-means** is vastly used for clustering in many data science applications, it is especially useful if you need to quickly discover insights from **unlabeled data**. In this notebook, you will learn how to use k-Means for customer segmentation.

Some real-world applications of k-means:

- Customer segmentation
- Understand what the visitors of a website are trying to accomplish
- Pattern recognition
- Machine learning
- Data compression

In this notebook we practice k-means clustering with 2 examples:

- k-means on a random generated dataset
- Using k-means for customer segmentation

Table of contents

- k-Means on a randomly generated dataset
 1. Setting up K-Means
 2. Creating the Visual Plot
- Customer Segmentation with K-Means
 1. Pre-processing
 2. Modeling
 3. Insights

Import libraries

Let's first import the required libraries. Also run `%matplotlib inline` since we will be plotting in this section.

```
import random
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
%matplotlib inline
```

k-Means on a randomly generated dataset

Let's create our own dataset for this lab!

First, we need to set a random seed. Use `NumPy's random.seed()` function, where the seed will be set to **0**.

```
np.random.seed(0)
```

Next we will be making *random clusters* of points by using the **make_blobs** class. The **make_blobs** class can take in many inputs, but we will be using these specific ones.

Input

- **n_samples:** The total number of points equally divided among clusters.
 - Value will be: 5000
- **centers:** The number of centers to generate, or the fixed center locations.
 - Value will be: [[4, 4], [-2, -1], [2, -3],[1,1]]
- **cluster_std:** The standard deviation of the clusters.
 - Value will be: 0.9

Output

- **X:** Array of shape [n_samples, n_features]. (Feature Matrix)
 - The generated samples.
- **y:** Array of shape [n_samples]. (Response Vector)
 - The integer labels for cluster membership of each sample.

```
X, y = make_blobs(n_samples=5000, centers= [[4,4], [-2, -1], [2, -3], [1, 1]], cluster_std=0.9)
```

Here, we create and display the scatter plot of the randomly generated data in **Figure 18-1**.

```
plt.scatter(X[:, 0], X[:, 1], marker='.'
```

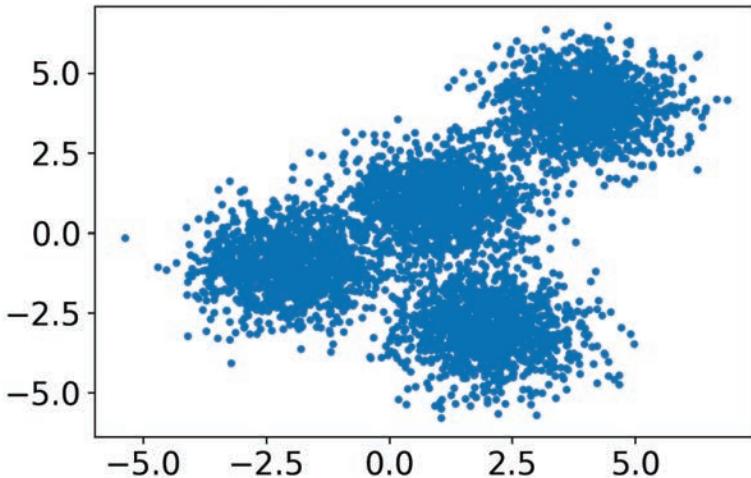


Figure 17-1. A scatterplot of randomly distributed data forming four possible clusters.

Setting up K-Means

Now that we have our random data, let's set up our K-Means Clustering.

The **KMeans** class has many parameters that can be used, but we will be using these three:

- **init**: Initialization method of the centroids.
 - Value will be: "k-means++"
 - k-means++: Selects initial cluster centers for k-mean clustering in a smart way to speed up convergence.
- **n_clusters**: The number of clusters to form as well as the number of centroids to generate.
 - Value will be: 4 (since we have 4 centers)

- **n_init**: Number of times the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.
 - Value will be: 12

Initialize KMeans with these parameters, where the output parameter is called **k_means**.

```
k_means = KMeans(init = "k-means++", n_clusters = 4,  
n_init = 12)
```

Now let's fit the **KMeans** model with the feature matrix we created above, **X**.

```
k_means.fit (X)  
KMeans(n_clusters=4, n_init=12)
```

Now let's grab the labels for each point in the model using **KMeans' .labels_** attribute and save it as **k_means_labels**.

```
k_means_labels = k_means.labels_  
k_means_labels  
array([0, 3, 3, ..., 1, 0, 0])
```

We will get the coordinates of the cluster centers using **KMeans' .cluster_centers_** and save it as **k_means_cluster_centers**.

```
k_means_cluster_centers = k_means.cluster_centers_  
k_means_cluster_centers  
array([[-2.03743147, -0.99782524],  
      [ 3.97334234,  3.98758687],  
      [ 0.96900523,  0.98370298],  
      [ 1.99741008, -3.01666822]])
```

Creating the Visual Plot

So now that we have the random data generated and the KMeans model initialized, let's plot them and see what it looks like!

Please read through the code and comments to understand how to plot the model. One outcome of the code is the cluster plot in **Figure 18-1**.

```

# Initialize the plot with the specified dimensions.
fig = plt.figure(figsize= (6, 4))

# Colors uses a color map or colormap, which will produce
# an array of colors based on the number of labels there
# are. We use set(k_means_labels) to get the unique labels.
colors = plt.cm.Spectral(np.linspace(0, 1,
len(set(k_means_labels)))))

# Create a plot
ax = fig.add_subplot(1, 1, 1)

# For loop that plots the data points and centroids.
# k will range from 0-3, which will match the possible
clusters that each
# Data point is in.
for k, col in zip(range(len([[4,4], [-2, -1], [2, -3], [1,
1]])), colors):

    # Create a list of all data points, where the data
    # points that are in the cluster (ex. cluster 0) are labeled
    # as true, else they are labeled as false.
    my_members = (k_means_labels == k)

    # Define the centroid, or cluster center.
    cluster_center = k_means_cluster_centers[k]

    # Plots the datapoints with color col.
    ax.plot(X[my_members, 0], X[my_members, 1], 'w',
markerfacecolor=col, marker='.')

    # Plots the centroids with specified color, but with a
    # darker outline
    ax.plot(cluster_center[0], cluster_center[1], 'o',
markerfacecolor=col, markeredgecolor='k', markersize=6)

# Title of the plot
ax.set_title('KMeans')

# Remove x-axis ticks
ax.set_xticks(())

```

```

# Remove y-axis ticks
ax.set_yticks(())

# Show the plot
plt.show()

```

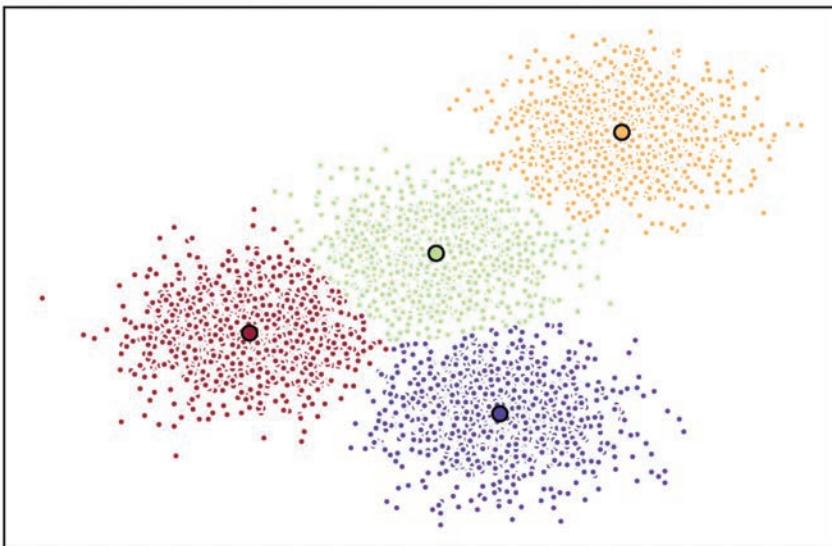


Figure 17-2. Color-coded K-means cluster of the randomly generated data used in **Figure 18-1**.

Example

Try to cluster the above dataset into 3 clusters, using the same data as above (do not generate new data). This plot appears in **Figure 18-3**.

```

k_means3 = KMeans (init = "k-means++", n_clusters = 3,
n_init = 12)
k_means3.fit (X)
fig = plt.figure(figsize= (6, 4))
colors = plt.cm.Spectral(np.linspace(0, 1,
len(set(k_means3.labels_))))
ax = fig.add_subplot(1, 1, 1)
for k, col in zip(range(len(k_means3.cluster_centers_)),
colors):
    my_members = (k_means3.labels_ == k)
    cluster_center = k_means3.cluster_centers_[k]

```

```

ax.plot(X[my_members, 0], X[my_members, 1], 'w',
        markerfacecolor=col, marker='.')
ax.plot(cluster_center[0], cluster_center[1], 'o',
        markerfacecolor=col, markeredgecolor='k', markersize=6)
plt.show()

```

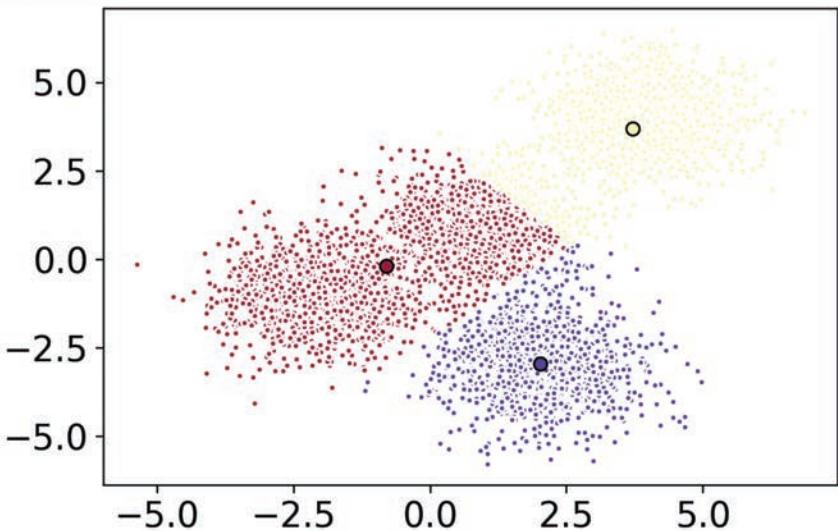


Figure 17-3. Color coded cluster of the data using for **Figure 18-2** but with only three clusters.

Customer Segmentation with K-Means

Imagine that you have a customer dataset, and you need to apply customer segmentation on this historical data. Customer segmentation is the practice of partitioning a customer base into groups of individuals that have similar characteristics. It is a significant strategy as a business can target these specific groups of customers and effectively allocate marketing resources. For example, one group might contain customers who are high-profit and low-risk, that is, more likely to purchase products, or subscribe for a service. A business task is to retain those customers. Another group might include customers from non-profit organizations and so on.

Let's download the dataset. To download the data, we will use my GitHub Jupyter Data repository.

```

path =
"https://raw.githubusercontent.com/stricje1/jupyter/main/d
ata/Cust_Segmentation.csv"

```

Load Data From CSV File

Before you can work with the data, you must use the URL to get the Cust_Segmentation.csv.

```

import pandas as pd
cust_df = pd.read_csv(path)
cust_df.head()

```

Customer			Years		Card		Other		Debt	
	Id	Age	Edu	Employed	Income	Debt	Debt	Defaulted	Income	Ratio
1	41	2		6	19.	0.12	1.07	0.0	6.3	
2	47	1		26	100.	4.58	8.22	0.0	12.8	
3	33	2		10	57.	6.11	5.80	1.0	20.9	
4	29	2		4	19.	0.68	0.52	0.0	6.3	
5	47	1		31	253.	9.31	8.91	0.0	7.2	

Pre-processing

As you can see, **Address** in this dataset is a categorical variable. The k-means algorithm isn't directly applicable to categorical variables because the Euclidean distance function isn't really meaningful for discrete variables. So, let's drop this feature and run clustering.

```

df = cust_df.drop ('Address', axis=1)
df.head()

```

Customer			Years		Card		Other		Debt	
	Id	Age	Edu	Employed	Income	Debt	Debt	Defaulted	Income	Ratio
1	41	2		6	19.	0.12	1.07	0.0	6.3	
2	47	1		26	100.	4.58	8.22	0.0	12.8	
3	33	2		10	57.	6.11	5.80	1.0	20.9	
4	29	2		4	19.	0.68	0.52	0.0	6.3	
5	47	1		31	253.	9.31	8.90	0.0	7.2	

Normalizing over the standard deviation

Now let's normalize the dataset. But why do we need normalization in the first place? Normalization is a statistical method that helps

mathematical-based algorithms to interpret features with different magnitudes and distributions equally. We use `StandardScaler()` to normalize our dataset.

```
from sklearn.preprocessing import StandardScaler
X = df.values[:,1:]
X = np.nan_to_num(X)
Clus_dataSet = StandardScaler().fit_transform (X)
Clus_dataSet
array([[ 0.7429,  0.3121, -0.3788, ..., -0.5905, -0.5238,
       -0.5765],
       [ 1.4895, -0.7663,  2.5737, ...,  1.5130, -0.5238,
        0.3914],
       [-0.2525,  0.3121,  0.2117, ...,  0.8017,  1.9091,
        1.5976],
       ... : : : : : : :
       [-1.2480,  2.4691, -1.2645, ...,  0.0386,  1.9091,
        3.4589],
       [-0.3769, -0.7663,  0.5070, ..., -0.7015, -0.5238,
        -1.0828],
       [ 2.1116, -0.7663,  1.0975, ...,  0.1646, -0.5238,
        -0.2340 ]])
```

Modeling

In our example (if we didn't have access to the k-means algorithm), it would be the same as guessing that each customer group would have certain age, income, education, etc., with multiple tests and experiments. However, using the K-means clustering we can do all this process much easier.

Let's apply k-means on our dataset, and take a look at cluster labels.

```
clusterNum = 3
k_means = KMeans (init = "k-means++",
                  n_clusters = clusterNum, n_init = 12)
k_means.fit (X)
labels = k_means.labels_
print(labels)
```

```
[0 2 0 0 1 2 0 2 0 2 2 0 0 0 0 0 0 2 0 0 0 2 2 2 0 0 2 0 2 0 2 0 0 0 0 0 0
0 0 2 0 2 0 1 0 2 0 0 0 2 2 0 0 2 2 0 0 0 2 0 2 0 2 2 0 0 2 0 0 0 2 2 0
0 0 0 0 2 0 2 2 1 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 2 0]
```

```

0000000200000020000000000200000002020
000000202202002000000020000000200020
0000200202002102000000120000200220202
    :           :           :
2000201002000000020002000002002000000
0020020202200020200000200002200220000
02000020000000000002022020220020000022
0000000200000012200000002000000200000
00000000000200000000000000000000000002]

```

Insights

We assign the labels to each row in the dataframe.

```

df["Clus_km"] = labels
df.head(5)

```

Customer Id	Years				Card Debt	Other Debt	Debt Income Ratio	Clus_km
	Age	Edu	Employed	Income				
1	41	2	6	19	0.124	1.073	6.3	0
2	47	1	26	100	4.582	8.218	12.8	2
3	33	2	10	57	6.111	5.802	20.9	0
4	29	2	4	19	0.681	0.516	6.3	0
5	47	1	31	253	9.308	8.908	7.2	1

We can easily check the centroid values by averaging the features in each cluster.

```

df.groupby ('Clus_km').mean ()

```

Customer ID	Years				Card Debt	Other Debt	Default ed	Debt Income Ratio
	Age	Edu	Employed	Income				
432.0	32.96	1.61	6.39	31.20	1.03	2.11	0.288	10.06
410.16	45.38	2.67	19.56	227.17	5.68	10.91	0.29	7.32
403.78	41.37	1.96	15.25	84.09	3.11	5.77	0.174	10.73

Now, let's look at the distribution of customers based on their age

and income. We display the scatterplot in **Figure 18-4**

```
fig=plt.figure()
area = np.pi * ( X[:, 1])**2
plt.scatter (X[:, 0], X[:, 3], s=area,
c=labels.astype(np.float), alpha=0.5)
plt.xlabel('Age', fontsize=18)
plt.ylabel('Income', fontsize=18)
plt.rc('xtick', labelsize = 16) # x tick label size
plt.rc('ytick', labelsize = 16) # y tick label size
plt.figure(figsize= (8, 8), dpi=180)
fig.tight_layout(pad=0.5, w_pad=0.5, h_pad=1.1)
fig.savefig('pic1.png', dpi=200)
plt.show()
```

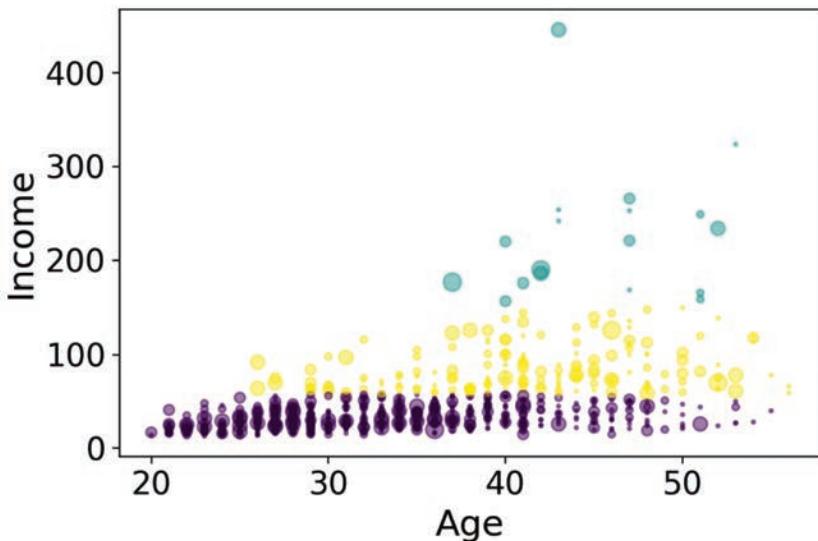


Figure 18-4. Distribution of customers based on age and income.

