SCMA248 Introduction to Data Science

Pairote Satiracoo

July 10, 2022

Contents

-1 Course Syllabus

Program Bachelor of Science Program in Mathematics Degree ☑ Bachelor ☐ Master ☐ Doctoral
Course Title Introduction to Data Science Faculty of Science
Course Code SCMA 248 Department of Mathematics

Course Specification

Section 1 General Information

1. Course Code and Title

In Thai วทคณ ๒๔๘ การแนะนำวิทยาการข้อมูล

In English SCMA 248 Introduction to Data Science

2. Number of Credits 3 (3-0-6) (Theory 3 hours Practice 0 hours Self-study 6 hours/week)

3. Curriculum and Course Type

3.1 Program of Study Bachelor of Science Program in Mathematics

3.2 Course Type Specific Courses

4. Course Coordinator and Instructor

4.1 Course Coordinator

1) Pairote Satiracoo, Ph.D.

Department of Mathematics, Faculty of Science, Mahidol University

Contact: 02-201-5340 E-mail : Pairote.sat@mahidol.edu

4.2 Instructor Pairote Satiracoo, Ph.D.

5. Semester/Class Level

5.1 Semester 2 (2021) / Class Level 2nd Year

5.2 Number of Students Allowed Approximately 40 Students

6. Pre-requisite none

7. Co-requisites none

8. Study Site Location Faculty of Science, Mahidol university, Phaya thai campus

9. Date of Preparation/Latest Revision of the Course Specifications December 5, 2021

Section 2 Aims and Objectives

1. Course Goals

This course will provide a broad introduction to four key aspects of data science: data retrieval and manipulation, data visualization, statistical computation and machine learning, and presentation and communication. Students will use data from a variety of sources, be introduced to contemporary computing and database environments such as R and SQL, and be exposed to case studies from outside the classroom. Through this unit, students will become acquainted with the challenges of contemporary data science and gain an appreciation of the foundational skills necessary to turn data into information. Finally, students will have the opportunity to practice a presentation about data science group projects.

2. Objectives of development/revision

2.1 Course Objectives

The instructor expects students to acquire skills and knowledge as follows. Students should:

1) Understand the concepts of data science.

2) Use computer programming and tools for data science purposes.

3) Search for the information and use it to solve the assigned problems.

2.2 Course-level Learning Outcomes (CLOs)

After successful completion of this course, students should be able to:

1) CLO1: Comply with the regulations of the curriculum and the university. Have academic and professional ethics, do not copy or bring the work of others to be their own.

2) CLO2: Use the appropriate Statistical and computational techniques to solve problem about interpret meaning of information from the given big data.

3) CLO3: Use computer programming concepts or computer tools to solve data science problems

4) CLO4: Use appropriate information technologies to search for data and use it to solve assigned problems.

Section 3 Course Description and Implementation

1. Course Description

(In Thai) การแนะนำสู่แง่มุมสำคัญของวิทยาการข้อมูล การดึงข้อมูล และการจัดการข้อมูล การแสดงข้อมูล การคำนวณเชิงสถิติ การเรียนรู้ด้วยเครื่อง การนำเสนอและการสื่อสาร การคำนวณร่วมสมัย สิ่งแวดล้อมด้านฐานข้อมูล เช่น อาร์ และ เอสคิวแอล กรณีศึกษาจากนอกห้องเรียน ทักษะพื้นฐานสำคัญสำหรับ การเปลี่ยนข้อมูล เป็นความรู้ การฝึกทักษะการสืบค้นข้อมูล เพื่อทำงานกลุ่มและนำเสนอในห้องเรียน

(In English) An introduction to key aspects of data science: data retrieval and manipulation, data visualization, statistical computation and machine learning, presentation and communication; an introduction to contemporary computing and database environments such as R and SQL; case studies from outside the classroom; foundational skills necessary to turn data into information; practicing of information searching skill for working on group assignments and doing presentation in the classroom

2. Number of hours per semester

Theory (hours)	**Practice** (hours)	**Self-study** (hours)
45 hours/semester (3 hours x 15 weeks)	None	90 hours/semester (6 hours x 15 weeks)

3. Number of Hours per Week for Individual Advice

Instructors provide academic counseling and guidance to individual at least 1 hour/week or upon request during office hours (Monday-Friday).

Section 4 Development of the Expected Learning Outcomes

1. **A brief summary of the knowledge or skills expected to develop in students; the course-level expected learning outcomes (CLOs)**

   By the end of the course, students who successfully complete the course will be able to:

   1) CLO1: Comply with the regulations of the curriculum and the university. Have academic and professional ethics, do not copy or bring the work of others to be their own.

   2) CLO2: Use the appropriate Statistical and computational techniques to solve problem about interpret meaning of information from the given big data.

   3) CLO3: Use computer programming concepts or computer tools to solve data science problems

   4) CLO4: Use appropriate information technologies to search for data and use it to solve assigned problems.

2. **How to organize learning experiences to develop the knowledge or skills stated in number 1 and how to measure the learning outcomes**

Course Code	Teaching strategies				Learning outcomes measurements	
	Interactive lecture	Effective questioning	problem solving activities	Brainstorm	Individual assignment	Written exam
CLO1		✓		✓	✓	
CLO2	✓	✓	✓	✓	✓	✓
CLO3	✓	✓	✓	✓	✓	✓
CLO4		✓	✓	✓	✓	✓

Section 5 Lesson Plan and Evaluation

Program Bachelor of Science Program in Mathematics Degree ☑ Bachelor ☐ Master ☐ Doctoral
Course Title Introduction to Data Science Faculty of Science
Course Code SCMA 248 Department of Mathematics

1. Lesson Plan

Week	Topic/Details	Number of hours		Teaching activities/ media	Instructors
		Class-room ses-sions	Practice sessions		
1	Introduction to data science and tools	3	0	**Teaching method**: Interactive lecture, effective questioning, formative assessment, problem solving, problem based activities **Media**: lecture notes, slides, individual assignments	Dr. Meechoke
2-3	Data retrieval and manipulation	3	0		Dr. Meechoke
4-5	statistical computation and machine learning	3	0		Dr. Meechoke
6	Data presentation and Communication	3	0		Dr.Tanapon
7-8	Data visualization	3	0		Dr.Tanapon
9	Midterm examination				
10-11	Introduction to contemporary computing and database environments such as R and SQL	3	0	**Teaching method**: Interactive lecture, effective questioning, formative assessment, problem solving, problem based activities **Media**: lecture notes, slides, individual assignments	Dr.Tanapon
12	Case studies from outside the classroom	3	0		Dr.Tanapon
13	foundational skills necessary to turn data into information	3	0		Dr.Tanapon
14	practicing of information searching skill for working on group assignments	3	0		Dr.Tanapon
15-16	Group Presentation	3	0		Dr.Tanapon
17	Final examination				
	Total	45	0		

2. Evaluation of the CLOs

2.1 Measurement and Evaluation of learning achievement

a. Formative assessment

During a lesson, instructor keeps the question going and monitors students' progress in general. There are also quick quizzes to check the current understanding of individual students.

b. Summative assessment

(1) Tool and weight for measurement and evaluation

Learning Outcomes	Measurement Method			Weight (Percentage)
	Individual assignment	Project	Written exam	
CLO1: Comply with the regulations of the curriculum and the university. Have academic and professional ethics, do not copy or bring the work of others to be their own.	5	5		10
CLO2: Use the appropriate Statistical and computational techniques to solve problem about interpret meaning of information from the given big data.		5	20	25
CLO3: Use computer programming concepts or computer tools to solve data science problems.	2	3	20	25
CLO4: Use appropriate information technologies to search for data and use it to solve assigned problems	8	12	20	40
Total	**15**	**25**	**60**	**100**

(2) Measurement and evaluation

Learning Outcomes	Measurement Method			Weight (Percentage)
	Individual assignment	Project	Written exam	
CLO1: Comply with the regulations of the curriculum and the university. Have academic and professional ethics, do not copy or bring the work of others to be their own.	5%	5%		10%
CLO2: Use the appropriate Statistical and computational techniques to solve problem about interpret meaning of information from the given big data.		5%	20%	25%
CLO3: Use computer programming concepts or computer tools to solve data science problems.	2%	3%	20%	25%
CLO4: Use appropriate information technologies to search for data and use it to solve assigned problems	8%	12%	20%	40%
Total	**15%**	**25%**	**60%**	**100%**

Students are evaluated their performance using assessment rubric according to course objectives and learning outcomes. Rubric scores for a single piece of individual assignment

Score	Description
5	Demonstrates the required work for all questions.
4	Demonstrates the required work for most questions with lower than 25% mistakes.
3	Demonstrates the required work for many questions with lower than 50% mistakes.
2	Demonstrates the required work for some questions with more than 50% mistakes.
1	Demonstrates the required work for few questions with more than 75% mistakes.
0	No response

The percentage of individual work is the average rubric scores of all pieces of individual work.

Grading system

After completion of the evaluation process each student is assigned a criterion-referenced grade (as shown in the table below). Evaluation and achievement will be justifying according to Faculty and University code, conducted by grading system of A, B+, B, C+, C, D+, D and F. To pass this course, student must earn a grade of at least D.

Total percentage of evaluation	Grade
80 – 100	A
75 – 79	B+
70 - 74	B
65 – 69	C+
60 – 64	C
55 – 59	D+
50 – 54	D
0 – 49	F

(3) Re-exam (if any)

None

3. Students' Appeal

Students may submit formal complaint or academic appeal directly to

International Education And Administration Unit, Division of Salaya Campus

Room SC1-116, SC1-Building, Faculty of Science (Salaya Campus), Mahidol University

999 Phuttamonthon 4 Road, A. Phuttamonthon, Nakhon Pathom 73170, Thailand

E-mail: scsim@mahidol.ac.th; Phone: + 66 2 4419820 ext. 1199.

If it is considered that a case exists, the matter will be investigated in accordance with the procedures, and the complainant informed of the outcome.

Section 6 Teaching Resources

1. Required Texts

1) Beginning data science in R [electronic resource]: data analysis, visualization, and modelling for the data scientist / Thomas Mailund, Berkeley, CA : Apress, c2017.

2) Python data science essentials: become an efficient data science practitioner by thoroughly understanding the key concepts of Python/ Alberto Boschetti, Luca Massaron, Birmingham : Packt, c2015

2. Suggested Materials

1) Handouts
2) PowerPoint presentations

3. Other Resources (if any)

1) Available through MU Library-subscribed databases
2) https://www.edx.org
3) https://www.coursera.org

Section 7 Evaluation and Improvement of Course Implementation

1. Strategy for Course Effectiveness Evaluation by Students

1.1. Evaluation of instructors

1.2. Course evaluation

2. Strategy for Teaching Evaluation

2.1. Students evaluation and students record

2.2. Instructors evaluation

3. Teaching Improvement

Study for more information e.g., study the further education requirement, brainstorming, and criticizing course.

4. Verification of Standard of Learning Outcome for the Course

Analysis of students learning outcomes using scores from class attendance, class attendance and assignment.

5. Revision Process and Improvement Plan for Course Effectiveness

After obtain the course evaluation, students' needs and comments will be developed by course responsible faculty member and instructors.

Appendix

Relations between the course and the program

Table 1 Relations between the course and the PLOs

Database Management	PLOs							
	PLO1	PLO2	PLO3	PLO4	PLO5	PLO6	PLO7	PLO8
(Course Code) SCIM 231		I	R					

Table 2 Relations between CLOs and PLOs

(Course Code) SCIM 231	PLOs				
	PLO1	PLO2	PLO3	PLO4	PLO5
CLO1: Comply with the regulations of the curriculum and the university. Have academic and professional ethics, do not copy or bring the work of others to be their own.		2.1			
CLO2: Use the appropriate Statistical and computational techniques to solve problem about interpret meaning of information from the given big data.		2.3	3.2 3.3		
CLO3: Use computer programming concepts or computer tools to solve data science problems.		2.3	3.2 3.3		
CLO4: Use appropriate information		2.1			

technologies to search for data and use it to solve assigned problems					

Table 3 PLOs that the course is responsible for

PLOs	SubPLOs
PLO2: Apply data analysis and data science strategy for trend prediction to support making data driven decision with the regard to data privacy, ethics, and protection..	2.1 Collect quantitative and qualitative data related to thesituation by using of information technology and ethics.
	2.2 Determine the domain of the problem that needs to be transformed into an appropriate mathematical model.
	2.3 Use the appropriate Statistical and computational techniques to solve problem about interpret meaning of information from the given big data.
PLO3: Create an independent project in industrial mathematics and data science based on related-concepts with professional code of conduct.	3.1 Describe the structure of input and output data, statements and conditions in algorithms.
	3.2 Analyze and define the scope of the problem. Design a sequence of statements in the program that fit the gien problem.
	3.3 Use appropriate computer programming language and statements to make the processing according to the designed results.
	3.4 Test the integrity of the output from the written computer program and modify the program codes for the better results.

0 Welcome to SCMA248 Introduction to Data Science

Welcome to SCMA248 Introduction to Data Science

Print to PDF

This course provides a comprehensive introduction to four key aspects of data science: data retrieval and manipulation, data visualization, statistical computing and machine learning, and presentation and communication. Students will use data from a variety of sources, be introduced to contemporary computing and database environments such as R and SQL, and be exposed to case studies from outside the classroom. Through this unit, students will become familiar with the challenges of contemporary data science and gain an understanding of the fundamental skills needed to turn data into information. Finally, students will have the opportunity to practice a presentation on group data science projects.

The class notes for this course can be found from my Github repository: https://github.com/pairote-sat/SCMA248.

In this course, you will learn:

- an introduction to key aspects of data science
- data retrieval and manipulation,
- data visualization,
- statistical computation and machine learning,
- presentation and communication
- an introduction to contemporary computing and database environments such as R and SQL.

Next
1. Introduction to Data Science ›

By Pairote Satiracoo
© Copyright 2021.

1 Introduction to Data Science

1. Introduction to Data Science

> "Learning from data is virtually universally useful. Master it and you will be welcomed anywhere." — John Elder, Elder Research

Image of John Elder

1.1. What is data science?

https://www.analytixlabs.co.in/blog/data-science-process/

Data Science is an associated field of Big Data that focuses on analyzing enormous amounts of complicated and raw data to provide valuable information to businesses. In order to analyze and present data for effective decision making by executives, various subjects such as statistics, mathematics, and computation are combined. Data Science enables businesses to improve performance, productivity, and customer satisfaction while making it easier to meet their financial goals. However, a thorough understanding of the Data Science process is essential for Data Scientists to successfully apply Data Science and deliver helpful, productive results. The many stages of the Data Science process help transform data into useful results. It helps in more efficient analysis, extraction, visualization, storage, and management of data.

https://ischoolonline.berkeley.edu/data-science/what-is-data-science/

Data Science continues to be one of the most promising and sought-after career paths for qualified individuals. Today's effective data scientists recognize that they must go beyond the traditional skills of large-scale data analysis, data mining, and programming. Data Scientists must be proficient across the spectrum of the data science lifecycle and possess a level of flexibility and awareness to maximize returns at each stage of the process to produce meaningful insights for their organizations.

The term "data scientist" was introduced in 2008 when companies recognized the need for data experts capable of organizing and analyzing massive amounts of data.

https://www.simplilearn.com/a-day-in-the-life-of-a-data-scientist-article

A Data Scientist is a specialist in statistics, data science, Big Data, R programming, Python, and SAS, and a job as a Data Scientist offers many opportunities and a high income. According to Harvard Business Review, Data Science is the most attractive job of the twenty-first century. Data Scientist has been named the best job in the United States by Glassdoor, with a job rating of 4.8 out of 5 and a satisfaction rate of 4.2 out of 5. The average base salary is $110,000, and there are currently hundreds of openings, with many more to come: By 2020, IBM predicts a 28 percent increase in demand for Data Scientists.

https://medium.datadriveninvestor.com/data-science-in-3-minutes-data-science-for-beginners-what-is-data-science-f4632bee9881

https://www.youtube.com/watch?v=X3paOmcrTjQ

1.2. A day in the life of a data scientist

Let us take a look at how a day in the life of a data scientist goes while working on a data science project.

1.2.1. Business problem

First and foremost, it's critical to understand the business problem. In meetings with clients, the data scientist asks relevant questions, understands, and defines objectives for the problem that needs to be solved.

Contents

Print to PDF

https://www.analytixlabs.co.in/blog/data-science-process/

Asking questions like these can help you get through this process.

- Who are the customers?
- How can they be identified?
- How is the sales process going right now?
- Why are they interested in your products?
- What products are they interested in?

To turn numbers into insights, You need much more context to the numbers to understand them. By the end of this phase, you should have as much information as possible.

1.2.2. Data acquisition

In the next step, the data scientist prepares for data acquisition, gathering and scraping data from multiple sources, including web servers, databases, APIs, and online archives. It appears that collecting the proper data takes both time and work.

1.2.3. Data preparation

Once the data has been collected, the next phase is data preparation, which involves data cleaning and transformation. The most time-consuming step is data cleaning, which deals with a variety of challenging conditions. The data scientist is responsible for data types that are not consistent. In data transformation, the data scientist modifies the data based on defined mapping rules. The project uses extract, transform, and load (ETL) tools such as Talend and Informatica to perform complex transformations that help the team better understand the data structure.

Below are the most common mistakes to watch out for:

- Values that are missing
- Values that have been manipulated, such as invalid entries
- Differences in time zones
- Errors related to the date range, such as a transaction that was recorded before the sale even started, are common.

You also need to look at the aggregate of all rows and columns in the file to determine if the results are correct. If they are not, you will need to delete or change the incorrect data.

1.2.4. Exploratory data analysis

It is very important to understand what you can do with your data. The data scientist performs exploratory data analysis (EDA). Using EDA, the data scientist defines and refines the selection of feature variables to be used in model development.

1.2.5. Data modeling

This step is where your knowledge of math, statistics, and technology comes in handy. To properly analyze the data and gain all possible insights, you will need to use all available data science tools. You may need to create a predictive model that compares your average customer to customers who are less successful. Several factors, such as age or social media engagement, could prove to be crucial elements in your research to predict who will buy a service or product.

Python is preferred for modeling the data but it can also be done using R and SAS.

1.2.6. Visualization and communication

After all these processes, it is crucial that you share your findings and insights with the sales manager and convince him of their importance. It is to your advantage to communicate well in order to accomplish the task assigned to you. Clear and concise communication will lead to action. Improper contact, on the other hand, can lead to inaction. Tools such as Tableau, Power Bi, and QlikView can be used to create powerful reports and dashboards.

You need to link the data you have collected and your insights to the sales manager's knowledge so that they can understand it better. Start by discussing why a product is not performing well and why certain demographics are not responding to the sales pitch. After you have laid out the problem, you can move on to the solution. You need to write a compelling story with clear goals and objectives.

> "If you can't explain it simply, you don't understand it well enough." — Albert Einstein

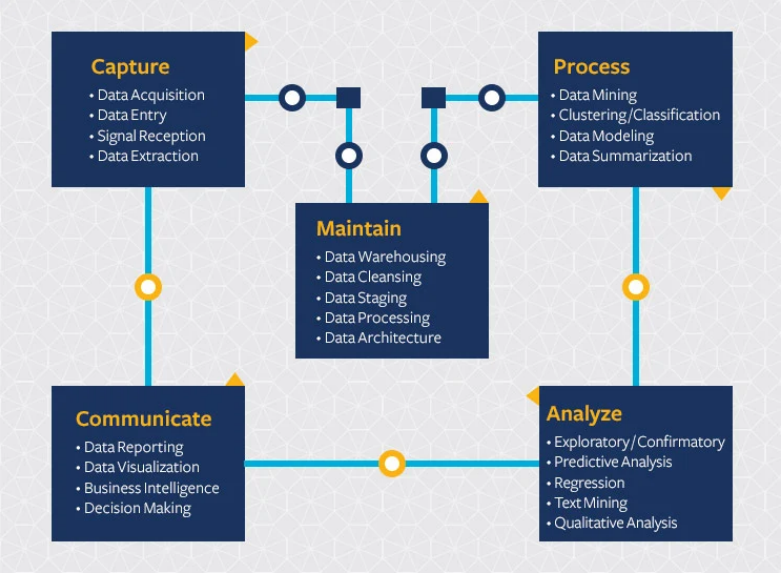
1.2.7. Deployment and maintainance

In the final phase, the data scientist deploys and maintains the model. The selected model is tested in a pre-production environment before being deployed in the production environment, which is the best practice.

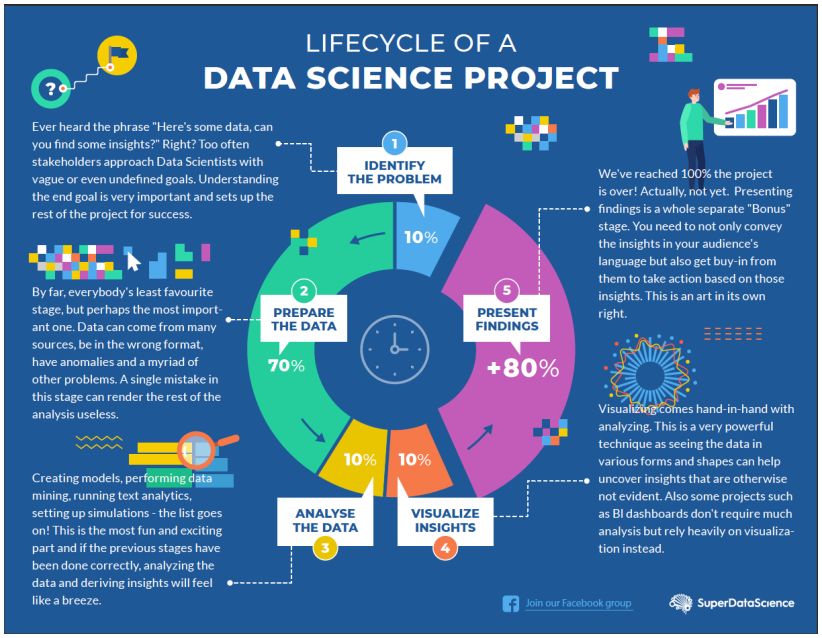
1.3. Data science life cycle

A Data Science lifecycle is a series of Data Science steps that you go through to complete a project or analysis. Because each Data Science project and team is unique, each Data Science lifecycle is also unique. However, most data science projects follow a similar generic data science life cycle.

The following diagram represents the five phases of the data science life cycle (from the School of Information, Berkeley: https://ischoolonline.berkeley.edu/data-science/what-is-data-science/):



Another useful infographic about the lifecycle of a data science project from SuperDataScience is given below (Data science life cycle: from SuperDataScience):



1.4. What is the difference between a data analyst, a data engineer, and a data scientist?

https://www.dataquest.io/blog/data-analyst-data-scientist-data-engineer/
https://www.edureka.co/blog/data-analyst-vs-data-engineer-vs-data-scientist/
https://www.simplilearn.com/tutorials/data-science-tutorial/data-scientist-vs-data-analyst-vs-data-engineer https://ischoolonline.berkeley.edu/data-science/what-is-data-science/

Data has always been crucial to any decision-making process. Today's world is entirely based on data, and no business could operate without data-driven strategic planning and decision making. Because of its invaluable insights and trust, data is used in a variety of professions in the industry today. In this section, we explore the important differences and similarities between a data analyst, a data engineer, and a data scientist.

Data Scientist

Data Scientists are concerned with what questions need to be answered and where to find the relevant data. They have analytical and business acumen, as well as the ability to extract, cleanse, and present data. They help companies find, organize, and analyze large amounts of unstructured data. Also, they use advanced data techniques to derive business insights, such as clustering, neural networks, decision trees, etc. The results are then summarized and distributed to key stakeholders to help the business make strategic decisions.

The following are examples of the work of data scientists:

- Evaluating statistical models to determine the validity of analyzes.
- Using machine learning to develop better predictive algorithms.
- Testing and continuously improving the accuracy of machine learning models.
- Creating data visualizations to summarize the results of advanced analysis.

Skills needed: Programming (SAS, R, Python), statistical and mathematical skills, storytelling and data visualization, Hadoop, SQL and machine learning are also required skills.

Data Analyst

In a data analytics team, data analysts are the entry-level position. They add value to their business by collecting data, analyzing it, and sharing the results to support business decisions. Cleaning data, performing analysis, and developing data visualizations are common tasks for data analysts.

The title of data analyst can vary by industry (e.g., business analyst, business intelligence analyst, operations analyst, database analyst). Regardless of the designation, the data analyst is a generalist who can work in a variety of roles and teams to help others make better data-driven decisions.

The specific skills required will vary depending on the needs of the business, but below are some general tasks:

- Cleaning and organizing unprocessed data.
- Using descriptive statistics to get an overview of the data
- Identifying and analyze significant data trends.
- Creating visual representations and dashboards to support data interpretation and decision making.
- Providing technical analysis results to business customers or internal teams.

Skills needed: Programming skills (SAS, R, Python), statistical and mathematical skills, data manipulation and data visualization are required.

Data Engineer

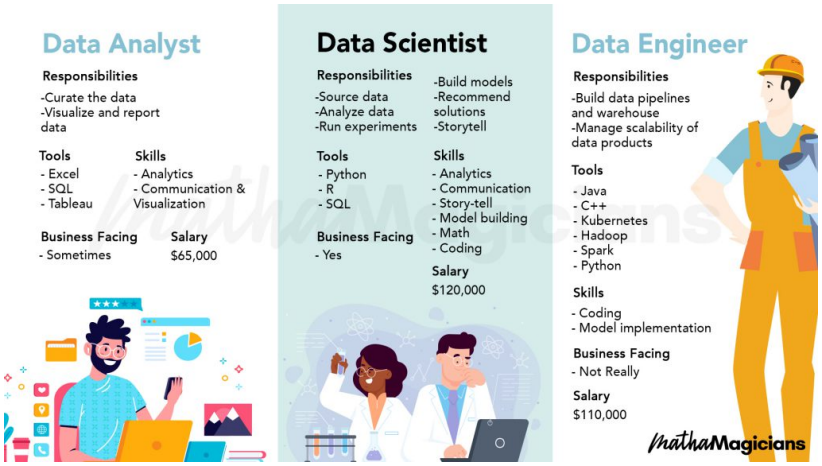
Data engineers are responsible for managing vast amounts of data that is constantly changing. They work to create data pipelines and infrastructures, deploy, manage, and optimise to transform and deliver data to data scientists and data analysts to perform their work.

The mindset of a data engineer is often more focused on building and optimizing. The responsibilities of a data engineer include, but are not limited to, the following:

- Creating APIs for data usage.
- Integrating external or new data sets into existing data pipelines.
- Applying feature transformations for machine learning models to new data.
- Continuous monitoring and testing of the system to ensure optimal performance.

Skills needed: Programming languages (Java, Scala), NoSQL databases (MongoDB, Cassandra DB) and frameworks are required (Apache Hadoop).

Summary of the various tools and skills required by a Data Analyst/Data Engineer/Data Scientist (from MathaMagicians, https://mathamagicians.co/differences-between-a-data-analyst-data-scientist-and-data-engineer/)



The 365 Data Science team created the What-Where-Who infographic in response to the need for a simple explanation of data science. They define the key data science processes and promote the field. Their viewpoint on data science is as follows. For more detail, please visit

https://365datascience.com/career-advice/career-guides/defining-data-science/ and https://www.youtube.com/watch?v=edZ_JYpOM8U.

1.5. The data in data science

There is always data before anything else. Data is the raw material on which all analysis is built, and it is the cornerstone of data science. In data science, there are two types of data: **traditional data** and **big Data**.

Traditional data is organized and stored in databases that analysts can access from a single computer, and it is in tabular form with numeric or text values. We use the term "traditional" only for clarity. It clarifies the distinction between Big Data and other forms of data.

Big Data, on the other hand, is much more extensive than traditional data. It comes from many different sources (numbers, text, but also photos, audio and mobile data).

Relational database management systems are used to store traditional data. Big data, on the other hand, is kept on multiple servers and is significantly more complex.

1.5.1. Where does data come from?

Traditional data sources include simple customer records and historical stock price data.

Big data, on the other hand, is all around us. Big data is being used and generated by more and more companies and industries. These include:

- Think of internet networks like Facebook, Google and LinkedIn, and data from financial transactions.
- Temperature measurement networks in different geographic regions and machine data from sensors in industrial equipment, are examples of Big Data.
- There is also wearable technology.

1.5.2. Where is data science applied in the world of data?

1.5.2.1. Where is Business Intelligence used?

Business intelligence is used in sales, marketing, finance and operations. Quantitative analysis, monitoring performance against business goals, gaining customer insights and sharing data to uncover new opportunities are just some of the tasks.

Analyst and data scientist:

https://www.netsuite.com/portal/resource/articles/business-strategy/business-intelligence-examples.shtml

Analysts are BI experts who use consolidated enterprise data along with powerful analytical tools to determine where improvements can be made and what strategic recommendations they can make to management.

Marketing:

Marketers can use business intelligence products to track campaign analytics from a central digital location. BI systems can track campaigns in real time, analyze their effectiveness, and plan future initiatives. This information provides marketing teams with a better understanding of overall performance, as well as contextual images to share with the rest of the organization.

Finance:

BI professionals can extract insights from financial data and combine it with operational, marketing, and sales data to make decisions and understand factors that affect profit and loss.

Sales:

For quick access to complicated information such as discount analysis, customer profitability, and customer lifetime value, sales data analysts and operations managers often use BI dashboards and key performance indicators (KPIs). Sales managers use dashboards with reports and data visualizations to track revenue goals, sales force performance, and the state of the sales pipeline.

Operations:

Managers can access and evaluate data such as supply chain analytics to find methods to optimize processes to save time and resources. Business intelligence can also help improve distribution channels and ensure compliance with service level agreements.

1.5.3. Where does data science apply?

The applications for the relevant methods are extremely diverse; data science is rapidly making its way into a wide range of businesses. However, four important areas should be included in the discussion.

Data Science and User Experience (UX) When a company launches a new product, it typically conducts surveys to determine how customers feel about it. After the BI team creates its dashboards, it must analyze the data by segmenting the observations (e.g., by region) and then analyzing each segment independently to derive useful prediction. The results of these operations often support the notion that in order to maximize consumer happiness,the product must have small but significant differences in each segment to maximize consumer satisfaction.

Sales Volume Prediction This form of analysis uses time series analysis. The data scientist wants to know what will happen in the next sales period or a year from now based on the sales data collected up to that point. They use mathematical and statistical models to run multiple simulations that present future scenarios to the analyst. This is the core of data science because based on these scenarios, the company can make better forecasts and apply appropriate strategies.

Fraud detection Banks can use machine learning, especially supervised learning, to collect historical data, categorize transactions as valid or fraudulent, and train fraud detection models. When these models detect even the slightest hint of theft, they flag the transactions and intervene in real time to prevent the fraud.

Customer Engagement Businesses can use machine learning algorithms to figure out which customers are likely to buy their products. This means that the store can efficiently offer discounts and a “personal touch,” which lowers marketing spend and increases revenue. Google and Amazon are two big names that come to mind.

1.6. Cases studies: Examples of data science applications

1. **Expedia:** Business intelligence improves customer experience

Expedia is the parent company of Expedia, Hotwire and TripAdvisor, all of which are leading travel companies.

- **The challenge:** Customer satisfaction is critical to the company’s purpose, strategy and long-term success. The online experience should reflect an enjoyable trip, but the company does not have access to the voice of the customer.
- **The solution:** the company was manually collecting tons of data, leaving little time for analysis. The customer satisfaction group was able to examine customer data from across the company using business intelligence and link the results to ten goals that were directly related to the company’s priorities. KPI managers create, monitor, and analyze data to identify trends or patterns.
- **The results:** The customer care team can monitor their performance against the KPIs in real time and make adjustments as needed. In addition, the data can be used by other departments. For example, a travel manager can use BI to identify large volumes of unsold tickets or offline bookings and develop

tactics to change behavior and increase overall savings.

1. **Lotte.com:** Business intelligence boosts revenue With 13 million customers, Lotte.com is the Korea's largest online shopping mall.

- **The challenge:** With more than 1 million page views daily, company executives wanted to know why customers abandon their shopping carts.
- **The solution:** Customer Experience Analytics, Korea's first online behavioral analytics system, was implemented by the assistant general manager of the marketing planning team. The manager used the information to better analyze customer behavior, conduct targeted marketing and redesign the website.
- **The results:** After one year, customer loyalty was up and sales were up $10 million, driven by insights from the new analytics program, BI. Adjustments resulted from identifying and correcting the causes of abandoned purchases, such as a lengthy checkout process and unexpected delivery times.

1.7. Use Cases: Examples of Data Science applications that prominent companies use

1. **Netflix:** With 148 million members, the Internet entertainment company has a significant BI advantage. What is Netflix's approach to business intelligence? Netflix uses data in a variety of ways. For example, the company develops and tests new programming concepts based on shows already watched. Business intelligence is also used by Netflix to encourage users to interact with its content. The service is so good at promoting targeted material that its recommendation system is responsible for over 80% of all streamed content.
2. **Tesla:** The forward-thinking automaker uses BI to wirelessly connect its vehicles to its corporate offices and collect data for research. This method connects the automaker to the customer and allows it to anticipate and fix problems such as component damage, traffic and hazard data. The result is high customer satisfaction and the selection of future improvements and goods is based on better information.
3. **Uber:** The company uses business intelligence to identify a variety of key components of its operations. One example is surge pricing. Algorithms continuously monitor traffic conditions, ride times, driver availability and customer demand, and adjust prices as demand increases and traffic conditions change. Airlines and hotel companies use real-time dynamic pricing to adjust costs to meet demand.

1.8. This video on "What is Data Science" from Simplilearn

This video on "What is Data Science" will give you an idea of what the life of a Data Scientist is like. This video on Data Science for Beginners also explains the process of a Data Science project, its many applications, and the positions and salaries of a Data Scientist.

```
from IPython.display import IFrame, YouTubeVideo, SVG, HTML
```

```
# https://www.youtube.com/watch?v=X3paOmcrTjQ
YouTubeVideo('X3paOmcrTjQ',640,360)
```

Previous
Welcome to SCMA248 Introduction to Data Science

Next
2. Python Basics

By Pairote Satiracoo
© Copyright 2021.

2 Python Basics

2. Python Basics

In this chapter, we get started with very useful Python basics.

2.1. Comments

A comment is a piece of text that is not executed within a program. It can be used to provide additional information to help with code comprehension.

A comment is started with the # character and continues until the end of the line.

```
# This is a comment line.
```

2.2. Whitespace Formatting

Curly brackets are used to separate code blocks in many languages. Indentation is used in Python: This makes Python code very readable, but it also implies that formatting must be done carefully.

```
for i in [1,2]:
    for j in [1,2]:
        print(i+j)
```

```
2
3
3
4
```

Inside parentheses and brackets, whitespace is ignored, which is useful for long-winded calculations:

```
my_list = [1, 2, 3]
my_list_of_lists = [[1,2],
                    [3,4]]
```

2.3. Python Data Types

In Python, every value is referred to as a “object.” And each object has its own data type. The following are the three most common data types:

2.3.1. Integers

Integers are integer numbers that can be used to represent objects such as “number 8.”

```
# Example integer numbers

a = 2
b = 1
c = -1
```

2.3.2. Floating-point numbers

Floating-point numbers are a type of number that can be used to represent real numbers or floating-point values.

```
# import the math module
import math

# Code to compute solution to the quadratic equation of the form a x^2 + b x + c = 0
sol1 = (-b + math.sqrt(b**2 - 4*a*c))/(2*a)
sol2 = (-b - math.sqrt(b**2 - 4*a*c))/(2*a)
```

```
print([sol1,sol2])
```

```
[0.5, -1.0]
```

The build-in Python `type()` function returns the type of these objects.

```
type(sol1)
```

```
float
```

2.3.3. Strings

A string is used to define a sequence of characters. Consider the word “hello.” Strings are **immutable** in Python 3. You can’t modify it afterwards if you’ve previously defined one.

```
my_string = "Hello world"
my_string.replace("world", " Mahidol")
```

```
'Hello  Mahidol'
```

```
print(my_string)
```

```
Hello world
```

While commands like `replace()` and `join()` can modify a string, they generate a copy of it and apply the changes to it rather than rewriting the original.

2.3.3.1. How to Create a String in Python

Using single, double, or triple quotations, you can make a string in three different ways. Here’s an example of each possibility:

```
second_string = 'Mahidol University'
```

The `print()` function can then be used to print your string in the console window. This allows you to go through your code and make sure everything works properly. Here’s a sample for you:

```
print(second_string)
```

```
Mahidol University
```

```
third_string = '''Department of Mathematics,
Faculty of Science,
Mahidol University'''
```

```
print(third_string)
```



```
Department of Mathematics,
Faculty of Science,
Mahidol University
```

2.3.3.2. Concatenating strings

The next skill you can learn is concatenation, which is a method of joining two strings using the "+" operator. Here's how you do it:

```
my_string + " from " +  second_string
```

```
'Hello world from Mahidol University'
```

Note that the + operator cannot be used on two different data types, such as string and integer. You'll get the following Python error if you try it:

2.3.3.3. Escaping Characters

In a Python string, backslashes (\) are used to escape characters.

For example, the given code can be used to print a string containing quote marks.

```
# Quote from Albert Einstein

"\"Imagination is the highest form of research\". – Albert Einstein"
```

```
'"Imagination is the highest form of research". – Albert Einstein'
```

2.3.3.4. String Indexing and Slicing

Because strings are lists of characters, Python strings can be indexed using the same notation as lists. Bracket notation ([index]) can be used to access a single character, or slicing can be used to access a substring ([start:end]).

Indexing with negative numbers counts from the end of the string.

```
print(my_string[0])
print(my_string[0:5])
print(my_string[-5:])
```

```
H
Hello
world
```

2.3.3.5. Iterate String

To iterate through a string in Python, "for…in" notation is used.

```
for c in second_string[:8]:
    print(c)
```

```
M
a
h
i
d
o
l
```

2.4. Python Variables

Variables are containers for storing data values. Variable names are case-sensitive.

Variables in Python 3 are special symbols that assign a specific storage location to a value that's tied to it. In essence, variables are like special labels that you place on some value to know where it's stored.

The code below demonstrates how to store a string in a variable.

```
my_string = "Hello world"
```

Let's break it down a bit further:

- my_string is the variable name.
- = is the assignment operator.
- "Hello world" is a value you tie to the variable name.

Variables do not need to be declared with any particular type, and can even change type after they have been set.

```
x = 10       # x is of type int
x = "SCMA"   # x is now of type str
print(x)
```

```
SCMA
```

2.4.1. Variable Names

A variable can have a short name (such as x and y) or a longer name (such as age, carname, or total volume). Variables in Python have the following rules:

- The name of a variable must begin with a letter or the underscore character.
- A number cannot be the first character in a variable name.
- Only alpha-numeric characters and underscores (A-z, 0-9, and _) are allowed in variable names.
- Case matters when it comes to variable names (age, Age and AGE are three different variables)

2.4.2. Casting

Casting can be used to specify the data type of a variable.

- `int()` function trims the values after the decimal point and returns only the integer/whole number part.
- `float()` function is used to convert any data type to a floating-point number.
- `str()` function is used to convert integer into a string.

```
x = str(10)
type(x)
```

```
str
```

```
y = float(10)
type(y)
```

```
float
```

```
z = int(10.1)
z
```

```
10
```

2.5. Python Lists

In Python, lists are another important data type for specifying an ordered series of elements. In particular, they allow you to group related data and perform the same operations on multiple variables at once. Lists, unlike strings, are mutable (that is, they may be changed).

Each value in a list is referred to as an item, and it is enclosed in square brackets `[ ]`, separated by commas. It is good practice to put a space between the comma and the next value. The values in a list do not need to be unique (the same value can be repeated).

Empty lists do not contain any values within the square brackets.

```
my_list = [1, 2, 3]
```

Alternatively, you can perform the same thing with the `list()` function:

```
third_list = list((1 , 2, 3))
print(third_list)
```

```
[1, 2, 3]
```

```
my_list == third_list
```

```
True
```

In Python, lists are a versatile data type that can contain multiple different data types within the same square brackets. The possible data types within a list include numbers, strings, other objects, and even other lists.

```
second_list = ["a", 2, "e", 4, "i", 6, "o", 8, "u"]
```

2.5.1. How to add items to a List

You can add new items to existing lists in two methods. The first involves the use of the `append()` method:

The `insert()` method can be used to add an item to the specified index:

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)
```

```
[1, 2, 3, 4]
```

```
my_list.insert(2, 2.5)
print(my_list)
```

```
[1, 2, 2.5, 3, 4]
```

2.5.2. How can I remove an item from a list?

You can do it in a variety of ways.

- To begin, use the `remove()` method.
- You can also use the `pop()` method. If no index is supplied, the last item will be removed.
- The final way is to remove a specific item using the "`del`" keyword. If you want to scrap the entire list, you can use del.

```
my_list.remove(2.5)
print(my_list)
```



```
[1, 2, 3, 4]
```

```
my_list.pop(1)
print(my_list)
```

```
[1, 3, 4]
```

```
my_list = [0, 1, 2, 3, 4]
del my_list [3]
print(my_list)
```

```
[0, 1, 2, 4]
```

```
my_list = [0, 1, 2, 3, 4]
del my_list
```

2.5.3. Combine two lists into one

Use the + operator to combine two lists.

```
my_list = [0, 1, 2, 3, 4]
second_list = ["a", 2, "e", 4, "i", 6, "o", 8, "u"]
print(my_list + second_list)
```

```
[0, 1, 2, 3, 4, 'a', 2, 'e', 4, 'i', 6, 'o', 8, 'u']
```

2.5.4. Change item value on your list

You can easily overwrite a value of one list items:

```
second_list = ["a", 2, "e", 4, "i", 6, "o", 8, "u"]

second_list[0] = 1
second_list[2] = 3
second_list[4] = 5
second_list[6] = 7
second_list[8] = 9
print(second_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2.5.5. Loop through the list

Using for loop you can multiply the usage of certain items, similarly to what * operator does. Here's an example:

```
last_list = []
for x in range(1,4):
    last_list += ["Math"]
print(last_list)
```

```
['Math', 'Math', 'Math']
```

Here the `range(start, stop, step)` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

- start(Optional) An integer number specifying at which position to start. Default is 0
- stop(Required) An integer number specifying at which position to stop **(not included)**.
- step(Optional) An integer number specifying the incrementation. Default is 1

```
for x in range(1,4):
    print(x)
```



```
1
2
3
```

2.5.6. Copy a List

Use the built-in copy() function to replicate your data. Alternatively, you can copy a list with the list() method.

```
old_list = ["apple", "banana", "orange"]
new_list = old_list
print(new_list)
```

```
['apple', 'banana', 'orange']
```

```
old_list = ["apple", "banana", "orange"]
new_list = old_list.copy()
print(new_list)
```

```
['apple', 'banana', 'orange']
```

```
old_list = ["apple", "banana", "orange"]
new_list = list(old_list)
print(new_list)
```

```
['apple', 'banana', 'orange']
```

2.5.7. Lists Comprehensions

List comprehensions are a convenient way to make new lists from existing ones. You can also create with strings and tuples when using them.

Here an example how to create a new list with list comprehension.

`list_variable = [x for x in iterable]`

Here's a more complex example that features math operators, integers, and the `range()` function:

```
squared_evens = [x ** 2 for x in range(11) if x % 2 == 0]
print(squared_evens)
```

```
[0, 4, 16, 36, 64, 100]
```

2.6. Python Booleans

Booleans represent one of two values: `True` or `False`. It is useful to perform a filtering operation on a data.

In programming you often need to know if an expression is True or False. For example, when you compare two values, the expression is evaluated and Python returns the Boolean answer:

```
print(8 > 2)
print(8 == 2)
print(2 < 8)
```

```
True
False
True
```

The next Python command prints a message based on whether the condition is `True` or `False`:

```
a = 22/7
b = 3.14

if b > a:
    print("b is greater than a")
else:
    print("b is not greater than a")
```

```
b is not greater than a
```

2.7. Python Dictionaries

Dictionaries are used to store key-value pairs. These key-value pairs offer a great way of organizing and storing data in Python. They are **mutable**, meaning you can change the stored information.

2.7.1. How to create a Python dictionary

Here's a quick example showcasing how to make an empty dictionary.

```
new_dict = {}
other_dict= dict()
```

When you want your values to be indexed by unique keys, they come in handy. Curly braces can be used to create a dictionary in Python. A colon is also used to separate a key and a value.

```
new_dict = {
    "brand": "Tesla",
    "model": "Model X",
    "year": 2021
}
print(new_dict)
```

```
{'brand': 'Tesla', 'model': 'Model X', 'year': 2021}
```

2.7.2. How to access a value in a dictionary

You can access any of the values in your dictionary the following way:

- dict["key"]

```
print(new_dict["model"])
```

```
Model X
```

2.7.3. Dictionary methods

- `keys()` method in Python Dictionary, returns a view object that displays a list of all the keys in the dictionary in order of insertion.
- `values()` method returns a view object that displays a list of all the values of the dictionary.
- `items()` method is used to return the list with all dictionary keys with values.

```
new_dict.keys()
```

```
dict_keys(['brand', 'model', 'year'])
```

```
new_dict.values()
```

```
dict_values(['Tesla', 'Model X', 2021])
```

```
new_dict.items()
```



```
dict_items([('brand', 'Tesla'), ('model', 'Model X'), ('year', 2021)])
```

Let's look at an example and see how lists compare to dictionaries.

Assume we have some movies and you want to keep track of their ratings. We also want to be able to quickly retrieve a movie's rating just knowing the title. We can accomplish this by employing two lists or a single dictionary. Take, for example, movies. The index("Ex Machina") code returns the index for the "Ex Machina" movie.

```
movies = ["Doctor Strage", "Venom", "Thor"]
ratings = [8.9, 7.8, 8.4]

movie_choice_index = movies.index("Venom")
print(ratings[movie_choice_index])
```

```
7.8
```

```
ratings = {
    "Doctor Strange": 8.9,
    "Venom": 7.8,
    "Thor" : 8.4
}

print(ratings["Venom"])
```

```
7.8
```

2.7.4. Operations with dictionaries

Our dictionaries allow us to add, edit, and delete information. We may simply use this code our `dict[key] = value` to add or update the data. When we wish to get rid of a key-value pair, we use `del(our dict[key])`.

```
ratings["Ant-Man"] = 8.3
print(ratings)

ratings["Doctor Strange"] = 9.1
print(ratings)

del(ratings["Thor"])
print(ratings)
```

```
{'Doctor Strange': 8.9, 'Venom': 7.8, 'Thor': 8.4, 'Ant-Man': 8.3}
```

```
{'Doctor Strange': 9.1, 'Venom': 7.8, 'Thor': 8.4, 'Ant-Man': 8.3}
{'Doctor Strange': 9.1, 'Venom': 7.8, 'Ant-Man': 8.3}
```

2.7.5. Loop Through the Dictionary

To implement looping, we can use `for` loop command.

```
ratings = {
    "Doctor Strange": 8.9,
    "Venom": 7.8,
    "Thor" : 8.4
}


#print all key names in the dictionary
for x in ratings:
    print(x)
```

```
Doctor Strange
Venom
Thor
```



```
#print all values in the dictionary
for x in ratings:
    print(ratings[x])
```

```
8.9
7.8
8.4
```

```
#loop through both keys and values
for x, y in ratings.items():
    print(x, y)
```

```
Doctor Strange 8.9
Venom 7.8
Thor 8.4
```

2.8. How to Define a Function

Python 3 allows you to define your own functions for your application in addition to using built-in functions. To summarize, a function is a set of programmed instructions that carry out a specific task. A function can be reused throughout your program once it has been properly defined, i.e. the same code can be reused.

A simple overview of how to define a function in Python can be found here:

Use `def` keyword followed by the function `name():`. The parentheses can contain any parameters that your function should take (or stay empty).

```
def name():
    print("What's your name?")
```

```
name()
```

```
What's your name?
```

```
# Define function with parameters
def product_info(productname, price):
    print("Product Name: " + productname)
    print("Price: " + str(price))

# Call function with parameters assigned as above
product_info("Apple Watch ",30000)
```

```
Product Name: Apple Watch
Price: 30000
```

```
# Call function with keyword arguments
product_info(productname= "Ipad Air 3", price=20000)
```

```
Product Name: Ipad Air 3
Price: 20000
```

2.9. If Statements (Conditional Statements) in Python

Similar to other programming languages, Python supports the basic logical conditions from math:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

You can leverage these conditions in various ways. But most likely, you'll use them in "if statements" and loops.

2.9.1. If Statement Example

The goal of a conditional statement is to check if it's True or False.

```
a = 22/7

if a > math.pi:
    print("a is greater than pi!")
```

```
a is greater than pi!
```

2.9.2. Elif Statements

The `elif` keyword instructs your program to try again, if the previous condition(s) were false.

The `else` keyword helps you add some additional filters to your condition clause.

Here's how an if-elif-else combo looks.

```
Score = 67

if Score >= 80:
    print("Your grade is A")
elif Score >= 75:
    print("Your grade is B+")
elif Score >= 70:
    print("Your grade is B")
elif Score >= 65:
    print("Your grade is C+")
elif Score >= 60:
    print("Your grade is C")
else:
    print("Your grade is D")
```

```
Your grade is C+
```

2.10. Loops in Python

For loops and **while loops** are two simple loop statements in Python that are useful to know.

Let's take a look at each one individually.

2.10.1. For Loop

For loop is a useful approach to iterate through a sequence such as a list, tuple, dictionary, string, and so on, as seen in the other sections of this Python basics.

An example of how to loop through a string is as follows:

```
for x in "Mahidol":
    print(x)
```

```
M
a
h
i
```

```
d
o
l
```


2.10.2. While Loops

A while loop allows you to run a group of statements as long as their condition is true.

```
#print as long as x is less than 8
i=1
while i< 8:
    print(i)
    i += 1
```

```
1
2
3
4
5
6
7
```

2.10.3. How to Break a Loop

Even if the condition is met, you can stop the loop from executing. Use the break statement in both while and for loops to accomplish this:

```
#print if x is less than 8, but skip four

i=1
while i< 8:
    print(i)
    if i == 4:
        break
    i += 1
```

```
1
2
3
4
```

```
print(i)
```

```
4
```

Previous
1. Introduction to Data Science

Next
3. Data Selection

By Pairote Satiracoo
© Copyright 2021.

3 Data Preparation

3. Data Preparation

3.1. Data Preparation with Pandas

This chapter will show you how to use the pandas package to import and preprocess data. Preprocessing is the process of pre-analyzing data before converting it to a standard and normalized format. The following are some of the aspects of preprocessing:

- missing values
- data normalization
- data standardization
- data binning

We'll simply be dealing with missing values in this session.

3.1.1. Importing data

We will utilize the Iris dataset in this tutorial, which can be downloaded from the UCI Machine Learning Repository, https://archive.ics.uci.edu/ml/datasets/iris.

In the pattern recognition literature, this is probably the most well-known database. Fisher's paper is considered a classic in the subject and is still cited frequently. (See, for example, Duda & Hart.) The data collection has three classes, each with 50 instances, each referring to a different species of iris plant. The three classes include

- Iris Setosa
- Iris Versicolour
- Iris Virginica.

To begin, use the pandas library to import data and transform it to a dataframe.

```
import pandas as pd
```

```
#iris = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data', header=None,
#                names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
'class'])

iris = pd.read_csv('/Users/Kaemyuijang/SCMA248/Data/iris.data', header=None, names =
['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class'])


#url = 'https://raw.githubusercontent.com/pairote-sat/SCMA248/main/Data/iris.data'

#iris = pd.read_csv(url, header=None, names = ['sepal_length', 'sepal_width',
'petal_length', 'petal_width', 'class'])

#iris = pd.read_csv('https://raw.githubusercontent.com/pairote-
sat/SCMA248/main/Data/iris.data', header=None,
#                names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
'class'])

type(iris)
```

```
pandas.core.frame.DataFrame
```

Here we specify whether there is a header (`header`) and the variable names (using `names` and a list).

The resulting object `iris` is a pandas DataFrame.

We will print the first 10 rows and the last 10 rows of the dataset using the head(10) method to get an idea of its contents.

```
iris.head(10)
```

	sepal_length	sepal_width	petal_length	petal_width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa

```
iris.tail(10)
```

	sepal_length	sepal_width	petal_length	petal_width	class
140	6.7	3.1	5.6	2.4	Iris-virginica
141	6.9	3.1	5.1	2.3	Iris-virginica
142	5.8	2.7	5.1	1.9	Iris-virginica
143	6.8	3.2	5.9	2.3	Iris-virginica
144	6.7	3.3	5.7	2.5	Iris-virginica
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

To get the names of the columns (the variable names), you can use `columns` method.

```
iris.columns
```

```
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class'],
dtype='object')
```

To extract the class column, you can simply use the following commands:

```
iris['class']
```

```
0         Iris-setosa
1         Iris-setosa
2         Iris-setosa
3         Iris-setosa
4         Iris-setosa
             ...
145    Iris-virginica
146    Iris-virginica
147    Iris-virginica
148    Iris-virginica
149    Iris-virginica
Name: class, Length: 150, dtype: object
```


The Pandas Series is a one-dimensional labeled array that may hold any type of data (integer, string, float, python objects, etc.)

```
type(iris['class'])
```

```
pandas.core.series.Series
```

```
iris.dtypes
```

```
sepal_length    float64
sepal_width     float64
petal_length    float64
petal_width     float64
class            object
dtype: object
```

Tab completion for column names (as well as public attributes), `iris.<TAB>` , is enabled by default if you're using Jupyter.

For example, type `iris.` and then follow with the TAB key. Look for the `shape` attribute.

The `shape` attribute of pandas.DataFrame stores the number of rows and columns as a tuple (number of rows, number of columns).

```
iris.shape
```

```
(150, 5)
```

```
iris.info
```

```
<bound method DataFrame.info of      sepal_length  sepal_width  petal_length  petal_width
class
0             5.1          3.5           1.4          0.2     Iris-setosa
1             4.9          3.0           1.4          0.2     Iris-setosa
2             4.7          3.2           1.3          0.2     Iris-setosa
3             4.6          3.1           1.5          0.2     Iris-setosa
4             5.0          3.6           1.4          0.2     Iris-setosa
..            ...          ...           ...          ...              ...
145           6.7          3.0           5.2          2.3  Iris-virginica
146           6.3          2.5           5.0          1.9  Iris-virginica
147           6.5          3.0           5.2          2.0  Iris-virginica
148           6.2          3.4           5.4          2.3  Iris-virginica
149           5.9          3.0           5.1          1.8  Iris-virginica

[150 rows x 5 columns]>
```

```
import pandas as pd

values = {'dates':  ['20210305','20210316','20210328'],
          'status': ['Opened','Opened','Closed']
          }

demo = pd.DataFrame(values)
```

```
demo
```

	dates	status
0	20210305	Opened
1	20210316	Opened
2	20210328	Closed

```
demo['dates'] = pd.to_datetime(demo['dates'], format='%Y%m%d')
```

```
demo
```

	dates	status
0	2021-03-05	Opened
1	2021-03-16	Opened
2	2021-03-28	Closed

```
demo.to_csv('demo_df.csv')
```

3.2. Data Selection

We'll concentrate on how to slice, dice, and retrieve and set subsets of pandas objects in general. Because Series and DataFrame have received greater development attention in this area, they will be the key focus.

The axis labeling information in pandas objects is useful for a variety of reasons:

- Data is identified (metadata is provided) using established indicators, which is useful for analysis, visualization, and interactive console display.
- Allows for both implicit and explicit data alignment.
- Allows you to access and set subsets of the data set in an intuitive way.

Three different forms of multi-axis indexing are currently supported by pandas.

1. The indexing operators [] and attribute operator. in Python and NumPy offer quick and easy access to pandas data structures in a variety of situations.
2. `.loc` is mostly label-based, but it can also be used with a boolean array. .When the items are not found, .loc will produce a KeyError.
3. `.iloc` works with an integer array (from 0 to length-1 of the axis), but it can also work with a boolean array.

Except for slice indexers, which enable out-of-bounds indexing, .iloc will throw IndexError if a requested indexer is out-of-bounds. (This is in line with the Python/NumPy slice semantics.)

In this section we will use the Iris dataset. First we obtain the row and column names by using the following commands:

```
print(iris.index)
print(iris.columns)
```

```
RangeIndex(start=0, stop=150, step=1)
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class'],
dtype='object')
```

```
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

3.2.1. The indexing operators []

To begin, simply indicate the column and line (by using its index) you're interested in.

You can use the following command to get the sepal width of the fifth line (index is 4):

```
iris['sepal_width'][4]
```



```
3.6
```

```
# Not working
# iris[4]['sepal_width']
```

Note: Be careful, because this is not a matrix, and you might be tempted to insert the row first, then the column. Remember that it's a pandas DataFrame, and the [] operator operates on columns first, then the element of the pandas Series that results."

Sub-matrix retrieval is a simple procedure that requires only the specification of lists of indexes rather than scalars.

```
iris['sepal_width'][0:4]
```

```
0    3.5
1    3.0
2    3.2
3    3.1
Name: sepal_width, dtype: float64
```

```
iris[['petal_width','sepal_width']][0:4]
```

	petal_width	sepal_width
0	0.2	3.5
1	0.2	3.0
2	0.2	3.2
3	0.2	3.1

```
iris['sepal_width'][range(4)]
```

```
0    3.5
1    3.0
2    3.2
3    3.1
Name: sepal_width, dtype: float64
```

3.2.2. .loc()

You can use the `.loc()` method to get something similar to the other approach (as in a matrix) of obtaining data.

```
iris.loc[4,'sepal_width']
```

```
3.6
```

```
# rows at index labels between 0 and 4 (inclusive)
# See https://stackoverflow.com/questions/31593201/how-are-iloc-and-loc-different

iris.loc[0:4,'sepal_width']
```

```
0    3.5
1    3.0
2    3.2
3    3.1
4    3.6
Name: sepal_width, dtype: float64
```

```
iris.loc[range(4),['petal_width','sepal_width']]
```

	petal_width	sepal_width
0	0.2	3.5
1	0.2	3.0
2	0.2	3.2
3	0.2	3.1

3.2.3. .iloc()

Finally, there is `.iloc()`, which is a fully optimized function that defines the positions (as in a matrix). It requires you to define the cell using the row and column numbers.

```
iris.iloc[4,1]
```

```
3.6
```

The following commands produce the same output as `iris.loc[0:4,'sepal_width']` and `iris.loc[range(4),['petal_width','sepal_width']]`

```
# rows at index locations between 0 and 4 (exclusive)
# See https://stackoverflow.com/questions/31593201/how-are-iloc-and-loc-different

iris.iloc[0:4,1]
```

```
0    3.5
1    3.0
2    3.2
3    3.1
Name: sepal_width, dtype: float64
```

```
iris.iloc[range(4),[3,1]]
```

	petal_width	sepal_width
0	0.2	3.5
1	0.2	3.0
2	0.2	3.2
3	0.2	3.1

Note: .loc, .iloc, and also [] indexing can accept a callable as indexer as illustrated from the following examples. The callable must be a function with one argument (the calling Series or DataFrame) that returns valid output for indexing.

```
iris.loc[:,lambda df: ['petal_length','sepal_length']]
```

	petal_length	sepal_length
0	1.4	5.1
1	1.4	4.9
2	1.3	4.7
3	1.5	4.6
4	1.4	5.0
...	...	...
145	5.2	6.7
146	5.0	6.3
147	5.2	6.5
148	5.4	6.2
149	5.1	5.9

150 rows × 2 columns

```
iris.loc[lambda df: df['sepal_width'] > 3.5, :]
```

	sepal_length	sepal_width	petal_length	petal_width	class
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
10	5.4	3.7	1.5	0.2	Iris-setosa
14	5.8	4.0	1.2	0.2	Iris-setosa
15	5.7	4.4	1.5	0.4	Iris-setosa
16	5.4	3.9	1.3	0.4	Iris-setosa
18	5.7	3.8	1.7	0.3	Iris-setosa
19	5.1	3.8	1.5	0.3	Iris-setosa
21	5.1	3.7	1.5	0.4	Iris-setosa
22	4.6	3.6	1.0	0.2	Iris-setosa
32	5.2	4.1	1.5	0.1	Iris-setosa
33	5.5	4.2	1.4	0.2	Iris-setosa
44	5.1	3.8	1.9	0.4	Iris-setosa
46	5.1	3.8	1.6	0.2	Iris-setosa
48	5.3	3.7	1.5	0.2	Iris-setosa
109	7.2	3.6	6.1	2.5	Iris-virginica
117	7.7	3.8	6.7	2.2	Iris-virginica
131	7.9	3.8	6.4	2.0	Iris-virginica

3.3. Dealing with Problematic Data

3.3.1. Problem in setting index in pandas DataFrame

It may happen that the dataset contains an index column. How to import it correctly with Pandas?

We will use a very simple dataset, namely demo_df.csv (the file can be download from my github repository), that contains an index column (this is just a counter and not a feature).

```
import pandas as pd

# How to read CSV file from GitHub using pandas
# https://stackoverflow.com/questions/55240330/how-to-read-csv-file-from-github-using-pandas

# url = 'https://raw.githubusercontent.com/pairote-sat/SCMA248/main/demo_df'
url = 'https://raw.githubusercontent.com/pairote-sat/SCMA248/main/Data/demo_df.csv'

df1 = pd.read_csv(url)
print(df1.head())
df1.columns
```

```
---------------------------------------------------------------------------
SSLCertVerificationError                  Traceback (most recent call last)
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/request.py in
do_open(self, http_class, req, **http_conn_args)
   1349                 h.request(req.get_method(), req.selector, req.data, headers,
-> 1350                           encode_chunked=req.has_header('Transfer-encoding'))
   1351             except OSError as err: # timeout error

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/http/client.py in
request(self, method, url, body, headers, encode_chunked)
   1261         """Send a complete request to the server."""
-> 1262         self._send_request(method, url, body, headers, encode_chunked)
   1263

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/http/client.py in
_send_request(self, method, url, body, headers, encode_chunked)
   1307             body = _encode(body, 'body')
-> 1308         self.endheaders(body, encode_chunked=encode_chunked)
   1309

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/http/client.py in
endheaders(self, message_body, encode_chunked)
   1256             raise CannotSendHeader()
-> 1257         self._send_output(message_body, encode_chunked=encode_chunked)
   1258

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/http/client.py in
_send_output(self, message_body, encode_chunked)
   1027         del self._buffer[:]
-> 1028         self.send(msg)
   1029

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/http/client.py in
send(self, data)
    967             if self.auto_open:
--> 968                 self.connect()
    969             else:

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/http/client.py in
connect(self)
   1431             self.sock = self._context.wrap_socket(self.sock,
-> 1432
server_hostname=server_hostname)
   1433

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/ssl.py in
wrap_socket(self, sock, server_side, do_handshake_on_connect, suppress_ragged_eofs,
server_hostname, session)
    422             context=self,
--> 423             session=session
    424         )

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/ssl.py in _create(cls,
sock, server_side, do_handshake_on_connect, suppress_ragged_eofs, server_hostname,
context, session)
    869                         raise ValueError("do_handshake_on_connect should not be
specified for non-blocking sockets")
--> 870                     self.do_handshake()
    871             except (OSError, ValueError):

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/ssl.py in
do_handshake(self, block)
   1138                 self.settimeout(None)
-> 1139             self._sslobj.do_handshake()
   1140         finally:

SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed:
unable to get local issuer certificate (_ssl.c:1091)

During handling of the above exception, another exception occurred:

URLError                                  Traceback (most recent call last)
/var/folders/kl/h_r05n_j76n32kt0dwy7kynw0000gn/T/ipykernel_2309/3690297895.py in <module>
      7 url = 'https://raw.githubusercontent.com/pairote-
sat/SCMA248/main/Data/demo_df.csv'
      8
----> 9 df1 = pd.read_csv(url)
     10 print(df1.head())
     11 df1.columns

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/util/_decorators.py in wrapper(*args, **kwargs)
    309                     stacklevel=stacklevel,
    310                 )
--> 311             return func(*args, **kwargs)
    312
    313         return wrapper

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
```

```
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/parsers/readers.py in read_csv(filepath_or_buffer, sep, delimiter,
header, names, index_col, usecols, squeeze, prefix, mangle_dupe_cols, dtype, engine,
converters, true_values, false_values, skipinitialspace, skiprows, skipfooter, nrows,
na_values, keep_default_na, na_filter, verbose, skip_blank_lines, parse_dates,
infer_datetime_format, keep_date_col, date_parser, dayfirst, cache_dates, iterator,
chunksize, compression, thousands, decimal, lineterminator, quotechar, quoting,
doublequote, escapechar, comment, encoding, encoding_errors, dialect, error_bad_lines,
warn_bad_lines, on_bad_lines, delim_whitespace, low_memory, memory_map, float_precision,
storage_options)
    584     kwds.update(kwds_defaults)
    585
--> 586     return _read(filepath_or_buffer, kwds)
    587
    588

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/parsers/readers.py in _read(filepath_or_buffer, kwds)
    480
    481     # Create the parser.
--> 482     parser = TextFileReader(filepath_or_buffer, **kwds)
    483
    484     if chunksize or iterator:

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/parsers/readers.py in __init__(self, f, engine, **kwds)
    809             self.options["has_index_names"] = kwds["has_index_names"]
    810
--> 811         self._engine = self._make_engine(self.engine)
    812
    813     def close(self):

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/parsers/readers.py in _make_engine(self, engine)
   1038             )
   1039         # error: Too many arguments for "ParserBase"
-> 1040         return mapping[engine](self.f, **self.options)  # type: ignore[call-arg]
   1041
   1042     def _failover_to_python(self):

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/parsers/c_parser_wrapper.py in __init__(self, src, **kwds)
     49
     50         # open handles
---> 51         self._open_handles(src, kwds)
     52         assert self.handles is not None
     53

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/parsers/base_parser.py in _open_handles(self, src, kwds)
    227             memory_map=kwds.get("memory_map", False),
    228             storage_options=kwds.get("storage_options", None),
--> 229             errors=kwds.get("encoding_errors", "strict"),
    230         )
    231

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/common.py in get_handle(path_or_buf, mode, encoding, compression,
memory_map, is_text, errors, storage_options)
    612         compression=compression,
    613         mode=mode,
--> 614         storage_options=storage_options,
    615     )
    616

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/common.py in _get_filepath_or_buffer(filepath_or_buffer, encoding,
compression, mode, storage_options)
    310         # assuming storage_options is to be interpreted as headers
    311         req_info = urllib.request.Request(filepath_or_buffer,
headers=storage_options)
--> 312         with urlopen(req_info) as req:
    313             content_encoding = req.headers.get("Content-Encoding", None)
    314             if content_encoding == "gzip":

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/pandas/io/common.py in urlopen(*args, **kwargs)
    210     import urllib.request
    211
--> 212     return urllib.request.urlopen(*args, **kwargs)
    213
    214
```



```
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/request.py in
urlopen(url, data, timeout, cafile, capath, cadefault, context)
    220     else:
    221         opener = _opener
--> 222     return opener.open(url, data, timeout)
    223
    224 def install_opener(opener):
```

```
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/request.py in
open(self, fullurl, data, timeout)
    523             req = meth(req)
    524
--> 525         response = self._open(req, data)
    526
    527         # post-process response

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/request.py in
_open(self, req, data)
    541         protocol = req.type
    542         result = self._call_chain(self.handle_open, protocol, protocol +
--> 543                                   '_open', req)
    544         if result:
    545             return result

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/request.py in
_call_chain(self, chain, kind, meth_name, *args)
    501         for handler in handlers:
    502             func = getattr(handler, meth_name)
--> 503             result = func(*args)
    504             if result is not None:
    505                 return result

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/request.py in
https_open(self, req)
   1391         def https_open(self, req):
   1392             return self.do_open(http.client.HTTPSConnection, req,
-> 1393                 context=self._context, check_hostname=self._check_hostname)
   1394
   1395         https_request = AbstractHTTPHandler.do_request_

/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/urllib/request.py in
do_open(self, http_class, req, **http_conn_args)
   1350                           encode_chunked=req.has_header('Transfer-encoding'))
   1351             except OSError as err: # timeout error
-> 1352                 raise URLError(err)
   1353             r = h.getresponse()
   1354         except:

URLError: <urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed:
unable to get local issuer certificate (_ssl.c:1091)>
```

```
## Uncomment these commands if the CSV dateset is stored locally.

# df1 = pd.read_csv('/Users/Kaemyuijang/SCMA248/demo_df.csv')
# print(df1.head())
# df1.columns
```

We want to specify that 'Unnamed: 0' is the index column while loading this data set with the following command (with the parameter `index_col`):

```
df1 = pd.read_csv(url, index_col = 0)
df1.head()
```

	dates	status
0	2021-03-05	Opened
1	2021-03-16	Opened
2	2021-03-28	Closed

The dataset is loaded and the index is correct after performing the command.

3.3.2. Convert Strings to Datetime

However, we see an issue right away: all of the data, including dates, has been parsed as integers (or, in other cases, as string). If the dates do not have a particularly unusual format, you can use the autodetection routines to identify the column that contains the date data. It works nicely with the following arguments when the data file is stored locally.


```
# df2 = pd.read_csv('/Users/Kaemyuijang/SCMA248/demo_df.csv', index_col = 0, parse_dates
= ['dates'])
# print(df2.head())
# df2.dtypes
```

For the same dataset downloaded from Github, if a column or index contains an unparseable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use pd.to_datetime after pd.read_csv:

```
pd.to_datetime(df['DataFrame Column'], format=specify your format)
```

Remember that the date format for our example is yyyymmdd.

The following is a representation of this date format `format = '%d%m%Y'` (or `format = '%Y%m%d'`).

See https://datatofish.com/strings-to-datetime-pandas/ for more details.

```
df2 = pd.read_csv(url, index_col = 0, parse_dates = ['dates'])
print(df2)
df2.dtypes
```

```
       dates  status
0 2021-03-05  Opened
1 2021-03-16  Opened
2 2021-03-28  Closed
```

```
dates     datetime64[ns]
status            object
dtype: object
```

```
df2['dates'] = pd.to_datetime(df2['dates'], format='%d%m%Y')
print(df2)
df2.dtypes
```

```
       dates  status
0 2021-03-05  Opened
1 2021-03-16  Opened
2 2021-03-28  Closed
```

```
dates     datetime64[ns]
status            object
dtype: object
```

3.3.3. Missing values

We will concentrate on missing values, which is perhaps the most challenging data cleaning operation.

It's a good idea to have an overall sense of a data set before you start cleaning it. After that, you can develop a plan for cleaning the data.

To begin, I like to ask the following questions:

- What are the features?
- What sorts of data are required (int, float, text, boolean)?
- Is there any evident data missing (values that Pandas can detect)?
- Is there any other type of missing data that isn't as clear (and that Pandas can't easily detect)?

Let's have a look at an example by using a small sample data namely property_data.csv. The file can be obtained from Github: https://raw.githubusercontent.com/pairote-sat/SCMA248/main/property_data.csv.

In what follows, we also specify that 'PID' (personal indentifier) is the index column while loading this data set with the following command (with the parameter index_col):

```
url = 'https://raw.githubusercontent.com/pairote-
sat/SCMA248/main/Data/property_data.csv'

df = pd.read_csv(url, index_col = 0)
df
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
10103.0	NaN	Silom	N	NaN	1	850
10104.0	201.0	Sukhumvit	12	1	NaN	700
NaN	203.0	Sukhumvit	Y	3	2	1600
10106.0	207.0	Sukhumvit	Y	NaN	1	800
10107.0	NaN	Thonglor	NaN	2	HURLEY	950
10108.0	213.0	Rama 1	Y	--	1	NaN
10109.0	215.0	Rama 1	Y	na	2	1800

We notice that the PID (personal identifiers) as the index name has a missing value, i.e. NaN (not any number). We will replace this missing PID with 10105 and also convert from floats to integers.

```
rowindex = df.index.tolist()
rowindex[4] = 10105.0
rowindex = [int(i) for i in rowindex]

df.index = rowindex

print(df.loc[:,'ST_NUM'])
```

```
10101    104.0
10102    197.0
10103      NaN
10104    201.0
10105    203.0
10106    207.0
10107      NaN
10108    213.0
10109    215.0
Name: ST_NUM, dtype: float64
```

Alternatively, one can use Numpy to produce the same result. Simply run the following commands. Here we use `.astype()` method to convert the type of an array.

```
df = pd.read_csv(url, index_col = 0)
df

import numpy as np
rowindex = df.index.to_numpy()

rowindex[4] = 10105.0

df.index = rowindex.astype(int)

print(df.loc[:,'ST_NUM'])
```

```
10101    104.0
10102    197.0
10103      NaN
10104    201.0
10105    203.0
10106    207.0
10107      NaN
10108    213.0
10109    215.0
Name: ST_NUM, dtype: float64
```

Now I can answer my first question: what are features? The following features can be obtained from the column names:

- ST_NUM is the street number
- ST_NAME is the street name
- OWN_OCCUPIED: Is the residence owner occupied?
- NUM_BEDROOMS: the number of rooms

We can also respond to the question, What are the expected types?

- ST_NUM is either a float or an int... a numeric type of some sort
- ST_NAME is a string variable.
- OWN_OCCUPIED: string; OWN_OCCUPIED: string; OWN _OCCUPIED N (“No”) or Y (“Yes”)
- NUM_BEDROOMS is a numeric type that can be either float or int.

3.3.4. Standard Missing Values

So, what exactly do I mean when I say “standard missing values?” These are missing values that Pandas can detect.

Let’s return to our initial dataset and examine the “Street Number” column.

There are an empty cell in the third row (from the original file). A value of “NaN” appears in the seventh row.

Both of these numbers are obviously missing. Let’s see how Pandas handle these situations. We can see that Pandas filled in the blank space with “NaN”.

We can confirm that both the missing value and “NA” were detected as missing values using the isnull() method. True for both boolean responses.

```
df['ST_NUM'].isnull()
```

```
10101    False
10102    False
10103     True
10104    False
10105    False
10106    False
10107     True
10108    False
10109    False
Name: ST_NUM, dtype: bool
```

Similarly, for the NUM_BEDROOMS column of the original CSV file, users manually entering missing values with different names “n/a” and “NA”. Pandas also recognized these as missing values and filled with “NaN”.

```
df['NUM_BEDROOMS']
```

```
10101      3
10102      3
10103    NaN
10104      1
10105      3
10106    NaN
10107      2
10108     --
10109     na
Name: NUM_BEDROOMS, dtype: object
```

3.3.5. Missing Values That Aren’t Standard

It is possible that there are missing values with different formats in some cases.

There are two other missing values in this column of different formats

- na
- --

Putting this different format in a list is a simple approach to detect them. When we import the data, Pandas will immediately recognize them. Here’s an example of how we might go about it.

```
# Making a list of missing value types
missing_values = ["na", "--"]

df = pd.read_csv(url, index_col = 0, na_values = missing_values)

df
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3.0	1	1000.0
10102.0	197.0	Silom	N	3.0	1.5	NaN
10103.0	NaN	Silom	N	NaN	1	850.0
10104.0	201.0	Sukhumvit	12	1.0	NaN	700.0
NaN	203.0	Sukhumvit	Y	3.0	2	1600.0
10106.0	207.0	Sukhumvit	Y	NaN	1	800.0
10107.0	NaN	Thonglor	NaN	2.0	HURLEY	950.0
10108.0	213.0	Rama 1	Y	NaN	1	NaN
10109.0	215.0	Rama 1	Y	NaN	2	1800.0

3.3.6. Unexpected Missing Values

We have observed both standard and non-standard missing data so far. What if we have a type that is not expected?

For instance, if our feature is supposed to be a string but it's a numeric type, it's technically a missing value.

Take a look at the column labeled "OWN_OCCUPIED" to understand what I'm talking about.

```
df['OWN_OCCUPIED']
```

```
PID
10101.0      Y
10102.0      N
10103.0      N
10104.0     12
NaN          Y
10106.0      Y
10107.0    NaN
10108.0      Y
10109.0      Y
Name: OWN_OCCUPIED, dtype: object
```

We know Pandas will recognize the empty cell in row seven as a missing value because of our prior examples.

The number 12 appears in the fourth row. This number type should be a missing value because the result for Owner Occupied should clearly be a string (Y or N). Because this example is a little more complicated, we will need to find a plan for identifying missing values. There are a few alternative routes to take, but this is how I'm going to go about it.

1. Loop through The OWN OCCUPIED column.
2. Convert the entry to an integer.
3. If the entry may be transformed to an integer, enter a missing value.
4. We know the number cannot be an integer if it cannott be an integer.

```
df = pd.read_csv(url, index_col = 0)
df

import numpy as np
rowindex = df.index.to_numpy()

rowindex[4] = 10105.0

df.index = rowindex.astype(int)
```

```
# Detecting numbers
cnt=10101
for row in df['OWN_OCCUPIED']:
    try:
        int(row)
        df.loc[cnt, 'OWN_OCCUPIED']=np.nan
    except ValueError:
        pass
    cnt+=1
```

```
df['OWN_OCCUPIED']
```

```
10101      Y
10102      N
10103      N
10104    NaN
10105      Y
10106      Y
10107    NaN
10108      Y
10109      Y
Name: OWN_OCCUPIED, dtype: object
```

In the code, we loop through each entry in the “Owner Occupied” column. To try to change the entry to an integer, we use int(row). If the value can be changed to an integer, we change the entry to a missing value using np.nan from Numpy. On the other hand, if the value cannot be changed to an integer, we pass it and continue.

You will notice that I have used try and except ValueError. This is called exception handling, and we use this to handle errors.

If we tried to change an entry to an integer and it could not be changed, a ValueError would be returned and the code would terminate. To deal with this, we use exception handling to detect these errors and continue.

3.4. Data Manipulation

We have learned how to select the data we want, we will need to learn how to manipulate it. Using aggregation methods to work with columns or rows is one of the most straightforward things we can perform.

All of these functions always **return a number** when applied to a row or column.

We can specify whether the function should be applied to

- the rows for each column using the `axis=0` keyword on the function argument, or
- the columns for each row using the `axis=1` keyword on the function argument.

Function	Description
df.describe()	Returns a summary statistics of numerical column
df.mean()	Returns the average of all columns in a dataset
df.corr()	Returns the correlation between columns in a DataFrame
df.count()	Returns the number of non-null values in each DataFrame column
df.max()	Returns the highest value in each column
df.min()	Returns the lowest value in each column
df.median()	Returns the median in each column
df.std()	Returns the standard deviation in each column

```
iris = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data', header=None,
                 names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
'class'])

type(iris)
```

```
pandas.core.frame.DataFrame
```

```
iris.describe()
```

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

You have the number of observations, their average value, standard deviation, minimum and maximum values, and certain percentiles for all numerical features" (25 percent, 50 percent, and 75 percent). This offers you a fair picture of how each feature is distributed.

The following command illustrate how to use the `max()` function.

```
iris.max(axis = 0)
```

```
sepal_length               7.9
sepal_width                4.4
petal_length               6.9
petal_width                2.5
class           Iris-virginica
dtype: object
```


We can perform operations on all values in rows, columns, or a subset of both.

The following example shows how to find the standardized values of each column of the Iris dataset. We need to firstly drop the class column, which is a categorical variable using `drop()` function with `axis = 1`.

Recall that the Z-scores and standardized values (sometimes known as standard scores or normal deviates) are the same thing. When you take a data point and scale it by population data, you get a standardized value. It informs us how distant we are from the mean in terms of standard deviations.

```
iris_drop = iris.drop('class', axis = 1)
```

```
print(iris_drop.mean())
print(iris_drop.std())
```

```
sepal_length    5.843333
sepal_width     3.054000
petal_length    3.758667
petal_width     1.198667
dtype: float64
sepal_length    0.828066
sepal_width     0.433594
petal_length    1.764420
petal_width     0.763161
dtype: float64
```

```
def z_score_standardization(series):
    return (series - series.mean()) / series.std(ddof = 1)

#iris_normalized = iris_drop
#for col in iris_normalized.columns:
#    iris_normalized[col] = z_score_standardization(iris_normalized[col])

iris_normalized = {}
for col in iris_drop.columns:
    iris_normalized[col] = z_score_standardization(iris_drop[col])

iris_normalized = pd.DataFrame(iris_normalized)
```

```
iris_normalized.head()
```

	sepal_length	sepal_width	petal_length	petal_width
0	-0.897674	1.028611	-1.336794	-1.308593
1	-1.139200	-0.124540	-1.336794	-1.308593
2	-1.380727	0.336720	-1.393470	-1.308593
3	-1.501490	0.106090	-1.280118	-1.308593
4	-1.018437	1.259242	-1.336794	-1.308593

Alternatively, the following commands produce the same result.

```
del(iris_normalized)

iris_normalized = (iris_drop - iris_drop.mean())/iris_drop.std(ddof=1)
iris_normalized.head()
```

	sepal_length	sepal_width	petal_length	petal_width
0	-0.897674	1.028611	-1.336794	-1.308593
1	-1.139200	-0.124540	-1.336794	-1.308593
2	-1.380727	0.336720	-1.393470	-1.308593
3	-1.501490	0.106090	-1.280118	-1.308593
4	-1.018437	1.259242	-1.336794	-1.308593

Alternatively, we can standardize a Pandas Column with Z-Score Scaling using scikit-learn.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(iris_drop)

iris_scaled = scaler.fit_transform(iris_drop)
iris_scaled = pd.DataFrame(iris_scaled, columns=iris_drop.columns)
print(iris_scaled)
```

```
     sepal_length  sepal_width  petal_length  petal_width
0       -0.900681     1.032057     -1.341272    -1.312977
1       -1.143017    -0.124958     -1.341272    -1.312977
2       -1.385353     0.337848     -1.398138    -1.312977
3       -1.506521     0.106445     -1.284407    -1.312977
4       -1.021849     1.263460     -1.341272    -1.312977
..            ...          ...           ...          ...
145      1.038005    -0.124958      0.819624     1.447956
146      0.553333    -1.281972      0.705893     0.922064
147      0.795669    -0.124958      0.819624     1.053537
148      0.432165     0.800654      0.933356     1.447956
149      0.068662    -0.124958      0.762759     0.790591

[150 rows x 4 columns]
```

Note scikit-learn uses np.std which by **default is the population standard deviation** (where the sum of squared deviations are divided by the number of observations), while the sample standard deviations (where the denominator is number of observations - 1) are used by pandas. This is a correction factor determined by the degrees of freedom in order to obtain an unbiased estimate of the population standard deviation (ddof). Numpy and scikit-learn calculations utilize ddof=0 by default, whereas pandas uses ddof=1 (docs).