Next, we construct a three-dimensional scatterplot of customers based on education and income, as seen in **Figure 18-5**

```
from mpl_toolkits.mplot3d import Axes3D
# Set up the custom figure size
fig = plt.figure(1, figsize=(8, 6))

# Clears the current figure
plt.clf()
```

```

# The 3-dimentional axes
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)
# Define the figure aesthetics
plt.cla() # Clears the current figure
# plt.ylabel('Age', fontsize=18)
# plt.xlabel('Income', fontsize=16)
# plt.zlabel('Education', fontsize=16)
ax.set_xlabel('Education')
ax.set_ylabel('Age')
ax.set_zlabel('Income')
fig.tight_layout()
# The scatterplot
ax.scatter (X[:, 1], X[:, 0], X[:, 3],
           c= labels.astype (np.float))

```

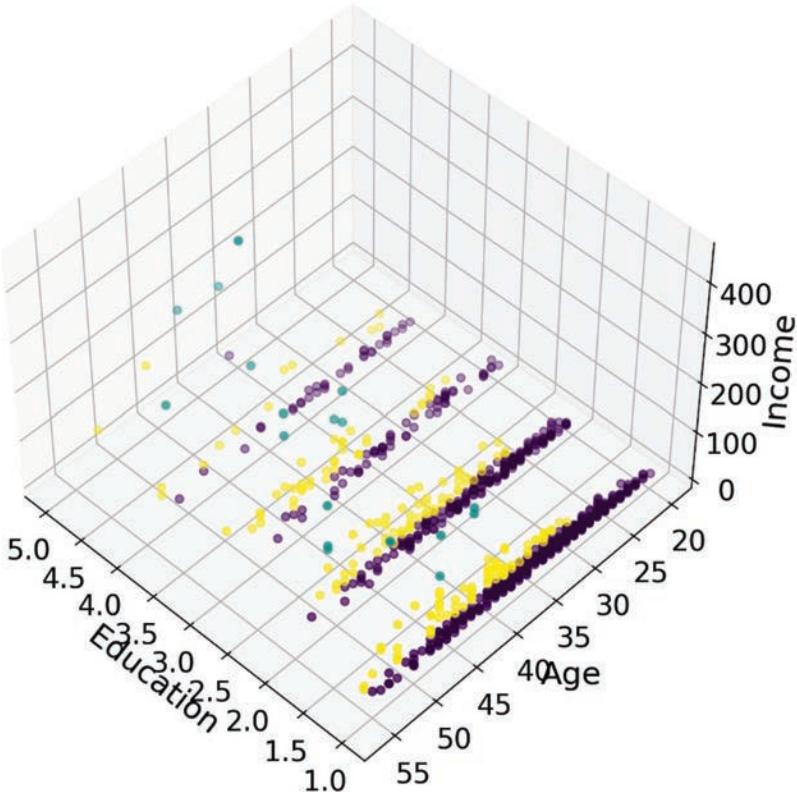


Figure 17-5. Three-dimensional scatterplot of customers based on education and income.

k-means will partition our customers into mutually exclusive groups, for example, into 3 clusters. The customers in each cluster are similar to each other demographically. Now we can create a profile for each group, considering the common characteristics of each cluster. For example, the 3 clusters can be:

- AFFLUENT, EDUCATED AND OLD AGED
- MIDDLE AGED AND MIDDLE INCOME
- YOUNG AND LOW INCOME

18. Hierarchical Clustering

Objectives

After completing this chapter, you will be able to:

- Use scikit-learn to do Hierarchical clustering
- Create dendrograms to visualize the clustering

Table of contents

- Hierarchical Clustering - Agglomerative
 1. Generating Random Data
 2. Agglomerative Clustering
 3. Dendrogram Associated for the Agglomerative Hierarchical Clustering
- Clustering on the Vehicle Dataset
 4. Data Cleaning
 5. Clustering Using Scipy
 6. Clustering using scikit-learn

Hierarchical Clustering - Agglomerative

We will be looking at a clustering technique, which is **Agglomerative Hierarchical Clustering**. Remember that agglomerative is the bottom-up approach.

In this chapter, we will be looking at Agglomerative clustering, which is more popular than **Divisive clustering**³.

We will also be using **Complete Linkage** as the Linkage Criteria.
NOTE: You we also try using Average Linkage wherever Complete Linkage would be used to see the difference!

```
import numpy as np
import pandas as pd
from scipy import ndimage
```

³ The divisive clustering algorithm is a top-down clustering approach, initially, all the points in the dataset belong to one cluster and split is performed recursively as one moves down the hierarchy.

```
import matplotlib.cm as cm

from scipy.cluster import hierarchy
from scipy.spatial import distance_matrix
from matplotlib import pyplot as plt
from sklearn import manifold, datasets
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets import make_blobs
%matplotlib inline
```

Generating Random Data

We will be generating a set of data using the **make_blobs** class. Input these parameters into make_blobs:

- **n_samples**: The total number of points equally divided among clusters.
 - Choose a number from 10-1500
- **centers**: The number of centers to generate, or the fixed center locations.
 - Choose arrays of x, y coordinates for generating the centers.
Have 1-10 centers (ex. centers=[[1,1], [2,5]])
- **cluster_std**: The standard deviation of the clusters. The larger the number, the further apart the clusters
 - Choose a number between 0.5-1.5

Save the result to **X1** and **y1**.

```
X1, y1 = make_blobs(n_samples=50, centers= [[4,4],
                                             [-2, -1], [1, 1], [10,4]], cluster_std=0.9)
```

Plot the scatter plot of the randomly generated data. The result is shown in **Figure 19-1**.

```
plt.figure(dpi=200)
plt.scatter (X1[:, 0], X1[:, 1], marker='o')
plt.rc('xtick', labelsize = 18) # x tick Label size
plt.rc('ytick', labelsize = 18) # y tick Label size
plt.tight_layout(pad=0.5, w_pad=0.5, h_pad=1.1)
plt.savefig('clust01.png', dpi=200)
```

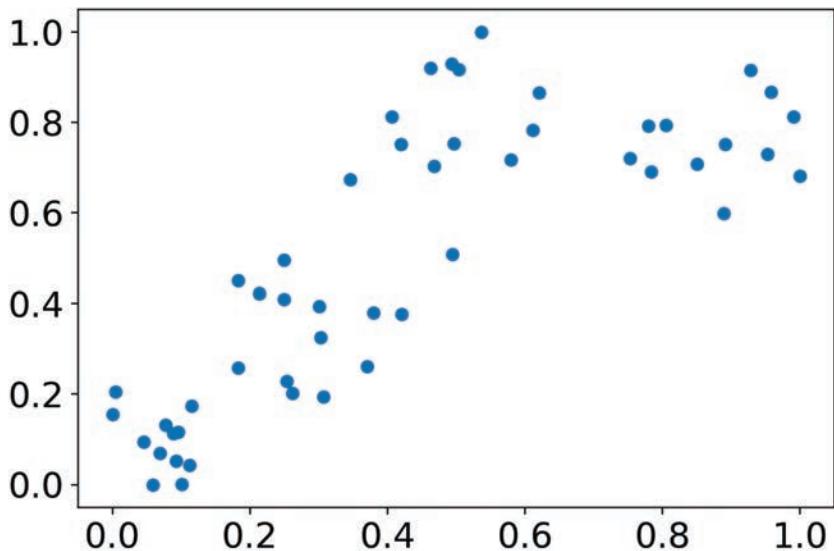


Figure 18-1. Scatter plot of the randomly generated data

Agglomerative Clustering

We will start by clustering the random data points we just created.

The **Agglomerative Clustering** class will require two inputs:

- **n_clusters**: The number of clusters to form as well as the number of centroids to generate.
 - Value will be: 4
- **linkage**: Which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of clusters that minimize this criterion.
 - Value will be: 'complete'
 - **Note**: It is recommended you try everything with 'average' as well

Save the result to a variable called **agglom**.

```
agglom = AgglomerativeClustering(n_clusters = 4, linkage =  
    'average')
```

Fit the model with **X2** and **y2** from the generated data above.

```
agglom.fit(X1,y1)
```

```
AgglomerativeClustering(linkage='average', n_clusters=4)
```

Run the following code to show the clustering. Remember to read the code and comments to gain more understanding on how the plotting works. The plot is displayed in **Figure 19-2**.

```
# Create a figure of size 6 inches by 4 inches.
plt.figure(dpi=200)

# These two lines of code are used to scale the data
# points down, or else the data points will be scattered
# very far apart.

# Create a minimum and maximum range of X1.
x_min, x_max = np.min(X1, axis=0), np.max(X1, axis=0)

# Get the average distance for X1.
X1 = (X1 - x_min) / (x_max - x_min)

# This loop displays all of the datapoints.
for i in range(X1.shape [0]):
    # Replace the data points with their respective
    # cluster value, (ex. 0), and color coded with a colormap
    # (plt.cm.spectral)
    plt.text(X1[i, 0], X1[i, 1], str(y1[i]),
             color=plt.cm.nipy_spectral(agglom.labels_[i]
/ 10.),
             fontdict={'weight': 'bold', 'size': 12})

# Remove the x ticks, y ticks, x and y axis
plt.xticks([])
plt.yticks([])
# plt.axis('off')

# Display the plot of the original data before clustering
plt.scatter (X1[:, 0], X1[:, 1], marker='.')
plt.show()

# Save the plot to the working directory as a PNG file
plt.savefig('clust02.png', dpi=200)
```

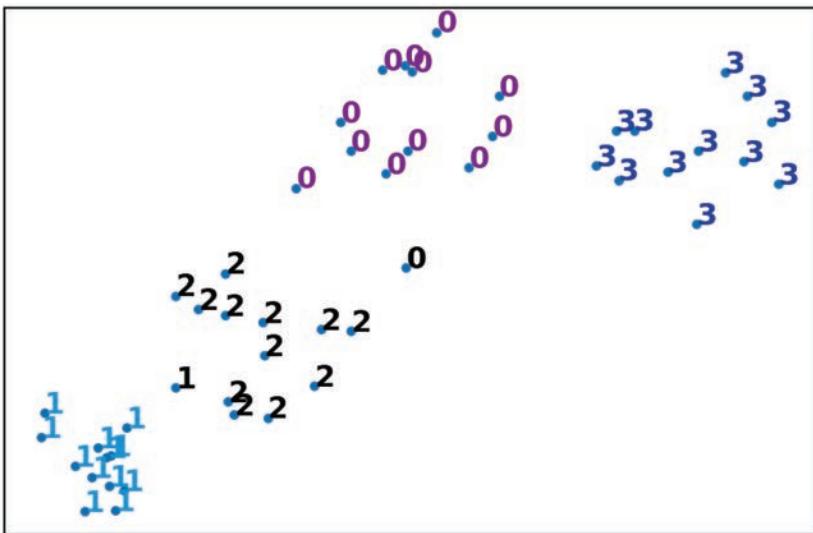


Figure 18-2. Plot of the *Agglomerative Clustering* class showing clusters of a set of data using the `make_blobs` class.

Dendrogram Associated for the Agglomerative Hierarchical Clustering

Remember that a distance matrix contains the distance from each point to every other point of a dataset .

Use the function `distance_matrix`, which requires **two inputs**. Use the Feature Matrix, `X1` as both inputs and save the distance matrix to a variable called `dist_matrix`.

Remember that the distance values are symmetric, with a diagonal of 0's. This is one way of making sure your matrix is correct. (Print out `dist_matrix` to make sure it's correct)

```
dist_matrix = distance_matrix(X1,X1)
print(dist_matrix)

[[0.          0.8493   0.0770   ...   0.1684   0.1712   0.1084]
 [0.8493    0.          0.8160   ...   1.0095   0.9881   0.8669]
 [0.0770    0.8160    0.          ...   0.2313   0.1738   0.0616]
 :           :           :           ...   :
 [0.1684    1.0095    0.2313   ...   0.          0.1667   0.2212 ]
 [0.1712    0.9881    0.1738   ...   0.1667   0.          0.1221]
 [0.1084    0.8669    0.0616   ...   0.2212   0.1221   0.        ]]
```

Using the **linkage** class from hierarchy, pass in the parameters:

- The distance matrix
- 'complete' for complete linkage

Save the result to a variable called **Z**.

```
Z = hierarchy.linkage (dist_matrix, 'complete')
```

A Hierarchical clustering is typically visualized as a dendrogram as shown in the following cell. Each merge is represented by a horizontal line. The y-coordinate of the horizontal line is the similarity of the two clusters that were merged, where cities are viewed as singleton clusters. By moving up from the bottom layer to the top node, a dendrogram allows us to reconstruct the history of merges that resulted in the depicted clustering.

Next, we will save the dendrogram to a variable called **dendro**. In doing this, the dendrogram will also be displayed. Using the **dendrogram** class from hierarchy, pass in the parameter: Z. The plot and display the result in **Figure 19-3**.

```
plt.figure(dpi=200)
dendro = hierarchy.dendrogram (Z)
plt.rc('font', size = 18)
plt.tight_layout()
plt.savefig('clust03.png', dpi=200)
```

In the **complete linkage**, also called **farthest neighbor**, the clustering method is the opposite of single linkage. Distance between groups is now defined as the distance between the most distant pair of objects, one from each group.

In the complete linkage method, $D(r, s)$ is computed as

$$D(r, s) = \text{Max} \{ d(i, j), \\ \text{where object } i \text{ is in cluster } r \text{ and object } j \text{ is in cluster } s \}$$

The distance between every possible object pair (i, j) is computed, where object i is in cluster r and object j is in cluster s and the maximum value of these distances is said to be the distance between clusters r and s . The distance between two clusters is given by the value of the longest link between the clusters.

At each stage of hierarchical clustering, the clusters r and s , for which $D(r,s)$ is minimum, are merged.

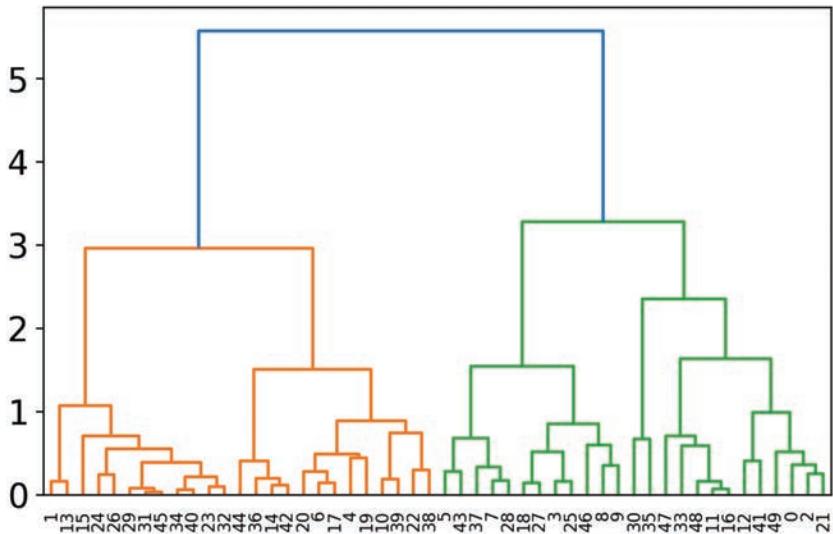


Figure 18-3. Dendrogram showing hierarchical clustering of a set of data using the `make_blobs` class.

Example

We used **complete linkage** for our case above, change it to **average linkage** to see how the dendrogram changes. We display the dendrogram in **Figure 19-4**.

```
plt.figure(dpi=200)
Z = hierarchy.linkage(dist_matrix, 'average')
dendro = hierarchy.dendrogram(Z)
plt.rc('font', size = 18)
plt.tight_layout()
plt.savefig('clust04.png', dpi=200)
```

In Average linkage clustering, the distance between two clusters is defined as the average of distances between all pairs of objects, where each pair is made up of one object from each group. In the average linkage method, $D(r,s)$ is computed as

$$D(r,s) = \frac{Trs}{(Nr * Ns)},$$

where Trs is the sum of all pairwise distances between cluster r and cluster s . Nr and Ns are the sizes of the clusters r and s , respectively.

At each stage of hierarchical clustering, the clusters r and s , for which $D(r,s)$ is the minimum, are merged.

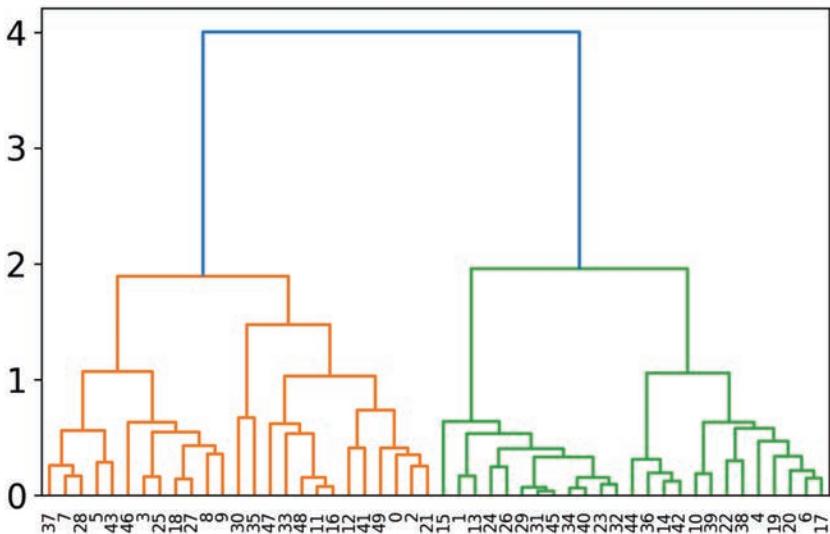


Figure 18-4. Dendrogram using complete linkage, which shows the hierarchical clustering of a set of data using the `make_blobs` class

Clustering on Vehicle dataset

Imagine that an automobile manufacturer has developed prototypes for a new vehicle. Before introducing the new model into its range, the manufacturer wants to determine which existing vehicles on the market are most like the prototypes--that is, how vehicles can be grouped, which group is the most similar with the model, and therefore which models they will be competing against.

Our objective here, is to use clustering methods, to find the most distinctive clusters of vehicles. It will summarize the existing vehicles and help manufacturers to make decision about the supply of new models.

Download data

To download the data, we will use **GitHub Jupyter Data** repository. The URL is provided in the path of the next block of code.

When it comes to Machine Learning, you will likely be working with large datasets. Since there is a file-size limitation on GitHub, we try to use small-sized data sets. The file we use here, cars_clus.csv, which is customize for our purposes.

```
path =  
"https://raw.githubusercontent.com/stricje1/jupyter/main/d  
ata/cars_clus.csv"
```

Read data

Let's read dataset to see what features the manufacturer has collected about the existing models.

```
# Read csv  
pdf = pd.read_csv(path)  
print ("Shape of dataset: ", pdf.shape)  
  
pdf.head(5)
```

Shape of dataset: (159, 16)

	manufact	model	sales	resale	price	...	mpg	Insales
0	Acura	Integra	16.919	16.36	21.5	...	28	2.828
1	Acura	TL	39.384	19.875	28.4	...	25	3.673
2	Acura	CL	14.114	18.225		...	26	2.647
3	Acura	RL	8.588	29.725	42	...	22	2.15
4	Audi	A4	20.397	22.255	23.99	...	27	3.015

The feature sets include price in thousands (`price`), engine size (`engine_s`), horsepower (`horsepow`), wheelbase (`wheelbas`), width (`width`), length (`length`), curb weight (`curb_wgt`), fuel capacity (`fuel_cap`) and fuel efficiency (`mpg`).

Data Cleaning

Let's clean the dataset by dropping the rows that have null value:

```
print ("Shape of dataset before cleaning: ", pdf.size)  
pdf[[ 'sales', 'resale', 'type', 'price', 'engine_s',  
      'horsepow', 'wheelbas', 'width', 'length',
```

```

'curb_wgt', 'fuel_cap',
'mpg', 'lnsales']] = pdf[['sales', 'resale',
'type', 'price', 'engine_s',
'horsepow', 'wheelbas', 'width', 'length',
'curb_wgt', 'fuel_cap',
'mpg', 'lnsales']].apply(pd.to_numeric,
errors='coerce')
pdf = pdf.dropna()
pdf = pdf.reset_index (drop=True)
print ("Shape of dataset after cleaning: ", pdf.size)
pdf.head(5)

```

Shape of dataset before cleaning: 2544

Shape of dataset after cleaning: 1872

	manufact	model	sales	resale	price	...	mpg	lnsales	partition
0	Acura	Integra	16.919	16.360	21.50	...	28.0	2.828	0.0
1	Acura	TL	39.384	19.875	28.40	...	25.0	3.673	0.0
2	Acura	RL	8.588	29.725	42.00	...	22.0	2.150	0.0
3	Audi	A4	20.397	22.255	23.99	...	27.0	3.015	0.0
4	Audi	A6	18.780	23.555	33.95	...	22.0	2.933	0.0

Feature selection

Let's select our feature set:

```

featureset = pdf[['engine_s', 'horsepow', 'wheelbas',
'width', 'length', 'curb_wgt', 'fuel_cap', 'mpg']]

```

Normalization

Now we can normalize the feature set. **MinMaxScaler** transforms features by scaling each feature to a given range. It is by default (0, 1). That is, this estimator scales and translates each feature individually such that it is between zero and one.

```

from sklearn.preprocessing import MinMaxScaler
x = featureset.values #returns a numpy array
min_max_scaler = MinMaxScaler()
feature_mtx = min_max_scaler.fit_transform (x)
feature_mtx [0:5]
array([[0.1143, 0.2152, 0.1866, 0.2814, 0.3063, 0.2310,
       0.1336, 0.4333],
       [0.3143, 0.4304, 0.3362, 0.4611, 0.5793, 0.5037,

```

```

0.3180, 0.3333],
[0.3571, 0.3924, 0.4772, 0.5269, 0.6285, 0.6071,
0.3548, 0.2333],
[0.1143, 0.2405, 0.2169, 0.3353, 0.3808, 0.3425,
0.2811, 0.4    ],
[0.2571, 0.3671, 0.3492, 0.8084, 0.5672, 0.5174,
0.3779, 0.2333]])

```

Clustering using SciPy

In this part we use **SciPy** package to cluster the dataset.

First, we calculate the distance matrix.

```

import scipy
leng = feature_mtx.shape[0]
D = scipy.zeros([leng,leng])
for i in range(leng):
    for j in range(leng):
        D[i,j] = scipy.spatial.distance.euclidean
(feature_mtx[i], feature_mtx[j])
D
array([[0.      , 0.5778, 0.7546, ... , 0.2853, 0.2492,
0.1888],
[0.5778 , 0.      , 0.2280, ... , 0.3609, 0.6635,
0.6220],
[0.7546, 0.2280, 0.      , ... , 0.5173, 0.8179,
0.7793],
:       :       :       :       :       :
[0.2492, 0.6635, 0.8179, ... , 0.4180, 0.      ,
0.1521],
[0.1888, 0.6220, 0.7793, ... , 0.3572, 0.1521,
0.      ]])

```

In agglomerative clustering, at each iteration, the algorithm must update the distance matrix to reflect the distance of the newly formed cluster with the remaining clusters in the forest. The following methods are supported in **SciPy** for calculating the distance between the newly formed cluster and each: – single – complete – average – weighted – centroid.

We use **complete** for our case, but feel free to change it to see how the results change.

```

import pylab
import scipy.cluster.hierarchy
Z = hierarchy.linkage(D, 'complete')

```

Essentially, hierarchical clustering does not require a pre-specified number of clusters. However, in some applications we want a partition of disjoint clusters just as in flat clustering. So, we can use a cutting line:

```

from scipy.cluster.hierarchy import fcluster
max_d = 3
clusters = fcluster(Z, max_d, criterion='distance')
clusters
array([ 1, 5, 5, 6, 5, 4, 6, 5, 5, 5, 5, 5, 5, 4, 4, 5, 1, 6,
       5, 5, 5, 4, 2, 11, 6, 6, 5, 6, 5, 1, 6, 6, 10, 9, 8,
       9, 3, 5, 1, 7, 6, 5, 3, 5, 3, 8, 7, 9, 2, 6, 6, 5,
       4, 2, 1, 6, 5, 2, 7, 5, 5, 5, 4, 4, 3, 2, 6, 6, 5,
       7, 4, 7, 6, 6, 5, 3, 5, 5, 6, 5, 4, 4, 1, 6, 5, 5,
       5, 6, 4, 5, 4, 1, 6, 5, 6, 6, 5, 5, 5, 7, 7, 7, 2,
       2, 1, 2, 6, 5, 1, 1, 1, 7, 8, 1, 1, 6, 1, 1],
      dtype=int32)

```

Also, you can determine the number of clusters directly:

```

from scipy.cluster.hierarchy import fcluster
k = 5
clusters = fcluster(Z, k, criterion='maxclust')
clusters
array([1,3,3,3,3,2,3,3,3,3,3,3,2,2,3,1,3,3,3,3,3,2,1,
       5,3,3,3,3,3,1,3,3,4,4,4,4,2,3,1,3,3,3,3,2,3,2,
       4,3,4,1,3,3,3,2,1,1,3,3,1,3,3,3,3,2,2,2,1,3,
       3,3,3,2,3,3,3,3,2,3,3,3,3,2,2,1,3,3,3,3,3,2,
       3,2,1,3,3,3,3,3,3,3,3,3,1,1,1,1,3,3,1,1,1,
       3,4,1,1,3,1,1], dtype=int32)

```

Now, plot the dendrogram shown in **Figure 19-5**.

```

fig = pylab.figure(figsize=(18,50), dpi=300)
def llf(id):
    return '[%s %s %s]' % (pdf['manufact'][id],
                           pdf['model'][id], int(float(pdf['type'][id])))
fig.tight_layout()

```

```

dendro = hierarchy.dendrogram(Z, leaf_label_func=llf,
leaf_rotation=0, leaf_font_size =18, orientation =
'right')
plt.savefig('clust05.png', dpi=300)

```

Clustering using scikit-learn

Let's redo the dendrogram again, but this time using the **scikit-learn** package, `sklearn`, with the result shown in **Figure 19-6**. From `sklearn`, we use the `euclidean_distances()` function

```

from sklearn.metrics.pairwise import euclidean_distances
dist_matrix = euclidean_distances(feature_mtx,
        feature_mtx)
print(dist_matrix)

[[0.          0.5778   0.7546 ... 0.2853   0.2492   0.1888]
 [0.5778     0.          0.2280 ... 0.3609   0.6635   0.6220]
 [0.7546     0.2280    0.          ... 0.5173   0.8179   0.7793]
 [          :       :       :       :       :       :]
 [0.2853     0.3609    0.5173 ... 0.          0.4180   0.3572]
 [0.2492     0.6635    0.8179 ... 0.4180   0.          0.1521]
 [0.1888     0.6220    0.7793 ... 0.3572   0.1521   0.        ]]

```

Considering the rows of X (and Y=X) as vectors, the function, `euclidean_distances()`, computes the distance matrix between each pair of vectors. For efficiency reasons, the Euclidean distance between a pair of row vector x and y is computed as:

$$dist(x, y) = \sqrt{dot(x, x) - 2 \times dot(x, y) + dot(y, y)}$$

This formulation has two advantages over other ways of computing distances. First, it is computationally efficient when dealing with sparse data. Second, if x varies but y remains unchanged, then the right-most **dot product** `dot(y, y)` can be pre-computed.

However, this is not the most precise way of doing this computation, and the distance matrix returned by this function may not be exactly symmetric as required by, e.g., `scipy.spatial.distance`.

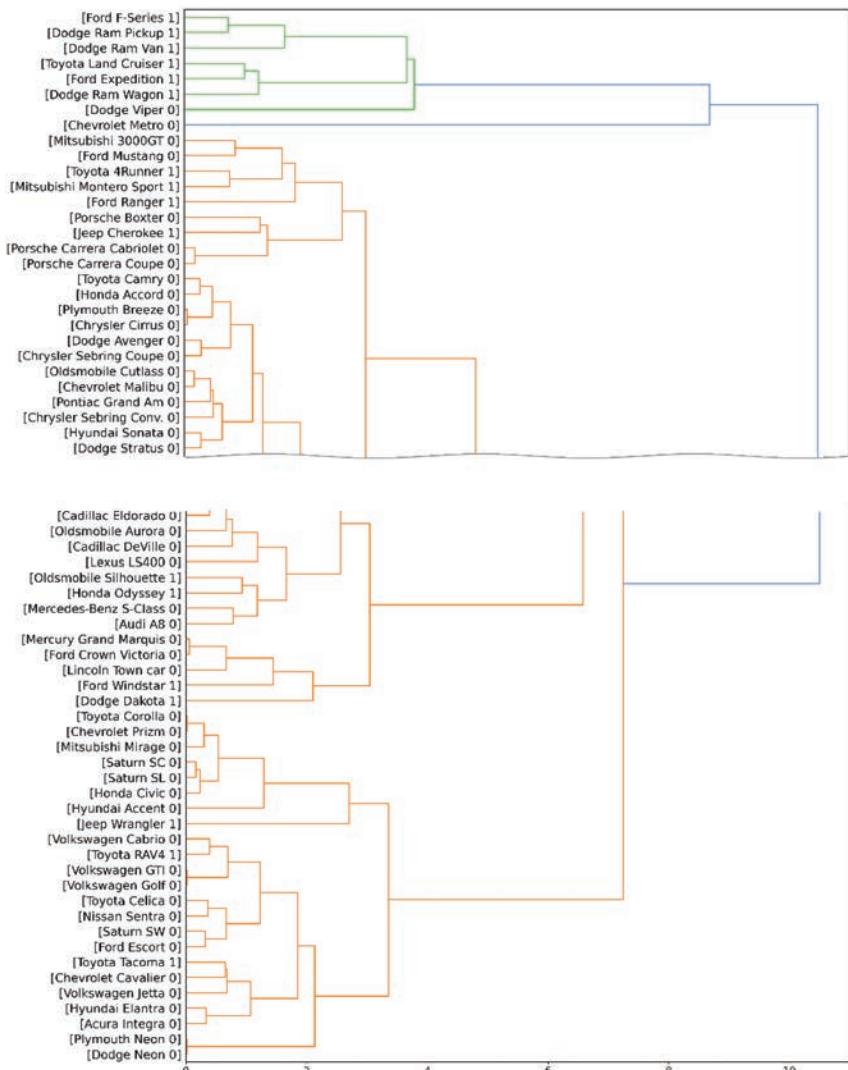


Figure 18-5. Dendrogram of the most distinctive clusters of vehicles using SciPy.

```
Z_using_dist_matrix = hierarchy.linkage(dist_matrix,
                                         'complete')
fig = pylab.figure(figsize=(18,50))
def l1f(id):
    return '[%s %s %s]' % (pdf['manufact'][id],
                           pdf['model'][id], int(float(pdf['type'][id])))
```

```

fig.tight_layout(pad=0.5, w_pad=0.5, h_pad=0.5)
dendro = hierarchy.dendrogram(Z_using_dist_matrix,
leaf_label_func=llf, leaf_rotation=0, leaf_font_size =18,
orientation = 'right')
plt.savefig('clust06.png', dpi=300)

```

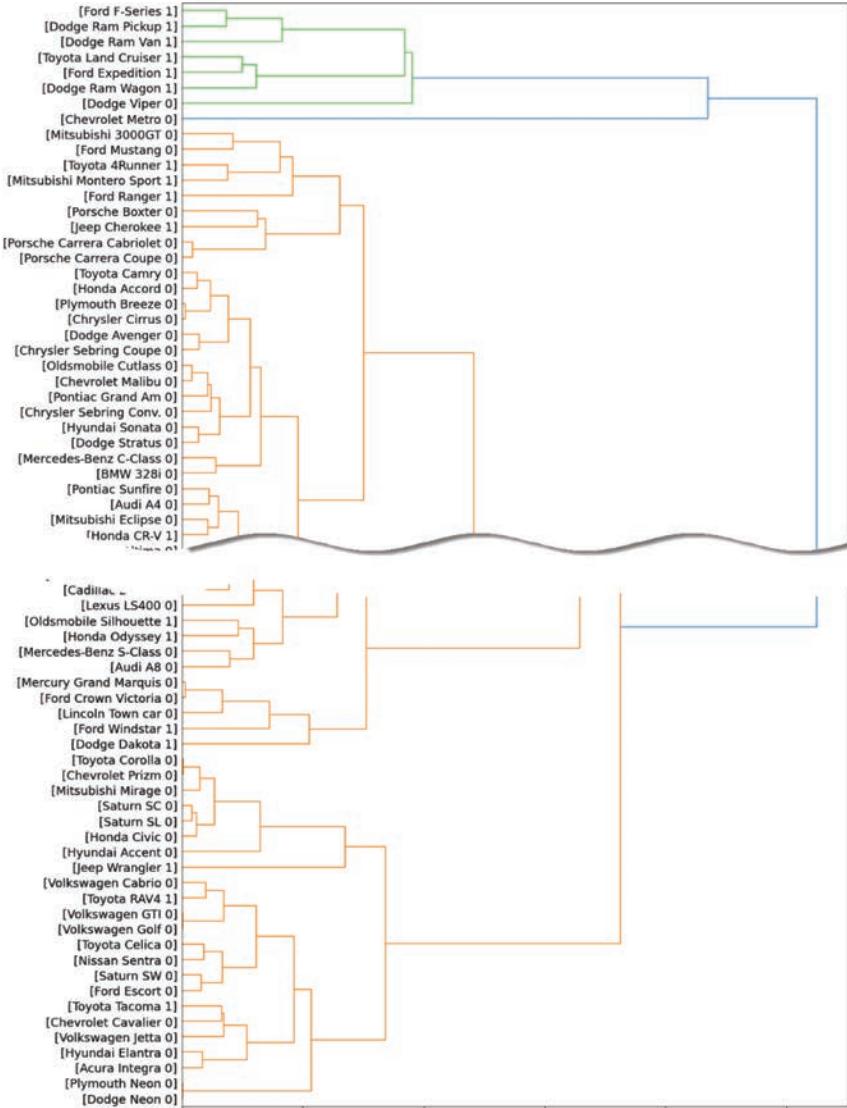


Figure 18-6. Dendrogram of the most distinctive clusters of vehicles using sklearn (scikit-learn)

We can use the `AgglomerativeClustering` function from `scikit-learn` library to cluster the dataset. The `AgglomerativeClustering` performs a hierarchical clustering using a bottom-up approach. The linkage criteria determine the metric used for the merge strategy:

- Ward minimizes the **sum of squared differences** within all clusters. It is a variance-minimizing approach and in this sense is similar to the **k-means** objective function but tackled with an agglomerative hierarchical approach (Maklin, 2018).
- Maximum or complete linkage minimizes the maximum distance between observations of pairs of clusters.
- **Average linkage** minimizes the average of the distances between all observations of pairs of clusters.

```
agglom = AgglomerativeClustering (n_clusters = 6,  
                                 linkage = 'complete')
```

```
agglom.fit (dist_matrix)  
agglom.labels_
```

```
array([1, 2, 2, 3, 2, 4, 3, 2, 2, 2, 2, 2, 2, 4, 4, 2, 1, 3,  
      2, 2, 2, 4, 1, 5, 3, 3, 2, 3, 2, 1, 3, 3, 0, 0, 0, 0, 4,  
      2, 1, 3, 3, 2, 4, 2, 4, 0, 3, 0, 1, 3, 3, 2, 4, 1, 1, 3,  
      2, 1, 3, 2, 2, 2, 4, 4, 4, 1, 3, 3, 2, 3, 4, 3, 3, 3, 2,  
      4, 2, 2, 3, 2, 4, 4, 1, 3, 2, 2, 2, 3, 4, 2, 4, 1, 3, 2,  
      3, 3, 2, 2, 2, 3, 3, 1, 1, 1, 1, 3, 2, 1, 1, 1, 3, 0,  
      1, 1, 3, 1, 1], dtype=int64)
```

We can add a new field to our dataframe to show the cluster of each row. Then we plot the clusters in **Figure 19-7**.

```
pdf[ 'cluster_ ' ] = agglom.labels_  
pdf.head()
```