As a result, the above results are different.

https://stackoverflow.com/questions/44220290/sklearn-standardscaler-result-different-to-manual-result

In the next example, we simply can apply any binary arithmetical operation (+,-,*,/) to an entire dataframe.

```
iris_drop**2
```

	sepal_length	sepal_width	petal_length	petal_width
0	26.01	12.25	1.96	0.04
1	24.01	9.00	1.96	0.04
2	22.09	10.24	1.69	0.04
3	21.16	9.61	2.25	0.04
4	25.00	12.96	1.96	0.04
...	...	...	...	...
145	44.89	9.00	27.04	5.29
146	39.69	6.25	25.00	3.61
147	42.25	9.00	27.04	4.00
148	38.44	11.56	29.16	5.29
149	34.81	9.00	26.01	3.24

150 rows × 4 columns

Any function can be applied to a DataFrame or Series by passing its name as an argument to the `apply` method. For example, in the following code, we use the NumPy library's `floor` function to return the floor of the input, element-wise of each value in the DataFrame.

```
import numpy as np

iris_drop.apply(np.floor)
```

	sepal_length	sepal_width	petal_length	petal_width
0	5.0	3.0	1.0	0.0
1	4.0	3.0	1.0	0.0
2	4.0	3.0	1.0	0.0
3	4.0	3.0	1.0	0.0
4	5.0	3.0	1.0	0.0
...	...	...	...	...
145	6.0	3.0	5.0	2.0
146	6.0	2.0	5.0	1.0
147	6.0	3.0	5.0	2.0
148	6.0	3.0	5.0	2.0
149	5.0	3.0	5.0	1.0

150 rows × 4 columns

If we need to design a specific function to apply it, we can write an in-line function, commonly known as a λ-function. A λ-function is a function without a name.

It is only necessary to specify the parameters it receives, between the lambda keyword and the colon (:).

In the next example, only one parameter is needed, which will be the value of each element in the DataFrame.

```
iris_drop.apply(lambda x: np.log10(x))
```

	sepal_length	sepal_width	petal_length	petal_width
0	0.707570	0.544068	0.146128	-0.698970
1	0.690196	0.477121	0.146128	-0.698970
2	0.672098	0.505150	0.113943	-0.698970
3	0.662758	0.491362	0.176091	-0.698970
4	0.698970	0.556303	0.146128	-0.698970
...	...	...	...	...
145	0.826075	0.477121	0.716003	0.361728
146	0.799341	0.397940	0.698970	0.278754
147	0.812913	0.477121	0.716003	0.301030
148	0.792392	0.531479	0.732394	0.361728
149	0.770852	0.477121	0.707570	0.255273

150 rows × 4 columns

3.4.1. Add A New Column To An Existing Pandas DataFrame

Adding new values in our DataFrame is another simple manipulation technique. This can be done directly over a DataFrame with the assign operator (=).

For example, we can assign a Series to a selection of a column that does not exist to add a new column to a DataFrame.

You should be aware that previous values will be overridden if a column with the same name already exists.

In the following example, we create a new column entitled sepal_length_normalized by adding the standardized values of the sepal_length column.

```
iris['sepal_length_normalized'] = (iris['sepal_length'] - iris['sepal_length'].mean()) /
iris['sepal_length'].std()
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	class	sepal_length_normalized
0	5.1	3.5	1.4	0.2	Iris-setosa	-0.897674
1	4.9	3.0	1.4	0.2	Iris-setosa	-1.139200
2	4.7	3.2	1.3	0.2	Iris-setosa	-1.380727
3	4.6	3.1	1.5	0.2	Iris-setosa	-1.501490
4	5.0	3.6	1.4	0.2	Iris-setosa	-1.018437

```
iris['sepal_length_normalized'] = (iris['sepal_length'] - iris['sepal_length'].mean()) /
iris['sepal_length'].std(ddof=0)
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	class	sepal_length_normalized
0	5.1	3.5	1.4	0.2	Iris-setosa	-0.900681
1	4.9	3.0	1.4	0.2	Iris-setosa	-1.143017
2	4.7	3.2	1.3	0.2	Iris-setosa	-1.385353
3	4.6	3.1	1.5	0.2	Iris-setosa	-1.506521
4	5.0	3.6	1.4	0.2	Iris-setosa	-1.021849

Alternatively, we can use `concat` function to add a Series (s) to a Pandas DataFrame (df) as a new column with an arguement `axis = 1`. The name of the new column can be set by using `Series.rename` as in the following command:

`df = pd.concat((df, s.rename('CoolColumnName')), axis=1)`

https://stackoverflow.com/questions/39047915/concat-series-onto-dataframe-with-column-name

```
pd.concat([iris, (iris['sepal_length']-1).rename('new_name') ], axis = 1)
```

	sepal_length	sepal_width	petal_length	petal_width	class	sepal_length_normalized	new_name
0	5.1	3.5	1.4	0.2	Iris-setosa	-0.900681	4.1
1	4.9	3.0	1.4	0.2	Iris-setosa	-1.143017	3.9
2	4.7	3.2	1.3	0.2	Iris-setosa	-1.385353	3.7
3	4.6	3.1	1.5	0.2	Iris-setosa	-1.506521	3.6
4	5.0	3.6	1.4	0.2	Iris-setosa	-1.021849	4.0
...	...	...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica	1.038005	5.7
146	6.3	2.5	5.0	1.9	Iris-virginica	0.553333	5.3
147	6.5	3.0	5.2	2.0	Iris-virginica	0.795669	5.5
148	6.2	3.4	5.4	2.3	Iris-virginica	0.432165	5.2
149	5.9	3.0	5.1	1.8	Iris-virginica	0.068662	4.9

150 rows × 7 columns

We can now use the `drop` function to delete a column from the DataFrame; this removes the indicated rows if axis=0, or the indicated columns if axis=1.

All Pandas functions that change the contents of a DataFrame, such as the drop function, return a duplicate of the updated data rather than overwriting the DataFrame. As a result, the original DataFrame is preserved. Set the keyword `inplace` to `True` if you do not wish to maintain the old settings. This keyword is set to `False` by default, which means that a copy of the data is returned.

The following commands remove the column, namely 'sepal_length_normalized', we have just added to the Iris dataset.

```
iris.columns
```

```
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class',
       'sepal_length_normalized'],
      dtype='object')
```

```
print(iris.drop('sepal_length_normalized', axis = 1).head())

print(iris.head())
```

```
   sepal_length  sepal_width  petal_length  petal_width        class
0           5.1          3.5           1.4          0.2  Iris-setosa
1           4.9          3.0           1.4          0.2  Iris-setosa
2           4.7          3.2           1.3          0.2  Iris-setosa
3           4.6          3.1           1.5          0.2  Iris-setosa
4           5.0          3.6           1.4          0.2  Iris-setosa
   sepal_length  sepal_width  petal_length  petal_width        class  \
0           5.1          3.5           1.4          0.2  Iris-setosa
1           4.9          3.0           1.4          0.2  Iris-setosa
2           4.7          3.2           1.3          0.2  Iris-setosa
3           4.6          3.1           1.5          0.2  Iris-setosa
4           5.0          3.6           1.4          0.2  Iris-setosa

   sepal_length_normalized
0                -0.900681
1                -1.143017
2                -1.385353
3                -1.506521
4                -1.021849
```

```
iris.drop('sepal_length_normalized', axis = 1, inplace = True)

print(iris.head())
```

```
   sepal_length  sepal_width  petal_length  petal_width        class
0           5.1          3.5           1.4          0.2  Iris-setosa
1           4.9          3.0           1.4          0.2  Iris-setosa
2           4.7          3.2           1.3          0.2  Iris-setosa
3           4.6          3.1           1.5          0.2  Iris-setosa
4           5.0          3.6           1.4          0.2  Iris-setosa
```

3.4.2. Appending a row to a dataframe and specify its index label

In this section, we will learn how to add/remove new rows and remove missing values. We will use the dataset, namely property_data.csv.

A general solution when appending a row to a dataframe and specify its index label is to create the row, transform the new row data into a pandas series, name it to the index you want to have and then append it to the data frame. Don't forget to overwrite the original data frame with the one with appended row. The following commands produce the required result.

See https://stackoverflow.com/questions/16824607/pandas-appending-a-row-to-a-dataframe-and-specify-its-index-label for more details.

See also https://www.geeksforgeeks.org/python-pandas-dataframe-append/ and https://thispointer.com/python-pandas-how-to-add-rows-in-a-dataframe-using-dataframe-append-loc-iloc/#3 .

```
import pandas as pd

filepath = '/Users/Kaemyuijang/SCMA248/Data/property_data.csv'

df = pd.read_csv(filepath, index_col = 0)
df.head()
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
10103.0	NaN	Silom	N	NaN	1	850
10104.0	201.0	Sukhumvit	12	1	NaN	700
NaN	203.0	Sukhumvit	Y	3	2	1600

```
url = 'https://raw.githubusercontent.com/pairote-sat/SCMA248/main/Data/property_data.csv'

df = pd.read_csv(url, index_col = 0)
df.head()
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
10103.0	NaN	Silom	N	NaN	1	850
10104.0	201.0	Sukhumvit	12	1	NaN	700
NaN	203.0	Sukhumvit	Y	3	2	1600

```
df.index
```

```
Float64Index([10101.0, 10102.0, 10103.0, 10104.0, nan, 10106.0, 10107.0,
              10108.0, 10109.0],
             dtype='float64', name='PID')
```

```
#new_observation = {'ST_NUM': 555 , 'OWN_OCCUPIED': 'Y', 'NUM_BEDROOMS': 2,'NUM_BATH':
1,'SQ_FT': 200}

new_observation = pd.Series({'ST_NUM': 555 , 'OWN_OCCUPIED': 'Y', 'NUM_BEDROOMS':
2,'NUM_BATH': 1,'SQ_FT': 200}, name = 10110.0)
```

```
new_observation
```

```
ST_NUM          555
OWN_OCCUPIED      Y
NUM_BEDROOMS      2
NUM_BATH          1
SQ_FT           200
Name: 10110.0, dtype: object
```

```
df.append(new_observation)
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
10103.0	NaN	Silom	N	NaN	1	850
10104.0	201.0	Sukhumvit	12	1	NaN	700
NaN	203.0	Sukhumvit	Y	3	2	1600
10106.0	207.0	Sukhumvit	Y	NaN	1	800
10107.0	NaN	Thonglor	NaN	2	HURLEY	950
10108.0	213.0	Rama 1	Y	--	1	NaN
10109.0	215.0	Rama 1	Y	na	2	1800
10110.0	555.0	NaN	Y	2	1	200

Note In case that, we do not define a name for our pandas series, i.e. we simply define `next_observation = pd.Series({'ST_NUM': 999 , 'OWN_OCCUPIED': 'Y', 'NUM_BEDROOMS': 2,'NUM_BATH': 1,'SQ_FT': 200})` without the flag `name = 10110.0` as an argument. We must set the `ignore_index` flag in the append method to `True`, otherwise the commands will produce an error as follows:

```
next_observation = pd.Series({'ST_NUM': 999 , 'OWN_OCCUPIED': 'N', 'NUM_BEDROOMS':
5,'NUM_BATH': 4,'SQ_FT': 3500})
```

```
print(next_observation)
print(df)
```

```
ST_NUM          999
OWN_OCCUPIED      N
NUM_BEDROOMS      5
NUM_BATH          4
SQ_FT          3500
dtype: object
        ST_NUM    ST_NAME OWN_OCCUPIED NUM_BEDROOMS NUM_BATH SQ_FT
PID
10101.0   104.0   Khao San            Y            3        1  1000
10102.0   197.0      Silom            N            3      1.5    --
10103.0     NaN      Silom            N          NaN        1   850
10104.0   201.0  Sukhumvit           12            1      NaN   700
NaN       203.0  Sukhumvit            Y            3        2  1600
10106.0   207.0  Sukhumvit            Y          NaN        1   800
10107.0     NaN   Thonglor          NaN            2   HURLEY   950
10108.0   213.0     Rama 1            Y           --        1   NaN
10109.0   215.0     Rama 1            Y           na        2  1800
```

```
# Without setting the ignore_index flag in the append method to True, this produces an
eror.

df.append(next_observation)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/kl/h_r05n_j76n32kt0dwy7kynw0000gn/T/ipykernel_1214/4163196459.py in <module>
      1 # Without setting the ignore_index flag in the append method to True, this
produces an eror.
      2
----> 3 df.append(next_observation)

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/frame.py in append(self, other,
ignore_index, verify_integrity, sort)
   8933             if other.name is None and not ignore_index:
   8934                 raise TypeError(
-> 8935                     "Can only append a Series if ignore_index=True "
   8936                     "or if the Series has a name"
   8937                 )

TypeError: Can only append a Series if ignore_index=True or if the Series has a name
```

```
# Setting the ignore_index flag in the append method to True

df.append(next_observation,ignore_index=True)
```

	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
0	104.0	Khao San	Y	3	1	1000
1	197.0	Silom	N	3	1.5	--
2	NaN	Silom	N	NaN	1	850
3	201.0	Sukhumvit	12	1	NaN	700
4	203.0	Sukhumvit	Y	3	2	1600
5	207.0	Sukhumvit	Y	NaN	1	800
6	NaN	Thonglor	NaN	2	HURLEY	950
7	213.0	Rama 1	Y	--	1	NaN
8	215.0	Rama 1	Y	na	2	1800
9	999.0	NaN	N	5	4	3500

Note The resulting (new) DataFrame's index is not same as original dataframe because ignore_index is passed as True in the `append()` function.

The next complete example illustrates how to add multiple rows to dataframe.

```
print(new_observation)
print(next_observation)
```

```
ST_NUM          555
OWN_OCCUPIED      Y
NUM_BEDROOMS      2
NUM_BATH          1
SQ_FT           200
Name: 10110.0, dtype: object
ST_NUM          999
OWN_OCCUPIED      N
NUM_BEDROOMS      5
NUM_BATH          4
SQ_FT          3500
dtype: object
```

```
listOfSeries = [new_observation,next_observation]

new_df = df.append(listOfSeries, ignore_index = True)
```


Finally, to remove the row(s), we can apply the drop method once more. Now we must set the axis to 0 and specify the row index we want to remove.

```
new_df.drop([9,10], axis = 0, inplace = True)
```

```
new_df
```

	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
0	104.0	Khao San	Y	3	1	1000
1	197.0	Silom	N	3	1.5	--
2	NaN	Silom	N	NaN	1	850
3	201.0	Sukhumvit	12	1	NaN	700
4	203.0	Sukhumvit	Y	3	2	1600
5	207.0	Sukhumvit	Y	NaN	1	800
6	NaN	Thonglor	NaN	2	HURLEY	950
7	213.0	Rama 1	Y	--	1	NaN
8	215.0	Rama 1	Y	na	2	1800

By applying the `drop()` function to the result of the `isnull()` method, missing values can be removed. This has the same effect as filtering the NaN values, as explained above, but instead of returning a view, a duplicate of the DataFrame minus the NaN values is returned.

```
index = new_df.index[new_df['ST_NUM'].isnull()]

# Altenatively, one can obtain the index
new_index = new_df['ST_NUM'].index[new_df['ST_NUM'].apply(np.isnan)]

print(new_df.drop(index, axis = 0))
print(new_df.drop(new_index, axis = 0))
```

```
   ST_NUM    ST_NAME OWN_OCCUPIED NUM_BEDROOMS NUM_BATH SQ_FT
0   104.0   Khao San            Y            3        1  1000
1   197.0      Silom            N            3      1.5    --
3   201.0  Sukhumvit           12            1      NaN   700
4   203.0  Sukhumvit            Y            3        2  1600
5   207.0  Sukhumvit            Y          NaN        1   800
7   213.0     Rama 1            Y           --        1   NaN
8   215.0     Rama 1            Y           na        2  1800
   ST_NUM    ST_NAME OWN_OCCUPIED NUM_BEDROOMS NUM_BATH SQ_FT
0   104.0   Khao San            Y            3        1  1000
1   197.0      Silom            N            3      1.5    --
3   201.0  Sukhumvit           12            1      NaN   700
4   203.0  Sukhumvit            Y            3        2  1600
5   207.0  Sukhumvit            Y          NaN        1   800
7   213.0     Rama 1            Y           --        1   NaN
8   215.0     Rama 1            Y           na        2  1800
```

Instead of using the generic drop function, we can use the particular `dropna()` function to remove NaN values. We must set the `how` keyword to `any` if we wish to delete any record that includes a NaN value. We can use the `subset` keyword to limit it to a specific subset of columns. The effect will be the same as if we used the drop function, as shown below:

Note The parameter `how{'any', 'all'}`, default 'any' determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- 'any' : If any NA values are present, drop that row or column.
- 'all' : If all values are NA, drop that row or column.

```
df.dropna(how = 'any', subset = ['ST_NUM'])
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
10104.0	201.0	Sukhumvit	12	1	NaN	700
NaN	203.0	Sukhumvit	Y	3	2	1600
10106.0	207.0	Sukhumvit	Y	NaN	1	800
10108.0	213.0	Rama 1	Y	--	1	NaN
10109.0	215.0	Rama 1	Y	na	2	1800

```
# Setting subset keyword with a subset of columns

df.dropna(how = 'any', subset = ['ST_NUM','NUM_BATH'])
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
NaN	203.0	Sukhumvit	Y	3	2	1600
10106.0	207.0	Sukhumvit	Y	NaN	1	800
10108.0	213.0	Rama 1	Y	--	1	NaN
10109.0	215.0	Rama 1	Y	na	2	1800

If we wish to fill the rows containing NaN with another value instead of removing them, we may use the `fillna()` method and specify the value to use. If we only want to fill certain columns, we must pass a dictionary as an argument to the `fillna()` function, with the names of the columns as the key and the character to use for filling as the value.

```
df.fillna(value = {'ST_NUM':0})
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
10103.0	0.0	Silom	N	NaN	1	850
10104.0	201.0	Sukhumvit	12	1	NaN	700
NaN	203.0	Sukhumvit	Y	3	2	1600
10106.0	207.0	Sukhumvit	Y	NaN	1	800
10107.0	0.0	Thonglor	NaN	2	HURLEY	950
10108.0	213.0	Rama 1	Y	--	1	NaN
10109.0	215.0	Rama 1	Y	na	2	1800

```
df.fillna(value = {'ST_NUM': -1, 'NUM_BEDROOMS': -1 })
```

PID	ST_NUM	ST_NAME	OWN_OCCUPIED	NUM_BEDROOMS	NUM_BATH	SQ_FT
10101.0	104.0	Khao San	Y	3	1	1000
10102.0	197.0	Silom	N	3	1.5	--
10103.0	-1.0	Silom	N	-1	1	850
10104.0	201.0	Sukhumvit	12	1	NaN	700
NaN	203.0	Sukhumvit	Y	3	2	1600
10106.0	207.0	Sukhumvit	Y	-1	1	800
10107.0	-1.0	Thonglor	NaN	2	HURLEY	950
10108.0	213.0	Rama 1	Y	--	1	NaN
10109.0	215.0	Rama 1	Y	na	2	1800

3.4.3. Sorting

Sorting by columns is another important feature we will need while examining at our data. Using the sort method, we can sort a DataFrame by any column. We only need to execute the following commands to see the first five rows of data sorted in descending order (i.e., from the largest to the lowest values) and using the Value column:

Here we will work with the Iris dataset.

```
print(iris.columns)
```

```
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class'],
dtype='object')
```

```
iris.sort_values(by = ['sepal_length'], ascending = False)
```

	sepal_length	sepal_width	petal_length	petal_width	class
131	7.9	3.8	6.4	2.0	Iris-virginica
135	7.7	3.0	6.1	2.3	Iris-virginica
122	7.7	2.8	6.7	2.0	Iris-virginica
117	7.7	3.8	6.7	2.2	Iris-virginica
118	7.7	2.6	6.9	2.3	Iris-virginica
...	...	...	...	...	...
41	4.5	2.3	1.3	0.3	Iris-setosa
42	4.4	3.2	1.3	0.2	Iris-setosa
38	4.4	3.0	1.3	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
13	4.3	3.0	1.1	0.1	Iris-setosa

150 rows × 5 columns

The following command sorts the sepal_lenghth column followed by the petal_length column.

```
iris.sort_values(by = ['sepal_length','petal_length'], ascending = False)
```

	sepal_length	sepal_width	petal_length	petal_width	class
131	7.9	3.8	6.4	2.0	Iris-virginica
118	7.7	2.6	6.9	2.3	Iris-virginica
117	7.7	3.8	6.7	2.2	Iris-virginica
122	7.7	2.8	6.7	2.0	Iris-virginica
135	7.7	3.0	6.1	2.3	Iris-virginica
...	...	...	...	...	...
41	4.5	2.3	1.3	0.3	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
38	4.4	3.0	1.3	0.2	Iris-setosa
42	4.4	3.2	1.3	0.2	Iris-setosa
13	4.3	3.0	1.1	0.1	Iris-setosa

150 rows × 5 columns

Note

1. The `inplace` keyword indicates that the DataFrame will be overwritten, hence no new DataFrame will be returned.
2. When `ascending = True` is used instead of `ascending = False`, the values are sorted in ascending order (i.e., from the smallest to the largest values).
3. If we wish to restore the original order, we can use the `sort_index` method and sort by an index.

```
# restore the original order

iris.sort_index(axis = 0, ascending = True, inplace = True)
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

```
# Importing sales dataset for the next section

filepath = '/Users/Kaemyuijang/SCMA248/Data/SalesData.csv'

# pd.read_csv(filepath, header = None, skiprows = 1, names =
['Trans_no','Name','Date','Product','Units','Dollars','Location'])

# header = 0 means that the first row to use as the column names
sales = pd.read_csv(filepath, header = 0,  index_col = 0)

# https://stackoverflow.com/questions/25015711/time-data-does-not-match-format
sales['Date'] =  pd.to_datetime(sales['Date'], format='%d/%m/%Y')

sales.head()
```

Trans_Number	Name	Date	Product	Units	Dollars	Location
1	Betsy	2004-04-01	lip gloss	45	137.20	south
2	Hallagan	2004-03-10	foundation	50	152.01	midwest
3	Ashley	2005-02-25	lipstick	9	28.72	midwest
4	Hallagan	2006-05-22	lip gloss	55	167.08	west
5	Zaret	2004-06-17	lip gloss	43	130.60	midwest

3.4.4. Grouping Data

Another useful method for inspecting data is to group it according to certain criteria. For the sales dataset, it would be useful to categorize all of the data by location, independent of the year. The `groupby` function in Pandas allows us to accomplish just that. This function returns a special grouped DataFrame as a result. As a result, an aggregate function must be used to create a suitable DataFrame. As a result, **all values in the same group will be subjected to this function**.

For instance, in our scenario, we can get a DataFrame that shows the number (count) of the transactions for each location across all years by grouping by location and using the count function as the aggregation technique for each group. As a consequence, a DataFrame with locations as indexes and counting values of transactions as the column would be created:

```
print(type(sales[['Location','Dollars']].groupby('Location')))

sales[['Location','Dollars']].groupby('Location').count()
```

```
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

Location	Dollars
east	456
midwest	424
south	521
west	490

```
# a DataFrame with locations as indexes and mean of sales income  as
# the column would be created

sales[['Location','Dollars']].groupby('Location').mean()
```

Location	Dollars
east	125.815965
midwest	129.258113
south	123.409616
west	129.467082

```
sales[['Location','Dollars','Units']].groupby('Location').mean()
```

Location	Dollars	Units
east	125.815965	41.267544
midwest	129.258113	42.417453
south	123.409616	40.466411
west	129.467082	42.491837

3.4.5. Rearranging Data

Until now, our indices were merely a numerical representation of rows with little meaning. We can change the way our data is organized by redistributing indexes and columns for better data processing, which usually results in greater performance. Using the `pivot_table` function, we may rearrange our data. We can define which columns will be the new indexes, values, and columns.

3.4.5.1. Simplest Pivot table

An `index` is required even for the smallest pivot table. Let us utilize the location as our index in this case. It uses the `'mean'` aggregation function on all available numerical columns by default.

```
sales.pivot_table(index='Location')
```

Location	Dollars	Units
east	125.815965	41.267544
midwest	129.258113	42.417453
south	123.409616	40.466411
west	129.467082	42.491837

To display multiple indexes, we can pass a list to index:

```
sales.pivot_table(index=['Location','Product'])
```

Location	Product	Dollars	Units
east	**eye liner**	132.521304	43.513043
	foundation	120.755248	39.584158
	lip gloss	119.001134	38.989691
	lipstick	136.096279	44.697674
	mascara	125.406000	41.120000
midwest	**eye liner**	127.295980	41.745098
	foundation	139.133333	45.712644
	lip gloss	130.765316	42.936709
	lipstick	136.465472	44.811321
	mascara	117.995340	38.669903
south	**eye liner**	121.481947	39.831858
	foundation	121.983852	39.977778
	lip gloss	121.492632	39.824561
	lipstick	115.709273	37.909091
	mascara	133.528462	43.846154
west	**eye liner**	136.215000	44.735849
	foundation	137.521553	45.194175
	lip gloss	115.428197	37.795082
	lipstick	132.699643	43.589286
	mascara	129.339223	42.446602

On the pivot table, the values to index are the keys to group by. To achieve a different visual representation, you can change the order of the values. For example, we can look at average values by grouping the region with the product category.

```
sales.pivot_table(index=['Product','Location'])
```

		Dollars	Units
Product	**Location**		
eye liner	**east**	132.521304	43.513043
	midwest	127.295980	41.745098
	south	121.481947	39.831858
	west	136.215000	44.735849
foundation	**east**	120.755248	39.584158
	midwest	139.133333	45.712644
	south	121.983852	39.977778
	west	137.521553	45.194175
lip gloss	**east**	119.001134	38.989691
	midwest	130.765316	42.936709
	south	121.492632	39.824561
	west	115.428197	37.795082
lipstick	**east**	136.096279	44.697674
	midwest	136.465472	44.811321
	south	115.709273	37.909091
	west	132.699643	43.589286
mascara	**east**	125.406000	41.120000
	midwest	117.995340	38.669903
	south	133.528462	43.846154
	west	129.339223	42.446602

3.4.5.2. Specifying values and performing aggregation

The mean aggregation function is applied to all numerical columns by default, and the result is returned. Use the `values` argument to specify the columns we are interested in.

```
sales.pivot_table(index=['Location'],
                          values = ['Dollars'])
```

	Dollars
Location	
east	125.815965
midwest	129.258113
south	123.409616
west	129.467082

We can specify a valid string function to `aggfunc` to perform an aggregation other than mean, for example, a sum:

```
sales.pivot_table(index=['Location'],
                          values = ['Dollars'], aggfunc = 'sum')
```

	Dollars
Location	
east	57372.08
midwest	54805.44
south	64296.41
west	63438.87

`aggfunc` can be a dict, and below is the dict equivalent.

```
sales.pivot_table(index=['Location'],
                        values = ['Dollars'], aggfunc = {'Dollars': 'sum'})
```

	Dollars
Location	
east	57372.08
midwest	54805.44
south	64296.41
west	63438.87

`aggfunc` can be a list of functions, and below is an example to display the `sum` and `count`

```
sales.pivot_table(index=['Location'],
                        values = ['Dollars'], aggfunc = ['sum','count'])
```

	sum	count
	Dollars	Dollars
Location		
east	57372.08	456
midwest	54805.44	424
south	64296.41	521
west	63438.87	490

3.4.5.3. Seeing break down using columns

If we would like to see sales broken down by product_category, the columns argument allows us to do that

```
sales.pivot_table(index=['Location'],
                  values = ['Dollars'],
                  aggfunc = 'sum',
                 columns='Product')
```

	Dollars				
Product	**eye liner**	**foundation**	**lip gloss**	**lipstick**	**mascara**
Location					
east	15239.95	12196.28	11543.11	5852.14	12540.60
midwest	12984.19	12104.60	10330.46	7232.67	12153.52
south	13727.46	16467.82	13850.16	6364.01	13886.96
west	14438.79	14164.72	14082.24	7431.18	13321.94

Note If there are missing values and we want to replace them, we could use `fill_value` argument, for example, to set `NaN` to a specific value.

```
sales.pivot_table(index=['Location'],
                  values = ['Dollars'],
                  aggfunc = 'sum',
                 columns='Product',
                 fill_value = 0)
```

	Dollars				
Product	**eye liner**	**foundation**	**lip gloss**	**lipstick**	**mascara**
Location					
east	15239.95	12196.28	11543.11	5852.14	12540.60
midwest	12984.19	12104.60	10330.46	7232.67	12153.52
south	13727.46	16467.82	13850.16	6364.01	13886.96
west	14438.79	14164.72	14082.24	7431.18	13321.94

3.4.6. Exercise

Apply `pivot_table` (in new worksheets) to answer the following questions.

1. The number of sales transactions for the given salesperson
2. For the given salesperson, the total revenue by the given product
3. Total revenue generated by the given salesperson broken down by the given location
4. Total revenue by the salesperson and the given year

3.4.7. Solutions to Exercise

1. The number of sales transactions for the given salesperson

```
sales.columns
```

```
Index(['Name', 'Date', 'Product', 'Units', 'Dollars', 'Location'], dtype='object')
```

```
sales.pivot_table(index=['Name'],
                  values=['Dollars'], aggfunc='count')
```

	Dollars
Name	
Ashley	197
Betsy	217
Cici	230
Colleen	206
Cristina	207
Emilee	203
Hallagan	200
Jen	217
Zaret	214

```
sales.pivot_table(index=['Name'],
                  values=['Dollars'], aggfunc = 'count').sort_values(by = 'Dollars')
```

	Dollars
Name	
Ashley	197
Hallagan	200
Emilee	203
Colleen	206
Cristina	207
Zaret	214
Betsy	217
Jen	217
Cici	230

1. For the given salesperson, the total revenue by the given product

```
sales.pivot_table(index='Name',
                  values='Dollars',
                   columns = 'Product', aggfunc = 'sum',  margins=True)
```

Product	eye liner	foundation	lip gloss	lipstick	mascara	All
Name						
Ashley	5844.95	4186.06	6053.67	3245.46	6617.10	25947.24
Betsy	6046.51	8043.48	5675.65	3968.62	4827.22	28561.48
Cici	5982.83	6198.22	5199.96	3148.83	7060.71	27590.55
Colleen	3389.61	6834.76	5573.35	2346.39	6746.54	24890.65
Cristina	5397.30	5290.97	5298.00	2401.68	5461.64	23849.59
Emilee	7587.37	5313.82	5270.26	2189.15	4719.34	25079.94
Hallagan	6964.64	6985.80	5603.10	3177.88	5703.32	28434.74
Jen	7010.45	5628.65	5461.65	3953.28	6887.21	28941.24
Zaret	8166.73	6451.66	5670.33	2448.71	3879.94	26617.37
All	56390.39	54933.42	49805.97	26880.00	51903.02	239912.80

The result above also show the total. In Panda pivot_table(), we can simply pass `margins=True`.

1. Total revenue generated by the given salesperson broken down by the given location

```
sales.pivot_table(index=['Name','Location'],
                  values='Dollars', aggfunc = 'sum').head(10)
```

		Dollars
Name	**Location**	
Ashley	**east**	7772.70
	midwest	4985.90
	south	7398.58
	west	5790.06
Betsy	**east**	8767.41
	midwest	4878.07
	south	7732.04
	west	7183.96
Cici	**east**	5956.31
	midwest	8129.60

1. Total revenue by the salesperson and the given year.

To generate a yearly report with Panda pivot_table(), here are the steps:

1. Defines a groupby instruction using Grouper() with key='Date' and freq='Y'.
2. Applies pivot_table.

Note To group our data depending on the specified frequency for the specified column, we'll use `pd.Grouper(key=INPUT COLUMN>, freq=DESIRED FREQUENCY>)`. The frequency in our situation is 'Y' and the relevant column is 'Date.'

Different standard frequencies, such as 'D','W','M', or 'Q', can be used instead of 'Y.' Check out the following for a list of less common useable frequencies, https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases.

```
year_gp = pd.Grouper(key='Date',freq='Y')
print(year_gp)

sales.pivot_table(index='Name', columns=year_gp, values='Dollars',aggfunc='sum')
```

```
TimeGrouper(key='Date', freq=<YearEnd: month=12>, axis=0, sort=True, closed='right',
label='right', how='mean', convention='e', origin='start_day')
```

Date	2004-12-31	2005-12-31	2006-12-31
Name			
Ashley	9495.09	9547.53	6904.62
Betsy	9420.24	9788.70	9352.54
Cici	8965.25	9024.96	9600.34
Colleen	9361.37	7996.81	7532.47
Cristina	9132.12	7976.34	6741.13
Emilee	7805.66	9326.44	7947.84
Hallagan	10676.88	9102.52	8655.34
Jen	9049.31	8920.30	10971.63
Zaret	9078.51	8639.68	8899.18

Exercise

Apply pivot_table (in new worksheets) to answer the following questions.

1. How many saleperson are there? Hint: use `groupby` to create a grouped DataFrame grouped by salepersons and then call `groups` on the grouped object, which will returns the list of indices for every group.

```
grouped_person = sales.groupby('Name')
```

```
print(grouped_person.groups.keys())
len(grouped_person.groups.keys())
```

```
dict_keys(['Ashley', 'Betsy', 'Cici', 'Colleen', 'Cristina', 'Emilee', 'Hallagan', 'Jen',
'Zaret'])
```

```
9
```

```
grouped_person.size()
```

```
Name
Ashley      197
Betsy       217
Cici        230
Colleen     206
Cristina    207
Emilee      203
Hallagan    200
Jen         217
Zaret       214
dtype: int64
```

Previous
2. Python Basics

Next
4. Data Visualization

By Pairote Satiracoo

© Copyright 2021.

4 Data Visualization

4. Data Visualization

In this section, you will learn how to use Python to create your first data visualization. The phrase "a picture is worth a thousand words" comes to mind. Data visualization is a valuable tool for communicating a data-driven conclusion. In certain circumstances, the visualization is so persuasive that there is no need for further investigation.

Python's most powerful feature is probably exploratory data visualization. With a unique blend of flexibility and ease, one may go from idea to data to plot quickly.

Excel may be easier than Python, but it lacks the flexibility of Python. Although D3.js is more powerful and flexible than Python, it takes substantially longer to create a plot.

There are many Python libraries that can do so:

- pandas
- matplotlib
- seaborn
- plotnine

We will work with the claims dataset from a Local Government Property Insurance Fund.

At its most basic level, insurance companies receive **premiums** in exchange for promises to compensate policyholders if an insured event occurs. The compensation provided by the insurer for incurred injury, loss, or damage that is covered by the policy. A claim is another term for this compensation. The extent of the payout, known as the **claim severity**, is a significant financial expenditure for an insurance.

In terms of financial outlay, an insurer does not always concern if there are ten claims of THB10,000 or one claim of THB100,000. Insurers, on the other hand, frequently monitor how frequently claims occur, which is known as **claim frequency**.

One of the important tasks of insurance analysts is to develop models to represent and manage the two outcome variables, **frequency** and **severity**. In this chapter, we will learn how to

```
import numpy as np
import pandas as pd

# Add this line so you can plot your charts into your Jupyter Notebook.
%matplotlib inline
```

```
claims = pd.read_csv('/Users/Kaemyuijang/SCMA248/Data/claimsNewFormat.csv')

# New format dataframe
# url2 = 'https://raw.githubusercontent.com/pairote-
sat/SCMA248/main/Data/claimsNewFormat.csv'
# claims = pd.read_csv(url2)
```

4.1. Using pandas to plot line, bar charts and Histogram in Python

To give you an example, the property fund had 1,110 policyholders in 2010 who had a total of 1,377 claims.

First we will create a table reporting the **2010 claims frequency distribution**.

Note: In a dataset, a histogram displays the amount of occurrences of certain values. It looks a lot like a bar chart at first glance.

First we will explore how to make a line chart and a bar chart that indicate the frequency of different values in the data set before plotting the histogram. As a result, you will be able to compare the various ways.

Contents

Print to PDF

Of course, if you have never plotted anything in Pandas before, it is a good idea to start with a simple line chart.

Simply type the `plot` method after the pandas data frame you wish to visualize to put your data on a chart. By default `plot` provides a line chart.

If you plot the `Freq` column of the claims2010 data frame as is, you will get the following results.

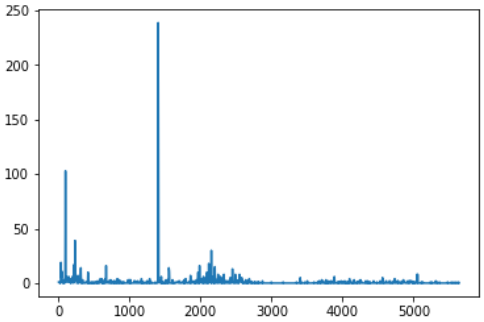
```
claims2010 = claims[(claims['Year']==2010)]
```

```
claims2010['Freq'].head()
```

```
4     1
9     1
14    1
19    0
24    1
Name: Freq, dtype: int64
```

```
claims2010['Freq'].plot()
```

```
<AxesSubplot:>
```



We will have to work with the original dataset a little further to accomplish what we want (plot the occurrence of each unique value in the dataset). Let uss combine a `.groupby()` aggregate function with a `.count()` aggregate function.

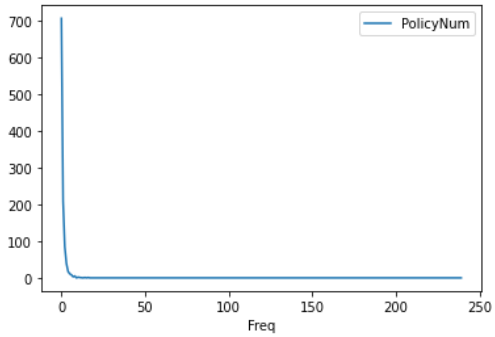
```
FreqDist = claims2010.groupby('Freq').count()[['PolicyNum']]

FreqDist.head()
```

	PolicyNum
Freq	
0	707
1	209
2	86
3	40
4	18

If you plot the output of this, you'll get a much nicer line chart.

```
FreqDist.plot()
```

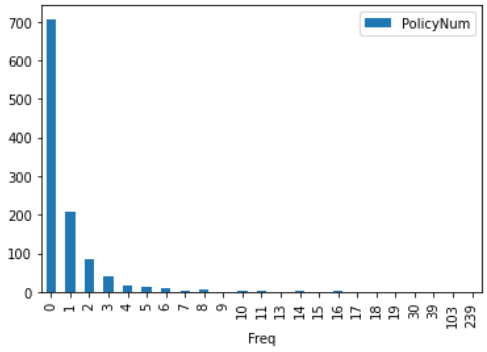
```
<AxesSubplot:xlabel='Freq'>
```

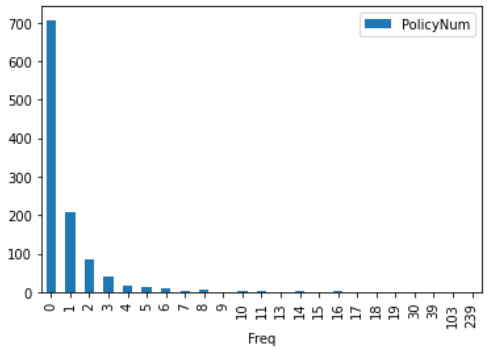


To turn your line chart into a bar chart, just add the bar keyword. Here we simply count the unique values in the dataset and put that on a bar chart

```
FreqDist.plot(kind='bar')

FreqDist.plot.bar()
```

```
<AxesSubplot:xlabel='Freq'>
```





However, there is one more step before plotting a histogram: these unique values will be clustered into ranges. Bins or buckets are the names for these ranges, and the default number of bins in Python is 10. As a result of the grouping, your histogram should now look like this:

We will use the `hist` function that is built into pandas. It is very simple to turn your pandas data frame into a histogram once you have the values in it.

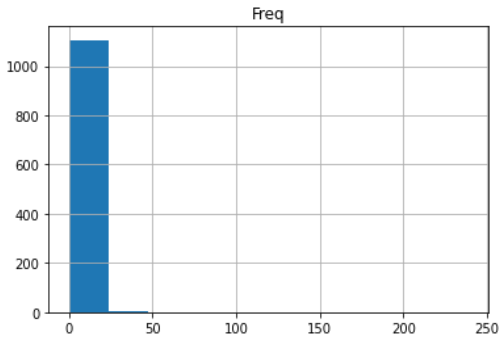
The `hist` function automatically does several of the useful tasks:

1. It does the grouping.
2. It does the counting.
3. It plots a histogram for each column in your data frame that has numerical values in it.

Note: You can change the number of bins/buckets from the default of 10 by setting a parameter, for example `bins=20`.


```
claims2010[['Freq']].hist()
```

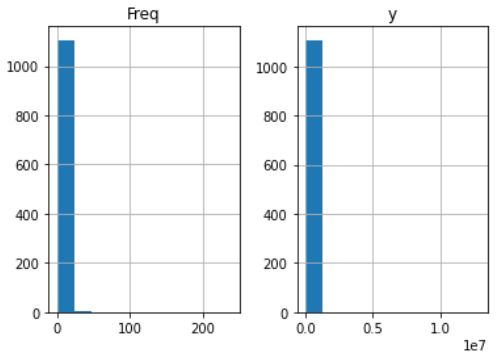
```
array([[<AxesSubplot:title={'center':'Freq'}>]], dtype=object)
```



Note one of an important tasks is to find a model (perhaps a statistical distribution) that adequately fits this claim frequency distribution)

```
claims2010[['Freq','y']].hist()
```

```
array([[<AxesSubplot:title={'center':'Freq'}>,
        <AxesSubplot:title={'center':'y'}>]], dtype=object)
```



Next we will focus on claim severity.

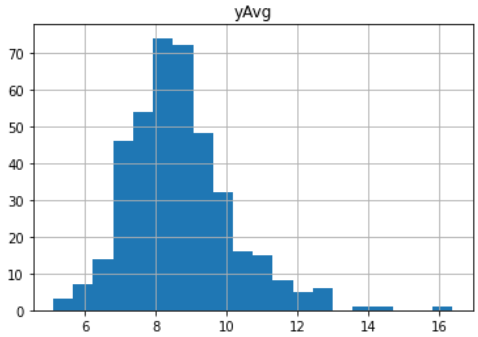
A common method for determining the severity distribution is to look at the distribution of the sample of 1,377 claims. Another common approach is to look at **the distribution of average claims among policyholders who have made claims**. There were 403 (=1110-707) such policyholders in our 2010 sample.

Exercise

1. Create a table that summarizes the sample distribution of average severities from the 403 policyholders who made a claim.
2. Use `hist` to plot the distribution of positive average severities. What conclusion can we draw from the plot?
3. Plot the distribution of positive average severities in logarithmic units

```
claims2010 = claims[(claims['Year']==2010) &  (claims['Freq'] >= 1)]
output = claims2010[['yAvg']]

output.apply(np.log).hist(bins=20)
#claims2010['lnyAvg'] = claims2010['yAvg'].apply(np.log)
#claims2010['lnyAvg'] = np.log(claims2010['yAvg'])
```

```
array([[<AxesSubplot:title={'center':'yAvg'}>]], dtype=object)
```



Note Examples of various plots from the widely-used packages can be seen on the following links. The pages provide access to the entire picture and source code for any image.

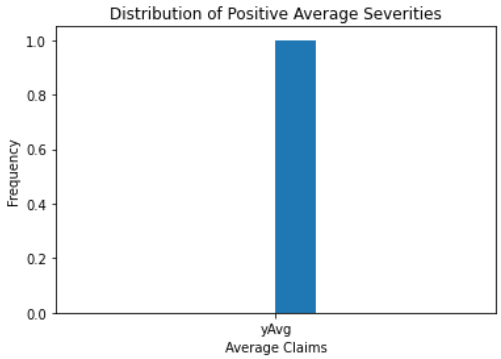
- matplotlib https://matplotlib.org/stable/gallery/index.html
- seaborn https://seaborn.pydata.org/examples/index.html

4.2. Plot a Histogram in Python using Matplotlib

To plot a histogram in Python using Matplotlib, use the following command:

```
claims2010 = claims[(claims['Year']==2010) &  (claims['Freq'] >= 1)]
output = claims2010[['yAvg']]

import matplotlib.pyplot as plt
plt.hist(output, bins = 10)
plt.xlabel('Average Claims')
plt.ylabel('Frequency')
plt.title('Distribution of Positive Average Severities')
```

```
Text(0.5, 1.0, 'Distribution of Positive Average Severities')
```



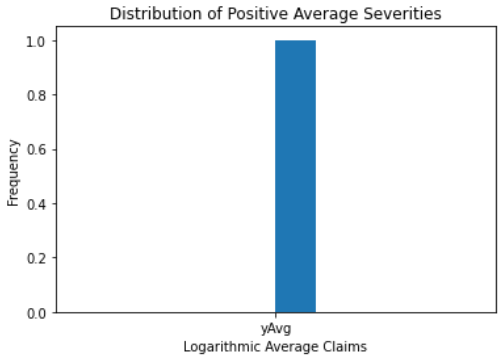
The image above shows more information about the sample claim distribution, revealing a distribution that is dominated by this single large claim, making the histogram useless.

Even when the large claim is removed, the distribution remains skewed to the right. Working with claims in logarithmic units, especially for graphical reasons, is a widely acknowledged practice.

The following figure in the right-hand panel is considerably easier to read.

```
plt.hist(np.log(output), bins = 10)
plt.xlabel('Logarithmic Average Claims')
plt.ylabel('Frequency')
plt.title('Distribution of Positive Average Severities')
```

```
Text(0.5, 1.0, 'Distribution of Positive Average Severities')
```



4.3. Using ggplot and Python to Create Your First Plot

You will learn how to construct data visualizations using **ggplot** in Python using a **grammar of graphics**. A grammar of graphics is a high-level tool for creating data displays in a consistent and efficient manner. It abstracts the majority of low-level details, **allowing you to concentrate on producing useful and beautiful data visualizations**.

A graphics grammar is provided by a number of Python libraries. We focus on **plotnine** since it is the most developed.

Plotnine is based on the R programming language's **ggplot2**, thus if you are familiar with R, you may think of plotnine as the Python counterpart of ggplot2.

The first example of plotnine being used to plot the distribution of positive average severities is seen below.

```
# We add the logarithms of claim average into the claims dataset.

claims['lnyAvg'] = np.log(claims[claims['Freq']>=1]['yAvg'])
```

```
claims2010 = claims[(claims['Year']==2010) &  (claims['Freq'] >= 1)]
```

```
from plotnine import *

(
    ggplot(claims2010)  # What data to use
    + aes(x='lnyAvg')  # What variable to use
    + geom_histogram(bins = 20)  # Geometric object to use for drawing
)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/var/folders/kl/h_r05n_j76n32kt0dwy7kynw0000gn/T/ipykernel_1954/476494641.py in <module>
----> 1 from plotnine import *
      2
      3 (
      4     ggplot(claims2010)  # What data to use
      5     + aes(x='lnyAvg')  # What variable to use

ModuleNotFoundError: No module named 'plotnine'
```

4.3.1. Building Your First Plot With ggplot and Python

You will learn how to use ggplot in Python to create your first data visualization.

The approach of making visuals with ggplot2 is intuitively based on **The Grammar of Graphics** (where the gg comes from).

This is comparable to how understanding grammar can help a beginner build hundreds of distinct sentences without having to memorize each one.

We will be able to make hundreds of different plots using ggplot2 by combining three basic components:

- **a data set**,
- **geoms** are visual marks that represent data points. There are several geometries including
  - scatterplot,
  - barplot,
  - histogram,
  - smooth densities,
  - qqplot, and
  - boxplot.
- **aesthetic mapping** describe how properties of the data (variables) connect with features of the graph, such as distance along an axis, size, or colour.

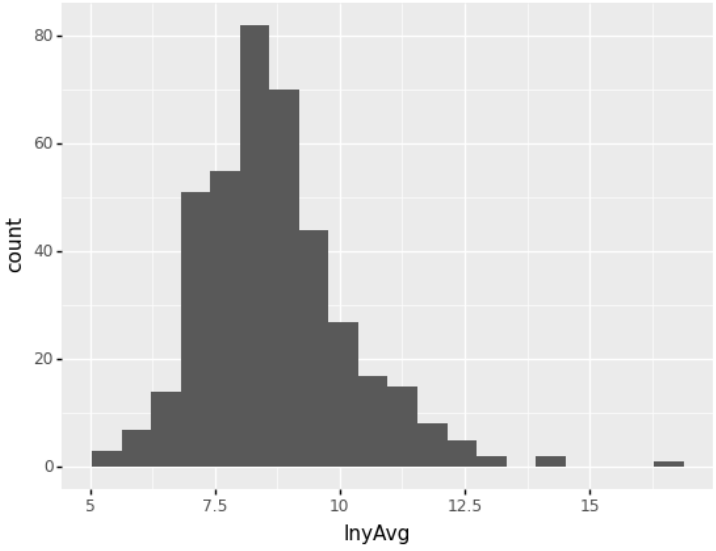
Visit the following website for additional information on plotnine and gallery:

https://plotnine.readthedocs.io/en/stable/

The claims dataset is plotted in this short code sample. Here's a brief breakdown of what's going on:

- Line 1: You use ggplot() to build a plot object and feed the claims data frame to the constructor.
- Line 2: The variable to utilize for each axis, in this case **lnyAvg**, is set with aes().
- Line 3: You use the geom_histogram() function to tell the chart to draw as a histogram.

```
(
    ggplot(claims2010)  # What data to use
    + aes(x='lnyAvg')  # What variable to use
    + geom_histogram(bins = 20)  # Geometric object to use
)
```




```
<ggplot: (311511861)>
```

You'll also learn about the following optional components:

1. **Statistical transformations** define computations and aggregations to be made to it before plotting data.
2. During the mapping from data to aesthetics, **scales** do some transformation. A logarithmic scale, for example, can be used to better represent specific elements of your data.
3. **Facets** let you divide data into groups depending on specific qualities, then plot each group in its own panel within the same visual.
4. The position of items is mapped to a 2D graphical place in the plot using **coordinate systems**. For example, if it makes more sense in the graphic you're creating, you can choose to reverse the vertical and horizontal axes.
5. **Themes** can be used to modify colors, fonts, and shapes are just a few of the visual aspects

Do not worry if you are not sure what each component means right now. Throughout this chapter, you will learn more about them.

4.3.1.1. Geometries

In plotnine we create graphs by adding layers. Layers can define geometries, compute summary statistics, define what scales to use, or even change styles.

To add layers, we use the symbol `+`. In general, a line of code will look like this:

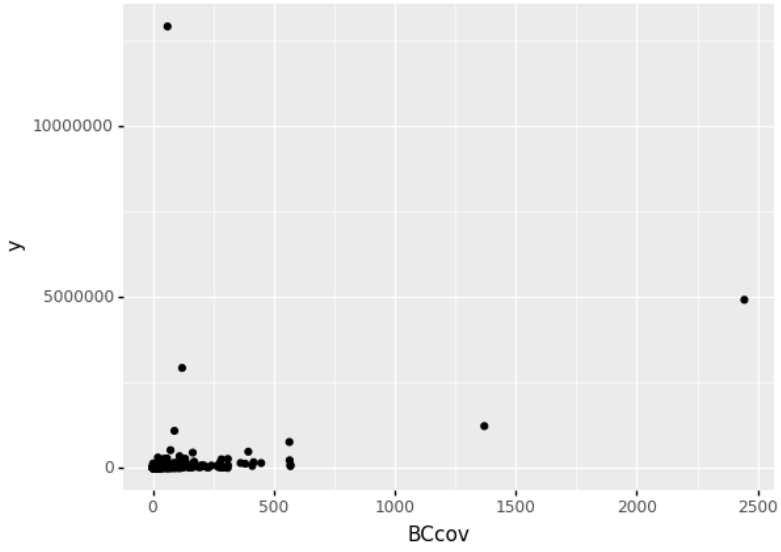
```
ggplot(data) + LAYER 1 + LAYER 2 + ... + LAYER N
```

Usually, the first added layer defines the geometry. For a scatterplot, the function used to create plots with this geometry is `geom_point`.

For `geom_point` to run properly we need to provide data and a mapping.

The figure below shows a scatter plot (relationship) between the coverage and the total claims.

```
claims = pd.read_csv('/Users/Kaemyuijang/SCMA248/Data/claimsNewFormat.csv')

# Here we express the coverage of building and contents in millions of dollars
claims['BCcov'] = claims['BCcov']/(10**6)
claims2010 = claims[(claims['Year']==2010) &  (claims['Freq'] >= 1)]

(
    ggplot(data = claims2010)  # What data to use
    + geom_point(aes(x = 'BCcov',y = 'y'))  # Geometric object to use for drawing
)
```




```
<ggplot: (311808725)>
```

4.3.1.2. Aesthetic mappings

The next step is to decide which variable you'll use for each axis in your graph. You must tell plotnine which variables you want to include in the visual because each row in a data frame can have several fields.

Data variables are mapped to graphical features such as 2D position and color by aesthetics.

In other words, aesthetic mappings describe how properties of the data (variables) connect with features of the graph, such as distance along an axis, size, or colour.

The `aes` function connects data with what we see on the graph by defining aesthetic mappings.

The code, for example, creates a graph with **BCcov** (coverages (in millions)) on the x-axis and **claims** on the y-axis:

4.3.1.3. Adding more arguments

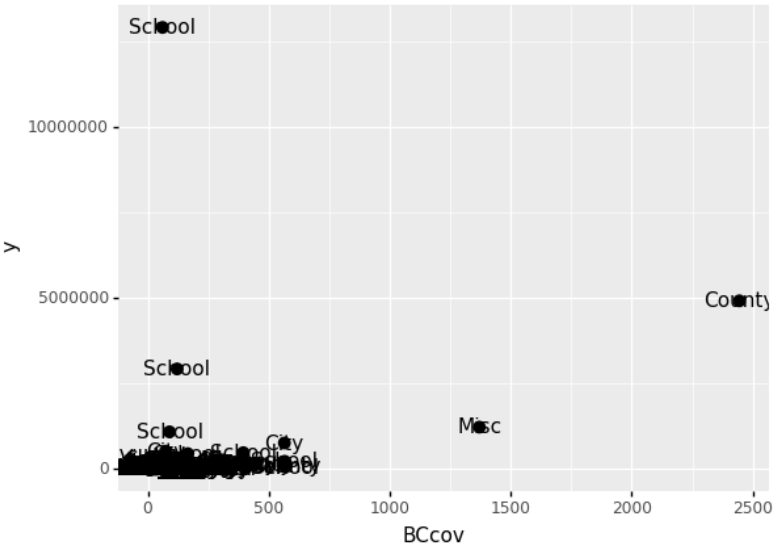
A second layer in the plot we wish to make involves adding a label to each point to identify the entity type.

The `geom_label` and `geom_text` functions permit us to add text to the plot **with** and **without** a rectangle behind the text, respectively.

Because each point (each claim in this case) has a label (type of local government entities), we need an aesthetic mapping to make the connection between points and labels.

By reading the help file, we learn that we supply the mapping between point and label through the label argument of `aes`. So the code looks like this:

```
(
    ggplot(data = claims2010)  # What data to use
    + geom_point(aes(x = 'BCcov',y = 'y'), size = 3)  # Geometric object to use for
drawing
    + geom_text(aes(x = 'BCcov', y = 'y', label = 'Type'))
)
```

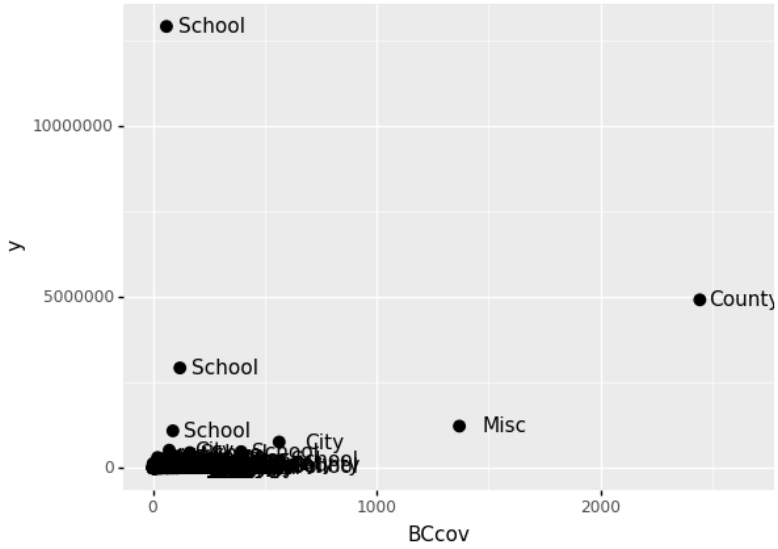



```
<ggplot: (303583217)>
```

`size` is not a mapping: whereas mappings use data from specific observations and need to be inside `aes()`, operations we want to affect all the points the same way do not need to be included inside `aes`.

If we read the help file for `geom_text`, we see the `nudge_x` argument, which moves the text slightly to the right or to the left:

```
(
    ggplot(data = claims2010)  # What data to use
    + geom_point(aes(x = 'BCcov',y = 'y'), size = 3)  # Geometric object to use for
drawing
    + geom_text(aes(x = 'BCcov', y = 'y', label = 'Type'), nudge_x = 200)
)
```




```
<ggplot: (303610161)>
```

4.4. Global and local aesthetic mappings

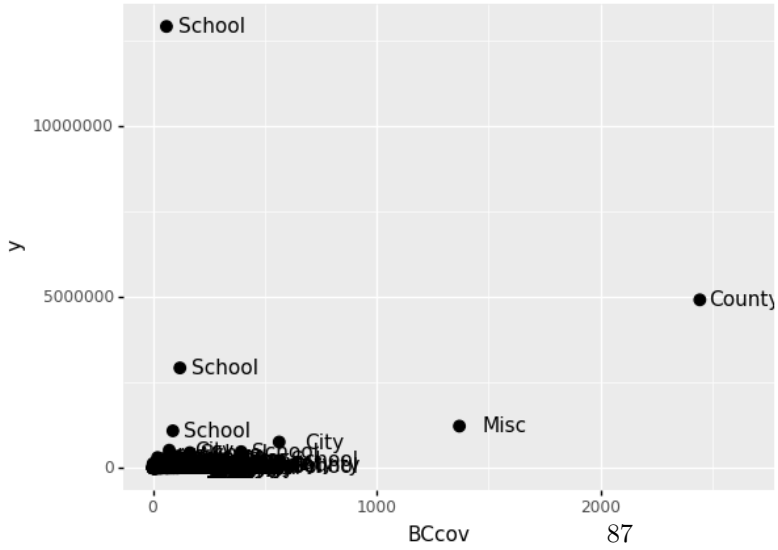
In the previous line of code, we define the mapping `aes(x = 'BCcov', y = 'y')` twice, once in each geometry.

We can avoid this by using a global aesthetic mapping.

We can do this when we define the blank slate ggplot object.

If we define a mapping in ggplot, all the geometries that are added as layers will default to this mapping. We redefine p:

```
p = ggplot(data = claims2010) + aes(x = 'BCcov', y = 'y', label = 'Type')

(
    p
    + geom_point(size = 3)
    + geom_text(nudge_x = 200)
)
```



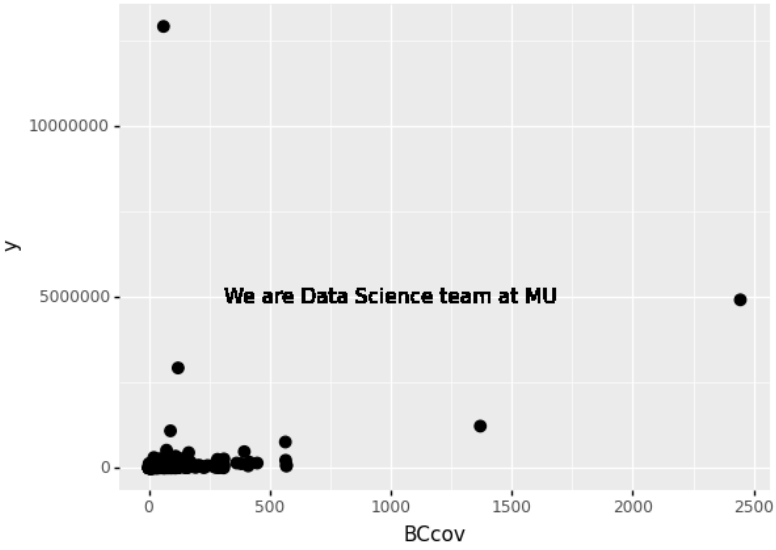

```
<ggplot: (312462793)>
```

In the code above, we keep the `size` and `nudge_x` arguments in `geom_point` and `geom_text`, respectively, because we want to only increase the size of points and only nudge the labels.

If we put those arguments in `aes` then they would apply to both plots. Also note that the `geom_point` function does not need a label argument and therefore ignores that aesthetic.

If necessary, we can override the global mapping by defining a new mapping within each layer. These local definitions override the global.

```
p = ggplot(data = claims2010) + aes(x = 'BCcov', y = 'y', label = 'Type')

(
    p
    + geom_point(size = 3)
    + geom_text(aes(x=1000, y = 5000000), label = 'We are Data Science team at MU')
)
```



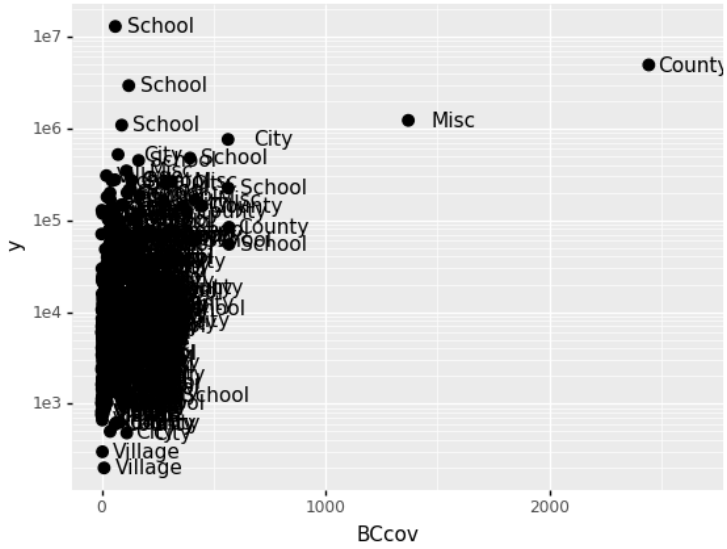

```
<ggplot: (312850877)>
```

4.4.1. Scales

First, our desired scale in claims is in log-scale. This is not the default, so this change needs to be added through a scales layer.

A quick look at the cheat sheet reveals the `scale_y_continuous` function lets us control the behavior of scale. We use it like this:

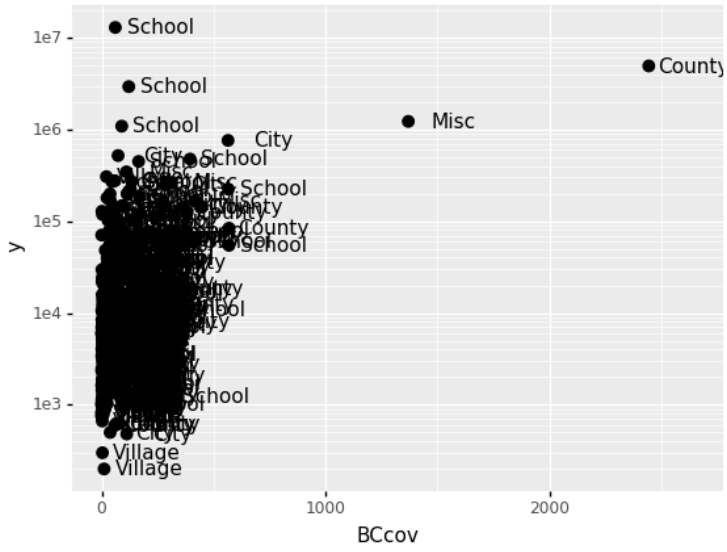
```
p = ggplot(data = claims2010) + aes(x = 'BCcov', y = 'y', label = 'Type')

(
    p
    + geom_point(size = 3)
    + geom_text(nudge_x = 200)
    + scale_y_continuous(trans = "log10")
)
```




```
<ggplot: (300296569)>
```

Alternative to the `scale_y_continuous`, we can use the function `scale_y_log10`, which we can use to rewrite the code like this:

```
(
    p
    + geom_point(size = 3)
    + geom_text(nudge_x = 200)
    + scale_y_log10()
)
```

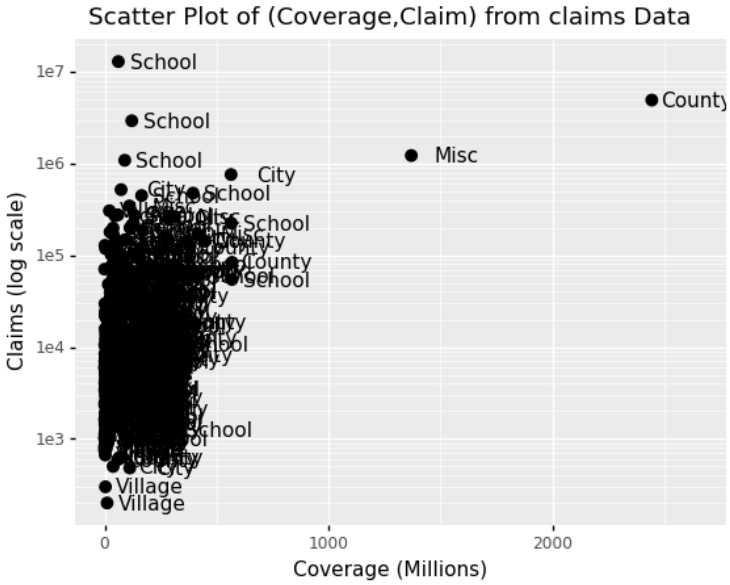



```
<ggplot: (313462709)>
```

4.5. Labels and titles

Similarly, the cheat sheet quickly reveals that to change labels and add a title, we use the following functions:

```
(
    p
    + geom_point(size = 3)
    + geom_text(nudge_x = 200)
    + scale_y_log10()
    + xlab("Coverage (Millions)")
    + ylab("Claims (log scale)")
    + ggtitle("Scatter Plot of (Coverage,Claim) from claims Data")
)
```




```
<ggplot: (313816497)>
```

4.6. Categories as colors

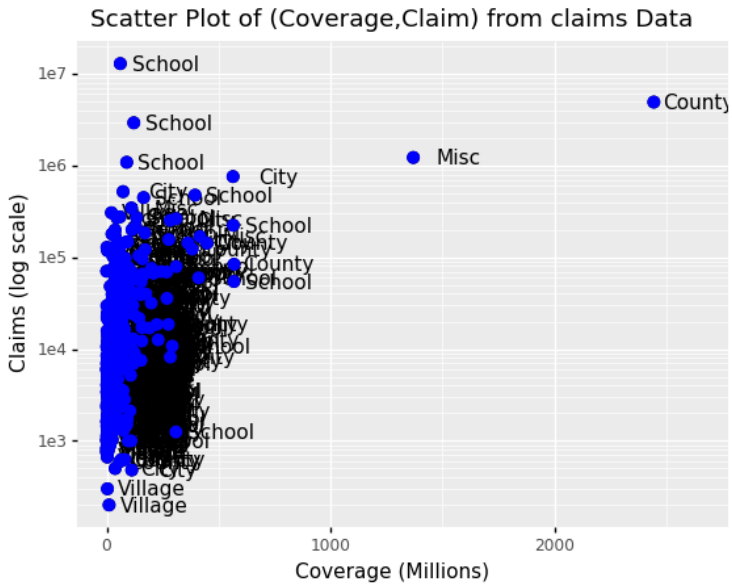
We can change the color of the points using the `colour` argument in the `geom_point` function.

To facilitate demonstration of new features, we will redefine `p` to be everything except the points layer:

```
p = ggplot(data = claims2010) + aes(x = 'BCcov', y = 'y', label = 'Type') +
geom_point(size = 3)  + geom_text(nudge_x = 200) + scale_y_log10() + xlab("Coverage
(Millions)")  + ylab("Claims (log scale)")  + ggtitle("Scatter Plot of (Coverage,Claim)
from claims Data")
```

and then test out what happens by adding different calls to `geom_point`. We can make all the points blue by adding the color argument:

```
(p + geom_point(size = 3, colour ="blue"))
```




```
<ggplot: (300294321)>
```

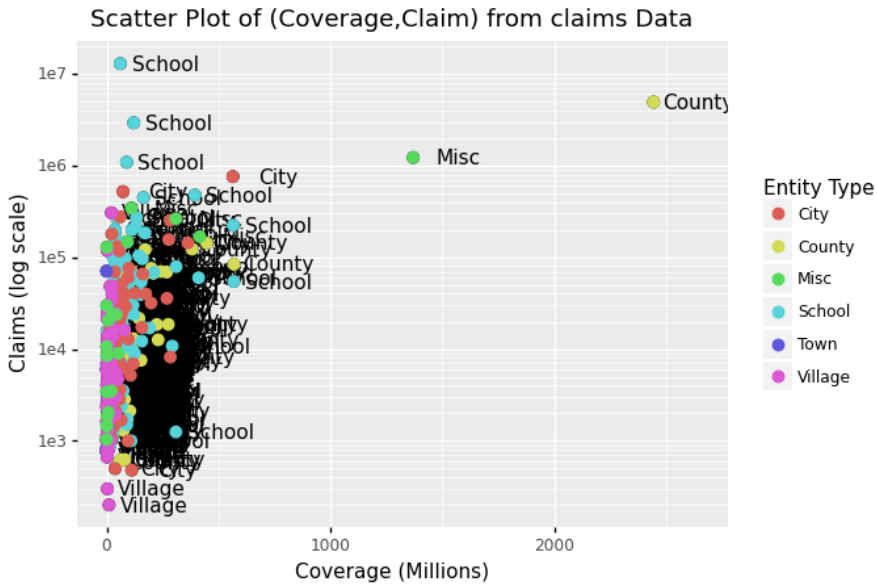
This, of course, is not what we want. We want to assign color depending on the entity type.

A nice default behavior of ggplot2 is that if we assign a **categorical** variable to colour, it automatically assigns a different color to each category and also adds a legend.

Since the choice of colour is determined by a feature of each observation, this is an aesthetic mapping.

To map each point to a color, we need to use `aes`. We use the following code:

```
(p + geom_point(aes(colour='Type'), size = 3)  + scale_color_discrete(name = "Entity
Type"))
```




```
<ggplot: (312026033)>
```

Comment: The above data visualization should provide insight into the relationship between claims and Coverage, and also claims and entity type.

4.7. Data visualization in practice

In this section, we will show how to use plotnine (ggplot2) to make plots that are both informative and visually appealing. We will use plots to help us better comprehend global health and economic patterns as motivation.

We will put what we learned in this chapter into practice and learn how to improve the plots by augmenting the code. We will go through general data visualization principles and ideas like faceting, time series plots, transformations, and ridge plots as we go through our case study.

Here is an example that uses an abstract from the Gapminder dataset, which became popular after **Hans Rosling's Ted Talk**. For more than 100 countries, it provides average life expectancy, GDP per capita, and population size.

In addition, according to the research result **Socioeconomic development and life expectancy relationship: evidence from the EU accession candidate countries** by Goran Miladinov in Journal of Population Sciences, Genus volume 76, Article number: 2 (2020), the results show that

- a country's population health and socioeconomic development have a significant impact on life expectancy at birth;
- in other words, as a country's population health and socioeconomic development improves, infant mortality rates decrease, and life expectancy at birth appears to rise.
- Through increased economic growth and development in a country, **GDP per capita raises life expectancy at birth, resulting in a longer lifespan**.

https://genus.springeropen.com/articles/10.1186/s41118-019-0071-0#:~:text=GDP per capita increases the,to the prolongation of longevity.)

4.7.1. Visualization with bubble plot

The link between world countries' life expectancy (y) and GDP per capita (x) is depicted by a bubble plot below. Each country's population is indicated by the size of its circles.

A **bubble plot** is a scatterplot with a third dimension: the size of the dots represents the value of an additional numeric variable. (source: data-to-viz).

As input, you will need three numerical variables: one for the X axis, one for the Y axis, and one for the dot size.

```
gapminder = pd.read_csv('/Users/Kaemyuijang/SCMA248/Data/gapminder_full.csv')

#url = 'https://raw.githubusercontent.com/STLinde/Anvendt-
Statistik/main/gapminder_full.csv'
#gapminder = pd.read_csv(url)
#gapminder['year'].unique()
```

Note: The gapminder data frames include six variables:

Variable	Meaning
country	
continent	
year	
life_exp	life expectancy at birth
population	total population
gdp_cap	per-capita GDP

The per-capita GDP is expressed in international dollars, which are "a hypothetical unit of money with the same purchasing power parity as the United States dollar at a specific point in time".

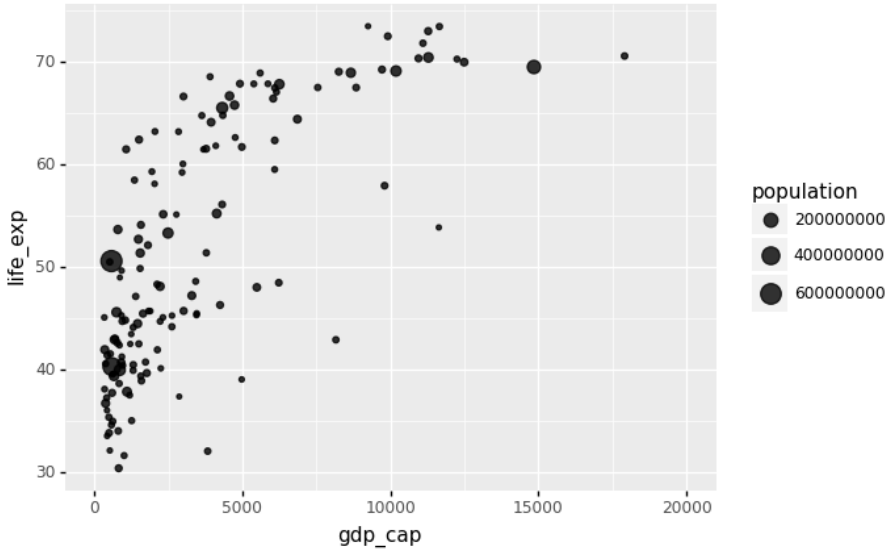
As the dataset covers the period between 1952 and 2007, we start by looking at 1957 about 50 years ago.

```
beginning_year = 1957
latest_year = gapminder['year'].max()

gapminder1957 = gapminder[gapminder['year']==beginning_year]
gapminder2007 = gapminder[gapminder['year']==latest_year]
```

The following Python code creates the most basic bubble chart using ggplot.

```
# With plotnine (ggplot2)

from plotnine import *

gapminder1957

p = ggplot(data = gapminder1957) + \
    aes(x = 'gdp_cap', y = 'life_exp', size = 'population') + \
    geom_point(alpha = 0.8)

(
p + xlim(0,20000)
)
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages/plotnine/layer.py:401:
PlotnineWarning: geom_point : Removed 1 rows containing missing values.
```




```
<ggplot: (312446985)>
```

The bubble size is the first thing we need to modify in the preceding chart. Using the range input, `scale_size()` allows you to change the size of the smallest and largest circles. It's worth noting that you can change the legend's name with your own.

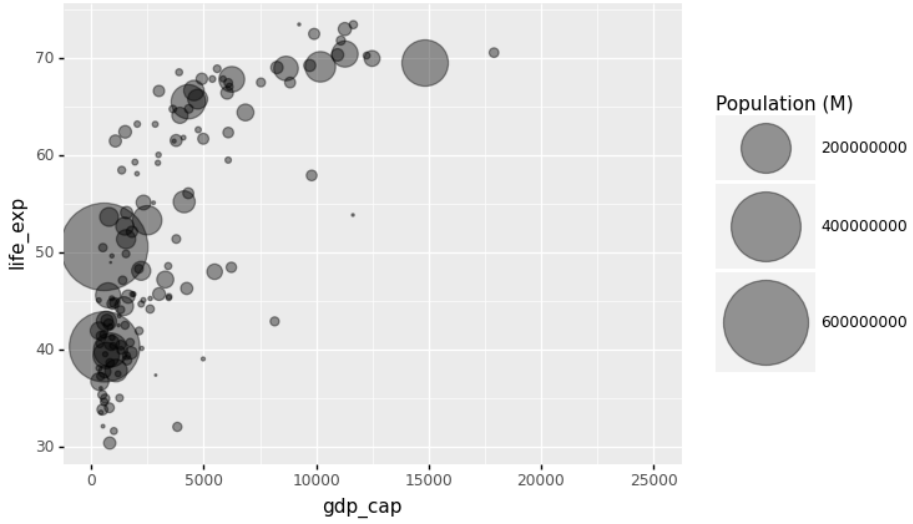
It is worth noting that circles frequently overlap. You must first sort your dataset, as shown in the code below, to avoid having large circles on top of the chart.

Note: Here we use `xlim()` to specify the left/lower limit and the right/upper limit of the scale.

```
data_sorted = gapminder1957.sort_values(by = ['population'], ascending = False)

p = ggplot(data = data_sorted) + aes(x = 'gdp_cap', y = 'life_exp', size = 'population')
+ geom_point(alpha = 0.4)

(
p + xlim(0,25000) + scale_size(range = (.1,28), name='Population (M)')
)
```



```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages/plotnine/layer.py:401:
PlotnineWarning: geom_point : Removed 1 rows containing missing values.
```




```
<ggplot: (314251261)>
```

Most points fall into **two distinct categories**:

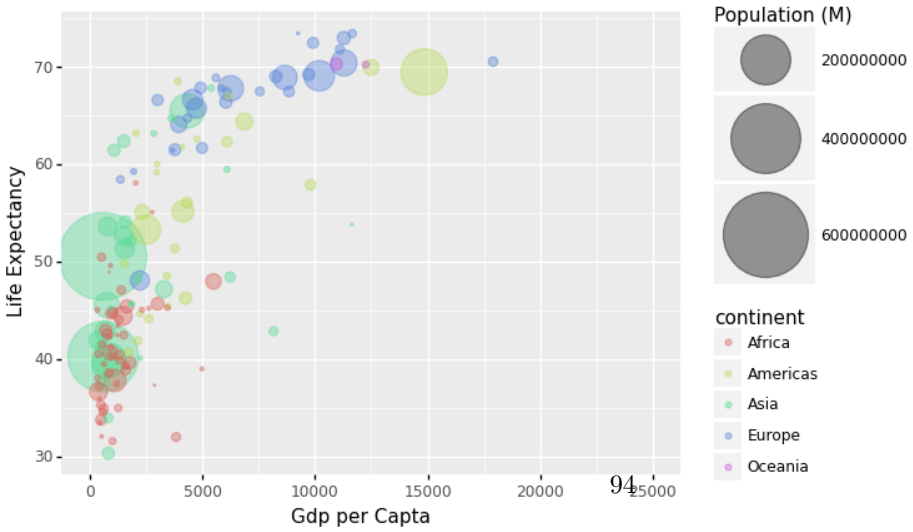
1. Life expectancy around 70 years and GDP per capita more than 5000.
2. Life expectancy lower than 55 years and GDP per capita lower than 2500.

We can utilize color to signify continent to check that these countries are really from the regions we expect.

Each country's continent is applied to control circle color in this example by adding `color='continent'`:

```
data_sorted = gapminder1957.sort_values(by = ['population'], ascending = False)

p = ggplot(data = data_sorted) + aes(x = 'gdp_cap', y = 'life_exp', size = 'population',
color='continent') + geom_point(alpha = 0.4)

(
p + xlim(0,25000) +
    scale_size(range = (.1,28), name='Population (M)') +
    xlab('Gdp per Capta') +
    ylab('Life Expectancy')
)
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages/plotnine/layer.py:401:
PlotnineWarning: geom_point : Removed 1 rows containing missing values.
```




```
<ggplot: (314401913)>
```

We may compare the results between 1957 and 2007 to see whether there is an improvement in life expectancy with GDP per capita.

To do this, we can simply repeat the whole process as above by replacing the data in 1957 with 2007.

Alternatively, we can use Pandas `pipe` function.

4.7.2. Pandas Pipe

We will briefly introduce the concept of **pipes (or method chaining)** in Python that allows us to make code more efficient and improve code readability.

Background In Data Processing, it is common to construct a function to perform operations (such as statistical calculations, splitting, or substituting values) on a certain row or column in order to get new data.

(source: https://towardsdatascience.com/using-pandas-pipe-function-to-improve-code-readability-96d66abfaf8)

For a typical code example below, we might save a result of a function in a variale, for example `res1`, and then pass it into the next function, `g`. As a result, you will have a lot of variables with potentially meaningless names, which will add to the complexity of your code.

```
# f(), g() and h() are user-defined function
# df is a Pandas DataFrame
res1 = h(df)
res2 = g(res1, arg1=1)
res3 = f(res2, arg2=b, arg3=c)
```

Alternative to the this approach, we can nest function calls as follows:

```
# f(), g() and h() are user-defined function
# df is a Pandas DataFrame
f(g(h(df), arg1=a), arg2=b, arg3=c)
```

Pipes will enable us to build code that offers the following advantages:

- arranging data operations in a left-to-right order (as opposed to from the inside and out),
- avoiding calls to nested functions
- reducing the number of local variables and function definitions required, and
- making it simple to add steps to the sequence of operations at any point.

For a better understanding how the piped code works the interpreted version is shown below: We can write with pipe as follows:

```
(
    df.pipe(h)
    .pipe(g, arg1=a)
    .pipe(f, arg2=b, arg3=c)
)
```

To make the code the way we want it, we'll use () brackets around it.

We also see that a pipe passes the results of one method (function) to another method (function):

- `df.pipe(h)` is the same as `h(df)`, which is then used as the input for another method, the method `g` in this case.
- `df.pipe(h).pipe(g, arg1=a)` results in `g(h(df), arg1=a)`. It is then passed to the function `f`.
- `df.pipe(h).pipe(g, arg1=a).pipe(f, arg2=b, arg3=c)` then produces the same final result as `f(g(h(df), arg1=a), arg2=b, arg3=c)`.