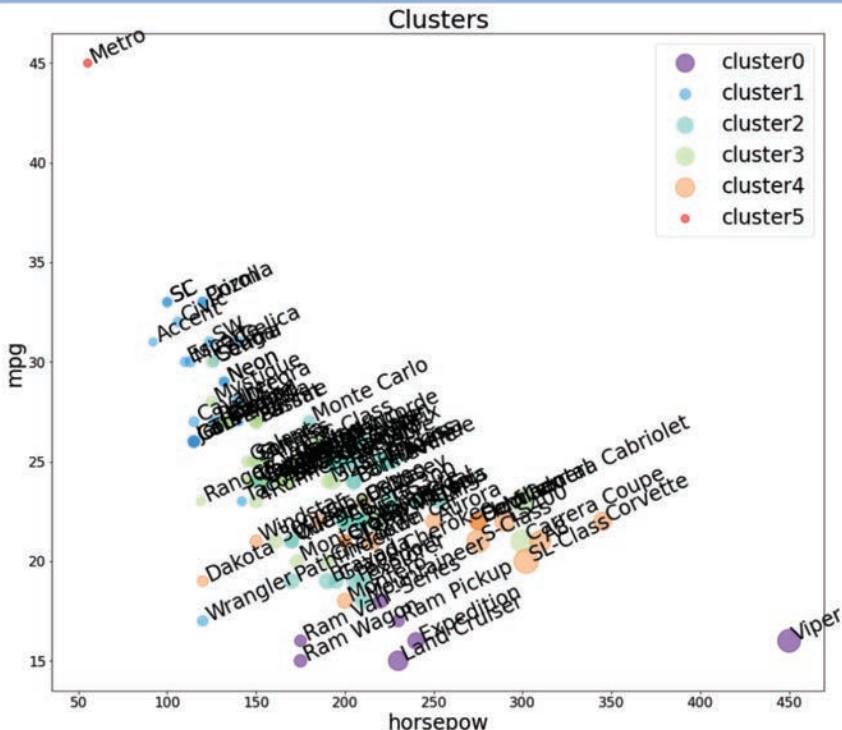
	manufact	model	sales	resale	price	...	mpg	cluster_
0	Acura	Integra	16.919	16.360	21.50	...	28.0	1
1	Acura	TL	39.384	19.875	28.40	...	25.0	2
2	Acura	RL	8.588	29.725	42.00	...	22.0	2
3	Audi	A4	20.397	22.255	23.99	...	27.0	3
4	Audi	A6	18.780	23.555	33.95	...	22.0	2

```
n_clusters = max(agglom.labels_)+1  
colors = cm.rainbow(np.linspace (0, 1, n_clusters))  
cluster_labels = list(range(0, n_clusters))
```

```

# Create a figure of size 6 inches by 4 inches.
plt.figure(figsize= (16,14))
plt.rc('font', size = 24)
for color, label in zip(colors, cluster_labels):
    subset = pdf[pdf.cluster_ == label]
    for i in subset.index:
        plt.text(subset.horsepow[i],
subset.mpg[i],str(subset['model'][i]), rotation=25)
    plt.scatter (subset.horsepow, subset.mpg, s =
subset.price*10, c=color, label='cluster' +
str(label),alpha=0.5)
#    plt.scatter(subset.horsepow, subset.mpg)
plt.legend()
plt.title( 'Clusters')
plt.xlabel('horsepow')
plt.ylabel('mpg')
plt.savefig('clust07.png', dpi=200)

```



As you can see, we are seeing the distribution of each cluster using the scatter plot, but it is not very clear where is the centroid of each cluster. Moreover, there are 2 types of vehicles in our dataset, `truck` (value of 1 in the type column) and `car` (value of 0 in the type column). So, we use them to distinguish the classes, and summarize the cluster. First, we count the number of cases in each group:

```
pdf.groupby(['cluster_', 'type'])['cluster_'].count()
cluster_  type
0         0.0    1
          1.0    6
1         0.0   20
          1.0    3
2         0.0   26
          1.0   10
3         0.0   28
          1.0    5
4         0.0   12
          1.0    5
5         0.0    1
Name: cluster_, dtype: int64
```

Now we can look at the characteristics of each cluster:

```
agg_cars = pdf.groupby(['cluster_', 'type'])['horsepow',
                                             'engine_s', 'mpg', 'price'].mean()
agg_cars
```

		horsepow	engine_s	mpg	price
cluster_	type				
0	0.0	450.000000	8.000000	16.000000	69.725000
	1.0	211.666667	4.483333	16.166667	29.024667
1	0.0	118.500000	1.890000	29.550000	14.226100
	1.0	129.666667	2.300000	22.333333	14.292000
2	0.0	203.615385	3.284615	24.223077	27.988692
	1.0	182.000000	3.420000	20.300000	26.120600
3	0.0	168.107143	2.557143	25.107143	24.693786
	1.0	155.600000	2.840000	22.000000	19.807000
4	0.0	267.666667	4.566667	21.416667	46.417417
	1.0	173.000000	3.180000	20.600000	24.308400
5	0.0	55.000000	1.000000	45.000000	9.235000

It is obvious that we have 3 main clusters with the majority of vehicles in those.

Cars:

- Cluster 1: with almost high `mpg`, and low in `horsepower`.
- Cluster 2: with good `mpg` and `horsepower`, but higher price than average.
- Cluster 3: with low `mpg`, high `horsepower`, highest `price`.

Trucks:

- Cluster 1: with almost highest `mpg` among trucks, and lowest in `horsepower` and `price`.
- Cluster 2: with almost low `mpg` and medium `horsepower`, but higher `price` than average.
- Cluster 3: with good `mpg` and `horsepower`, low `price`.

Please notice that we did not use `type` and `price` of cars in the clustering process, but Hierarchical clustering could forge the clusters and discriminate them with quite a high accuracy. We display the result in **Figure 19-8**.

```
plt.figure(figsize= (10,12))
for color, label in zip(colors, cluster_labels):
    subset = agg_cars.loc [(label,),]
    for i in subset.index:
        plt.text(subset.loc[i][0] + 5, subset.loc[i][2],
                 'type=' + str(int(i)) + ', price=' +
                 str(int(subset.loc[i][3])) + 'k')
plt.scatter (subset.horsepow, subset.mpg,
            s=subset.price*20, c=color,
            label='cluster' +str(label))
plt.tight_layout(pad=1.25, w_pad=1.25, h_pad=1.25)
plt.legend()
plt.title('Clusters')
plt.xlabel('horsepow')
plt.ylabel('mpg')
plt.savefig('clust08.png', dpi=150, figsize=(10,12))
```

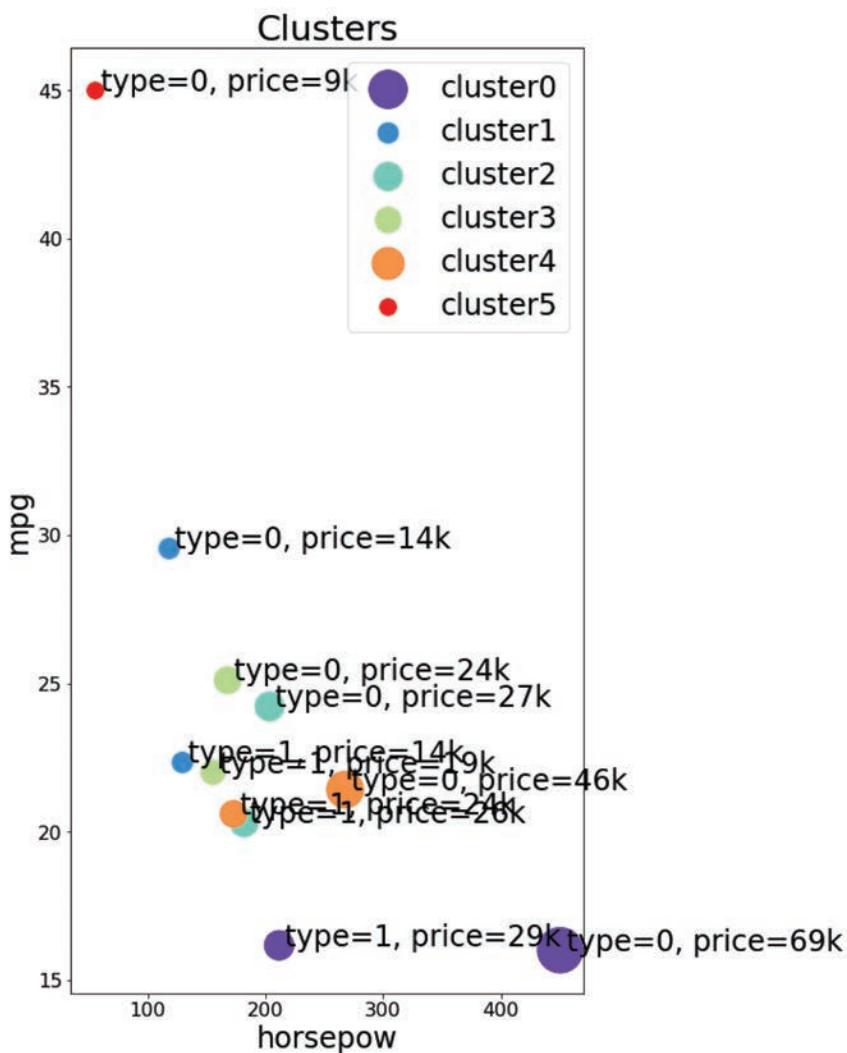


Figure 18-8. Hierarchical clustering that forges the clusters and discriminate them with quite a high accuracy.

19. Density-Based Clustering

Objectives

After completing this chapter, you will be able to:

- Use **DBSCAN** to do Density based clustering
- Use **Matplotlib** to plot clusters

Most of the traditional clustering techniques, such as k-means, hierarchical and fuzzy clustering, can be used to group data without supervision.

However, when applied to tasks with arbitrary shape clusters, or clusters within a cluster, the traditional techniques might be unable to achieve good results. That is, elements in the same cluster might not share enough similarity or the performance may be poor. Additionally, Density-based clustering locates regions of high density that are separated from one another by regions of low density. Density, in this context, is defined as the number of points within a specified radius.

- In this section, the main focus will be manipulating the data and properties of **DBSCAN** and observing the resulting clustering.

Import the following libraries:

- **numpy as np**
- **DBSCAN** from **sklearn.cluster**
- **make_blobs** from **sklearn.datasets**
- **StandardScaler** from **sklearn.preprocessing**
- **matplotlib.pyplot as plt**

```
!pip install 'basemap'
```

```
import numpy as np
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
%matplotlib inline
```

Data generation

The function below will generate the data points and requires these inputs:

- **centroidLocation**: Coordinates of the centroids that will generate the random data.
 - Example: input: [[4,3], [2,-1], [-1,4]]
- **numSamples**: The number of data points we want generated, split over the number of centroids (# of centroids defined in **centroidLocation**)
 - Example: 1500
- **clusterDeviation**: The standard deviation of the clusters. The larger the number, the further the spacing of the data points within the clusters.
 - Example: 0.5

```
def createDataPoints(centroidLocation, numSamples,
clusterDeviation):
    # Create random data and store in feature matrix X and
    # response vector y.
    X, y = make_blobs(n_samples=numSamples,
    centers=centroidLocation, cluster_std=clusterDeviation)
    # Standardize features by removing the mean and
    # scaling to unit variance
    X = StandardScaler().fit_transform(X)
    return X, y
```

Use `createDataPoints` with the 3 inputs and store the output into variables `X` and `y`.

```
X, y = createDataPoints([[4,3], [2,-1], [-1,4]] , 1500,
0.5)
```

Modeling

DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise. This technique is one of the most common clustering algorithms which works based on density of object. The whole idea is that if a particular point belongs to a cluster, it should be near to lots of other points in that cluster (Pedregosa, et al., 2011).

It works based on two parameters: Epsilon and Minimum Points.

Epsilon determine a specified radius that if includes enough number of points within, we call it dense area minimumSamples determine the minimum number of data points we want in a neighborhood to define a cluster.

```
epsilon = 0.3
minimumSamples = 7
db = DBSCAN (eps = epsilon, min_samples =
              minimumSamples).fit(X)
labels = db.labels_
labels
array([0, 0, 1, ..., 1, 1, 0], dtype=int64)
```

Distinguish outliers

Let's replace all elements with `True` in `core_samples_mask` that are in the cluster, `False` if the points are outliers. First, we'll create an array of Booleans using the `labels` from `db`.

```
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
core_samples_mask
array([ True,  True,  True, ...,  True,  True,  True])
```

Next, we'll get the number of clusters in labels, ignoring noise if present.

```
n_clusters_ = len(set(labels)) - (1 if -1 in labels
                                else 0)
n_clusters_
```

3

Now, we'll remove repetition in labels by turning it into a set.

```
unique_labels = set(labels)
unique_labels
```

{0, 1, 2}

Data visualization

We start here by creating color-coded clusters from simulated data. The scatterplot of the data is displayed in **Figure 19-1**.

```
# Create colors for the clusters.
colors = plt.cm.Spectral(np.linspace(0, 1,
len(unique_labels)))

# Plot the points with colors
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = 'k'
    class_member_mask = (labels == k)

    # Plot the datapoints that are clustered
    xy = X[class_member_mask & core_samples_mask]
    plt.scatter(xy[:, 0], xy[:, 1], s=50, c=[col],
                marker = u'o', alpha=0.5)
    # Plot the outliers
    xy = X[class_member_mask & ~core_samples_mask]
    plt.scatter(xy[:, 0], xy[:, 1], s = 50, c = [col],
    marker = u'o', alpha = 0.5)
```

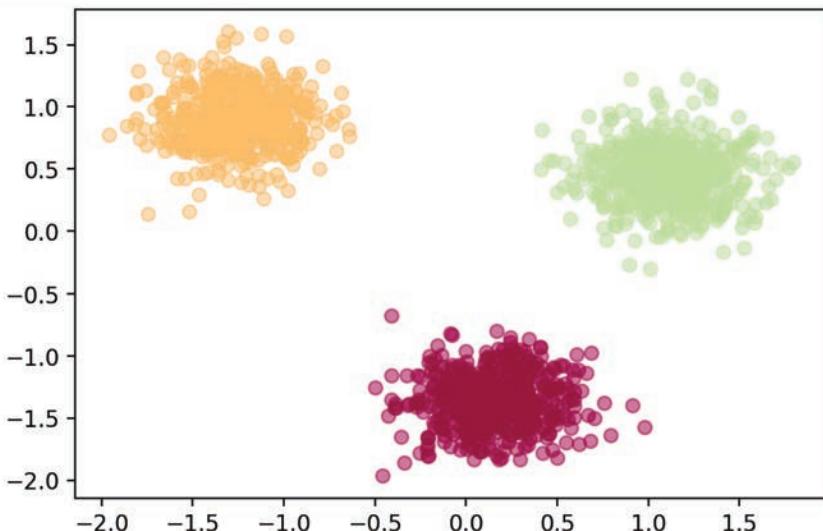


Figure 19-1. Color-coded clusters of simulated data.

Example

To better understand differences between partitional and density-based clustering, try to cluster the above dataset into 3 clusters using k-Means.

Let's not generate data again, but use the same dataset as above.

```
from sklearn.cluster import KMeans
k = 3
k_means3 = KMeans (init = "k-means++", n_clusters = k,
n_init = 12)
k_means3.fit (X)
fig = plt.figure(figsize=(6, 4))
ax = fig.add_subplot(1, 1, 1)
for k, col in zip(range(k), colors):
    my_members = (k_means3.labels_ == k)
    plt.scatter (X[my_members, 0], X[my_members, 1],
c=col, marker=u'o', alpha=0.5)
plt.show()
```

Figure 19-2 shows three clusters we found using k-means calculate and plot them.

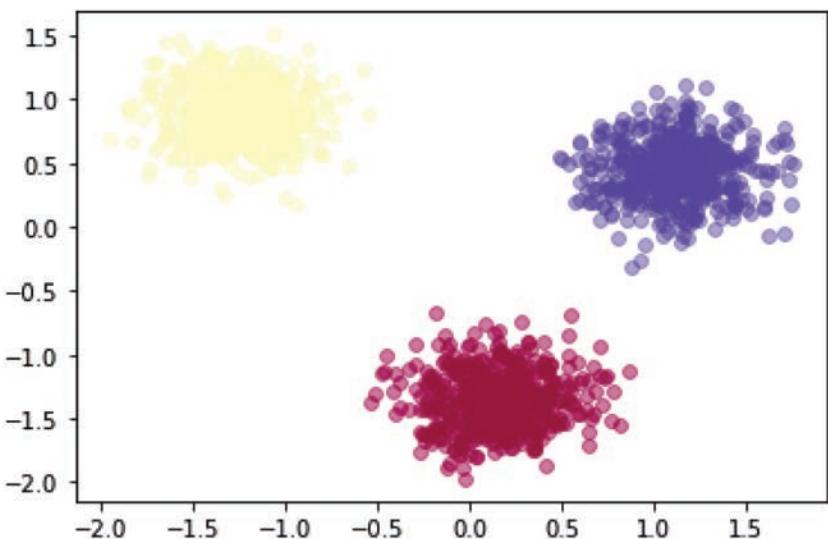


Figure 19-2. Here are the same three clusters, but we use k-means calculate and plot them.

Weather Station Clustering using DBSCAN & scikit-learn

DBSCAN is especially very good for tasks like class identification in a spatial context. The wonderful attribute of DBSCAN algorithm is that it can find out any arbitrary shape cluster without getting affected by noise. For example, this following example cluster the location of weather stations in Canada. DBSCAN can be used here, for instance, to find the group of stations which show the same weather condition. As you can see, it not only finds different arbitrary shaped clusters, cut can find the denser part of data-centered samples by ignoring less-dense areas or noises (Kotian, 2021).