When applying numerous methods to a Python iterable, pipe makes code look clearer.

4.7.3. Applying Pandas pipes to gapminder

We will apply the concept of Pandas pipes to gapminder dataset. In the data visualization process above, we proceed in the following manner:

1. We select rows (observations) from year 1957 in the Gapminder dataset.
2. The result filtered data frame is used as the data for the `ggplot` function to produce the bubble plt.

Therefore, to apply pipes for this example, we first define a Python function (see the preceding code) that returns the subset of the data frame filtered by selecting rows with a specified year.

To select rows with the year 2007, we previously used the following code `gapminder[gapminder['year']==latest_year]`.

Using pipe together with our user-defined function `year_filter` produces the same results `year_filter(gapminder, latest_year)`, which will selects rows with observations in 2007.

Note The function `year_filter` is defined for illustrative purposes. Aalternatively, we can directly apply the pandas `query` method to select the specific observations (or rows). Details are given below.

```
def year_filter(df,year_selected):
    cond=df['year']==year_selected
    return df.loc[cond,]
```

```
gapminder.pipe(year_filter,latest_year)
```

	country	year	population	continent	life_exp	gdp_cap
11	Afghanistan	2007	31889923	Asia	43.828	974.580338
23	Albania	2007	3600523	Europe	76.423	5937.029526
35	Algeria	2007	33333216	Africa	72.301	6223.367465
47	Angola	2007	12420476	Africa	42.731	4797.231267
59	Argentina	2007	40301927	Americas	75.320	12779.379640
...	...	...	...	...	...	...
1655	Vietnam	2007	85262356	Asia	74.249	2441.576404
1667	West Bank and Gaza	2007	4018332	Asia	73.422	3025.349798
1679	Yemen, Rep.	2007	22211743	Asia	62.698	2280.769906
1691	Zambia	2007	11746035	Africa	42.384	1271.211593
1703	Zimbabwe	2007	12311143	Africa	43.487	469.709298

142 rows × 6 columns

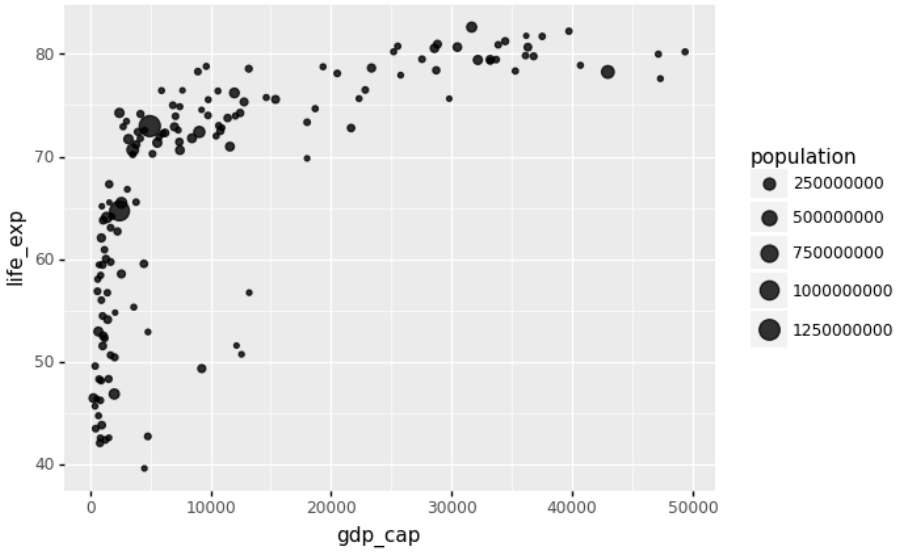
```
year_filter(gapminder, latest_year)
```

	country	year	population	continent	life_exp	gdp_cap
11	Afghanistan	2007	31889923	Asia	43.828	974.580338
23	Albania	2007	3600523	Europe	76.423	5937.029526
35	Algeria	2007	33333216	Africa	72.301	6223.367465
47	Angola	2007	12420476	Africa	42.731	4797.231267
59	Argentina	2007	40301927	Americas	75.320	12779.379640
...	...	...	...	...	...	...
1655	Vietnam	2007	85262356	Asia	74.249	2441.576404
1667	West Bank and Gaza	2007	4018332	Asia	73.422	3025.349798
1679	Yemen, Rep.	2007	22211743	Asia	62.698	2280.769906
1691	Zambia	2007	11746035	Africa	42.384	1271.211593
1703	Zimbabwe	2007	12311143	Africa	43.487	469.709298

142 rows × 6 columns

Now we will proceed to our next step to use the current filtered data frame to create a bubble plot with ggplot. The following command with pipe completes our task:

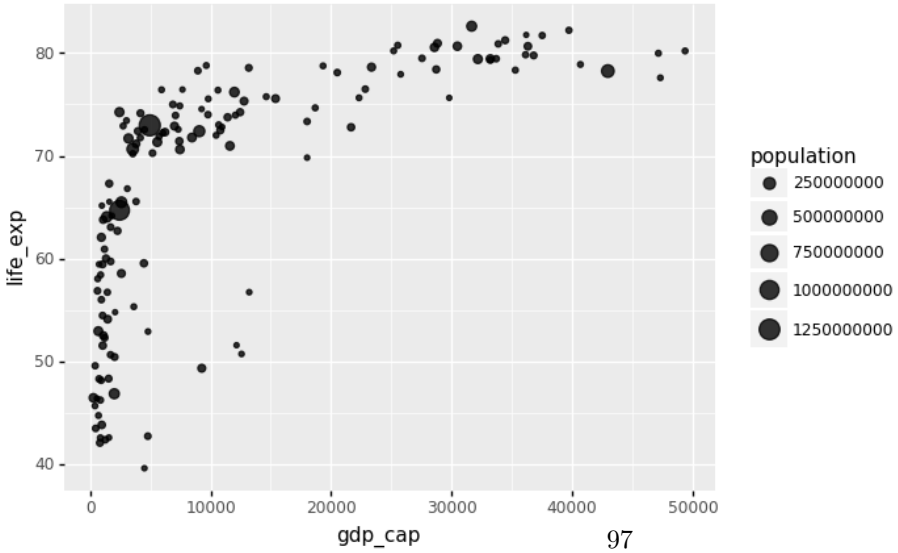
```
p = gapminder.pipe(year_filter,2007).pipe(ggplot)

( p +
  aes(x = 'gdp_cap', y = 'life_exp', size = 'population') + geom_point(alpha = 0.8)
)
```




```
<ggplot: (314400861)>
```

Please also note that we can also use pandas `query` method to filter or subset this data frame. Filtering conditions can be applied as a string using the query function. You can reference variables using @ (see example below). The `query` gives you more options than many other approaches.

```
#  In pandas query, you can reference variables using @:
# ref: https://stackoverflow.com/questions/57297077/use-variable-in-pandas-query

(
    ggplot(gapminder.query('year == @latest_year')) +
    aes(x = 'gdp_cap', y = 'life_exp', size = 'population') +
    geom_point(alpha = 0.8)
)
```




```
<ggplot: (314870913)>
```

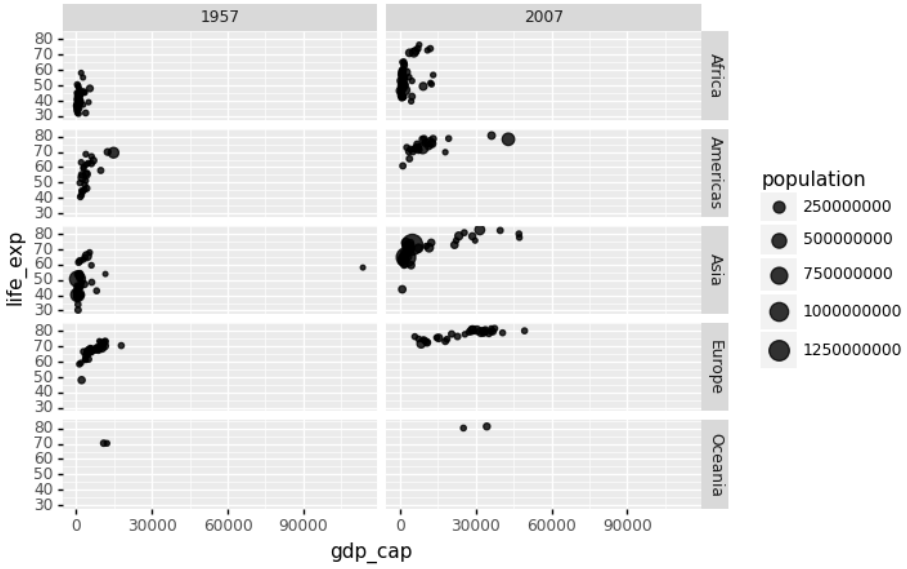
4.7.4. Faceting

We could easily plot the data from 2007 in the same way we plotted the data from 1962. However, side by side plots are recommended for comparisons.

Faceting variables in ggplot2 allows us to do this: we stratify the data by some variable and create the same plot for each strata. We add a layer with the function facet grid to create faceting, which automatically separates the plots. This function allows you to facet by up to two variables, with one variable represented by columns and the other by rows. The function expects a space between the row and column variables.

We can create a very simple grid with two facets (each subplot is a facet) as follows:

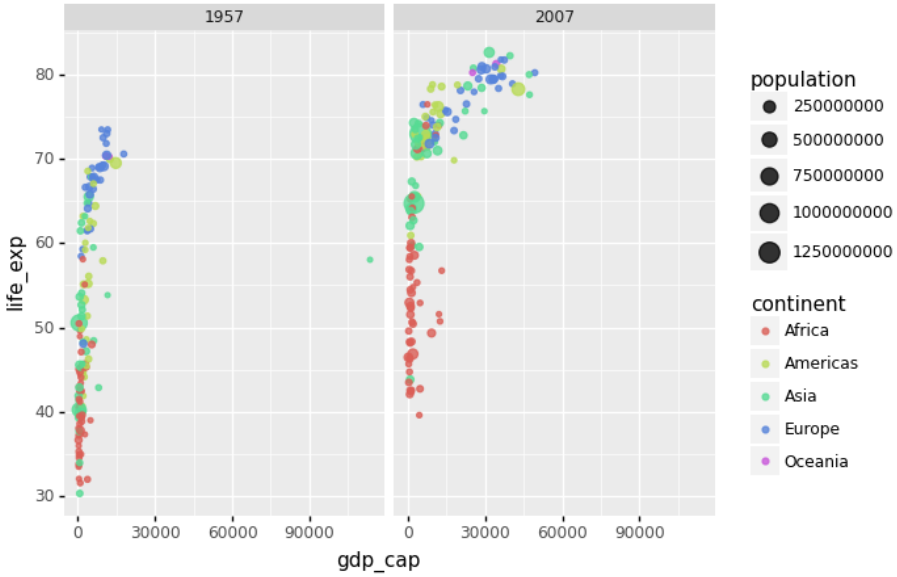
```
(
    gapminder.
    query('year == @beginning_year or year == @latest_year').
    pipe(ggplot) +
    aes(x = 'gdp_cap', y = 'life_exp', size = 'population') +
    geom_point(alpha = 0.8) +
    facet_grid('continent~year')
)
```




```
<ggplot: (312593245)>
```

Each continent/year pair has its own plot. However, this is merely an example and goes beyond what we are looking for, which is a simple comparison between 1957 and 2007. There is only one variable in this example, therefore we use `.` to tell `Facet` that we are not utilizing one of the variables:

```
(
    gapminder.
    query('year == @beginning_year or year == @latest_year').
    pipe(ggplot) +
    aes(x = 'gdp_cap', y='life_exp', size='population', color='continent') +
    geom_point(alpha = 0.8) +
    facet_grid('.~year')
)
```




```
<ggplot: (314650981)>
```

Many countries have moved from the developing world cluster to the western world cluster, as shown in this graph. This is especially evident when comparing Europe to Asia, which has some countries that have made significant progress.

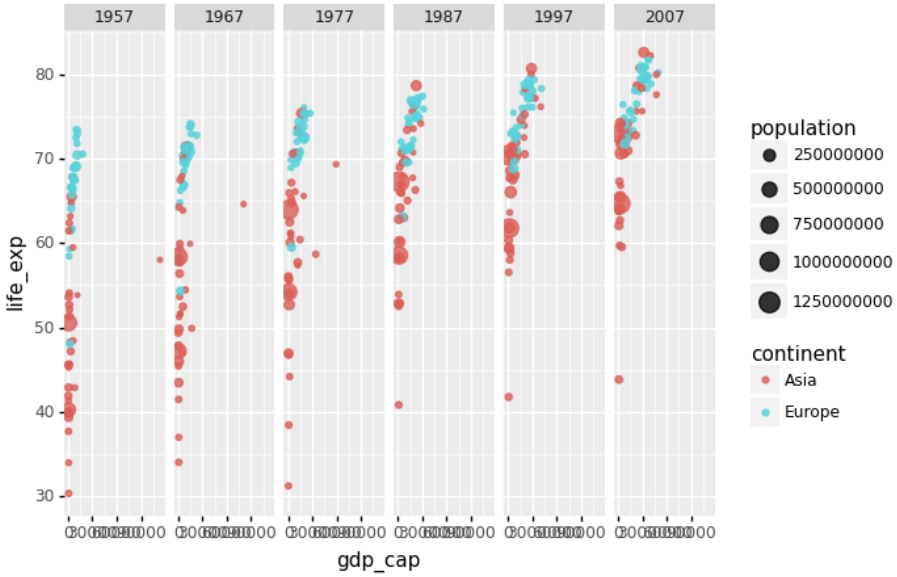
```
gapminder['year'].unique()
```

```
array([1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992, 1997, 2002,
       2007])
```

We can build a plot for multiple years to see how this transition occurred for over time. For instance, we can include the years 1957, 1967, 1977, 1987, 1997, 2007. If we do this, we will not want all the plots on the same row, which is `facet_grid`'s default behavior. We will want to create many rows and columns instead. This is made possible by the function `facet_wrap`, which wraps a series of plots so that each display has readable dimensions.

The next two plots compare between using `facet_grid` and `facet_wap`.

```
#continent_selected = ['Asia','Europe']

#(
#    gapminder.
#    query('year in [1957, 1967, 1977, 1987, 1997, 2007] and continent in
@continent_selected').
#    pipe(ggplot) +
#    aes(x = 'gdp_cap', y='life_exp', size='population', color='continent') +
#    geom_point(alpha = 0.8) +
#    facet_grid('.~year')
#)

# Notice that we use both single and double quotes.

(
    gapminder.
    query("year in [1957, 1967, 1977, 1987, 1997, 2007] and continent in
['Asia','Europe']").
    pipe(ggplot) +
    aes(x = 'gdp_cap', y='life_exp', size='population', color='continent') +
    geom_point(alpha = 0.8) +
    facet_grid('.~year')
)
```




```
<ggplot: (312923501)>
```

```
# Notice that we use both single and double quotes.

# using Facet_wrap

(
    gapminder.
    query("year in [1957, 1967, 1977, 1987, 1997, 2007] and continent in
['Asia','Europe']").
    pipe(ggplot) +
    aes(x = 'gdp_cap', y='life_exp', size='population', color='continent') +
    geom_point(alpha = 0.8) +
    facet_wrap('year')
)
```


```
<ggplot: (314202637)>
```

4.7.5. Data transformations

In this section, we refer to the following note: Chapter 11 of the following lecture note: https://rafalab.github.io/dsbook/

We will now turn our attention to the question, **which concerns the common belief that global wealth distribution has gotten worse in recent decades**. When asked if poor countries have gotten poorer and rich ones have gotten richer, the majority of people say **yes**.

We will be able to tell if this is the case by using stratification, histograms, smooth densities, and boxplots.

To begin, we will look at **how transformations can occasionally aid in the building of more useful summaries and plots**.

In the gapminder dataset, the market value of goods and services generated by a country in a year is measured by GDP. The GDP per capita is frequently used to give an approximate idea of a country's wealth. The GDP figures have been adjusted for inflation, so they should be comparable across the years.

We **divide this number by 365** to get the more easily understandable metric of `dollars_per_day`. A person living in absolute poverty is defined as someone who survives on less than $2 a day in current dollars. This variable can be added to the Gapminder data frame using `pipe` and `assign` below.

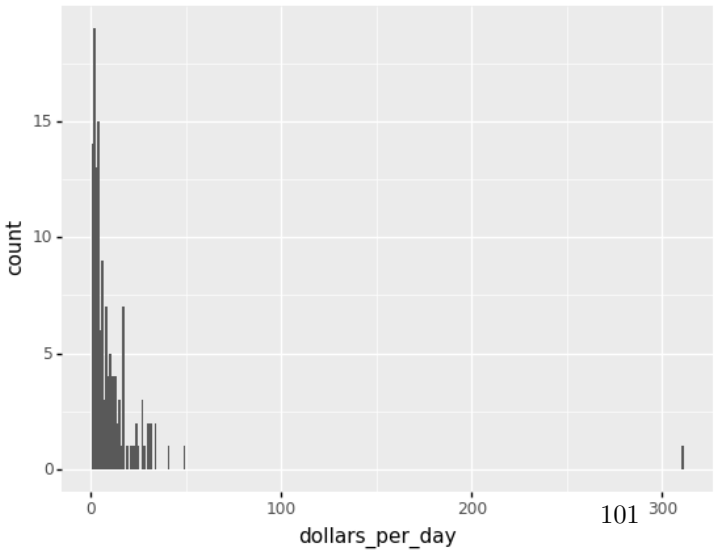
Recall that the `assign` method adds new columns to a DataFrame and returns a new object (a duplicate) with the new columns combined with the original. Existing columns will be overwritten if they are re-assigned.

```
# Ref: https://stackoverflow.com/questions/44631279/assign-in-pandas-pipeline

gapminder.pipe(lambda x: x.assign(dollars_per_day = x.gdp_cap/365)).head()
```

	country	year	population	continent	life_exp	gdp_cap	dollars_per_day
0	Afghanistan	1952	8425333	Asia	28.801	779.445314	2.135467
1	Afghanistan	1957	9240934	Asia	30.332	820.853030	2.248912
2	Afghanistan	1962	10267083	Asia	31.997	853.100710	2.337262
3	Afghanistan	1967	11537966	Asia	34.020	836.197138	2.290951
4	Afghanistan	1972	13079460	Asia	36.088	739.981106	2.027345

Here is a histogram of per day incomes from the 1957 data.

```
from plotnine import *

beginning_year = 1957

(
    gapminder.pipe(lambda x: x.assign(dollars_per_day = x.gdp_cap/365)).
    query('year == @beginning_year').
    pipe(ggplot) +
    aes('dollars_per_day') +
    geom_histogram(binwidth = 1)
)
```




```
<ggplot: (312926869)>
```

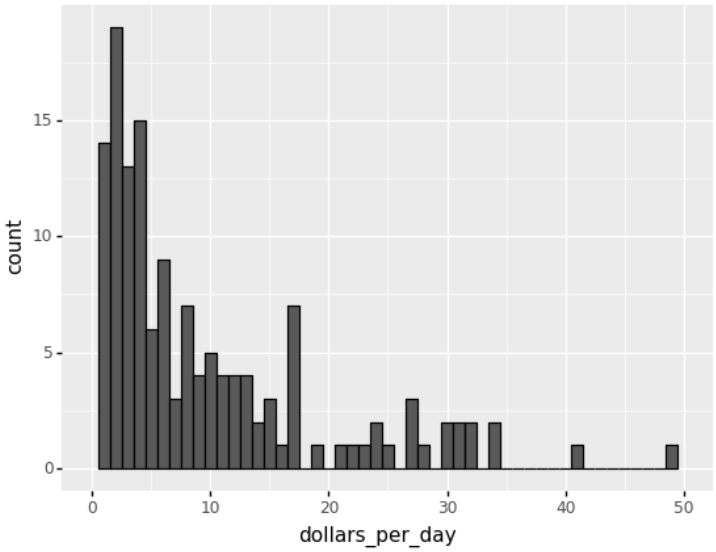
We set `xlim(0,50)` to remove an outlier.

You might want to investigate what's causing this outlier.

It is important to understand what causes them. Outliers can be caused by data entry/experiment measurement errors, sampling issues, or natural variation.

```
# Adjusting the x-limit

(
    gapminder.pipe(lambda x: x.assign(dollars_per_day = x.gdp_cap/365)).
    query('year == @beginning_year').
    pipe(ggplot) +
    aes('dollars_per_day') +
    geom_histogram(binwidth = 1, color = 'black') +
    xlim(0,50)
)
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages/plotnine/layer.py:324:
PlotnineWarning: stat_bin : Removed 1 rows containing non-finite values.
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages/plotnine/layer.py:401:
PlotnineWarning: geom_histogram : Removed 2 rows containing missing values.
```




```
<ggplot: (314277509)>
```

```
#(
#    (gapminder.pipe(lambda x: x.assign(dollars_per_day = x.gdp_cap/365)).
#    query('year == @beginning_year').sort_values(by = 'dollars_per_day',
ascending=False)['dollars_per_day']>= 20).value_counts()
#)
```

```
#(
#    gapminder.pipe(lambda x: x.assign(dollars_per_day = x.gdp_cap/365)).
#    query('year == @beginning_year').isnull().value_counts()
#)
```

The majority of countries' average daily income are less than \$20, as shown in the above figure.

20 countries with averages above \
$20, on the other hand, take up the majority of the x-axis. As a result, the plot is not very useful for countries with daily values below$ 20.

It could be more useful to know how many countries have average daily incomes of

- \$1 (extremely poor),
- \$2 (very poor),
- \$4 (poor),
- \$8 (middle),
- \$16 (well off),
- \$32 (rich),
- \$64 (very rich) each day.

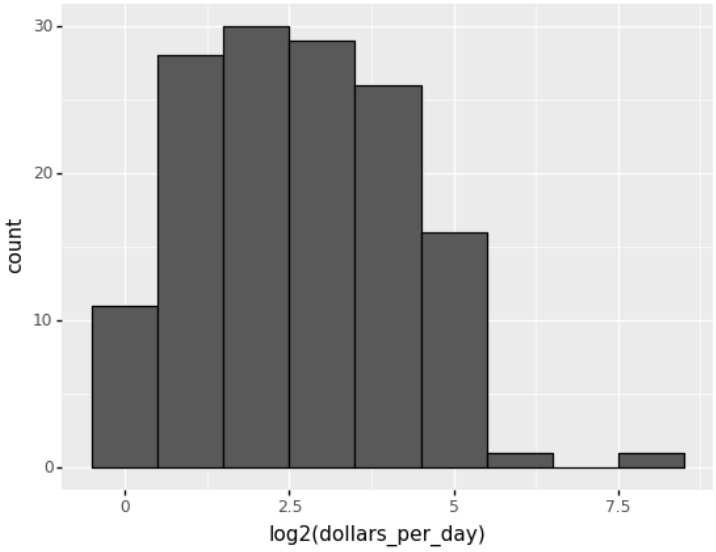
2**2 -> 2**3 log2(2**2) = 2 -> log2(**2**3) = 3

These changes are **multiplicative**, and log transformations turn multiplicative changes into **additive** ones: using base 2, a doubling of a value turns into an increase by 1 in the log base 2 scale.

4.7.5.1. Logging the values

If we use a log, we get the following distribution.

```
#Taking log base 2 of dollars income per day
# The variable name should be log(dollars_per_day)

(
    gapminder.pipe(lambda x: x.assign(dollars_per_day = np.log2(x.gdp_cap/365))).
    query('year == @beginning_year').
    pipe(ggplot) +
    aes('dollars_per_day') +
    geom_histogram(binwidth = 1, color = 'black') +
    xlab('log2(dollars_per_day)')
)
```




```
<ggplot: (312028493)>
```

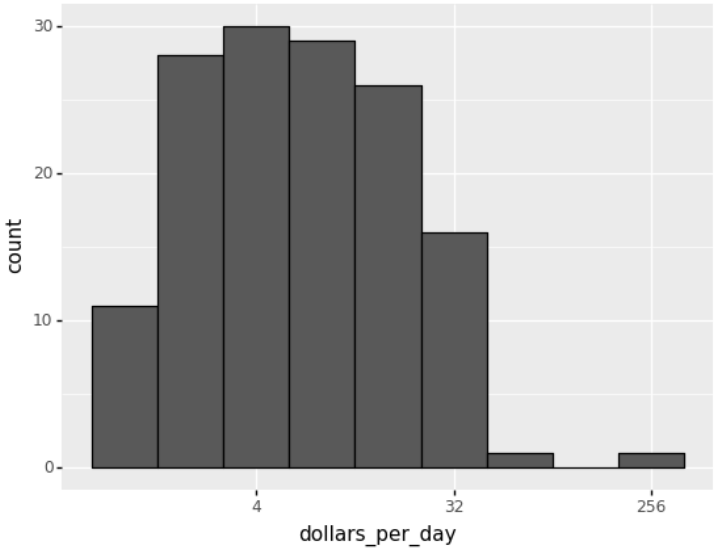
4.7.5.2. Scaling the axis

The scale_x_continuous function can be used to scale the axis with logs. We use this layer instead of logging the values first.

In plots, we can utilize log transformations in two ways. Before graphing, we can log the values or use log scales in the axes. Both approaches have advantages and disadvantages.

- We can more easily interpret **intermediate values in the scale** if we log the data.
- The benefit of using logged scales is that the **original values** are shown on the axes.

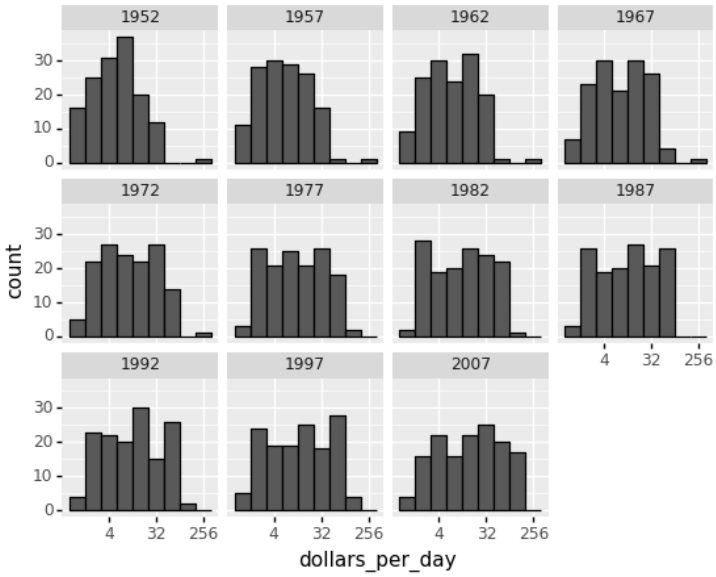
```
(
    gapminder.pipe(lambda x: x.assign(dollars_per_day = x.gdp_cap/365)).
    query('year == @beginning_year').
    pipe(ggplot) +
    aes('dollars_per_day') +
    geom_histogram(binwidth = 1, color = 'black') +
    scale_x_continuous(trans = "log2")
)
```




```
<ggplot: (312601081)>
```

Now let us apply `facet_wrap` to dig deeper into the variations of the histogram of incomes per day from 1952 to 2007.

```
(
    gapminder.pipe(lambda x: x.assign(dollars_per_day = x.gdp_cap/365)).
    query('year  in [1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992, 1997, 2007]
').
    pipe(ggplot) +
    aes('dollars_per_day') +
    geom_histogram(binwidth = 1, color = 'black') +
    scale_x_continuous(trans = "log2") +
    facet_wrap('year')
)
```




```
<ggplot: (311545613)>
```

From the figures above, the histogram in the 1967 income distribution values revealed a dichotomy (two groups). However, the histogram does not indicate whether the two groups of countries are developed and developing.

In what follows, we will read another CSV file, namely gapminder_dslab, which contains information about the regional classification. Then we will merge this regional classification into our original Gapminder dataset.

```
past_year = 1967
```

4.8. Adding regional classfication into Gapminder dataset

In addition to the variable `continent` variable in the Gapminder dataset, we will preprocess our data by adding another variable, region, which gives a classification of countries by region.

```
gapminder_dslab = pd.read_csv('/Users/Kaemyuijang/SCMA248/Data/gapminder_dslab.csv')


# url = 'https://raw.githubusercontent.com/pairote-
sat/SCMA248/main/Data/gapminder_dslab.csv'
# gapminder_dslab = pd.read_csv(url)
```

We can check that countries are classified into 22 different regions as follows:

```
region = gapminder_dslab.query('year== 2007')[['country','region']]
region['region'].unique()
```

```
array(['Southern Europe', 'Northern Africa', 'Middle Africa', 'Caribbean',
       'South America', 'Western Asia', 'Australia and New Zealand',
       'Western Europe', 'Southern Asia', 'Eastern Europe',
       'Central America', 'Western Africa', 'Southern Africa',
       'South-Eastern Asia', 'Eastern Africa', 'Northern America',
       'Eastern Asia', 'Northern Europe', 'Melanesia', 'Polynesia',
       'Central Asia', 'Micronesia'], dtype=object)
```

```
print(region.head()) #dslab
gapminder.head()
```

```
                country          region
8695             Albania  Southern Europe
8696             Algeria  Northern Africa
8697              Angola    Middle Africa
8698  Antigua and Barbuda        Caribbean
8699           Argentina    South America
```

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106

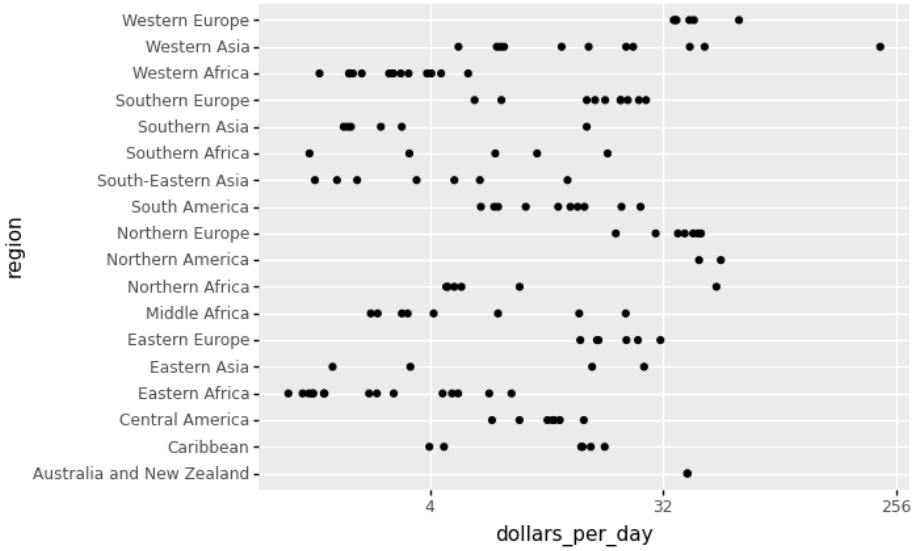
merge() is most useful when you want to combine rows that share data (in our case `country` also known as **key column**).

See https://realpython.com/pandas-merge-join-and-concat/ for more detail.

```
gapminder = pd.merge(gapminder,region, how='left',on='country')
```

Next, let us begin by looking at the data by region.

```
# note that past_year = 1967

(
    gapminder.pipe(lambda x: x.assign(dollars_per_day = x.gdp_cap/365)).
    query('year == @past_year').dropna().
    pipe(ggplot) +
    aes('dollars_per_day','region') +
    geom_point() +
    scale_x_continuous(trans='log2')
)
```




```
<ggplot: (311986217)>
```

By default, the discrete values along axis are ordered alphabetically.

To get more information from the plot above, we will **reorder the regions by the median value** (from low to high or vice versa). If we want a specific ordering we use a `pandas.Categorical` variable with categories ordered to our preference defined by keyword `categories`.

(see the links below for more detail https://plotnine.readthedocs.io/en/stable/tutorials/miscellaneous-order-plot-series.html

https://pandas.pydata.org/pandas-docs/stable/user_guide/categorical.html)

Notes

1. Categoricals are a data type in Pandas that represents statistical categorical variables. Gender, social status, blood type, country affiliation, observation time, and Likert scale ratings are some examples.
2. Categorical data values are either in categories or np.nan. Order is defined by the order of `categories`, not lexical order of the values.

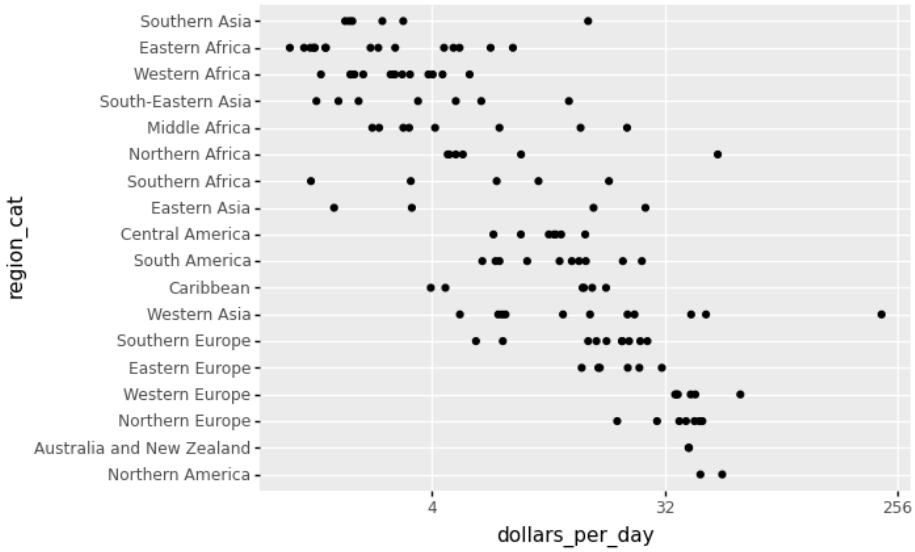
To enhance data visualization, we begin by creating a new data frame which is the filtered dataset of Gapminder, namely **gapminder1967**.

```
gapminder1967 = gapminder.pipe(lambda x: x.assign(dollars_per_day =
x.gdp_cap/365)).query('year == @past_year').dropna()
```

```
gapminder1967.head()
```

	country	year	population	continent	life_exp	gdp_cap	region	dollars_per_day
15	Albania	1967	1984060	Europe	66.220	2760.196931	Southern Europe	7.562183
27	Algeria	1967	12760499	Africa	51.407	3246.991771	Northern Africa	8.895868
39	Angola	1967	5247469	Africa	35.985	5522.776375	Middle Africa	15.130894
51	Argentina	1967	22934225	Americas	65.634	8052.953021	South America	22.062885
63	Australia	1967	11872264	Oceania	71.100	14526.124650	Australia and New Zealand	39.797602

```
# reorder the regions by the median value
region_list = gapminder1967.groupby('region').median().sort_values(by =
['dollars_per_day'], ascending = False).index.tolist()

# categories=region_list defines the order of Categoricals
region_cat = pd.Categorical(gapminder1967['region'], categories=region_list)

# assign to a new column in the DataFrame
gapminder1967 = gapminder1967.assign(region_cat = region_cat)

(
    gapminder1967.
    pipe(ggplot) +
    aes('dollars_per_day','region_cat') +
    geom_point() +
    scale_x_continuous(trans='log2')
)
```




```
<ggplot: (311707625)>
```

Based on the sorted median values of incomes per day, we clearly see that there is a "west vs the rest" dichotomy: there are two distinct groups,

- the rich group consisting of North America, Northern and Western Europe, New Zealand, and Australia, and
- the poor group consisting of the rest of the world.

In what follows, we form five different groups as based on our observations:

```
region_cat
```

```
['Southern Europe', 'Northern Africa', 'Middle Africa', 'South America', 'Australia and
New Zealand', ..., 'South America', 'South-Eastern Asia', 'Western Asia', 'Eastern
Africa', 'Eastern Africa']
Length: 133
Categories (18, object): ['Northern America', 'Australia and New Zealand', 'Northern
Europe', 'Western Europe', ..., 'South-Eastern Asia', 'Western Africa', 'Eastern Africa',
'Southern Asia']
```

```
gapminder1967
```

	country	year	population	continent	life_exp	gdp_cap	region	dollars_per_day	region_cat
15	Albania	1967	1984060	Europe	66.220	2760.196931	Southern Europe	7.562183	Southern Europe
27	Algeria	1967	12760499	Africa	51.407	3246.991771	Northern Africa	8.895868	Northern Africa
39	Angola	1967	5247469	Africa	35.985	5522.776375	Middle Africa	15.130894	Middle Africa
51	Argentina	1967	22934225	Americas	65.634	8052.953021	South America	22.062885	South America
63	Australia	1967	11872264	Oceania	71.100	14526.124650	Australia and New Zealand	39.797602	Australia and New Zealand
...	...	...	...	...	...	...	...	...	...
1635	Venezuela	1967	9709552	Americas	63.479	9541.474188	South America	26.141025	South America
1647	Vietnam	1967	39463910	Asia	47.838	637.123289	South-Eastern Asia	1.745543	South-Eastern Asia
1659	West Bank and Gaza	1967	1142636	Asia	51.631	2649.715007	Western Asia	7.259493	Western Asia
1683	Zambia	1967	3900000	Africa	47.768	1777.077318	Eastern Africa	4.868705	Eastern Africa
1695	Zimbabwe	1967	4995432	Africa	53.995	569.795071	Eastern Africa	1.561082	Eastern Africa

133 rows × 9 columns

```
gapminder1967.region.isin(["Western Europe", "Northern Europe","Southern Europe",
"Northern America",
"Australia and New Zealand"])
```

```
15       True
27      False
39      False
51      False
63       True
        ...
1635    False
1647    False
1659    False
1683    False
1695    False
Name: region, Length: 133, dtype: bool
```

Exercise:

1. Append the column called `group` that groups countries into 5 different groups as follows:

- West: [“Western Europe”, “Northern Europe”,”Southern Europe”, “Northern America”, “Australia and New Zealand”]
- East Asia: [“Eastern Asia”, “South-Eastern Asia”]
- Latin America: [“Caribbean”, “Central America”, “South America”]
- Sub-Saharan: [continent == “Africa”] & [region != “Northern Africa”]
- Other: All remaining countries (also including NAN).

1. We now want to compare the distribution across these five groups to confirm the “west versus the rest” dichotomy. We could generate five histograms or five density plots, but it may be more practical to have all the visual summaries in one plot. Write Python code to stack boxplots next to each other.
2. We now want to compare the distribution across these five groups to confirm the “west versus the rest” dichotomy. To do this, we will work with the 1967 data. We could generate five histograms or five smooth density plots, but it may be more practical to have all the visual summaries **in one plot**. Write Python code to stack smooth density plots (or histograms) vertically (with slightly overlapping lines) that share the same x-axis.

The Numpy `select` function is more powerful for creating a new column based on some conditions on the other columns. It can be used to specify a set of conditions and values. As a result, each condition can be assigned a specific value.

See https://towardsdatascience.com/3-methods-to-create-conditional-columns-with-python-pandas-and-numpy-a6cd4be9da53 and https://towardsdatascience.com/create-new-column-based-on-other-columns-pandas-5586d87de73d

```
gapminder.query("year==2007 & continent == 'Africa'").filter(items=
['country','continent','region']).head()
```

	country	continent	region
35	Algeria	Africa	Northern Africa
47	Angola	Africa	Middle Africa
131	Benin	Africa	Western Africa
167	Botswana	Africa	Southern Africa
203	Burkina Faso	Africa	Western Africa

Let's start by defining the conditions and their values.

```
# https://stackoverflow.com/questions/19960077/how-to-filter-pandas-dataframe-using-in-and-not-in-like-in-sql

conditions = [
    gapminder1967.region.isin(["Western Europe", "Northern Europe","Southern Europe",
"Northern America",
"Australia and New Zealand"]),
    gapminder1967.region.isin(["Eastern Asia", "South-Eastern Asia"]),
    gapminder1967.region.isin(["Caribbean", "Central America",
"South America"] ),
    (gapminder1967.continent.isin(["Africa"])) & (~gapminder1967.region.isin(["Northern
Africa"] ))
]

values = ['West','East Asia','Latin America','Sub-Saharan']

np.select(conditions,values, default="Others")
```

```
array(['West', 'Others', 'Sub-Saharan', 'Latin America', 'West', 'West',
       'Others', 'Others', 'West', 'Sub-Saharan', 'Latin America', 'West',
       'Sub-Saharan', 'Latin America', 'Others', 'Sub-Saharan',
       'Sub-Saharan', 'East Asia', 'Sub-Saharan', 'West', 'Sub-Saharan',
       'Sub-Saharan', 'Latin America', 'East Asia', 'Latin America',
       'Sub-Saharan', 'Sub-Saharan', 'Sub-Saharan', 'Latin America',
       'Sub-Saharan', 'West', 'Latin America', 'Others', 'West',
       'Sub-Saharan', 'Latin America', 'Latin America', 'Others',
       'Latin America', 'Sub-Saharan', 'Sub-Saharan', 'Sub-Saharan',
       'West', 'West', 'Sub-Saharan', 'Sub-Saharan', 'West',
       'Sub-Saharan', 'West', 'Latin America', 'Sub-Saharan',
       'Sub-Saharan', 'Latin America', 'Latin America', 'East Asia',
       'Others', 'West', 'Others', 'East Asia', 'Others', 'Others',
       'West', 'Others', 'West', 'Latin America', 'East Asia', 'Others',
       'Sub-Saharan', 'Others', 'Others', 'Sub-Saharan', 'Sub-Saharan',
       'Others', 'Sub-Saharan', 'Sub-Saharan', 'East Asia', 'Sub-Saharan',
       'Sub-Saharan', 'Sub-Saharan', 'Latin America', 'East Asia', 'West',
       'Others', 'Sub-Saharan', 'Sub-Saharan', 'Others', 'West', 'West',
       'Latin America', 'Sub-Saharan', 'Sub-Saharan', 'West', 'Others',
       'Others', 'Latin America', 'Latin America', 'Latin America',
       'East Asia', 'Others', 'West', 'Latin America', 'Others',
       'Sub-Saharan', 'Others', 'Sub-Saharan', 'West', 'Sub-Saharan',
       'East Asia', 'Others', 'West', 'Sub-Saharan', 'West', 'Others',
       'Others', 'Sub-Saharan', 'West', 'West', 'Others', 'Sub-Saharan',
       'East Asia', 'Sub-Saharan', 'Latin America', 'Others', 'Others',
       'Sub-Saharan', 'West', 'West', 'Latin America', 'Latin America',
       'East Asia', 'Others', 'Sub-Saharan', 'Sub-Saharan'], dtype='<U13')
```

```
gapminder1967['group']=np.select(conditions,values, default="Others")

gapminder1967.head()
```

	country	year	population	continent	life_exp	gdp_cap	region	dollars_per_day	region_cat	group
15	Albania	1967	1984060	Europe	66.220	2760.196931	Southern Europe	7.562183	Southern Europe	West
27	Algeria	1967	12760499	Africa	51.407	3246.991771	Northern Africa	8.895868	Northern Africa	Others
39	Angola	1967	5247469	Africa	35.985	5522.776375	Middle Africa	15.130894	Middle Africa	Sub-Saharan
51	Argentina	1967	22934225	Americas	65.634	8052.953021	South America	22.062885	South America	Latin America
63	Australia	1967	11872264	Oceania	71.100	14526.124650	Australia and New Zealand	39.797602	Australia and New Zealand	West

We turn this group variable into a factor to control the order of the levels.

```
group_list = ["Others", "Latin America","East Asia", "Sub-Saharan", "West"]

gapminder1967['group'] = pd.Categorical(gapminder1967['group'], categories=group_list)
```

Two characteristics of the average income distribution in 1967 were identified by the exploratory data analysis above. We discovered a **bimodal distribution** using a histogram, with the modes corresponding to poor and rich countries.

To confirm the “west vs the rest” dichotomy, we will compare the distribution across these five groups. A summary plot may be beneficial because the number of points in each category is large enough.

We could make five histograms or density plots, but having all of the visual summaries in one figure may be more practical. As a result, we begin by stacking boxplots adjacent to one another.

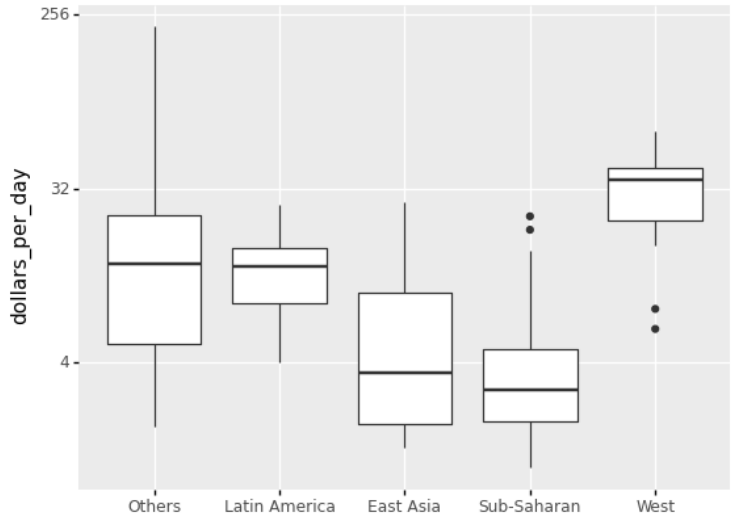
Note A **boxplot** is a standard method of displaying a dataset based on a five-number summary: minimum, maximum, sample median, first and third quartiles.

In addition to the lowest and maximum values required to create a box-plot, the interquartile range (IQR), as shown in the plot, is another essential feature that can be used to create a box-plot.

As seen from the plot, a box plot normally consists of two parts: a box and a pair of whiskers. The lowest point on the box-plot (i.e. the lower whisker’s boundary) represents the data set’s minimum value, while the highest point (i.e. the upper whisker’s boundary) represents the data set’s maximum value (excluding any **outliers**). A horizontal line is drawn in the middle of the box to signify the median, which drawn from Q1 to Q3.

The whisker limits are found inside the 1.5 IQR value. Outliers are all other observed data points that are outside the whisker’s boundary.

```
(
    ggplot(gapminder1967) +
    aes('group','dollars_per_day') +
    geom_boxplot() +
    scale_y_continuous(trans='log2') +
    xlab('')
)
```



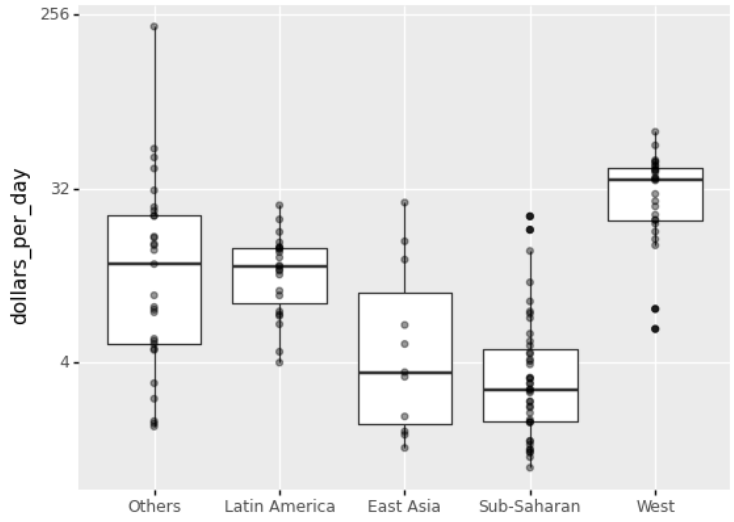

```
<ggplot: (311822765)>
```

```
#gapminder1967.query('region == polynesia')
gapminder1967.region.head()
```

```
15              Southern Europe
27              Northern Africa
39                Middle Africa
51                South America
63    Australia and New Zealand
Name: region, dtype: object
```

The shortcoming of boxplots is that by summarizing the data into five numbers, we may miss important data properties. Showing the data is one approach to avoid this.

```
(
    ggplot(gapminder1967) +
    aes('group','dollars_per_day') +
    geom_boxplot() +
    scale_y_continuous(trans='log2') +
    xlab('') +
    geom_point(alpha=0.4)
)
```




```
<ggplot: (311707437)>
```



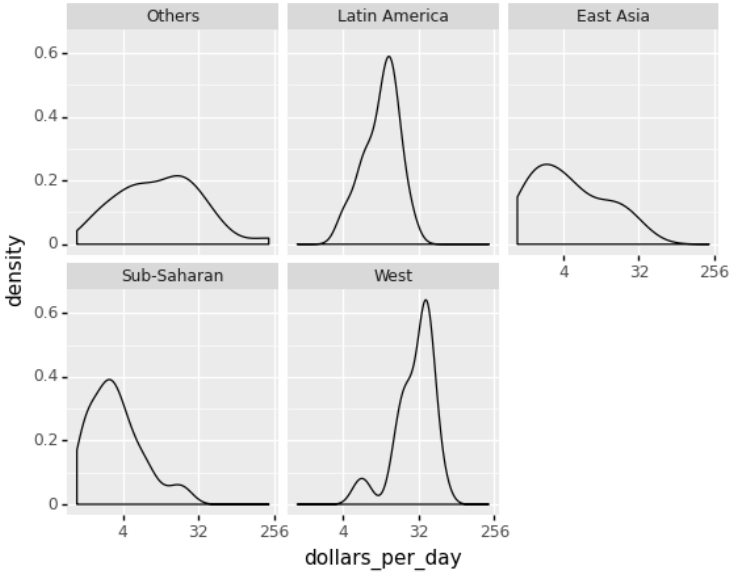
```
len(gapminder1967.region.unique())
```

```
18
```

If we are concerned that the boxplot summary is too basic, we can present smooth densities or histograms with `facet_wrap`.

Note if you are working with R programming, the library **ggridges** (together with ggplot) can be used to visualize distributions of several groups of a category. This allows us to stack smooth densities or histograms vetically. We refer to the stacked plot as **ridge plots**. See for an example https://cran.r-project.org/web/packages/ggridges/vignettes/gallery.html and https://www.analyticsvidhya.com/blog/2021/06/ridgeline-plots-visualize-data-with-a-joy/#:~:text=Ridgeline Plot or Joy Plot,beautiful piece of the plot.

```
(
    ggplot(gapminder1967) + aes('dollars_per_day') +
    geom_density() +
    facet_wrap('group') +
    scale_x_continuous(trans='log2')
)
```




```
<ggplot: (313847509)>
```

4.8.1. In Python, plot a ridgeline plot.

In Python, a Ridgeline Plot can be created using a variety of tools, including the popular Matplotlib and Plotly libraries. However, plotting a Ridgeline Plot with joypy is rather simple. Here is an example to the Gapminder dataset.

See the following for more detail: https://towardsdatascience.com/ridgeline-plots-the-perfect-way-to-visualize-data-distributions-with-python-de99a5493052

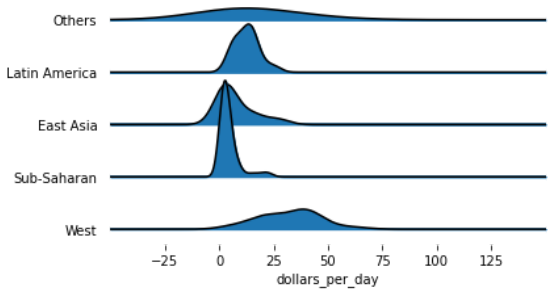
```
!pip install joypy
```

```
Requirement already satisfied: joypy in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (0.2.6)
Requirement already satisfied: pandas>=0.20.0 in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from joypy) (1.3.4)
Requirement already satisfied: scipy>=0.11.0 in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from joypy) (1.7.1)
Requirement already satisfied: numpy>=1.16.5 in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from joypy) (1.21.2)
Requirement already satisfied: matplotlib in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from joypy) (3.5.0)
Requirement already satisfied: python-dateutil>=2.7.3 in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from pandas>=0.20.0->joypy)
(2.8.2)
Requirement already satisfied: pytz>=2017.3 in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from pandas>=0.20.0->joypy)
(2021.3)
Requirement already satisfied: six>=1.5 in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from python-
dateutil>=2.7.3->pandas>=0.20.0->joypy) (1.16.0)
Requirement already satisfied: cycler>=0.10 in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from matplotlib->joypy)
(0.11.0)
Requirement already satisfied: packaging>=20.0 in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from matplotlib->joypy)
(21.3)
Requirement already satisfied: pyparsing>=2.2.1 in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from matplotlib->joypy)
(3.0.4)
Requirement already satisfied: fonttools>=4.22.0 in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from matplotlib->joypy)
(4.25.0)
Requirement already satisfied: pillow>=6.2.0 in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from matplotlib->joypy)
(8.4.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from matplotlib->joypy)
(1.3.1)
```

```
from joypy import joyplot
import matplotlib.pyplot as plt
```

```
joyplot(gapminder1967, by='group', column='dollars_per_day', x_range=[-50,150])
plt.xlabel('dollars_per_day')
```

```
Text(0.5, 15.0, 'dollars_per_day')
```



Previous
3. Data Preparation

Next
5. Practical Statistics

By Pairote Satiracoo
© Copyright 2021.

5 Practical Statistics

5. Practical Statistics

5.1. Density Plots and Estimates

A histogram is a visual representation of a frequency table, with the data count on the y-axis and the bins on the x-axis. It displays the amount of occurrences of certain values. In Chapter 4, we have learned how to make a histogram with pandas and plotnine.

A density plot is similar to a histogram in that it representts the distribution of data values as a continuous line.

A density plot can be thought of as a smoothed histogram, which is normally produced directly from the data using a **kernel density estimate (KDE)**.

Both histograms and KDEs are supported by the majority of major data science libraries. For example, in pandas, we can use `df.hist` to plot a histogram of data for a given DataFrame. `df.plot.density()`, on the other hand, returns a KDE plot with Gaussian kernels.

Recall that the histogram in the 1967 income distribution values of the Gapminder dataset revealed a dichotomy (two groups). In the following example, a density estimate is superimposed on a histogram of the income distribution with the following Python commands.

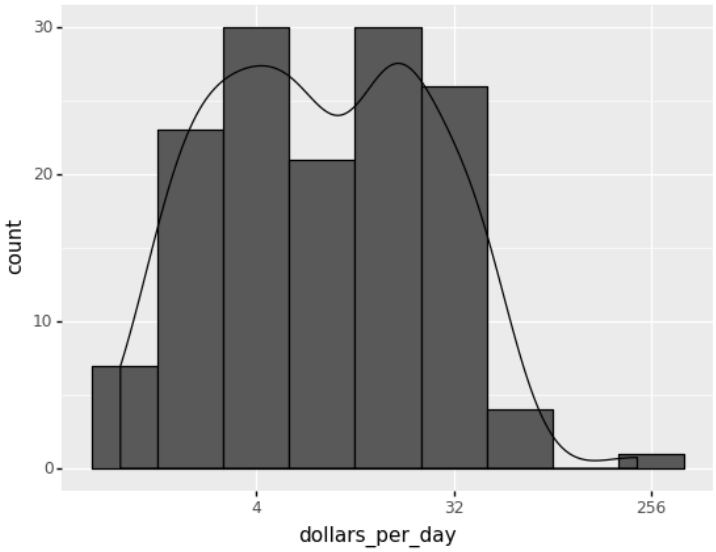
```
import numpy as np
import pandas as pd
from plotnine import *
from scipy.stats import *

from IPython.display import IFrame, YouTubeVideo, SVG, HTML

# Add this line so you can plot your charts into your Jupyter Notebook.
%matplotlib inline
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/var/folders/kl/h_r05n_j76n32kt0dwy7kynw0000gn/T/ipykernel_5496/3222504104.py in <module>
      1 import numpy as np
      2 import pandas as pd
----> 3 from plotnine import *
      4 from scipy.stats import *
      5

ModuleNotFoundError: No module named 'plotnine'
```

```
# for inline plots in jupyter
%matplotlib inline
# import matplotlib
import matplotlib.pyplot as plt
# for latex equations
from IPython.display import Math, Latex
# for displaying images
from IPython.core.display import Image
```

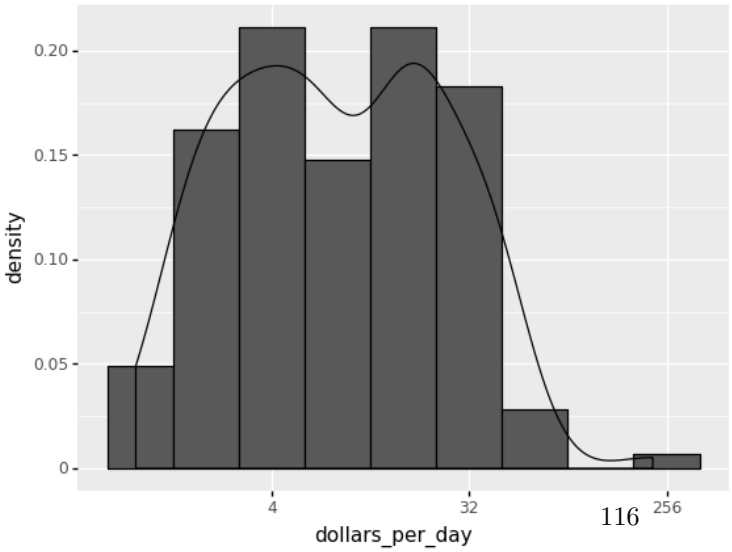
```
gapminder = pd.read_csv('/Users/Kaemyuijang/SCMA248/Data/gapminder_full.csv')

#url = 'https://raw.githubusercontent.com/STLinde/Anvendt-
Statistik/main/gapminder_full.csv'
#gapminder = pd.read_csv(url)
#gapminder['year'].unique()

past_year = 1967

gapminder1967 = gapminder.pipe(lambda x: x.assign(dollars_per_day =
x.gdp_cap/365)).query('year == @past_year').dropna()
```

```
# see
https://plotnine.readthedocs.io/en/stable/generated/plotnine.stats.stat_density.html#plo
tnine.stats.stat_density
# plotnine.mapping.after_stat evaluates mapping after statistic has been calculated

(
    ggplot(gapminder1967) +
    aes('dollars_per_day') +
    geom_histogram(aes(y=after_stat('count')), binwidth = 1, color = 'black') +
    geom_density(aes(y=after_stat('count'))) +
    scale_x_continuous(trans = 'log2')
)
```




```
<ggplot: (310164697)>
```

```
# see
https://plotnine.readthedocs.io/en/stable/generated/plotnine.stats.stat_density.html#plo
tnine.stats.stat_density
# plotnine.mapping.after_stat evaluates mapping after statistic has been calculated

(
    ggplot(gapminder1967) +
    aes('dollars_per_day') +
    geom_histogram(aes(y=after_stat('density')), binwidth = 1, color = 'black') +
    geom_density(aes(y=after_stat('density'))) +
    scale_x_continuous(trans = 'log2')
)
```




```
<ggplot: (278383825)>
```

The scale of the y-axis of the KDE differs from the histogram presented in Figures above. A density plot corresponds to plotting the histogram as a **proportion** rather than counts (you indicate this with the `aes(y=after_stat('density')` parameter). You calculate areas under the curve between any two points on the x-axis, which correspond to the proportion of the distribution residing between those two locations, instead of counting in bins.

Exercise use `plot.hist` and `plot.density` to display a density estimate of income distribution superposed on a histogram.

```
# gapminder1967['dollars_per_day'].plot.hist(density=True)
```

5.2. Correlation

In many modeling initiatives (whether in data science or research), exploratory data analysis requires looking for correlations among predictors and between predictors and a target variable.

Positively correlated variables X and Y (each with measured data) are those in which high values of X correspond to high values of Y and low values of X correspond to low values of Y.

The variables are **negatively correlated** if high values of X correspond to low values of Y and vice versa.

Correlation Term Glossary

- **(Pearson's) Correlation coefficient** This metric, which goes from –1 to +1, quantifies the degree to which numeric variables are associated to one another.
- **Correlation matrix** The variables are displayed on both rows and columns in this table, and the cell values represent the correlations between the variables.

To get Pearson's correlation coefficient, we multiply deviations from the mean for variable 1 times those for variable 2, and divide by the product of the standard deviations: Given paired data $ $\{(x_1, y_1), \ldots, (x_n, y_n)\}\{(x_1, y_1), \ldots, (x_n, y_n)\}consistingof\{\displaystyle n\}$ pairs

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

or

$$r_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{s_x}\right)\left(\frac{y_i - \bar{y}}{s_y}\right)$$

5.2.1. The correlation matrix

The correlation matrix presents the relationship between all of the variable (feature) pairs. It is frequently the initial step in dimensionality reduction because it shows you how many features are tightly connected (and so may be discarded) versus how many are independent.

For illustration, let us use the Iris Data Set, containing four features of three Iris classes. The correlation matrix may be simply computed using the following code:

```
# Need to transpose the Iris dataset (which is the numpy array) before
# applying corrcoef

from sklearn import datasets
import numpy as np
iris = datasets.load_iris()


#cov_data = np.corrcoef(iris.data.T)
#import matplotlib.pyplot as plt
#img = plt.matshow(cov_data, cmap=plt.cm.winter)
#plt.colorbar(img, ticks=[-1, 0, 1])
#print(cov_data)
```

```
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)

df["target"] = iris.target

df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

5.2.2. Finding Correlation Between Two Variables

The pandas dataframe provides the method called `corr()` to find the correlation between the variables. It calculates the correlation between the two variables.

```
correlation = df["sepal length (cm)"].corr(df["petal length (cm)"])

correlation
```

```
0.8717537758865831
```

The correlation between the features sepal length and petal length is around 0.8717. The number is closer to 1, which means these two features are highly correlated.

5.2.3. Plotting Correlation Matrix

In this section, you'll plot the correlation matrix by using the background gradient colors. This internally uses the matplotlib library.

First, find the correlation between each variable available in the dataframe using the `corr()` method. The `corr()` method will give a matrix with the correlation values between each variable.

Now, set the background gradient for the correlation data. Then, you'll see the correlation matrix colored.

```
corr = df.corr()
corr
# corr.style.background_gradient(cmap='coolwarm')
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
sepal length (cm)	1.000000	-0.117570	0.871754	0.817941	0.782561
sepal width (cm)	-0.117570	1.000000	-0.428440	-0.366126	-0.426658
petal length (cm)	0.871754	-0.428440	1.000000	0.962865	0.949035
petal width (cm)	0.817941	-0.366126	0.962865	1.000000	0.956547
target	0.782561	-0.426658	0.949035	0.956547	1.000000

5.2.4. Plotting Correlation HeatMap

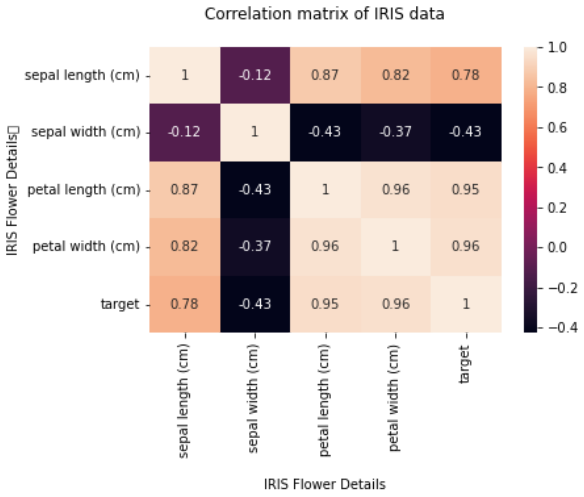
We can visualize the correlation matrix in a graphical form using a heatmap from the Seaborn library.

In what follows, you can add title and axes labels using the `heatmap.set(xlabel='X Axis label', ylabel='Y axis label', title='title')`.

After setting the values, you can use the `plt.show()` method to plot the heat map with the x-axis label, y-axis label, and the title for the heat map.

```
import seaborn as sns
import matplotlib.pyplot as plt
hm = sns.heatmap(df.corr(), annot = True)
hm.set(xlabel='\nIRIS Flower Details', ylabel='IRIS Flower Details\t', title =
"Correlation matrix of IRIS data\n")
plt.show()
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 9 ( ) missing from current
font.
```



The value of the diagonals is 1 as you can see in the preceding figure. There is also a strong link (high correlation) between the first and third features, the first and fourth features, and the third and fourth features. As a result, we can observe that only the second feature is nearly independent of the others; the rest are associated in some way.

The correlation coefficient, like the mean and standard deviation, is sensitive to outliers in the data. Robust alternatives to the traditional correlation coefficient are available in software packages. The methods in the `sklearn.covariance` scikit-learn module implement a range of approaches. See http://scikit-learn.org/stable/modules/covariance.html for more detail.

To use plotnine to create the heatmap, see https://www.r-bloggers.com/2021/06/plotnine-make-great-looking-correlation-plots-in-python/

5.3. Data and Sampling Distributions

When we are working on a problem with a large data set, it is usually not possible or necessary to work with the entire data set unless you want to wait hours for processing transformations and feature engineering to complete.

Drawing a sample from your data that is informative enough to discover important insights is a more effective method that will still allow you to draw accurate conclusions from your results.

Let us have a look at some fundamental terminology.

The term **"population"** refers to a grouping of items that share some property. The population size is determined by the number of elements in the population.

The term **"sample"** refers to a portion of the population. Sampling is the procedure for picking a sample. The sample size is the number of elements in the sample.

The term **"probability Sampling method"** employs randomization to ensure that every member of the population has an equal probability of being included in the chosen sample. **Random sampling** is another name for it.

The figure below depicts a diagram that explains the principles of data and sampling distributions that we will cover in this chapter.

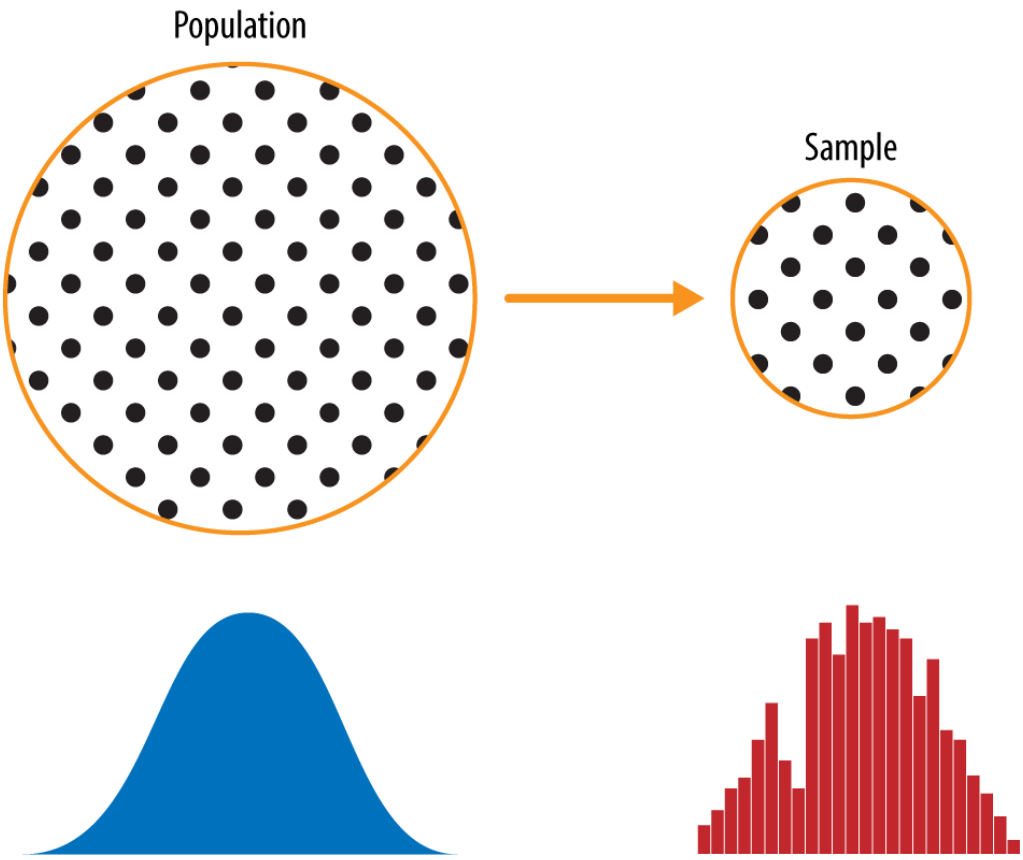


Image source: https://www.oreilly.com/library/view/practical-statistics-for/9781491952955/assets/psds_0201.png

The population on the left represents a population that is believed to **follow an underlying but unknown distribution in statistics**. Only the sample data and its empirical distribution, as shown on the right, are available.

A sampling process is utilized to move from the left to the right side (represented by an arrow).

- Traditional statistics emphasized the left side, employing theory based on strong assumptions about the population.
- Modern statistics has shifted to the righthand side, eliminating the need for such assumptions.

When compared to working with full or complete datasets, sampling offers numerous advantages, including lower costs and faster processing.

To sample data, you must first specify your population and the procedure for selecting (and sometimes rejecting) observations for inclusion in your sample. The population parameters you want to estimate with the sample could very well describe this.

Before obtaining a data sample, think about the following points:

- Sample Goal. The population property (parameters) that you wish to estimate using the sample.
- Population. The range or domain within which observations could be made.
- Selection Criteria. The procedure for accepting or rejecting observations from your sample.
- Sample Size. The number of observations that will constitute the sample.

5.3.1. Sampling bias

One of the most common types of biases seen in real-world scenarios is sampling bias.

Sampling bias arises when some members of a population are systematically more likely to be selected in a sample than others.

In machine learning, it arises when the data used to train a model does not accurately reflect the distribution of samples that the model would encounter in the production.

5.3.2. Types of sampling techniques

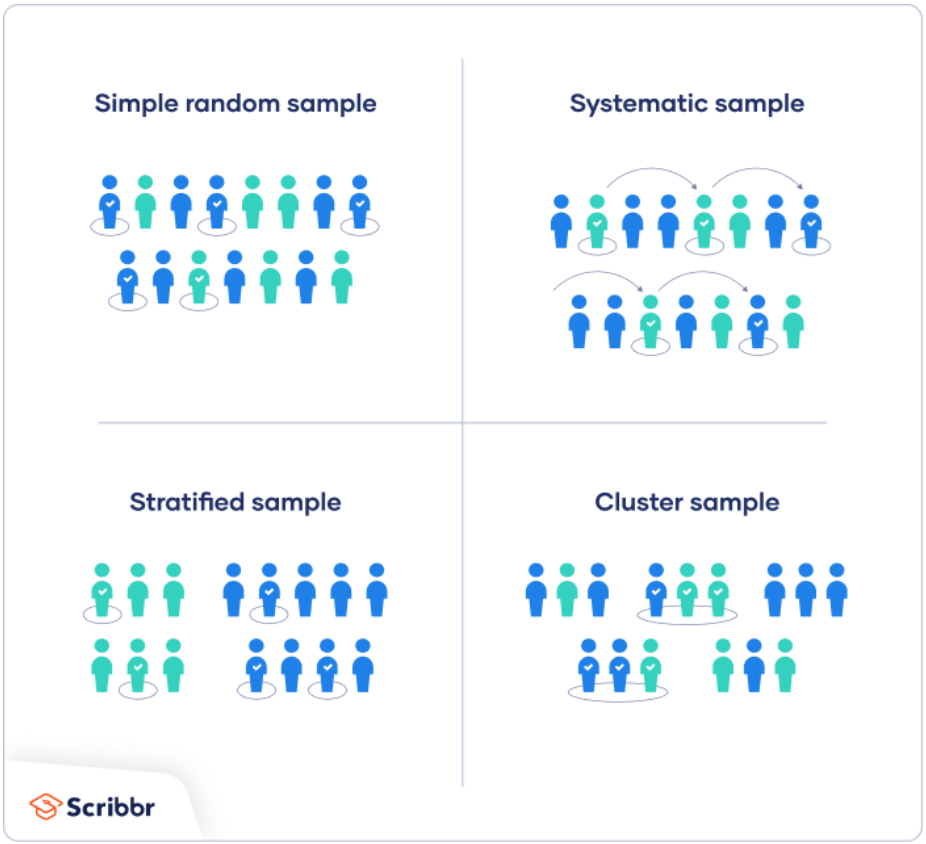


Image source: https://cdn.scribbr.com/wp-content/uploads/2019/09/probability-sampling.png

5.3.2.1. Simple random sampling

The simple random sampling is the simplest straightforward approach to sample data. In essence, the subset is made up of observations that were randomly selected from a bigger set; each observation has the same chance of being chosen from the larger set.

Simple random sampling is simple and straightforward to implement. However, it's still feasible that we'll introduce bias into our sample data. Consider a scenario in which we have a large dataset with unbalanced labels (or categories). We may mistakenly fail to collect enough cases to represent the minority class by using simple random sampling.

Example You wish to choose a simple random sample of 100 employees of the company . You assign a number from 1 to 1000 to each employee in the company database, and then choose 100 numbers using a random number generator.

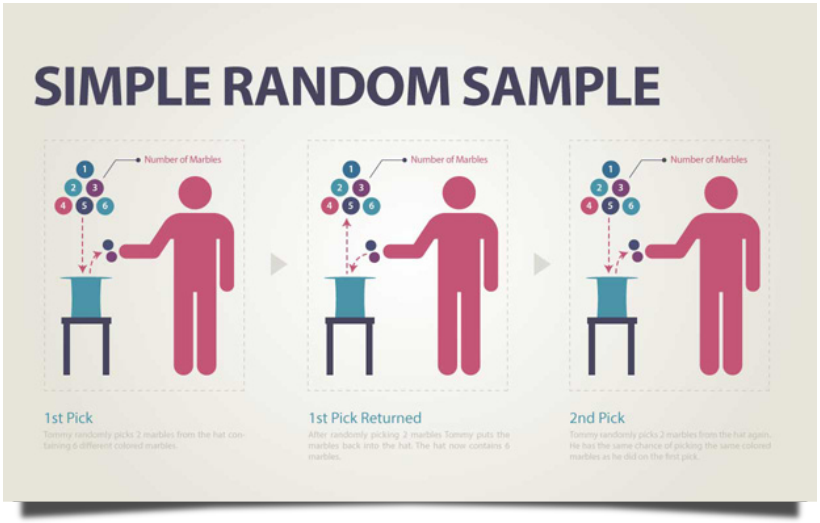


Image source: https://lc.gcumedia.com/hlt362v/the-visual-learner/images/simple-random-sample.png

For illustration, we will be using synthetic data which can be prepared in Python as follows:

```
# create synthetic data
id = np.arange(0, 10).tolist()
height = np.round(np.random.normal(loc=165, scale =15, size=len(id)), 2)

# convert to pandas dataframe
data = {"id":id, "height": height}
df = pd.DataFrame(data=data)
df
```

	id	height
0	0	166.34
1	1	126.03
2	2	166.33
3	3	191.29
4	4	147.31
5	5	153.55
6	6	183.67
7	7	154.53
8	8	177.61
9	9	187.80

To perform random sampling, Python pandas includes a method called `sample()`. You can use `random_state` for reproducibility.

```
# simple sampling example
simple_random_sample = df.sample(n=5, random_state=888)
simple_random_sample
```

	id	height
2	2	166.33
0	0	166.34
8	8	177.61
5	5	153.55
4	4	147.31

5.3.2.2. Systematic sampling

Systematic sampling is similar to simple random sampling, but it is usually slightly easier to carry out. Every person in the population is assigned a number, but rather than assigning numbers at random, individuals are chosen at regular intervals.

When the observations are randomized, systematic sampling usually yields a better sample than simple random sampling. If our data contains periodicity or repeating patterns, however, systematic sampling is not suitable.

Example All of the company's employees are listed alphabetically. You choose a starting point at random from the first ten numbers: number 6. Every tenth individual on the list is chosen from number 6 onwards (6, 16, 26, 36, and so on), resulting in a sample of 100 persons.

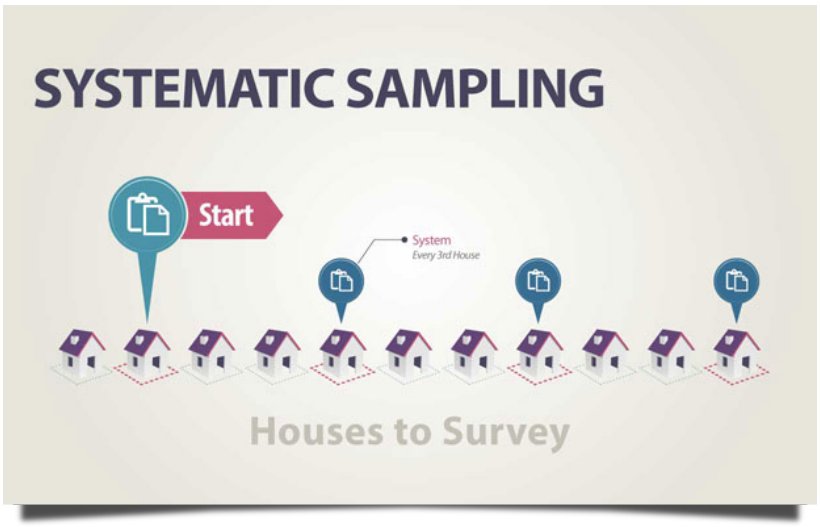


Image source: https://lc.gcumedia.com/hlt362v/the-visual-learner/images/systematic-sampling.png

```
# interval sampling example
idx = np.arange(0, len(df), step=2) #Return evenly spaced values within a given
interval.
interval_sample = df.iloc[idx]
interval_sample
```

	id	height
0	0	166.34
2	2	166.33
4	4	147.31
6	6	183.67
8	8	177.61

5.3.2.3. Stratified Sampling

Stratified random sampling is a kind of probability sampling in which a research organization divides the total population into many non-overlapping, homogenous groups (strata) and selects final members for research at random from the various strata. Each of these groupings' members should be unique enough that every member of each group has an equal chance of being chosen using basic probability. The number of instances from each stratum to choose from is proportionate to the stratum's size.

Arranging or classifying by age, socioeconomic divisions, nationality, religion, educational achievements is a common practice.

Example: There are 800 female employees and 200 male employees at the company. You select the population into two strata based on gender to ensure that the sample reflects the company's gender balance. Then you select 80 women and 20 men at random from each group, giving you a representative sample of 100 people.

Example: Consider the following scenario: a study team is looking for opinions on investing in Crypto from people of various ages. Instead of polling all Thai nationals, a random sample of roughly 10,000 people could be chosen for research. These ten thousand people can be separated into age groups, such as 18-29, 30-39, 40-49, 50-59, and 60 and up. Each stratum will have its own set of members and numbers.

Image source: https://lc.gcumedia.com/hlt362v/the-visual-learner/images/stratified-sample.png

Here we will use Scikit-learn for stratified sampling. Note that you will see this in more details later in the chapter on Introduction to Machine Leaning.

```
# create synthetic data
# id = np.arange(0, 10).tolist()
# height = np.round(np.random.normal(loc=165, scale =15, size=len(id)), 2)

# convert to pandas dataframe
# data = {"id":id, "height": height}
# df = pd.DataFrame(data=data)
# df
```

The `StratifiedKFold` module in Scikit-learn sets up `n_splits` (folds, partitions or groups) of the dataset in a way that the folds are made by **preserving the percentage of samples for each class**.

The brief explaination for the code below (also see the diagram below) is as follows:

1. The dataset has been split into K (K = 2 in our example) equal partitions (or folds).
2. (In iteration 1) use fold 1 as the testing set and the union of the other folds as the training set.
3. Repeat step 2 for K times, using a different fold as the testing set each time.