Loading the data

Let's start exploring with the data. We will be operating according to the following workflow:

1. Loading data
 - Overview data
 - Data cleaning
 - Data selection
 - Clustering

About the dataset

Table 19-1. Environment Canada Monthly Values for July - 2015

Name in the table	Meaning
Stn_Name	Station Name
Lat	Latitude (North+, degrees)
Long	Longitude (West -, degrees)
Prov	Province
Tm	Mean Temperature (°C)
DwTm	Days without Valid Mean Temperature
D	Mean Temperature difference from Normal (1981-2010) (°C)

Name in the table	Meaning
Tx	Highest Monthly Maximum Temperature (°C)
DwTx	Days without Valid Maximum Temperature
Tn	Lowest Monthly Minimum Temperature (°C)
DwTn	Days without Valid Minimum Temperature
S	Snowfall (cm)
DwS	Days without Valid Snowfall
S%N	Percent of Normal (1981-2010) Snowfall
P	Total Precipitation (mm)
DwP	Days without Valid Precipitation
P%N	Percent of Normal (1981-2010) Precipitation
S_G	Snow on the ground at the end of the month (cm)
Pd	Number of days with Precipitation 1.0 mm or more
BS	Bright Sunshine (hours)
DwBS	Days without Valid Bright Sunshine
BS%	Percent of Normal (1981-2010) Bright Sunshine
HDD	Degree Days below 18 °C
CDD	Degree Days above 18 °C

Stn_No Climate station identifier (first 3 digits indicate drainage basin; last 4 characters are for sorting alphabetically).

1-Download data

To download the data, we will use my GitHub Jupyter Data repository.

```
path =
"https://raw.githubusercontent.com/stricje1/jupyter/main/da
ta/weather-stations20140101-20141231.csv"
```

2- Load the dataset

We will import the .csv then we create the columns for year, month and day.

```

import csv
import pandas as pd
import numpy as np

filename = 'weather-stations20140101-20141231.csv'

# Read csv
pdf = pd.read_csv(filename)
pdf.head(5)

```

	Stn_Name	Lat	Long	Prov	Tm	...	BS%	HDD	CDD
0	CHEMAINUS	48.935	-123.742	BC	8.2	...	NaN	273.3	0
1	COWICHAN LAKE FORESTRY	48.824	-124.133	BC	7	...	NaN	307	0
2	LAKE COWICHAN	48.829	-124.052	BC	6.8	...	NaN	168.1	0
3	DISCOVERY ISLAND	48.425	-123.226	BC	Nan	...	NaN	Nan	Nan
4	DUNCAN KELVIN CREEK	48.735	-123.728	BC	7.7	...	NaN	267.7	0

5 rows × 25 columns (8 showing plus index)

3-Cleaning

Let's remove rows that don't have any value in the Tm field.

```

pdf = pdf[pd.notnull(pdf["Tm"])]
pdf = pdf.reset_index (drop=True)
pdf.head(5)

```

	Stn_Name	Lat	Long	Prov	Tm	...	BS%	HDD	CDD
0	CHEMAINUS	48.935	-123.742	BC	8.2	...	NaN	273.3	0
1	COWICHAN LAKE FORESTRY	48.824	-124.133	BC	7	...	NaN	307	0
2	LAKE COWICHAN	48.829	-124.052	BC	6.8	...	NaN	168.1	0
3	DUNCAN KELVIN CREEK	48.735	-123.728	BC	7.7	...	NaN	267.7	0
4	ESQUIMALT HARBOUR	48.432	-123.439	BC	8.8	...	NaN	258.6	0

5 rows × 25 columns

4-Visualization

We will create visualization of stations on map using basemap package. It is similar in functionality to the MATLAB mapping toolbox. The matplotlib basemap toolkit is a library for plotting 2D data on maps in Python.

Basemap does not do any plotting on its own, but provides the facilities to transform coordinates to a map projection. Matplotlib is then used to plot contours, images, vectors, lines or points in the transformed coordinates. Shoreline, river and political boundary datasets (from Generic Mapping Tools) are provided, along with methods for plotting them (Whitaker, 2016).

Basemap is geared toward the needs of earth scientists, particularly oceanographers and meteorologists. Jeff Whitaker originally wrote Basemap to help in his research (climate and weather forecasting), since at the time CDAT was the only other tool in python for plotting data on map projections. Over the years, the capabilities of Basemap have evolved as scientists in other disciplines (such as biology, geology and geophysics) requested and contributed new features.

Starting in 2016, Basemap came under new management. The Cartopy project will replace Basemap, but it hasn't yet implemented all of Basemap's features. Ben Root has volunteered to take over maintenance of Basemap until 2020. Pull requests will be reviewed, and regressions will be fixed. Also, this maintenance will ensure compatibility with packages like NumPy and Matplotlib.

Notice in **Figure 19-3** that the size of each data points represents the average of maximum temperature for each station in a year.

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from pylab import rcParams
%matplotlib inline
rcParams ["figure.dpi"] = 200.
llon = -140
ulon = -50
llat = 40
ulat = 65
```

```

pdf = pdf[(pdf['Long'] > llon) & (pdf['Long'] < ulon) &
           (pdf['Lat'] > llat) & (pdf['Lat'] < ulat)]

# See Table 19-2 for definitions
my_map = Basemap(projection='merc',
                  resolution = 'l', area_thresh = 1000.0,
                  llcrnrlon=llon, llcrnrlat=llat, #min Longitude
                  (llcrnrlon) and latitude (llcrnrlat)
                  urcrnrlon=ulon, urcrnrlat=ulat) #max Longitude
                  (urcrnrlon) and latitude (urcrnrlat)

my_map.drawcoastlines()
my_map.drawcountries()
# my_map.drawmapboundary()
my_map.fillcontinents(color = 'white', alpha = 0.3)
my_map.shadedrelief()

# To collect data based on stations
xs, ys = my_map(np.asarray(pdf.Long), np.asarray(pdf.Lat))
pdf['xm'] = xs.tolist()
pdf['ym'] = ys.tolist()
# Visualization1
for index, row in pdf.iterrows():
# x, y = my_map(row.Long, row.Lat)
    my_map.plot (row.xm, row.ym,
                 markerfacecolor = ([1,0,0]),
                 marker = 'o', markersize = 5, alpha = 0.75)
# plt.text(x, y, stn)
plt.show()

```

Table 19-2. The basemap map projection region specification keywords.

Keyword Description

llcrnrlon	longitude of lower left-hand corner of the desired map domain (degrees).
llcrnrlat	latitude of lower left-hand corner of the desired map domain (degrees).
urcrnrlon	longitude of upper right-hand corner of the desired map domain (degrees).
urcrnrlat	latitude of upper right-hand corner of the desired map domain (degrees).



Figure 19-3. DBSCAN can be used here, for instance, to plot the locations of Canadian weather stations (like a scatterplot) provided in our data.

5- Clustering of stations based on their location i.e., Lat & Lon

DBSCAN from sklearn library can run DBSCAN clustering from vector array or distance matrix. In our case, we pass it the Numpy array `Clus_dataSet` to find core samples of high density and expands clusters from them.

```
from sklearn.cluster import DBSCAN
import sklearn.utils
from sklearn.preprocessing import StandardScaler
sklearn.utils.check_random_state(1000)
Clus_dataSet = pdf[['xm','ym']]
Clus_dataSet = np.nan_to_num(Clus_dataSet)
Clus_dataSet = StandardScaler().fit_transform(Clus_dataSet)
# Compute DBSCAN
db = DBSCAN(eps = 0.15, min_samples=10).fit(Clus_dataSet)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_
pdf["Clus_Db"] = labels

realClusterNum = len(set(labels)) -
    (1 if -1 in labels else 0)
clusterNum = len(set(labels))
# A sample of clusters
pdf[["Stn_Name", "Tx", "Tm", "Clus_Db"]].head(5)
```

	Stn_Name	Tx	Tm	Clus_Db
0	CHEMAINUS	13.5	8.2	0
1	COWICHAN LAKE FORESTRY	15	7	0
2	LAKE COWICHAN	16	6.8	0
3	DUNCAN KELVIN CREEK	14.5	7.7	0
4	ESQUIMALT HARBOUR	13.1	8.8	0

As you can see for outliers, the cluster label is -1

```
set(labels)
{-1, 0, 1, 2, 3, 4}
```

6- Visualization of clusters based on location

Now, we can visualize the clusters using basemap in **Figure 19-4**:
from mpl_toolkits.basemap import Basemap

```
from pylab import rcParams
%matplotlib inline
rcParams ["figure.dpi"] = 200.

my_map = Basemap(projection='merc',
                  resolution = 'l', area_thresh = 1000.0,
                  llcrnrlon=llon, llcrnrlat = llat, #min Longitude
                  (llcrnrlon) and latitude (llcrnrlat)
                  urcrnrlon=ulon, urcrnrlat = ulat) #max Longitude
                  (urcrnrlon) and latitude (urcrnrlat)

my_map.drawcoastlines()
my_map.drawcountries()
my_map.fillcontinents(color = 'white', alpha = 0.3)
my_map.shadedrelief()
# To create a color map
colors = plt.get_cmap('jet')(np.linspace (0.0, 1.0,
clusterNum))
# Visualization1
for clust_number in set(labels):
    c=(([0.4, 0.4, 0.4])) if clust_number == -1 else
        colors[np.int(clust_number)])
    clust_set = pdf[pdf.Clus_Db == clust_number]
    my_map.scatter (clust_set.xm, clust_set.ym, color = c,
                    marker = 'o', s = 20, alpha = 0.85)
    if clust_number != -1:
        cenx = np.mean (clust_set.xm)
        ceny = np.mean(clust_set.ym)
        plt.text(cenx, ceny, str(clust_number),
                  fontsize = 24, color = 'magenta',)
        print ("Cluster " +str(clust_number)+ ',Avg Temp: '
              + str(np.mean(clust_set.Tm)))
```

Cluster 0, Avg Temp: -5.538747553816051
Cluster 1, Avg Temp: 1.9526315789473685
Cluster 2, Avg Temp: -9.195652173913045
Cluster 3, Avg Temp: -15.300833333333333
Cluster 4, Avg Temp: -7.769047619047619

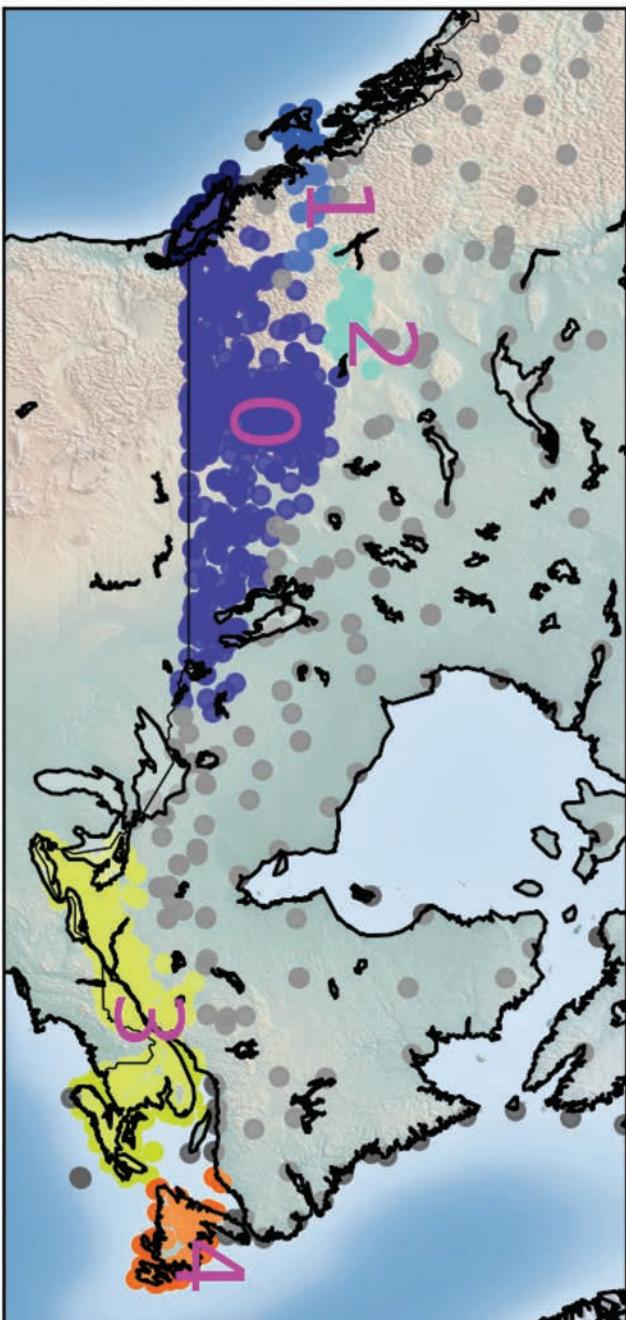


Figure 19-4. DBSCAN clustering from vector array or distance matrix.

7- Clustering of stations based on their location, mean, max, and min Temperature

In this section we re-run DBSCAN, but this time on a 5-dimensional dataset. The map is displayed in **Figure 19-5**.

```
from sklearn.cluster import DBSCAN
import sklearn.utils
from sklearn.preprocessing import StandardScaler
sklearn.utils.check_random_state(1000)
Clus_dataSet = pdf[['xm', 'ym', 'Tx', 'Tm', 'Tn']]
Clus_dataSet = np.nan_to_num(Clus_dataSet)
Clus_dataSet = StandardScaler().fit_transform
(Clus_dataSet)

# Compute DBSCAN
db = DBSCAN(eps = 0.3, min_samples=10).fit(Clus_dataSet)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_
pdf["Clus_Db"] = labels
realClusterNum = len(set(labels)) - (1 if -1 in
                                       labels else 0)
clusterNum = len(set(labels))

# A sample of clusters
pdf[["Stn_Name", "Tx", "Tm", "Clus_Db"]].head(5)
```

	Stn_Name	Tx	Tm	Clus_Db
0	CHEMAINUS	13.5	8.2	0
1	COWICHAN LAKE FORESTRY	15.0	7.0	0
2	LAKE COWICHAN	16.0	6.8	0
3	DUNCAN KELVIN CREEK	14.5	7.7	0
4	ESQUIMALT HARBOUR	13.1	8.8	0

8- Visualization of Clusters Based on Location and Temperature

The clusters of locations and temperature similarities are shown in **Figure 19-5**.

```

from pylab import rcParams
%matplotlib inline
rcParams ["figure.dpi"] = 200.
my_map = Basemap(projection='merc',
                  resolution = 'l', area_thresh = 1000.0,
                  llcrnrlon = llon, llcrnrlat = llat, #min Longitude
                  (llcrnrlon) and latitude (llcrnrlat)
                  Urccrnrlon = ulon, urccrnrlat = ulat) #max Longitude
                  (urccrnrlon) and latitude (urccrnrlat)

my_map.drawcoastlines()
my_map.drawcountries()
my_map.fillcontinents(color = 'white', alpha = 0.3)
my_map.shadedrelief()

# To create a color map
colors = plt.get_cmap('jet')(np.linspace(0.0, 1.0,
                                         clusterNum))
# Visualization1
for clust_number in set(labels):
    c = (([0.4, 0.4, 0.4])) if clust_number == -1 else
colors[np.int(clust_number)])
    clust_set = pdf[pdf.Clus_Db == clust_number]
    my_map.scatter (clust_set.xm, clust_set.ym, color = c,
                    marker = 'o', s = 20, alpha = 0.85)
    if clust_number != -1:
        cenx = np.mean(clust_set.xm)
        ceny = np.mean(clust_set.ym)
        plt.text(cenx, ceny, str(clust_number),
                  fontsize = 20, color = 'magenta')
        print ("Cluster " +str(clust_number)+', Avg Temp: '
+ str(np.mean (clust_set.Tm)))

```

Cluster 0, Avg Temp:	6.221192052980133
Cluster 1, Avg Temp:	6.790000000000001
Cluster 2, Avg Temp:	-0.49411764705882355
Cluster 3, Avg Temp:	-13.877209302325586
Cluster 4, Avg Temp:	-4.186274509803922
Cluster 5, Avg Temp:	-16.301503759398482
Cluster 6, Avg Temp:	-13.599999999999998
Cluster 7, Avg Temp:	-9.753333333333334
Cluster 8, Avg Temp:	-4.258333333333334

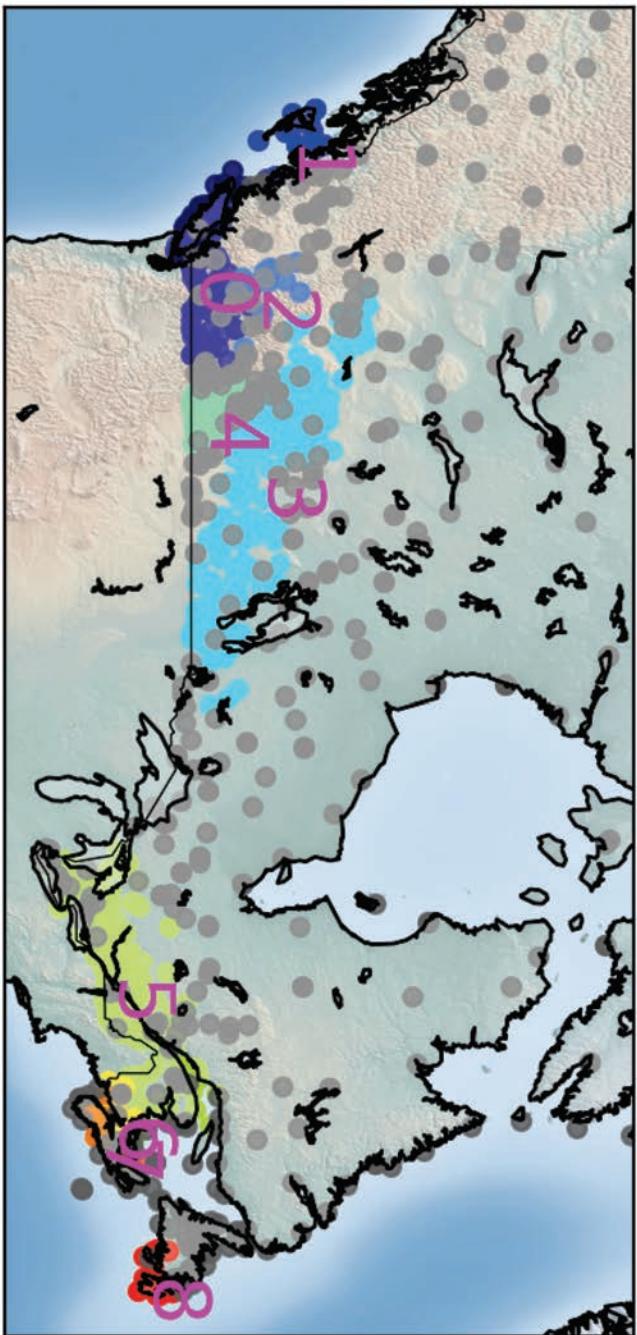


Figure 19-5. DBSCAN clusters of locations and temperature similarities.

20. Content Based Filtering

Objectives

After completing this chapter, you will be able to:

- Create a recommendation system using collaborative filtering

Recommendation systems are a collection of algorithms used to recommend items to users based on information taken from the user. These systems have become ubiquitous, and can be commonly seen in online stores, movies databases and job finders. In this notebook, we will explore Content-based recommendation systems and implement a simple version of one using Python and the Pandas library.

Table of contents

1. Acquiring the Data
2. Preprocessing
3. Content-Based Filtering

Acquiring the Data

The dataset we'll use is acquired from **GroupLens**. To acquire the data, well need a function that downloads a compressed folder (ZIP), saves it to our working directory by writing it to a temporary file that we close afterward. Then we use the defined function to unzip the compressed file and saving the files it contains in a selected folder.

```
def uzip(path, filename, filepath):  
  
    import os  
    import zipfile  
    from urllib.request import urlopen  
    from zipfile import ZipFile  
  
    zipurl = path  
    # Download the file from the URL  
    zipresp = urlopen(zipurl)
```

```
# Create a new file on the hard drive
    tempzip = open(filepath, "wb")
    # Write the contents of the downloaded file into the
new file
    tempzip.write(zipresp.read())
    # Close the newly-created file
    tempzip.close()
    # Re-open the newly-created file with ZipFile()
    with ZipFile(filename, 'r') as zipObj:
        # Extract all the contents of zip file in current
directory
            zipObj.extractall()
    zipObj.close()
```

Now, we use the `uzip()` function we just defined, has three parameters: `path`, `filename`, and `filepath`.

```
path = 'https://files.grouplens.org/datasets/movielens/ml-
latest-small.zip'
filename = 'moviedataset.zip'
filepath = './moviedataset.zip'
uzip(path, filename, filepath)
```

Now you're ready to start working with the data!

Preprocessing

First, let's get all of the imports out of the way:

```
# Dataframe manipulation library
import pandas as pd
# Math functions, we'll only need the sqrt function so
let's import only that
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Now let's read each file into their Dataframes:

```
# Storing the movie information into a pandas dataframe
movies_df = pd.read_csv('./ml-latest/movies.csv')
# Storing the user information into a pandas dataframe
```

```

ratings_df = pd.read_csv('./ml-latest/ratings.csv')
# Head is a function that gets the first N rows of a
# dataframe. N's default is 5.
movies_df.head()

```

Id	title	genres
1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy
2	Jumanji (1995)	Adventure Children Fantasy
3	Grumpier Old Men (1995)	Comedy Romance
4	Waiting to Exhale (1995)	Comedy Drama Romance
5	Father of the Bride Part II (1995)	Comedy

Let's also remove the year from the **title** column by using **pandas'** replace function and store in a new **year** column. We'll use regular expressions to find a year stored between parentheses. We specify the parentheses so we don't conflict with movies that have years in their titles

```

movies_df['year'] =
movies_df.title.str.extract('(\(\d\d\d\d\))', expand=False)
# Removing the parentheses
movies_df['year'] =
movies_df.year.str.extract('(\d\d\d\d)', expand=False)
# Removing the years from the 'title' column
movies_df['title'] =
movies_df.title.str.replace('(\(\d\d\d\d\))', '')
# Applying the strip function to get rid of any ending
# whitespace characters that may have appeared
movies_df['title'] = movies_df['title'].apply(lambda x:
x.strip())
movies_df.head()

```

Id	title	genres	year
1	Toy Story	Adventure Animation Children Comedy Fantasy	1995
2	Jumanji	Adventure Children Fantasy	1995
3	Grumpier Old Men	Comedy Romance	1995
4	Waiting to Exhale	Comedy Drama Romance	1995
5	Father of the Bride Part II	Comedy	1995

With that, let's also split the values in the **Genres** column into a **list of Genres** to simplify for future use. This can be achieved by applying Python's split string function on the correct column.

```
# Every genre is separated by a | so we simply have to
# call the split function on |
movies_df['genres'] = movies_df.genres.str.split('|')
movies_df.head()
```

Id	title	genres	year
1	Toy Story	[Adventure, Animation, Children, Comedy, Fantasy]	1995
2	Jumanji	[Adventure, Children, Fantasy]	1995
3	Grumpier Old Men	[Comedy, Romance]	1995
4	Waiting to Exhale	[Comedy, Drama, Romance]	1995
5	Father of the Bride Part II	[Comedy]	1995

Since keeping genres in a list format isn't optimal for the content-based recommendation system technique, we will use the One Hot Encoding technique to convert the list of genres to a vector where each column corresponds to one possible value of the feature. This encoding is needed for feeding categorical data. In this case, we store every different genre in columns that contain either 1 or 0. 1 shows that a movie has that genre and 0 shows that it doesn't. Let's also store this dataframe in another variable since genres won't be important for our first recommendation system.

```
# Copying the movie dataframe into a new one since we
# won't need to use the genre information in our first case.

moviesWithGenres_df = movies_df.copy()

#For every row in the dataframe, iterate through the list
#of genres and place a 1 into the corresponding column

for index, row in movies_df.iterrows():
    for genre in row['genres']:
        moviesWithGenres_df.at[index, genre] = 1
# Filling in the NaN values with 0 to show that a movie
# doesn't have that column's genre
moviesWithGenres_df = moviesWithGenres_df.fillna(0)
moviesWithGenres_df.head()
```

Id	title	genres	year	ture	Adven-	Anima-	Child-	Com-	...
1	Toy Story	[Adventure, Animation, Children, Comedy, Fantasy]	1995	1	1	1	1	1	1 ...
2	Jumanji	[Adventure, Children, Fantasy]	1995	1	0	1	0	0	0 ...
3	Grumpier Old Men	[Comedy, Romance]	1995	0	0	0	1	1	1 ...
4	Waiting to Exhale	[Comedy, Drama, Romance]	1995	0	0	0	1	1	1 ...
5	Father of the Bride Part II	[Comedy]	1995	0	0	0	1	1	1 ...

5 rows × 24 columns

Next, let's look at the ratings dataframe.

```
ratings_df.head()
```

userId	movielid	rating	timestamp
1	307	3.5	1256677221
1	481	3.5	1256677456
1	1091	1.5	1256677471
1	1257	4.5	1256677460
1	1449	4.5	1256677264

Every row in the ratings dataframe has a user id associated with at least one movie, a rating and a timestamp showing when they reviewed it. We won't be needing the timestamp column, so let's drop it to save memory.

```
# Drop removes a specified row or column from a dataframe
ratings_df = ratings_df.drop ('timestamp', 1)
ratings_df.head()
```

	userId	movieId	rating
0	1	307	3.5
1	1	481	3.5
2	1	1091	1.5
3	1	1257	4.5
4	1	1449	4.5

Content-Based recommendation system

Now, let's take a look at how to implement **Content-Based** or **Item-Item recommendation systems**. This technique attempts to figure out what a user's favorite aspects of an item is, and then recommends items that present those aspects. In our case, we're going to try to figure out the input's favorite genres from the movies and ratings given.

Let's begin by creating an input user to recommend movies to:

Notice: To add more movies, simply increase the number of elements in the **userInput**. Feel free to add more in! Just be sure to write it in with capital letters and if a movie starts with a "The", like "The Matrix" then write it in like this: 'Matrix, The' .

```
userInput = [
    {'title':'Breakfast Club, The', 'rating':5},
    {'title':'Toy Story', 'rating':3.5},
    {'title':'Jumanji', 'rating':2},
    {'title':'Pulp Fiction', 'rating':5},
    {'title':'Akira', 'rating':4.5}
]
inputMovies = pd.DataFrame (userInput)
inputMovies
```

	title	rating
0	Breakfast Club, The	5.0
1	Toy Story	3.5
2	Jumanji	2.0
3	Pulp Fiction	5.0
4	Akira	4.5

With the input complete, let's extract the input movie's ID's from the movies dataframe and add them into it.

We can achieve this by first filtering out the rows that contain the input movie's title and then merging this subset with the input dataframe. We also drop unnecessary columns for the input to save memory space.

```
# Filtering out the movies by title
inputId =
movies_df[movies_df['title'].isin(inputMovies['title'].tolist())]
# Then merging it so we can get the movieId. It's
# implicitly merging it by title.
inputMovies = pd.merge(inputId, inputMovies)
# Dropping information we won't use from the input
# dataframe
inputMovies = inputMovies.drop ('genres', 1).drop('year',
1)
# Final input dataframe
# If a movie you added in above isn't here, then it might
not be in the original
# Dataframe or it might spelled differently, please check
capitalization.
inputMovies
```

	moviedb_id	title	rating
0	1	Toy Story	3.5
1	2	Jumanji	2.0
2	296	Pulp Fiction	5.0
3	1274	Akira	4.5
4	1968	Breakfast Club	5.0

We're going to start by learning the input's preferences, so let's get the subset of movies that the input has watched from the Dataframe containing genres defined with binary values.

```
# Filtering out the movies from the input
userMovies =
moviesWithGenres_df[moviesWithGenres_df['movieId'].isin(inputMovies['movieId'].tolist())]
userMovies
```

Id	title	genres	year	Adven-		Anima-		Child-		...
				ture	tion	ren	...			
1	Toy Story	[Adventure, Animation, Children, Com...]	1995	1	1	1	1	...		
2	Jumanji	[Adventure, Children, Fantasy]	1995	1	0	1	1	...		
296	Pulp Fiction	[Comedy, Crime, Drama, Thriller]	1994	0	0	0	0	...		
1274	Akira	[Action, Adventure, Animation...]	1988	1	1	0	0	...		
1968	Breakfast Club, The	[Comedy, Drama]	1985	0	0	0	0	...		
164600	Akira	[Action, Crime, Thriller]	2016	0	0	0	0	...		

5 rows × 24 columns

We'll only need the actual genre table, so let's clean this up a bit by resetting the index and dropping the `movieId (ID)`, `title`, `genres` and `year` columns.

```
# Resetting the index to avoid future issues
userMovies = userMovies.reset_index (drop=True)
# Dropping unnecessary issues due to save memory and to
# avoid issues
userGenreTable = userMovies.drop ('movieId',
1).drop('title', 1).drop('genres', 1).drop('year', 1)
userGenreTable
```

	Adventure	Animation	Children	Comedy	Fantasy	Drama	Action
0	1	1	1	1	1	0	0
1	1	0	1	0	1	0	0
2	0	0	0	1	0	1	0
3	1	1	0	0	0	0	1
4	0	0	0	1	0	1	0
5	0	0	0	0	0	0	1

Now we're ready to start learning the input's preferences!

To do this, we're going to turn each genre into weights. We can do this by using the input's reviews and multiplying them into the input's genre table and then summing up the resulting table by column. This operation is actually a dot product between a matrix and a vector, so we can simply accomplish by calling the Pandas "dot" function.

```
inputMovies['rating']  
0    3.5  
1    2.0  
2    5.0  
3    4.5  
4    5.0  
Name: rating, dtype: float64
```

```
# Dot product to get weights  
userProfile =  
userGenreTable.transpose().dot(inputMovies['rating'])  
# The user profile  
userProfile
```

```
Adventure           10.0  
Animation          8.0  
Children           5.5  
Comedy             13.0  
Fantasy            5.5  
Romance            0.0  
Drama              9.5  
Action              9.5  
Crime              10.0  
Thriller           10.0  
Horror              0.0  
Mystery             0.0  
Sci-Fi              4.5  
IMAX                0.0  
Documentary         0.0  
War                 0.0  
Musical              0.0  
Western              0.0  
Film-Noir            0.0  
(no genres listed)  0.0  
dtype: float64
```

Now, we have the weights for every of the user's preferences. This is

known as the User Profile. Using this, we can recommend movies that satisfy the user's preferences.

Let's start by extracting the genre table from the original dataframe:

```
# Now let's get the genres of every movie in our original
# dataframe
genreTable =
    moviesWithGenres_df.set_index(moviesWithGenres_df['
        movieId'])
# And drop the unnecessary information
genreTable = genreTable. drop('movieId', 1).drop('title',
    1).drop('genres', 1).drop ('year', 1)
genreTable.head()
```

	Adventure	Animation	Children	Comedy	Fantasy	Romance
Id						
1	1.0	1.0	1.0	1.0	1.0	0.0
2	1.0	0.0	1.0	0.0	1.0	0.0
3	0.0	0.0	0.0	1.0	0.0	1.0
4	0.0	0.0	0.0	1.0	0.0	1.0
5	0.0	0.0	0.0	1.0	0.0	0.0

```
genreTable.shape
```

```
(58098, 20)
```

With the input's profile and the complete list of movies and their genres in hand, we're going to take the weighted average of every movie based on the input profile and recommend the top twenty movies that most satisfy it.

```
# Multiply the genres by the weights and then take the
# weighted average
recommendationTable_df =
((genreTable*userProfile).sum(axis=1))/(userProfile.sum())
recommendationTable_df.head()

movieId
1    0.491228
2    0.245614
3    0.152047
4    0.263158
5    0.152047
dtype: float64
```

```

# Sort our recommendations in descending order
recommendationTable_df =
recommendationTable_df.sort_values(ascending=False)
# Just a peek at the values
recommendationTable_df.head()

```

```

movieId
5018      0.730994
122787    0.725146
144324    0.725146
64645     0.725146
81132     0.725146
dtype: float64

```

Now here's the recommendation table!

```

# The final recommendation table
movies_df.loc [movies_df['movieId'].isin
(recommendationTable_df.head(20).keys())]

```

movieId	title	genres	year
4625	4719	Osmosis Jones	[Action, Animation, Comedy, Crime, Drama, Roma...]
4923	5018	Motorama	[Adventure, Comedy, Crime, Drama, Fantasy, Mys...]
6394	6503	Charlie's Angels: Full Throttle	[Action, Adventure, Comedy, Crime, Thriller]
6793	6902	Interstate 60	[Adventure, Comedy, Drama, Fantasy, Mystery, S...]
8285	8968	After the Sunset	[Action, Adventure, Comedy, Crime, Thriller]
8672	26184	Diamond Arm, The (Brilliantovaya ruka)	[Action, Adventure, Comedy, Crime, Thriller]
9300	27344	Revolutionary Girl Utena: Adolescence...	[Action, Adventure, Animation, Comedy, Drama...]
13271	64645	The Wrecking Crew	[Action, Adventure, Comedy, Crime, Drama, Thri...]
15038	75408	Lupin III: Sweet Lost Night (Rupan Sanse...	[Action, Animation, Comedy, Crime, Drama, Myst...]
15112	76153	Lupin III: First Contact (Rupan Sansei...)	[Action, Animation, Comedy, Crime, Drama, Myst...]
15872	80219	Machete	[Action, Adventure, Comedy, Crime, Thriller]
16105	81132	Rubber	[Action, Adventure, Comedy, Crime, Drama, Film...]

movieId	title	genres	year
18447	91542	Sherlock Holmes: A Game of Shadows	[Action, Adventure, Comedy, Crime, Mystery, Thr...]
24934	115333	Charlie Chan in Panama	[Adventure, Comedy, Crime, Drama, Mystery, Thr...]
24971	115479	Whip Hand, The	[Action, Adventure, Crime, Drama, Sci-Fi, Thr...]
25746	117646	Dragonheart 2: A New Beginning	[Action, Adventure, Comedy, Drama, Fantasy, Thr...]
27500	122787	The 39 Steps	[Action, Adventure, Comedy, Crime, Drama, Thr...]
27862	123598	The Spy in the Green Hat	[Action, Adventure, Comedy, Crime, Thriller]
36321	144324	Once Upon a Time	[Action, Adventure, Comedy, Crime, Drama, Roma...]
38457	149488	Christmas Town	[Action, Children, Comedy, Drama, Fantasy, Thr...]

Advantages and Disadvantages of Content-Based Filtering

Advantages

- Learns user's preferences
- Highly personalized for the user

Disadvantages

- Doesn't take into account what others think of the item, so low-quality item recommendations might happen
- Extracting data is not always intuitive
- Determining what characteristics of the item the user dislikes or likes is not always obvious

21. Collaborative Filtering

Objectives

After completing this chapter, you will be able to:

- Create recommendation system based on collaborative filtering

Recommendation systems are a collection of algorithms used to recommend items to users based on information taken from the user. These systems have become ubiquitous and can be commonly seen in online stores, movies databases and job finders. In this notebook, we will explore recommendation systems based on Collaborative Filtering and implement simple version of one using Python and the Pandas library (Vsevolodovna, 2020).

Table of contents

1. Acquiring the Data
2. Preprocessing
3. Collaborative Filtering

Acquiring the Data

Dataset acquired from GroupLens (Konstan., 2015). The compressed files are available for download from my GitHub *Jupyter Data* repository.

<https://files.grouplens.org/datasets/movielens/ml-latest-small.zip>

To acquire the data, we start with defining a function that will fetch the compressed (ZIP) file and save it to our local working directory, once executed.

```
def uzip(path, filename, filepath):
    import os
    import zipfile
    from urllib.request import urlopen
    from zipfile import ZipFile

    zipurl = path
    # Download the file from the URL
    zipresp = urlopen(zipurl)
```

```

# Create a new file on the hard drive
tempzip = open("./moviedataset.zip", "wb")
# Write the contents of the downloaded file into the
# new file
tempzip.write(zipresp.read())
# Close the newly-created file
tempzip.close()
# Re-open the newly-created file with ZipFile()
with ZipFile(filename, 'r') as zipObj:
    # Extract all the contents of zip file in current
    # directory
    zipObj.extractall()
    zipObj.close()

```

Next, we execute the code (unzip) to download and unzip the compressed file we downloaded to our working directory.

```

path = https://files.grouplens.org/datasets/movielens/ml-
latest-small.zip'
filename = 'moviedataset.zip'
filepath = './moviedataset.zip'
uzip(path, filename, filepath)

```

Now you're ready to start working with the data!