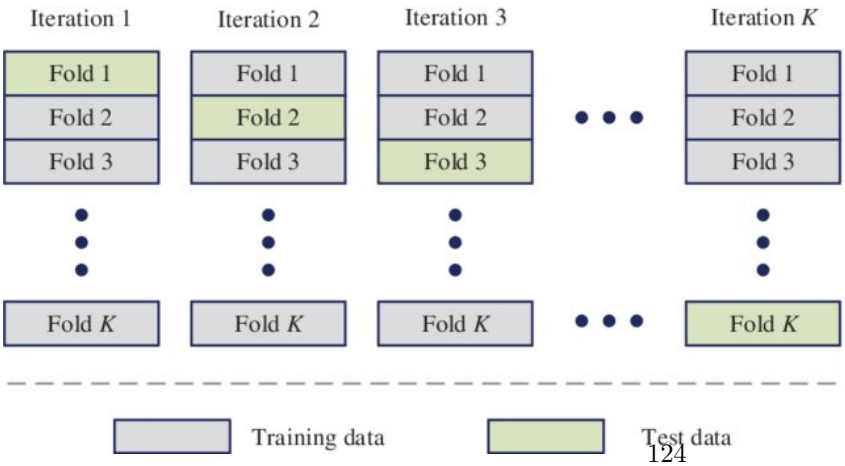


Image source: https://www.researchgate.net/profile/Mingchao-Li/publication/331209203/figure/fig2/AS:728070977748994@1550597056956/K-fold-cross-validation-method_W640.jpg

This sampling strategy tends to improve the representativeness of the sample by reducing the amount of bias we introduce; in the worst-case scenario, our resulting sample would be no worse than random sampling. Determining the strata, on the other hand, can be a tough operation because it necessitates a thorough understanding of the data's features. It's also the most time-consuming of the approaches discussed.

```
# create synthetic data
# population size of 20

id = np.arange(0, 20).tolist()
height = np.round(np.random.normal(loc=165, scale =15, size=len(id)), 2)

# convert to pandas dataframe
data = {"id":id, "height": height}
df = pd.DataFrame(data=data)
```

```
from sklearn.model_selection import StratifiedKFold

# dividing the data into groups
df["strata"] = np.repeat([1, 2], len(df)/2).tolist()

# instantiating stratified sampling
stratified = StratifiedKFold(n_splits=2)

for x, y in stratified.split(df, df["strata"]):
    print("TRAIN INDEX:", x, "TEST INDEX:", y)
    stratified_random_sample = df.iloc[x]

#stratified_random_sample
```

```
TRAIN INDEX: [ 5  6  7  8  9 15 16 17 18 19] TEST INDEX: [ 0  1  2  3  4 10 11 12 13 14]
TRAIN INDEX: [ 0  1  2  3  4 10 11 12 13 14] TEST INDEX: [ 5  6  7  8  9 15 16 17 18 19]
```

By supplying `shufle=True`, each class's samples will be shuffled before splitting into batches. Note also that the samples within each split will not be shuffled. We also use `n_splits=4` in the following code below:

```
from sklearn.model_selection import StratifiedKFold

# dividing the data into groups
df["strata"] = np.repeat([1, 2], len(df)/2).tolist()

# instantiating stratified sampling
stratified = StratifiedKFold(n_splits=4, shuffle=True, random_state=888)

for x, y in stratified.split(df, df["strata"]):
    print("TRAIN INDEX:", x, "TEST INDEX:", y)
    stratified_random_sample = df.iloc[x]

#stratified_random_sample
```

```
TRAIN INDEX: [ 2  3  4  6  7  8  9 11 12 13 14 15 16 17 18] TEST INDEX: [ 0  1  5 10 19]
TRAIN INDEX: [ 0  1  2  5  6  7  9 10 11 12 13 14 16 17 19] TEST INDEX: [ 3  4  8 15 18]
TRAIN INDEX: [ 0  1  2  3  4  5  6  8 10 11 14 15 17 18 19] TEST INDEX: [ 7  9 12 13 16]
TRAIN INDEX: [ 0  1  3  4  5  7  8  9 10 12 13 15 16 18 19] TEST INDEX: [ 2  6 11 14 17]
```

The following code can be used to acsess a single batch instead of using `for loop`.

```
## sklearn Kfold access single fold instead of for loop
## https://stackoverflow.com/questions/27380636/sklearn-kfold-acces-single-fold-instead-of-for-loop

skf = stratified.split(df, df["strata"])
mylist = list(skf)

x,y = mylist[3]

stratified_random_sample_train = df.iloc[x]
stratified_random_sample_test = df.iloc[y]

print(stratified_random_sample_train)
```

```
    id  height  strata
0    0  160.56       1
1    1  141.48       1
3    3  147.74       1
4    4  186.11       1
5    5  130.88       1
7    7  159.12       1
8    8  172.77       1
9    9  147.52       1
10  10  136.51       2
12  12  183.23       2
13  13  171.81       2
15  15  141.52       2
16  16  162.69       2
18  18  156.08       2
19  19  167.08       2
```

Alternative to the above approach, we use the `next()` function that returns the next item from the iterator. See https://www.programiz.com/python-programming/methods/built-in/next for more detail.

```
# https://stackoverflow.com/questions/27380636/sklearn-kfold-acces-single-fold-instead-of-for-loop
# https://stackoverflow.com/questions/2300756/get-the-nth-item-of-a-generator-in-python
# In Python, Itertools is the inbuilt module that allows us to handle the iterators in an efficient way. They make iterating through the iterables like lists and strings very easily. One such itertools function is islice().


#from itertools import islice, count
import itertools


skf = stratified.split(df, df["strata"])

index = 0
x, y = next(itertools.islice(skf,index,None))

stratified_random_sample_train = df.iloc[x]
stratified_random_sample_test = df.iloc[y]

print(stratified_random_sample_train)
print(stratified_random_sample_test)
```

```
    id  height  strata
2    2  171.95       1
3    3  147.74       1
4    4  186.11       1
6    6  147.88       1
7    7  159.12       1
8    8  172.77       1
9    9  147.52       1
11  11  147.56       2
12  12  183.23       2
13  13  171.81       2
14  14  178.27       2
15  15  141.52       2
16  16  162.69       2
17  17  162.48       2
18  18  156.08       2
    id  height  strata
0    0  160.56       1
1    1  141.48       1
5    5  130.88       1
10  10  136.51       2
19  19  167.08       2
```

This sampling strategy tends to improve the representativeness of the sample by reducing the amount of bias we introduce; in the worst-case scenario, our resulting sample would be no worse than random sampling. Determining the strata, on the other hand, can be a tough operation because it necessitates a thorough understanding of the data's features. It's also the most time-consuming of the approaches discussed.

Exercise: This exercise aims to explain how `StratifiedKFold` can be used for stratified sampling.

1. What is the class ratio for the column "strata", i.e. the proportion of data which are in strata (groups) 1 and 2?

Note that when we create our folds we want each split to have this same percentage of categories (groups).

When we perform the splits we will need to tell the function which column we are going to use as the target, strata in this case. The command will be `stratified.split(df, df["strata"])`.

Then we use a for loop and StratifiedKFold's split operation to get the train and test row indexes for each split.

We can then use these indexes to split our data into train and test dataframes.

1. What are the indexes used in the train data and test data in the first batch (or split)?
2. Determine the class ratio for each batch (split) from the test set (you may also want to try for the training set).

```
# create synthetic data
# Population size of 100

id = np.arange(0, 100).tolist()
height = np.round(np.random.normal(loc=165, scale =15, size=len(id)), 2)

# convert to pandas dataframe
data = {"id":id, "height": height}
df = pd.DataFrame(data=data)

# dividing the data into groups
df["strata"] = np.repeat([1, 2], [0.2*len(df),0.8*len(df)]).tolist()
```

Solutions to exercise

1. What is the class ratio for the column "strata", i.e. the proportion of data which are in strata (groups) 1 and 2?

```
df.groupby('strata').id.count()
```

```
strata
1    20
2    80
Name: id, dtype: int64
```

Ans: the class ratio of strata 1 to 2 is 1:4.

1. What are the indexes used in the train data and test data in the first batch (or split)?

Ans: skf is actually a **generator**, which does not compute the train-test split until it is needed. This **improves memory usage**, as you are not storing items you do not need. Making a list of the skf object forces it to make all values available.

The following Python commands can be applied to access the indexes of the train data and test data in the first split (or the n-th split). Simply change the value of the `index` variable to access different splits:

```
## sklearn Kfold acces single fold instead of for loop
## https://stackoverflow.com/questions/27380636/sklearn-kfold-acces-single-fold-instead-of-for-loop

# instantiating stratified sampling

K = 4
stratified = StratifiedKFold(n_splits=K)

skf = stratified.split(df, df["strata"])
mylist = list(skf)

index = 0
x,y = mylist[index]

stratified_random_sample_train = df.iloc[x]
stratified_random_sample_test = df.iloc[y]

print('Training set: \n', stratified_random_sample_train)
print('The class ratio of groups 1 to 2 in this batch of the training set is')
print(stratified_random_sample_train['strata'].value_counts())

print('Test set: \n', stratified_random_sample_test)
print('The class ratio of groups 1 to 2 in this batch of the test set is')
print(stratified_random_sample_test['strata'].value_counts())
```

```
Training set:
     id  height  strata
5    5  168.29       1
6    6  163.27       1
7    7  164.88       1
8    8  163.55       1
9    9  157.62       1
..  ..     ...     ...
95  95  151.83       2
96  96  163.17       2
97  97  155.09       2
98  98  164.62       2
99  99  171.49       2

[75 rows x 3 columns]
The class ratio of groups 1 to 2 in this batch of the training set is
2    60
1    15
Name: strata, dtype: int64
Test set:
     id  height  strata
0    0  188.69       1
1    1  163.28       1
2    2  153.06       1
3    3  170.40       1
4    4  175.91       1
20  20  159.42       2
21  21  172.81       2
22  22  161.30       2
23  23  152.85       2
24  24  184.57       2
25  25  136.93       2
26  26  160.15       2
27  27  146.00       2
28  28  156.73       2
29  29  161.16       2
30  30  158.09       2
31  31  173.13       2
32  32  164.89       2
33  33  190.48       2
34  34  160.57       2
35  35  165.80       2
36  36  184.03       2
37  37  147.90       2
38  38  189.03       2
39  39  162.01       2
The class ratio of groups 1 to 2 in this batch of the test set is
2    20
1     5
Name: strata, dtype: int64
```

1. Determine the class ratio for each batch (split) from the test set (you may also want to try for the training set).

Ans: We can use for loop to go through each split of the **generator, StratifiedKFold(n_splits=K) within the test split.**

```
split_no = 1
for x, y in stratified.split(df, df["strata"]):
    # print("TRAIN INDEX:", x, "TEST INDEX:", y)
    stratified_random_sample = df.iloc[x]
    stratified_random_sample = df.iloc[y]
    print('Batch',str(split_no),': The class ratio of groups (strata) 1 to 2 is',
stratified_random_sample_test['strata'].value_counts()
[1]/len(stratified_random_sample_test['strata']))
    split_no += 1
```

```
Batch 1 : The class ratio of groups (strata) 1 to 2 is 0.2
Batch 2 : The class ratio of groups (strata) 1 to 2 is 0.2
Batch 3 : The class ratio of groups (strata) 1 to 2 is 0.2
Batch 4 : The class ratio of groups (strata) 1 to 2 is 0.2
```

```
# Determind the class ratio
#stratified_random_sample_test['strata'].value_counts()
[1]/len(stratified_random_sample_test['strata'])
```


Conclusion In this example, the population size is 100. If we want a sample based on stratified sampling with the sample size of 25, then we can use the test sets from any of the splits as a sample. For example, the sample using the stratified sampling technique is

Alternative to `StratifiedKFold`, one can use `train_test_split` for stratified sampling. The train_test_split is the most basic one which just divides the data into two parts according to the specified partitioning ratio. For instance, `train_test_split(test_size=0.2)`.

See https://towardsdatascience.com/how-to-train-test-split-kfold-vs-stratifiedkfold-281767b93869 for more detail.

```
# Using train_test_split for stratified sampling:

from sklearn.model_selection import train_test_split

# create synthetic data
id = np.arange(0, 50).tolist()
height = np.round(np.random.normal(loc=165, scale =15, size=len(id)), 2)

# convert to pandas dataframe
data = {"id":id, "height": height}
df = pd.DataFrame(data=data)
df


# dividing the data into groups
df["strata"] = np.repeat([1, 2], [0.2*len(df),0.8*len(df)]).tolist()

X_train, X_test = train_test_split(df, test_size = 0.2, stratify=df["strata"],
random_state = 888)
```

5.3.2.4. Cluster Sampling

Cluster Sampling is a method where the entire population is divided into clusters or portions Some of these clusters are then chosen at random. For this sampling, **all of the selcted clusters' elements** are used.

The term **cluster** refers to a natural intact (but heterogeneous) grouping of the members of the population.

Researchers use this sampling technique to examine a sample that contain multiple sample parameters such as demographics, habits, background – or any other population attribute that is relevant to the research being undertaken.

Example: The company has offices in twenty different provinces around Thailand (all with roughly the same number of employees in similar roles). Because we do not have the resources to visit every office to collect data, we use random sampling to select three offices as your clusters.

Example: A researcher in Thailand intends to undertake a study to evaluate sophomores' performance in science education. It is impossible to undertake a research study in which every university's student participates. Instead, the researcher can combine the universities from each region into a single cluster through cluster sampling. The sophomore student population in Thailand is then defined by these groupings. Then, using either simple random sampling or systematic random sampling, select clusters for the research project at random.

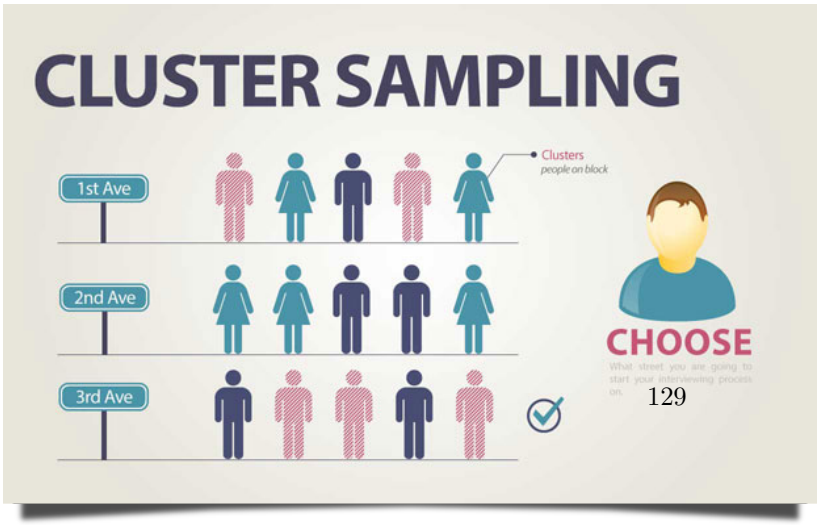


Image source: https://lc.gcumedia.com/hlt362v/the-visual-learner/images/cluster-sampling.png

In the following example, the population is divided into 5 clusters of equal size. Note that heterogeneity is internal within clusters (or groupings), while homogeneity is external (among clusters). A systematic random sampling of the clusters (by select clusters with even cluster_id) is chosen and the elements in each of these clusters are then sampled.

```
# cluster sampling example
# removing the strata
df.drop("strata", axis=1, inplace=True)

# Divide the units into 5 clusters of equal size
df['cluster_id'] = np.repeat([range(1,6)], len(df)/5)

# Append the indexes from the clusters that meet the criteria
idx = []
# add all observations with an even cluster_id to idx
for i in range(0, len(df)):
    if df['cluster_id'].iloc[i] % 2 == 0:
        idx.append(i)

cluster_random_sample = df.iloc[idx]
cluster_random_sample
```

	id	height	cluster_id
10	10	163.59	2
11	11	161.27	2
12	12	157.23	2
13	13	171.97	2
14	14	150.42	2
15	15	179.87	2
16	16	176.95	2
17	17	136.71	2
18	18	141.73	2
19	19	163.07	2
30	30	158.31	4
31	31	143.46	4
32	32	170.21	4
33	33	176.85	4
34	34	152.70	4
35	35	174.23	4
36	36	159.46	4
37	37	142.95	4
38	38	189.67	4
39	39	161.76	4

This cluster sampling approach is particularly cost-effective because it involves minimal sample preparation labor and is also simple to use. On the other hand, this samplint approach makes it easy to generate biased data.

5.4. Distribution of Random Variables

In this tutorial, we will learn about probability distributions that are often used in machine learning literature and how to implement them in Python.

The underlying components of Data Science are probability and statistics. In truth, statistical mathematics and linear algebra are the core principles of machine learning and artificial intelligence.

You will frequently find yourself in circumstances, particularly in Data Science, where you will need to read a research article that contains a lot of math to understand a certain issue, therefore if you want to improve at Data Science, you will need to have a solid mathematical and statistical understanding.

In this section, we will look at some of the most often used probability distributions in machine learning research.

We will cover the following topics:

- Learn about probability terminologies such as random variables, density curves, and probability functions.
- Discover the various probability distributions and their distribution functions, as well as some of their features.
- Learn how to use Python to construct and plot these distributions.

Before you begin, you need be familiar with some mathematical terms, which will be covered in the next section.

5.4.1. Random Variable

A **random variable** is a variable whose possible values are numerical results of a random event. Discrete and continuous random variables are the two forms of random variables.

5.4.1.1. Discrete random variables

A **discrete random variable** is one that can only have a finite number of different values and can thus be quantified.

For example, a random variable X can be defined as the number that appears when a fair dice is rolled. X is a discrete random variable with the following values: [1,2,3,4,5,6].

The probability distribution of a discrete random variable is a list of probabilities associated with each of its potential values. It's also known as the **probability mass function (pmf) or the probability function**. Consider a random variable X that can take k distinct values, with the probability that $X = x_i$ being defined as $P(X = x_i) = p_i$. The probabilities p_i must then satisfy the following conditions:

1. $0 < p_i < 1$ for each i
2. $p_1 + p_2 + \ldots + p_k = 1$.

Bernoulli distribution, Binomial distribution, Poisson distribution, and other discrete probability distributions are examples.

The following python code generate random samples from the random variable X that can be defined as the number that appears when a fair dice is rolled, i.e. generate a uniform random sample from `np.arange(1,7)` of size n.

```
# generate random integer values
from random import seed
from random import randint
# seed random number generator
seed(1)

# sample size
n = 3

# generate some integers
for _ in range(n):
    value = randint(1, 6)
    print(value)
```



```
2
5
1
```

It is much more convenient to use `numpy.random.choice` to generates a random sample from a given 1-D array.

```
X = np.arange(1,7)
np.random.choice(X, 10, replace = True)
```

```
array([3, 2, 5, 4, 3, 3, 4, 5, 4, 2])
```

5.4.2. Continuous random variables

A **continuous random variable** can take an infinite number of different values. For example, a random variable X can be defined as the height of pupils in a class.

The area under a curve is used to represent a continuous random variable because it is defined throughout a range of values (or the integral).

Probability distribution functions (pdf) are functions that take on continuous values and represent the probability distribution of a continuous random variable. Because the number of possible values for the random variable is unlimited, the probability of seeing any single value is zero.

A random variable X, for example, could take any value within a range of real integers. The area above and below a curve is defined as the probability that X is in the set of outcomes A, $P(A)$. The curve that represents the function $p(x)$ must meet the following requirements:

1. There are no negative values on the curve ($p(x) > 0$ for all x).
2. The total area under the curve is 1.

The term **density curve** refers to a curve that meets certain criteria.

Normal distribution, exponential distribution, beta distribution, and other continuous probability distributions are examples.

5.4.2.1. The uniform distribution

The **uniform distribution** is one of the most basic and useful distributions. The probability distribution function of the continuous uniform distribution's is:

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{for } a \leq x \leq b, \\ 0 & \text{for } x < a \text{ or } x > b \end{cases}$$

The curve depicting the distribution is a rectangle, with constant height across the interval and 0 height everywhere, because each interval of numbers of equal width has an equal chance of being seen.

Because the area under the curve must equal 1, the height of the curve is determined by the length of the gap.

A uniform distribution in intervals (a, b) is depicted in the diagram below. Because the area must be 1, $1/(b-a)$ is the height setting.

5.4.2.2. Uniform Distribution in Python

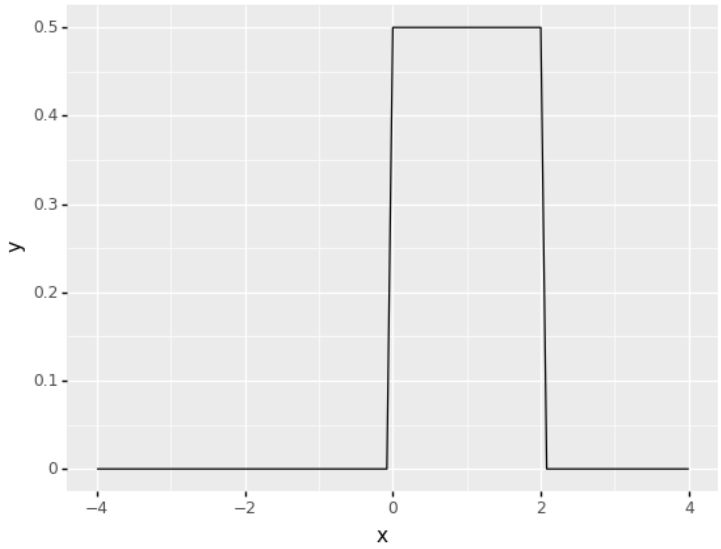
We can also use plotnine to construct more complicated statistical visualisations than simple plots like bar and scatterplots.

We can combine the capabilities of the **scipy** package with **plotnine** to chart some probability density functions in the plot below.

Here we use `stat_function` to superimpose a function (or add the graph of a function) onto a plot.

Also, we also specify the density function of a uniform distributin in scipy as `uniform.pdf()`.

```
# https://stackoverflow.com/questions/48349713/how-to-graph-a-function-in-python-using-plotnine-library
# https://t-redactyl.io/blog/2019/10/making-beautiful-plots-in-python-plus-a-shameless-book-plug.html
a = 0
b = 2

(ggplot(pd.DataFrame(data={"x": [-4, 4]}), aes(x="x"))
    + stat_function(fun=lambda x:  uniform.pdf(x, loc = a, scale = b-a)))
```




```
<ggplot: (310012209)>
```

5.4.2.3. Uniform random variate

In probability and statistics, a **random variate** is a particular outcome of a random variable: the random variates which are other outcomes of the same random variable might have different values (random numbers).

The `uniform.rvs` function, with its `loc` and `scale` arguments, creates a uniform continuous variable between the provided intervals.

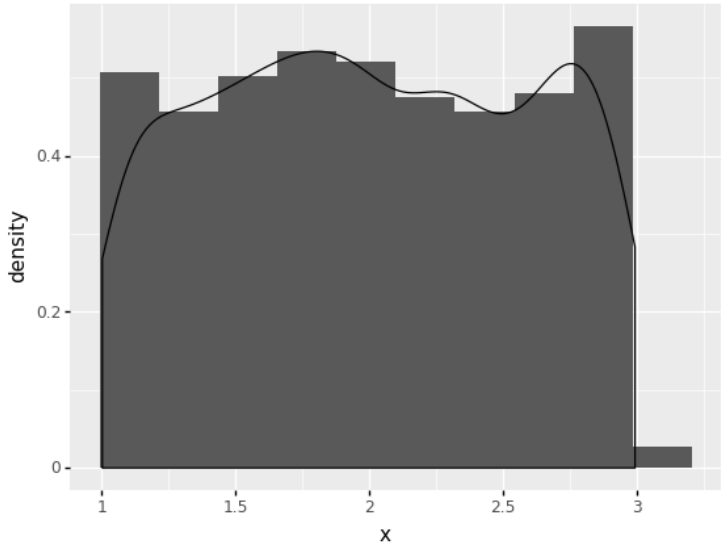
In the standard form, the distribution is uniform on [0, 1]. Using the parameters loc and scale, one obtains the uniform distribution on [loc, loc + scale].

The `size` arguments specify how many random variates there are. Include a random state argument with a number if you wish to keep things consistent.

```
# random numbers from uniform distribution
n = 1000
start = 1
width = 2

data_uniform = pd.DataFrame({'x':uniform.rvs(size=n, loc = start, scale=width)})
```

To visualize the histogram of the distribution you just built together with the kernel density estimate, use plotnine as follows:

```
(
    ggplot(data_uniform) +  # What data to use
    aes('x') +  # What variable to use
    geom_histogram(aes(y=after_stat('density')), bins = 10) + # Geometric object to use for drawing
    geom_density(aes(y=after_stat('density')))
)
```




```
<ggplot: (278390077)>
```

5.4.3. Normal Distribution Function

In Data Science, the **Normal Distribution**, commonly known as the **Gaussian Distribution**, is often used, especially when it comes to statistical inference. Many data science techniques make this assumption as well.

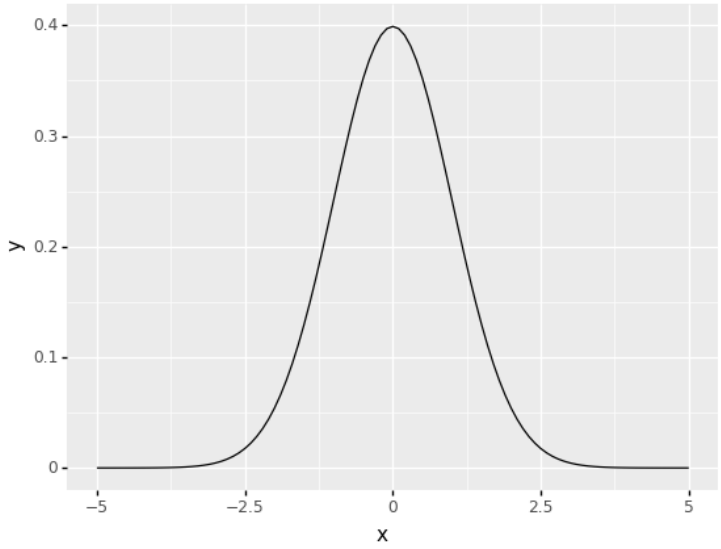
The mean μ and standard deviation σ of a normal distribution define a bell-shaped density curve. The density curve is symmetrical, centered around its mean, and its spread is determined by its standard deviation, indicating that data close to the mean occur more frequently than those further from it.

At a given point x, the probability distribution function of a normal density curve with mean μ and standard deviation σ is: $$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

```
# df = pd.DataFrame({'x':np.linspace(-10.0,10.0,200)})
# df = df.assign(y = norm.pdf(df.x))
```

The following code plots the density of the normal distribution where the location (`loc`) keyword specifies the mean. The scale (`scale`) keyword specifies the standard deviation

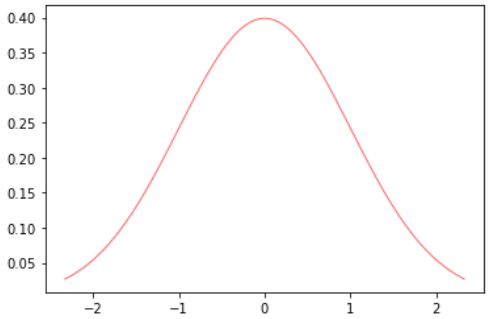
```
mu = 0
sd = 1

(ggplot(pd.DataFrame(data={"x": [-5, 5]}), aes(x="x"))
    + stat_function(fun=lambda x:  norm.pdf(x, loc = mu, scale = sd)))
```




```
<ggplot: (311062993)>
```

```
from scipy.stats import *
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 1)

x = np.linspace(norm.ppf(0.01),
                norm.ppf(0.99), 100)
ax.plot(x, norm.pdf(x),
       'r-', lw=1, alpha=0.6, label='norm pdf')
```

```
[<matplotlib.lines.Line2D at 0x128930e50>]
```



Exercise:

1. What percent of data falls within 1 standard deviation above the mean?
2. What percent of data falls within 2 standard deviation above the mean?

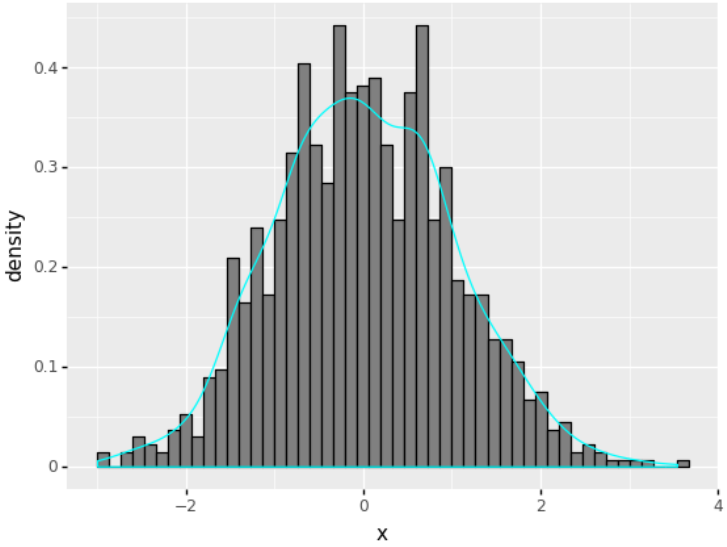
Hint: use `norm.cdf(x, loc, scale)` cumulative distribution function.

Exercise:

1. Generate $n = 10000$ random numbers from standard normal distribution.
2. Plot the histogram of those n random numbers and superimpose the kernel density estimate of the histogram.

```
# random numbers from normal distribution
n = 1000
mu = 0
sd = 1

data_norm = pd.DataFrame({'x':norm.rvs(size=n, loc = mu, scale=sd)})
```

```
(
    ggplot(data_norm) +  # What data to use
    aes('x') +  # What variable to use
    geom_histogram(aes(y=after_stat('density')), bins = 50, fill ='gray', colour
='black') + # Geometric object to use for drawing
    geom_density(aes(y=after_stat('density')),colour ='cyan')
)
```




```
<ggplot: (311126465)>
```

5.4.4. Poisson Distribution

Typically, a Poisson random variable is used to model the number of times an event occurs in a certain time frame. A Poisson process, for example, might be considered as the number of users who visit a website at a certain interval.

The rate (μ) at which events occur is given by the Poisson distribution. In a certain interval, an event can occur 0, 1, 2,... times.

The average number of events in an interval is designated λ (lambda). The event rate, commonly known as the rate parameter, is λ.

The following equation gives the likelihood of seeing k events in a given interval: $
$f(k;\lambda) = \Pr(X{=}k) = \frac{\lambda^k e^{-\lambda}}{k!}$,$

5.4.4.1. Poisson Distribution Function

It is worth noting that

1. the normal distribution is a special case of the Poisson distribution with parameter $\lambda \to \infty$.
2. In addition, if the intervals between random events follow an exponential distribution with rate λ, then the total number of occurrences in a time period of length t follows the Poisson distribution with parameter λt.

Exercise:

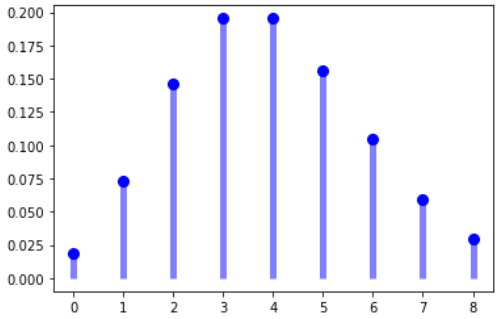
1. Generate $n = 1000$ random numbers from a Poisson distribution with rates $\lambda = 1$ and $\lambda = 4$.
2. Plot the histogram of those n random numbers.
3. Create the frequency distribution of the random numbers generated and compare with those numbers (frequencies) obtained from the Poisson distribution with the specified parameter values.

Before attempting to answer questions, we first plot the pmf of a Poisson distribution.

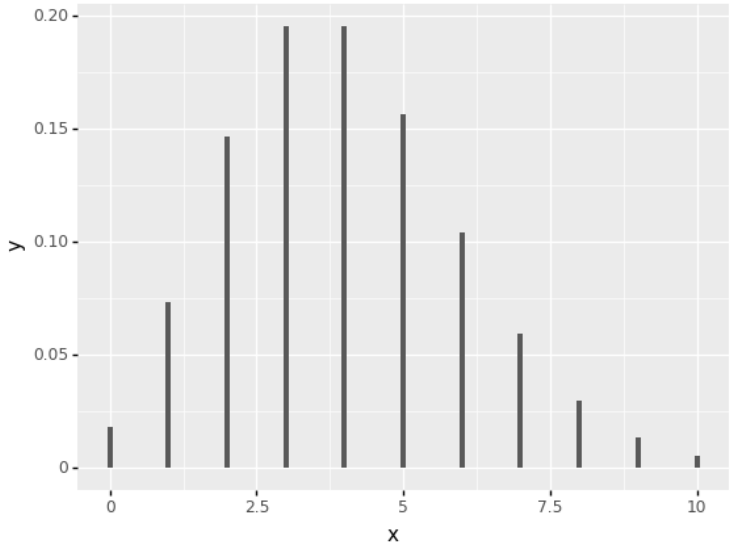
The following python codes plot the pmf of a Poisson distribution using matplotlib and plotnine.

```
fig, ax = plt.subplots(1, 1)

mu = 4

x = np.arange(poisson.ppf(0.01, mu),
              poisson.ppf(0.99, mu))
ax.plot(x, poisson.pmf(x, mu), 'bo', ms=8, label='poisson pmf')
ax.vlines(x, 0, poisson.pmf(x, mu), colors='b', lw=5, alpha=0.5)
```

```
<matplotlib.collections.LineCollection at 0x128b25590>
```




```
mu = 4

(ggplot(pd.DataFrame(data={"x": [0,10]}), aes(x="x"))
    + stat_function(geom='bar', fun=lambda x:  poisson.pmf(x, mu)))
```




```
<ggplot: (311160229)>
```

1. Generate $n = 1000$ random numbers from a Poisson distribution with rates $\lambda = 1$ and $\lambda = 4$.

Ans: We use `poisson.rvs` to generate n random number from the Poisson distributin with $\lambda = 4$.

```
# random numbers from Poisson distribution

n = 1000
mu = 4

data_poisson = pd.DataFrame({'x':poisson.rvs(mu, size=n, random_state=888)})
```

By the method of moments, we can obtain the parameter of the Poisson distribution by matching

Model : X ~ Poisson(lambda)

E[X] = lambda = X_bar = sample mean

The estimate of lambda (lambda~ = sample mean)

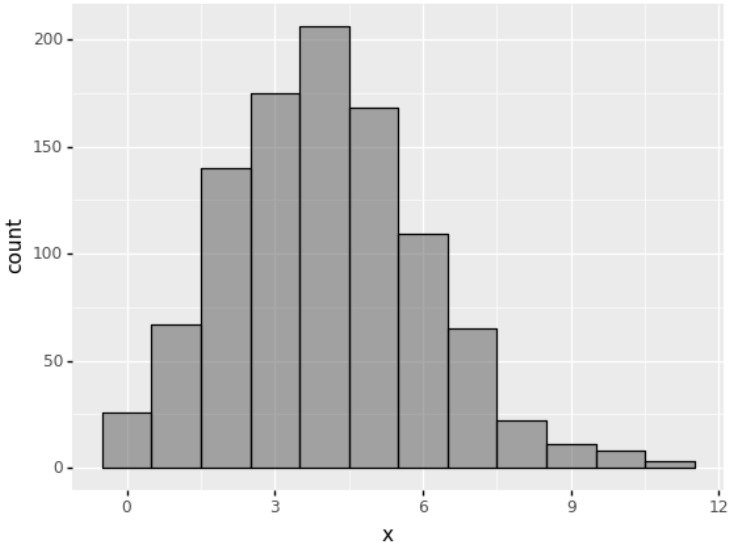
```
lambda_est = data_poisson['x'].mean()

print(lambda_est)
```

```
4.033
```

1. Plot the histogram of those n random numbers.

Ans: The histogram can be obtained as follows:

```
(
    ggplot(data_poisson) +
    aes('x') +
    geom_histogram(aes(y=after_stat('count')), binwidth=1,color='black',alpha=0.5 )
)
```




```
<ggplot: (279189929)>
```

1. Create the frequency distribution of the random numbers generated and compare with those numbers (frequencies) obtained from the Poisson distribution with the specified parameter values.

Ans: We can simply create a frequency table using `value_counts`. Note that we also specify `sort=False`.

```
data_poisson.value_counts(sort = False)
```

```
x
0      26
1      67
2     140
3     175
4     206
5     168
6     109
7      65
8      22
9      11
10      8
11      3
dtype: int64
```

Alternatively, we can also create the frequency table using `groupby` method.

```
data_poisson['simulation']= data_poisson['x']
#data_poisson['freq']= data_poisson['x']
data_poisson.groupby('x').count()
```

x	simulation
0	26
1	67
2	140
3	175
4	206
5	168
6	109
7	65
8	22
9	11
10	8
11	3

With the specified value of n (the number of simulated random numbers), we calculate the expected numbers from the Poisson distribution and add them into a new column called **poisson**.

The results also show that the distribution of the simulated random numbers comes from the Poisson distribution. How do we confirm our findings?

```
output_freq = data_poisson.groupby('x').count()
output_freq['x'] = output_freq.index
output_freq.sort_index(axis=1,ascending=False,inplace=True)
output_freq['poisson'] =  n*poisson.pmf(output_freq['x'],mu)
output_freq
```

x	x	simulation	poisson
0	0	26	18.315639
1	1	67	73.262556
2	2	140	146.525111
3	3	175	195.366815
4	4	206	195.366815
5	5	168	156.293452
6	6	109	104.195635
7	7	65	59.540363
8	8	22	29.770181
9	9	11	13.231192
10	10	8	5.292477
11	11	3	1.924537

```
#pd.melt(output_freq, id_vars = 'x', value_vars=['freq','poisson'], value_name='values')
pd.melt(output_freq, id_vars = 'x', value_vars=['simulation','poisson'],
value_name='values')
```

	x	variable	values
0	0	simulation	26.000000
1	1	simulation	67.000000
2	2	simulation	140.000000
3	3	simulation	175.000000
4	4	simulation	206.000000
5	5	simulation	168.000000
6	6	simulation	109.000000
7	7	simulation	65.000000
8	8	simulation	22.000000
9	9	simulation	11.000000
10	10	simulation	8.000000
11	11	simulation	3.000000
12	0	poisson	18.315639
13	1	poisson	73.262556
14	2	poisson	146.525111
15	3	poisson	195.366815
16	4	poisson	195.366815
17	5	poisson	156.293452
18	6	poisson	104.195635
19	7	poisson	59.540363
20	8	poisson	29.770181
21	9	poisson	13.231192
22	10	poisson	5.292477
23	11	poisson	1.924537

To compare the plots of frequency plots (bar plots) side by side (from two variables **simulation** and **poisson** of our data frame, we need to **reshape** our data frame into a more computer-friendly form using Pandas in Python.

To achieve this, `pandas.melt()` unpivots a DataFrame from wide format to long format.

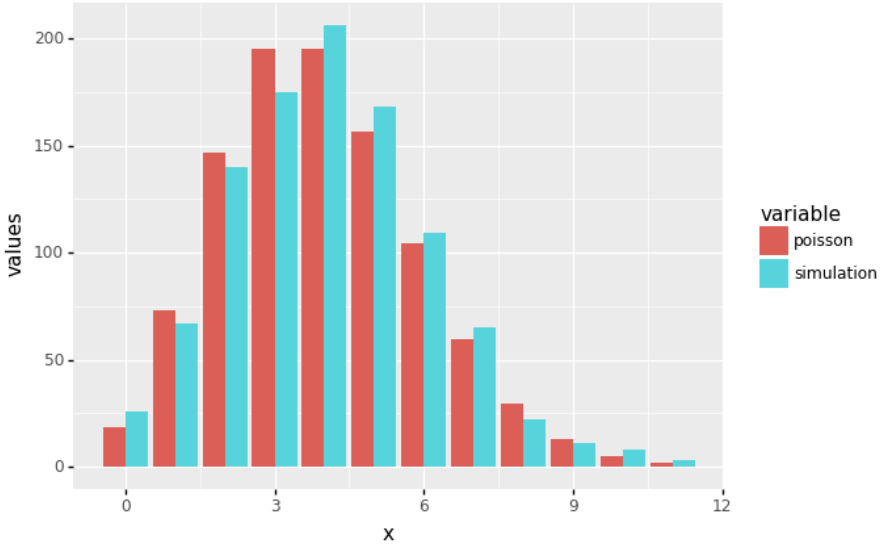
`melt()` function is useful to reshape a DataFrame into a format where one or more columns are identifier variables, while all other columns, considered measured variables, are unpivoted to the row axis, leaving just two non-identifier columns, variable and value.

See https://stackoverflow.com/questions/42820677/ggplot-bar-plot-side-by-side-using-two-variables

https://www.geeksforgeeks.org/python-pandas-melt/ for more detail.

```
# https://stackoverflow.com/questions/42820677/ggplot-bar-plot-side-by-side-using-two-variables
# https://www.geeksforgeeks.org/python-pandas-melt/

output_freq_melted = pd.melt(output_freq, id_vars = 'x', value_vars=
['simulation','poisson'], value_name='values')

print(output_freq_melted)

(
    ggplot(output_freq_melted, aes(x='x', y='values', fill='variable')) +
    geom_bar(stat='identity', position='dodge')
)
```

```
     x    variable      values
0    0  simulation   26.000000
1    1  simulation   67.000000
2    2  simulation  140.000000
3    3  simulation  175.000000
4    4  simulation  206.000000
5    5  simulation  168.000000
6    6  simulation  109.000000
7    7  simulation   65.000000
8    8  simulation   22.000000
9    9  simulation   11.000000
10  10  simulation    8.000000
11  11  simulation    3.000000
12   0     poisson   18.315639
13   1     poisson   73.262556
14   2     poisson  146.525111
15   3     poisson  195.366815
16   4     poisson  195.366815
17   5     poisson  156.293452
18   6     poisson  104.195635
19   7     poisson   59.540363
20   8     poisson   29.770181
21   9     poisson   13.231192
22  10     poisson    5.292477
23  11     poisson    1.924537
```




```
<ggplot: (312077221)>
```

5.5. Fitting Models to Data

We will learn how to find the best-matching statistical distributions for your data points. Modeling quantities of interest for example claim numbers and sizes is the subject of this section, which involves fitting probability distributions from selected families to sets of data containing observed claim numbers or sizes.

After an exploratory investigation of the data set, the family may be chosen by looking at numerical summaries such as mean, median, mode, standard deviation (or variance), skewness, kurtosis, and graphs like the empirical distribution function.

Of course, one might want to fit a distribution from each of several families to compare the fitted models, as well as compare them to earlier work and make a decision.

In statistics, **probability distributions** are a fundamental concept. They are employed in both theoretical and practical settings.

The following are some examples of probability distributions in use:

- It is frequently used in the case of univariate data to determine an appropriate distributional model for the data.
- Specific distributional assumptions are frequently used in statistical intervals and hypothesis tests.

- Calculate parameter confidence intervals as well as critical regions for hypothesis testing.
- Continuous probability distributions are frequently employed in machine learning models, particularly in the distribution of numerical input and output variables, as well as the distribution of model errors.

To fit a **parametric model** (i.e. a probability distribution), we must obtain estimates of the probability distribution's unknown parameters. The method of moments, the method of maximum likelihood, the method of percentiles, and the method of minimum distance are among the criteria offered.

5.5.1. Generate test data and fit it

For the first illustration, we will begin by generating some normally distributed test data using the **NumPy** module and fit them by using the **Fitter** library to see whether the fitter is able to identify the distribution.

The **Fitter** package offers a basic class that can be used to identify the distribution from which a data sample is drawn. It employs 80 Scipy distributions and allows you to plot the results to see which distribution is the most likely and which parameters are the best.

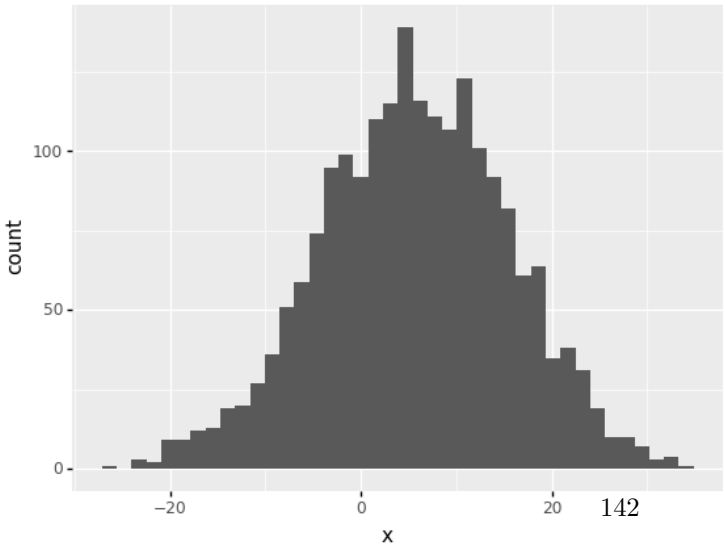
```
# Set random seed and generate test data
np.random.seed(88)
data = np.random.normal(loc=5, scale=10, size=2000, )
data = pd.DataFrame({'x':data})

# Alternatively, we can run
# data_norm = pd.DataFrame({'x':norm.rvs(size=n, loc = mu, scale=sd)})
```

```
data.head()
```

	x
0	6.068843
1	27.058152
2	14.565627
3	5.684111
4	15.685138

We can use the plotnine to plot the histogram. The number of bins provided here is 20. The plot shows that the data overall follows a normal distribution.

```
ggplot(data) + aes('x') + geom_histogram(bins = 40)
```




```
<ggplot: (313611497)>
```

5.5.2. Fitting distributions

The next step is to start fitting different distributions to the data and determining which one is best for the data.

The steps are as follows:

1. Call the Fitter method to create a Fitter instance.
2. If you have a basic idea of the distributions that might fit your data, provide the data and distributions list.
3. Use the `.fit()` method.
4. Using the `.summary()` method, create a summary of the fitted distribution.

Note: If you have no idea what distribution might fit your data at first, you can run Fitter() and merely provide the data.

The Fitter class in the backend uses the **Scipy** library, which supports 80 different distributions. The **Fitter** class will scan all of them, call the fit function for you, ignore any that fail or run forever, and then provide you a summary of the best distributions in terms of sum of square errors.

However, because it will try so many different distributions, this may take some time, and the fitting time varies depending on the size of your sample. As a result, it is recommended that you display a histogram to have a general idea of the types of distributions that might match the data and supply those distribution names in a list. This will save you a lot of time.

Fitter library utilizes SciPy's fit method to extract the parameters of a distribution that best fit the data given a data sample. This process is repeated for all available distributions. Finally, we present a summary so that the quality of the fit for those distributions may be determined.

```
# pip install fitter
```

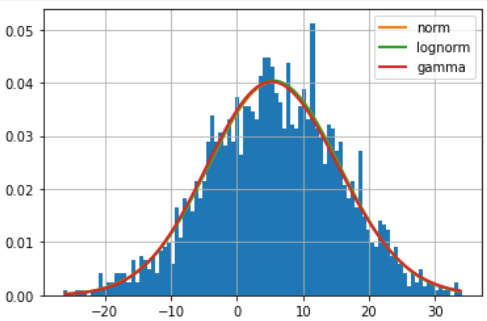
```
# Required libraries

from fitter import Fitter, get_common_distributions, get_distributions
```

Here we speficy a list of possible undelying distribution that could have generated a data set including gamma, lognormal, beta and normal distributions.

However, the distribution's parameters are unknown, thus there are numerous distributions. As a result, an automated method of fitting multiple distributions to the data would be beneficial.

```
f = Fitter(data,
           distributions=['gamma',
                          'lognorm',
                          "norm"])
f.fit()
f.summary()
```

	sumsquare_error	aic	bic	kl_div
norm	0.001267	960.095049	-28528.628648	inf
lognorm	0.001281	964.877953	-28499.085772	inf
gamma	0.001310	964.163478	-28453.859526	inf



5.5.3. Choosing the most appropriate distribution and indentifying the parameters

We may also use the `.get_best()` method to retrieve the best distribution, where we can additionally specify the technique for picking the best distribution.

As selection criteria, we can use the `sumsquare_error` in the method argument. It will print the name of the distribution with the lowest sum square error, as well as the relevant parameters. 'sumsquare_error' is a formula used to measure the difference between the given data and the expected data obtained by the fitted model, i.e. **lower is better**.

Based on the sum square error criteria, we can observe that the normal distribution is the best fit. It also prints the normal distribution's optimum parameters including location (loc), and scale parameters (scale).

Note that sum of the square errors between the data Y_i and the fitted distribution $pdf(X_i)$ is defined by $$\sum_i (Y_i - pdf(X_i))^2.$$ (see https://fitter.readthedocs.io/en/latest/references.html?highlight=sum square#fitter.fitter.Fitter.fit)

Alternative to the square errors, we can also test the goodness of fit using the **Kolmogorov–Smirnov test** after fitting a probability distribution to our data. The Kolmogorov–Smirnov test is a widely used option. The test essentially provides you with a statistic and a p-value, which you must interpret using a K-S test table.

See https://medium.com/@amirarsalan.rajabi/distribution-fitting-with-python-scipy-bb70a42c0aed for more detail.

For details about relative fit measures and goodness of fit see https://vortarus.com/assessing-distribution-fit/

```
f.get_best(method = 'sumsquare_error')
```

```
{'norm': {'loc': 5.630744055249008, 'scale': 9.899359274217787}}
```

In addition, we can also print the fitted parameters using the fitted_param attribute and indexing it out using the distribution name for example 'gamma'.

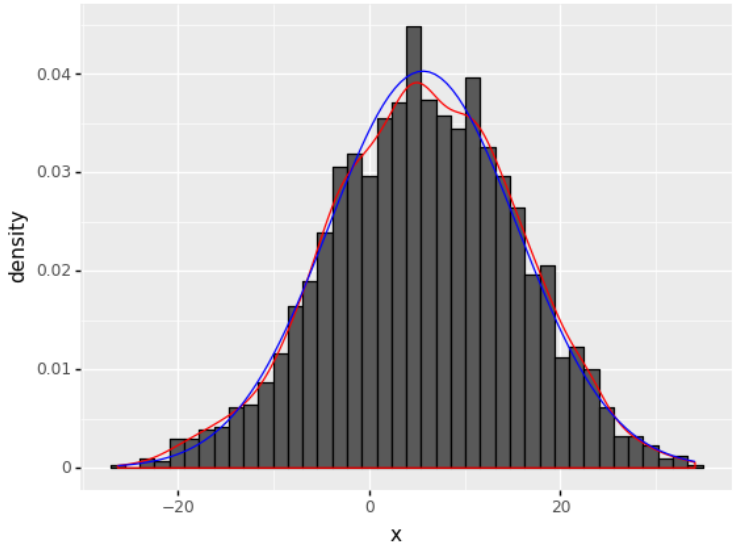
```
f.fitted_param['norm']
```

```
(5.630744055249008, 9.899359274217787)
```

```
f.fitted_param['lognorm']
```

```
(0.010345149115340332, -948.2039689041416, 953.8081165452257)
```

```
(
    ggplot(data) +  # What data to use
    aes('x') +  # What variable to use
    geom_histogram(aes(y=after_stat('density')), bins = 40,color='black') + # Geometric
object to use for drawing
    geom_density(aes(y=after_stat('density')),color='red') +
    stat_function(fun=lambda x:  norm.pdf(x, loc = f.fitted_param['norm'][0], scale =
f.fitted_param['norm'][1]),color='blue')
)
```



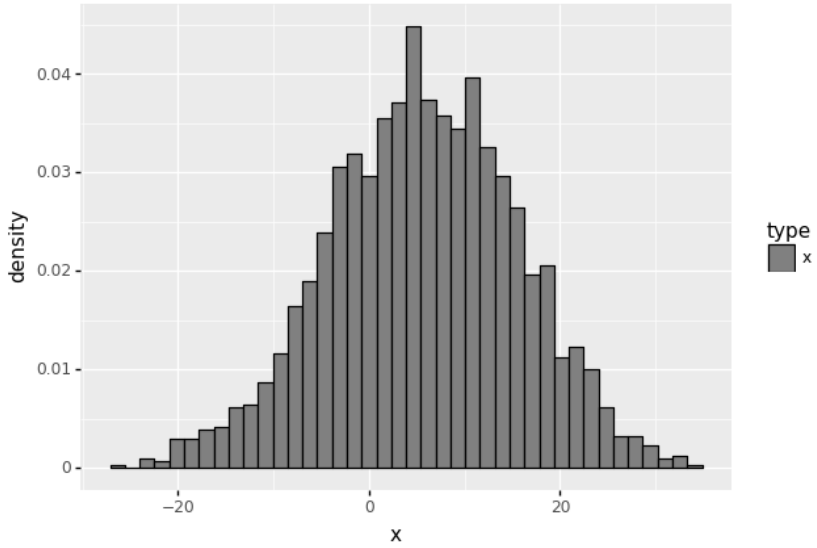

```
<ggplot: (313770689)>
```

```
# data2.drop('fitted',axis=1,inplace=True)
```

```
# Aim: add type so that legend will be created

data2 = data
data2['type'] = 'x'
data2['fitted'] = norm.pdf(data2['x'], loc = f.fitted_param['norm'][0], scale =
f.fitted_param['norm'][1]  )
```

```
#data2.drop('type',axis=1,inplace=True)
```

```
# (ggplot(data2) + aes(x = 'x', y = 'fitted',color='type') + geom_line() +
geom_histogram(aes(y=after_stat('density'))  ))

(
    ggplot(data2) +  # What data to use
    aes(x='x',color='type') +  # What variable to use
    geom_histogram(aes(y=after_stat('density')),fill='gray',bins=40  ) +
    scale_color_manual(values = ['black'])
)
```




```
<ggplot: (313762393)>
```

Note alternative to Fitter library, we can also use **Distfit** Python library to automatically fit distributions to data.

See the following links for more details:

https://erdogant.github.io/distfit/pages/html/index.html

https://towardsdatascience.com/find-the-best-matching-distribution-for-your-data-effortlessly-bcc091aa08ab

5.5.4. Distribution fitting with distfit

The distfit library uses the goodness of fit test with the Sum of Squared Errors (or estimates) (SSE) to determine the best probability distribution.

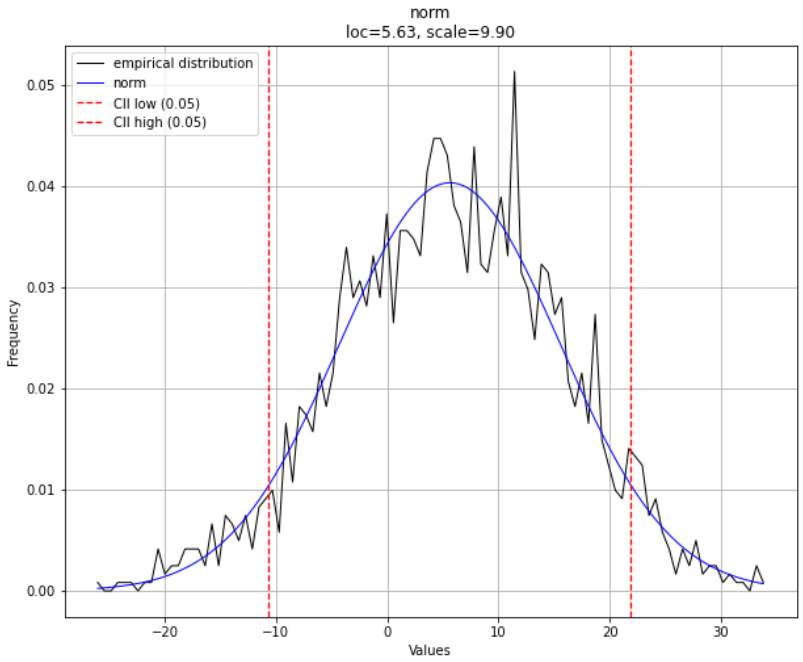
The Sum of Squared Errors (or estimates) (SSE), also named Residual Sum of Squares (RSS) works by comparing the observed frequency (f) to the expected frequency from the model (f-hat), and computing the residual sum of squares (RSS).

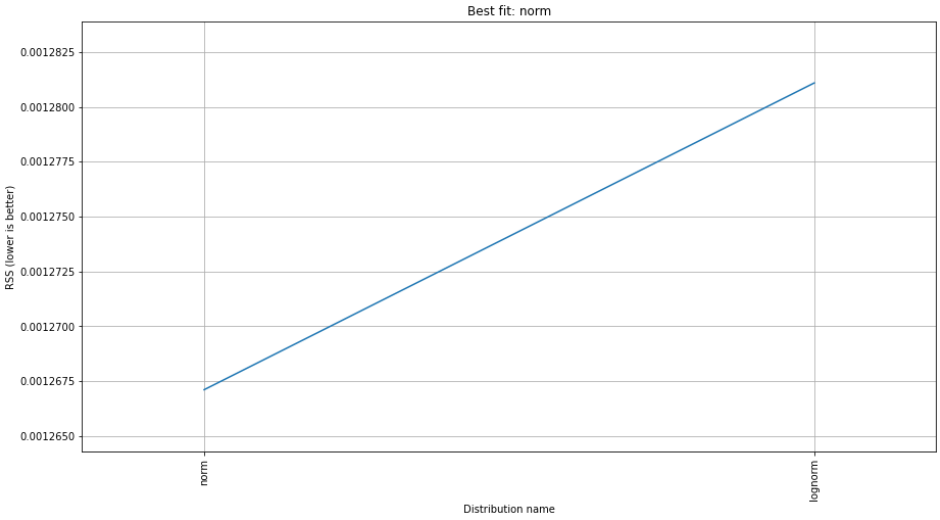
```
# Load library
from distfit import distfit
# Initialize model and test only for normal distribution
#dist = distfit(distr='norm')
# Set multiple distributions to test for
# see for bin size and smoothing
https://erdogant.github.io/distfit/pages/html/Performance.html#probability-density-function-fitting
dist = distfit(distr=['norm','lognorm'],bins=100)

# Search for best theoretical fit on your empirical data
results = dist.fit_transform(data.x)
```

```
[distfit] >fit..
[distfit] >transform..
[distfit] >[norm   ] [0.00 sec] [RSS: 0.0012671] [loc=5.631 scale=9.899]
[distfit] >[lognorm] [0.28 sec] [RSS: 0.0012811] [loc=-948.204 scale=953.808]
[distfit] >Compute confidence interval [parametric]
```

```
dist.plot()

# Make plot
dist.plot_summary()

print(dist.summary)
```

```
[distfit] >plot..
[distfit] >plot summary..
     distr     score  LLE         loc       scale                      arg
0     norm  0.001267  NaN    5.630744    9.899359                       ()
1  lognorm  0.001281  NaN -948.203969  953.808117  (0.010345149115340332,)
```





5.5.5. Distribution Fitting with Python SciPy

```
from scipy import stats
```

```
data

dist = getattr(stats, 'norm')
parameters = dist.fit(data['x'])
print(parameters)
```

```
(5.630744055249008, 9.899359274217787)
```

```
#dist = getattr(stats, 'lognorm')
#parameters = dist.fit(data['x'])
#print(parameters)
```

5.5.5.1. Calculating the sum of squared errors


```
data
```

	x	type	fitted
0	6.068843	x	0.040260
1	27.058152	x	0.003872
2	14.565627	x	0.026817
3	5.684111	x	0.040299
4	15.685138	x	0.024060
...	...	...	...
1995	10.669909	x	0.035403
1996	17.114355	x	0.020563
1997	8.697924	x	0.038411
1998	5.497944	x	0.040296
1999	14.580694	x	0.026780

2000 rows × 3 columns

```
# Get histogram of original data
#y, x = np.histogram(data, bins=100, density=True)
y, x = np.histogram(data.x, bins=100, density=True)
x_mid = (x + np.roll(x, -1))[:-1] / 2.0  # go from bin edges to bin middles
```

```
#(ggplot(pd.DataFrame(data={"x": x, "y":y}), aes(x="x",y="y"))
#    + geom_col())
```

```
parameters[1]
```

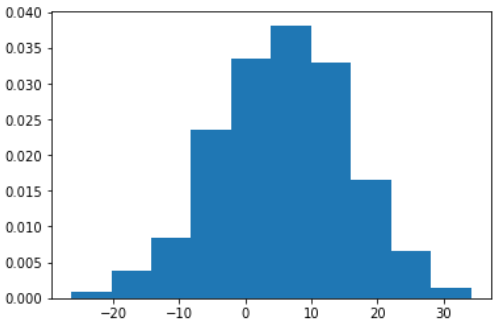
```
9.899359274217787
```

```
# Calculate fitted PDF and error with fit in distribution
pdf = norm.pdf(x_mid, loc=parameters[0], scale=parameters[1])

# To go from pdf back to counts need to un-normalise the pdf
# See Fitting All of Scipy's Distributions:
https://nedyoxall.github.io/fitting_all_of_scipys_distributions.html
# pdf_scaled = pdf * bin_width * N # to go from pdf back to counts need to un-normalise
the pdf

sse = np.sum(np.power(y - pdf, 2.0))

print(sse)
```

```
0.0012671148087710314
```

5.5.5.2. Get data points from a histogram in Python from matplotlib

```
#counts, bins, bars = plt.hist(data, density=True,bins=10)
counts, bins, bars = plt.hist(data.x, density=True,bins=10)
```




```
bins
```

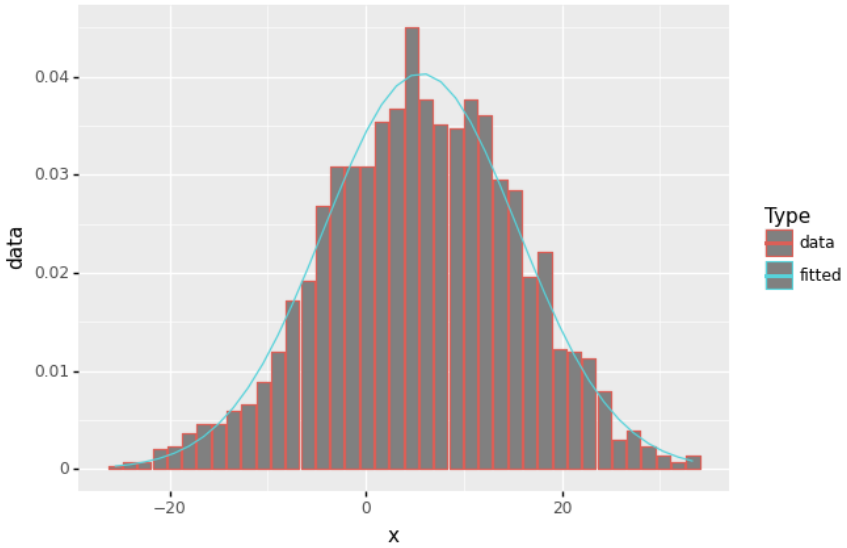
```
array([-26.34134829, -20.29646977, -14.25159126,  -8.20671274,
        -2.16183422,   3.88304429,   9.92792281,  15.97280133,
        22.01767984,  28.06255836,  34.10743688])
```

```
# Get histogram of original data
y, x = np.histogram(data.x, bins=40, density=True)
x_mid = (x + np.roll(x, -1))[:-1] / 2.0 # go from bin edges to bin middles
```

```
# Aim: plot histogram of original data (from y and x obtained abov)
# and plot the fitted values

df_freq = pd.DataFrame({'x':x_mid, 'data':y })

df_freq['fitted'] = norm.pdf(df_freq.x, loc = f.fitted_param['norm'][0], scale =
f.fitted_param['norm'][1])

df_freq.head()
```

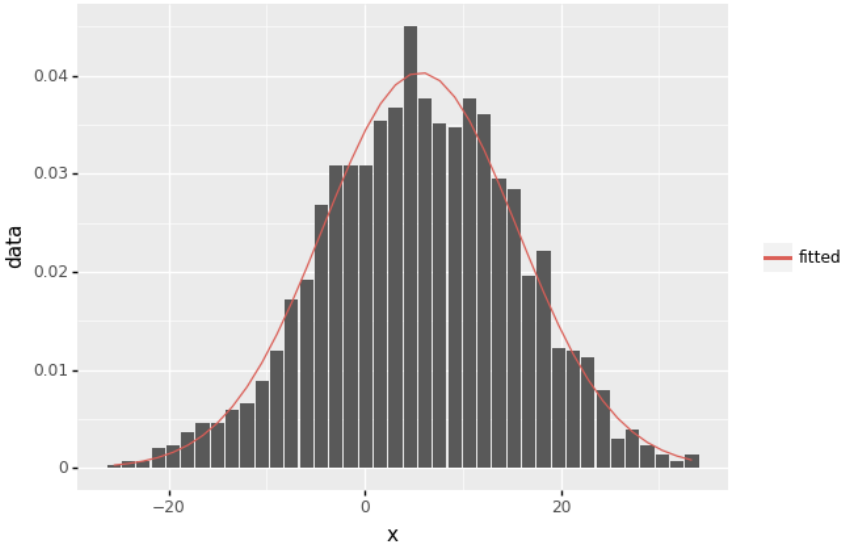
	x	data	fitted
0	-25.585738	0.000331	0.000279
1	-24.074519	0.000662	0.000447
2	-22.563299	0.000662	0.000698
3	-21.052080	0.001985	0.001066
4	-19.540860	0.002316	0.001590

```
(ggplot(df_freq, aes('x')) +
     geom_col(aes(y= 'data',color='"data"'),fill='grey') +
     geom_line(aes(y='fitted',color='"fitted"')) +
     scale_color_manual(values = ['black','blue']) +
     scale_color_discrete(name = "Type")
)
```




```
<ggplot: (313112281)>
```

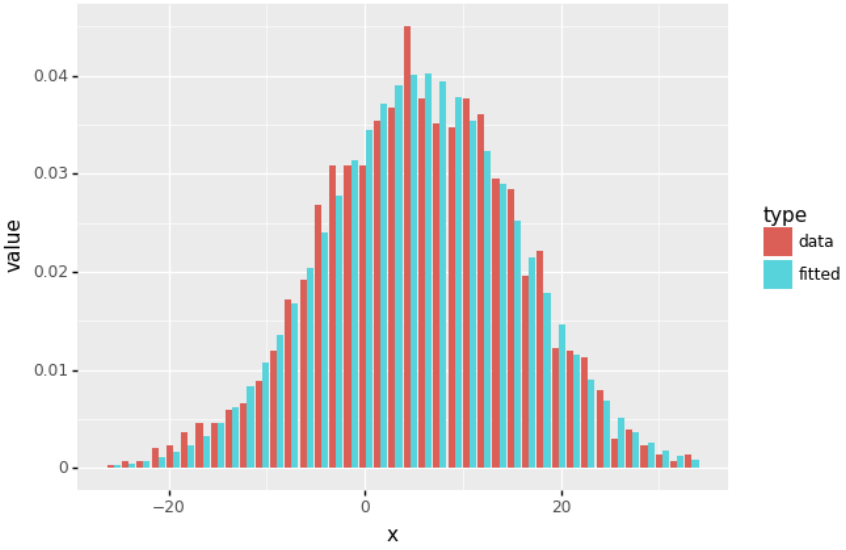
```
(ggplot(df_freq, aes('x')) +
     geom_col(aes(y='data')) +
     geom_line(aes(y='fitted',color='"fitted"')) +
     scale_color_discrete(name = " ")
)
```




```
<ggplot: (313105013)>
```

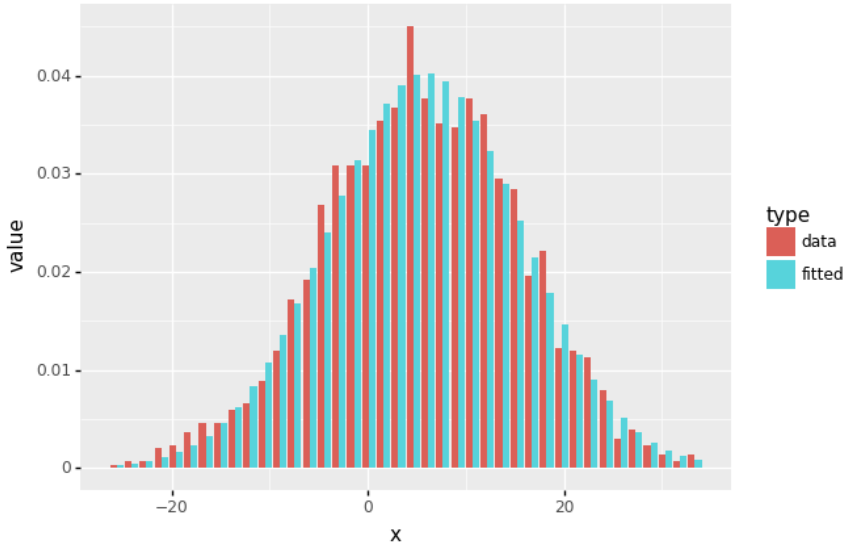
```
df_freq_melted = pd.melt(df_freq, id_vars = 'x', value_vars=['data','fitted'],
var_name='type')
```

```
# https://rpubs.com/Mentors_Ubiqum/geom_col_1

(
    ggplot(df_freq_melted) +
    geom_col(aes('x','value',fill='type'), position='dodge')
)
```



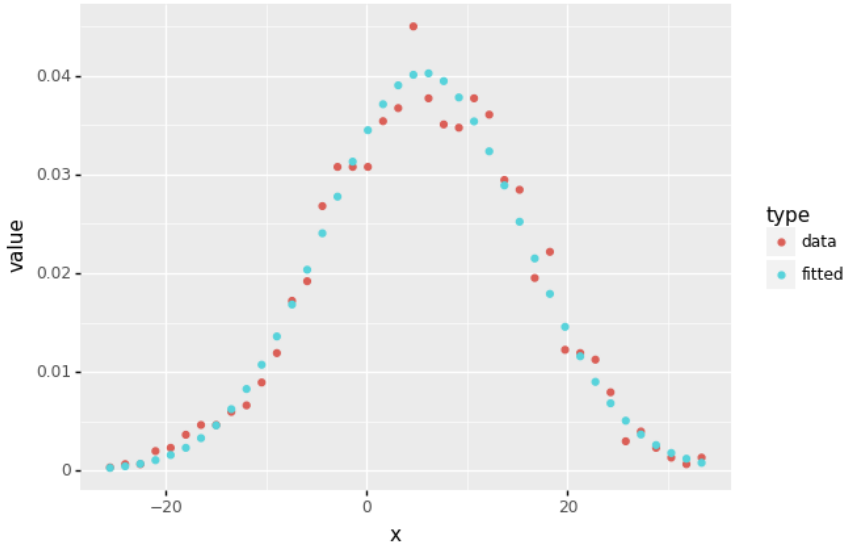

```
<ggplot: (314094577)>
```

```
(
    ggplot(df_freq_melted) +
    aes('x','value',fill='type') + geom_bar(stat='identity',position='dodge')
)
```




```
<ggplot: (313475537)>
```

```
df_freq_melted = pd.melt(df_freq, id_vars = 'x', value_vars=['data','fitted'],
var_name='type')


(ggplot(df_freq_melted) +
     aes('x','value',color='type') + geom_point()
)
```




```
<ggplot: (313466893)>
```

Previous
4. Data Visualization

Next
6. Regression Analysis

By Pairote Satiracoo
© Copyright 2021.

6 Regression Analysis

6. Regression Analysis

Regression analysis is a statistical technique that allows us to determine the relationship between two or more quantitative variables in order to predict a response or outcome variable from one or more of them. This methodology is frequently employed in a variety of fields, including business, social and behavioral sciences, biology, and many more.

Here are a few examples of applications:

1. In business, the relationship between product sales and advertising expenditures can be used to forecast product sales.
2. A financial firm may wish to know the top five causes that drive a customer to default in order to reduce risk in their portfolio.
3. A automobile insurance firm might use expected claims to Insured Declared Value ratio to create a suggested premium table using linear regression. The risk might be calculated based on the car's characteristics, the driver's information, or demographics.

The goal of regression is to create a model that expresses the relationship between a response $Y_i \in R^n$ and a set of one or more (independent) variables $X_i \in R^n$. We can use the model to predict the response Y_i based on the factors (or variables).

The following elements are included in regression models:

- The unknown parameters, which are frequently denoted as a scalar or vector β.
- The independent variables, which are observed in data and are frequently expressed as a vector X_i (where i denotes a data row).
- The dependent variable, which can be seen in data and is frequently represented by the scalar Y_i.
- The error terms, which are not readily visible in data and are frequently represented by the scalar e_i.

Most regression models propose that Y_i is a function of X_i and β,

$$Y_i = f(X_i, \beta) + e_i.$$

In other words, regression analysis aims to find the function $f(X_i, \beta)$ that best matches the data. The function's form must be supplied in order to do regression analysis. The form of this function is sometimes based on knowledge of the relationship between X_i and Y_i. If such information is not available, a flexible or convenient form is adopted.

6.1. Simple linear regression

The simplest model to consider is a linear model, in which the response Y_i is linearly proportional to the variables X_i:

$$Y_i = \beta_0 + \beta_1 X_i + e_i$$

6.2. Insurance Premium Prediction: Example

For analysis, we used the Kaggle public dataset "Insurance Premium Prediction.". There are 1338 samples and 7 features.

We will learn how to predict insurance costs based on the features age, sex, bmi, children, smoking status, and region.

Let us start by loading the data and performing a preliminary analysis with df.info ()

Contents

```
import pandas as pd
import numpy as np

import statsmodels.api as sm

from scipy.stats import *

import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import math as m
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error,mean_absolute_error,r2_score
from scipy.stats import pearsonr,spearmanr
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/var/folders/kl/h_r05n_j76n32kt0dwy7kynw0000gn/T/ipykernel_5503/1614355564.py in <module>
      2 import numpy as np
      3
----> 4 import statsmodels.api as sm
      5
      6 from scipy.stats import *

ModuleNotFoundError: No module named 'statsmodels'
```

```
df = pd.read_csv('/Users/Kaemyuijang/SCMA248/Data/insuranceKaggle.csv')
```

Exploration of the dataset is always a good practice.

We will use charges as the dependent variable and the other variables as independent variables, which means we will have to predict charges based on the independent variables.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1338 entries, 0 to 1337
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       1338 non-null   int64
 1   sex       1338 non-null   object
 2   bmi       1338 non-null   float64
 3   children  1338 non-null   int64
 4   smoker    1338 non-null   object
 5   region    1338 non-null   object
 6   charges   1338 non-null   float64
dtypes: float64(2), int64(2), object(3)
memory usage: 73.3+ KB
```

In out data set, types of variables are as follows:

- categorical variables: sex, smoker, region
- numerical variables: age, bmi, children, expenses

There is no missing data among 1338 records.

6.2.1. Exploratory data analysis

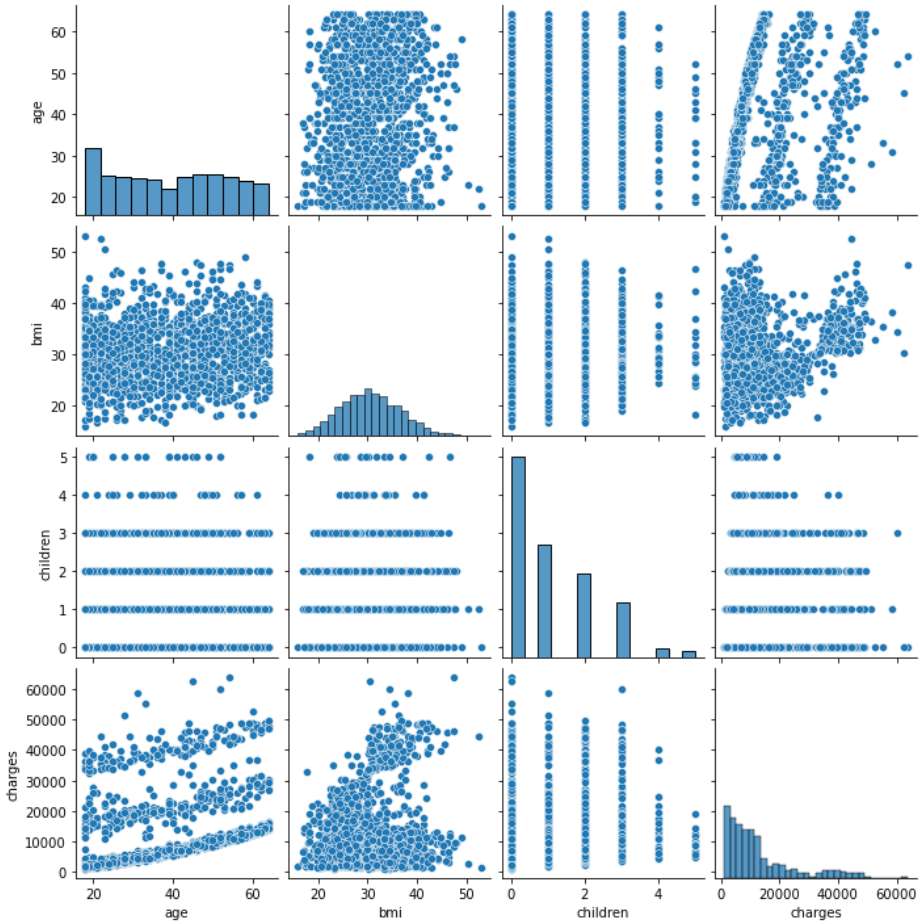
It is now time to experiment with the data and make some visualizations.

In our dataset, a **pairplot** plots pairwise relationships. The pairplot function creates a grid of Axes in which each variable in the data is shared across a single row and a single column on the y-axis.

A pairs plot shows the distribution of single variables as well as the relationships between them. Pair plots are a great way to detect trends for further study, and they're simple to create in Python!


```
sns.pairplot(df)
```

```
<seaborn.axisgrid.PairGrid at 0x1274db110>
```



Note: We can see that age is correlated with the response variable charges.

6.2.2. Make a research question

For our first regression analysis, we are interested in the following question.

> Is there an association between charges and age?

The normal error regression model will be used to determine the linear relationship: $
$Y_i = \beta_0 + \beta_1 X_i + e_i$$

We need to define a null hypothesis and an alternative hypothesis before we can run the numbers:

H0 (null hypothesis): There is no linear relationship between charge and age;

H1 (Alternative hypothesis): There is a linear relationship between charge and age.

If this hypothesis is accepted, one should discard the X information and base inferences on the Y values alone (so $Y_i = \bar{Y}$).

6.2.3. Running and reading a simple linear regression

statsmodels is a Python package that may be used to fit a variety of statistical models, run statistical tests, and explore and visualize data. Other libraries contain Bayesian methods and machine learning models, whereas Statsmodels contains more "traditional" frequentist statistical approaches.

In statsmodels, there are various types of linear regression models, ranging from the most basic (e.g., ordinary least squares) to the most complicated (e.g., logistic regression) (e.g., iteratively reweighted least squares).

In what follows, we will use Python's statsmodels module to implement **Ordinary Least Squares(OLS) method** of linear regression.

6.2.4. Method of least squares

The general model takes the following form:

$$Y_i = f(X_i, \beta) + e_i.$$

Given a data set consists of n points (data pairs) $(x_i, y_i), i = 1, \ldots, n$, where x_i is an independent variable and y_i is a dependent variable, we define the **residual** as the difference between the observed value of the dependent variable and the value predicted by the model. It is used to assess a model's fit to a data point:

$$r_i = y_i - f(x_i, \beta).$$

The least-squares method finds the optimal parameter values β by minimizing the sum of squared residuals, S

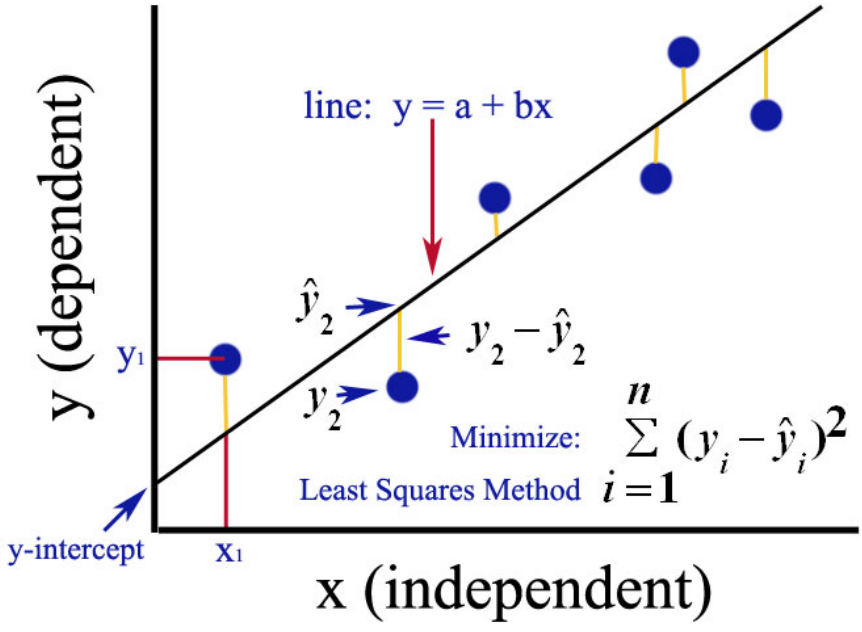
$$S = \sum_{i=1}^{n} r_i^2.$$

6.2.5. Linear least squares

For the simplest linear model $$Y_i = \beta_0 + \beta_1 X_i + e_i, \\ the method of least squares reduces to find the vaues of \beta _{0} and \beta _{1} that minimise \\ S = \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2.$$

Follow the link for more details: https://en.wikipedia.org/wiki/Linear_least_squares#Example

Here is the diagram to illustrate the idea of method of least squares.



The main interfaces for linear models in statsmodels are array-based and formula-based. These can be accessible via the following API module imports:


```
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

```
model1 = smf.ols(formula='charges ~ age', data=df).fit()
```

- smf calls the package Statsmodel
- ols command indicates that we're employing an Ordinary Least Square (OLS) regression (a type linear regression)
- formula= is a formula that is used to write the dependent as well as all of the independent variables (s)
- dependent variable/outcome: first variable within the parenthesis before "~": Our only dependent variable is the first one. This is our result, the variable that defines the type of regression to use and the one that will be linked to all other covariates.
- ~ inside parenthesis: Creates a division between the outcome (dependent variable) on the left and the covariates (independent variables) on the right.
- independent covariates/independent variables: All other variables after the "," inside parenthesis, are independent covariates/independent variables.
- The + symbol is used to differentiate various independent variables within the same model (helpful for multivariable models, aka many variables).
- ,data= is used to specify the data frame's name.
- fit() tell Python that we want to fit our function ("run the function").

6.2.6. Interpreting the results

To view the results of the model, you can use `summary()` function:

```
model1.summary()
```

Dep. Variable:	charges	**R-squared:**	0.089
Model:	OLS	**Adj. R-squared:**	0.089
Method:	Least Squares	**F-statistic:**	131.2
Date:	Sat, 09 Jul 2022	**Prob (F-statistic):**	4.89e-29
Time:	17:35:38	**Log-Likelihood:**	-14415.
No. Observations:	1338	**AIC:**	2.883e+04
Df Residuals:	1336	**BIC:**	2.884e+04
Df Model:	1		
Covariance Type:	nonrobust		

OLS Regression Results

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	3165.8850	937.149	3.378	0.001	1327.440	5004.330
age	257.7226	22.502	11.453	0.000	213.579	301.866

Omnibus:	399.600	**Durbin-Watson:**	2.033
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	864.239
Skew:	1.733	**Prob(JB):**	2.15e-188
Kurtosis:	4.869	**Cond. No.**	124.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

We can see some important general data about the model on the upper half of the output.

On the left, we can observe

- The dependent variable is charges.
- We are running an ordinary least square (OLS) regression model.

- We included 1338 observations (patients).

We can notice some key information about the model diagnosis in the right column:

- **Determination coefficient** (R-squared) tells us the fraction (%) of variation in the response variable Y that can be explained by the predictor variable X. Its value goes from 0 (no predictability) to 1 (100 percent), indicating total predictability. A high R-squared means that the response variable may be predicted with less error.

See the following link for more details: https://vitalflux.com/linear-regression-explained-python-sklearn-examples/

In our example, R-squared = 0.089, indicating that the model explains 9 percent of the outcome variability.

Note also that the low R-squared graph shows that even noisy, high-variability data may have a significant trend. The trend indicates that the predictor variable still provides information about the response even though data points fall further from the regression line.

- **The F-statistic value** is the outcome of a test in which the null hypothesis is that all regression coefficients are equal to zero. To put it another way, the model has no predictive capability.

Essentially, the F-test compares your model against a model with no predictor variables (the intercept alone model) to see if adding coefficients improved the model. If you get a significant result, it means that the coefficients you added to your model improved its fit.

For example, consider a multiple linear regression model:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_4 + e_i,$$

The F-statistic allows us to examine whether ANY of the independent variables X_1, X_2, X_3, X_4, etc. are connected to the result Y.

For a 0.05 significance level:

- If the p-value associated with the F-statistic is less than 0.05, there is no relationship between any of the independent variables and Y.
- If the p-value associated with the F-statistic is less than 0.05, AT LEAST 1 independent variable is related to Y.

In this example, the Prob (F-statistic), which here is <0.05, indicates that our linear regression model provides a better fit to the data than a model that contains no independent variables.

Regarding the output on the bottom half, we will read the output one column at a time. We are looking at the association between our charges and our independent variable, age (first column),

- **Age coefficient** represents the change in the output Y (charges) due to a change of one unit in age. The age coefficient in this case is 257.7226.
- **std err** reflects the level of accuracy of the coefficients. The lower the standard error is, the higher is the level of accuracy. The results shows the standard error of the coefficient of 22.502.
- P >|t| is our p-value with a t-ratio (for a t-distribution) of 11.453 (fourth column). A **p-value** of less than 0.05 is considered to be statistically significant.
- **Confidence Interval** represents the range in which the (age) coefficient is likely to fall, i.e. corresponding to a 95% confidence interval for the age coefficient between 213.579 and 301.866.

For the Intercept, this is the y-intercept of the regression equation, with a value of 3165.8850. You can plug this into your regression equation if you want to predict charges across the range of age that you have observed:

Note: we will not partition our data into training and testing because we are conducting an observational study to observe/test an association rather than constructing a predictive model. As a result, do not be surprised if this method differs from what you'll find in most data science classes.

6.2.7. Presenting the results

From our analysis, we found that the regression coefficients are $ $\beta_0 = 3165.885006$ and $\beta_1 = 257.722619$.$

The estimated regression function is $$\hat{Y} = 3165.885 + 257.7226X.$$

When reporting your results, include the estimated effect (i.e. the regression coefficient), standard error of the estimate, and the p-value, we should additionally explain what your regression coefficient means to your viewers by interpreting your your regression coefficient means:

Age and charges have a significant relationship (p < 0.001), with a 257.7226-unit (95% CI of β_1: 257.7226 $\pm$ 22.502) increase in charges for each increase in age.

It is also a good idea to provide a graph with your findings. You can plot the observations on the x and y axes and then add the regression line and regression function for a simple linear regression (show below):

```
# Model parameters

print('Estimates of model parameters:\n',model1.params, '\n')

print('t values:\n', model1.tvalues)
```

```
Estimates of model parameters:
 Intercept    3165.885006
age           257.722619
dtype: float64

t values:
 Intercept     3.378207
age          11.453122
dtype: float64
```

6.2.8. Predicted values and residuals

We can compute predicted values given the estimated model parameters using `predict` function.

In what follows, we calculate the predicted values and use them to calculate the residuals.

Recall that the ith residual is the difference between the observed value Y_i and the corresponding fitted value $\hat{Y}_i$.

```
x=df[['age']]
y=df[['charges']]

predictions = model1.predict(x)
df['slr_result'] = predictions

df['slr_error'] =  df['charges'] - df['slr_result']
```

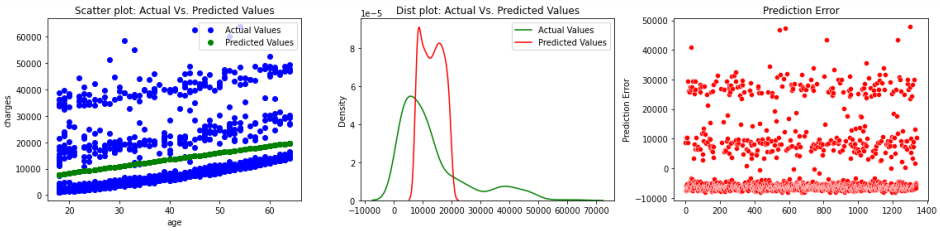
```
df[['age','charges','slr_result','slr_error']]
```

	age	charges	slr_result	slr_error
0	19	16884.92400	8062.614761	8822.309239
1	18	1725.55230	7804.892142	-6079.339842
2	28	4449.46200	10382.118329	-5932.656329
3	33	21984.47061	11670.731422	10313.739188
4	32	3866.85520	11413.008803	-7546.153603
...	...	...	...	...
1333	50	10600.54830	16052.015939	-5451.467639
1334	18	2205.98080	7804.892142	-5598.911342
1335	18	1629.83350	7804.892142	-6175.058642
1336	21	2007.94500	8578.059998	-6570.114998
1337	61	29141.36030	18886.964745	10254.395555

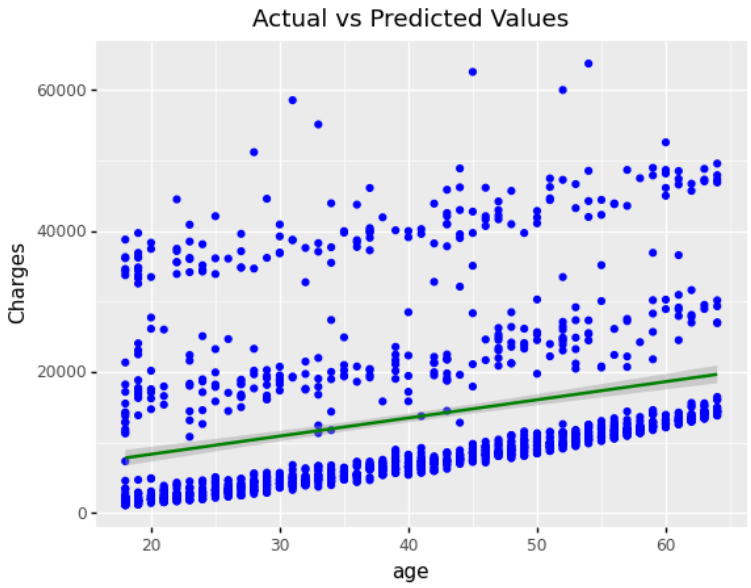
1338 rows × 4 columns

```
fig, axes =plt.subplots(1,3, figsize=(16,4))
axes[0].plot(x['age'], y,'bo',label='Actual Values')
axes[0].plot(x['age'], predictions,'go',label='Predicted Values')
axes[0].set_title("Scatter plot: Actual Vs. Predicted Values")
axes[0].set_xlabel("age")
axes[0].set_ylabel("charges")
axes[0].legend()

sns.distplot(y, hist=False, color="g", label="Actual Values",ax=axes[1])
sns.distplot(predictions, hist=False, color="r", label="Predicted Values" , ax=axes[1])
axes[1].set_title("Dist plot: Actual Vs. Predicted Values")
axes[1].legend()

sns.scatterplot(x=y.index,y='slr_error',data=df,color="r", ax=axes[2])
axes[2].set_title("Prediction Error")
axes[2].set_ylabel("Prediction Error")

fig.tight_layout()
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated
function and will be removed in a future version. Please adapt your code to use either
`displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level
function for kernel density plots).
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated
function and will be removed in a future version. Please adapt your code to use either
`displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level
function for kernel density plots).
```

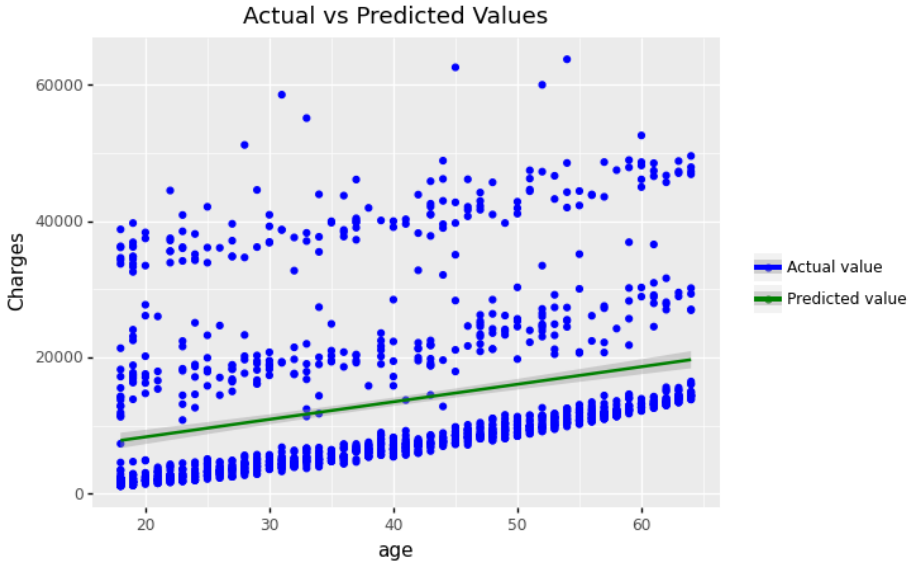



```
import pandas as pd
import numpy as np

from plotnine import *
from plotnine.data import *

%matplotlib inline

(
    ggplot(df, aes(x='age', y='charges'))
    + geom_point(color='blue')
    + geom_smooth(method='lm',color='green')
    + labs(y='Charges', title = 'Actual vs Predicted Values')
    + scale_fill_manual(
    values = ['Actual values','Predicted values'],
    name = " ", labels = ['Actualvalues','Predicted values'] )
)
```



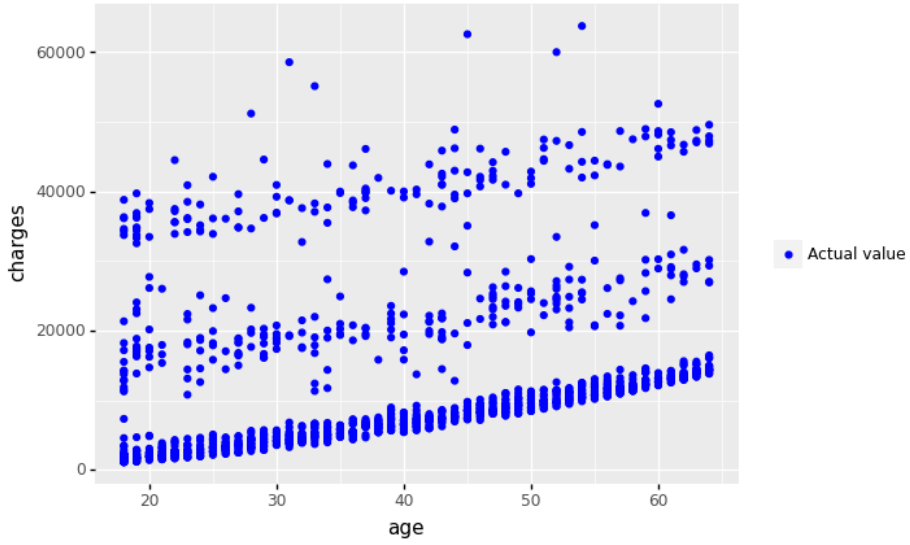

```
<ggplot: (300420313)>
```

```
(
    ggplot(df,aes(x='age',y='charges', color='"Actual value"'))
    + geom_point()
    + geom_smooth(mapping = aes(color='"Predicted value"'), method='lm')
    + labs(y='Charges', title = 'Actual vs Predicted Values')
    #+ scale_fill_manual(
    #values = ['Actual values','Predicted values'],
    #name = " ", labels = ['Actual values','Predicted values'] )
    + scale_color_manual(values = ["blue", "green"], # Colors
        name = " ") # Remove the legend title

)
```



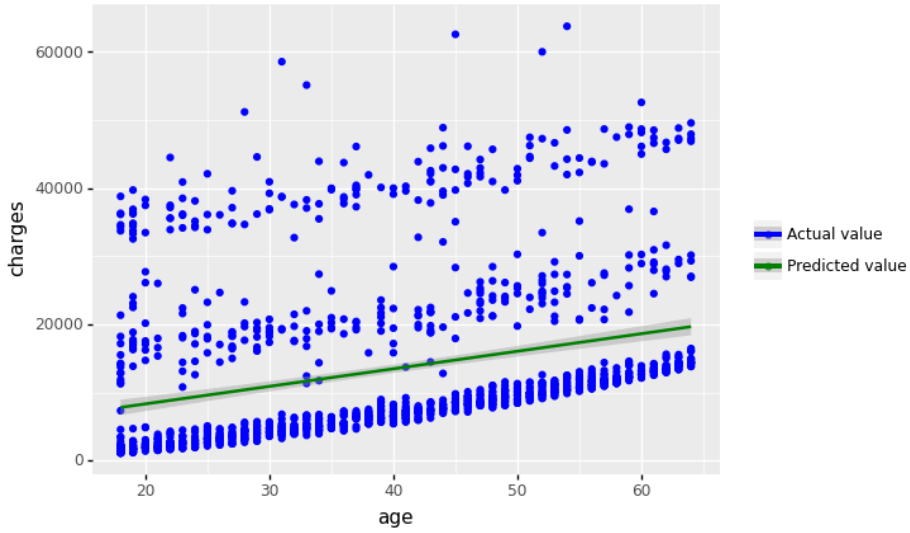

```
<ggplot: (305221401)>
```

```
(
    ggplot(df)
    + geom_point(aes(x = 'age', y='charges',color='"Actual value"'))
    + scale_color_manual(values = ["blue"], # Colors
        name = " ")
)
```



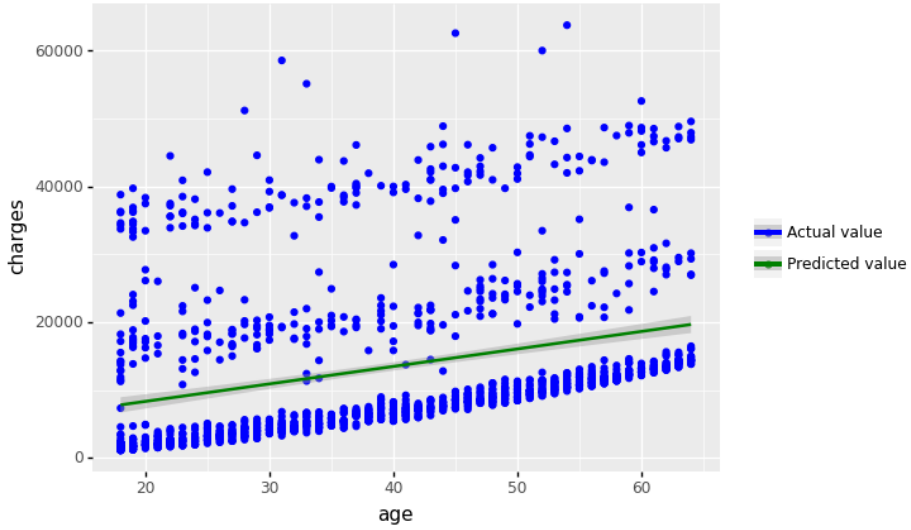

```
<ggplot: (300775873)>
```

```
(
    ggplot(df)
    + geom_point(aes(x = 'age', y='charges',color='"Actual value"'))
    + geom_smooth(aes(x='age',y='charges',color='"Predicted value"'), method='lm')
    + scale_color_manual(values = ['blue','green'], # Colors
        name = " ")
)
```



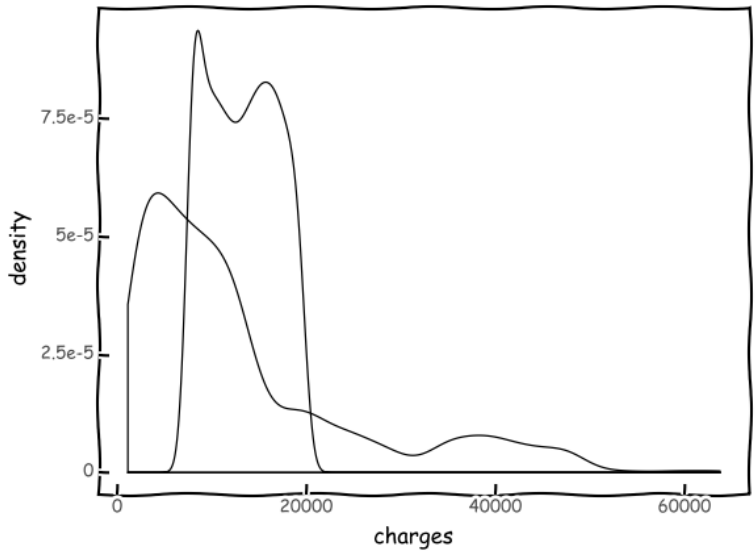

```
<ggplot: (300807313)>
```

```
(
    ggplot(df) + aes(x = 'age', y='charges')
    + geom_point(aes(color='"Actual value"'))
    + geom_smooth(aes(color='"Predicted value"'), method='lm')
    + scale_color_manual(values = ['blue','green'], # Colors
        name = " ")
)
```



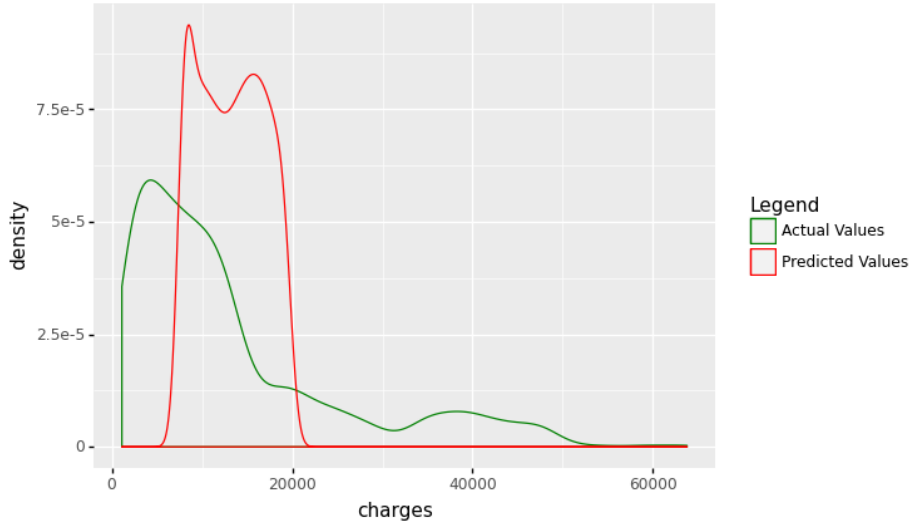

```
<ggplot: (304867401)>
```

```
(
    ggplot(df, aes(x='charges'))
    +  geom_density(aes(y=after_stat('density')))
    +  geom_density(aes(x='slr_result',y=after_stat('density')))
    + theme_xkcd()
)
```



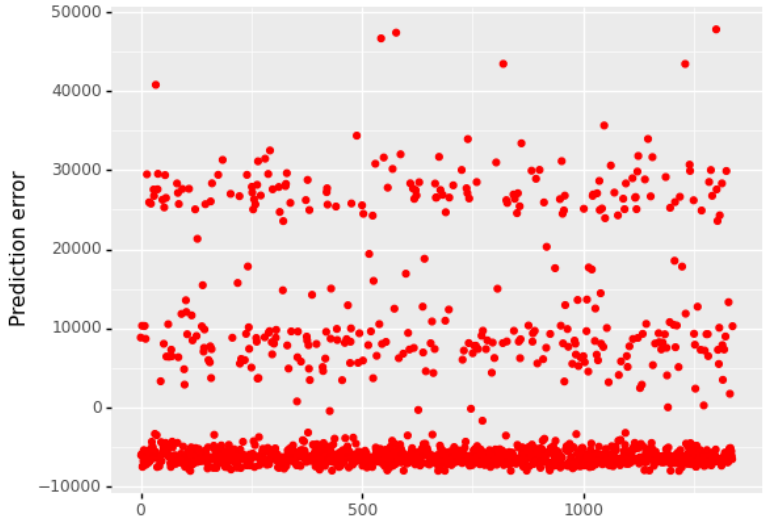

```
<ggplot: (306741109)>
```

```
(
    ggplot(df, aes(x='charges'))
    +  geom_density(aes(y=after_stat('density'),color='"Actual Values"' ))
    +  geom_density(aes(x='slr_result',y=after_stat('density'),color='"Predicted
Values"'))
    +  scale_color_manual(values = ['green','red'], name = 'Legend')
)
```




```
<ggplot: (306278033)>
```

```
(
    ggplot(df, aes(x='df.index'))
    +  geom_point(aes(y='slr_error'),color='red')
    + labs(x = ' ', y='Prediction error')
)
```




```
<ggplot: (305154745)>
```

Exercise In this exercise, we will use the gapminder dataset. Our aim is to determine a relationship between life expectancy and year?

1. Plot life expectancy over time in a scatter plot.
2. Is there a general trend in life expectancy over time (e.g., increasing or decreasing)? Is this a linear trend?
3. How would you characterize the distribution of life expectancy for individual years among countries? Is the data skewed or not? Is it better to be unimodal or multimodal? Is it symmetrical around the center?
4. Fit a linear regression model using statsmodels.
5. Will you reject the null hypothesis of no association if I run a linear regression model of life expectancy vs. year (considering it as a continuous variable) and test for a relationship between year and life expectancy?

```
gapminder = pd.read_csv('/Users/Kaemyuijang/SCMA248/Data/gapminder_full.csv')
```

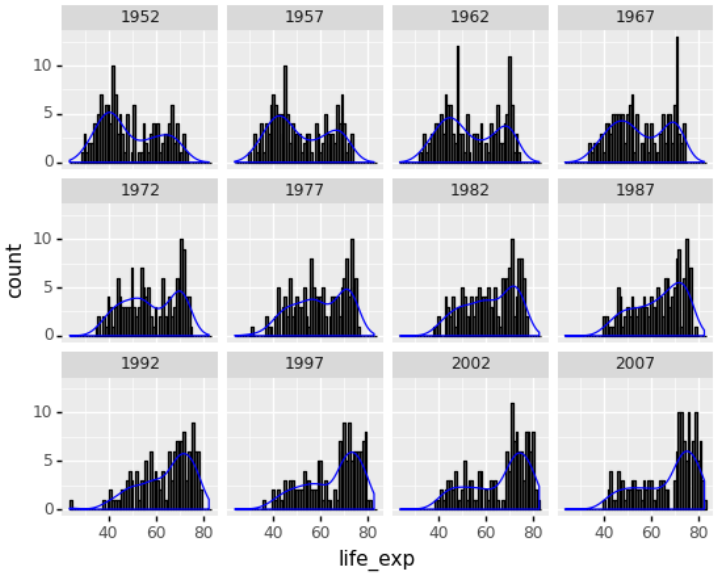
```
gapminder.head()
```

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106

```
(
    ggplot(gapminder) +
    aes(x= 'year',y = 'life_exp') +
    geom_point()  +
    labs(x = 'year', y='Life expectancy', title = 'Life expectancy over time')
)
```


```
<ggplot: (304622001)>
```

```
(
    ggplot(gapminder) +
    aes('life_exp') +
    geom_histogram(aes(y=after_stat('count')), binwidth = 1, color = 'black') +
    geom_density(aes(y=after_stat('count')),color='blue') +
    facet_wrap('year')
)
```



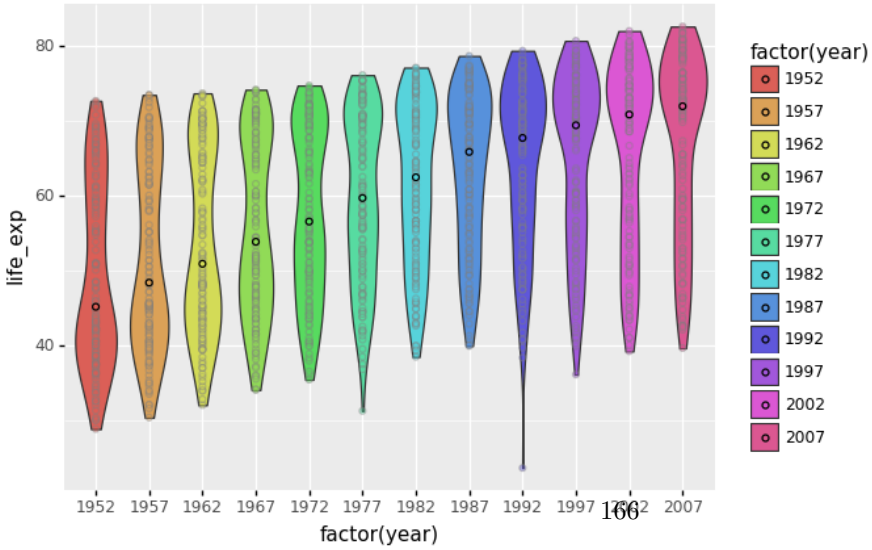

```
<ggplot: (309754613)>
```

6.2.9. Violin plots

A **violin plot** displays the densities of different groups, allowing you to compare their distributions.

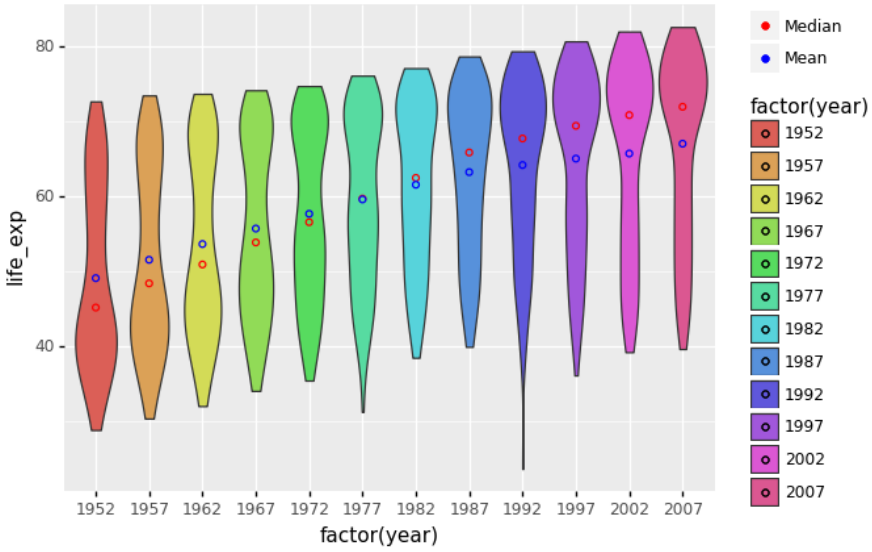
Violin plots are similar to box plots in that they display the probability density of the data at various values, which is normally smoothed using a kernel density estimator. Typically, a violin plot will include all of the data found in a box plot: a marker for the data's median; a box or marker representing the interquartile range; and, assuming the number of samples is not too large, all sample points.

```
# useful link: https://r-charts.com/distribution/violin-plot-mean-ggplot2/

(
    ggplot(gapminder) +
    aes(x= 'factor(year)',y = 'life_exp', fill = 'factor(year)') +
    geom_violin() +
    geom_point(color='gray',alpha=0.4) +
    stat_summary(fun_y=np.median,
                 geom = 'point',
                 color = 'black')
)
```



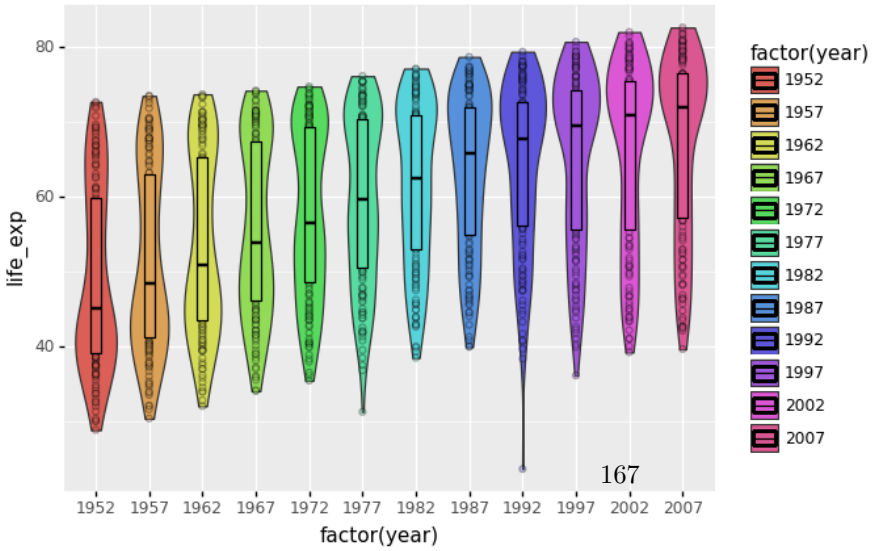

```
<ggplot: (311171093)>
```

```
(
    ggplot(gapminder) +
    aes(x= 'factor(year)',y = 'life_exp', fill = 'factor(year)') +
    geom_violin() +
    stat_summary(mapping=aes(color='"Median"'),
                 fun_y=np.median,
                 geom = 'point') +
    stat_summary(mapping=aes(color='"Mean"'),
                 fun_y=np.mean,
                 geom = 'point') +
    scale_color_manual(values = ["red", "blue"], # Colors
                name = " ") # Remove the legend title
)
```




```
<ggplot: (300058737)>
```

```
(
    ggplot(gapminder) +
    aes(x= 'factor(year)',y = 'life_exp', fill = 'factor(year)') +
    geom_violin() +
    geom_point(alpha=0.3) +
    stat_summary(fun_y=np.median,
                 fun_ymin=lambda x:  np.percentile(x,25),
                 fun_ymax=lambda x:  np.percentile(x,75),
                geom = 'crossbar',  # try pointrange
                color = 'black', width = 0.2)
)
```




```
<ggplot: (306789701)>
```

```
model2 = smf.ols(formula='life_exp ~ year', data=gapminder).fit()
```

```
model2.summary()
```

Dep. Variable:	life_exp	**R-squared:**	0.190
Model:	OLS	**Adj. R-squared:**	0.189
Method:	Least Squares	**F-statistic:**	398.6
Date:	Sat, 09 Jul 2022	**Prob (F-statistic):**	7.55e-80
Time:	17:36:01	**Log-Likelihood:**	-6597.9
No. Observations:	1704	**AIC:**	1.320e+04
Df Residuals:	1702	**BIC:**	1.321e+04
Df Model:	1		
Covariance Type:	nonrobust		

OLS Regression Results

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-585.6522	32.314	-18.124	0.000	-649.031	-522.273
year	0.3259	0.016	19.965	0.000	0.294	0.358

Omnibus:	386.124	**Durbin-Watson:**	0.197
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	90.750
Skew:	-0.268	**Prob(JB):**	1.97e-20
Kurtosis:	2.004	**Cond. No.**	2.27e+05

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 2.27e+05. This might indicate that there are
strong multicollinearity or other numerical problems.

Exercise

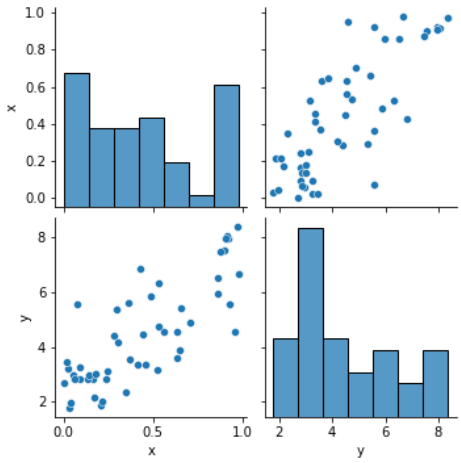
1. Write Python code to create a random dataset and perform a linear regression analysis on this dataset with `sd_error = 1`, the standard deviation of the error term.
2. Repeat the problem with `sd_error = 1`.
3. Compare the results between these two samples.

```
np.random.seed(888)

# number of sample
nsample = 50

# model parameter values
a = 2
b = 5

sd_error = 1

xsample = np.random.rand(nsample)
ysample = a + b*xsample + norm.rvs(size=nsample, loc = 0, scale=sd_error, )

random_data = pd.DataFrame({'x':xsample,'y':ysample})

random_data.head()
```

	x	y
0	0.859561	5.959411
1	0.164569	2.820203
2	0.483476	5.866683
3	0.921027	7.971193
4	0.428556	6.834657

```
sns.pairplot(random_data)
```

```
<seaborn.axisgrid.PairGrid at 0x122e9cfd0>
```




```
results = smf.ols('ysample ~ xsample', data = random_data).fit()
```

```
results.summary()
```

Dep. Variable:	ysample	**R-squared:**	0.629
Model:	OLS	**Adj. R-squared:**	0.621
Method:	Least Squares	**F-statistic:**	81.41
Date:	Sat, 09 Jul 2022	**Prob (F-statistic):**	6.57e-12
Time:	17:36:03	**Log-Likelihood:**	-76.771
No. Observations:	50	**AIC:**	157.5
Df Residuals:	48	**BIC:**	161.4
Df Model:	1		
Covariance Type:	nonrobust		

OLS Regression Results

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	2.3954	0.277	8.635	0.000	1.838	2.953
xsample	4.6387	0.514	9.023	0.000	3.605	5.672

Omnibus:	1.034	**Durbin-Watson:**	1.780
Prob(Omnibus):	0.596	**Jarque-Bera (JB):**	1.081
Skew:	0.304	**Prob(JB):**	0.582
Kurtosis:	2.613	**Cond. No.**	3.83

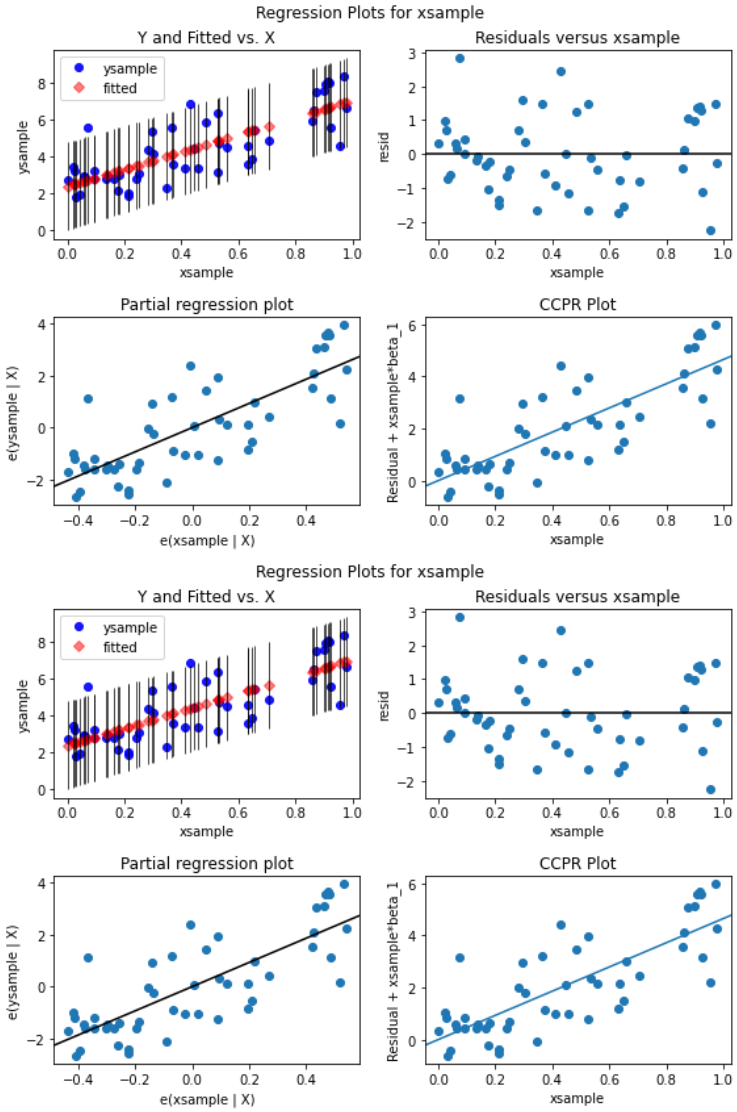
Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

A 2 by 2 figure based on `xsample` showing fitted versus actual `ysample`, residuals versus the `xsample`, partial regression plot of `xsample`, and CCPR Alot for `xsample`.

```
fig = plt.figure(figsize=(8, 6))
sm.graphics.plot_regress_exog(results, 'xsample', fig=fig)
```

```
eval_env: 1
```



6.3. Multivariate Analysis

Our next analysis is to estimate the association between two or more independent variables and one dependent variable. Multiple linear regression will be used to answer the following:

1. The degree to which two or more independent variables and one dependent variable are related (e.g. how strong the relationship is between independent variables (age, bmi, and smoker) and dependent variable (charges)).
2. The value of the dependent variable at a given value of the independent variables (e.g. age, bmi, and smoking status addition)

```
df = pd.read_csv('/Users/Kaemyuijang/SCMA248/Data/insuranceKaggle.csv')
```



```
df.head()
```

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

6.3.1. Exploratory Data Analysis (EDA)

We will perform exploratory data analysis to analyze data quality as well as uncover hidden relationships between variables. In this exercise, I'll go through three strategies relevant to regression analysis:

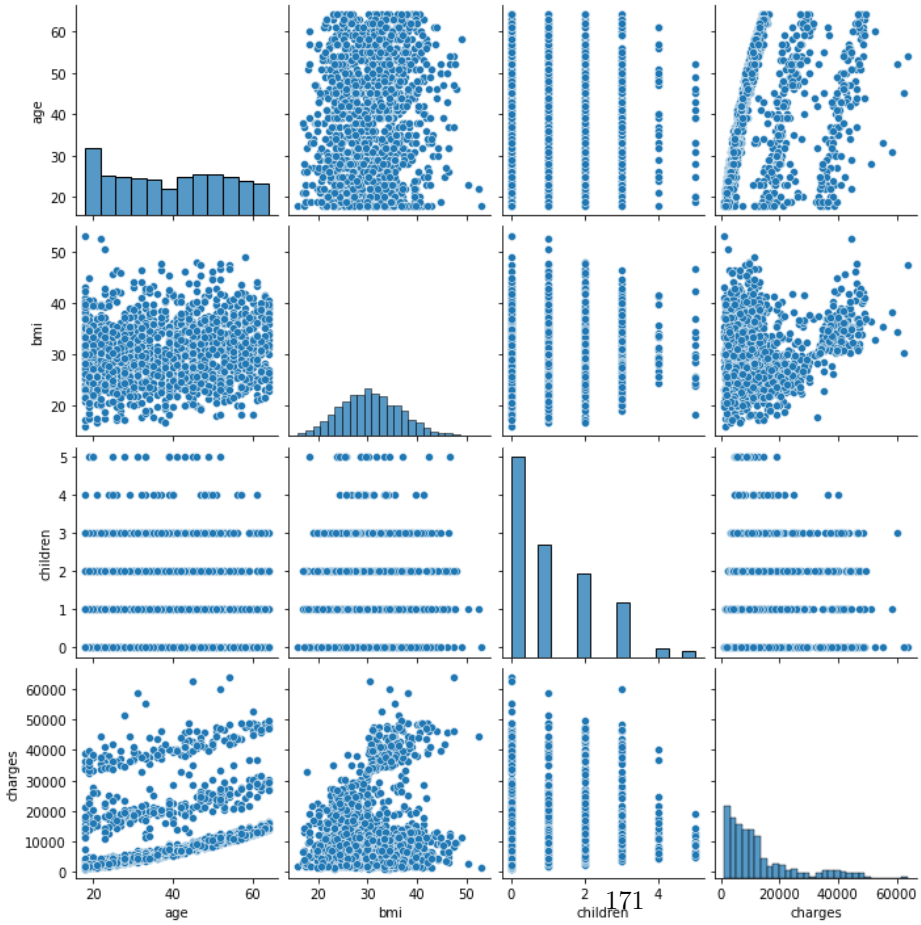
1. Multivariate analysis
2. Preprocessing categorical variables
3. Correlation Analysis

6.3.1.1. Multivariate analysis

We have already created a pairs plot to shows the distribution of single variables as well as the relationships between them.

```
sns.pairplot(df)
```

```
<seaborn.axisgrid.PairGrid at 0x124a95290>
```



Note Our dataset contains both continuous and categorical variables. However, the pairs plot shows only the relationship of the continuous variables.

In Seaborn, pairs plots (scatter plots or box plots) can take an additional hue argument to add another variable for comparison (e.g., for comparison between different categories).

6.3.1.2. Pairs Plot with Color by Category

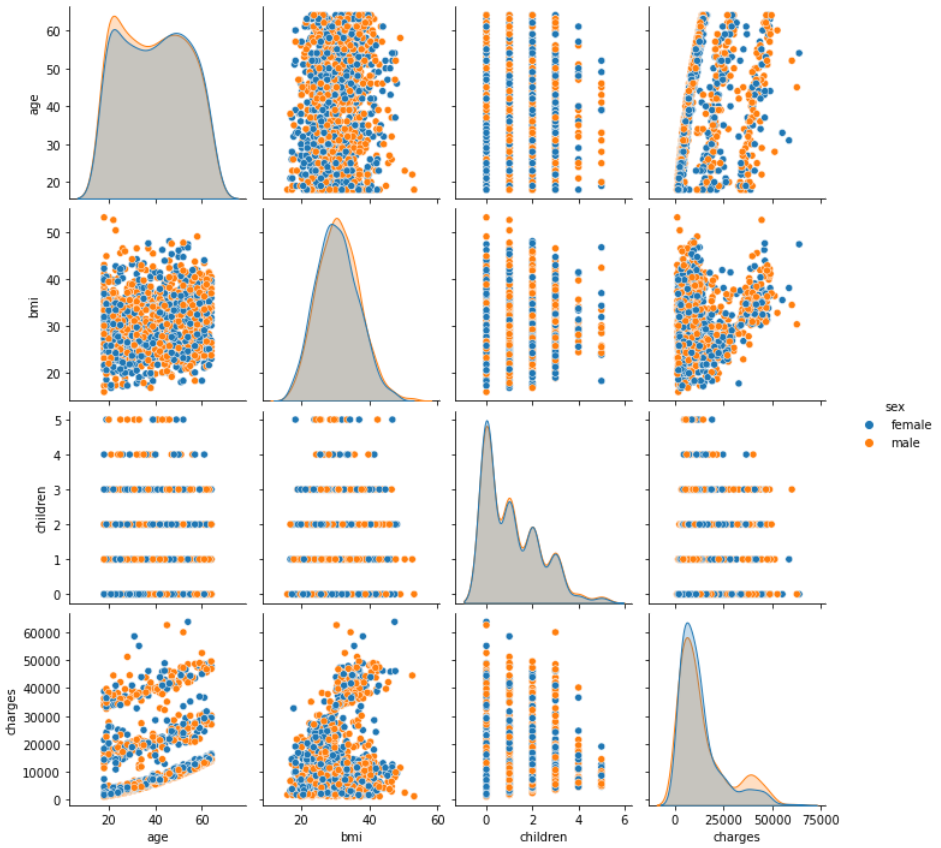
Consider making the categorical variable the legend. To do this, we first create the lists of numeric variables **num_list** and categorical variables **cat_list**.

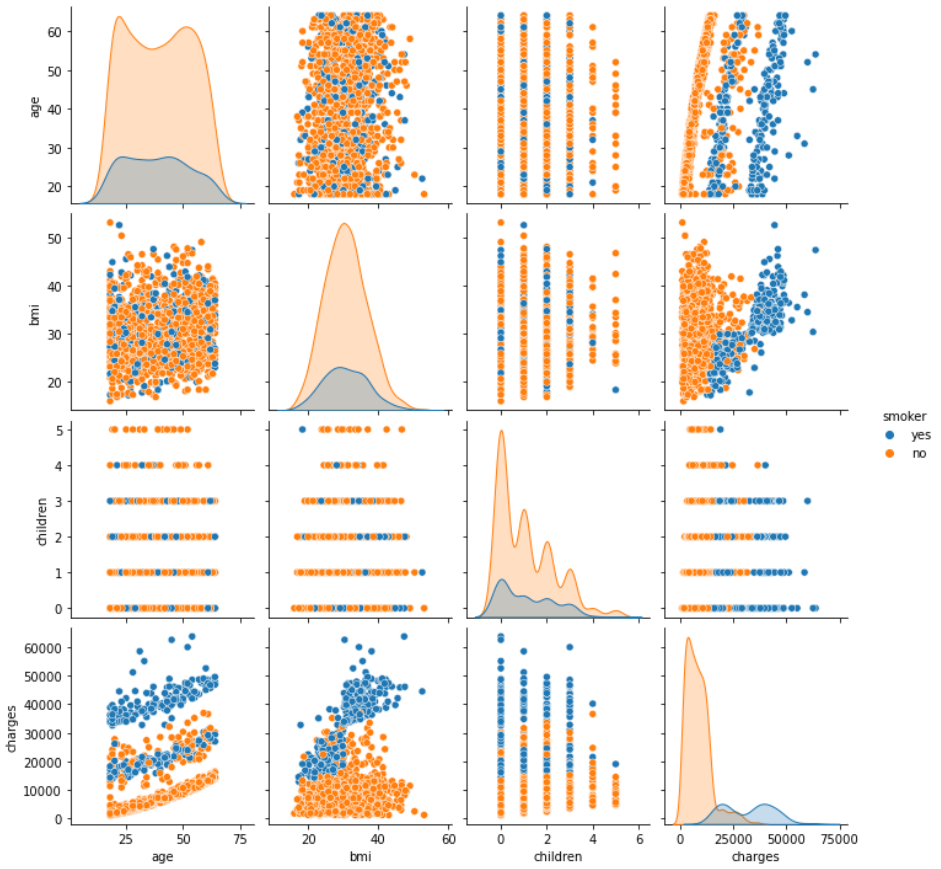
```
df.columns
```

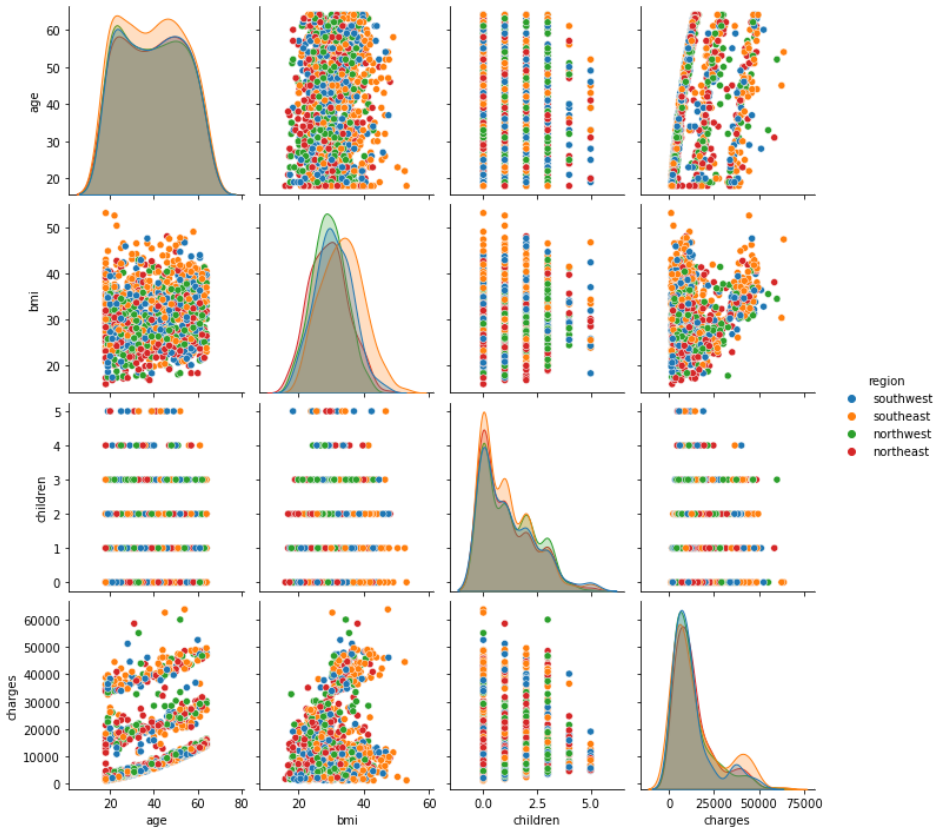
```
Index(['age', 'sex', 'bmi', 'children', 'smoker', 'region', 'charges'], dtype='object')
```

```
# Asserting column(s) data type in Pandas
# https://stackoverflow.com/questions/28596493/asserting-columns-data-type-in-pandas

import pandas.api.types as ptypes


num_list = []
cat_list = []

for column in df.columns:
    #print(column)
    if ptypes.is_numeric_dtype(df[column]):
        num_list.append(column)
    elif ptypes.is_string_dtype(df[column]):
        cat_list.append(column)
```

```
# pairplot with hue

for i in range(0,len(cat_list)):
    hue_cat = cat_list[i]
    sns.pairplot(df, hue = hue_cat)
```


age
bmi
children
charges
sex
female
male

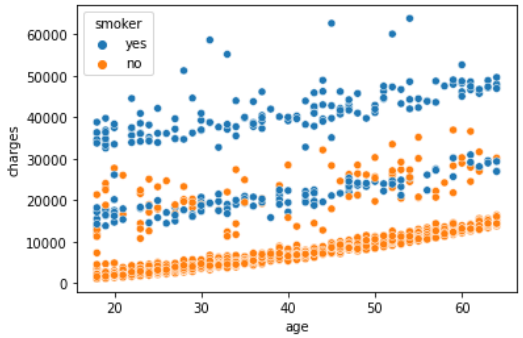

age
bmi
children
charges
smoker
yes
no




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

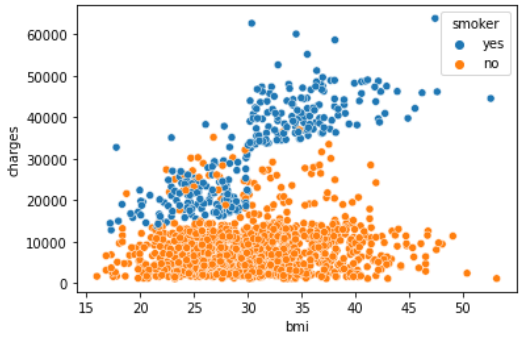
```
sns.scatterplot(x='age',y='charges', hue = 'smoker', data =df)
```

```
<AxesSubplot:xlabel='age', ylabel='charges'>
```




```
sns.scatterplot(x='bmi',y='charges', hue = 'smoker', data =df)
```

```
<AxesSubplot:xlabel='bmi', ylabel='charges'>
```



We can observe that age, bmi, and smoking status are all correlated to the target variable charges. Charges increase as you become older and your BMI rises. The Smoker variable clearly separates the data set into two groups. It suggests that the feature “smoker” could be a good predictor of charges.

Conclusion: age, bmi, and smoker are all key factors in determining charges. In the data set, sex and region do not exhibit any significant patterns.

Exercise Use plotnine to create the two scatter plots above.

6.3.1.3. Preprocessing categorical variables

Categorical variables cannot be handled by most machine learning algorithms unless they are converted to numerical values. The categorical variables must be preprocessed before they may be used. We must transform these categorical variables to integers in order for the model to comprehend and extract useful information.

We can encode these categorical variables as integers in a variety of ways and use them in our regression analysis. We'll take a look at two of them, including

- One Hot Encoding
- Label Encoding

6.3.1.4. One Hot Encoding

In this method, we map each category to a vector that comprises 1 and 0, indicating whether the feature is present or not. The number of vectors is determined by the number of feature categories. If the number of categories for the feature is really large, this approach produces many columns, which considerably slows down the model processing.

The **get_dummies** function in Pandas is quite simple to use. The following is a sample DataFrame code:

```
df_encoding1 = pd.get_dummies(df, columns = cat_list)
```

```
df_encoding1
```

	age	bmi	children	charges	sex_female	sex_male	smoker_no	smoker_yes	region_northeast	region_northl
0	19	27.900	0	16884.92400	1	0	0	1	0	
1	18	33.770	1	1725.55230	0	1	1	0	0	
2	28	33.000	3	4449.46200	0	1	1	0	0	
3	33	22.705	0	21984.47061	0	1	1	0	0	
4	32	28.880	0	3866.85520	0	1	1	0	0	
...	...	...	...	...	...	...	...	...	...	
1333	50	30.970	3	10600.54830	0	1	1	0	0	
1334	18	31.920	0	2205.98080	1	0	1	0	1	
1335	18	36.850	0	1629.83350	1	0	1	0	0	
1336	21	25.800	0	2007.94500	1	0	1	0	0	
1337	61	29.070	0	29141.36030	1	0	0	1	0	

1338 rows × 12 columns

Alternative to the above command, we also add two arguments

- `drop_first=True` to get k-1 dummies out of k categorical levels by removing the first level,

*`columns=['smoker','sex']` to specify the column names in the DataFrame to be encoded.


```
# Alternative to the above command, we also specify drop_first=True

df_encoding1 = pd.get_dummies(data=df, columns=['smoker','sex'], drop_first=True)
```

```
# print(df.head())
# df_encoding1.head()

# rename the column names
df_encoding1 = df_encoding1.rename(columns={"smoker_1":"smoker_yes","sex_1":"sex_male"})

df_encoding1
```

	age	bmi	children	region	charges	smoker_yes	sex_male
0	19	27.900	0	southwest	16884.92400	1	0
1	18	33.770	1	southeast	1725.55230	0	1
2	28	33.000	3	southeast	4449.46200	0	1
3	33	22.705	0	northwest	21984.47061	0	1
4	32	28.880	0	northwest	3866.85520	0	1
...	...	...	...	...	...	...	...
1333	50	30.970	3	northwest	10600.54830	0	1
1334	18	31.920	0	northeast	2205.98080	0	0
1335	18	36.850	0	southeast	1629.83350	0	0
1336	21	25.800	0	southwest	2007.94500	0	0
1337	61	29.070	0	northwest	29141.36030	1	0

1338 rows × 7 columns

6.3.1.5. Label Encoding

In this encoding, each category in this encoding is given a number between 1 and N (where N is the number of categories for the feature). One fundamental problem with this technique is that there is no relationship or order between these classes, despite the fact that the algorithm may treat them as such. (region0 < region1 < region2 and 0 < 1 < 2) is an example of what it might look like. The data-frame has the following Scikit-learn code:

```
df = pd.read_csv('/Users/Kaemyuijang/SCMA248/Data/insuranceKaggle.csv')
```

```
from sklearn import preprocessing
from sklearn.preprocessing import LabelEncoder

df_encoding2 = df

for i in cat_list:
    df_encoding2[i] = LabelEncoder().fit_transform(df[i])
```

```
df_encoding2
```

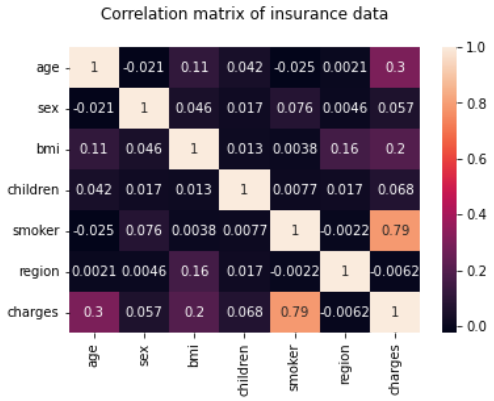
	age	sex	bmi	children	smoker	region	charges
0	19	0	27.900	0	1	3	16884.92400
1	18	1	33.770	1	0	2	1725.55230
2	28	1	33.000	3	0	2	4449.46200
3	33	1	22.705	0	0	1	21984.47061
4	32	1	28.880	0	0	1	3866.85520
...	...	...	...	...	...	...	...
1333	50	1	30.970	3	0	1	10600.54830
1334	18	0	31.920	0	0	0	2205.98080
1335	18	0	36.850	0	0	2	1629.83350
1336	21	0	25.800	0	0	3	2007.94500
1337	61	0	29.070	0	1	1	29141.36030

1338 rows × 7 columns

6.3.1.6. Correlation Analysis

The linear correlation between variable pairs is investigated using correlation analysis. This may be accomplished by combining the `corr()` and `sns.heatmap()` functions.

```
import seaborn as sns
import matplotlib.pyplot as plt

hm = sns.heatmap(df.corr(), annot = True)
hm.set(title = "Correlation matrix of insurance data\n")
plt.show()
```



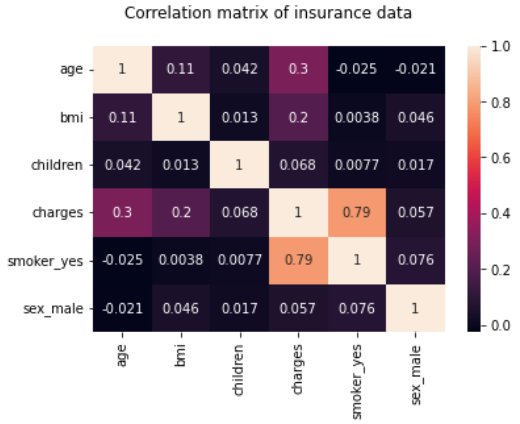
Note: The correlation matrix also confirms that age, bmi, and smoker are all key factors in determining charges.

In what follows, we will use one hot encoding for the **smoker** category.

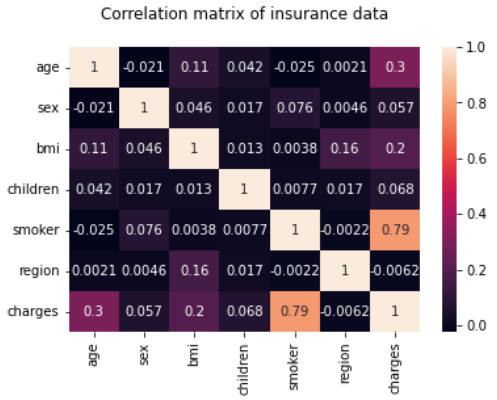
See https://python-graph-gallery.com/91-customize-seaborn-heatmap for customization of heatmap using seaborn.

```
#df = pd.read_csv('/Users/Kaemyuijang/SCMA248/Data/insuranceKaggle.csv')
#df = pd.get_dummies(df, columns = cat_list)
```

```
hm = sns.heatmap(df_encoding1.corr(), annot = True)
hm.set(title = "Correlation matrix of insurance data\n")
plt.show()
```




```
hm = sns.heatmap(df_encoding2.corr(), annot = True)
hm.set(title = "Correlation matrix of insurance data\n")
plt.show()
```



6.3.2. Multiple Linear Regression

Multiple inputs and a single output are possible in multiple linear regression. We'll look at how numerous input variables interact to affect the output variable, as well as how the calculations differ from those of a simple linear regression model. We'll also use Python's **Statsmodel** to create a multiple regression model.

Recall that the fitted regression model equation is

$$\hat{Y} = 3165.885 + 257.7226X.$$

Multiple regression model

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p + \varepsilon$$

Here, Y is the output variable, and $X_1, \ldots, X_p$ terms are the corresponding input variables.

Hence, our Linear Regression model can now be expressed as:

$$charges = \beta_0 + \beta_1 age + \beta_1 bmi + \beta_3 smoker\_yes + \varepsilon$$

Finding the values of these constants (β) is the task of the regression model, minimizing the error function and fitting the best line or hyperplane (depending on the number of input variables).

This is done by minimizing the Residual Sum of Squares (RSS), which is obtained by squaring the differences between the actual and predicted results.

6.3.2.1. Building the model and interpreting the coefficients

Similar to linear regression analysis, the results can be obtained by running the following code:

Fit and summary:

```
df_encoding1
```

	age	bmi	children	region	charges	smoker_yes	sex_male
0	19	27.900	0	southwest	16884.92400	1	0
1	18	33.770	1	southeast	1725.55230	0	1
2	28	33.000	3	southeast	4449.46200	0	1
3	33	22.705	0	northwest	21984.47061	0	1
4	32	28.880	0	northwest	3866.85520	0	1
...	...	...	...	...	...	...	...
1333	50	30.970	3	northwest	10600.54830	0	1
1334	18	31.920	0	northeast	2205.98080	0	0
1335	18	36.850	0	southeast	1629.83350	0	0
1336	21	25.800	0	southwest	2007.94500	0	0
1337	61	29.070	0	northwest	29141.36030	1	0

1338 rows × 7 columns

```
model3 = smf.ols(formula='charges ~ age + bmi + smoker_yes', data=df_encoding1).fit()
```

```
model3.summary()
```

Dep. Variable:	charges	**R-squared:**	0.747
Model:	OLS	**Adj. R-squared:**	0.747
Method:	Least Squares	**F-statistic:**	1316.
Date:	Sat, 09 Jul 2022	**Prob (F-statistic):**	0.00
Time:	17:36:44	**Log-Likelihood:**	-13557.
No. Observations:	1338	**AIC:**	2.712e+04
Df Residuals:	1334	**BIC:**	2.714e+04
Df Model:	3		
Covariance Type:	nonrobust		

OLS Regression Results

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-1.168e+04	937.569	-12.454	0.000	-1.35e+04	-9837.561
age	259.5475	11.934	21.748	0.000	236.136	282.959
bmi	322.6151	27.487	11.737	0.000	268.692	376.538
smoker_yes	2.382e+04	412.867	57.703	0.000	2.3e+04	2.46e+04

Omnibus:	299.709	**Durbin-Watson:**	2.077
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	710.137
Skew:	1.213	**Prob(JB):**	6.25e-155
Kurtosis:	5.618	**Cond. No.**	289.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Quantities of interest can be extracted directly from the fitted model. Type dir(results) for a full list. Here are some examples:


```
print("Parameters: ", model3.params)
print("R2: ", model3.rsquared)
```

```
Parameters:  Intercept    -11676.830425
age             259.547492
bmi             322.615133
smoker_yes    23823.684495
dtype: float64
R2:  0.7474771588119513
```

6.3.2.2. Fitted multiple regression function

Based on the previous results, the equation to predict the output variable using age, bmi, and smoker yes as input would be as follows:

$$charges = -11676.83042519 + (259.54749155 * age) + (322.61513282 * bmi) + (23823.6844953$$

6.3.2.3. Example of calculating fitted values

If sample data with actual output value 8240.5896 has

- age = 46,
- bmi = 33.44 and *smoker yes = 0,

then we substitute the value of age for age, bmi and smoker_yes in the previous equation, we get

charges = 11050.6042276108 = - 11676.83042519 + (259.5474915*546) + (322.6151328*33.44) + (23823.68449531 * 0)

If we construct a multiple linear regression model and use age, bmi, and smoker yes as input variables, a 46-year-old person will have to pay an insurance charge of 11050.6042276108.

We can see that the anticipated value is rather close to the real value. As a result, we can conclude that the Multiple Linear Regression model outperforms the Simple Linear Regression model.

Note Based on the linear regression result, equation to predict output variable **using only age as an input** would be like,

charges = (257.72261867 * age) + 3165.88500606

If we will put value of age = 46 in above equation then,

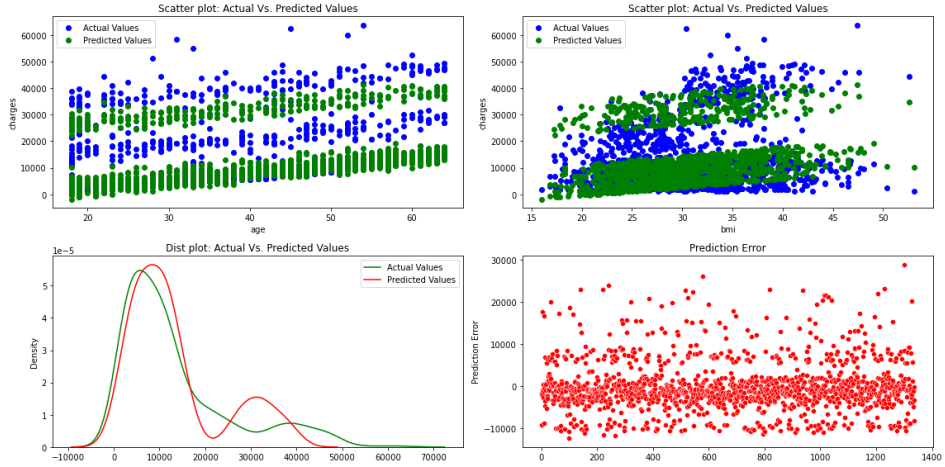
charges = 15021.12546488 = (257.72261867 * 46) + 3165.88500606

Here we can see that the predicted value is almost double. So we can say that the simple linear regression model does not work well.

```
x=df[['age']]
y=df[['charges']]

predictions = model1.predict(x)
df['slr_result'] = predictions

df['slr_error'] =  df['charges'] - df['slr_result']
```

```
x=df_encoding1[['age','bmi','smoker_yes']]
y=df_encoding1[['charges']]

predictions = model3.predict(x)
df_encoding1['mlr_result'] = predictions

df_encoding1['mlr_error'] =  df_encoding1['charges'] - df_encoding1['mlr_result']


fig, axes =plt.subplots(2,2, figsize=(16,8))
axes[0][0].plot(x['age'], y,'bo',label='Actual Values')
axes[0][0].plot(x['age'], predictions,'go',label='Predicted Values')
axes[0][0].set_title("Scatter plot: Actual Vs. Predicted Values")
axes[0][0].set_xlabel("age")
axes[0][0].set_ylabel("charges")
axes[0][0].legend()


axes[0][1].plot(x['bmi'], y,'bo',label='Actual Values')
axes[0][1].plot(x['bmi'], predictions,'go',label='Predicted Values')
axes[0][1].set_title("Scatter plot: Actual Vs. Predicted Values")
axes[0][1].set_xlabel("bmi")
axes[0][1].set_ylabel("charges")
axes[0][1].legend()

sns.distplot(y, hist=False, color="g", label="Actual Values",ax=axes[1][0])
sns.distplot(predictions, hist=False, color="r", label="Predicted Values" , ax=axes[1]
[0])
axes[1][0].set_title("Dist plot: Actual Vs. Predicted Values")
axes[1][0].legend()

sns.scatterplot(x=y.index,y='mlr_error',data=df_encoding1,color="r", ax=axes[1][1])
axes[1][1].set_title("Prediction Error")
axes[1][1].set_ylabel("Prediction Error")

fig.tight_layout()
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated
function and will be removed in a future version. Please adapt your code to use either
`displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level
function for kernel density plots).
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated
function and will be removed in a future version. Please adapt your code to use either
`displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level
function for kernel density plots).
```



The figures show that the multiple linear regression model fits better than the simple linear regression because the predicted values are close to the observed data values

```
(
    ggplot(df, aes(x='df.index'))
    +  geom_point(aes(y='slr_error'),color='red')
    + labs(x = ' ', y='Prediction error')
    + labs(title = 'Prediction Error: Linear Regression')
)
```


```
<ggplot: (311177589)>
```

```
# print('length of x:',len(x))
#print('length of y:',len(y))


# print(df_encoding1)
```

```
df_encoding1
```

	age	bmi	children	region	charges	smoker_yes	sex_male	mlr_result	mlr_error
0	19	27.900	0	southwest	16884.92400	1	0	26079.218615	-9194.294615
1	18	33.770	1	southeast	1725.55230	0	1	3889.737458	-2164.185158
2	28	33.000	3	southeast	4449.46200	0	1	6236.798721	-1787.336721
3	33	22.705	0	northwest	21984.47061	0	1	4213.213387	17771.257223
4	32	28.880	0	northwest	3866.85520	0	1	5945.814340	-2078.959140
...	...	...	...	...	...	...	...	...	...
1333	50	30.970	3	northwest	10600.54830	0	1	11291.934816	-691.386516
1334	18	31.920	0	northeast	2205.98080	0	0	3292.899462	-1086.918662
1335	18	36.850	0	southeast	1629.83350	0	0	4883.392067	-3253.558567
1336	21	25.800	0	southwest	2007.94500	0	0	2097.137324	-89.192324
1337	61	29.070	0	northwest	29141.36030	1	0	37357.672966	-8216.312666

1338 rows × 9 columns

6.4. Scikit-learn regression models

Scikit-learn lets you perform linear regression in Python in a different way.

When it comes to machine learning in Python, Scikit-learn (or SKLearn) is the gold standard. There are many learning techniques available for regression, classification, clustering, and dimensionality reduction.

To use linear regression, we first need to import it:


```
from sklearn import linear_model
```

Let us use the same insurance cost dataset as before. Importing the datasets from SKLearn and loading the insurance dataset would initially be the same procedure:

```
df = pd.read_csv('/Users/Kaemyuijang/SCMA248/Data/insuranceKaggle.csv')
```

6.4.1. Create a model and fit it

In a simple linear regression, there is only one input variable and one output variable.

First Let us take age as the input variable and charges as the output. We will define our x and y.

We then import the LinearRegression class from Scikit-Learn's `linear_model`. Then you can instantiate a new `LinearRegression` object. In this case it was called `lm`.

```
# Creating new variables

x=df[['age']]    #a two-dimensional array
y=df['charges']  #a one-dimensional array

# Creating a new model
lm = linear_model.LinearRegression()
```

```
print(x.shape)
print(y.shape)
```

```
(1338, 1)
(1338,)
```

The `.fit()` function fits a linear model. We want to make predictions with the model, so we use `.predict()`:

Note The method requires two arrays: X and y, according to the help documentation. The capital X denotes a two-dimensional array, whereas the capital Y denotes a one-dimensional array.

```
# fit the model
lm_model = lm.fit(x,y)

# Make predictions with the model
predictions = lm_model.predict(x)

df['slr_result'] = predictions

df['slr_error'] =  df['charges'] - df['slr_result']
```

Note that `.predict()` uses the linear model we fitted to predict the y (dependent variable).

You have probably noticed that when we use SKLearn to perform a linear regression, we do not get a nice table like we do with **Statsmodels**.

We may return the score, coefficients, and estimated intercepts by using built-in functions. Let us take a look at how it works:

The following code returns the R-squared (or the coefficient of determination) score. This is the proportion of explained variation of the predictions, as you may remember.

```
from sklearn.metrics import r2_score

r2_score(y.values , predictions)
```

```
0.08940589967885804
```

```
# alternative to r2_score

lm.score(x,y)
```



```
0.08940589967885804
```

```
print ('Slope: ', lm.coef_)
print ('Intercept: ',lm.intercept_)
```

```
Slope:  [257.72261867]
Intercept:  3165.885006063021
```

The results of the regression analysis, including the R-squared, slope, and intercept, are the same as for **statsmodel**.

6.4.2. The algorithm's evaluation

The final stage is to evaluate the algorithm's performance.

This phase is especially significant when comparing the performance of multiple algorithms on a given data set. Regression methods are often evaluated using three metrics:

1. **Mean Absolute Error (MAE)** is the mean of the absolute value of the errors. It is calculated as:

$$\text{MAE} = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n}$$

where $\hat{y}_i$ is the prediction and y_i the true value.

1. Mean Squared Error (MSE) is the mean of the squared errors and is calculated as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2.$$

1. Root Mean Squared Error (RMSE) is the square root of the mean of the squared errors:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}.$$

Fortunately, we do not have to do these calculations manually. The Scikit-Learn library has pre-built functions that we can use to determine these values for us.

Let us determine the values for these metrics using our test data. Run the following code:

```
from sklearn.metrics import *

print('Mean Absolute Error:', mean_absolute_error(y.values , predictions))
print('Mean Squared Error:', mean_squared_error(y.values , predictions))
print('Root Mean Squared Error:', np.sqrt(mean_squared_error(y.values , predictions)))
```

```
Mean Absolute Error: 9055.14962050455
Mean Squared Error: 133440978.61376347
Root Mean Squared Error: 11551.66562075632
```

```
#Alternatively,

output_df = pd.DataFrame(columns=['MAE','MSE','R2-Score'],index=['Linear
Regression','Multiple Linear Regression'])


output_df['MAE']['Linear Regression'] = np.mean(np.absolute(predictions - y.values))
output_df['MSE']['Linear Regression'] = np.mean((predictions - y.values) ** 2)
output_df['R2-Score']['Linear Regression'] = r2_score(y.values , predictions)
```

```
output_df
```

	MAE	MSE	R2-Score
Linear Regression	9055.149621	133440978.613763	0.089406
Multiple Linear Regression	NaN	NaN	NaN

In all three approaches, the errors are calculated by taking the difference between predicted values and actual value, and **the lower the difference, the better**.

The primary distinction is that MSE/RMSE **punish large errors** and are differentiable, whereas MAE is not, making it difficult to use in gradient descent.

RMSE, in contrast to MSE, takes the square root, preserving the original data scale.

Exercise

1. Perform a regression analysis using scikit-learn with the multiple regression model with age, bmi, and smoker as predictors.
2. Evaluate the model using the three matrices MAE, MSE, RMSE and store the values in the DataFrame **output_df**.

6.5. Multivariate Linear Regression in Scikit-Learn

We will perform linear regression using multiple variables.

Scikit-Learn makes creating these models a breeze. Remember passing a two-dimensional array `x` when you originally trained your model? In that array, there was only one column.

To match your data with multiple variables, you can simply pass in an array with multiple columns. Let us take a look at how this works:

6.5.1. Categorical variable encoding

```
df = pd.read_csv('/Users/Kaemyuijang/SCMA248/Data/insuranceKaggle.csv')

# Categorical variable encoding
df_encoding1 = pd.get_dummies(data=df, columns=['smoker','sex'], drop_first=True)
```

```
# print(df.head())
# df_encoding1.head()

# rename the column names
df_encoding1 = df_encoding1.rename(columns={"smoker_1":"smoker_yes","sex_1":"sex_male"})

df_encoding1
```

	age	bmi	children	region	charges	smoker_yes	sex_male
0	19	27.900	0	southwest	16884.92400	1	0
1	18	33.770	1	southeast	1725.55230	0	1
2	28	33.000	3	southeast	4449.46200	0	1
3	33	22.705	0	northwest	21984.47061	0	1
4	32	28.880	0	northwest	3866.85520	0	1
...	...	...	...	...	...	...	...
1333	50	30.970	3	northwest	10600.54830	0	1
1334	18	31.920	0	northeast	2205.98080	0	0
1335	18	36.850	0	southeast	1629.83350	0	0
1336	21	25.800	0	southwest	2007.94500	0	0
1337	61	29.070	0	northwest	29141.36030	1	0

1338 rows × 7 columns

```
# Creating new variables
x=df_encoding1[['age','bmi','smoker_yes']]
y=df_encoding1['charges']


# Creating a new model and fitting it
multi = linear_model.LinearRegression()
multi_model = multi.fit(x, y)


predictions = multi_model.predict(x)
df_encoding1['mlr_result'] = predictions

mlr_error = y - predictions
df_encoding1['mlr_error'] = mlr_error

print ('Slope: ', multi_model.coef_)
print ('Intercept: ',multi_model.intercept_)

print("Mean absolute error: %.2f" % np.mean(np.absolute(predictions - y.values)))
print("Residual sum of squares (MSE): %.2f" % np.mean((predictions - y.values) ** 2))
print("R2-score: %.2f" % r2_score(y.values , predictions) )

output_df['MAE']['Multiple Linear Regression'] = np.mean(np.absolute(predictions -
y.values))
output_df['MSE']['Multiple Linear Regression'] = np.mean((predictions - y.values) ** 2)
output_df['R2-Score']['Multiple Linear Regression'] = r2_score(y.values , predictions)
```

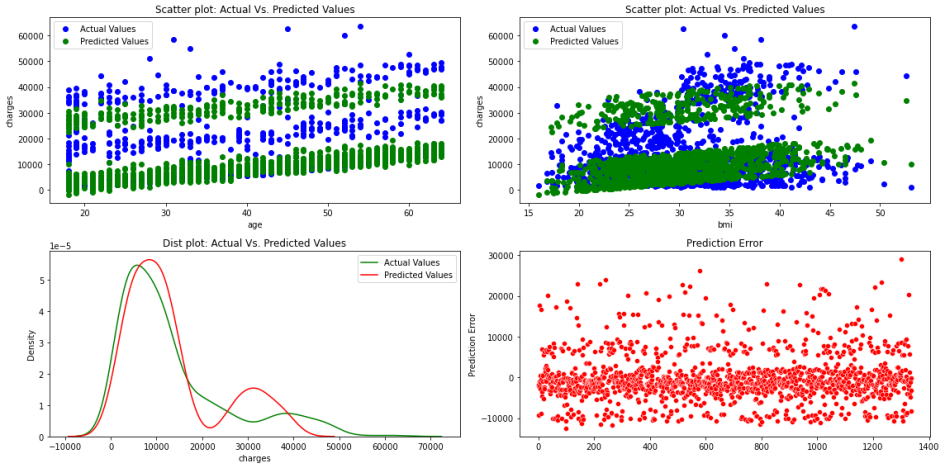
```
Slope:  [  259.54749155   322.61513282 23823.68449531]
Intercept:  -11676.830425187785
Mean absolute error: 4216.78
Residual sum of squares (MSE): 37005395.75
R2-score: 0.75
```

Again, the results of the multiple regression analysis, including the R-squared, the coefficients, and the intercept, are the same as for **statsmodel**.

6.5.2. Visualize Predictions Of Multiple Linear Regression with Plotnine

```
fig, axes =plt.subplots(2,2, figsize=(16,8))
axes[0][0].plot(x['age'], y,'bo',label='Actual Values')
axes[0][0].plot(x['age'], predictions,'go',label='Predicted Values')
axes[0][0].set_title("Scatter plot: Actual Vs. Predicted Values")
axes[0][0].set_xlabel("age")
axes[0][0].set_ylabel("charges")
axes[0][0].legend()

axes[0][1].plot(x['bmi'], y,'bo',label='Actual Values')
axes[0][1].plot(x['bmi'], predictions,'go',label='Predicted Values')
axes[0][1].set_title("Scatter plot: Actual Vs. Predicted Values")
axes[0][1].set_xlabel("bmi")
axes[0][1].set_ylabel("charges")
axes[0][1].legend()

sns.distplot(y, hist=False, color="g", label="Actual Values",ax=axes[1][0])
sns.distplot(predictions, hist=False, color="r", label="Predicted Values" , ax=axes[1]
[0])
axes[1][0].set_title("Dist plot: Actual Vs. Predicted Values")
axes[1][0].legend()

sns.scatterplot(x=y.index,y='mlr_error',data=df_encoding1,color="r", ax=axes[1][1])
axes[1][1].set_title("Prediction Error")
axes[1][1].set_ylabel("Prediction Error")

fig.tight_layout()
```

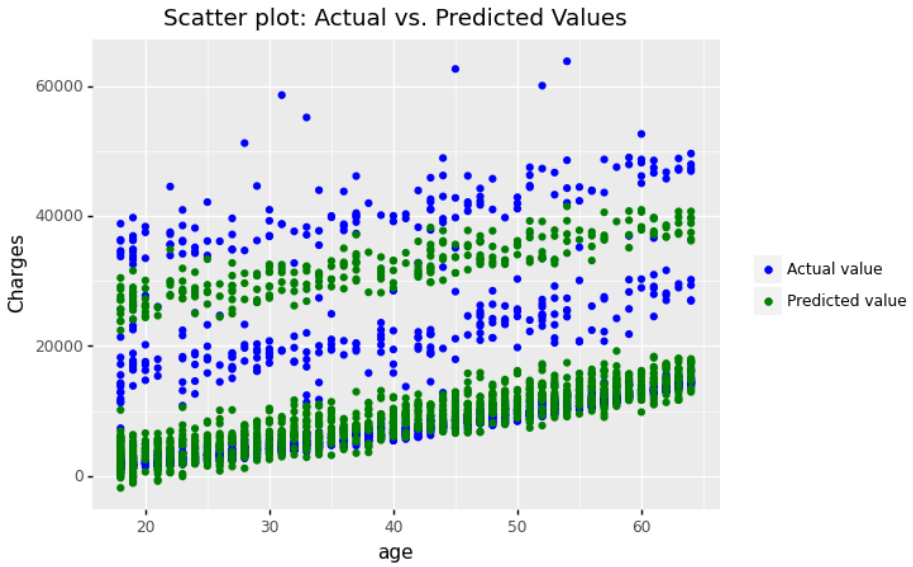
```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated
function and will be removed in a future version. Please adapt your code to use either
`displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level
function for kernel density plots).
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated
function and will be removed in a future version. Please adapt your code to use either
`displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level
function for kernel density plots).
```




```
df_encoding1.head()
```

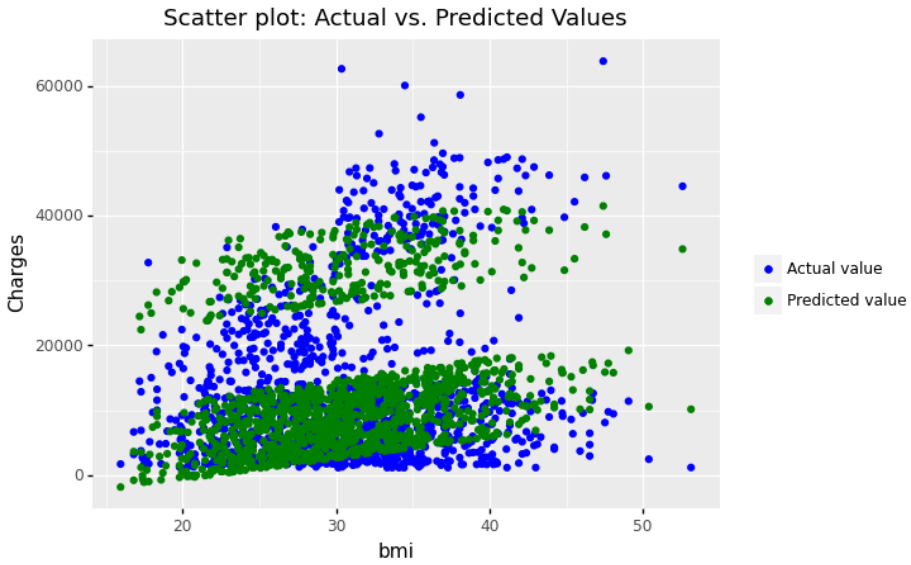
	age	bmi	children	region	charges	smoker_yes	sex_male	mlr_result	mlr_error
0	19	27.900	0	southwest	16884.92400	1	0	26079.218615	-9194.294615
1	18	33.770	1	southeast	1725.55230	0	1	3889.737458	-2164.185158
2	28	33.000	3	southeast	4449.46200	0	1	6236.798721	-1787.336721
3	33	22.705	0	northwest	21984.47061	0	1	4213.213387	17771.257223
4	32	28.880	0	northwest	3866.85520	0	1	5945.814340	-2078.959140

```
(
    ggplot(df_encoding1)
    + geom_point(aes(x = 'age', y='charges',color='"Actual value"'))
    + geom_point(aes(x = 'age', y='mlr_result',color='"Predicted value"'))
    #+ geom_smooth(aes(x = 'bmi', y = 'mlr_result', color='"Predicted value"'))
    + scale_color_manual(values = ['blue','green'], # Colors
        name = " ")
    + labs(y='Charges', title = 'Scatter plot: Actual vs. Predicted Values')
)
```



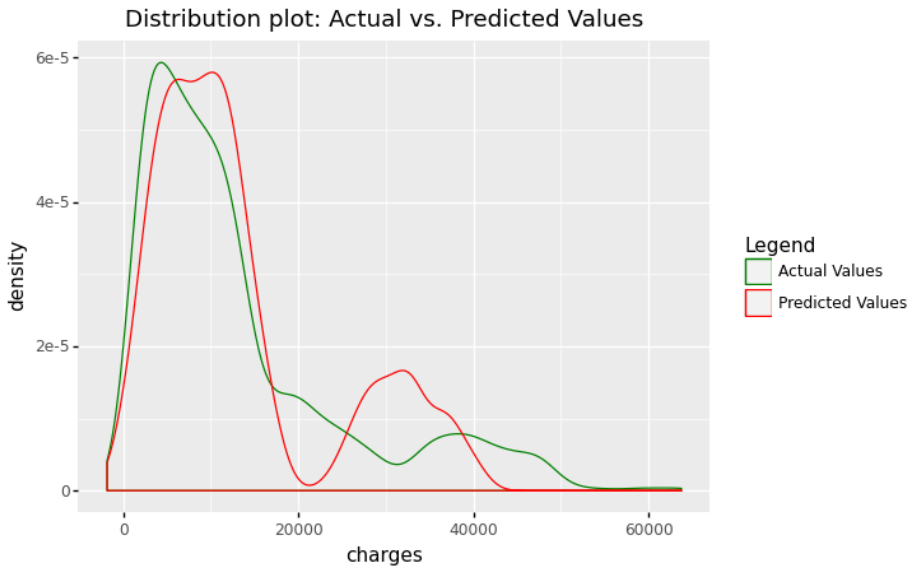

```
<ggplot: (306833141)>
```

```
(
    ggplot(df_encoding1)
    + geom_point(aes(x = 'bmi', y='charges',color='"Actual value"'))
    + geom_point(aes(x = 'bmi', y='mlr_result',color='"Predicted value"'))
    #+ geom_smooth(aes(x = 'bmi', y = 'mlr_result', color='"Predicted value"'))
    + scale_color_manual(values = ['blue','green'], # Colors
        name = " ")
    + labs(y='Charges', title = 'Scatter plot: Actual vs. Predicted Values')
)
```



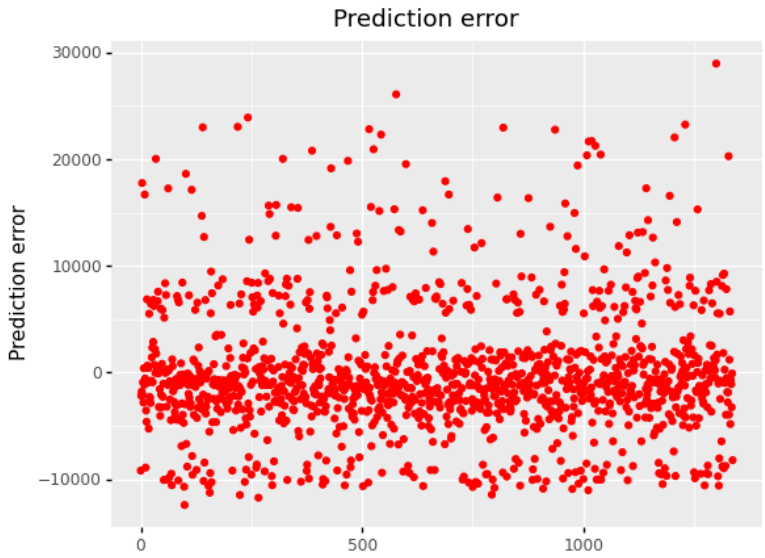

```
<ggplot: (305165713)>
```

```
(
    ggplot(df_encoding1, aes(x='charges'))
    +  geom_density(aes(y=after_stat('density'),color='"Actual Values"' ))
    +  geom_density(aes(x='mlr_result',y=after_stat('density'),color='"Predicted
Values"'))
    +  scale_color_manual(values = ['green','red'], name = 'Legend')
    +  labs(title = 'Distribution plot: Actual vs. Predicted Values')


)
```




```
<ggplot: (311690677)>
```

```
(
    ggplot(df_encoding1, aes(x='df.index'))
    +  geom_point(aes(y='mlr_error'),color='red')
    + labs(x = ' ', y='Prediction error')
    + labs(title = 'Prediction error')
)
```




```
<ggplot: (304769825)>
```

6.5.3. Compare Results and Conclusions

The error distribution can be used to assess the quality of a linear regression model (details given elsewhere). Quantitative measurements like MAE, MSE, RMSE, and R squared are also available for model comparison.

```
print(output_df)
```

```
                                MAE              MSE  R2-Score
Linear Regression          9055.149621  133440978.613763  0.089406
Multiple Linear Regression  4216.775692   37005395.750508  0.747477
```

6.5.3.1. Conclusions

By comparing the simple linear regression and multiple linear regression models, we conclude that the multiple linear regression is the best model, achieving 74.75% accuracy (R-squared).