Preprocessing

First, let's get all of the imports out of the way:

```

# Dataframe manipulation Library
import pandas as pd
# Math functions, we'll only need the sqrt function so
# Let's import only that
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

```

Now let's read each file into their Dataframes:

```

# Storing the movie information into a pandas dataframe
movies_df = pd.read_csv('./ml-latest/movies.csv')
# Storing the user information into a pandas dataframe
ratings_df = pd.read_csv('./ml-latest/ratings.csv')

```

Let's also take a peek at how each of them are organized:

```
# Head is a function that gets the first N rows of a
# dataframe. N's default is 5.
movies_df.head()
```

Id	title	genres
1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy
2	Jumanji (1995)	Adventure Children Fantasy
3	Grumpier Old Men (1995)	Comedy Romance
4	Waiting to Exhale (1995)	Comedy Drama Romance
5	Father of the Bride Part II (1995)	Comedy

So, each movie has a unique ID, a title with its release year along with it (Which may contain Unicode characters) and several different genres in the same field. Let's remove the year from the title column and place it into its own one by using the handy extract function that Pandas has.

Let's remove the year from the **title** column by using [pandas'](#) replace function and store it in a new **year** column.

```
# Using regular expressions to find a year stored between
# parentheses
# We specify the parentheses so we don't conflict with
# movies that have years in their titles
movies_df['year'] =
movies_df.title.str.extract('(\(\d\d\d\d\))',expand=False)
# Removing the parentheses
movies_df['year'] =
movies_df.year.str.extract('(\d\d\d\d)',expand=False)
# Removing the years from the 'title' column
movies_df['title'] =
movies_df.title.str.replace('(\(\d\d\d\d\))', '')
# Applying the strip function to get rid of any ending
# whitespace characters that may have appeared
movies_df['title'] = movies_df['title'].apply(lambda x:
x.strip())
```

Let's look at the result!

```
movies_df.head()
```

Id	title	genres	year
1	Toy Story	Adventure Animation Children Comedy Fantasy	1995
2	Jumanji	Adventure Children Fantasy	1995
3	Grumpier Old Men	Comedy Romance	1995
4	Waiting to Exhale	Comedy Drama Romance	1995
5	Father of the Bride Part II	Comedy	1995

With that, let's also drop the genres column since we won't need it for this particular recommendation system.

```
# Dropping the genres column  
movies_df = movies_df.drop ('genres', 1)
```

Here's the final movies dataframe:

```
movies_df.head()
```

moviedId	title	year
1	Toy Story	1995
2	Jumanji	1995
3	Grumpier Old Men	1995
4	Waiting to Exhale	1995
5	Father of the Bride Part II	1995

Next, let's look at the ratings dataframe.

```
ratings_df.head()
```

userId	moviedId	rating	timestamp
1	169	2.5	1204927694
1	2471	3.0	1204927438
1	48516	5.0	1204927435
2	2571	3.5	1436165433
2	109487	4.0	1436165496

Every row in the ratings dataframe has a user id associated with at least one movie, a rating and a timestamp showing when they reviewed it. We won't be needing the timestamp column, so let's drop it to save on memory.

```
# Drop removes a specified row or column from a dataframe  
ratings_df = ratings_df.drop('timestamp', 1)
```

Here's how the final ratings Dataframe looks like:

```
ratings_df.head()
```

	userId	movieId	rating
0	1	169	2.5
1	1	2471	3.0
2	1	48516	5.0
3	2	2571	3.5
4	2	109487	4.0

Collaborative Filtering

Now it's time to start our work on recommendation systems.

The first technique we're going to take a look at is called **Collaborative Filtering**, which is also known as **User-User Filtering**. As hinted by its alternate name, this technique uses other users to recommend items to the input user. It attempts to find users that have similar preferences and opinions as the input and then recommends items that they have liked to the input. There are several methods of finding similar users (Even some making use of Machine Learning), and the one we will be using here is going to be based on the **Pearson Correlation Function**.

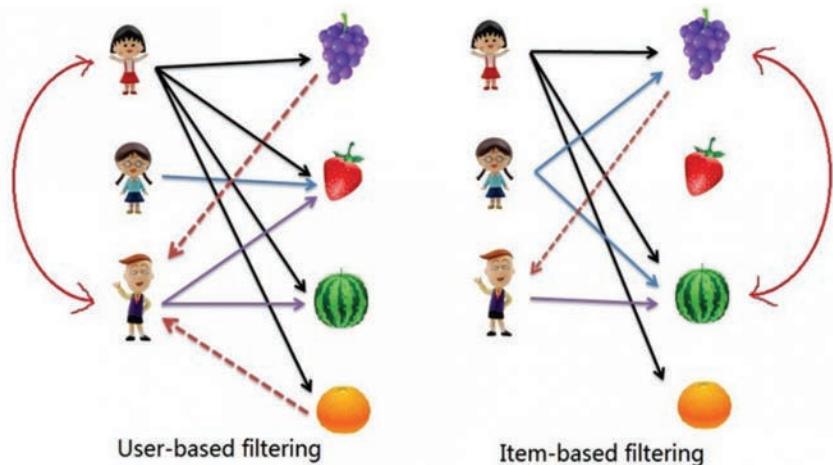


Figure 21-1. User-based filtering vs. item-based filtering (Pinela, 2017)

The process for creating a User Based recommendation system is as follows:

- Select a user with the movies the user has watched
- Based on his rating of the movies, find the top X neighbors
- Get the watched movie record of the user for each neighbor
- Calculate a similarity score using some formula
- Recommend the items with the highest score

Let's begin by creating an input user to recommend movies to. To add more movies, simply increase the number of elements in the [user Input](#). Feel free to add more in! Just be sure to write it in with capital letters and if a movie starts with a "The", like "The Matrix" then write it in like this: 'Matrix, The'.

```
userInput = [
    {'title': 'Breakfast Club, The', 'rating':5},
    {'title': 'Toy Story', 'rating':3.5},
    {'title': 'Jumanji', 'rating':2},
    {'title': 'Pulp Fiction', 'rating':5},
    {'title': 'Akira', 'rating':4.5}
]
inputMovies = pd.DataFrame (userInput)
inputMovies
```

	title	rating
0	Breakfast Club, The	5.0
1	Toy Story	3.5
2	Jumanji	2.0
3	Pulp Fiction	5.0
4	Akira	4.5

Add movieID to input user

With the input complete, let's extract the input movies' ID's from the movies dataframe and add them into it.

We can achieve this by first filtering out the rows that contain the input movies' title and then merging this subset with the input dataframe. We also drop unnecessary columns for the input to save memory space.

```

#Filtering out the movies by title
inputId = movies_df[movies_df['title'].isin
(inputMovies['title'].tolist ())]
# Then merging it so we can get the movieId. It's
implicitly merging it by title.
inputMovies = pd.merge (inputId, inputMovies)
# Dropping information we won't use from the input
dataframe
inputMovies = inputMovies.drop ('year', 1)
# Final input dataframe
# If a movie you added in above isn't here, then it might
not be in the original
# Dataframe or it might spelled differently, please check
capitalization.
inputMovies

```

	movielid	title	rating
0	1	Toy Story	3.5
1	2	Jumanji	2.0
2	296	Pulp Fiction	5.0
3	1274	Akira	4.5
4	1968	Breakfast Club, The	5.0

Users Who have Seen the Same Movies

Now with the movie ID's in our input, we can now get the subset of users that have watched and reviewed the movies in our input.

```

#Filtering out users that have watched movies that the
input has watched and storing it
userSubset =
    ratings_df[ratings_df['movieId'].isin(inputMovies
    ['movieId'].tolist())]
userSubset.head()

```

	userId	movielid	rating
19	4	296	4.0
441	12	1968	3.0
479	13	2	2.0
531	13	1274	5.0
681	14	296	2.0

We now group up the rows by user ID. Here, we use `groupby` to

create several sub dataframes where they all have the same value in the column, specified as the parameter

```
userSubsetGroup = userSubset.groupby(['userId'])
```

Let's look at one of the users, e.g., the one with userID=1130.

```
userSubsetGroup.get_group (21)
```

	userId	movieId	rating
104167	1130	1	0.5
104168	1130	2	4.0
104214	1130	296	4.0
104363	1130	1274	4.5
104443	1130	1968	4.5

Let's also sort these groups so the users that share the most movies in common with the input have higher priority. This provides a richer recommendation since we won't go through every single user. Sorting the data so users with movie most in common with the input will have priority

```
userSubsetGroup = sorted(userSubsetGroup, key=lambda x:  
    len(x[1]), reverse=True)
```

Now let's look at the first user.

```
userSubsetGroup[0:3]
```

```
[(75,      userId  movieId  rating  
  7507     75       1        5.0  
  7508     75       2        3.5  
  7540     75       296      5.0  
  7633     75       1274     4.5  
  7673     75       1968     5.0), (106,      userId  
movieId  rating  
  9083     106      1        2.5  
  9084     106      2        3.0  
  9115     106      296      3.5  
  9198     106      1274     3.0  
  9238     106      1968     3.5), (686,      userId  
movieId  rating  
  61336    686      1        4.0
```

61337	686	2	3.0
61377	686	296	4.0
61478	686	1274	4.0
61569	686	1968	5.0)]

Similarity of Users to Input User

Next, we are going to compare all users (not really all !!!) to our specified user and find the one that is most similar. We're going to find out how similar each user is to the input through the **Pearson Correlation Coefficient**. It is used to measure the strength of a linear association between the two variables. The formula for finding this coefficient between sets X and Y with N values can be seen in the image below.

Pearson Correlation

Pearson correlation is invariant to scaling, i.e., multiplying all elements by a nonzero constant or adding any constant to all elements. For example, if you have two vectors X and Y, then, $\text{pearson}(X, Y) == \text{pearson}(X, 2 * Y + 3)$. This is a pretty important property in recommendation systems because, for example, two users might rate two series of items totally differently in terms of absolute rates, but they would be similar users (i.e., with similar ideas) with similar rates in various scales .

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

The values given by the formula vary from $r = -1$ to $r = 1$, where 1 forms a direct correlation between the two entities (it means a perfect positive correlation) and -1 forms a perfect negative correlation.

In our case, a 1 means that the two users have similar tastes while a -1 means the opposite. We will select a subset of users to iterate through. This limit is imposed because we don't want to waste too much time going through every single user.

```
userSubsetGroup = userSubsetGroup[0:100]
```

Calculating the Pearson Correlation

Now, we calculate the Pearson Correlation between input user and subset group, and store it in a dictionary, where the key is the user Id and the value is the coefficient.

```
#Store the Pearson Correlation in a dictionary, where the
key is the user Id and the value is the coefficient
pearsonCorrelationDict = {}

# For every user group in our subset
for name, group in userSubsetGroup:
    # Let's start by sorting the input and current user
    # group so the values aren't mixed up later on
    group = group.sort_values(by='movieId')
    inputMovies = inputMovies.sort_values(by='movieId')

    # Get the N for the formula
    nRatings = len(group)

    # Get the review scores for the movies that they both
    # have in common
    temp_df = inputMovies[inputMovies['movieId'].isin(
        (group['movieId'].tolist ()))]

    # And then store them in a temporary buffer variable
    # in a List format to facilitate future calculations
    tempRatingList = temp_df['rating'].tolist()

    # Let's also put the current user group reviews in a
    # List format
    tempGroupList = group['rating'].tolist()

    # Now let's calculate the Pearson correlation between
    # two users, so called, x and y
    Sxx = sum([i**2 for i in tempRatingList]) -
        pow(sum(tempRatingList),2)/float(nRatings)
    Syy = sum([i**2 for i in tempGroupList]) -
        pow(sum(tempGroupList),2)/float(nRatings)
    Sxy = sum( i*j for i, j in zip(tempRatingList,
        tempGroupList)) -
        sum(tempRatingList)*sum(tempGroupList)/
        float(nRatings)
```

```

# If the denominator is different than zero, then
# divide, else, 0 correlation.
if Sxx != 0 and Syy != 0:
    pearsonCorrelationDict[name] = Sxy/sqrt(Sxx*Syy)
else:
    pearsonCorrelationDict[name] = 0
pearsonCorrelationDict.items()

dict_items([(75, 0.8272781516947562), (106,
0.5860090386731182), (686, 0.8320502943378437), (815,
0.5765566601970551), (1040, 0.9434563530497265), (1130,
0.2891574659831201), (1502, 0.8770580193070299), (1599,
0.4385290096535153), (1625, 0.7161148740394320), (1950,
0.1790287185098580), (2065, 0.4385290096535153), (2128,
0.5860090386731196), (2432, 0.1386750490563073), (2791,
0.8770580193070299), (2839, 0.8204126541423674), (2948,
0.1172018077346239), (3025, 0.4512426281971397), (3040,
0.8951435925492900), (3186, 0.6784622064861935), (3271,
:
:
0.9607689228305227), (13142, 0.6016568375961863), (13260,
0.7844645405527362), (13366, 0.8951435925492911), (13768,
0.8770580193070289), (13888, 0.2508726030021272), (13923,
0.3516054232038718), (13934, 0.1720052290384455), (14529,
0.7417901772340937), (14551, 0.5370861555295740), (14588,
0.21926450482675766), (14984, 0.716114874039432), (15137,
0.5860090386731196), (15157, 0.9035841064985974), (15466,
0.7205766921228921), (15670, 0.516015687115336), (15834,
0.2256213140985698), (16292, 0.657793514480271), (16456,
0.7161148740394331), (16506, 0.548161262066894), (17246,
0.4803844614152613), (17438, 0.709316988616438), (17501,
0.8168748513121271), (17502, 0.827278151694756), (17666,
0.7689238340176859), (17735, 0.704238182012342), (17742,
0.3922322702763681), (17757, 0.646575750139840), (17854,
0.5370861555295740), (17897, 0.877058019307028), (17944,
0.2713848825944774), (18301, 0.298381197516430), (18509,
0.1322214713369862)])]

pearsonDF = pd.DataFrame.from_dict(pearsonCorrelationDict,
orient='index')
pearsonDF.columns = ['similarityIndex']
pearsonDF['userId'] = pearsonDF.index
pearsonDF.index = range(len(pearsonDF))
pearsonDF.head()

```

	similarityIndex	userId
0	0.827278	75
1	0.586009	106
2	0.832050	686
3	0.576557	815
4	0.943456	1040

The Top X Similar Users to Input User

Now let's get the top 50 users that are most similar to the input.

```
topUsers=pearsonDF.sort_values(by='similarityIndex',
ascending=False)[0:50]
topUsers.head()
```

	similarityIndex	userId
64	0.961678	12325
34	0.961538	6207
55	0.961538	10707
67	0.960769	13053
4	0.943456	1040

Now, let's start recommending movies to the input user.

Rating of Selected Users to All Movies

We're going to do this by taking the weighted average of the ratings of the movies using the Pearson Correlation as the weight. But to do this, we first need to get the movies watched by the users in our **pearsonDF** from the ratings dataframe and then store their correlation in a new column called _similarityIndex". This is achieved below by merging of these two tables.

```
topUsersRating=topUsers.merge(ratings_df,
left_on='userId', right_on='userId', how='inner')
topUsersRating.head()
```

	similarityIndex	userId	movieId	rating
0	0.961678	12325	1	3.5
1	0.961678	12325	2	1.5
2	0.961678	12325	3	3.0
3	0.961678	12325	5	0.5
4	0.961678	12325	6	2.5

Now all we need to do is simply multiply the movie rating by its weight (the similarity index), then sum up the new ratings and divide it by the sum of the weights.