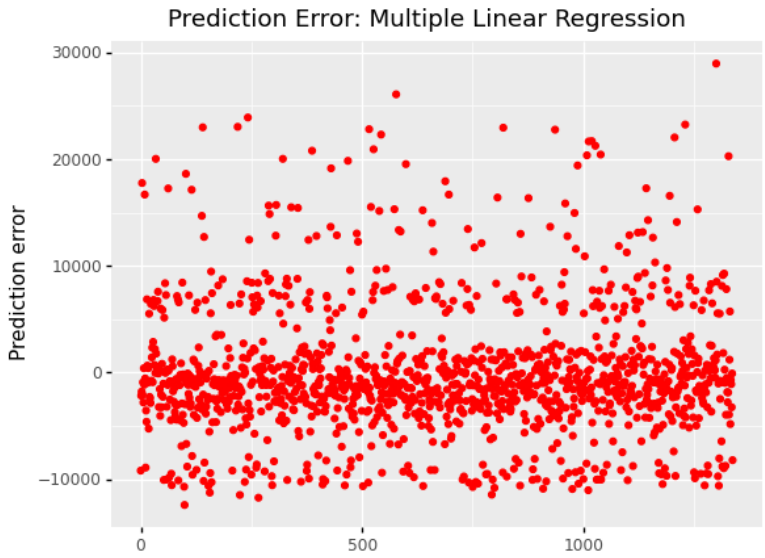
In this situation, an R-squared value of 0.7475 indicates that the multiple linear regression model explains 74.75 percent of the variation in the target variable, which is generally considered as a good rate.

When we look at error graphs, we can see that

- The error varies from -10,000 to 50,000 in simple linear regression.
- The error ranges from -10,000 to 30,000 in Multiple Linear Regression.

```
#(
#    ggplot(df, aes(x='df.index'))
#    +  geom_point(aes(y='slr_error'),color='red')
#    + labs(x = ' ', y='Prediction error')
#    + labs(title = 'Prediction Error: Linear Regression')
#)
```

```
(
    ggplot(df_encoding1, aes(x='df.index'))
    +  geom_point(aes(y='mlr_error'),color='red')
    + labs(x = ' ', y='Prediction error')
    + labs(title = 'Prediction Error: Multiple Linear Regression')
)
```




```
<ggplot: (309356937)>
```

Previous
5. Practical Statistics

Next
7. Machine learning: Introduction

By Pairote Satiracoo
© Copyright 2021.

7 Machine learning: Introduction

7. Machine learning: Introduction

Machine learning is clearly one of the most powerful and significant technologies in the world today. And more importantly, we have yet to fully realize its potential. It will undoubtedly continue to make headlines for the foreseeable future.

Machine learning is **a technique for transforming data into knowledge**. In the last 50 years, there has been a data explosion. This vast amount of data is worthless until we analyze it and uncover the underlying patterns.

Machine learning techniques are being used to **discover useful underlying patterns in complex data** that would otherwise be difficult to find. Hidden patterns and problem knowledge can be used to predict future events and make a variety of complex decisions.

7.1. What Is Machine Learning?

Machine learning is the study of computer algorithms that can learn and develop on their own with experience and data.

It is considered to be a component of **artificial intelligence**.

Machine learning algorithms create a model based on **training data** to make predictions or decisions without having to be explicitly programmed to do so.

7.1.1. Building models of data

It makes more sense to think of machine learning as a means of **building models of data**.

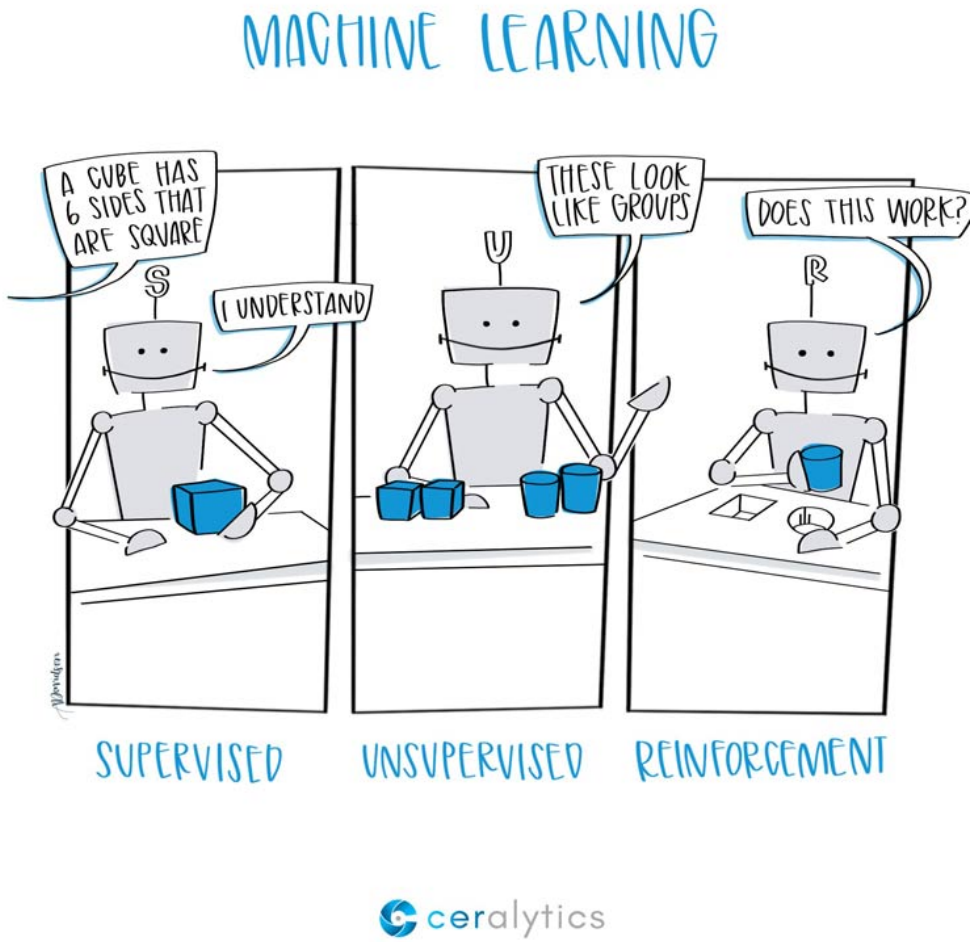
Machine learning is fundamentally about building mathematical models that facilitate the understanding of data.

If we provide these models with **tunable parameters** that can be adapted to the observed data, we can call the program **“learning” from the data**.

These models can be used to **predict and understand features of newly observed data** after fitting them to previously seen data.

Contents

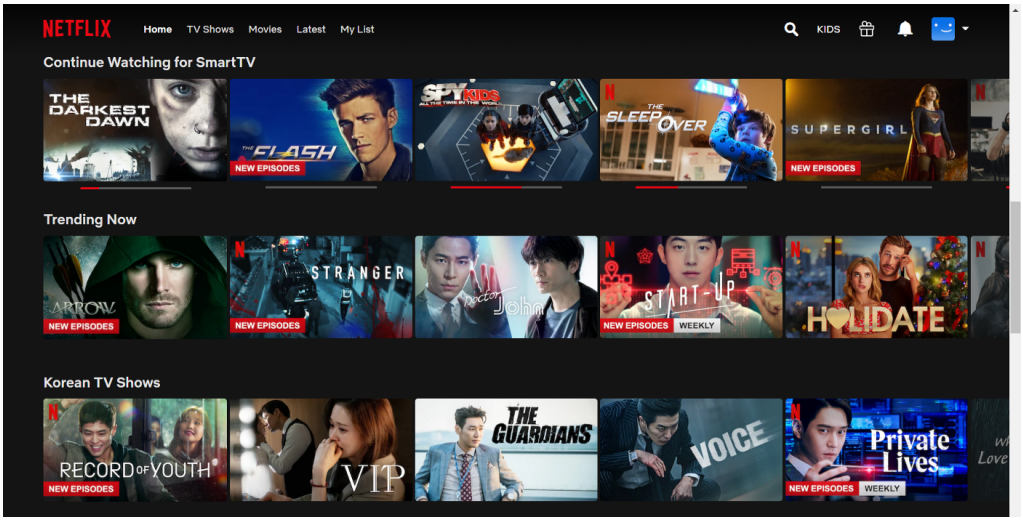
Categories of Machine Learning: image from ceralytics



7.2. Applications of Machine learning

Machine learning tasks can be used for a variety of things. Here are some examples of traditional machine learning tasks:

1. **Recommendation systems**



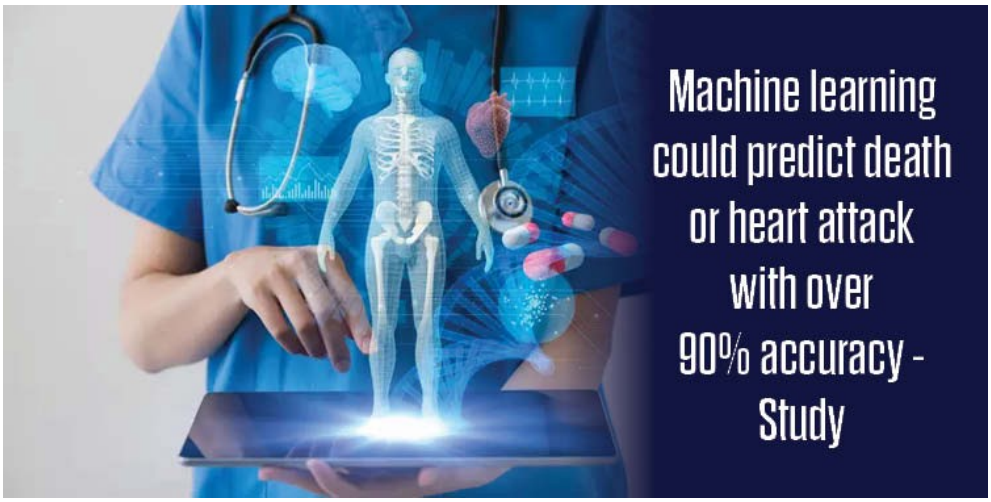
We come across a variety of online recommendation engines and methods. Many major platforms, such as Amazon, Netflix, and others, use these technologies. These recommendation engines use a machine learning system that takes into account user search results and preferences.

The algorithm uses this information to make similar recommendations the next time you open the platform.

You will receive notifications about new programs on Netflix. Netflix's algorithm checks the entire viewing history of its subscribers. It uses this information to suggest new series based on the preferences of its millions of active viewers.

The same recommendation engine can also be used to create ads. Take Amazon, for example. Let us say you go to Amazon to store or just search for something. Amazon's machine learning technology analyzes the user's search results and then generates ads as recommendations.

1. **Machine learning for Illness Prediction Healthcare use cases in healthcare**.



Doctors can warn patients ahead of time if they can predict a disease. They can even tell if a disease is dangerous or not, which is quite remarkable. But even though using ML is not an easy task, it can be of great benefit.

In this case, the ML algorithm first looks for symptoms on the patient's body. It would use abnormal body functions as input, train the algorithm, and then make a prediction based on that. Since there are hundreds of diseases and twice as many symptoms, it may take some time to get the results.

1. **Credit score - banking machine learning examples**.

It can be difficult to determine whether a bank customer is creditworthy. This is critical because whether or not the bank will grant you a loan depends on it.

Traditional credit card companies only check to see if the card is current and perform a history check. If the cardholder does not have a card history, the assessment becomes more difficult. For this, there are a number of machine learning algorithms that take into account the user's financial situation, previous credit repayments, debts and so on.

Due to a large number of defaulters, banks have already suffered significant financial losses. To limit these types of losses, we need an effective machine learning system that can prevent any of these scenarios from occurring. This would save banks a lot of money and allow them to provide more services to real consumers.

7.3. Categories of Machine Learning

Machine learning can be divided into two forms at the most basic level: supervised learning and unsupervised learning.

7.3.1. Supervised learning

Supervised learning involves determining how to model the relationship between measured data features and a label associated with the data; once this model is determined, it can be used to apply labels to new, unknown data. This is further divided into **classification** and **regression** tasks, where the **labels in classification are discrete categories** and the **labels in regression are continuous values**.

7.3.1.1. Classification problems (more examples below)

The output of a classification task is a discrete value. "Likes adding sugar to coffee" and "does not like adding sugar to coffee," for example, are discrete. There is no such thing as a middle ground. This is similar to teaching a child to recognize different types of animals, whether they are pets or not.

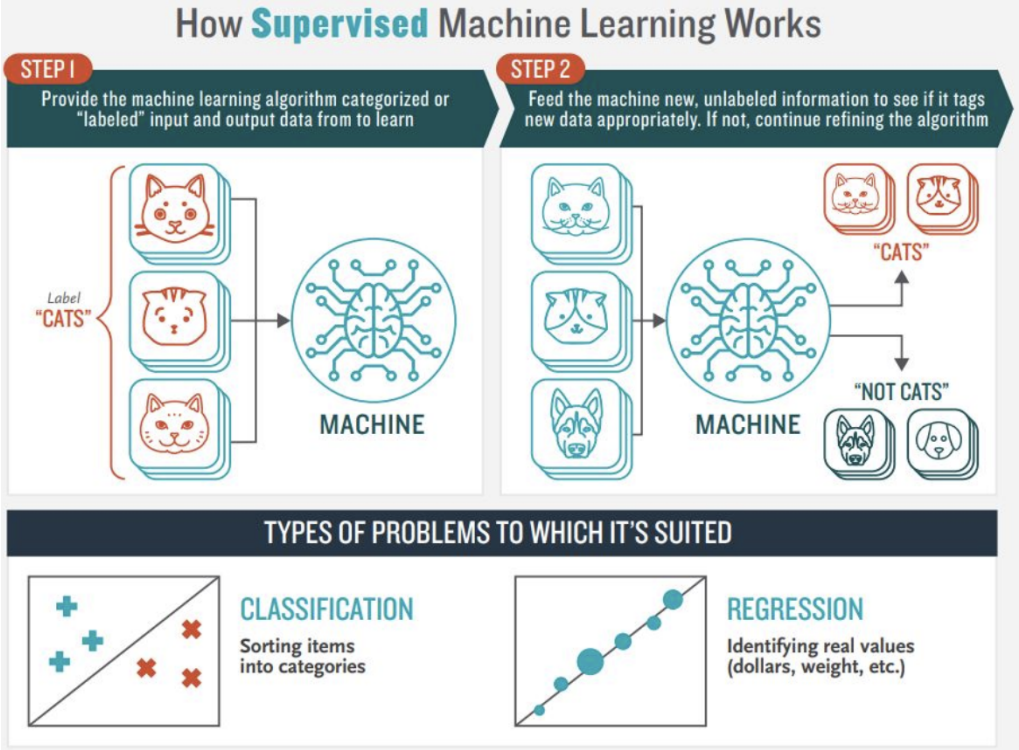
The output (label) of a classification method is typically represented as an integer number such as 1, -1, or 0. These figures are solely symbolic in this situation. Mathematical operations should not be performed with them because this would be pointless. Consider this for a moment. What is the difference between "Likes adding sugar to coffee" and "does not like adding sugar to coffee"? Exactly. We won't be able to add them, therefore we won't.

7.3.1.2. Regression problem (discussed in our last chatpter)

The outcome of a regression problem is a real number (a number with a decimal point). We could, for example, use the height and weight information to estimate someone's weight based on their height.

The data for a regression analysis will like the data in insurance data set. A **dependent variable** (or set of independent variables) and an **independent variable** (the thing we are trying to guess given our independent variables) are both present.

We could state that height is the independent variable and weight is the dependent variable, for example. In addition, each row in the dataset is commonly referred to as an **example, observation, or data point**, but each column (without the **label/dependent variable**) is commonly referred to as a **predictor, independent variable, or feature**.

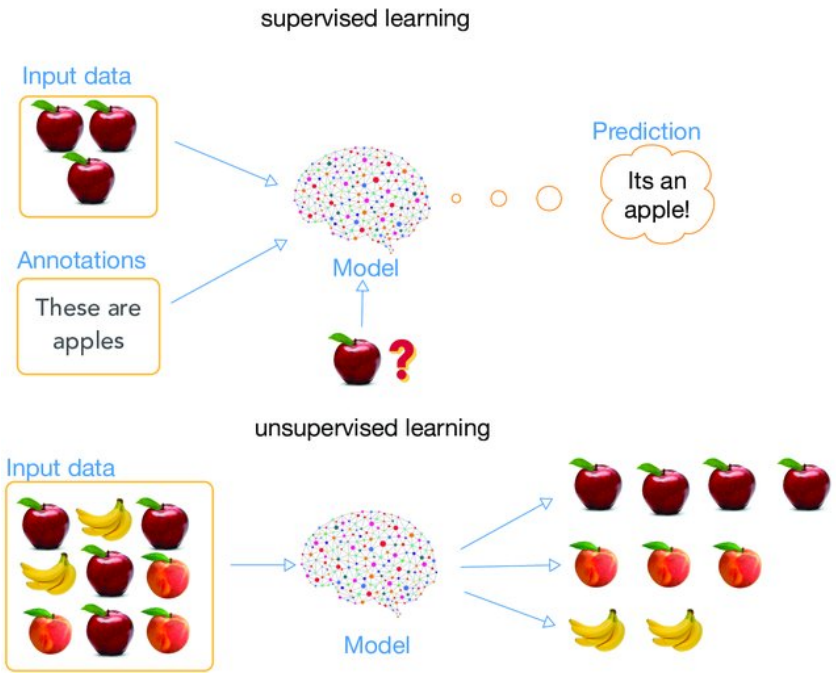


7.3.2. Unsupervised learning (more details in the next chapter)

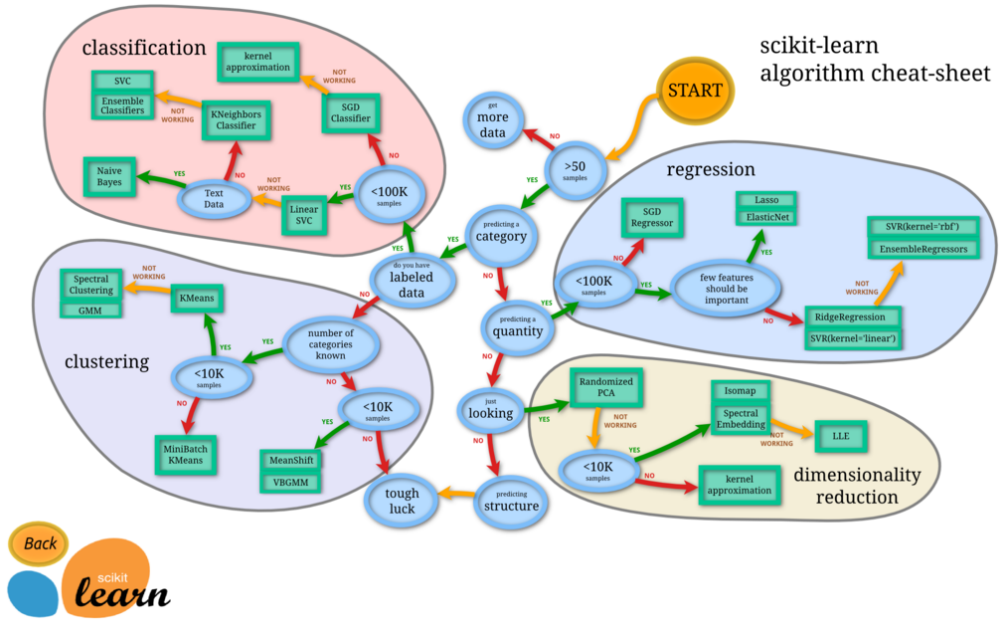
Unsupervised learning, sometimes known as "letting the dataset speak for itself," models the features of a dataset without reference to a label. Clustering and dimensionality reduction are among the tasks these models perform.

Clustering methods find unique groups of data, while **dimensionality reduction** algorithms look for more concise representations.

7.3.3. Supervised vs Unsupervised Learning: image from researchgate



7.3.4. Various classification, regression and clustering algorithms: image from scikit-learn



7.4. Machine Learning Basics with the K-Nearest Neighbors Algorithm

We will learn what **K-nearest neighbours (KNN)** is, how it works, and how to find the right k value. We will utilize the well-known Python library sklearn to demonstrate how to use KNN.

7.4.1. K-nearest neighbours can be summarized as follows:

- K- Nearest Neighbors is a **supervised machine learning** approach since the target variable is known,

- It is **non-parametric**, since no assumptions are made about the underlying data distribution pattern.
- It predicts the cluster into which the new point will fall based on feature similarity.

Both classification and regression prediction problems can be solved with KNN. However, since most analytical problems require making a decision, it is more commonly used in classification problems .

7.4.2. KNN algorithm's theory

The KNN algorithm's concept is one of the most straightforward of all the supervised machine learning algorithms.

It simply calculates the distance between a new data point and all previous data points in the training set.

Any form of distance can be used, such as

- Euclidean or
- Manhattan distances.

The K-nearest data points are then chosen, where K can be any integer. Finally, the data point is assigned to the class that contains the majority of the K data points.

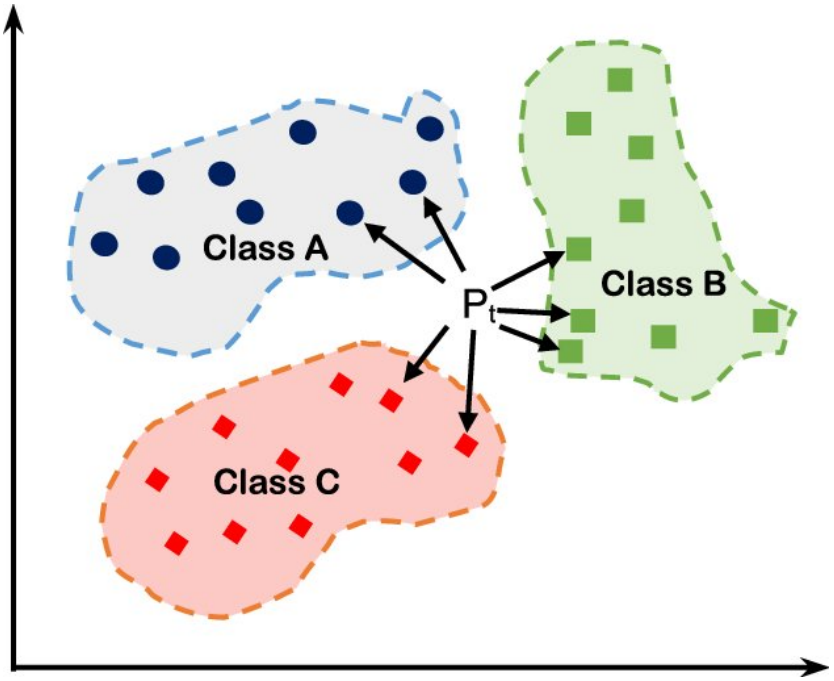
Note that the Manhattan distance, d_1, between two vectors $\mathbf{p}, \mathbf{q}$ in an n-dimensional real vector space with fixed Cartesian coordinate system is defined as

$$d_1(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|_1 = \sum_{i=1}^{n} |p_i - q_i|,$$

where $(\mathbf{p}, \mathbf{q})$ are vectors

$$\mathbf{p} = (p_1, p_2, \ldots, p_n) \text{ and } \mathbf{q} = (q_1, q_2, \ldots, q_n)\,.$$

7.4.3. Example on KNN classifiers: image from Researchgate



Our goal in this diagram is to identify a new data point with the symbol 'Pt' into one of three categories: "A," "B," or "C."

Assume that K is equal to 7. The KNN algorithm begins by computing the distance between point 'Pt' and all of the other points. The 7 closest points with the shortest distance to point 'Pt' are then found. This is depicted in the diagram below. Arrows have been used to denote the seven closest points.

The KNN algorithm's final step is to assign a new point to the class that contains the majority of the seven closest points. Three of the seven closest points belong to the class "B," while two of the seven belongs to the classes "A" and "C". Therefore the new data point will be classified as "B".

7.4.4. Dataset: https://raw.githubusercontent.com/susanli2016/Machine-Learning-with-Python/master/fruit_data_with_colors.txt

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

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

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/var/folders/kl/h_r05n_j76n32kt0dwy7kynw0000gn/T/ipykernel_5515/278679566.py in <module>
      3 from sklearn.model_selection import train_test_split
      4
----> 5 import seaborn as sns
      6 import matplotlib.pyplot as plt
      7 get_ipython().run_line_magic('matplotlib', 'inline')

ModuleNotFoundError: No module named 'seaborn'
```

```
from plotnine import *
```

```
import warnings

warnings.filterwarnings( "ignore", module = "matplotlib\..*" )
```

We will be using the fruit_data_with_colors dataset, avalable here at github page, https://raw.githubusercontent.com/susanli2016/Machine-Learning-with-Python/master/fruit_data_with_colors.txt.

The mass, height, and width of a variety of oranges, lemons, and apples are included in the file. The heights were taken along the fruit's core. The widths were measured perpendicular to the height at their widest point.

7.4.4.1. Our goals

To predict the appropriate fruit label, we'll use the mass, width, and height of the fruit as our feature points (target value).

```
url = 'https://raw.githubusercontent.com/susanli2016/Machine-Learning-with-Python/master/fruit_data_with_colors.txt'

df = pd.read_table(url)
```

```
df.head()
```

	fruit_label	fruit_name	fruit_subtype	mass	width	height	color_score
0	1	apple	granny_smith	192	8.4	7.3	0.55
1	1	apple	granny_smith	180	8.0	6.8	0.59
2	1	apple	granny_smith	176	7.4	7.2	0.60
3	2	mandarin	mandarin	86	6.2	4.7	0.80
4	2	mandarin	mandarin	84	6.0	4.6	0.79

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 59 entries, 0 to 58
Data columns (total 7 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   fruit_label    59 non-null     int64
 1   fruit_name     59 non-null     object
 2   fruit_subtype  59 non-null     object
 3   mass           59 non-null     int64
 4   width          59 non-null     float64
 5   height         59 non-null     float64
 6   color_score    59 non-null     float64
dtypes: float64(3), int64(2), object(2)
memory usage: 3.4+ KB
```

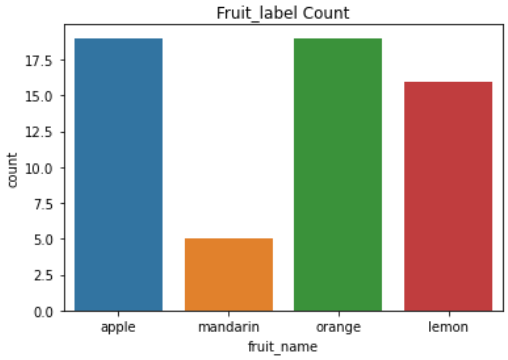
```
df.groupby('fruit_label').count()['fruit_name']
```

```
fruit_label
1    19
2     5
3    19
4    16
Name: fruit_name, dtype: int64
```

```
plt.title('Fruit_label Count')
sns.countplot(df['fruit_name'])
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages/seaborn/_decorators.py:43:
FutureWarning: Pass the following variable as a keyword arg: x. From version 0.12, the
only valid positional argument will be `data`, and passing other arguments without an
explicit keyword will result in an error or misinterpretation.
```

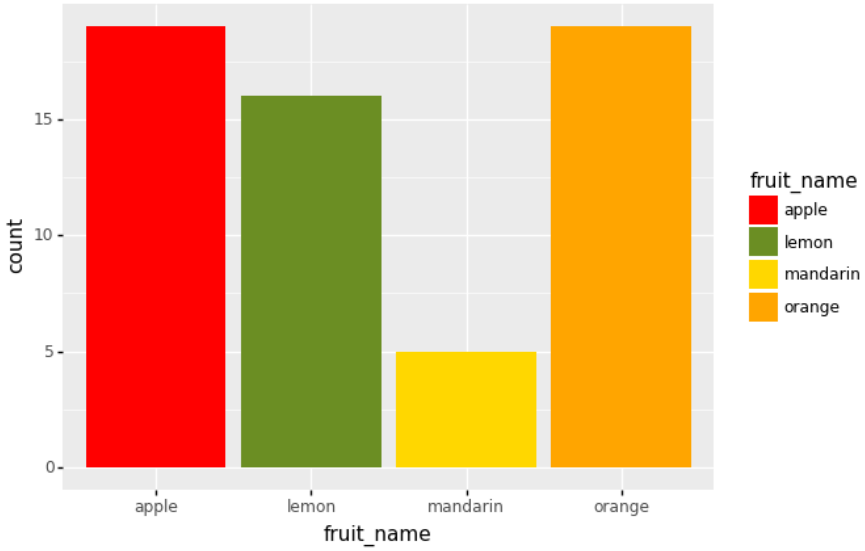
```
<AxesSubplot:title={'center':'Fruit_label Count'}, xlabel='fruit_name', ylabel='count'>
```




```
df.head()
```

	fruit_label	fruit_name	fruit_subtype	mass	width	height	color_score
0	1	apple	granny_smith	192	8.4	7.3	0.55
1	1	apple	granny_smith	180	8.0	6.8	0.59
2	1	apple	granny_smith	176	7.4	7.2	0.60
3	2	mandarin	mandarin	86	6.2	4.7	0.80
4	2	mandarin	mandarin	84	6.0	4.6	0.79

```
(
    ggplot(df)
    + aes('fruit_name',fill = 'fruit_name')
    + geom_bar()
    + scale_fill_manual(values=['red', 'olivedrab', 'gold', 'orange'])

)
```




```
<ggplot: (305317857)>
```

```
# Check whether there are any missing values.

df.isnull().sum()
```

```
fruit_label      0
fruit_name       0
fruit_subtype    0
mass             0
width            0
height           0
color_score      0
dtype: int64
```

```
print(df.fruit_label.unique())
print(df.fruit_name.unique())
```

```
[1 2 3 4]
['apple' 'mandarin' 'orange' 'lemon']
```

To make the results easier to understand, we first establish a mapping from fruit label value to fruit name.

```
# create a mapping from a mapping from fruit label value to fruit name
lookup_fruit_name = dict(zip(df.fruit_label.unique(), df.fruit_name.unique()))
lookup_fruit_name
```

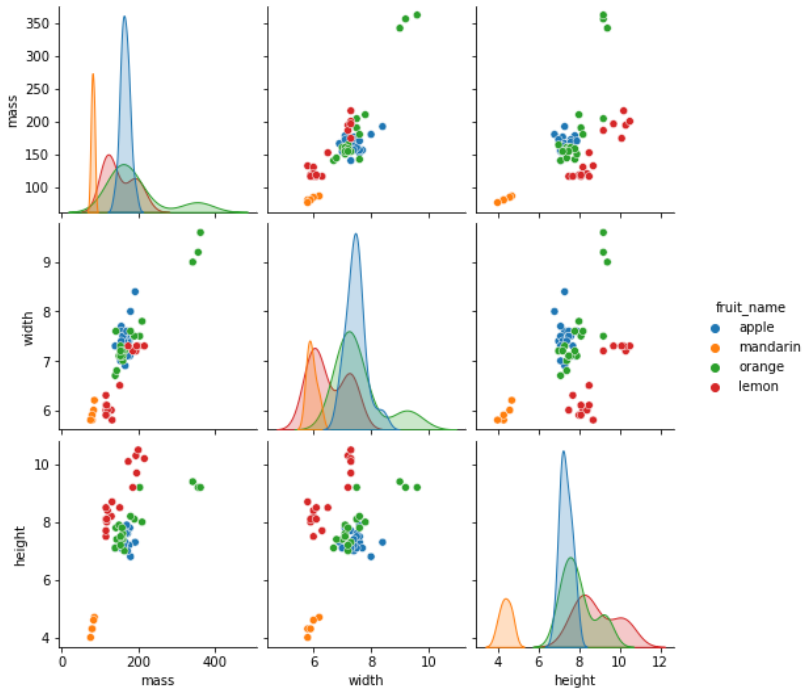
```
{1: 'apple', 2: 'mandarin', 3: 'orange', 4: 'lemon'}
```

7.4.4.2. Exploratory data analysis

It is now time to experiment with the data and make some visualizations.

```
sns.pairplot(df[['fruit_name','mass','width','height']],hue='fruit_name')
```

```
<seaborn.axisgrid.PairGrid at 0x1232cd210>
```



7.4.4.3. What we observe from the figures?

1. Mandarin has both lower mass and height. It also has the lower average widths.
2. Orange has higher average masses and widths.
3. There is a **clear separation** of lemon from the other both width-height plot and height-mass plot.
4. What else can we observe from the figures?

```
df.groupby('fruit_name').describe()[['mass']]
```

	mass							
	count	mean	std	min	25%	50%	75%	max
fruit_name								
apple	19.0	165.052632	11.969747	140.0	156.0	164.0	172.0	192.0
lemon	16.0	150.000000	37.487776	116.0	117.5	131.0	188.0	216.0
mandarin	5.0	81.200000	3.898718	76.0	80.0	80.0	84.0	86.0
orange	19.0	193.789474	73.635422	140.0	154.0	160.0	197.0	362.0

```
df.groupby('fruit_name').describe()[['height']]
```

	height							
	count	mean	std	min	25%	50%	75%	max
fruit_name								
apple	19.0	7.342105	0.291196	6.8	7.1	7.3	7.55	7.9
lemon	16.0	8.856250	0.997977	7.5	8.1	8.5	9.80	10.5
mandarin	5.0	4.380000	0.277489	4.0	4.3	4.3	4.60	4.7
orange	19.0	7.936842	0.769712	7.0	7.4	7.8	8.15	9.4

```
df.groupby('fruit_name').describe()[['width']]
```

	width							
	count	mean	std	min	25%	50%	75%	max
fruit_name								
apple	19.0	7.457895	0.345311	6.9	7.3	7.4	7.600	8.4
lemon	16.0	6.512500	0.624900	5.8	6.0	6.2	7.225	7.3
mandarin	5.0	5.940000	0.167332	5.8	5.8	5.9	6.000	6.2
orange	19.0	7.557895	0.813986	6.7	7.1	7.2	7.600	9.6

7.4.4.4. Preprocessing: Train Test Split.

Because training and testing on the same data is inefficient, we partition the data into two sets: **training and testing**.

To split the data, we use the `train_test_split` function.

The split percentage is determined by the optional parameter `test_size.` The default values are 75/25% train and test data.

The `random state` parameter ensures that the data is split in the same way each time the program is executed.

Because we are training and testing on distinct sets of data, the testing accuracy will be a better indication of how well the model will perform on new data.

```
# Train Test Split

X = df[['height', 'width', 'mass']]
y = df['fruit_label']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=0)
```

```
#shape of train and test data
print(X_train.shape)
print(X_test.shape)
```

```
(44, 3)
(15, 3)
```

```
#shape of new y objects
print(y_train.shape)
print(y_test.shape)
```

```
(44,)
(15,)
```

7.4.4.5. Training and Predictions

Scikit-learn is divided into modules so that we may quickly import the classes we need.

Import the `KNeighborsClassifer` class from the `neighbors` module.

Instantiate the estimator (a model in scikit-learn is referred to as a **estimator**). Because their major function is to estimate unknown quantities, we refer to the model as an estimator.

In our example, we have generated an instance (`knn`) of the class `KNeighborsClassifer`, which means we have constructed an object called 'knn' that knows how to perform KNN classification once the data is provided.

The **tuning parameter/hyper parameter** (k) is the parameter `n_ neighbors`. All other parameters are set to default.

The `fit` method is used to train the model using training data (X train,y train), while the `predict` method is used to test the model using testing data (X test).

In this example, we take `n_neighbors` or k = 5.

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors = 5)
```

7.4.4.6. Train the classifier (fit the estimator) using the training data¶

We then train the classifier by passing the training set data in **X_train** and the labels in **y_train** to the classifier's fit method.

```
knn.fit(X_train,y_train)
```

```
KNeighborsClassifier()
```

7.4.4.7. Estimate the accuracy of the classifier on future data, using the test data

Remember that the KNN classifier has not seen any of the fruits in the **test set** during the training phase.

To do this, we use the score method for the classifier object. This takes the points in the test set as input and calculates the **accuracy**.

The **accuracy** is defined as the proportion of points in the test set whose true label was correctly predicted by the classifier.

```
knn.score(X_test, y_test)

print('Accuracy:', knn.score(X_test, y_test))
```

```
Accuracy: 0.5333333333333333
```

We obtain a classficiation rate of 53.3%, considered as good accuracy.

Can we further improve the accuracy of the KNN algorithm?

7.4.4.8. Use the trained KNN classifier model to classify new, previously unseen objects

So, here for example. We are entering the mass, width, and height for a hypothetical piece of fruit that is pretty small.

And if we ask the classifier to predict the label using the predict method.

We can see that the output says that it is a mandarin.

for example: a small fruit with a mass of 20 g, a width of 4.5 cm and a height of 5.2 cm

```
X_train.columns
```

```
Index(['height', 'width', 'mass'], dtype='object')
```

```
#sample1 = pd.DataFrame({'height':[5.2], 'width':[4.5],'mass':[20]})
# Notice we use the same column as the X training data (or test sate)
sample1 = pd.DataFrame([[5.2,4.5,20]],columns = X_train.columns)
fruit_prediction = knn.predict(sample1)

print('The prediction is:', lookup_fruit_name[fruit_prediction[0]])
```

```
The prediction is: mandarin
```

Here another example

```
#sample2 = pd.DataFrame({'height':[6.8], 'width':[8.5],'mass':[180]})

sample2 = pd.DataFrame([[6.8,8.5,180]],columns = X_train.columns)
fruit_prediction = knn.predict(sample2)

print('The prediction is:', lookup_fruit_name[fruit_prediction[0]])
```

```
The prediction is: apple
```

Exercise

1. Create a DataFrame comparing the y_test and the predictions from the model.
2. Confirm that the accuracy is the same as obtained by knn.score(X_test, y_test).

Here is the list of predictions of the test set obtained from the model.

```
#fruit_prediction = knn.predict([[20,4.5,5.2]])
y_pred = knn.predict(X_test)
```

7.4.4.9. Calculating the agorithm accuracy

```
# the accuracy of the test set

# https://stackoverflow.com/questions/59072143/pandas-mean-of-boolean
((knn.predict(X_test) == y_test.to_numpy())*1).mean()
(knn.predict(X_test) == y_test.to_numpy()).mean()
```

```
0.5333333333333333
```

```
# the accuracy of the original data set

# (knn.predict(X) == y.to_numpy()).mean()
```

```
# the accuracy of the training set

(knn.predict(X_train) == y_train.to_numpy()).mean()
```

```
0.7954545454545454
```

7.4.4.10. Comparison of the observed and predicted values from both training and test data sets

```
# Printing out the observed and predicted values of the test set

for i in range(len(X_test)):
    print('Observed: ', lookup_fruit_name[y_test.iloc[i]], 'vs Predicted:',
lookup_fruit_name[knn.predict(X_test)[i]])
```

```
Observed:  orange vs Predicted: orange
Observed:  orange vs Predicted: apple
Observed:  lemon vs Predicted: lemon
Observed:  orange vs Predicted: lemon
Observed:  apple vs Predicted: apple
Observed:  apple vs Predicted: apple
Observed:  orange vs Predicted: orange
Observed:  lemon vs Predicted: orange
Observed:  orange vs Predicted: apple
Observed:  apple vs Predicted: lemon
Observed:  mandarin vs Predicted: mandarin
Observed:  apple vs Predicted: apple
Observed:  orange vs Predicted: orange
Observed:  orange vs Predicted: apple
Observed:  orange vs Predicted: lemon
```



```
print(X_test.index)
y_test.index
```

```
Int64Index([26, 35, 43, 28, 11, 2, 34, 46, 40, 22, 4, 10, 30, 41, 33], dtype='int64')
```


```
Int64Index([26, 35, 43, 28, 11, 2, 34, 46, 40, 22, 4, 10, 30, 41, 33], dtype='int64')
```


```
pd.DataFrame({'observed': y_test
,'predicted':knn.predict(X_test)}).set_index(X_test.index)
```

	observed	predicted
26	3	3
35	3	1
43	4	4
28	3	4
11	1	1
2	1	1
34	3	3
46	4	3
40	3	1
22	1	4
4	2	2
10	1	1
30	3	3
41	3	1
33	3	4

```
pd.DataFrame({'observed': y_train
,'predicted':knn.predict(X_train)}).set_index(X_train.index).head()
```

	observed	predicted
42	3	1
48	4	1
7	2	2
14	1	1
32	3	1

```
pd.DataFrame({'observed': y ,'predicted':knn.predict(X)}).set_index(X.index).head()
```

	observed	predicted
0	1	4
1	1	1
2	1	1
3	2	2
4	2	2

```
df_result = df
df_result['predicted_label'] = knn.predict(X)
df_result.head()
```

	fruit_label	fruit_name	fruit_subtype	mass	width	height	color_score	predicted_label
0	1	apple	granny_smith	192	8.4	7.3	0.55	4
1	1	apple	granny_smith	180	8.0	6.8	0.59	1
2	1	apple	granny_smith	176	7.4	7.2	0.60	1
3	2	mandarin	mandarin	86	6.2	4.7	0.80	2
4	2	mandarin	mandarin	84	6.0	4.6	0.79	2

7.4.4.11. Visualize the decision regions of a classifier

After a classifier has been trained on training data, a classification model is created. What criteria does your machine learning classifier consider when deciding which class a sample belongs to? Plotting a decision region can provide some insight into the decision made by your ML classifier.

A **decision region** is a region in which a classifier predicts the same class label for data.

The boundary between areas of various classes is known as the **decision boundary**.

The plot decision regions function in `mlxtend` is a simple way to plot decision areas. We can also use mlxtend to plot decision regions of **Logistic Regression, Random Forest, RBF kernel SVM, and Ensemble classifier**.

Important Note for the 2D scatterplot, we can **only visualize 2 features** at a time. So, if you have a 3-dimensional (or more than 3) dataset, it will essentially be a **2D slice through this feature space with fixed values for the remaining feature(s)**.

```
from mlxtend.plotting import plot_decision_regions
```

```
# Importing a dataset
url = 'https://raw.githubusercontent.com/susanli2016/Machine-Learning-with-
Python/master/fruit_data_with_colors.txt'
df = pd.read_table(url)

# Train Test Split

X = df[['height', 'width', 'mass']]
y = df['fruit_label']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=0)

# Instantiate the estimator
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 5)

# Training the classifier by passing in the training set X_train and the labels in
y_train
knn.fit(X_train,y_train)

# Predicting labels for unknown data
y_pred = knn.predict(X_test)
```

```
X_train.describe()
```

	height	width	mass
count	44.000000	44.000000	44.000000
mean	7.643182	7.038636	159.090909
std	1.370350	0.835886	53.316876
min	4.000000	5.800000	76.000000
25%	7.200000	6.175000	127.500000
50%	7.600000	7.200000	157.000000
75%	8.250000	7.500000	172.500000
max	10.500000	9.200000	356.000000

7.4.4.12. Decision regions for two features, height and mass.

We may want to go beyond the numerical prediction (of the class or of the probability) and visualize the actual decision boundaries between the classes for many classification problems in the domain of supervised ML.

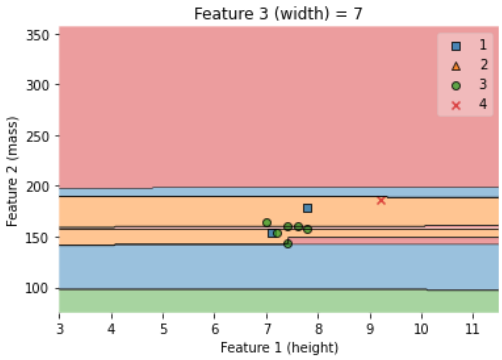
The visualization is displayed on a **2-dimensional (2D) plane**, which makes it particularly ideal for **binary classification** tasks and for **a pair of features**.

We will be using the `plot_decision_regions()` function in the `MLxtend` library to draw a classifier's decision regions.

```
knn = KNeighborsClassifier(n_neighbors = 5)
knn.fit(X_train.values, y_train.values)

# Plotting decision regions
fig, ax = plt.subplots()

# Decision region for feature 2 = width = value
value=7

# Plot training sample points with
# feature 2 = width = value +/- width
width=0.3

fig = plot_decision_regions(X_train.values, y_train.values, clf=knn,
              feature_index=[0,2],   #these one will be plotted
              filler_feature_values={1: value},  #these will be ignored
              filler_feature_ranges={1: width})
ax.set_xlabel('Feature 1 (height)')
ax.set_ylabel('Feature 2 (mass)')
ax.set_title('Feature 3 (width) = {}'.format(value))
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/mlxtend/plotting/decision_regions.py:279: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
```

```
Text(0.5, 1.0, 'Feature 3 (width) = 7')
```



Important Notes: the arguments used in the `plot_decision_regions` are explained below:

1. **feature_index**: feature indices to use for plotting.

This arguement defines the indices to be displayed in the 2D plot of the decision boundary. The first index in feature_index will be on the x-axis, the second index will be on the y-axis. (for array-like (default: (0,) for 1D, (0, 1) otherwise))

1. **filler_feature_values**:

This arguement defines **the indices and their (fixed) values** of the features that are not included in the 2D plot of the decision boundary (index-value pairs for the features that are not displayed). Required only for Number Features > 2.

The function 'plot_decision_regions' fits the data and makes a prediction to create the appropriate decision boundary by finding the predicted value for each point (the values of the ignored features are **fixed** and equal to the values specified by the user) in a grid-like scatter plot.

1. **filler_feature_ranges**:

The last argument we included for the 'plot_decision_regions' function is 'filler_feature_ranges'. This arguement defines the ranges of features that will not be plotted, and these regions are used to select (training) sample points for plotting.

The following Python gives sample points that are plotted in the above decision region, i.e. the range of the widths is in this interval (value - width, value + width).

```
# Points to be included in the plot above,
# i.e. those sample points with width between (value - width, value + width)

# X_train

X_train.head()
#X_train.head()
print(X_train[(X_train.width < value + width) & (X_train.width > value -
width)].sort_values(by = 'height'))
#print(y_train[(X_train.width < value + width) & (X_train.width > value - width)])
```

```
    height  width  mass
32     7.0    7.2   164
12     7.1    7.0   154
42     7.2    7.2   154
29     7.4    7.0   160
39     7.4    6.8   144
36     7.6    7.1   160
8      7.8    7.1   178
38     7.8    7.2   158
45     9.2    7.2   186
```

```
#df.head()
#X_train.head()
#df[(df.width < 7.2) & (df.width > 6.8)].sort_values(by = 'height')
```

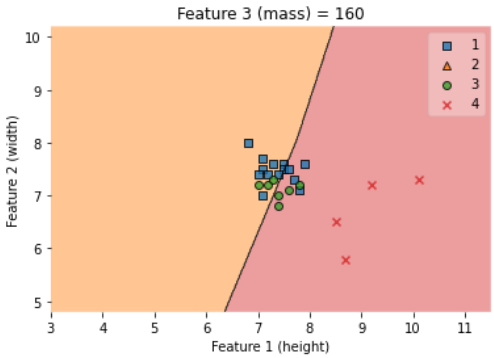
7.4.4.13. Decision regions for two features, height and width.

```
#X_train.describe()
```

```
knn = KNeighborsClassifier(n_neighbors = 5)
knn.fit(X_train.values, y_train.values)

# Plotting decision regions
fig, ax = plt.subplots()

# Decision region for feature 3 = mass = value
value=160
# Plot training sample with feature 3 = mass = value +/- width
width=30

fig = plot_decision_regions(X_train.values, y_train.values, clf=knn,
              feature_index=[0,1],               #these one will be plotted
              filler_feature_values={2: value},  #these will be ignored
              filler_feature_ranges={2: width})
ax.set_xlabel('Feature 1 (height)')
ax.set_ylabel('Feature 2 (width)')
ax.set_title('Feature 3 (mass) = {}'.format(value))
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/mlxtend/plotting/decision_regions.py:279: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
```

```
Text(0.5, 1.0, 'Feature 3 (mass) = 160')
```



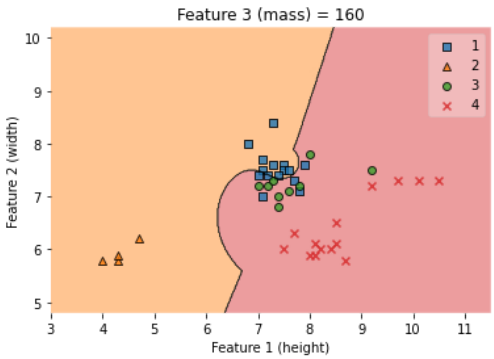
Important Note: Do not be surprised that the samples (with the cross symbol) are correctly classified. Why?

```
#print(X_train.head())
#print(y_train)
```

```
##### Convert pandas DataFrame to Numpy before applying classification
#####
#### weights{'uniform', 'distance'}

#### 'uniform' : uniform weights. All points in each neighborhood are weighted equally.

#### 'distance' : weight points by the inverse of their distance. in this case, closer
neighbors of a query point will have a greater influence than neighbors which are
further away.


X_train_np = X_train.to_numpy()
y_train_np = y_train.to_numpy()


clf = KNeighborsClassifier(n_neighbors = 5,weights='distance')
clf.fit(X_train_np, y_train_np)

# Plotting decision regions
fig, ax = plt.subplots()

# Decision region for feature 3 = mass = value
value=160
# Plot training sample with feature = mass = value +/- width
width=100

fig = plot_decision_regions(X_train_np, y_train_np, clf=clf,
              feature_index=[0,1],               #these one will be plotted
              filler_feature_values={2: value},  #these will be ignored
              filler_feature_ranges={2: width})
ax.set_xlabel('Feature 1 (height)')
ax.set_ylabel('Feature 2 (width)')
ax.set_title('Feature 3 (mass) = {}'.format(value))
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/mlxtend/plotting/decision_regions.py:279: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
```

```
Text(0.5, 1.0, 'Feature 3 (mass) = 160')
```



```
# clf.predict(X_train_np)
# y_train_np
# X_train.describe()
```

```
# X_train_np = X_train.iloc[:,0:2].to_numpy()
# y_train_np = y_train.to_numpy()

# X_train_np.shape
```

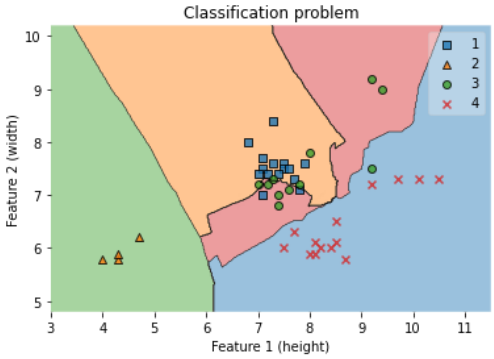
7.4.4.14. Visualization of Dicision Boundary with KNN classification for a dataset with only two features.

We will visualize the actual decision boundaries by training the KNN model with the dataset consisting of only one pair of features. We will then make a comparison the decision boundaries with the previous results.

```
##### Convert pandas DataFrame to Numpy before applying classification

X_train_np = X_train.iloc[:,0:2].to_numpy()
y_train_np = y_train.to_numpy()


clf = KNeighborsClassifier(n_neighbors = 5)
clf.fit(X_train_np, y_train_np)

# Plotting decision regions
fig, ax = plt.subplots()

# Decision region for feature 3 = mass = value
# value=160
# Plot training sample with feature = mass = value +/- width
# width=20

fig = plot_decision_regions(X_train_np, y_train_np, clf=clf)
ax.set_xlabel('Feature 1 (height)')
ax.set_ylabel('Feature 2 (width)')
ax.set_title('Classification problem')
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/mlxtend/plotting/decision_regions.py:279: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
```

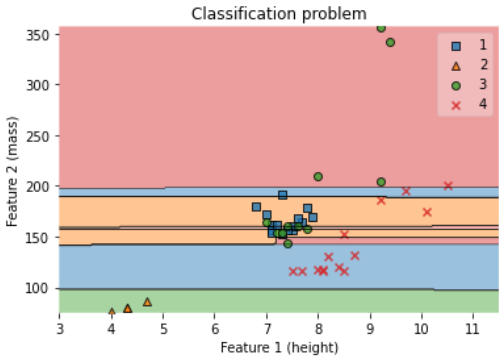
```
Text(0.5, 1.0, 'Classification problem')
```



```
##### Convert pandas DataFrame to Numpy before applying classification

X_train_np = X_train.iloc[:,[0,2]].to_numpy()
y_train_np = y_train.to_numpy()


clf = KNeighborsClassifier(n_neighbors = 5)
clf.fit(X_train_np, y_train_np)

# Plotting decision regions
fig, ax = plt.subplots()

# Decision region for feature 3 = mass = value
value=160
# Plot training sample with feature = mass = value +/- width
width=20

fig = plot_decision_regions(X_train_np, y_train_np, clf=clf)
ax.set_xlabel('Feature 1 (height)')
ax.set_ylabel('Feature 2 (mass)')
ax.set_title('Classification problem')
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/mlxtend/plotting/decision_regions.py:279: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
```

```
Text(0.5, 1.0, 'Classification problem')
```



7.4.4.15. Width-Mass visualization

```
#print(X_train.head())
#print(X_train.values)
#X_train.to_numpy()

#print(type(X_train.values))
#print(type(X_train.to_numpy()))

#print(X_train.values.shape)
#print(X_train.to_numpy().shape)
```

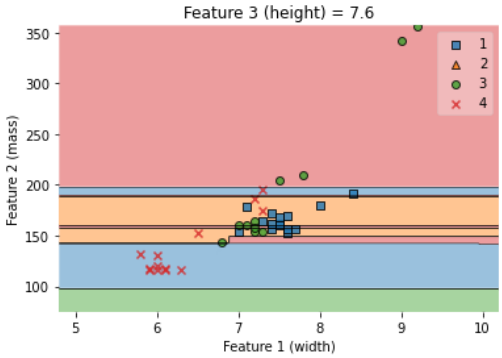
```
X_train.describe()
```

	height	width	mass
count	44.000000	44.000000	44.000000
mean	7.643182	7.038636	159.090909
std	1.370350	0.835886	53.316876
min	4.000000	5.800000	76.000000
25%	7.200000	6.175000	127.500000
50%	7.600000	7.200000	157.000000
75%	8.250000	7.500000	172.500000
max	10.500000	9.200000	356.000000

```
knn = KNeighborsClassifier(n_neighbors = 5)
knn.fit(X_train.values, y_train.values)

# Plotting decision regions
fig, ax = plt.subplots()

# Decision region for feature 1 = height = value
value=7.6
# Plot training sample with feature 1 = height = value +/- width
width=2.6

fig = plot_decision_regions(X_train.values, y_train.values, clf=knn,
              feature_index=[1,2],               #these one will be plotted
              filler_feature_values={0: value},  #these will be ignored
              filler_feature_ranges={0: width})
ax.set_xlabel('Feature 1 (width)')
ax.set_ylabel('Feature 2 (mass)')
ax.set_title('Feature 3 (height) = {}'.format(value))
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/mlxtend/plotting/decision_regions.py:279: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
```

```
Text(0.5, 1.0, 'Feature 3 (height) = 7.6')
```

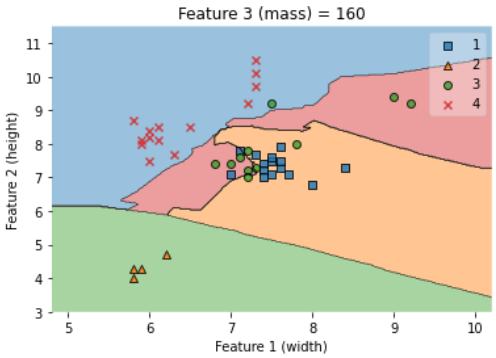


7.4.4.16. Width-height visualization

```
#### Applying KNN classfication with only two features

X_train[['width','height']].to_numpy()
y_train.to_numpy()

knn = KNeighborsClassifier(n_neighbors = 5)
knn.fit(X_train[['width','height']].to_numpy(), y_train.to_numpy())

# Plotting decision regions
fig, ax = plt.subplots()

# Decision region for feature 3 = mass = value
value=160
# Plot training sample with feature 3 = mass = value +/- width
width=50

fig = plot_decision_regions(X_train[['width','height']].to_numpy(), y_train.to_numpy(),
clf=knn)
ax.set_xlabel('Feature 1 (width)')
ax.set_ylabel('Feature 2 (height)')
ax.set_title('Feature 3 (mass) = {}'.format(value))
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/mlxtend/plotting/decision_regions.py:279: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
```

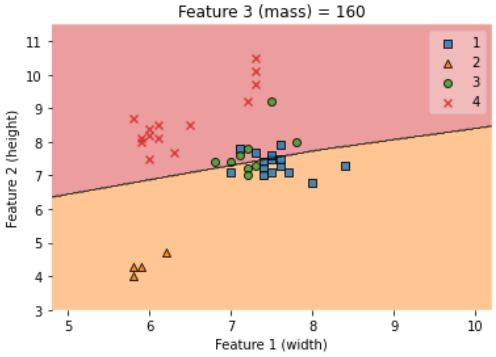
```
Text(0.5, 1.0, 'Feature 3 (mass) = 160')
```




```
knn = KNeighborsClassifier(n_neighbors = 5)
knn.fit(X_train.values, y_train.values)

# Plotting decision regions
fig, ax = plt.subplots()

# Decision region for feature 3 = mass = value
value=160
# Plot training sample with feature 3 = mass = value +/- width
width=100

fig = plot_decision_regions(X_train.values, y_train.values, clf=knn,
              feature_index=[1,0],                 #these one will be plotted
              filler_feature_values={2: value},  #these will be ignored
              filler_feature_ranges={2: width})
ax.set_xlabel('Feature 1 (width)')
ax.set_ylabel('Feature 2 (height)')
ax.set_title('Feature 3 (mass) = {}'.format(value))
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/mlxtend/plotting/decision_regions.py:279: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
```

```
Text(0.5, 1.0, 'Feature 3 (mass) = 160')
```

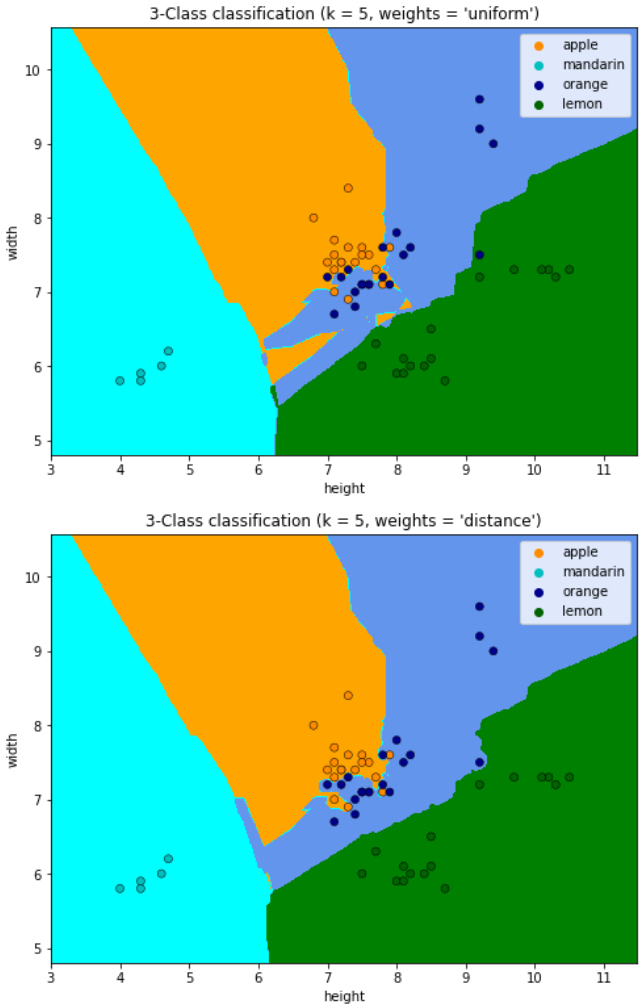


7.4.4.17. Visualize (from scratch) the decision regions of a classifier

```
from matplotlib.colors import ListedColormap
from sklearn import neighbors
```

```
#X = df[['height', 'width']].to_numpy()
#y = df['fruit_label'].to_numpy()
```

```
# The code below has been modified based on https://scikit-
learn.org/stable/auto_examples/neighbors/plot_classification.html

# we only take the first two features. We could avoid this ugly
# slicing by using a two-dim dataset
#X = iris.data[:, :2]
#y = iris.target

X = df[['height', 'width']].to_numpy()
y = df['fruit_label'].to_numpy()

n_neighbors = 5

h = 0.02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(["orange", "cyan", "cornflowerblue","green"])
cmap_bold = ["darkorange", "c", "darkblue","darkgreen"]

for weights in ["uniform", "distance"]:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(n_neighbors, weights=weights)
    clf.fit(X, y)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(8, 6))
    plt.contourf(xx, yy, Z, cmap=cmap_light)

    # Plot also the training points
    sns.scatterplot(
        x=X[:, 0],
        y=X[:, 1],
        hue= df.fruit_name.to_numpy(),
        palette=cmap_bold,
        alpha=1.0,
        edgecolor="black",
    )
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title(
        "3-Class classification (k = %i, weights = '%s')" % (n_neighbors, weights)
    )
    plt.xlabel('height')
    plt.ylabel('width')

plt.show()
```

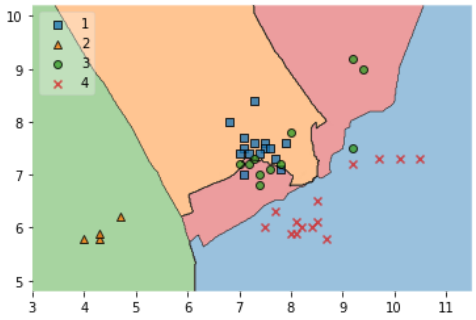



```
#https://stackoverflow.com/questions/52952310/plot-decision-regions-with-error-filler-values-must-be-provided-when-x-has-more
# You can use PCA to reduce your data multi-dimensional data to two dimensional data. Then pass the obtained result in plot_decision_region and there will be no need of filler values.

from mlxtend.plotting import plot_decision_regions
```

```
# Decision region of the training set

X = df[['height', 'width']]
y = df['fruit_label']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=0)

clf = KNeighborsClassifier(n_neighbors = 5)
clf.fit(X_train,y_train)

plot_decision_regions(X_train.to_numpy(), y_train.to_numpy(), clf=clf, legend=2)
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages/sklearn/base.py:446:
UserWarning: X does not have valid feature names, but KNeighborsClassifier was fitted
with feature names
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/mlxtend/plotting/decision_regions.py:279: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
```

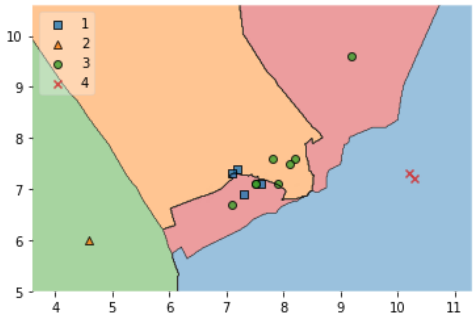
```
<AxesSubplot:>
```




```
# Decision region of the test set

plot_decision_regions(X_test.to_numpy(), y_test.to_numpy(), clf=clf, legend=2)
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages/sklearn/base.py:446:
UserWarning: X does not have valid feature names, but KNeighborsClassifier was fitted
with feature names
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/mlxtend/plotting/decision_regions.py:279: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
```

```
<AxesSubplot:>
```



7.5. Model Evaluation Metrics in Machine Learning

Machine learning has become extremely popular in recent years. Machine learning is used to infer new situations from past data, and there are far too many machine learning algorithms to choose from.

Machine learning techniques such as

- linear regression,
- logistic regression,
- decision tree,
- Naive Bayes, K-Means, and
- Random Forest

are widely used.

When it comes to predicting data, we **do not use just one algorithm**. Sometimes we use multiple algorithms and then proceed with the one that gives the best data predictions.

7.5.1. How can we figure out which algorithm is the most effective?

Model evaluation metrics allow us to evaluate the accuracy of our trained model and track its performance.

Model evaluation metrics, which distinguish adaptive from non-adaptive machine learning models, indicate how effectively the model generalizes to new data.

We could improve the overall predictive power of our model before using it for production on unknown data by using different performance evaluation metrics.

Choosing the right metric is very important when evaluating machine learning models. Machine learning models are evaluated using a variety of metrics in different applications. Let us look at the metrics for evaluating the performance of a machine learning model.

This is a critical phase in any data science project as it aims to estimate the generalization accuracy of a model for future data.

Evaluation Metrics For Regression Models: image from enjoyalgorithms

Evaluation Metrics For Regression Models from enjoyalgorithms

Evaluation Metrics For Classification Models: image from enjoyalgorithms

Evaluation Metrics For Classification Models from enjoyalgorithms

7.5.1.1. Regression Related Metrics

The most common measures for evaluating a regression model (as used in our previous chapter) are:

- **Mean Absolute Error (MAE)**: The average of the difference between the actual and anticipated values is the Mean Absolute Error. It determines how close the predictions are to the actual results. The better the model, the lower the MAE.
- **Mean Squared Error (MSE)**: The average of the square of the difference between the actual and predicted values is calculated by MSE.
- **R2 score**: The proportion of variance in Y that can be explained by X is called the R2 score.

7.5.1.2. Classification Metrics

1. **Confusion Matrix (Accuracy, Sensitivity, and Specificity)**

A confusion matrix contains the results of any binary testing that is commonly used to describe the classification model's performance.

In a binary classification task, there are only two classes to categorize, preferably a **positive class** and a **negative class**.

Let us take a look at the metrics of the confusion matrix.

- **Accuracy**: indicates the overall accuracy of the model, i.e., the percentage of all samples that were correctly identified by the classifier. Use the following formula to calculate accuracy: (TP +TN)/(TP +TN+ FP +FN).
    - True Positive (TP): This is the number of times the classifier successfully predicted the positive class to be positive.
    - True Negative (TN): The number of times the classifier correctly predicts the negative class as negative.
    - False Positive (FP): This term refers to the number of times a classifier incorrectly predicts a negative class as positive.
    - False Negative (FN): This is the number of times the classifier predicts the positive class as negative.
- **The misclassification rate**: tells you what percentage of predictions were incorrect. It is also called classification error. You can calculate it with (FP +FN)/(TP +TN+ FP +FN) or (1-accuracy).
- **Sensitivity (or Recall)**: It indicates the proportion of all positive samples that were correctly predicted to be positive by the classifier. It is also referred to as **true positive rate (TPR), sensitivity, or probability of detection**. To calculate recall, use the following formula: TP /(TP +FN).
- **Specificity**: it indicates the proportion of all negative samples that are correctly predicted to be negative by the classifier. It is also referred to as the **True Negative Rate (TNR)**. To calculate the specificity, use the following formula: TN /(TN +FP).

1. **Precision**: When there is an **imbalance between classes**, accuracy can become an unreliable metric for measuring our performance. Therefore, we also need to address class-specific performance metrics. Precision is one such metric, defined as **positive predictive values** (Proportion of

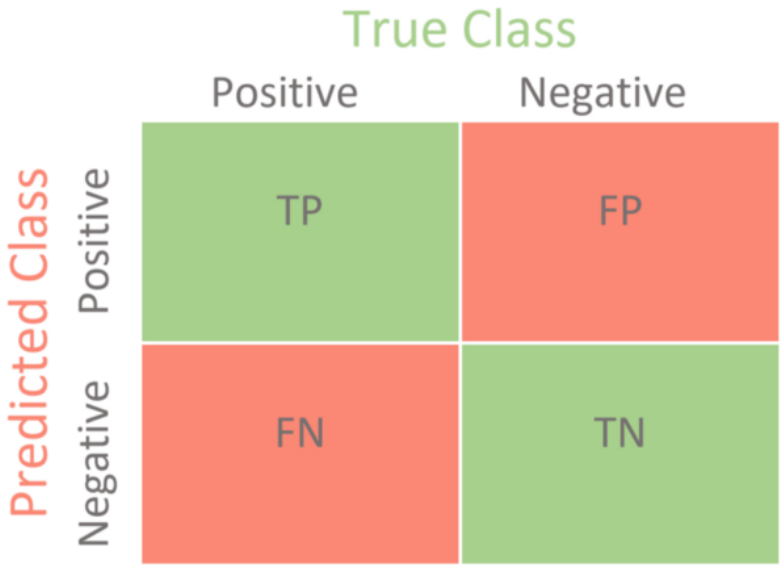
predictions as positive class were actually positive). To calculate precision, use the following formula: TP/(TP+FP).

2. **F1-score**: it combines precision and recall in a single measure. Mathematically, it is the harmonic mean of Precision and Recall. It can be calculated as follows:

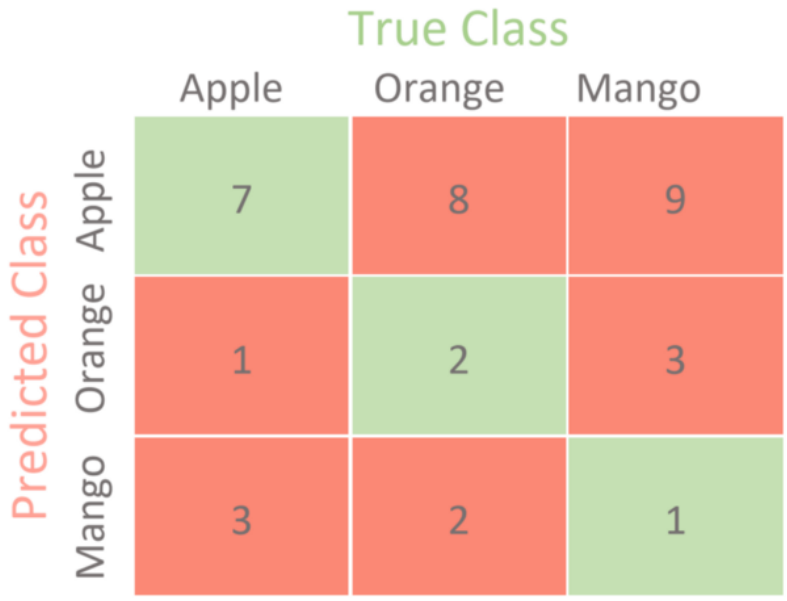
$$F_1 = \frac{2}{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}} = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}} = \frac{\mathrm{tp}}{\mathrm{tp} + \frac{1}{2}(\mathrm{fp} + \mathrm{fn})}.$$

In a perfect world, we would want a model that has a precision of 1 and a recall of 1. This means an F1 score of 1, i.e. 100% accuracy, which is often not the case for a machine learning model. So we should try to achieve a higher precision with a higher recall value.

Confusion Matrix for Binary Classification: image from https://towardsdatascience.com/



Confusion Matrix for Multi-class Classification: image from https://towardsdatascience.com/



```
# Importing a dataset
url = 'https://raw.githubusercontent.com/susanli2016/Machine-Learning-with-
Python/master/fruit_data_with_colors.txt'
df = pd.read_table(url)

# Train Test Split

X = df[['height', 'width', 'mass']]
y = df['fruit_label']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=0)

# Instantiate the estimator
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 5)

# Training the classifier by passing in the training set X_train and the labels in
y_train
knn.fit(X_train,y_train)

# Predicting labels for unknown data
y_pred = knn.predict(X_test)
```

```
# Various attributes of the knn estimator

print(knn.classes_)
print(knn.feature_names_in_)
print(knn.n_features_in_)
print(knn.n_neighbors)
```

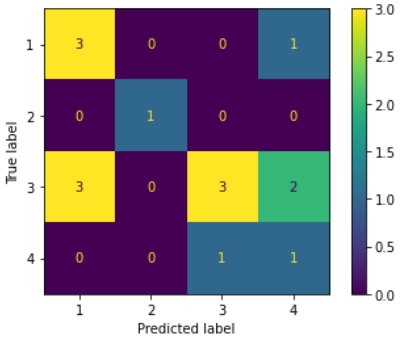
```
[1 2 3 4]
['height' 'width' 'mass']
3
5
```

```
#importing confusion matrix

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
confusion = confusion_matrix(y_test, y_pred)
print('Confusion Matrix\n')
print(confusion)
```

```
Confusion Matrix

[[3 0 0 1]
 [0 1 0 0]
 [3 0 3 2]
 [0 0 1 1]]
```

```
# Confusion Matrix visualization.

cm = confusion_matrix(y_test, y_pred, labels=knn.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,display_labels=knn.classes_)
disp.plot()
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x125e687d0>
```




```
pd.DataFrame({'observed': y_test
,'predicted':knn.predict(X_test)}).set_index(X_test.index).sort_values(by = 'observed')
```

	observed	predicted
11	1	1
2	1	1
22	1	4
10	1	1
4	2	2
26	3	3
35	3	1
28	3	4
34	3	3
40	3	1
30	3	3
41	3	1
33	3	4
43	4	4
46	4	3

Unlike the binary classification, there are no positive or negative classes here.

At first glance, it might be a little difficult to find TP, TN, FP, and FN since there are no positive or negative classes, but it's actually pretty simple.

What we need to do here is find TP, TN, FP and FN for each and every class. For example, let us take the **Apple class**. Let us look at what values the metrics have in the confusion matrix. (DO NOT FORGET TO TRANSPOSE)

- TP = 3
- TN = (1 + 3 + 2 + 1 + 1) = 8 (the sum of the numbers in rows 2-4 and columns 2-4)
- FP = (0 + 3 + 0) = 3
- FN = (0 + 0 + 1) = 1

Now that we have all the necessary metrics for the Apple class from the confusion matrix, we can calculate the performance metrics for the Apple class. For example, the class Apple has

- Precision = 3/(3+3) = 0.5
- Recall = 3/(3+1) = 0.75
- F1-score = 0.60

In a similar way, we can calculate the measures for the other classes. Here is a table showing the values of each measure for each class.

```
from sklearn.metrics import classification_report
print('\nClassification Report\n')
print(classification_report(y_test, y_pred, target_names=['Class 1', 'Class 2', 'Class
3', 'Class 4']))
```

```
Classification Report

              precision    recall  f1-score   support

     Class 1       0.50      0.75      0.60         4
     Class 2       1.00      1.00      1.00         1
     Class 3       0.75      0.38      0.50         8
     Class 4       0.25      0.50      0.33         2

    accuracy                           0.53        15
   macro avg       0.62      0.66      0.61        15
weighted avg       0.63      0.53      0.54        15
```

Now we can do more with these measures. We can combine the F1 score of each class to get a single measure for the entire model. There are several ways to do this, which we will now look at.

- **Macro F1** This is the macro-averaged F1 score. It calculates the metrics for each class separately and then takes the unweighted average of the measures. As we saw in the figure "Precision, recall and F1 score for each class",
- **Weighted F1** The final value is the weighted mean F1 score. Unlike Macro F1, this uses a weighted mean of the measures. The weights for each class are the total number of samples in that class. Since we had 4 apples, 1 mandarin, 8 oranges, and 3 lemons,

We obtain a classficiation rate of 53.3%, considered as good accuracy.

Can we further improve the accuracy of the KNN algorithm?

In our example, we have created an instance ('knn') of the class 'KNeighborsClassifer,' which means we have constructed an object called 'knn' that knows how to perform KNN classification once the data is provided.

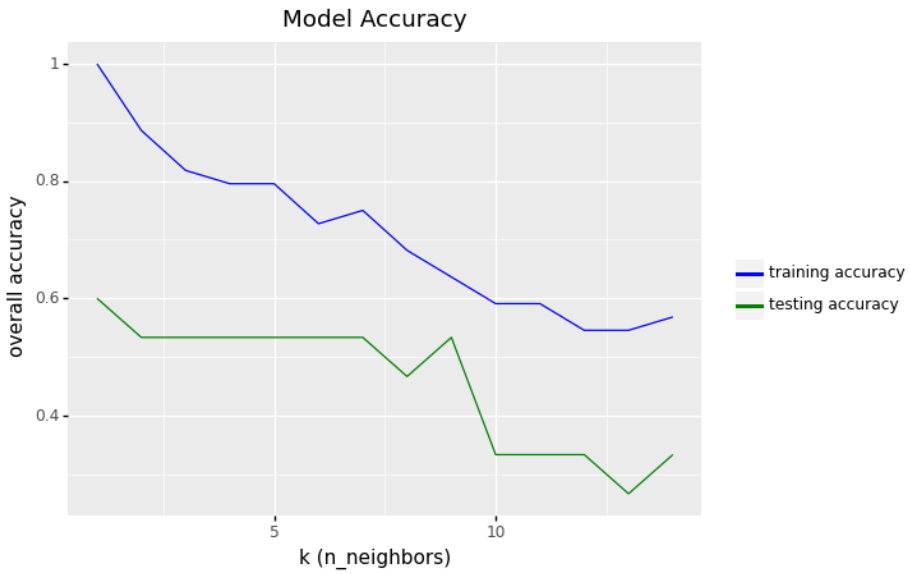
The tuning parameter/hyper parameter (K) is the parameter **n_neighbors**. All other parameters are set to default values.

Exercises

1. Fit the model and test it for different values for K (from 1 to 5) using a for loop and record the KNN's testing accuracy of the KNN in a variable.
2. Plot the relationship between the values of K and the corresponding testing accuracy.
3. Select the optimal value of K that gives the highest testing accuray.
4. Compare the results between the optimal value of K and K = 5.

```
# Importing a dataset
url = 'https://raw.githubusercontent.com/susanli2016/Machine-Learning-with-
Python/master/fruit_data_with_colors.txt'
df = pd.read_table(url)

# Train Test Split

X = df[['height', 'width', 'mass']]
y = df['fruit_label']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=0)

# Instantiate the estimator
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()

k_range = range(1,15)

train_accuracy = {}
train_accuracy_list = []

test_accuracy = {}
test_accuracy_list = []

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors = k)

    # Training the classifier by passing in the training set X_train and the labels in
y_train
    knn.fit(X_train,y_train)

    # Compute accuracy on the training set
    train_accuracy[k] = knn.score(X_train, y_train)
    train_accuracy_list.append(knn.score(X_train, y_train))


    test_accuracy[k] = knn.score(X_test,y_test)
    test_accuracy_list.append(knn.score(X_test,y_test))
```

```
df_output = pd.DataFrame({'k':k_range,
                          'train_accuracy':train_accuracy_list,
                          'test_accuracy':test_accuracy_list
                         })

(
    ggplot(df_output)
    + geom_line(aes(x = 'k',  y = 'train_accuracy',color='"training accuracy"'))
    + geom_line(aes(x = 'k',  y = 'test_accuracy',color='"testing accuracy"'))
    + labs(x='k (n_neighbors)', y='overall accuracy', title = 'Model Accuracy')
    + scale_color_manual(values = ["blue", "green"], # Colors
        name = " ")
)
```




```
<ggplot: (306375789)>
```

```
df_output.sort_values(by = 'test_accuracy',  ascending=False).head()
```

	k	train_accuracy	test_accuracy
0	1	1.000000	0.600000
1	2	0.886364	0.533333
2	3	0.818182	0.533333
3	4	0.795455	0.533333
4	5	0.795455	0.533333

```
row_max = df_output.test_accuracy.idxmax()

print(f'Best accuracy was {df_output.test_accuracy[row_max]}, which corresponds to a
value of K={df_output.k[row_max]}')
```

```
Best accuracy was 0.6, which corresponds to a value of K=1
```

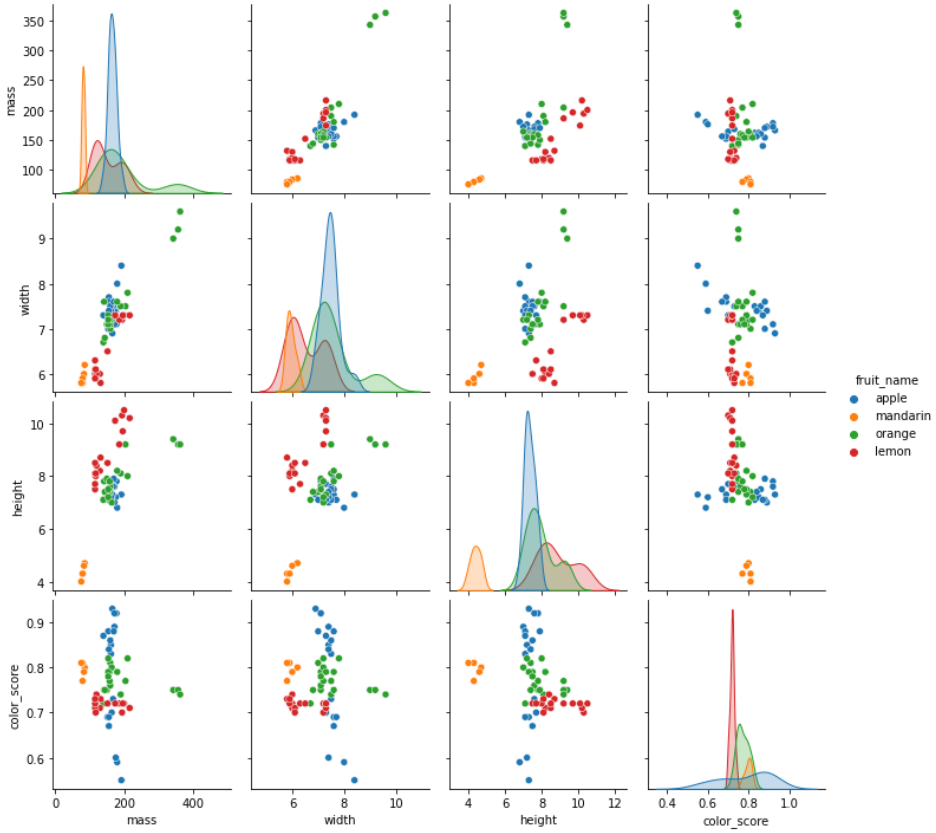
The overall accuracy of 0.6 shows that the model does not perform well in the predictions for the test data set.

```
#y_test.shape
#np.sqrt(15)
```

```
sns.pairplot(df[['fruit_name','mass','width','height','color_score']],hue='fruit_name')
```



```
<seaborn.axisgrid.PairGrid at 0x124397c50>
```



7.5.1.3. KNN model with **color_score** and other feature(s)

We will see if we can improve the accuracy of our model by including the feature **color_score**.

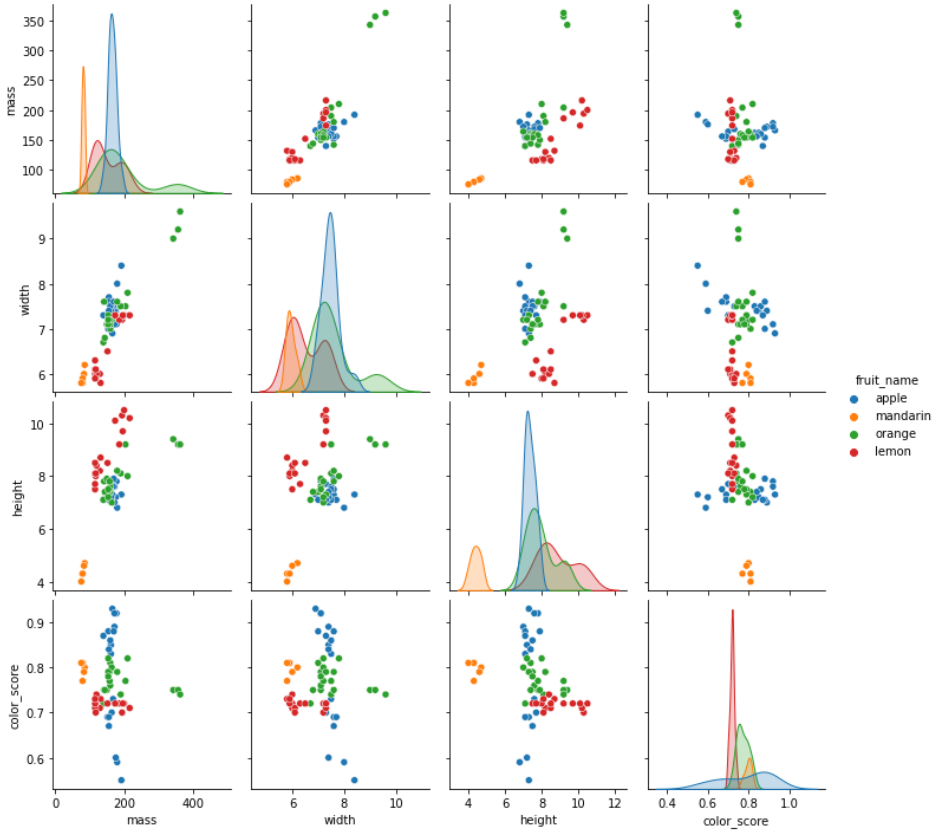
As we can see from the pair plots and correlation matrix below,

- there is a clear nonlinear separation of the 4 fruit types when we consider **color_score**.
- Also, we see that both mass and width have a positive correlation with height.

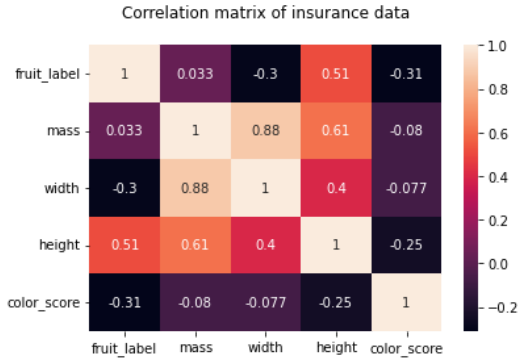
Therefore, we may omit the features **weight** and **mass** and train the KNN model with the two features **height** and **color_score** instead.

```
sns.pairplot(df[['fruit_name','mass','width','height','color_score']],hue='fruit_name')
```

```
<seaborn.axisgrid.PairGrid at 0x124c38810>
```



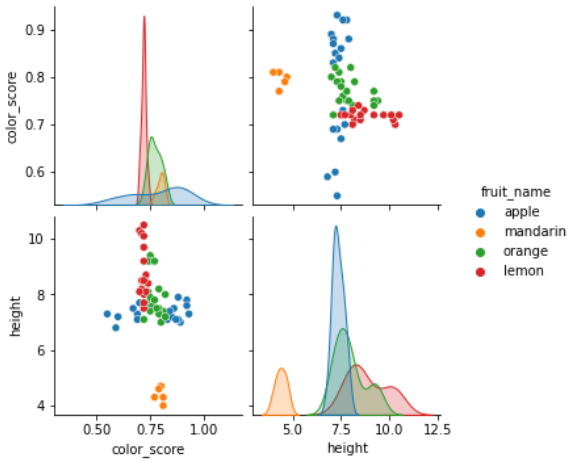

```
import seaborn as sns
import matplotlib.pyplot as plt

hm = sns.heatmap(df.corr(), annot = True)
hm.set(title = "Correlation matrix of insurance data\n")
plt.show()
```



Let us make some visualizations with pair plots of the two features **height** and **color_score** instead.

```
sns.pairplot(df[['fruit_name','color_score','height']],hue='fruit_name')
```

```
<seaborn.axisgrid.PairGrid at 0x126ef57d0>
```

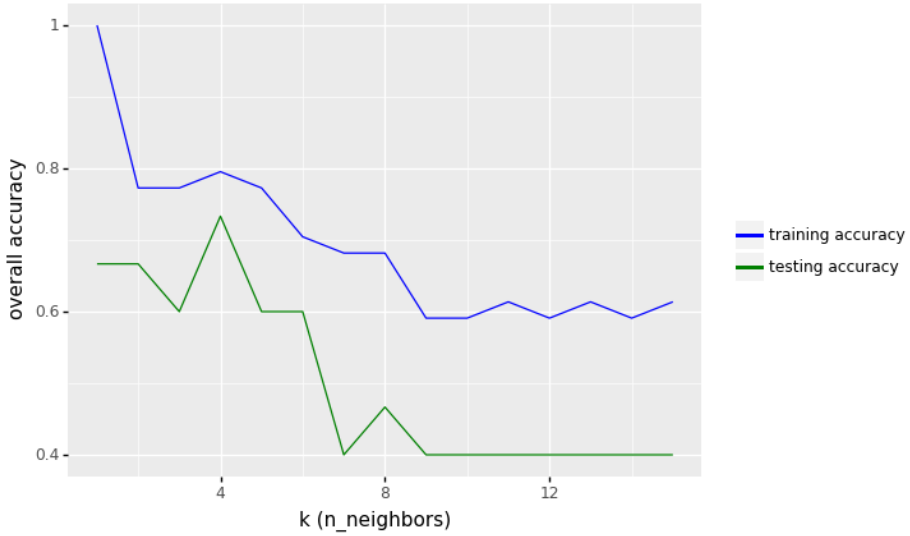


We will repeat the same process as before:

- Spliting the data into training and test sets,
- Fitting the model on the training dataset,
- Making predictions on new dataset (test set), and
- Evaluating the predictive performances on the test set

```
# Importing a dataset
url = 'https://raw.githubusercontent.com/susanli2016/Machine-Learning-with-
Python/master/fruit_data_with_colors.txt'
df = pd.read_table(url)

# Train Test Split

X = df[['height', 'color_score']]
y = df['fruit_label']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=0)

# Instantiate the estimator
from sklearn.neighbors import KNeighborsClassifier
#knn = KNeighborsClassifier()

k_range = range(1,16)

train_accuracy = {}
train_accuracy_list = []

test_accuracy = {}
test_accuracy_list = []

# misclassfication error
train_error_list = []
test_error_list = []

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors = k)

    # Training the classifier by passing in the training set X_train and the labels in
y_train
    knn.fit(X_train,y_train)

    # Compute accuracy on the training set
    train_accuracy[k] = knn.score(X_train, y_train)
    train_accuracy_list.append(knn.score(X_train, y_train))
    train_error_list.append(1 - knn.score(X_train, y_train))

    test_accuracy[k] = knn.score(X_test,y_test)
    test_accuracy_list.append(knn.score(X_test,y_test))
    test_error_list.append(1 - knn.score(X_test,y_test))

df_output = pd.DataFrame({'k':k_range,
                          'train_accuracy':train_accuracy_list,
                          'test_accuracy':test_accuracy_list,
                          'train_error':train_error_list,
                          'test_error':test_error_list
                         })
```

```
# Accuracy over the number of K neighbors

(
    ggplot(df_output)
    + geom_line(aes(x = 'k',  y = 'train_accuracy',color='"training accuracy"'))
    + geom_line(aes(x = 'k',  y = 'test_accuracy',color='"testing accuracy"'))
    + labs(x='k (n_neighbors)', y='overall accuracy')
    + scale_color_manual(values = ["blue", "green"], # Colors
        name = " ")
)
```




```
<ggplot: (309539749)>
```

```
row_max = df_output.test_accuracy.idxmax()

print(f'Best accuracy was {df_output.test_accuracy[row_max]}, which corresponds to a
value of K={df_output.k[row_max]}')
```

```
Best accuracy was 0.7333333333333333, which corresponds to a value of K=4
```

From the results above, we see that the performance of KNN model increase to values around 73.33% in accuracy.

7.5.1.4. Overfitting and Underfitting (The Misclassification Rate vs K)

When implementing KNN (or other machine learning algorithms), one of the most important questions to consider is related to the choice of the number of neighbors (k) to use.

However, you should be aware of two issues that may arise as a result of the number of neighbors (k) you choose: **underfitting** and **overfitting**.

1. **Underfitting**

Underfitting occurs when there are

- too few predictors or
- a model that is too simplistic to accurately capture the data's relationships/patterns (large K in KNN).

As a result, a biased model arises, one that **performs poorly on both the data we used to train it and new data (low accuracy or high misclassification error)**.

1. **Overfitting** Overfitting is the opposite of underfitting, and it occurs when

- we use too many predictors or
- a model that is too complex (small K in KNN),

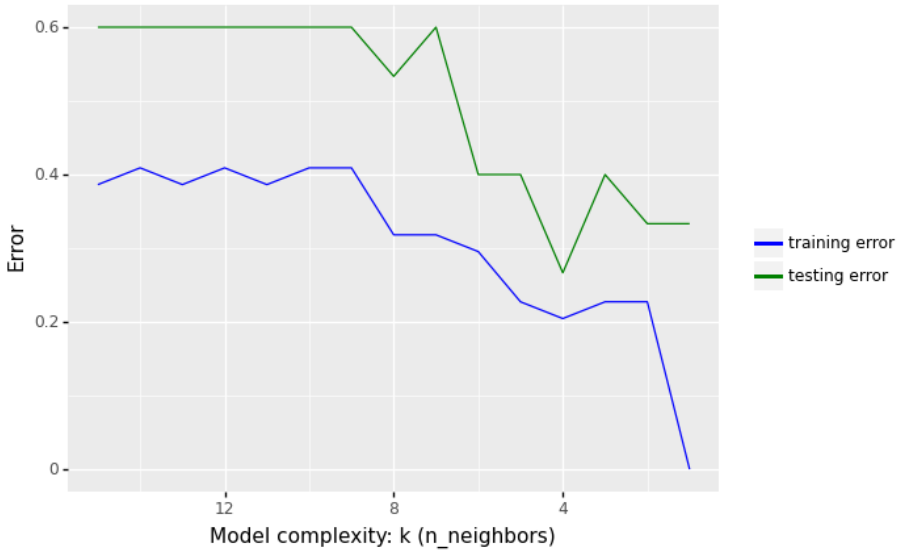
resulting in a model that models not just the relationships/patterns in our data, but also the noise.

Noise in a dataset is variance that is not consistently related to the variables we have observed, but is instead caused by inherent variability and/or measurement error.

Because the pattern of noise is so unique to each dataset, if we try to represent it, our model may perform **very well on the data we used to train it but produce poor prediction results on new datasets**.

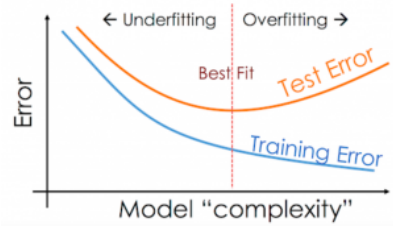
The following Python illustrate the concepts of underfitting and overfitting. It plots the misclassification rate (1 - accuracy) over the number of K neighbors.

```
# Error over the number of K neighbors

(
    ggplot(df_output)
    + geom_line(aes(x = 'k',  y = 'train_error',color='"training error"'))
    + geom_line(aes(x = 'k',  y = 'test_error',color='"testing error"'))
    + labs(x='Model complexity: k (n_neighbors)', y='Error')
    + scale_color_manual(values = ["blue", "green"], # Colors
        name = " ")
    + scale_x_continuous(trans = "reverse")
)
```




```
<ggplot: (309779273)>
```

7.5.1.4.1. Overfitting and Underfitting



In the figure above, we see that increasing K increases our error rate in the training set. The predictions begin to become biased (i.e., "over smoothing") by creating prediction values that approach the mean of the observed data set

In contrast, we get minimal error in the training set if we use K = 1, i.e. the model is just memorising the data.

What we are interested in, however, is the generalization error (test error), i.e., the expected value of the misclassification rate when averaged over new data

This value can be approximated by computing the misclassification rate on a large independent test set that was not used in training the model.

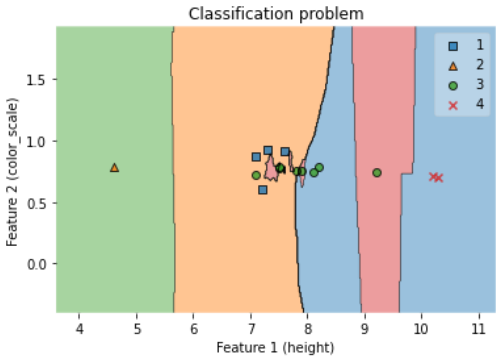
We plot the test error against K in green (upper curve). Now we see a **U-shaped curve**: for complex models (small K), the method overfits, and for simple models (big K), the method underfits.

7.5.1.5. Visualization of decision regions (KNN model with two features, **height** and **color_score**)

```
##### Convert pandas DataFrame to Numpy before applying classification

X_train_np = X_train.to_numpy()
y_train_np = y_train.to_numpy()

X_test_np = X_test.to_numpy()
y_test_np = y_test.to_numpy()

clf = KNeighborsClassifier(n_neighbors = 4)
clf.fit(X_train_np, y_train_np)

# Plotting decision regions
fig, ax = plt.subplots()

# Decision region for feature 3 = mass = value
#value=160
# Plot training sample with feature = mass = value +/- width
#width=20

fig = plot_decision_regions(X_test_np, y_test_np, clf=clf)
ax.set_xlabel('Feature 1 (height)')
ax.set_ylabel('Feature 2 (color_scale)')
ax.set_title('Classification problem')
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/mlxtend/plotting/decision_regions.py:279: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
```

```
Text(0.5, 1.0, 'Classification problem')
```




```
#clf.predict([[ 8.5 ,  0],
#       [9.2 ,  0]])
```

7.6. Scaling Features in KNN

KNN is a distance-based algorithm, where KNN classifies data based on proximity to K neighbors. Then, we often find that the features of the data we are using are not on the same scale/unit. An example of this is the characteristics of weight and height. Obviously these two features have different units, the feature weight is in kilograms and height is in centimeters.

Because this unit difference causes Distance-Based algorithms including KNN to perform poorly, rescaling features with different units to the same scale/units is required to **improve performance of K-Nearest Neighbors**.

Another reason for using feature scaling is that some algorithms, such as gradient descent in neural networks, converge faster with it than without it.

7.6.1. Common techniques of feature scaling

There are a variety of methods for rescaling features including

- **Min-Max Scaling**

The simplest method, also known as **min-max scaling** or **min-max normalization**, consists of rescaling the range of features to scale the range in [0, 1] or [1, 1]. The goal range is determined by the data's type. The following is the general formula for a min-max of [0, 1]:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

where x is the original value and x' is the normalized value.

- **Standard Scaling**

Feature standardization, also known as Z-score Normalization, ensures that the values of each feature in the data have a mean of zero (when subtracting the mean in the numerator) and a unit variance.

$$x' = \frac{x - \bar{x}}{\sigma}$$

where x is the original feature vector, $\bar{x} = \text{average}(x)$ is the mean of that feature vector, and σ is its standard deviation.

This method is commonly utilized in various machine learning methods for normalization (e.g., support vector machines, logistic regression, and artificial neural networks)

- Robust Scaling.

This Scaler is robust to outliers, as the name suggests. The mean and standard deviation of the data will not scale well if our data contains several outliers.

For this robust scaling, the median is removed, and the data is scaled according to the interquartile range The interquartile range (IQR) is the distance between the first and third quartiles (25th and 3rd quantiles) (75th quantile). Because this Scaler's centering and scaling statistics are based on percentiles, they are unaffected by a few large marginal outliers.

For more details of other scaling features, please follow the links:

https://towardsdatascience.com/all-about-feature-scaling-bcc0ad75cb35

7.6.2. Feature Scaling the Fruit Dataset

In this section, we will perform min-max scaling to rescaling the features **height** and **color_score** before training the KNN model.

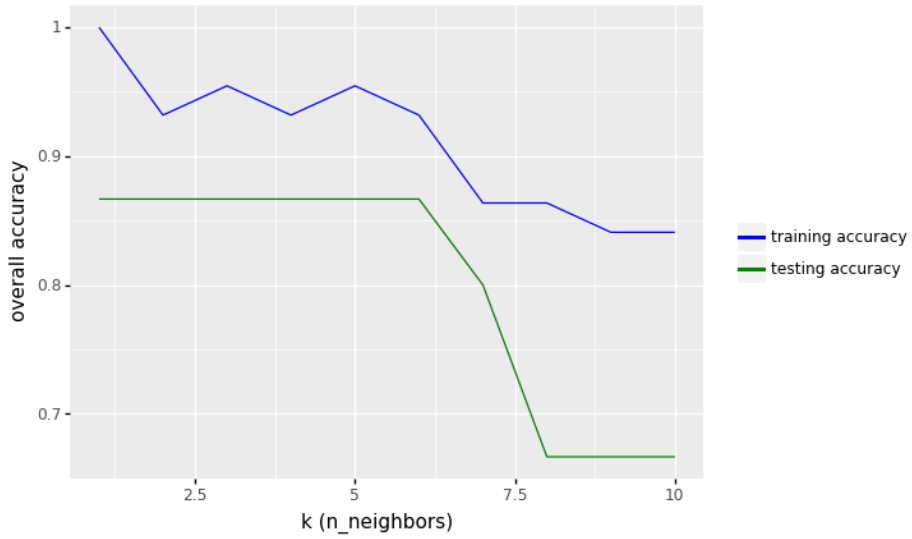
```
#df.head()
```

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.metrics import accuracy_score


# Importing a dataset
url = 'https://raw.githubusercontent.com/susanli2016/Machine-Learning-with-
Python/master/fruit_data_with_colors.txt'
df = pd.read_table(url)

# Make a copy of the original dataset
df_model = df.copy()

#Rescaling features 'mass','width','height','color_score'.
#scaler = StandardScaler()
#scaler = RobustScaler()
scaler = MinMaxScaler()


features = [['mass','width','height','color_score']]
for feature in features:
    df_model[feature] = scaler.fit_transform(df_model[feature])

#print(df_model.head())

# Instantiate the estimator
knn = KNeighborsClassifier()

#Create x and y variable
#X = df_model[['mass','width','height','color_score']]
#y = df_model['fruit_label']

X = df_model[['height', 'color_score']]
y = df_model['fruit_label']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=0)

k_range = range(1,11)

train_accuracy = {}
train_accuracy_list = []

test_accuracy = {}
test_accuracy_list = []

# misclassfication error
train_error_list = []
test_error_list = []

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors = k)

    # Training the classifier by passing in the training set X_train and the labels in
y_train
    knn.fit(X_train,y_train)

    # Compute accuracy on the training set
    train_accuracy[k] = knn.score(X_train, y_train)
    train_accuracy_list.append(knn.score(X_train, y_train))
    train_error_list.append(1 - knn.score(X_train, y_train))

    test_accuracy[k] = knn.score(X_test,y_test)
    test_accuracy_list.append(knn.score(X_test,y_test))
    test_error_list.append(1 - knn.score(X_test,y_test))

df_output = pd.DataFrame({'k':k_range,
                          'train_accuracy':train_accuracy_list,
                          'test_accuracy':test_accuracy_list,
                          'train_error':train_error_list,
                          'test_error':test_error_list
                         })
```

```
df_model.describe()
```

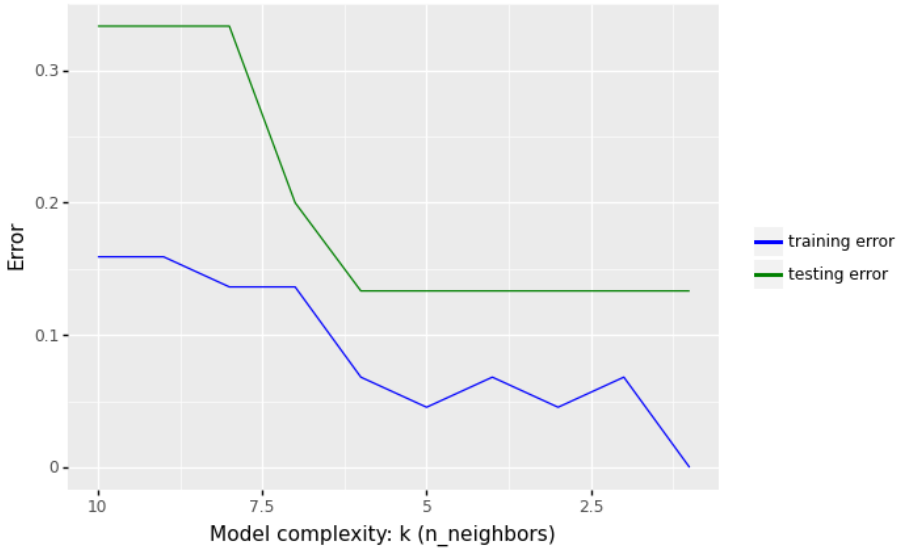
	fruit_label	mass	width	height	color_score
count	59.000000	59.000000	59.000000	59.000000	59.000000
mean	2.542373	0.304611	0.343443	0.568188	0.560214
std	1.208048	0.192374	0.214984	0.209387	0.202257
min	1.000000	0.000000	0.000000	0.000000	0.000000
25%	1.000000	0.223776	0.210526	0.492308	0.447368
50%	3.000000	0.286713	0.368421	0.553846	0.526316
75%	4.000000	0.353147	0.447368	0.646154	0.684211
max	4.000000	1.000000	1.000000	1.000000	1.000000

```
# Accuracy over the number of K neighbors

(
    ggplot(df_output)
    + geom_line(aes(x = 'k',  y = 'train_accuracy',color='"training accuracy"'))
    + geom_line(aes(x = 'k',  y = 'test_accuracy',color='"testing accuracy"'))
    + labs(x='k (n_neighbors)', y='overall accuracy')
    + scale_color_manual(values = ["blue", "green"], # Colors
        name = " ")
)
```




```
<ggplot: (309897553)>
```

```
# Error over the number of K neighbors

(
    ggplot(df_output)
    + geom_line(aes(x = 'k',  y = 'train_error',color='"training error"'))
    + geom_line(aes(x = 'k',  y = 'test_error',color='"testing error"'))
    + labs(x='Model complexity: k (n_neighbors)', y='Error')
    + scale_color_manual(values = ["blue", "green"], # Colors
        name = " ")
    + scale_x_continuous(trans = "reverse")
)
```




```
<ggplot: (310429937)>
```

```
row_max = df_output.test_accuracy.idxmax()

print(f'Best accuracy was {df_output.test_accuracy[row_max]}, which corresponds to a
value of K between 1 and 6')
```

```
Best accuracy was 0.8666666666666667, which corresponds to a value of K between 1 and 6
```

```
df_output.sort_values(by='test_error', ascending=True)
```

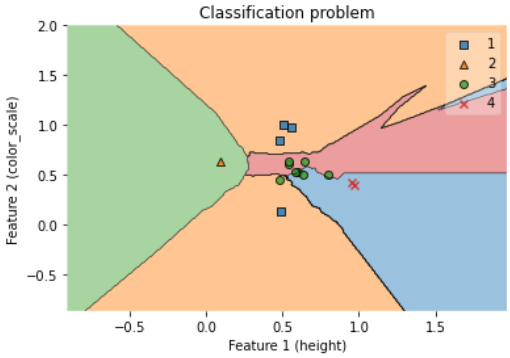
	k	train_accuracy	test_accuracy	train_error	test_error
0	1	1.000000	0.866667	0.000000	0.133333
1	2	0.931818	0.866667	0.068182	0.133333
2	3	0.954545	0.866667	0.045455	0.133333
3	4	0.931818	0.866667	0.068182	0.133333
4	5	0.954545	0.866667	0.045455	0.133333
5	6	0.931818	0.866667	0.068182	0.133333
6	7	0.863636	0.800000	0.136364	0.200000
7	8	0.863636	0.666667	0.136364	0.333333
8	9	0.840909	0.666667	0.159091	0.333333
9	10	0.840909	0.666667	0.159091	0.333333

From the results above, we see that the performance of KNN model with two features (height and color_score) after rescaling the features increase from 73.33% to values around 86.67% in accuracy (for K between 1 to 6).

```
##### Convert pandas DataFrame to Numpy before applying classification

X_train_np = X_train.to_numpy()
y_train_np = y_train.to_numpy()

X_test_np = X_test.to_numpy()
y_test_np = y_test.to_numpy()

clf = KNeighborsClassifier(n_neighbors = 6)
clf.fit(X_train_np, y_train_np)

# Plotting decision regions
fig, ax = plt.subplots()

# Decision region for feature 3 = mass = value
value=160
# Plot training sample with feature = mass = value +/- width
width=20

fig = plot_decision_regions(X_test_np, y_test_np, clf=clf)
ax.set_xlabel('Feature 1 (height)')
ax.set_ylabel('Feature 2 (color_scale)')
ax.set_title('Classification problem')
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/mlxtend/plotting/decision_regions.py:279: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
```

```
Text(0.5, 1.0, 'Classification problem')
```



7.7. Model validation

7.7.1. Holdout sets

We can gain a better understanding of a model's performance by employing a **holdout** (or test) set, which is when we hold back a subset of data from the model's training and then use it to verify the model's performance. This may be done **train_test_split** in Scikit-train as previously done.

```
# Importing a dataset
url = 'https://raw.githubusercontent.com/susanli2016/Machine-Learning-with-
Python/master/fruit_data_with_colors.txt'
df = pd.read_table(url)

# Make a copy of the original dataset
df_model = df.copy()

#Rescaling features 'mass','width','height','color_score'.
#scaler = StandardScaler()
#scaler = RobustScaler()
scaler = MinMaxScaler()


features = [['mass','width','height','color_score']]
for feature in features:
    df_model[feature] = scaler.fit_transform(df_model[feature])


#Create x and y variable
#X = df_model[['mass','width','height','color_score']]
#y = df_model['fruit_label']

X = df_model[['height', 'color_score']]
y = df_model['fruit_label']

print(X.head())
```

```
     height  color_score
0  0.507692     0.000000
1  0.430769     0.105263
2  0.492308     0.131579
3  0.107692     0.657895
4  0.092308     0.631579
```

In this example, we use half of the data as a training set and the other half as the validation set with `n_neighbors=5`.

```
X1, X2, y1, y2 = train_test_split(X, y, test_size=0.5, random_state=0)

model1 = KNeighborsClassifier(n_neighbors=5)

model1.fit(X1,y1)
accuracy_score(y2, model1.predict(X2))
```

```
0.7
```

We obtain an accuracy score of 0.7 which indicates that 70% of points were correctly labeled by our model

```
# checking the number of fruits by type in the training data

y1.value_counts()
```

```
1    12
4    10
3     5
2     2
Name: fruit_label, dtype: int64
```

7.7.2. Cross-validation

One drawback of using a holdout set for model validation is that we lost some of our data during model training. In the preceding example, half of the dataset is not used to train the model! This is inefficient and can lead to issues, particularly if the initial batch of training data is small.

One solution is to employ **cross-validation**, which requires performing a series of fits in which each subset of the data is used as both a training and a validation set.

We perform two validation trials here, utilizing each half of the data as a holdout set alternately.

With the split data from earlier, we could implement it like this:

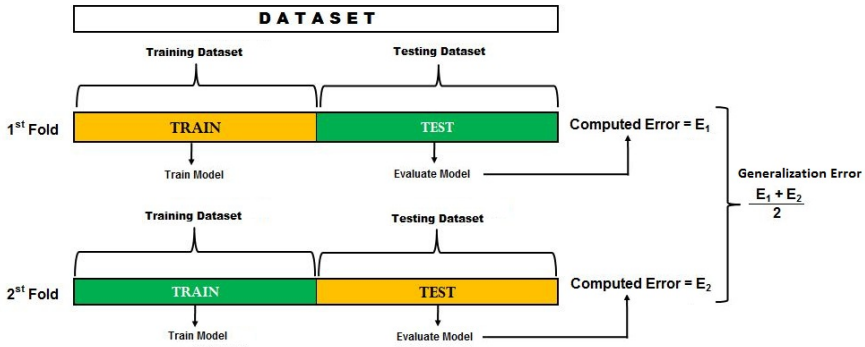
```
model1 = KNeighborsClassifier(n_neighbors=5)
model2 = KNeighborsClassifier(n_neighbors=5)

model1.fit(X1,y1)
model2.fit(X2,y2)

print('Accuracy score of model 1:', accuracy_score(y2, model1.predict(X2)))
print('Accuracy score of model 2:', accuracy_score(y1, model2.predict(X1)))

# use the average as an estimate of the accuracy
print('\nAn estimate of the accuracy: ',(accuracy_score(y2,
model1.fit(X1,y1).predict(X2)) + accuracy_score(y1, model2.fit(X2,y2).predict(X1)))/2)
```

```
Accuracy score of model 1: 0.7
Accuracy score of model 2: 0.6896551724137931

An estimate of the accuracy:  0.6948275862068966
```

The result is two accuracy scores, which we may combine (for example, by **taking the mean**) to produce a more **accurate estimate** of the global model performance.

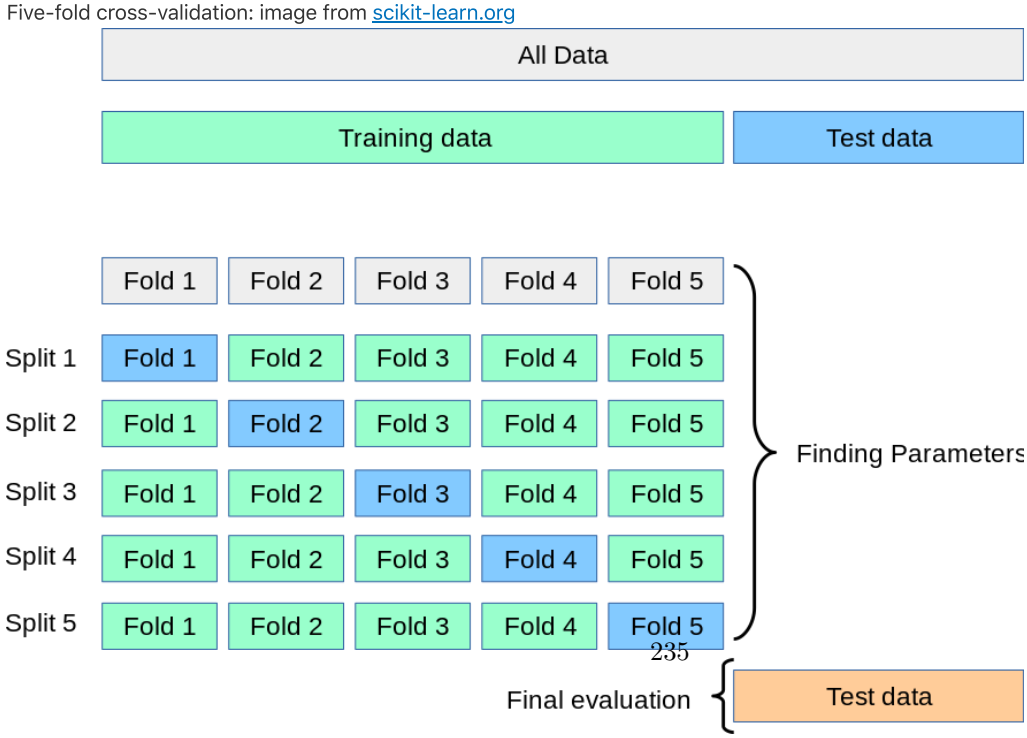
This type of cross-validation is a **two-fold cross-validation**, in which the data is split into two sets and each is used as a validation set in turn.

Two-fold cross-validation: image from datavedas



This notion might be expanded to include more trials and folds in the data—for example, the below figure from scikit-learn.org shows a visual representation of five-fold cross-validation.

Five-fold cross-validation: image from scikit-learn.org



we can use Scikit-Learn's cross_val_score convenience routine to do it k-fold cross-validation. The Python code below performs k-fold cross-validation.

In what follows, we perform StratifiedKFold to split the data in 2 folds (defined by `n_splits = 2`) (see Chapter 5 for more details).

The `StratifiedKFold` module in Scikit-learn sets up `n_splits` (folds, partitions or groups) of the dataset in a way that the folds are made by **preserving the percentage of samples for each class**.

The brief explaination for the code below (also see the diagram below) is as follows:

1. The dataset has been split into K (K = 2 in our example) equal partitions (or folds).
2. (In iteration 1) use fold 1 as the testing set and the union of the other folds as the training set.
3. Repeat step 2 for K times, using a different fold as the testing set each time.

```
from sklearn.model_selection import cross_val_score, StratifiedKFold

knn = KNeighborsClassifier(n_neighbors=5)

# Create StratifiedKFold object.
skf = StratifiedKFold(n_splits=2, shuffle=True, random_state=0)

# store cross-validation scores in cv_scores object
cv_scores = []

for train_index, test_index in skf.split(X, y):
    X_train_fold, X_test_fold = X.loc[train_index,], X.loc[test_index,]
    y_train_fold, y_test_fold = y.loc[train_index,], y.loc[test_index,]
    knn.fit(X_train_fold, y_train_fold)
    cv_scores.append(knn.score(X_test_fold, y_test_fold))

print(cv_scores)
```

```
[0.7, 0.7931034482758621]
```

The accuracy scores of the two-fold cross-validation are 0.7 and 0.793.

7.7.2.1. Computing cross-validated metrics¶

We can use Scikit-Learn's cross_val_score convenience routine to do it k-fold cross-validation without the need to create a StratifiedKFold object (`skf` above). The Python code below computes the cross-validation scores.

```
from sklearn.model_selection import cross_val_score

knn = KNeighborsClassifier(n_neighbors=5)

cross_val_score(knn, X, y, cv=2)

print('The accuracy scores after performing two-fold cross-vlaidation
are:',cross_val_score(knn, X, y, cv=2))
```

```
The accuracy scores after performing two-fold cross-vlaidation are: [0.66666667 0.5862069
]
```

It is also possible to use other cross validation strategies by passing a cross validation iterator instead, for instance:

See the link below for more details:

https://scikit-learn.org/stable/modules/cross_validation.html

```
from sklearn.model_selection import ShuffleSplit

knn = KNeighborsClassifier(n_neighbors=5)

cv = ShuffleSplit(n_splits=2, test_size=0.5, random_state=1)

cross_val_score(knn, X, y, cv=cv)
```

```
array([0.8       , 0.86666667])
```

Note The validation scores from the previous results are differents due to the training and test sets are randomly sampled from the original dataset.

Exercise Plot the (mean) misclassification error over the number of K neighbours to find the optimal K value that gives the maximum mean accuracy score (the mean of the accuracy scores obtained from the two-fold cross-validation).

7.7.3. Hypertuning Model Parameters using Grid Search

In the validation section, we set the parameter 'n_neighbors' to 5 as a starting point when we generated our KNN models.

When we go through a process to **identify the best parameters** for our model to improve accuracy, we are **hypertuning parameters**. We will use **GridSearchCV** to find the best value for 'n_neighbors' in our situation.

GridSearchCV works by repeatedly training our model on a set of parameters that we define. That manner, we can test our model with each parameter and determine the best settings for maximum accuracy. In order to see which value for 'n_neighbors' works best for our model, we will specify a range of values.

To perform so, we'll make a dictionary with the key 'n_neighbors' and use numpy to build an array of values ranging from 1 to 15. In order to determine the best value for 'n_neighbors,' our new grid search model will use a new k-NN classifier, our param grid, and a cross-validation value of 2 (use 2 in our case why?).

```
# Importing a dataset
url = 'https://raw.githubusercontent.com/susanli2016/Machine-Learning-with-
Python/master/fruit_data_with_colors.txt'
df = pd.read_table(url)

# Make a copy of the original dataset
df_model = df.copy()

#Rescaling features 'mass','width','height','color_score'.
#scaler = StandardScaler()
#scaler = RobustScaler()
scaler = MinMaxScaler()


features = [['mass','width','height','color_score']]
for feature in features:
    df_model[feature] = scaler.fit_transform(df_model[feature])


#Create X and y variable
X = df_model[['height', 'color_score']]
y = df_model['fruit_label']

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=0)
```

To get started, we need to import `GridSearchCV` from the Sklearn library.

The model we use for hyperparameter tuning is required (`knn` in our case) by the **estimator parameter** of GridSearchCV.

A list of parameters and the range of values for each parameter of the specified estimator are required by the **param_grid parameter**. When working with the KNN model, the most important parameter is **n_neighbors**.

Cross-validation is performed to determine the hyperparameter value set that provides the best accuracy.

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

```
# the estimator used in out example
knn = KNeighborsClassifier()

# create a parameter grid: map the parameter names to the values that should be searched
# simply a python dictionary
# key: parameter name
# value: list of values that should be searched for that parameter
# single key-value pair for param_grid
param_grid = {'n_neighbors':np.arange(1,16)}

#k_range = list(range(1, 16))
#param_grid = dict(n_neighbors=k_range)
```

```
# instantiate the grid
grid = GridSearchCV(knn, param_grid, cv=2, scoring='accuracy')

#grid = GridSearchCV(knn, param_grid, cv=2, scoring='balanced_accuracy')
```

The grid object is ready for 2-fold cross-validation of a KNN model with classification accuracy as the evaluation measure.

- There is also a grid parameter to repeat the 2-fold cross-validation 15 times.
- Each time, the parameter **n_neighbors** should get a different value from the list.
- Here we specify that n_neighbors should take the values 1 to 15.
- We can designate a scorer object with the scoring parameter, `scoring='accuracy'` in this example. (for more details https://scikit-learn.org/stable/modules/model_evaluation.html)
- You can set n_jobs = -1 to run calculations in parallel (if your computer and OS support it). This is also called **parallel programming**.

```
# fit the grid with the training set
grid.fit(X_train, y_train)
```

```
GridSearchCV(cv=2, estimator=KNeighborsClassifier(),
             param_grid={'n_neighbors': array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10,
11, 12, 13, 14, 15])},
             scoring='accuracy')
```

`cv_results_` returns a dictionary of all the evaluation metrics from the gridsearch. To visualize it properly, you can convert into a pandas DataFrame. The results include the test scores for the two folds, the mean and standard deviation of the test scores and also the ranking of the mean test scores.

```
# view the complete results
# cv_results_dict: this attributes gives a dict with keys as column headers
# and values as columns, that can be imported into a pandas DataFrame.

pd.DataFrame(grid.cv_results_).head()
```

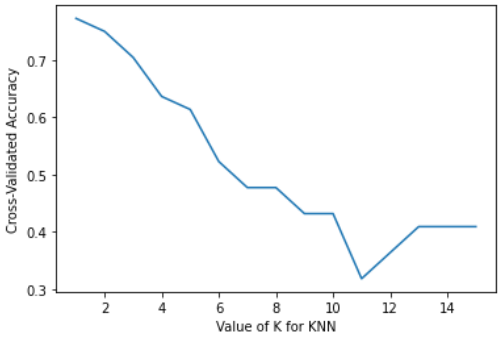
	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_neighbors	params	split0_test_score	spli
0	0.005679	0.002084	0.005561	0.000824	1	{'n_neighbors': 1}	0.636364	
1	0.002670	0.000004	0.006662	0.003166	2	{'n_neighbors': 2}	0.727273	
2	0.002560	0.000067	0.003976	0.000336	3	{'n_neighbors': 3}	0.636364	
3	0.007777	0.005566	0.005177	0.001293	4	{'n_neighbors': 4}	0.500000	
4	0.002213	0.000024	0.003185	0.000199	5	{'n_neighbors': 5}	0.545455	

```
# Create a list of the mean scores

grid_mean_scores = grid.cv_results_['mean_test_score']
```


Plot the value of K for KNN (x-axis) against the cross-validated accuracy (y-axis).

```
# Plot the result
k_range = list(range(1, 16))
plt.plot(k_range, grid_mean_scores)
plt.xlabel('Value of K for KNN')
plt.ylabel('Cross-Validated Accuracy')
```

```
Text(0, 0.5, 'Cross-Validated Accuracy')
```



After training, we can check which of the values we tested for 'n_neighbors' worked best. For this purpose we call 'best_params_' in our model.

```
# examine the best model

# Single best score achieved across all params (k)
print(grid.best_score_)

# Dictionary containing the parameters (k) used to generate that score
print(grid.best_params_)

# Actual model object fit with those best parameters
# Shows default parameters that we did not specify
print(grid.best_estimator_)
```

```
0.7727272727272727
{'n_neighbors': 1}
KNeighborsClassifier(n_neighbors=1)
```

The results show that the cross-validated accuracy is 77.27% when the value of the K is 1.

7.7.3.1. Using the best parameter to make prediction and result evaluation

The classfification report on the validation set can be created on a per-class basis. It provides a deeper understanding of the behavior of the classifier than global accuracy, which can disguise functional weaknesses in a class of a multiclass problem.

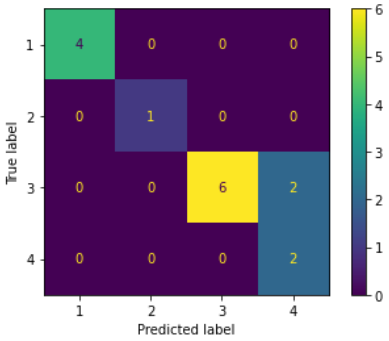
```
# Using the best parameter obtained earlier

best_model = KNeighborsClassifier(n_neighbors=1)

# Train out best_model on the training set
best_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = best_model.predict(X_test)
```

We can crate the confusion metrix and the summary of the predictions.

```
# Confusion Matrix visualization.

cm = confusion_matrix(y_test, y_pred, labels=best_model.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,display_labels=best_model.classes_)
disp.plot()
```



```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x1287f8e10>
```




```
print('Classification report: \n')
print(classification_report(y_test, y_pred))
```

```
Classification report:

              precision    recall  f1-score   support

           1       1.00      1.00      1.00         4
           2       1.00      1.00      1.00         1
           3       1.00      0.75      0.86         8
           4       0.50      1.00      0.67         2

    accuracy                           0.87        15
   macro avg       0.88      0.94      0.88        15
weighted avg       0.93      0.87      0.88        15
```

Observation of the classification report for the prediction model as follows.

- The classifier made a total of 15 predictions.
- Out of those 15 samples, the classifier predicted 4 apples, 1 mandarins, 6 oranges and 4 lemons.
- In reality, out of 4 predicted lemons, two of them are oranges.

The overall accuracy of the prediction of the fruit data set by applying the KNN classifier model is 86.67 which means the model performs well in our example.

```
accuracy_score(y_test,y_pred)
```

```
0.8666666666666667
```

7.7.3.2. Swarmplots

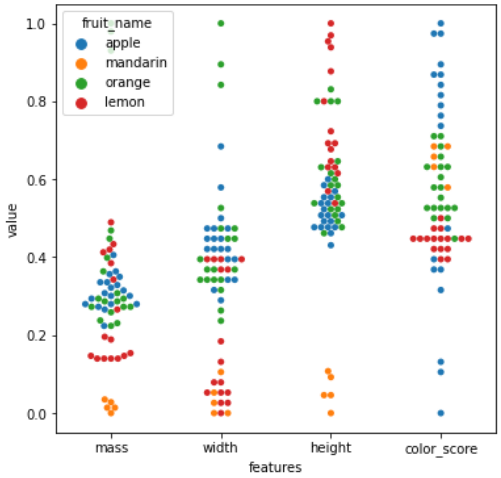
```
df_model.columns
```

```
Index(['fruit_label', 'fruit_name', 'fruit_subtype', 'mass', 'width', 'height',
       'color_score'],
      dtype='object')
```

```
#df_model[['fruit_name','mass', 'width','height', 'color_score']]
```

```
data = pd.melt(df_model[['fruit_name','mass', 'width','height', 'color_score']], id_vars
= 'fruit_name', var_name = 'features',value_name = 'value')
```

```
#data
# swarmplot for analysing the different attributes
plt.figure(figsize = (6,6))
sns.swarmplot(x = 'features', y = 'value', hue = 'fruit_name', data = data)
plt.show()
```




```
X = df_model[['mass', 'width','height', 'color_score']]
y = df_model['fruit_label']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=0)
```

```
from sklearn.feature_selection import chi2, SelectKBest, f_classif
```

```
# Get the two best(k = 2) features using the SelectKBest method
ft = SelectKBest(chi2, k = 2).fit(X_train, y_train)
print('Score: ', ft.scores_)
print('Columns: ', X_train.columns)
```

```
Score:  [2.0299807  3.68664781 2.56587049 0.43604189]
Columns:  Index(['mass', 'width', 'height', 'color_score'], dtype='object')
```

```
ft = SelectKBest(f_classif, k= 2).fit(X_train, y_train)
print('Score: ', ft.scores_)
print('Columns: ', X_train.columns)
```

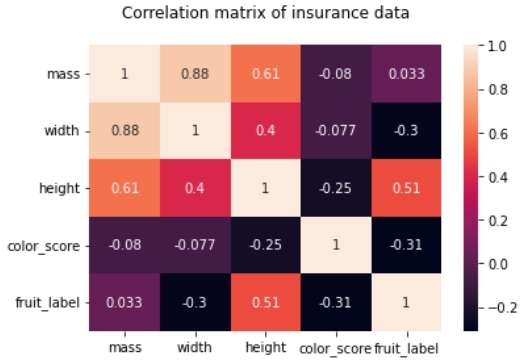
```
Score:  [ 8.69254183 18.22929591 40.53528918  2.26802478]
Columns:  Index(['mass', 'width', 'height', 'color_score'], dtype='object')
```

```
# Importing a dataset
url = 'https://raw.githubusercontent.com/susanli2016/Machine-Learning-with-
Python/master/fruit_data_with_colors.txt'
df = pd.read_table(url)

from sklearn.preprocessing import MinMaxScaler

features = [['mass','width','height','color_score']]

X = df[['mass','width','height','color_score']]
y = df['fruit_label']

scaler = MinMaxScaler()

scaled_X = pd.DataFrame(scaler.fit_transform(X),
                        columns=X.columns)
```

```
scaled_X.head()
```

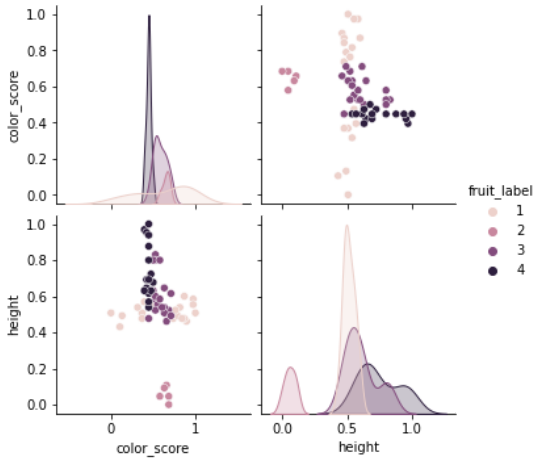
	mass	width	height	color_score
0	0.405594	0.684211	0.507692	0.000000
1	0.363636	0.578947	0.430769	0.105263
2	0.349650	0.421053	0.492308	0.131579
3	0.034965	0.105263	0.107692	0.657895
4	0.027972	0.052632	0.092308	0.631579

```
scaled_df = scaled_X
scaled_df['fruit_label'] = df[['fruit_label']]
#scaled_df

hm = sns.heatmap(scaled_df.corr(), annot = True)
hm.set(title = "Correlation matrix of insurance data\n")
plt.show()
```




```
sns.pairplot(scaled_df[['fruit_label','color_score','height']],hue='fruit_label')
```

```
<seaborn.axisgrid.PairGrid at 0x127982c50>
```



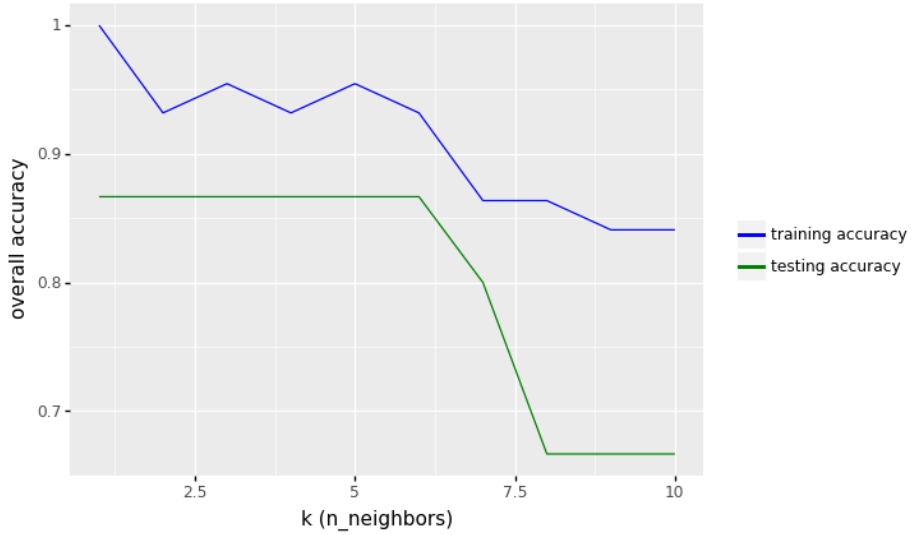
```
# Importing a dataset
#url = 'https://raw.githubusercontent.com/susanli2016/Machine-Learning-with-
Python/master/fruit_data_with_colors.txt'
#df = pd.read_table(url)

# Train Test Split

X = scaled_df[['height', 'color_score']]
y = scaled_df['fruit_label']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=0)

# Instantiate the estimator
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()

k_range = range(1,11)

train_accuracy = {}
train_accuracy_list = []

test_accuracy = {}
test_accuracy_list = []

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors = k)

    # Training the classifier by passing in the training set X_train and the labels in
y_train
    knn.fit(X_train,y_train)

    # Compute accuracy on the training set
    train_accuracy[k] = knn.score(X_train, y_train)
    train_accuracy_list.append(knn.score(X_train, y_train))


    test_accuracy[k] = knn.score(X_test,y_test)
    test_accuracy_list.append(knn.score(X_test,y_test))
```

```
train_accuracy
print(test_accuracy)
```

```
{1: 0.8666666666666667, 2: 0.8666666666666667, 3: 0.8666666666666667, 4:
0.8666666666666667, 5: 0.8666666666666667, 6: 0.8666666666666667, 7: 0.8, 8:
0.6666666666666666, 9: 0.6666666666666666, 10: 0.6666666666666666}
```

```
#scaled_df_output
```

```
scaled_df_output = pd.DataFrame({'k':k_range,
                         'train_accuracy':train_accuracy_list,
                         'test_accuracy':test_accuracy_list
                        })

(
    ggplot(scaled_df_output)
    + geom_line(aes(x = 'k',  y = 'train_accuracy',color='"training accuracy"'))
    + geom_line(aes(x = 'k',  y = 'test_accuracy',color='"testing accuracy"'))
    + labs(x='k (n_neighbors)', y='overall accuracy')
    + scale_color_manual(values = ["blue", "green"], # Colors
        name = " ")
)
```

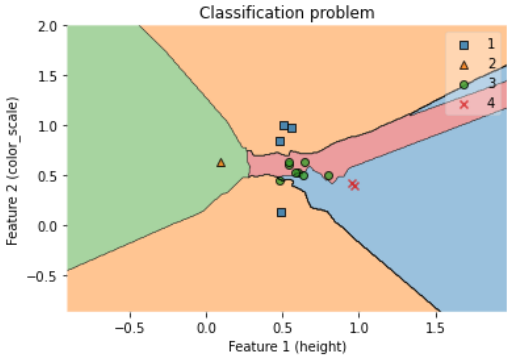



```
<ggplot: (300097561)>
```

```
##### Convert pandas DataFrame to Numpy before applying classification

X_train_np = X_train.to_numpy()
y_train_np = y_train.to_numpy()

X_test_np = X_test.to_numpy()
y_test_np = y_test.to_numpy()

clf = KNeighborsClassifier(n_neighbors = 4)
clf.fit(X_train_np, y_train_np)

# Plotting decision regions
fig, ax = plt.subplots()

# Decision region for feature 3 = mass = value
value=160
# Plot training sample with feature = mass = value +/- width
width=20

fig = plot_decision_regions(X_test_np, y_test_np, clf=clf)
ax.set_xlabel('Feature 1 (height)')
ax.set_ylabel('Feature 2 (color_scale)')
ax.set_title('Classification problem')
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/mlxtend/plotting/decision_regions.py:279: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
```

```
Text(0.5, 1.0, 'Classification problem')
```




```
test_accuracy
```

```
{1: 0.8666666666666667,
 2: 0.8666666666666667,
 3: 0.8666666666666667,
 4: 0.8666666666666667,
 5: 0.8666666666666667,
 6: 0.8666666666666667,
 7: 0.8,
 8: 0.6666666666666666,
 9: 0.6666666666666666,
 10: 0.6666666666666666}
```

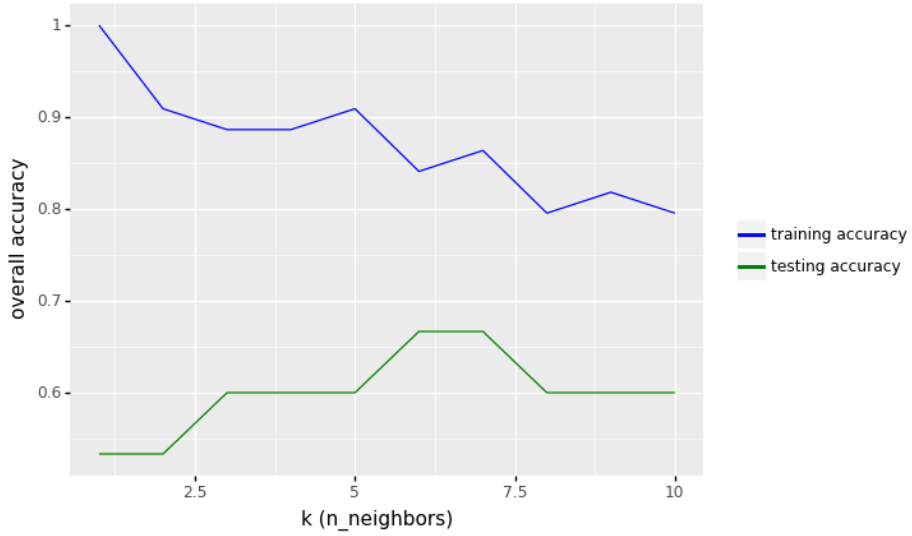
7.7.3.3. Feature Scaling the Fruit Dataset (width and height)

In this section, we will perform min-max scaling to rescaling the features **height** and **color_score** before training the KNN model.

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

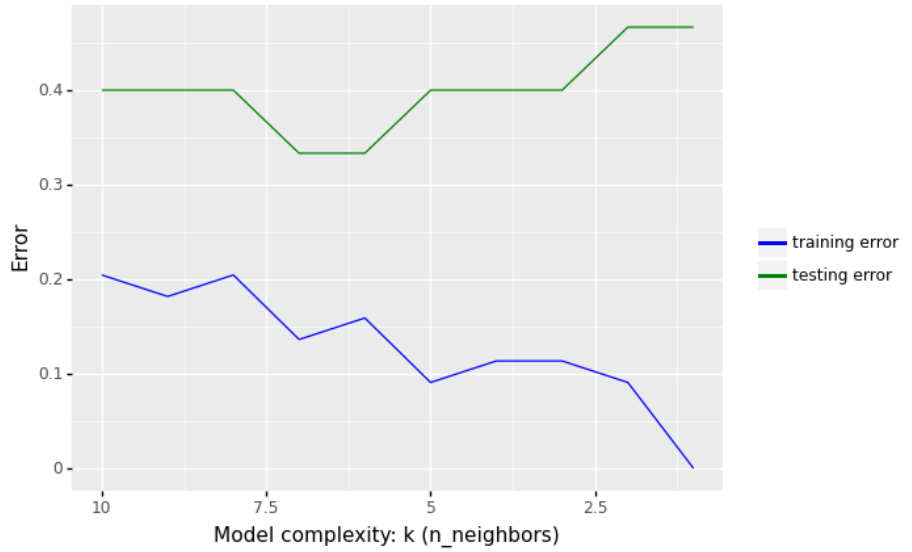
```
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.metrics import accuracy_score


# Importing a dataset
url = 'https://raw.githubusercontent.com/susanli2016/Machine-Learning-with-
Python/master/fruit_data_with_colors.txt'
df = pd.read_table(url)

# Make a copy of the original dataset
df_model = df.copy()

#Rescaling features 'mass','width','height','color_score'.
#scaler = StandardScaler()
#scaler = RobustScaler()
scaler = MinMaxScaler()


features = [['mass','width','height','color_score']]
for feature in features:
    df_model[feature] = scaler.fit_transform(df_model[feature])

#print(df_model.head())

# Instantiate the estimator
knn = KNeighborsClassifier()

#Create x and y variable
#X = df_model[['mass','width','height','color_score']]
#y = df_model['fruit_label']

X = df_model[['height', 'width']]
y = df_model['fruit_label']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=0)

k_range = range(1,11)

train_accuracy = {}
train_accuracy_list = []

test_accuracy = {}
test_accuracy_list = []

# misclassfication error
train_error_list = []
test_error_list = []

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors = k)

    # Training the classifier by passing in the training set X_train and the labels in
y_train
    knn.fit(X_train,y_train)

    # Compute accuracy on the training set
    train_accuracy[k] = knn.score(X_train, y_train)
    train_accuracy_list.append(knn.score(X_train, y_train))
    train_error_list.append(1 - knn.score(X_train, y_train))

    test_accuracy[k] = knn.score(X_test,y_test)
    test_accuracy_list.append(knn.score(X_test,y_test))
    test_error_list.append(1 - knn.score(X_test,y_test))

df_output = pd.DataFrame({'k':k_range,
                          'train_accuracy':train_accuracy_list,
                          'test_accuracy':test_accuracy_list,
                          'train_error':train_error_list,
                          'test_error':test_error_list
                         })
```

```
# Accuracy over the number of K neighbors

(
    ggplot(df_output)
    + geom_line(aes(x = 'k',  y = 'train_accuracy',color='"training accuracy"'))
    + geom_line(aes(x = 'k',  y = 'test_accuracy',color='"testing accuracy"'))
    + labs(x='k (n_neighbors)', y='overall accuracy')
    + scale_color_manual(values = ["blue", "green"], # Colors
        name = " ")
)
```




```
<ggplot: (313184393)>
```

```
# Error over the number of K neighbors

(
    ggplot(df_output)
    + geom_line(aes(x = 'k',  y = 'train_error',color='"training error"'))
    + geom_line(aes(x = 'k',  y = 'test_error',color='"testing error"'))
    + labs(x='Model complexity: k (n_neighbors)', y='Error')
    + scale_color_manual(values = ["blue", "green"], # Colors
       name = " ")
    + scale_x_continuous(trans = "reverse")
)
```




```
<ggplot: (312075721)>
```

```
row_max = df_output.test_accuracy.idxmax()

print(f'Best accuracy was {df_output.test_accuracy[row_max]}, which corresponds to a
value of K between 1 and 6')
```

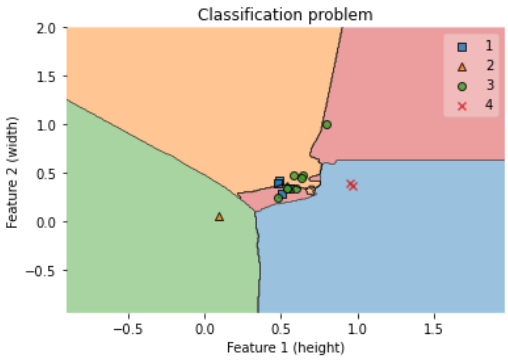
```
Best accuracy was 0.6666666666666666, which corresponds to a value of K between 1 and 6
```

From the results above, we see that the performance of KNN model with two features (height and color_score) increase from 73.33% to values around 86.67% in accuracy.

```
##### Convert pandas DataFrame to Numpy before applying classification

X_train_np = X_train.to_numpy()
y_train_np = y_train.to_numpy()

X_test_np = X_test.to_numpy()
y_test_np = y_test.to_numpy()

clf = KNeighborsClassifier(n_neighbors = 6)
clf.fit(X_train_np, y_train_np)

# Plotting decision regions
fig, ax = plt.subplots()

# Decision region for feature 3 = mass = value
#value=160
# Plot training sample with feature = mass = value +/- width
#width=20

fig = plot_decision_regions(X_test_np, y_test_np, clf=clf)
ax.set_xlabel('Feature 1 (height)')
ax.set_ylabel('Feature 2 (width)')
ax.set_title('Classification problem')
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-
packages/mlxtend/plotting/decision_regions.py:279: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').  Matplotlib is ignoring the
edgecolor in favor of the facecolor.  This behavior may change in the future.
```

```
Text(0.5, 1.0, 'Classification problem')
```



Previous
6. Regression Analysis

Next
8. Unsupervised Machine Learning: K-means Clustering

By Pairote Satiracoo
© Copyright 2021.

8 Unsupervised Machine Learning: K-means Clustering

8. Unsupervised Machine Learning: K-means Clustering

In this chapter, we will take an in-depth look at **k-means clustering** and its components. We explain why clustering is important, how to use it, and how to perform it in Python with a real dataset.

8.1. What is clustering and how does it work?

Clustering is a collection of methods for dividing data into groups or clusters. Clusters are roughly described as collections of data objects that are more similar to each other than data objects from other clusters.

There are numerous cases where automatic grouping of data can be very beneficial. Take the case of developing an Internet advertising campaign for a brand new product line that is about to be launched. While we could show a single generic ad to the entire population, a far better approach would be to divide the population into groups of people with common characteristics and interests, and then show individualized ads to each group. **K-means** is an algorithm that finds these groupings in large data sets where manual searching is impossible.

8.2. How is clustering a problem of unsupervised learning?

Imagine a project where you need to predict whether a loan will be approved or not. We aim to predict the loan status based on the customer's gender, marital status, income, and other factors. Such challenges are called **supervised learning** problems when we have a target variable that we need to predict based on a set of predictors or independent variables.

There may be cases where we do not have an outcome variable that we can predict.

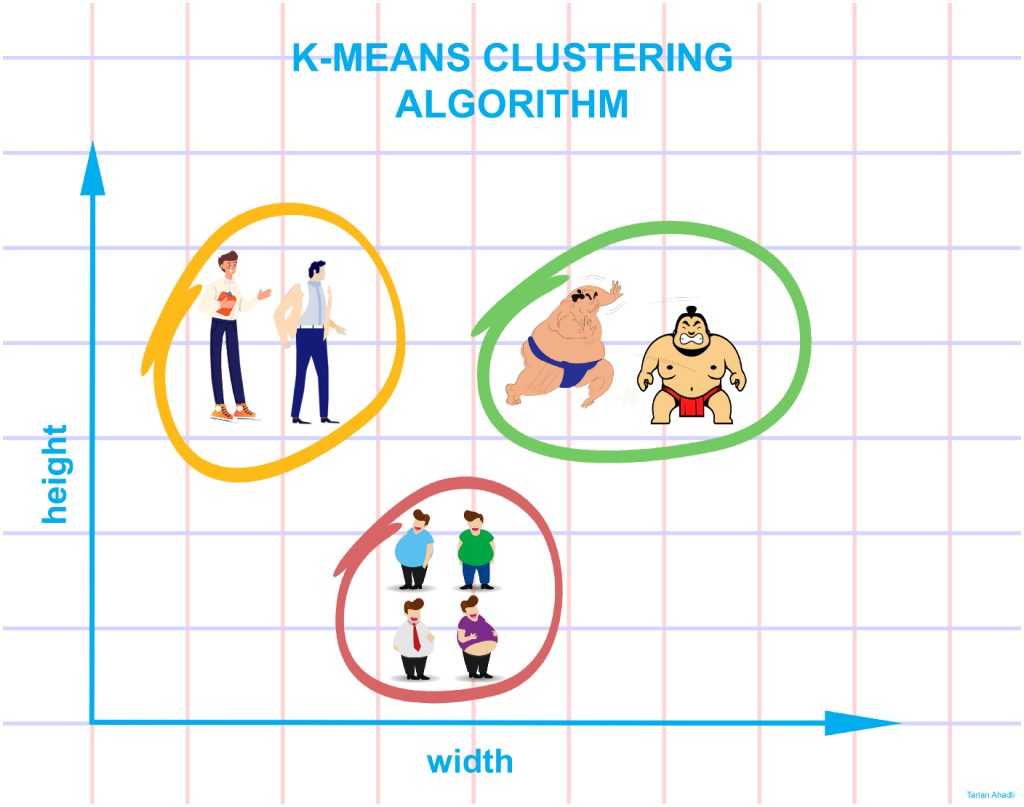
Unsupervised learning problems are those where there is no fixed target variable.

In clustering, we do not have a target that we can predict. We examine the data and try to categorize comparable observations into different groups. Consequently, it is a challenge of unsupervised learning. There are only independent variables and no target/dependent variable.

k-means clustering: image from medium

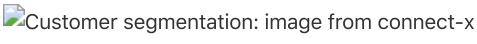
Contents

Print to PDF



8.3. Applications of Clustering

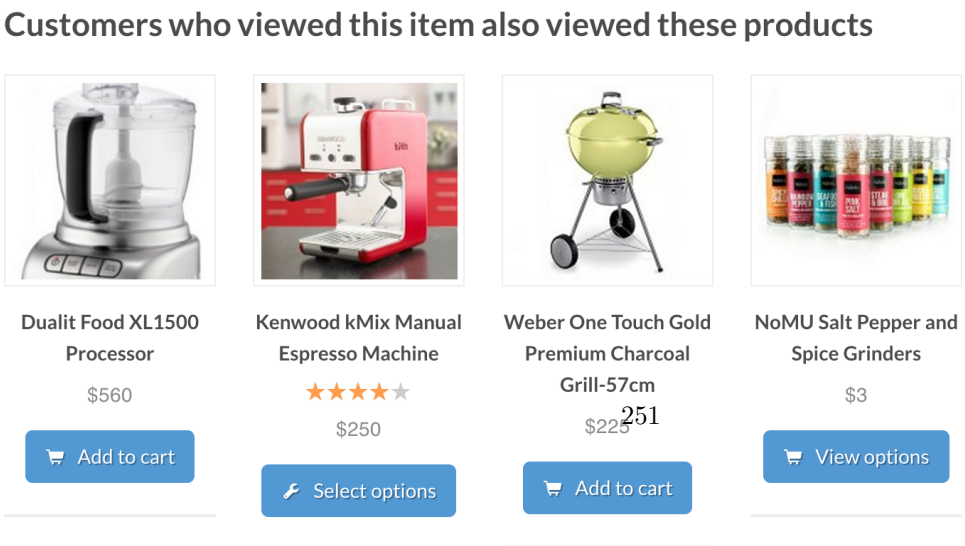
1. **Customer segmentation** is the process of obtaining information about a company's customer base based on their interactions with the company. In most cases, this interaction is about their purchasing habits and behaviors. Businesses can use to create targeted advertising campaigns.



1. **Recommendation engines** The recommendation system is a popular way for offering automatic personalized product, service, and information recommendations.

The recommendation engine, for example, is widely used to recommend products on Amazon, as well as to suggest songs of the same genre on YouTube.

In essence, each cluster will be given to specific preferences based on the preferences of customers in the cluster. Customers would then receive suggestions based on cluster estimates within each cluster.



1. **Medical application** In the medical field, clustering has been used in gene expression experiments. The clustering results identify groupings of people who respond to medical treatments differently.

8.4. The K-means Algorithm: An Overview

This section takes you step by step through the traditional form of the k-means algorithm. This section will help you decide if k-means is the best method to solve your clustering problem.

The traditional k-means algorithm consists of only a few steps.

To begin, we randomly select k centroids, where k is the number of clusters you want to use.

Centroids are data points that represent the center of the cluster.

The main component of the algorithm is based on a **two-step procedure** known as **expectation maximization**.

1. **The expectation step (E-step):** each data point is assigned to the closest centroid.
2. **The maximization step (M-step):** Update the centroids (mean) as being the centre of their respective observation.

We repeat these two steps until the centroid positions do not change (or until there is no further change in the clusters)

Note The "E-step" or "expectation step" is so called because it updates our expectation of which cluster each point belongs to.

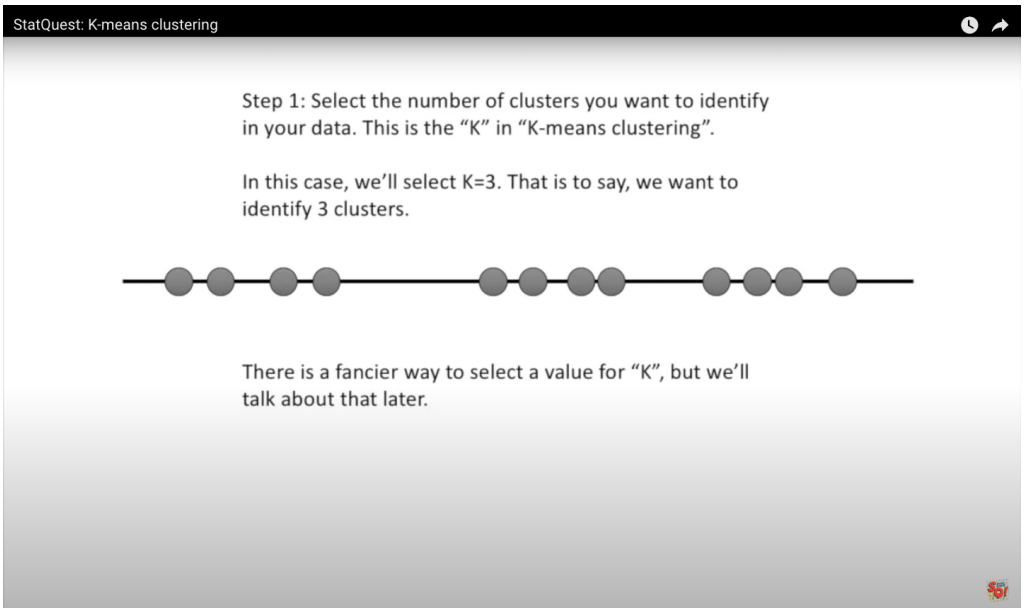
The "M-step" or "maximization step" is so named because it involves maximizing a fitness function that defines the location of the cluster centers - in this case, this maximization is achieved by simply averaging the data in each cluster.

8.5. K-means clustering in 1 dimension: Explained

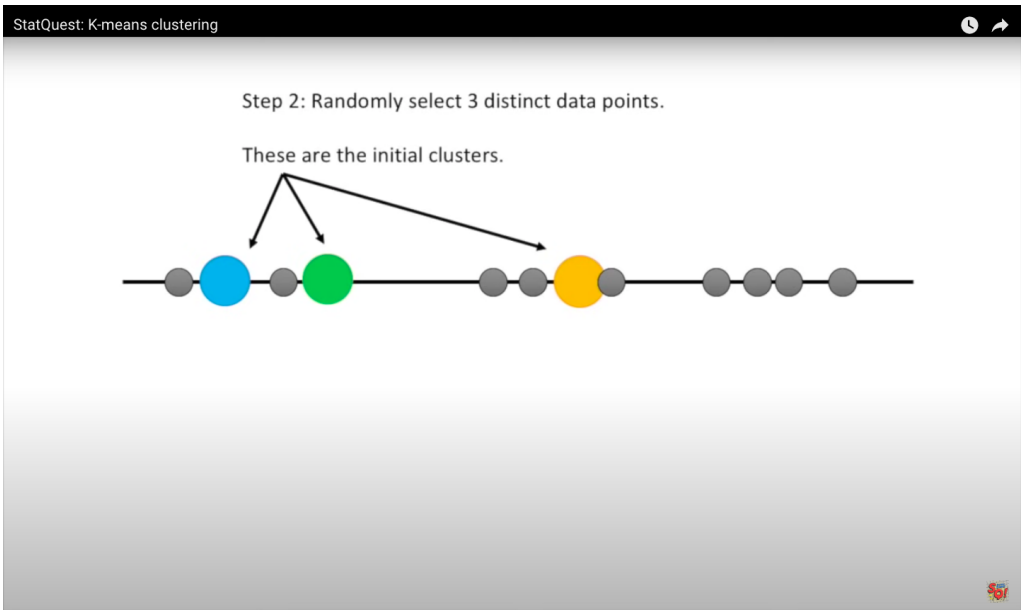
We will learn how to cluster samples that can be put on a line, and the concept can be extended more generally. Imagine that you had some data that you could plot on a line, and you knew that you had to divide it into three clusters. Perhaps they are measurements from three different tumor types or other cell types. In this case, the data yields three relatively obvious clusters, but instead of relying on our eye, we want to see if we can get a computer to identify the same three clusters, using k-means clustering. We start with raw data that we have not yet clustered.

Step one: choose the number of clusters you want to identify in your data - that's the K in k-means clustering. In this case, we choose K to equal three, meaning we want to identify three clusters. There is a more sophisticated way to select a value for K, but we will get into that later.

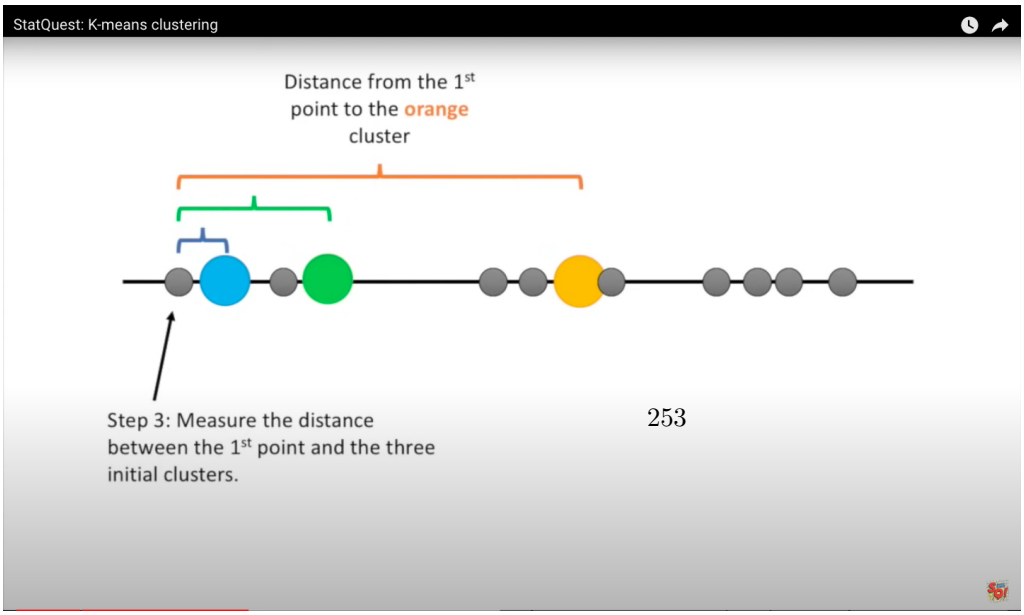
Images below from statquest.



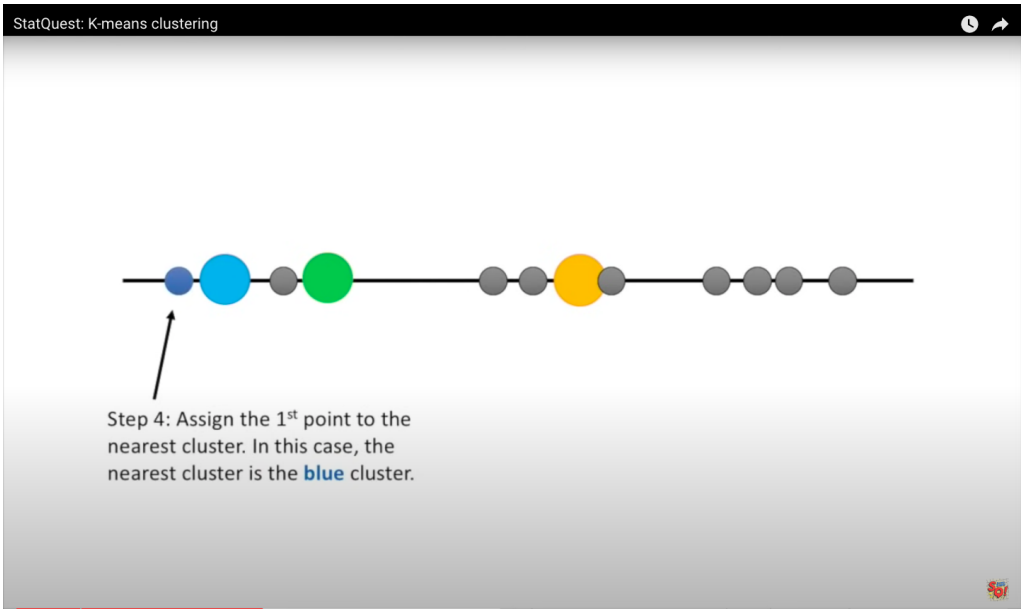
Step 2: Randomly select three unique data points - these are the first clusters.