We can easily do this by simply multiplying two columns, then grouping up the dataframe by movieId and then dividing two columns:

It shows the idea of all similar users to candidate movies for the input user:

```
#Multiplies the similarity by the user's ratings
topUsersRating['weightedRating'] =
topUsersRating['similarityIndex']*topUsersRating['rating']
topUsersRating.head()
```

	similarityIndex	userId	movieId	rating	weightedRating
0	0.961678	12325	1	3.5	3.365874
1	0.961678	12325	2	1.5	1.442517
2	0.961678	12325	3	3.0	2.885035
3	0.961678	12325	5	0.5	0.480839
4	0.961678	12325	6	2.5	2.404196

Now, we'll apply a sum to the `topUsers` after grouping it up by `userId`.

```
tempTopUsersRating = topUsersRating.groupby
('movieId').sum()[['similarityIndex','weightedRating']]
tempTopUsersRating.columns = ['sum_similarityIndex',
'sum_weightedRating']
tempTopUsersRating.head()
```

	sum_similarityIndex	sum_weightedRating
movieId		
1	38.376281	140.800834
2	38.376281	96.656745

	sum_similarityIndex	sum_weightedRating
3	10.253981	27.254477
4	0.929294	2.787882
5	11.723262	27.151751

```
#Creates an empty dataframe
recommendation_df = pd.DataFrame ()
#Now we take the weighted average
```

```
recommendation_df[ 'weighted average recommendation score' ]
    = tempTopUsersRating[ 'sum_weightedRating' ]/
      tempTopUsersRating[ 'sum_similarityIndex' ]
recommendation_df[ 'movieId' ] = tempTopUsersRating.index
recommendation_df.head()
```

movieId	weighted average recommendation score	movieId
1	3.668955	1
2	2.518658	2
3	2.657941	3
4	3.000000	4
5	2.316058	5

Now let's sort it and see the top 20 movies that the algorithm recommended!

```
recommendation_df =
recommendation_df.sort_values(by='weighted average
recommendation score', ascending=False)
recommendation_df.head(10)
```

movieId	weighted average recommendation score	movieId
5073	5.0	5073
3329	5.0	3329
2284	5.0	2284
26801	5.0	26801
6776	5.0	6776
6672	5.0	6672
3759	5.0	3759
3769	5.0	3769

	weighted average recommendation score	movieId
3775	5.0	3775
90531	5.0	90531

```
movies_df.loc[movies_df['movieId'].isin(
    recommendation_df.head(10)['movieId'].tolist())]
```

	movieId	title	year
2200	2284	Bandit Queen	1994
3243	3329	Year My Voice Broke, The	1987
3669	3759	Fun and Fancy Free	1947
3679	3769	Thunderbolt and Lightfoot	1974
3685	3775	Make Mine Music	1946
4978	5073	Son's Room, The (Stanza del figlio, La)	2001
6563	6672	War Photographer	2001
6667	6776	Lagaan: Once Upon a Time in India	2001
9064	26801	Dragon Inn (Sun lung moon hak chan)	1992
18106	90531	Shame	2011

Advantages & Disadvantages of Collaborative Filtering

Advantages

- Takes other user's ratings into consideration
- Doesn't need to study or extract information from the recommended item
- Adapts to the user's interests which might change over time

Disadvantages

- Approximation function can be slow
- There might be a low number of users to approximate
- Privacy issues when trying to learn the user's preferences

References

- Asuncion, A., & Newman, D. J. (2007). *UCI Machine Learning Repository*. Retrieved from Center for Machine Learning and Intelligent Systems: <http://www.ics.uci.edu/~mlearn/MLRepository.html>
- Chen, J. (2020, November 11). Data Analysis Project — Telco Customer Churn. *Towards Data Science*. Retrieved from <https://towardsdatascience.com/data-analysis-project-telco-customer-churn-fe5c0144e708>
- Eddie. (2021, May 19). *Dealing With Missing Values in Python – A Complete Guide*. Retrieved from Analytics Vidhya: <https://www.analyticsvidhya.com/blog/2021/05/dealing-with-missing-values-in-python-a-complete-guide/>
- Haghghi, S., Jasemi, M., Hessab, S., & Zolanvari, A. (2018, May). Multiclass confusion matrix library in Python. *The Journal of Open Source Software* 3(25):729, 3(25), 729. doi: DOI: 10.21105/joss.00729
- Hunter, J., Dale, D., Firing, E., & Droettboom, M. (2012). *Creating multiple subplots using plt.subplots*. Retrieved from matplotlib.org: https://matplotlib.org/stable/gallery/subplots_axes_and_figures/subplots_demo.html
- IBM. (2022, Jan 30). *IBMDveloperSkillsNetwork-DV0101EN-SkillsNetwork*. Retrieved from IBM Cloud: <https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDveloperSkillsNetwork-DV0101EN-SkillsNetwork/labs/Module%203/images/Mod3Fig4SplitApplyCombine.png>
- IBM. (2022, Feb 04). *IBMDveloperSkillsNetwork-DV0101EN-SkillsNetwork*. Retrieved from IBM Cloud: <https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDveloperSkillsNetwork-DV0101EN-SkillsNetwork/labs/Module%203/images/Mod3Fig4SplitApplyCombine.png>

data.s3.us.cloud-object-storage.appdomain.cloud/IBMDveloperSkillsNetwork-DV0101EN-SkillsNetwork/labs/Module%203/images/boxplot_complete.png

Konstan., F. M. (2015). The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(4), 19:1–19:19. doi:<https://doi.org/10.1145/2827872>

Kotian, I. (2021, May). Weather Station Clustering using DBSCAN. *Kaggle*. Retrieved from <https://www.kaggle.com/code/lykin22/weather-station-clustering-using-dbscan/notebook>

Maklin, C. (2018, December 31). Hierarchical Agglomerative Clustering Algorithm Example In Python. *Towards Data Science*, Web Version. Retrieved from <https://towardsdatascience.com/machine-learning-algorithms-part-12-hierarchical-agglomerative-clustering-example-in-python-1e18e0075019>

Mehta, A. (2019, January 19). *Frequentist vs Bayesian- Which Approach Should You Use?* Retrieved from Digital Vidya: <https://www.digitalvidya.com/blog/frequentist-vs-bayesian/>

Navlani, A. (2018, August 2). *KNN Classification Tutorial using Scikit-learn*. Retrieved from datacamp.com: <https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn>

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B. a., Blondel, M., . . . Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.

Pinela, C. (2017, November 5). *Recommender Systems — User-Based and Item-Based Collaborative Filtering*. Retrieved from Medium.com: <https://medium.com/@cfpinela/recommender->

systems-user-based-and-item-based-collaborative-filtering-5d5f375a127f

Schafer, J. (1997). *Analysis of multivariate incomplete data*. London: Chapman&Hall.

Strickland, J. (2020). *Data Science Applications using Python and R*. Lulu, Inc.

Strickland, J. (2020). *Data Science Applications using Python and R*. Lulu, Inc. Retrieved from
https://www.lulu.com/spotlight/strickland_jeffrey/

Trab, N. (2019, March 20). *Machine Learning: Polynomial Regression with Python*. Retrieved from Towards Data Science:
<https://towardsdatascience.com/machine-learning-polynomial-regression-with-python-5328e4e8a386>

Vsevolodovna, R. M. (2020, October 2020). Collaborative Filtering with Python and Spark. *ruslanmw*, Web . Retrieved from
<https://ruslanmv.com/blog/Collaborative-filtering-with-Python-and-Spark>

Whitaker, J. (2016). Plotting data on a map (Example Gallery). *matplotlib.org*. Retrieved from
<https://matplotlib.org/basemap/users/examples.html>

Index

A

- accuracy 79, 265, 310, 318, 330, 333, 340, 341, 345, 379
ANOVA 56
area chart 143, 144
area plot *See* area chart
artificial neural network xv
artist layer 125, 147, 149, 152
Artist object 190
Artists 193
automobile 3, 41, 368
automobile dataset *See* datasets
automobile manufacturer 368
automobile prices 26, 37, 38, 39, 40, 42
Axes Artist 191

B

- bar plot 135, 163, 165, 169
binning 24, 28
Boolean vector 122

C

- Canadian Immigration *See* immigration
Canadian weather stations *See* weather stations
car clusters 369
car price model 83
car prices 1, 48, 51, 59, 61, 63, 65, 73, 74, 85, 379
cars 3, 13, 25, 45, 51, 61, 72, 75, 88, 103, 276, 378, 379
categorical variables 30, 40, 163, 169, 315, 354
cell sample, human *See* datasets
China GDP *See* datasets
China's GDP 295
churn 326
classification xvi
classification model 313, 340, 341

coefficients	29, 66, 216, 266, 267, 276, 284, 286, 324
comma separated value.....	1
confusion matrix.....	323, 330, 331, 333, 341, 342, 343, 344
continuous variables	43
correlation	55, 56
COVID.....	xvii
cross-validation.....	92
customer churn.....	<i>See</i> churn
customer segmentation.....	<i>See</i> segmentation
D	
data analytics.....	xiv, xv, 61
data types	8, 9, 34, 337
dataframe	1
datasets	
automobile EDA	33, 61
automobile prices	8
car clusters.....	369
car import names	4
car imports.....	1, 2, 4, 7
cell samples.....	336
China GDP	295
churn dataset.....	326
customer segmentation.....	354
drug treatments.....	314
fuel consumption.....	259
Fuel Consumption Ratings.....	259, 271, 279
movies ratings.....	400
telecommunications customers	305
weather stations	387
dependent variable	7, 29, 62, 64, 276, 282, 289, 290, 323, 324
distribution plot.....	71, 73, 103
E	
entropy	313, 319, 320, 321
F	
fitted values	71, 73
forecast.....	63

frequency distribution	154, 159, 306
F-test score.....	56, 58
fuel consumption.....	13, 21, 259, 271, 279, 281
G	
genetic algorithms	xv
Genres	402
GitHub	xvii, xviii, 33, 61, 87, 88, 138, 174, 227, 259, 271, 279, 314, 326, 336, 353, 368, 387, 411
graphic object	
Axes.....	189
Figure	188
H	
hierarchical clustering	367, 368, 372, 376, 377
histogram.....	25, 28, 135, 154, 157, 306
hyperparameter-tuning.....	90
I	
immigration....	112, 127, 129, 130, 132, 133, 150, 151, 152, 153, 154, 157, 161, 162, 167, 169, 177, 199, 200, 204, 210, 213, 214, 216, 218, 219, 221, 224, 225, 234, 241, 245, 247, 248, 256, 257
independent variable.....	62, 64, 70, 133, 276, 282
intercept.....	62, 63, 64, 66, 217, 267, 276, 284, 286, 323
J	
Jaccard index.....	330
K	
kernel functions	339
K-Folds cross validation	103
k-nearest neighbor	308
k-nearest neighbors	<i>See KNN</i>
KNN	303, 307, 310
L	
line plot.....	126, 127, 130, 131, 133, 135, 142, 204, 214
linear model.....	63, 64, 66, 68, 70, 71, 74, 75, 79, 266, 289, 295
linear regression.....	63

logistic function	297, 298, 324
logistic regression.....	xix, 269, 323, 324, 325, 328, 329, 334

M

machine learning.....	xiv, xvii, 60, 102, 306, 313, 329
MAE.....	270, 285, 301
malignant cells	335, 336, 337, 339
maximum value.....	10, 23, 26, 43, 366
Mean Absolute Error	<i>See</i> MAE
mean squared error	<i>See</i> MSE
member variables	190
minimum value	10, 26, 43
missing data	<i>See</i> missing values
missing values	1, 7, 13, 14, 17, 18, 20, 47
mitoses	
MLR.....	84
model diagnostics	70
movie ratings	401, 404, 405, 408, 411
MSE	79, 80, 81, 82, 83, 84, 268, 269, 270, 276, 285, 286, 287, 301
multiple linear regression	<i>See</i> MLR

N

NaN	11, 13, 19, 43, 143
non-linear regression.....	289
normalization	20, 23, 24, 109, 330, 331, 343, 354
Normalization.....	23, 330, 343, 354, 370
normalized 2, 3, 5, 6, 7, 8, 9, 12, 15, 17, 18, 23, 24, 34, 43, 52, 67, 105, 221, 222, 223, 331, 344	
Not a Number.....	<i>See</i> NaN

O

object containers.....	188, 189, 191, 194, 195, 196
object primitives.....	188, 189, 191, 192, 195
OLS	276
ordinary least squares	<i>See</i> OLS
overfitting.....	25, 90, 94, 98, 308, 329

P

Pearson correlation.....	36, 51, 53, 54, 55, 56, 415, 419, 420, 422
--------------------------	--

pearson correlation dictionary.....	420
Pearson Correlation Coefficient	53, 54, 55, 56, 419
percentile.....	10, 201
pie chart	181
pipeline.....	77, 78
polynomial fit.....	81, 84
polynomial model	74, 75, 76, 96, 287
polynomial regression.....	73, 76, 96, 98, 282, 283, 285, 286, 287
prediction 18, 60, 61, 72, 78, 81, 82, 87, 90, 94, 96, 98, 106, 277, 299, 304, 310, 332, 341	
predictor variables	62, 65, 66, 73
p-value	52, 53, 54, 55, 56, 57
python functions	
accuracy_score	309, 310
AgglomerativeClustering.....	363, 376
arange.....	49, 82, 89, 189, 235, 239, 284, 289, 293, 297
array.....	24, 106, 244
asarray.....	328, 338, 390
astype.....	18, 25, 78, 219, 232, 256, 307, 327, 330, 339, 343, 358
boxplot.....	41, 42, 187, 197, 200, 201, 202, 203, 206, 208, 212
columns.....	6, 8, 22, 24, 44, 47, 114, 115, 121, 124, 140, 175, 215, 219, 229, 306, 421, 423
confusion_matrix (cm)	330
confusion_matrix.....	331, 344
corr	35, 37, 39, 51, 69, 70
cross_val_predict.....	93
cross_val_score	92, 93, 105
DataFrame	24, 211, 214, 219, 231, 249, 256, 404, 416, 421, 424
dataframe.dropna()	7, 20, 370
dataframe.head4, 6, 14, 20, 24, 27, 33, 45, 57, 61, 87, 113, 119, 127, 133, 139, 150, 154, 164, 178, 187, 210, 215, 219, 228, 246, 249, 272, 280, 295, 305, 336, 354, 369, 376, 388, 401, 402, 403, 408, 413, 414, 421, 424	
dataframe.tail	4
DBSCAN.....	383, 392, 395
dendrogram.....	366, 373, 375
describe	10, 11, 43, 117, 198, 211, 260
drop	31, 90, 116, 139, 175, 229, 354, 403, 405, 406, 408, 414, 417
dtypes.....	9, 34, 338
euclidean_distances	371, 373
f_oneway	58

fcluster	372
featureNames	319
fit	92, 94, 102, 105, 108, 109, 217, 267, 269, 276, 278, 284, 286, 307, 309, 315, 328, 329, 340, 350, 352, 355, 376, 385, 395
fit_transform	77, 89, 97, 99, 101, 105, 283, 286, 355, 370, 382, 392, 395
gca	74, 235, 236, 239
get_group	58, 418
GridSearchCV	108
groupby	46, 48, 57, 178, 356, 378, 418, 423
grouped_pivot	47, 49
histogram	154, 156, 159, 160, 162, 261
iloc	119, 121
index	45, 49, 114, 119, 128, 134, 143, 150, 152, 181, 184, 214, 222, 238, 249, 377, 379, 421, 424
isin	405, 409, 417, 420, 425
isinstance	140
isnull	14, 15, 20, 27, 44, 45, 117
KMeans	350, 352, 355, 385
LabelEncoder	315
Line2D	188, 189, 191, 192, 193, 196
LinearRegression	63, 64, 67, 79, 82, 92, 94, 97, 100, 267, 269, 276, 278, 284, 286
linkage	366, 367, 372
linspace	26, 74, 300, 351, 376, 384, 393
loc	119, 121, 130, 157, 164, 187, 199, 211, 231, 256, 379, 409, 425
Logistic Collaborative Filtering Regression	329, 334
mean	18, 48, 93, 105, 177, 269, 277, 285, 301, 356, 393, 396
merge	405, 417, 422
minimumSamples	383
model_selection	91, 93, 105, 108, 308, 316, 328, 335
nan_to_num	355, 392, 395
normalize	24, 92, 94, 330, 331, 343, 344
notnull	14, 388
optimize	300
pearsonCorrelationDict	421
pearsonr	53, 54, 55
pie chart	179, 180, 182, 184
pipeline	77
plot	299, 300, 310, 337, 390
PollyPlot	89, 97, 100

<code>polyfit</code>	75, 216
<code>PolynomialFeatures</code>	77, 96, 97, 99, 100, 101, 105, 282, 283, 285, 286
<code>power</code>	292, 294
<code>predict</code>	63, 72, 78, 79, 81, 89, 94, 97, 269, 270, 277, 285, 309, 310, 318, 329, 340, 345
<code>predict_proba</code>	329, 334
<code>print</code>	4, 15, 24, 53, 75, 93, 105, 122, 133, 154, 174, 192, 228, 233, 267, 284, 301, 311, 318, 330, 339, 344, 365, 373, 393	
<code>r2_score</code>	81, 269, 270, 286
<code>random</code>	193, 266, 275, 282, 289, 291, 292, 301, 348
<code>random_state</code>	91, 92, 97, 100, 108, 308, 316, 328, 339
<code>rcParams</code>	..	128, 129, 134, 143, 145, 150, 152, 155, 162, 166, 176, 230, 262, 389, 393, 396
<code>read_csv</code>	2, 8, 33, 61, 87, 260, 272, 280, 295, 305, 314, 326, 336, 354, 369, 388, 400, 412	
<code>read_excel</code>	113, 138, 173, 227
<code>regplot</code>	37, 39, 67, 250, 252, 254, 257
<code>rename</code>	139, 175, 229
<code>replace</code>	7, 13, 18, 20
<code>reset_index</code>	20, 119, 213, 215, 219, 222, 249, 256, 370, 388, 406
<code>residplot</code>	70
<code>Ridge</code>	105, 106, 108, 109
<code>scatter</code>	215, 349, 357, 358, 362, 364, 377, 379, 385, 393, 396
<code>score</code>	80, 92, 98, 106, 108
<code>set_postfix</code>	106
<code>shape</code>	..	49, 77, 91, 115, 124, 139, 141, 174, 228, 273, 310, 314, 317, 328, 331, 339, 343, 364, 369, 408
<code>StandardScaler</code>	77, 79, 328, 355, 382, 392, 395
<code>tolist</code>	15, 115, 390, 405, 417, 420, 425
<code>train_test_split</code>	91, 92, 97, 308, 316, 328, 339
<code>transform</code>	285, 307, 315, 328
<code>unique</code>	46, 319
<code>value_counts</code>	305

R

<code>read_csv</code>	<i>See python functions</i>
regression line	...36, 37, 39, 62, 63, 67, 68, 70, 79, 216, 217, 248, 250, 256, 257, 269	
regression plot	67, 68
residual plot	70, 71

residual sum of squares.....	266, 277
response variable.....	62, 63, 65, 66, 79
Ridge regression.....	105, 108
RSE	285
R-squared	79, 82, 83, 84, 85, 268
S	
scatterplot.....	145, 215, 216, 218, 226, 263, 266, 268, 274, 275, 276, 281, 285, 357
scripting layer.....	125, 147, 150, 162, 165, 169, 204
<i>Seaborn</i>	34, 248, 249
segmentation	347, 353
sigmoid	294, 295, 298, 299, 300, 301, 324
simple linear model.....	80
simple linear regression.....	63
SSE.....	276
sum of squared errors.....	<i>See</i> SSE
Support Vector Machines.....	<i>See</i> SVM
SVM	335, 339, 344
T	
target variable	49, 60, 104, 276, 316
telecommunications.....	304, 323, 325, 326
telecommunications dataset	<i>See</i> datasets
text analytics	xv
train/test split	265, 274, 282, 308
transparency value.....	150, 152, 162
U	
used cars	4
V	
validation set.....	91
variance.....	xvi, 23, 68, 70, 77, 104, 277, 307, 376, 382
vertical bar graph.....	164
W	
waffle chart.....	233, 237
weather forecasting	389

weather stations	386
word cloud	241, 242, 243, 245, 247, 248
wrangling.....	13, 111, 137, 173, 227