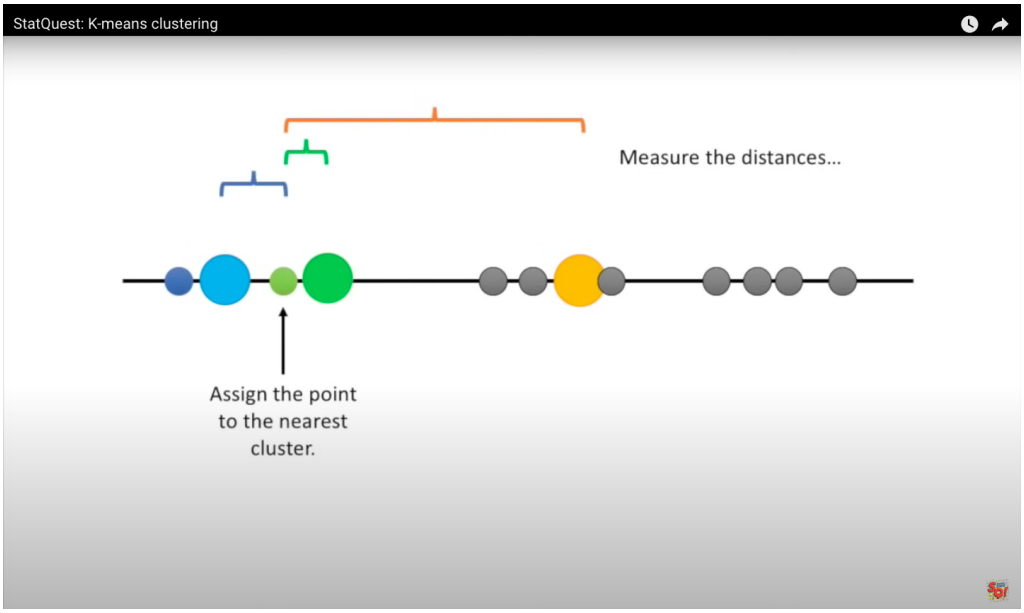


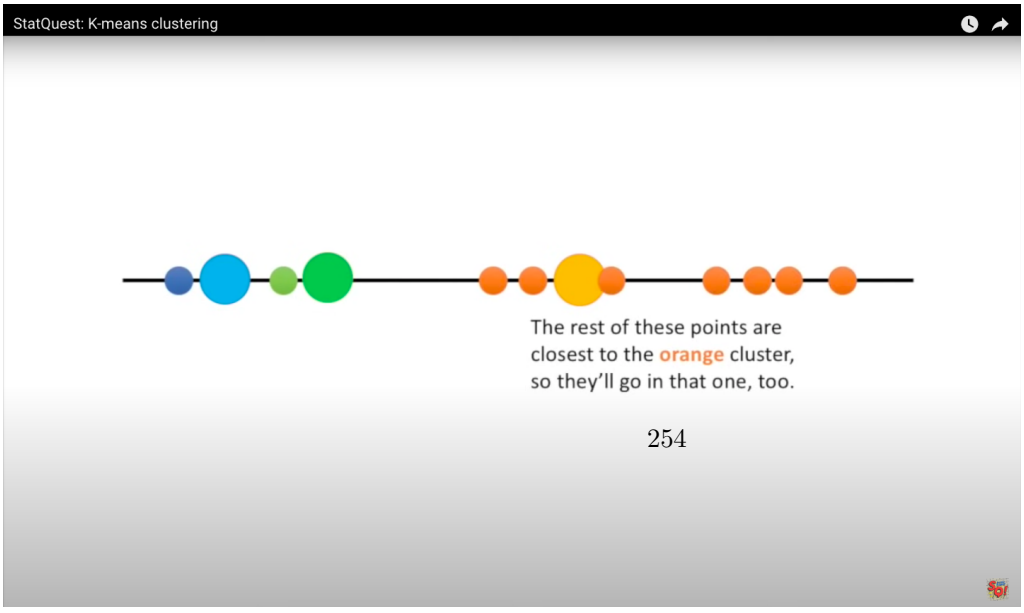
Step 3: Measure the distance between the first point and the three initial clusters this is the distance between the first point and the three clusters.



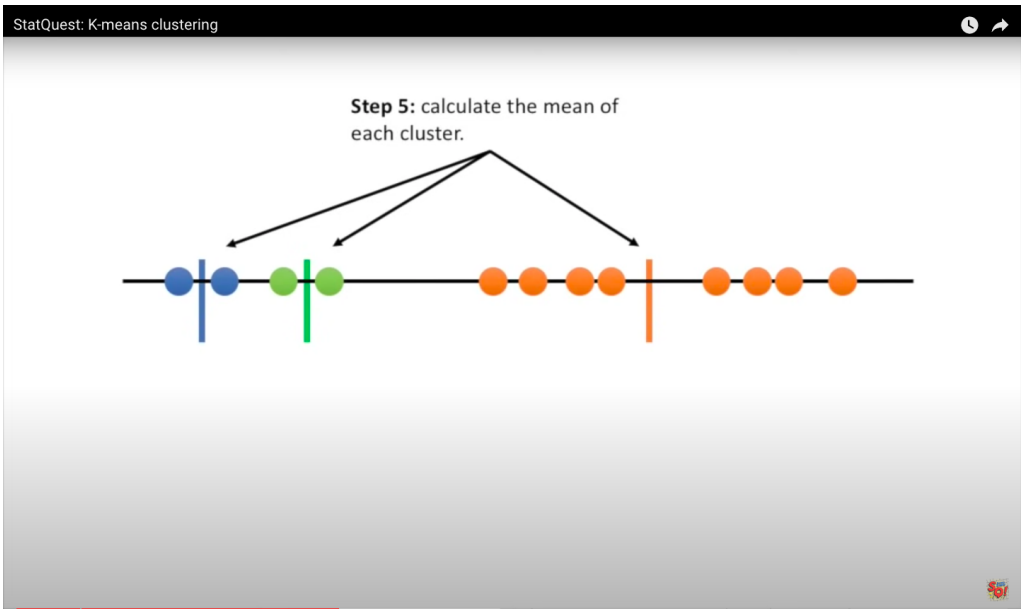
Step 4: Assign the first point to the nearest cluster. In this case, the closest cluster is the blue cluster. Then we do the same for the next point. We measure the distances and then assign the point to the nearest cluster. When all the points are in clusters, we proceed to step 5.



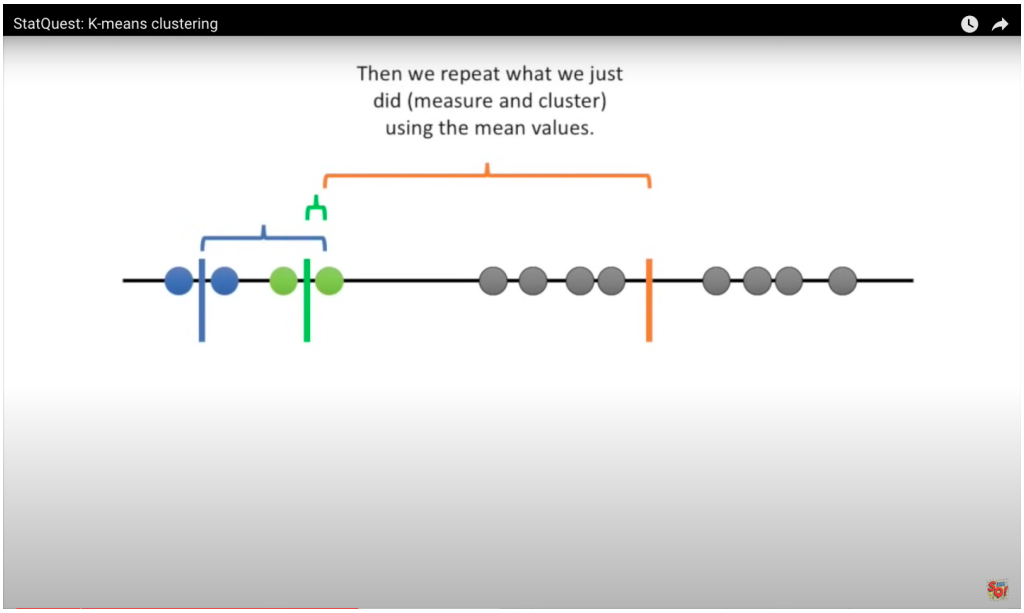




Step 5: We calculate the mean value of each cluster.

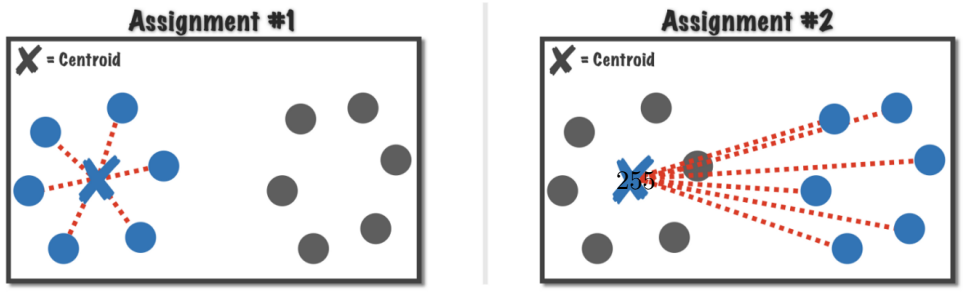


Then we repeat the measurement and clustering based on the mean values. Since the clustering did not change during the last iteration, we are done.



8.5.1. The quality of the cluster assignments

Before going any further, it's important to define what we mean by a good clustering solution. What do we mean when we say "best potential clustering solution"? Consider the following diagram, which depicts two potential observations assignments to the same centroid.



The first assignment is clearly better than the second, but how can we quantify this using the k-means algorithm?

A good clustering solution is the one that minimizes the total sum of these (dotted red) lines, or more formally the sum of squared error (SSE), is the one we are aiming for.

In mathematical language, we want to identify the centroid C that minimizes: given a cluster of observations $x_i \in \{x_1, x_2, \ldots, x_m\}$.

$$J(x) = \sum_{i=1}^{m} ||x_i - C||^2.$$

Since the centroid is simply the center of its respective observations, we can calculate:

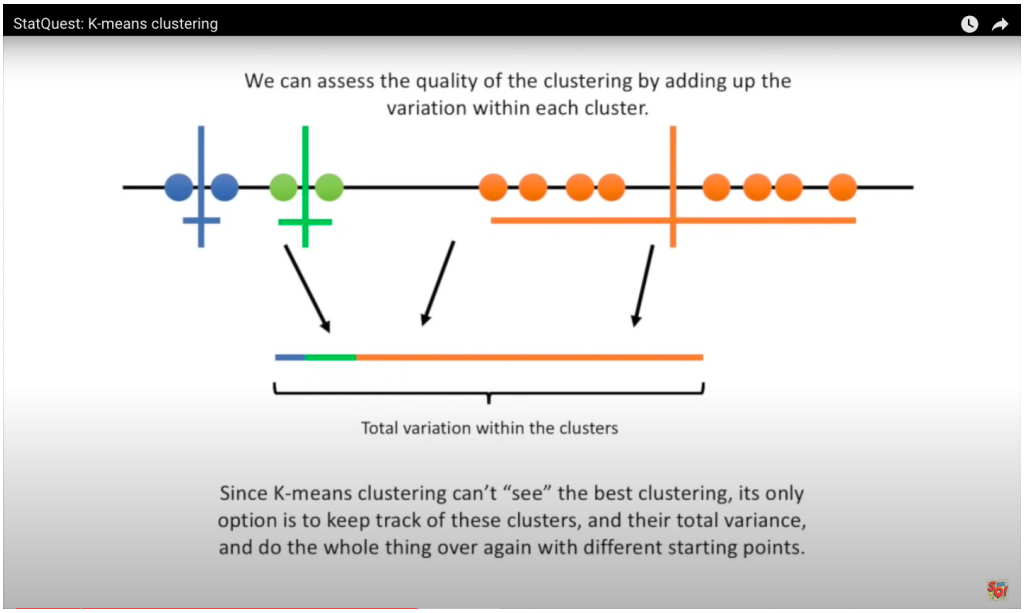
$$C = \frac{\sum_{i=1}^{m} x_i}{m}.$$

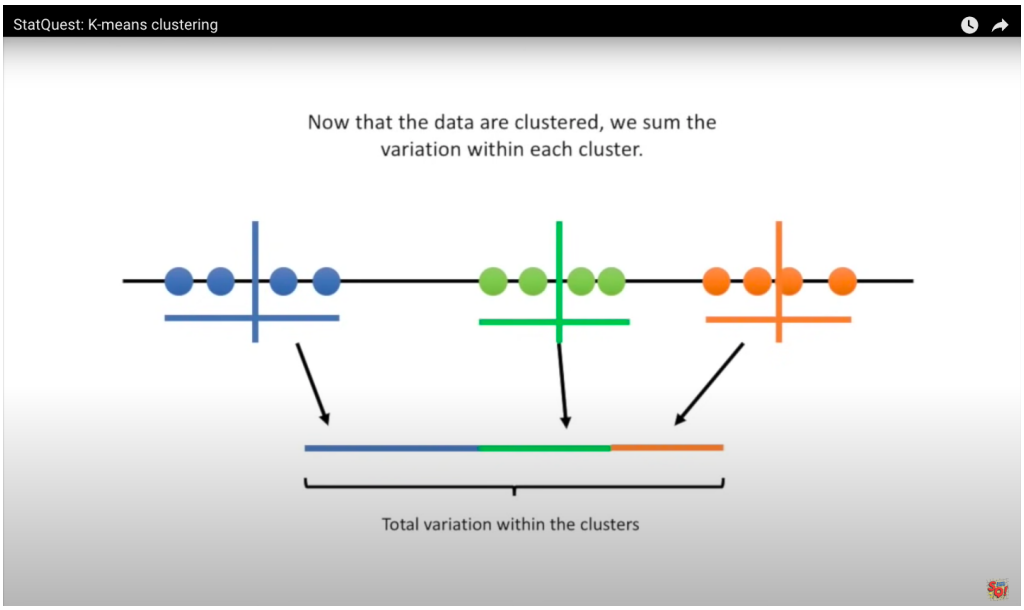
This equation provides us the sum of squared error for a single centroid C, but we really want to minimize the sum of squared errors for all centroids $c_j \in \{c_1, c_2, \ldots, c_k\}$ for all observations $x_i \in \{x_1, x_2, \ldots, x_n\}$ (here n not m). The objective function of k-means is to minimize the total sum of squared error (SST):

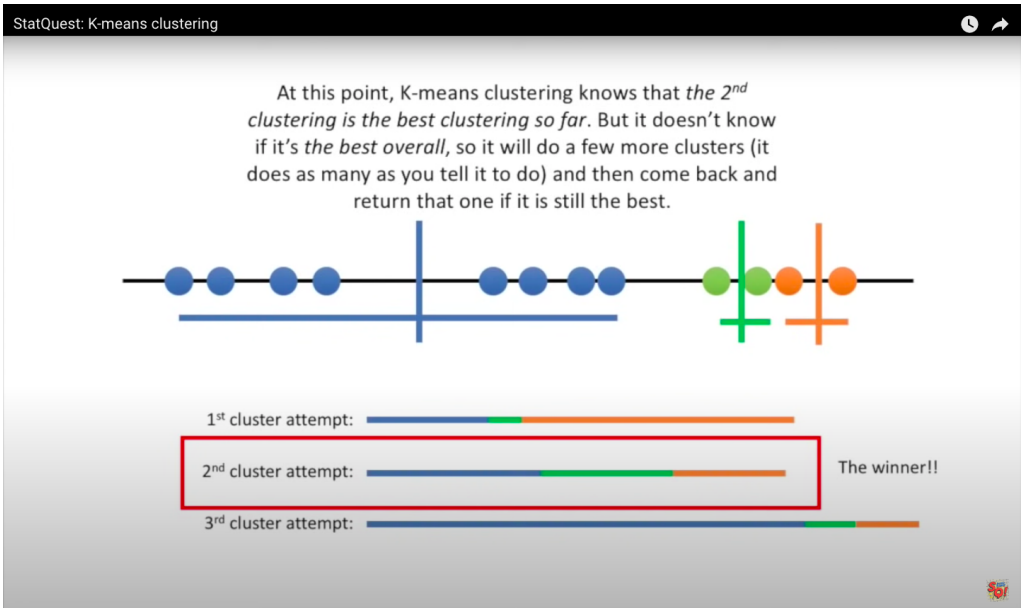
$$J(x) = \sum_{j=1}^{k} \sum_{i=1}^{n} ||x_i^{(j)} - c_j||^2,$$

where the points $x_i^{(j)}$ are assigned to the centriod c_j, the closest centroid .

Note The nondeterministic nature of the k-means algorithm is due to the random initialization stage, which means that cluster assignments will differ if the process is run repeatedly on the same dataset. Researchers frequently execute several initializations of the entire k-means algorithm and pick the cluster assignments from the one with the lowest SSE.







8.6. K-means clustering in Python: a toy example

```
#!pip install kneed
```

```
# Importing Libraries

import matplotlib.pyplot as plt
from kneed import KneeLocator
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler

import numpy as np

import seaborn as sns
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/var/folders/kl/h_r05n_j76n32kt0dwy7kynw0000gn/T/ipykernel_5539/1718519535.py in <module>
      2
      3 import matplotlib.pyplot as plt
----> 4 from kneed import KneeLocator
      5 from sklearn.datasets import make_blobs
      6 from sklearn.cluster import KMeans

ModuleNotFoundError: No module named 'kneed'
```

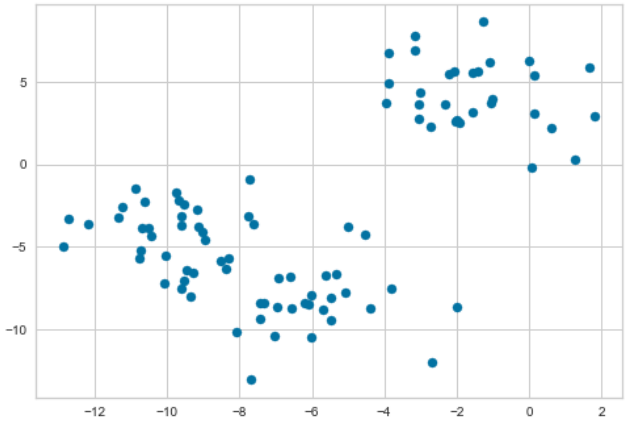
8.6.1. A simple example

Let us get started by using scikit-learn to generate some random data points. It has a `make_blobs()` function that generates a number of gaussian clusters. We will create a two-dimensional dataset containing three distinct blobs. We set `random_state=1` to ensure that this example will always generate the same points for reproducibility.

To emphasize that this is an **unsupervised algorithm**, we will leave the labels out of the visualization.

```
features, true_labels = make_blobs(
    n_samples=90,
    centers=3,
    cluster_std=2,
    random_state=1
)
```

```
plt.scatter(features[:, 0], features[:, 1], s=50);
```

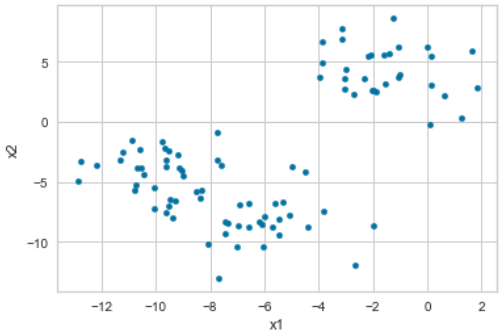


We can also put the **features** into a pandas DataFrame to make working with it easier. We plot it to see how it appears, color-coding each point according to the cluster from which it was produced.

```
%matplotlib inline

import pandas as pd

df = pd.DataFrame(features, columns=["x1", "x2"])
df.plot.scatter("x1", "x2")
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as
value-mapping will have precedence in case its length matches with *x* & *y*.  Please use
the *color* keyword-argument or provide a 2D array with a single row if you intend to
specify the same RGB or RGBA value for all points.
```

```
<AxesSubplot:xlabel='x1', ylabel='x2'>
```




```
#plt.scatter(features[:, 0], features[:, 1], s=50, c=true_labels);
```

Exercise

1. Machine learning algorithms must take all features into account. This means that all feature values must be scaled to the same scale. Perform feature scaling standardization in the dataset.
2. Perform exploratory data analysis (EDA) to analyze and explore data sets and summarize key characteristics, using data visualization methods.

```
df.head()
```

	x1	x2
0	-2.043231	2.631232
1	0.143622	5.411479
2	-9.020676	-4.104492
3	-8.358716	-6.350254
4	-1.021482	3.907749

```
# Make a copy of the original dataset
# df_copy = df.copy()
```

```
scaler = StandardScaler()

df_scaled = df
features = [['x1','x2']]
for feature in features:
    df_scaled[feature] = scaler.fit_transform(df_scaled[feature])
```

```
df_scaled.describe()
```

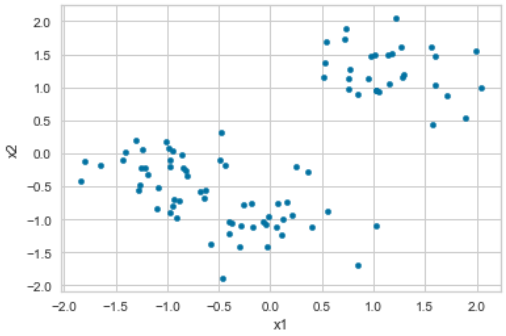
	x1	x2
count	9.000000e+01	9.000000e+01
mean	-5.956655e-16	3.256654e-16
std	1.005602e+00	1.005602e+00
min	-1.835782e+00	-1.890076e+00
25%	-9.043642e-01	-8.222624e-01
50%	-5.623279e-02	-2.030913e-01
75%	8.487898e-01	9.694556e-01
max	2.039518e+00	2.043826e+00

```
df_scaled.plot.scatter("x1", "x2")
```



```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as
value-mapping will have precedence in case its length matches with *x* & *y*.  Please use
the *color* keyword-argument or provide a 2D array with a single row if you intend to
specify the same RGB or RGBA value for all points.
```

```
<AxesSubplot:xlabel='x1', ylabel='x2'>
```



We can see that two or three distinct clusters. This is a great situation to apply k-means clustering in, and the results will be highly valuable.

To facilitate the analysis of tabular data, we can use Pandas in Python to convert the data into a more computer-friendly form.

`Pandas.melt()` converts a DataFrame from wide format to long format. The melt() function is useful to convert a DataFrame to a format where one or more columns are identifier variables, while all other columns considered as measurement variables are subtracted from the row axis, leaving only two non-identifier columns, variable and value.

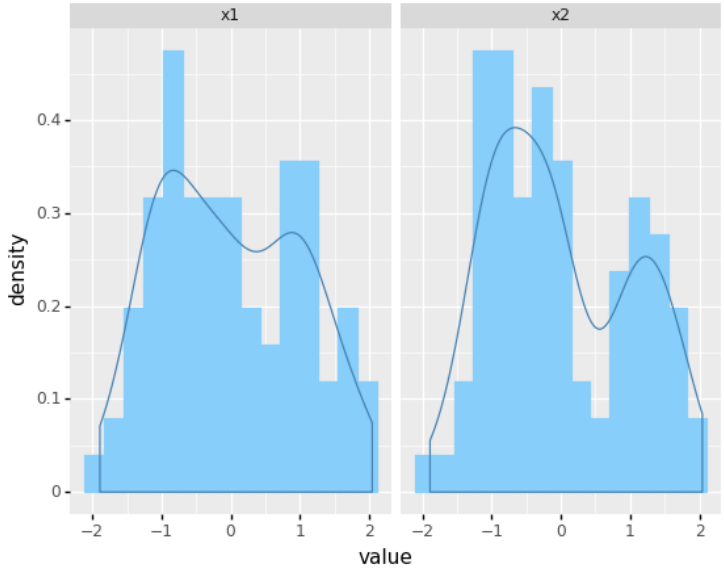
```
df_melted = pd.melt(df_scaled[['x1','x2']], var_name = 'features',value_name = 'value')
```

```
df_melted.head()
```

	features	value
0	x1	1.019528
1	x1	1.595412
2	x1	-0.817907
3	x1	-0.643587
4	x1	1.288594

```
from plotnine import *
```

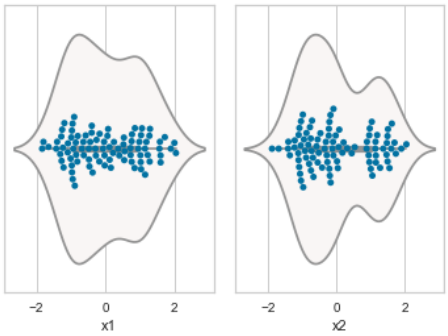
```
(
    ggplot(df_melted) + aes('value') + geom_histogram(aes(y=after_stat('density')),
color = 'lightskyblue', fill = 'lightskyblue', bins = 15)
    + geom_density(aes(y=after_stat('density')), color = 'steelblue')
    + facet_wrap('features')
)
```




```
<ggplot: (304892637)>
```

8.6.2. Violin and swarm plots

```
n = 0
for cols in ['x1' , 'x2']:
    n += 1
    plt.subplot(1 , 2 , n)
    plt.subplots_adjust(hspace = 0.5 , wspace = 0.1)
    sns.violinplot(x = cols, data = df , palette = 'vlag')
    sns.swarmplot(x = cols, data = df)
    #plt.ylabel('Gender' if n == 1 else '')
    #plt.title('Boxplots & Swarmplots' if n == 2 else '')
plt.show()
```



8.7. Clustering using K- means

The data is now ready for clustering. Before fitting the estimator to the data, you set the algorithm parameters in the KMeans estimator class in scikit-learn. The scikit-learn implementation is adaptable, with various adjustable parameters.

The following are the parameters that were utilized in this example.

- **init** The initialization method is controlled by `init`. Setting init to "random" implements the normal version of the k-means algorithm. Setting this to "k-means++" uses a more advanced technique to speed up convergence, which you'll learn about later.
- **n_clusters** For the clustering step, `n_clusters` sets k. For k-means, this is the most essential parameter.
- **n_init** The number of initializations to do is specified by `n_init`. Because two runs can converge on different cluster allocations, this is critical. The scikit-learn algorithm defaults to running 10 k-means runs and returning the results of the one with the lowest SSE.

- **max_iter** For each initialization of the k-means algorithm, `max_iter` specifies the maximum number of iterations.

First we create a new instance of the KMeans class with the following parameters. Then passing the data, we want to fit to the `fit()` method, the algorithm will actually run. You can use nested lists, numpy arrays (as long as they have the shape (nsample,nfeatures) or Pandas DataFrames.

```
kmeans = KMeans(
                init="random",
                n_clusters=3,
                n_init=10,
                max_iter=300,
                random_state=88)

kmeans.fit(df_scaled)
```

```
KMeans(init='random', n_clusters=3, random_state=88)
```

Now that we have calculated the cluster centres, we can use the `cluster_centers_` data attribute of our model to see which clusters it has chosen.

```
centroids = kmeans.cluster_centers_
print('Final centroids:\n', centroids)
```

```
Final centroids:
 [[ 0.0235795  -1.04710637]
 [-1.03913289 -0.30604602]
 [ 1.15724837  1.25434435]]
```

The sum of the squared distances of data points to their closest cluster center is another data characteristic of the model, which can be obtained by using the attribute `inertia_`. This property is used by the algorithm to determine whether it has converged. In general, a lower number indicates a better match.

```
print('The sum of the squared distrance:\n', kmeans.inertia_)
```

```
The sum of the squared distrance:
 24.20252704670535
```

The cluster assignments are stored as a one-dimensional NumPy array in kmeans.labels_ or using the function `predict`.

```
pred = kmeans.predict(df_scaled)
print('predicted labels:\n',pred[0:5])
```

```
predicted labels:
 [2 2 1 1 2]
```

```
#labels = kmeans.labels_
```

```
print('predicted labels:\n',kmeans.labels_[0:5])
```

```
predicted labels:
 [2 2 1 1 2]
```

```
df_scaled['cluster'] = pred
print('the number of samples in each cluster:\n', df_scaled['cluster'].value_counts())
```

```
the number of samples in each cluster:
 1    34
2    30
0    26
Name: cluster, dtype: int64
```

```
df_scaled.drop('cluster',axis='columns', inplace=True)
```

8.8. Choosing the appropriate number of clusters: Elbow Method

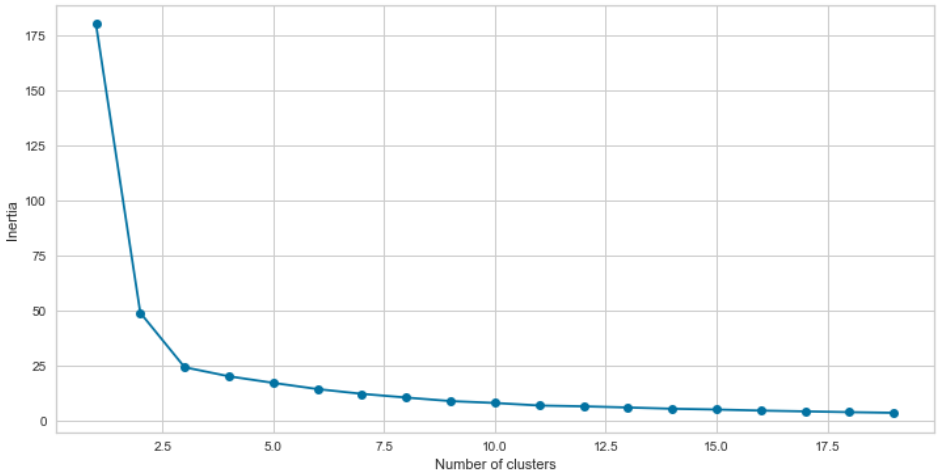
The **elbow method** is a widely used approach to determine the proper number of clusters.

To perform the elbow method, we run many k-means by increasing k at each iteration and plot the SST (also known as **intetia**) as a function of the number of clusters. We will observe that it continues to decrease as we increase k. The distance between each point and its nearest centroid will decrease as more centroids are added.

The **elbow point** is a position on the SSE curve when **it begins to bend**. This point is believed to be a good compromise between error and cluster count. From the figure below, the elbow is positioned at k=3 in our example.

```
# fitting multiple k-means algorithms and storing the values in an empty list
SSE = []
for cluster in range(1,20):
    kmeans = KMeans(n_clusters = cluster)
    kmeans.fit(df_scaled)
    SSE.append(kmeans.inertia_)

# converting the results into a dataframe and plotting them
frame = pd.DataFrame({'Cluster':range(1,20), 'SSE':SSE})
plt.figure(figsize=(12,6))
plt.plot(frame['Cluster'], frame['SSE'], marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
```

```
Text(0, 0.5, 'Inertia')
```



From the figure, the elbow is positioned at k=3 in our example where the curve starts to bend.

8.9. Ploting the cluster boundary and clusters

To plot the cluster boundary and the clusters, we run the Code below, which is modified from https://www.kaggle.com/code/irfanasrullah/customer-segmentation-analysis/notebook

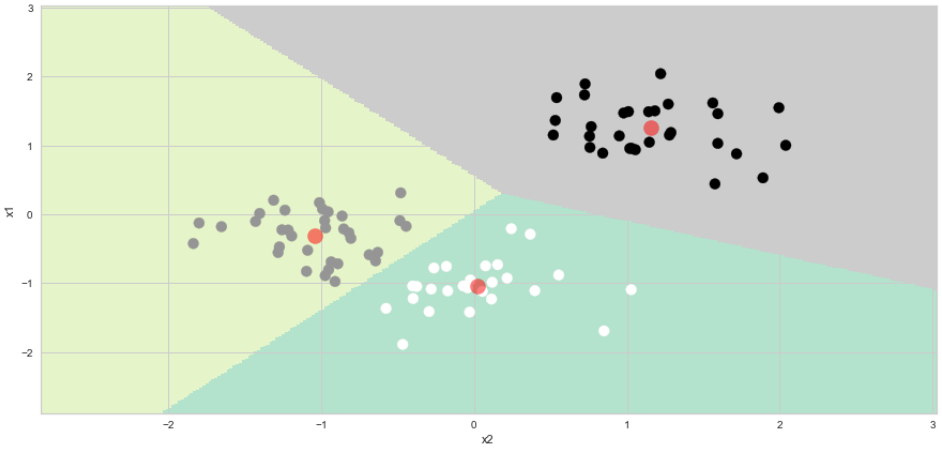
```
# Using n_clusters = 3 as suggested by the elbow method

kmeans = KMeans(
                init="random",
                n_clusters=3,
                n_init=10,
                max_iter=300,
                random_state=88)

kmeans.fit(df_scaled)
```



```
KMeans(init='random', n_clusters=3, random_state=88)
```

```
X1 = df_scaled[['x1','x2']].values

h = 0.02
x_min, x_max = X1[:, 0].min() - 1, X1[:, 0].max() + 1
y_min, y_max = X1[:, 1].min() - 1, X1[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
##Z = kmeans.predict(np.c_[xx.ravel(), yy.ravel()])

meshed_points = pd.DataFrame(np.c_[xx.ravel(), yy.ravel()], columns = ['x1','x2'])
Z = kmeans.predict(meshed_points)
```

```
plt.figure(1 , figsize = (15 , 7) )
plt.clf()
Z = Z.reshape(xx.shape)
plt.imshow(Z , interpolation='nearest',
           extent=(xx.min(), xx.max(), yy.min(), yy.max()),
           cmap = plt.cm.Pastel2, aspect = 'auto', origin='lower')

plt.scatter( x = 'x1' ,y = 'x2' , data = df_scaled , c = pred , s = 100 )
plt.scatter(x = centroids[: , 0] , y =  centroids[: , 1] , s = 200 , c = 'red' , alpha =
0.5)
plt.ylabel('x1') , plt.xlabel('x2')
plt.show()
```



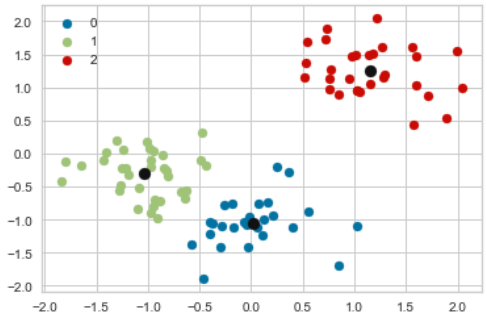

```
#np.c_[xx.ravel(), yy.ravel()]
```

```
#np.c_[xx.ravel(), yy.ravel()][:,0]
```

```
#pd.DataFrame(np.c_[xx.ravel(), yy.ravel()], columns = ['x1','x2'])
```

Alternatively, one can run the following Python code: (modified from https://www.askpython.com/python/examples/plot-k-means-clusters-python)

```
#Getting the Centroids
centroids = kmeans.cluster_centers_
u_labels = np.unique(kmeans.predict(df_scaled.iloc[:,0:2]))

#plotting the results:

for i in u_labels:
    plt.scatter(df_scaled.iloc[pred == i , 0] , df_scaled.iloc[pred == i , 1] , label =
i)

plt.scatter(centroids[:,0] , centroids[:,1] , s = 80, color = 'k')
plt.legend()
plt.show()
```




```
#u_labels
```

```
#df_scaled.iloc[pred == 0 , 0]
#pred == 0
```

8.10. Choosing the appropriate number of clusters: Silhouette Method

In K-Means clustering, the number of clusters (k) is the most crucial hyperparameter. If we already know how many clusters we want to group the data into, tuning the value of k is unnecessary.

The **silhouette method** is also used to determine the optimal number of clusters, as well as to interpret and validate consistency within data clusters.

The silhouette method calculates each point's silhouette coefficients, which measure how well a data point fits into its assigned cluster based on two factors:

- How close the data point is to other points in the cluster.
- How far the data point is from points in other clusters.

This method also displays a clear graphical depiction of how effectively each object has been classified

The silhouette value is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation).

8.10.1. Steps to find the silhouette score

To find the silhouette coefficient of each point, follow these steps:

1. For data point i in the cluster C_I, we calculate the mean distance between i and all other data points in the same cluster

$$a(i) = \frac{1}{|C_I| - 1} \sum_{j \in C_I, i \neq j} d(i, j),$$

This $a(i)$ measures how well i is assigned to its cluster (the smaller the value, the better the assignment).

1. We then calculate the mean dissimilarity of point i to its **neighboring cluster**.

$$b(i) = \min_{J \neq I} \frac{1}{|C_J|} \sum_{j \in C_J} d(i, j)$$

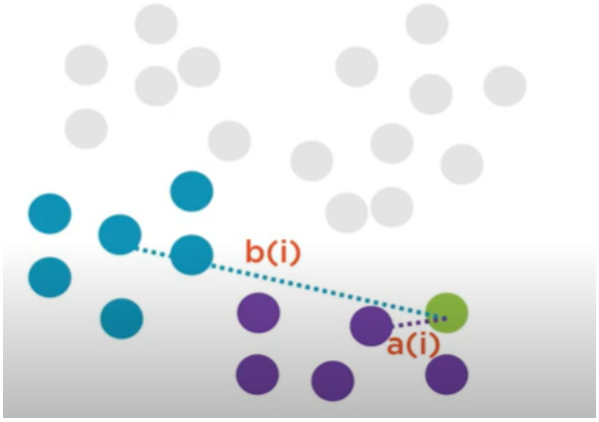
1. We now define a silhouette coefficient of one data point i

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \quad 265$$

1. To determine the **silhouette score**, compute silhouette coefficients for each point and average them over all samples.

The silhouette value varies from [1, -1], with a high value indicating that the object is well matched to its own cluster but poorly matched to nearby clusters.

- The clustering setup is useful if the majority of the objects have a high value.
- The clustering setup may have too many or too few clusters if many points have a low (the sample is very close to the neighboring clusters) or negative value (the sample is assigned to the wrong clusters).



8.10.1.1. Silhouette Visualizer

The **Silhouette Visualizer** visualizes which clusters are dense and which are not by displaying the silhouette coefficient for each sample on a per-cluster basis. This is especially useful for determining cluster imbalance or comparing different visualizers to determine a K value.

We will import the **Yellowbrik** library for silhoutte analsis, an extension to the Scikit-Learn API that simplifies model selection and hyperparameter tuning.

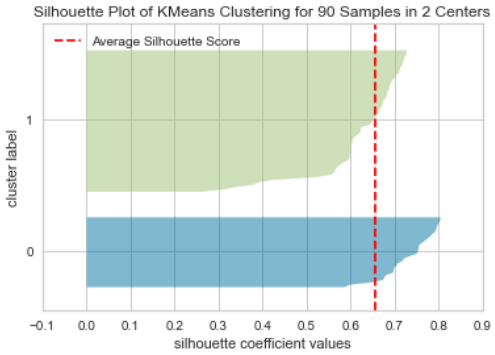
For more details about the library, please refer to https://www.scikit-yb.org/en/latest/.

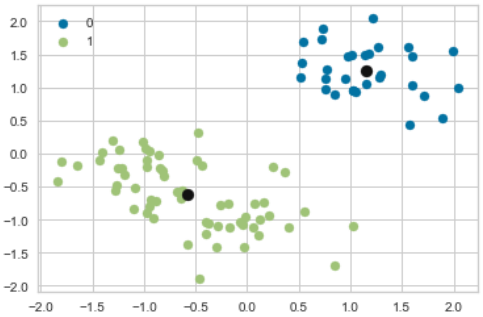
The following results compares different visualizers with n_cluster = 2, 3 and 4.

```
from yellowbrick.cluster import SilhouetteVisualizer
```

```
# Instantiate the clustering model and visualizer

kmeans = KMeans(
                init="random",
                n_clusters=2,
                n_init=10,
                max_iter=300,
                random_state=88)

visualizer = SilhouetteVisualizer(kmeans, colors='yellowbrick')
# Fit the data to the visualizer
visualizer.fit(df_scaled)
# Compute silhouette_score
print('The silhouette score:', visualizer.silhouette_score_)
# Finalize and render the figure
visualizer.show()



# For scatter plot
kmeans.fit(df_scaled)
pred2 = kmeans.predict(df_scaled)
#Getting the Centroids
centroids = kmeans.cluster_centers_
u_labels = np.unique(kmeans.predict(df_scaled.iloc[:,0:2]))

#plotting the results:
for i in u_labels:
    plt.scatter(df_scaled.iloc[pred2 == i , 0] , df_scaled.iloc[pred2 == i , 1] , label
= i)

plt.scatter(centroids[:,0] , centroids[:,1] , s = 80, color = 'k')
plt.legend()
plt.show()
```

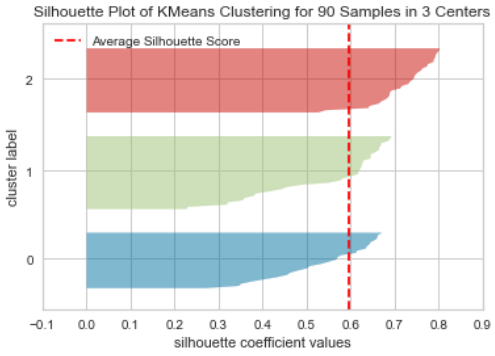
```
The silhouette score: 0.6558340654970938
```

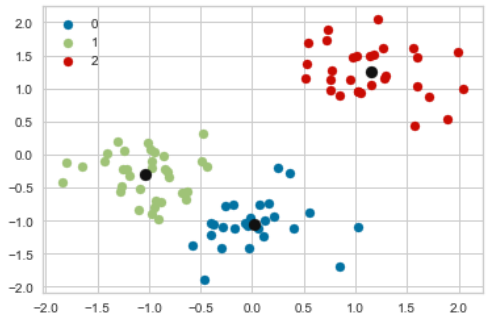





```
# Instantiate the clustering model and visualizer

kmeans = KMeans(
                init="random",
                n_clusters=3,
                n_init=10,
                max_iter=300,
                random_state=88)

visualizer = SilhouetteVisualizer(kmeans, colors='yellowbrick')
# Fit the data to the visualizer
visualizer.fit(df_scaled)
# Compute silhouette_score
print('The silhouette score:', visualizer.silhouette_score_)
# Finalize and render the figure
visualizer.show()



# For scatter plot
kmeans.fit(df_scaled)
pred3 = kmeans.predict(df_scaled)
#Getting the Centroids
centroids = kmeans.cluster_centers_
u_labels = np.unique(kmeans.predict(df_scaled.iloc[:,0:2]))

#plotting the results:
for i in u_labels:
    plt.scatter(df_scaled.iloc[pred3 == i , 0] , df_scaled.iloc[pred3 == i , 1] , label
= i)

plt.scatter(centroids[:,0] , centroids[:,1] , s = 80, color = 'k')
plt.legend()
plt.show()
```

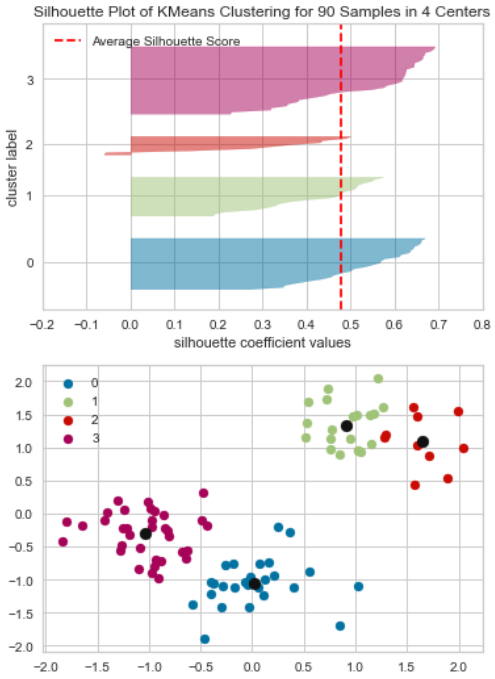
```
The silhouette score: 0.5965646018263415
```






```
# Instantiate the clustering model and visualizer

kmeans = KMeans(
                init="random",
                n_clusters=4,
                n_init=10,
                max_iter=300,
                random_state=88)


visualizer = SilhouetteVisualizer(kmeans, colors='yellowbrick')
# Fit the data to the visualizer
visualizer.fit(df_scaled)
# Compute silhouette_score
print('The silhouette score:', visualizer.silhouette_score_)
# Finalize and render the figure
visualizer.show()



# For scatter plot
kmeans.fit(df_scaled)
pred4 = kmeans.predict(df_scaled)

#Getting the Centroids
centroids = kmeans.cluster_centers_
u_labels = np.unique(kmeans.predict(df_scaled.iloc[:,0:2]))

#plotting the results:
for i in u_labels:
    plt.scatter(df_scaled.iloc[pred4 == i , 0] , df_scaled.iloc[pred4 == i , 1] , label
= i)

plt.scatter(centroids[:,0] , centroids[:,1] , s = 80, color = 'k')
plt.legend()
plt.show()
```

```
The silhouette score: 0.47977223451628676
```



Based on the above results, we observe the following:

- At **n_clusters=4**, the silhouette plot shows that the n_cluster value of 4 is a poor choice because most points in the cluster with cluster_label=2 have below average silhouette values.
- At **n_clusters=2**, the thickness of the silhouette graph for the cluster with cluster_label=1 is larger due to the grouping of the 2 sub-clusters into one large cluster. Moreover, n_clusters=2 has the best average silhouette score of around 0.65, and all clusters are above the average, indicating that it is a good choice.
- For **n_clusters=3**, all plots are more or less similar in thickness and therefore similar in size, which can be considered the one of the optimal values.
- Silhouette analysis is more inconclusive in deciding between 2 and 3.

The bottom line is that **good n clusters** will have a silhouette average score far above 0.5, and all of the clusters will have a score higher than the average.

8.10.2. Evaluating Clustering Performance Using Advanced Techniques: Adjusted Rand Index

Without the use of ground truth labels, the elbow technique and silhouette coefficient evaluate clustering performance.

Ground truth labels divide data points into categories based on a human's or an algorithm's classification. When used without context, these measures do their best to imply the correct number of clusters, but they can be misleading.

Note that datasets with ground truth labels are uncommon in practice.

8.10.2.1. Adjusted Rand Index

Because the ground truth labels are known, a clustering metric that considers labels in its evaluation can be used. The scikit-learn version of a standard metric known as the **adjusted rand index (ARI)** can be used.

The Rand Index computes a similarity measure between two clusterings by evaluating all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusterings,

The Rand index has been adjusted to account for chance. For more information, please refer to https://towardsdatascience.com/performance-metrics-in-machine-learning-part-3-clustering-d69550662dc6.

The output values of the ARI vary from -1 to 1. A score around 0 denotes random assignments, whereas a score near 1 denotes precisely identified clusters.

```
from sklearn.metrics import adjusted_rand_score
```

Compare the clustering results with n_clusters= 2 and 3 using ARI as the performance metric.

```
print('ARI with n_clusters=2:',adjusted_rand_score(true_labels, pred2))
print('ARI with n_clusters=3:',adjusted_rand_score(true_labels, pred3))
```

```
ARI with n_clusters=2: 0.5658536585365853
ARI with n_clusters=3: 0.8164434485919437
```

The silhouette coefficient was misleading, as evidenced by the above output. In comparison to k-means, using 3 clusters is the better solution for the dataset.

The quality of clustering algorithms can also be assessed using a variety of measures. See the following link for more details: https://scikit-learn.org/stable/modules/model_evaluation.html

Final Remarks: Challenges with the K-Means Clustering Algorithm

One of the most common challenges when working with K-Means is that the size of the clusters varies. See https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-k-means-clustering/.

Another challenge with K-Means is that the densities of the original points are different. Let us assume these are the original points:

One of the solutions is to use a higher number of clusters. So, in all the above scenarios, instead of using 3 clusters, we can use a larger number. Perhaps setting k=10 will lead to more meaningful clusters.

```
centroids
```

```
array([[ 0.0235795 , -1.04710637],
       [ 0.90891951,  1.33856737],
       [ 1.65390611,  1.08589831],
       [-1.03913289, -0.30604602]])
```

```
pred = kmeans.predict(df_scaled)
df_scaled['cluster'] = pred
print('the number of samples in each cluster:\n', df_scaled['cluster'].value_counts())
```

```
the number of samples in each cluster:
 3    34
0    26
1    20
2    10
Name: cluster, dtype: int64
```

```
df_scaled.drop('cluster',axis='columns', inplace=True)
```

```
visualizer.silhouette_score_
```

```
0.47977223451628676
```

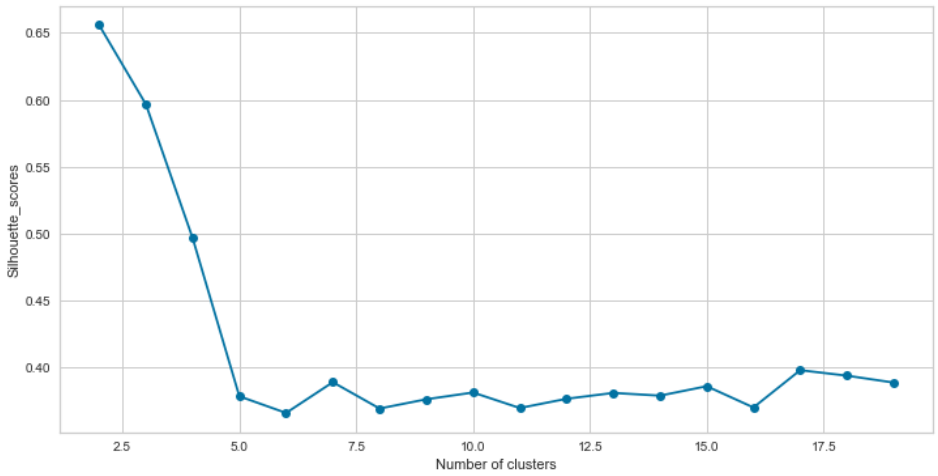
```
#silhouette_score(X, cluster_labels)
silhouette_score(df_scaled, kmeans.labels_)
```



```
0.47977223451628676
```

```
# fitting multiple k-means algorithms and storing the values in an empty list
silhouette_scores = []
for cluster in range(2,20):
    kmeans = KMeans(n_clusters = cluster)
    kmeans.fit(df_scaled)
    silhouette_scores.append(silhouette_score(df_scaled, kmeans.labels_)

)

# converting the results into a dataframe and plotting them
frame = pd.DataFrame({'Cluster':range(2,20), 'Silhouette_scores':silhouette_scores})
plt.figure(figsize=(12,6))
plt.plot(frame['Cluster'], frame['Silhouette_scores'], marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Silhouette_scores')
```

```
Text(0, 0.5, 'Silhouette_scores')
```



8.11. Customer segmentation

Customer segmentation is the process of categorizing your customers based on a variety of factors. These can be personal information,

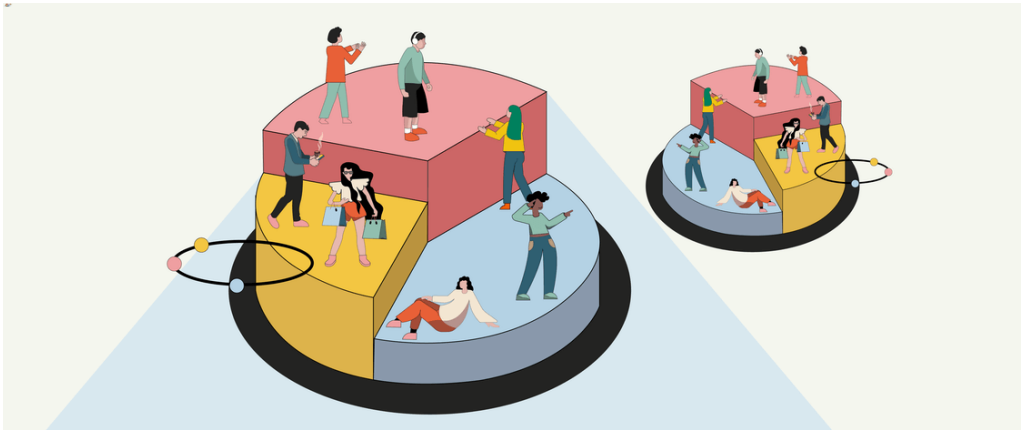
- buying habits,
- demographics and
- so on.

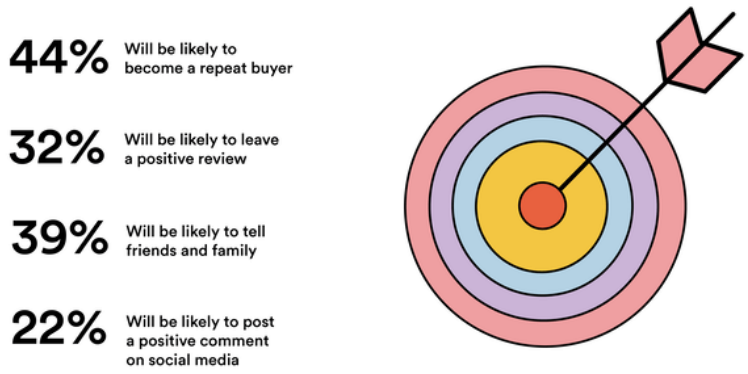
The goal of customer segmentation is to gain a better understanding of each group so you can market and promote your brand more effectively.

To understand your consumer persona, you may need to use a technique to achieve your goals. Customer segmentation can be achieved in a number of ways. One is to develop a set of machine learning algorithms. In the context of customer segmentation, this article focuses on the differences between

- the kmeans and
- knn algorithms.

Customer Segmentation: image from segmentify





8.11.1. Mall Customer Data

Mall Customer Segmentation Data is a dataset from Kaggle that contains the following information:

- individual unique customer IDs,
- a categorical variable in the form of gender, and
- three columns of age, annual income, and spending level.

These numeric variables are our main targets for identifying patterns in customer buying and spending behaviour.

The data can be downloaded from https://www.kaggle.com/datasets/vjchoudhary7/customer-segmentation-tutorial-in-python

```
# Importing Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler

import seaborn as sns
```

```
from plotnine import *
```



```
url = 'https://raw.githubusercontent.com/pairote-sat/SCMA248/main/Data/Mall_Customers.csv'

df = pd.read_csv(url)
```

```
df.head()
```

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

The dataset contains 200 observations and 5 variables. However, below is a description of each variable:

- CustomerID = Unique ID, assigned to the customer.
- Gender = Gender of the customer
- Age = Age of the customer
- Annual Income = (k$) annual income of the customer
- Spending Score = (1-100) score assigned by the mall based on customer behavior and spending type

8.11.2. Exploratory Data Analysis

Exercise

1. Perform exploratory data analysis to understand the data set before modeling it, which includes:

- Observe the data set (e.g., the size of the data set, including the number of rows and columns),
- Find any missing values,
- Categorize the values to determine which statistical and visualization methods can work with your data set,
- Find the shape of your data set, etc.

1. Perform feature scaling standardization in the data set when preprocessing the data for the K-Means algorithm.
2. Implement the K-Means algorithm on the annual income and spending score variables.

- Determine the optimal number of K based on the elbow method or the silhouette method.
- Plot the cluster boundary and clusters.

1. Based on the optimal value of K, create a summary by averaging the age, annual income, and spending score for each cluster. Explain the main characteristics of each cluster.
2. (Optional) Implement the K-Means algorithm on the variables annual income, expenditure score, and age. Determine the optimal number of K and visualize the results (by creating a 3D plot).

```
df.describe()
```

	CustomerID	Age	Annual Income (k$)	Spending Score (1-100)
count	200.000000	200.000000	200.000000	200.000000
mean	100.500000	38.850000	60.560000	50.200000
std	57.879185	13.969007	26.264721	25.823522
min	1.000000	18.000000	15.000000	1.000000
25%	50.750000	28.750000	41.500000	34.750000
50%	100.500000	36.000000	61.500000	50.000000
75%	150.250000	49.000000	78.000000	73.000000
max	200.000000	70.000000	137.000000	99.000000

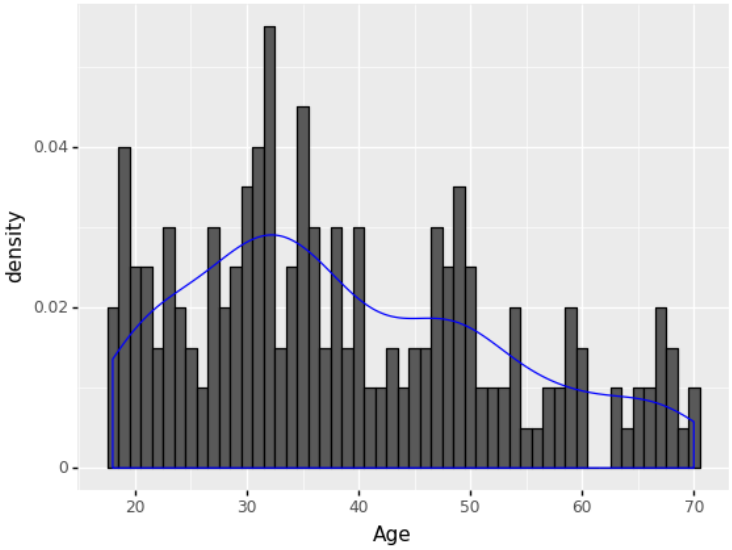

```
df.dtypes
```

```
CustomerID                int64
Gender                   object
Age                       int64
Annual Income (k$)        int64
Spending Score (1-100)    int64
dtype: object
```

```
df.isnull().sum()
```

```
CustomerID              0
Gender                  0
Age                     0
Annual Income (k$)      0
Spending Score (1-100)  0
dtype: int64
```

```
(
    ggplot(df) +
    aes('Age') +
    geom_histogram(aes(y=after_stat('density')), binwidth = 1, color = 'black') +
    geom_density(aes(y=after_stat('density')),color='blue')
)
```




```
<ggplot: (306017193)>
```

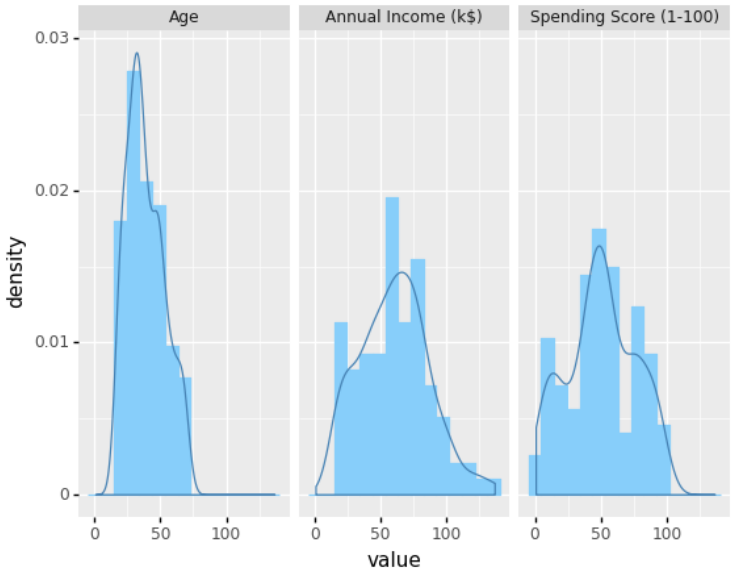
```
df_melted = pd.melt(df[['Age','Annual Income (k$)','Spending Score (1-100)']], var_name
= 'features',value_name = 'value')
```

```
df_melted
```

	features	value
0	Age	19
1	Age	21
2	Age	20
3	Age	23
4	Age	31
...	...	...
595	Spending Score (1-100)	79
596	Spending Score (1-100)	28
597	Spending Score (1-100)	74
598	Spending Score (1-100)	18
599	Spending Score (1-100)	83

600 rows × 2 columns

```
(
    ggplot(df_melted) + aes('value') + geom_histogram(aes(y=after_stat('density')),
color = 'lightskyblue', fill = 'lightskyblue', bins = 15)
    + geom_density(aes(y=after_stat('density')), color = 'steelblue')
    + facet_wrap('features')
)
```




```
<ggplot: (305720949)>
```

```
# Make a copy of the original dataset
df_model = df.copy()

# Data scaling

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

df_scaled = df_model
features = [['Age','Annual Income (k$)','Spending Score (1-100)']]
for feature in features:
    df_scaled[feature] = scaler.fit_transform(df_scaled[feature])
```

```
df_scaled
```

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	-1.424569	-1.738999	-0.434801
1	2	Male	-1.281035	-1.738999	1.195704
2	3	Female	-1.352802	-1.700830	-1.715913
3	4	Female	-1.137502	-1.700830	1.040418
4	5	Female	-0.563369	-1.662660	-0.395980
...	...	...	...	...	...
195	196	Female	-0.276302	2.268791	1.118061
196	197	Female	0.441365	2.497807	-0.861839
197	198	Male	-0.491602	2.497807	0.923953
198	199	Male	-0.491602	2.917671	-1.250054
199	200	Male	-0.635135	2.917671	1.273347

200 rows × 5 columns

8.11.3. Clustering based on two features: annual income and spending score

The figure below do appear to be some patterns in the data.

```
#df_scaled['Gender'].tolist()
```

```
#df_scaled.plot.scatter("Annual Income (k$)","Spending Score (1-100)", c =
df_scaled['Gender'])

fig, ax = plt.subplots()

colors = {'Male':'lightblue', 'Female':'pink'}

ax.scatter(df_scaled['Annual Income (k$)'], df_scaled['Spending Score (1-100)'],
c=df['Gender'].map(colors))

plt.title('Annual income vs spending score w.r.t gender')

plt.show()
```


```
sns.scatterplot('Annual Income (k$)', 'Spending Score (1-100)', data=df_scaled,
hue='Gender').set(title='Annual income vs spending score w.r.t gender')
```

```
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages/seaborn/_decorators.py:43:
FutureWarning: Pass the following variables as keyword args: x, y. From version 0.12, the
only valid positional argument will be `data`, and passing other arguments without an
explicit keyword will result in an error or misinterpretation.
```

```
[Text(0.5, 1.0, 'Annual income vs spending score w.r.t gender')]
```

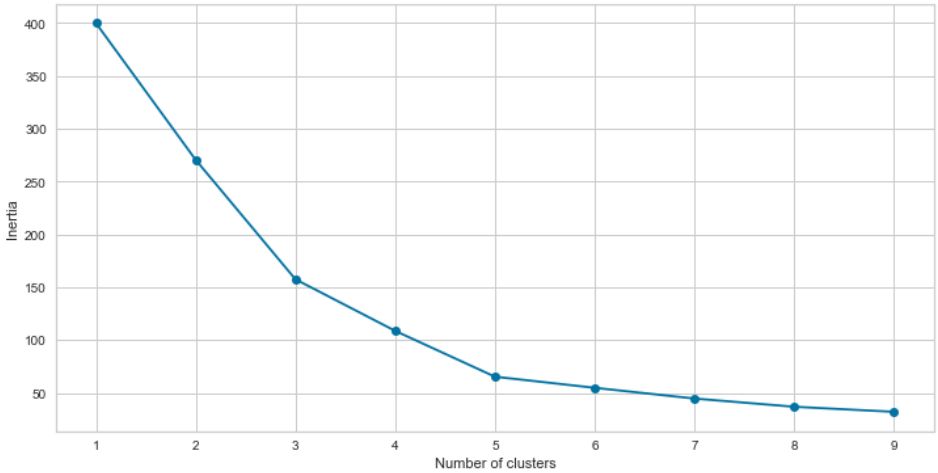

```
(
    ggplot(df_scaled)
    + aes(x = 'Annual Income (k$)', y = 'Spending Score (1-100)', color = 'Gender')
    + geom_point()
    + labs(title = 'Annual income vs spending score w.r.t gender')
)
```


```
<ggplot: (304898945)>
```

8.11.3.1. Choosing the appropriate number of clusters: Elbow Method

```
# Taking the annual income and spending score
features = ['Annual Income (k$)','Spending Score (1-100)']
model1 = df_scaled[features]


# fitting multiple k-means algorithms and storing the values in an empty list

SSE = []
for cluster in range(1,10):
    kmeans = KMeans(n_clusters = cluster)
    kmeans.fit(model1)
    SSE.append(kmeans.inertia_)

# converting the results into a dataframe and plotting them
frame = pd.DataFrame({'Cluster':range(1,10), 'SSE':SSE})
plt.figure(figsize=(12,6))
plt.plot(frame['Cluster'], frame['SSE'], marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
```

```
Text(0, 0.5, 'Inertia')
```



The best number of clusters for our data is clearly 5, as the curve slope is not severe enough after that.

8.11.3.2. Choosing the appropriate number of clusters: Silhouette Method

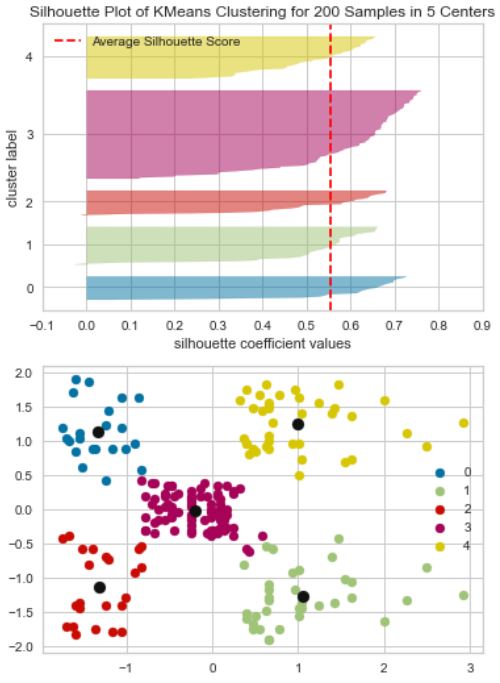
The silhouette method calculates each point's silhouette coefficients, which measure how well a data point fits into its assigned cluster based on two factors:

- How close the data point is to other points in the cluster.
- How far the data point is from points in other clusters.

```
from yellowbrick.cluster import SilhouetteVisualizer
```

```
# Instantiate the clustering model and visualizer

kmeans = KMeans(
                init="random",
                n_clusters=5,
                n_init=10,
                max_iter=300,
                random_state=88)

visualizer = SilhouetteVisualizer(kmeans, colors='yellowbrick')
# Fit the data to the visualizer
visualizer.fit(model1)
# Compute silhouette_score
print('The silhouette score:', visualizer.silhouette_score_)
# Finalize and render the figure
visualizer.show()



# For scatter plot
kmeans.fit(model1)
pred1 = kmeans.predict(model1)
#Getting the Centroids
centroids = kmeans.cluster_centers_
u_labels = np.unique(kmeans.predict(model1))

#plotting the results:
for i in u_labels:
    plt.scatter(model1.iloc[pred1 == i , 0] , model1.iloc[pred1 == i , 1] , label = i)

plt.scatter(centroids[:,0] , centroids[:,1] , s = 80, color = 'k')
plt.legend()
plt.show()
```

```
The silhouette score: 0.5546571631111091
```



`n_clusters=5` has the best average silhouette score of around 0.55, and all clusters are above the average, indicating that it is a good choice.

```
df['cluster'] = pred1
df.groupby('cluster').mean()[['Age','Annual Income (k$)','Spending Score (1-100)']]
```

cluster	Age	Annual Income (k$)	Spending Score (1-100)
0	25.272727	25.727273	79.363636
1	41.114286	88.200000	17.114286
2	45.217391	26.304348	20.913043
3	42.716049	55.296296	49.518519
4	32.692308	86.538462	82.128205

8.11.3.3. Cluster Analysis

Based on the above results, which clusters should be the target group?

There are five clusters created by the model including

1. **Cluster 0: Low annual income, high spending (young age spendthrift)**

Customers in this category earn less but spend more. People with low income but high spending scores can be viewed as possible target customers. We can see that people with low income but high spending scores are those who, for some reason, love to buy things more frequently despite their limited income. Perhaps it's because these people are happy with the mall's services. The shops/malls **may not be able to properly target these customers**, but they will not be lost.

1. **Cluster 1: High annual income, low spending (miser)**

Customers in this category earn a lot of money while spending little. It's amazing to observe that customers have great income yet low expenditure scores. Perhaps they are the customers who are **dissatisfied with the mall's services**. These are likely to be the mall's primary objectives, as they have the capacity to spend money. As a result, mall officials will **attempt to provide additional amenities in order to attract** these customers and suit their expectations.

1. **Cluster 2: Low annual income, low spending (pennywise)**

They make less money and spend less money. Individuals with low yearly income and low expenditure scores are apparent, which is understandable given that people with low wages prefer to buy less; in fact, these are the smart people who know how to spend and save money. People from this cluster will be of little interest to the shops/mall.

1. **Cluster 3: Medium annual income, medium spending**

In terms of income and spending, customers are average. We find that people have average income and average expenditure value. These people will **not be the primary target** of stores or malls, but they will be taken into account and other data analysis techniques can be used to increase their spending value.

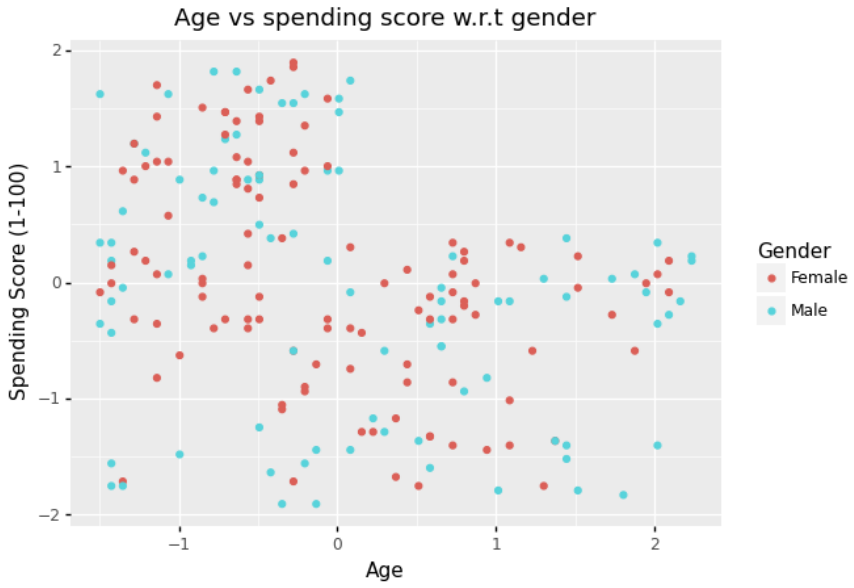
1. **Cluster 4: High annual income, high spending (young age wealthy and target customer)**

Target Customers that earn a lot of money and spend a lot of money. A **target consumer** with a high annual income and a high spending score. People with high income and high spending scores are great customers for malls and businesses because they are the primary profit generators. These individuals may be regular mall patrons who have been persuaded by the mall's amenities.

8.11.4. Clustering based on three features: annual income, spending score and age

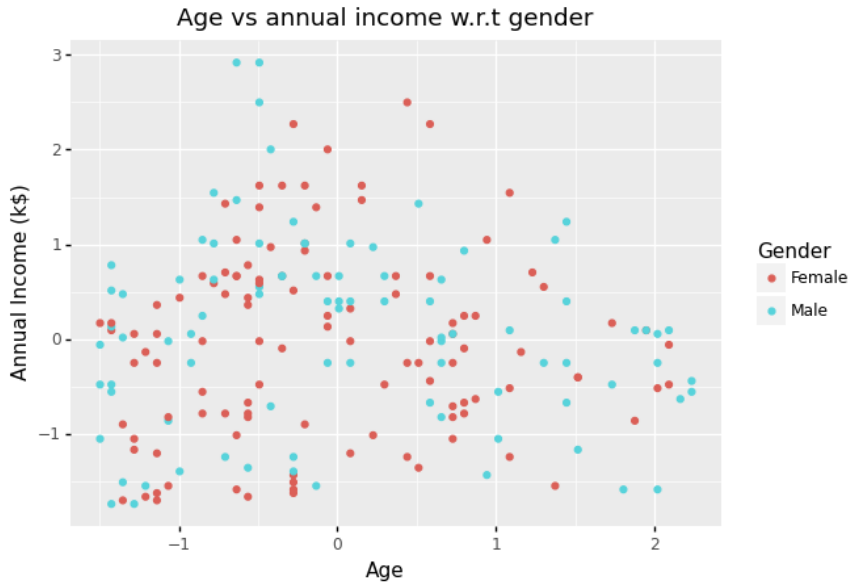
8.11.4.1. Ploting the relation between age and other features

```
(
    ggplot(df_scaled)
    + aes(x = 'Age', y = 'Spending Score (1-100)', color = 'Gender')
    + geom_point()
    + labs(title = 'Age vs spending score w.r.t gender')
)
```




```
<ggplot: (305033681)>
```

```
(
    ggplot(df_scaled)
    + aes(x = 'Age', y = 'Annual Income (k$)', color = 'Gender')
    + geom_point()
    + labs(title = 'Age vs annual income w.r.t gender')
)
```



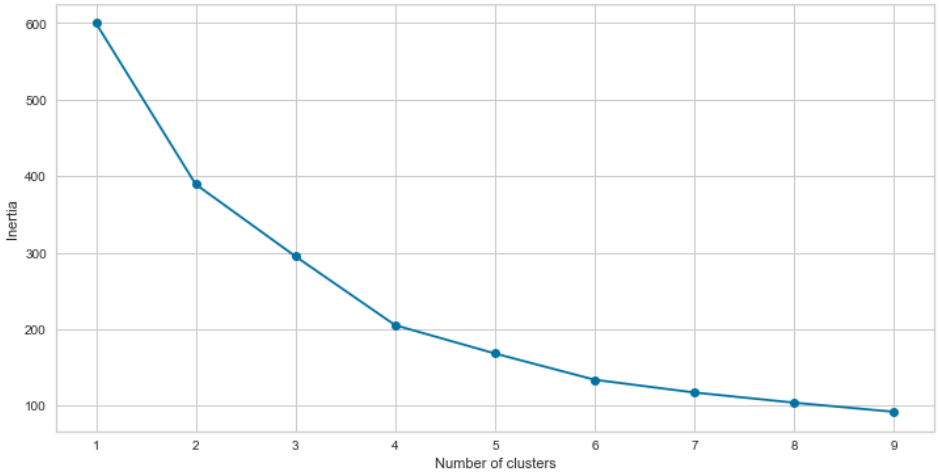

```
<ggplot: (307000729)>
```

Note Using a 2D visualization, we can't see any distinct patterns in the data set.

8.11.4.2. The elbow method

```
# Taking the annual income and spending score
features = ['Age','Annual Income (k$)','Spending Score (1-100)']
model2 = df_scaled[features]


# fitting multiple k-means algorithms and storing the values in an empty list

SSE = []
for cluster in range(1,10):
    kmeans = KMeans(n_clusters = cluster)
    kmeans.fit(model2)
    SSE.append(kmeans.inertia_)

# converting the results into a dataframe and plotting them
frame = pd.DataFrame({'Cluster':range(1,10), 'SSE':SSE})
plt.figure(figsize=(12,6))
plt.plot(frame['Cluster'], frame['SSE'], marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
```

```
Text(0, 0.5, 'Inertia')
```

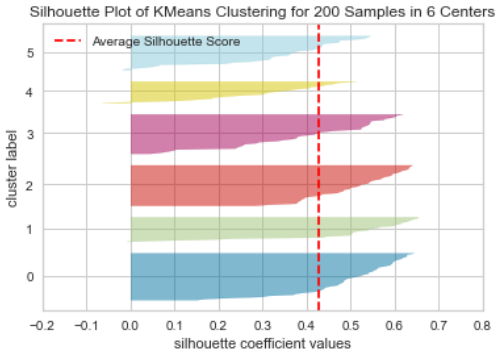


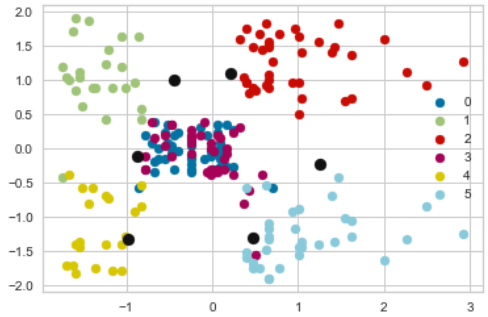
8.11.4.3. The Silhouette Method

```
from sklearn import metrics
```

```
# Instantiate the clustering model and visualizer

kmeans = KMeans(
                init="random",
                n_clusters=6,
                n_init=10,
                max_iter=300,
                random_state=88)

visualizer = SilhouetteVisualizer(kmeans, colors='yellowbrick')
# Fit the data to the visualizer
visualizer.fit(model2)
# Compute silhouette_score
print('The silhouette score:', visualizer.silhouette_score_)
# Finalize and render the figure
visualizer.show()



# For scatter plot
kmeans.fit(model2)
pred2 = kmeans.predict(model2)
#Getting the Centroids
centroids = kmeans.cluster_centers_
u_labels = np.unique(kmeans.predict(model2))

#plotting the results:
for i in u_labels:
    plt.scatter(model1.iloc[pred2 == i , 0] , model1.iloc[pred2 == i , 1] , label = i)

plt.scatter(centroids[:,0] , centroids[:,1] , s = 80, color = 'k')
plt.legend()
plt.show()
```

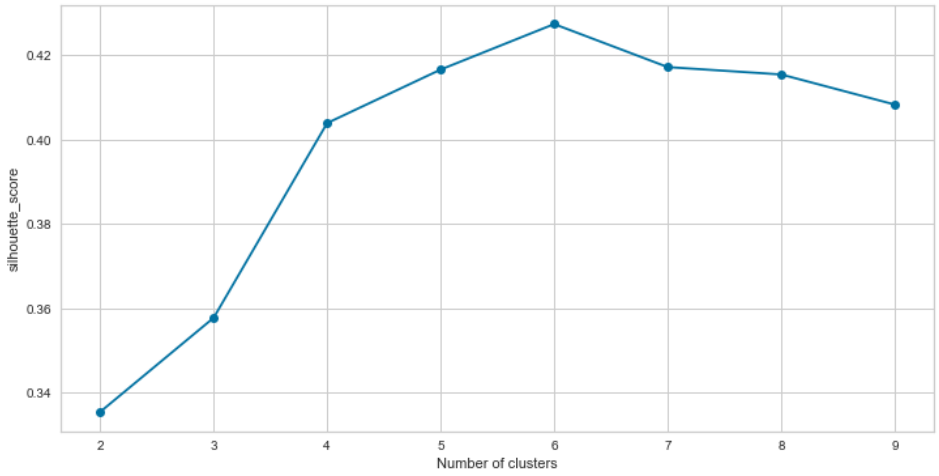
```
The silhouette score: 0.42742814991580175
```






```
silhouette_score = []
for cluster in range(2,10):
    kmeans = KMeans(
                init="random",
                n_clusters=cluster,
                n_init=10,
                max_iter=300,
                random_state=88)

    kmeans.fit(model2)
    labels = kmeans.labels_
    silhouette_score.append(metrics.silhouette_score(model2, labels))

# converting the results into a dataframe and plotting them
frame = pd.DataFrame({'Cluster':range(2,10), 'Silhouette_score':silhouette_score})
plt.figure(figsize=(12,6))
plt.plot(frame['Cluster'], frame['Silhouette_score'], marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('silhouette_score')
```

```
Text(0, 0.5, 'silhouette_score')
```




```
!pip install plotly
```

```
Requirement already satisfied: plotly in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (5.7.0)
Requirement already satisfied: six in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from plotly) (1.16.0)
Requirement already satisfied: tenacity>=6.2.0 in
/Users/Kaemyuijang/opt/anaconda3/lib/python3.7/site-packages (from plotly) (8.0.1)
```

```
import plotly as py
import plotly.graph_objs as go
```

```
df
```

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)	cluster
0	1	Male	19	15	39	2
1	2	Male	21	15	81	0
2	3	Female	20	16	6	2
3	4	Female	23	16	77	0
4	5	Female	31	17	40	2
...	...	...	...	...	...	...
195	196	Female	35	120	79	4
196	197	Female	45	126	28	1
197	198	Male	32	126	74	4
198	199	Male	32	137	18	1
199	200	Male	30	137	83	4

200 rows × 6 columns

```
# Clustering with n_clusters = 6

#algorithm = (KMeans(n_clusters = 6 ,init='k-means++', n_init = 10 ,max_iter=300,
#                    tol=0.0001,  random_state= 111  , algorithm='elkan') )

kmeans = KMeans(
               init="random",
               n_clusters=6,
               n_init=10,
               max_iter=300,
               random_state=88)
kmeans.fit(model2)

pred2 = kmeans.predict(model2)
#Getting the Centroids
centroids2 = kmeans.cluster_centers_

#algorithm.fit(X3)
#labels3 = algorithm.labels_
#centroids3 = algorithm.cluster_centers_
```

```
df
```

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)	cluster
0	1	Male	19	15	39	2
1	2	Male	21	15	81	0
2	3	Female	20	16	6	2
3	4	Female	23	16	77	0
4	5	Female	31	17	40	2
...	...	...	...	...	...	...
195	196	Female	35	120	79	4
196	197	Female	45	126	28	1
197	198	Male	32	126	74	4
198	199	Male	32	137	18	1
199	200	Male	30	137	83	4

200 rows × 6 columns

```
# x and y given as array_like objects
import plotly.express as px
fig = px.scatter(x=[0, 1, 2, 3, 4], y=[0, 1, 4, 9, 16])
fig.show()
```

```
labels3 = kmeans.predict(model2)



df['label3'] =  labels3
trace1 = go.Scatter3d(
    x= df['Age'],
    y= df['Spending Score (1-100)'],
    z= df['Annual Income (k$)'],
    mode='markers',
     marker=dict(
        color = df['label3'],
        size= 20,
        line=dict(
            color= df['label3'],
            width= 12
        ),
        opacity=0.8
     )
)
data = [trace1]
layout = go.Layout(
#     margin=dict(
#         l=0,
#         r=0,
#         b=0,
#         t=0
#     )
    title= 'Clusters',
    scene = dict(
            xaxis = dict(title  = 'Age'),
            yaxis = dict(title  = 'Spending Score'),
            zaxis = dict(title  = 'Annual Income')
        )
)
fig = go.Figure(data=data, layout=layout)
py.offline.iplot(fig)
```

```
# x and y given as array_like objects
import plotly.express as px
fig = px.scatter(x=[0, 1, 2, 3, 4], y=[0, 1, 4, 9, 16])
fig.show()
```

Previous
7. Machine learning: Introduction

By Pairote Satiracoo
© Copyright 2021.