Data Science Foundations: Data Mining in R

James Evans

06/08/2025

Introduction

R for data mining

Back in 1890, William James, the founder of American psychology, wrote about how newborn babies were inundated with sensory stimuli, calling it a "great blooming, buzzing confusion." It takes time and effort for the baby to make sense of what's me, what's not me, what matters, and what can be safely ignored. That same feeling often comes to people when they first start working with data—especially large and diverse datasets. The problem is that data, which holds the secrets to so much value, can become an embarrassment of riches: there's so much available, with even more pouring in, but it's hard to know what to do with it. How can you make sense of it all? How can you find the real value in such an enormity of raw materials? What matters, and what can be safely ignored? Unlike a newborn who must wait for the slow processes of cognitive and social development, we can turn immediately to data mining for answers. I'm Barton Poulson, and in this course, we'll explore some of the most important principles and techniques in modern data mining to help you cut through the noise. We'll look at methods including dimensionality reduction to help identify reliable indicators, clustering to sort cases into useful groups, classification to automate difficult categorizations, association analysis to uncover if/then relationships, time series analysis for understanding and forecasting temporal patterns, and text mining—particularly for extracting insights from unstructured reviews and evaluations. We'll emphasize hands-on practice using one of my favorite tools: the R programming language and the RStudio environment. You'll learn how to use both simple and sophisticated methods to mine data for value and pursue your goals. All of these techniques will help you navigate the great blooming and buzzing confusion of the data world. Let's get started with *Data Science Foundations: Data Mining with R.

Who should watch this course

While this course is designed as an accessible introduction to the field of data mining, it works best if there are a few things you're already familiar with. First, you should have a working knowledge of the R programming language and the RStudio environment. If you need to get up to speed, I recommend the course *Learning R*, or the two-part series *R Essential Training*, where part one focuses on wrangling and visualizing data, and part two covers modeling data. In addition, it's helpful to be familiar with basic statistical concepts. For a refresher, I suggest the course *Data Fluency: Exploring and Describing Data*, along with several other courses available in our online learning library.

Exercise Files

If you have access to the exercise files for this course, you can download them to your desktop, as I've done here. When you open the exercise files folder, you'll see that it contains three subfolders: one for all of the code or R scripts used in this course, another called `data` that contains several example datasets, and a third folder named `output` that is currently empty but can be used to save any images or files you create. Additionally, there are two other files in the exercise files folder: a README file that explains the folder structure and contents, and an R project file that helps organize the code and data within RStudio, which is the environment we'll be using in this course. If you don't have access to the exercise files, that's okay—you can still follow along by watching how I use the course materials.

Preliminaries

Tools for data mining

If you're going to go out mining, then you need the right tools—and when you're mining for data gold, that means software. The two most common tools for this are Python and R. In fact, this course is available in two versions: one using Python and the other using R. There are good reasons to focus on these languages. Python is currently the most popular language for data science and machine learning. As a general-purpose language, it is widely known and highly adaptable, with a clean, readable syntax that makes it easy to learn. R, on the other hand, was developed specifically for data analysis and is especially popular among scientists and researchers; it was the first language I personally used for working with data. Both are excellent for mining data, but it's also worth knowing about other tools. Point-and-click GUI (Graphical User Interface) applications designed for data work range in cost and capability—some resemble spreadsheets, while others include full programming environments. These include SPSS (popular in the social sciences), SAS (with strong programming features), jamovi (a free, open-source application based on R and similar to SPSS), JASP (also free and similar to SPSS and jamovi), and Tableau (a leading tool for data visualization and exploratory data analysis). Beyond these tools, anyone working with data should be familiar with SQL (Structured Query Language), which is essential for accessing databases—especially in organizational settings. SQL offers a high return on investment, requiring minimal effort to achieve significant results in querying and managing data. Spreadsheets like Microsoft Excel and Google Sheets are also crucial; they are the most commonly used data tools in the world, widely accessible, and often the default format for data sharing. They're useful for browsing, basic analysis, and quick visualizations, and anyone working with data should be proficient with them. My philosophy on data tools follows a progression: start with simple tools like spreadsheets; move to GUI-based applications like jamovi or SPSS when necessary; and only transition to programming languages like Python or R when the complexity of the data or analysis demands it. Though programming languages are the most powerful and flexible, they are also the most challenging to master. To provide context, KDnuggets—a popular site for data mining news—conducts regular surveys on tool usage. In a recent survey, 66% of respondents reported using Python, 51% used RapidMiner (noting possible bias from survey promotion), 47% used R, and 35% used Excel. Anaconda, which is a Python distribution, and SQL were also commonly cited. Job postings from Indeed (as of 2017) reinforce these trends: Python and SQL remain highly demanded, and R continues to be important. Ultimately, choose your tools wisely. While this course focuses on Python and R, many other tools can serve your needs depending on your goals and your clients' requirements. The key is using what works best for your project, so you can begin mining your data for valuable insights.

The CRISP-DM data mining model

While this course focuses substantially on how to program analyses to mine data, it's also important to consider the broader context in which that programming occurs—namely, the structure of the entire project. In data mining, there is a well-established framework called CRISP-DM, which stands for Cross Industry Standard Process for Data Mining. CRISP-DM outlines six key phases. The first is **business understanding**, which involves identifying business objectives, conducting a situation assessment, defining data mining goals, and producing a project plan. The second phase is **data understanding**, where you collect initial data, describe and explore it, and assess its quality. The third phase is **data preparation**, which consists of selecting relevant data, cleaning it, constructing derived attributes or generated records, integrating it with other sources, and formatting it to fit the intended software tools. The fourth phase is **modeling**, where you choose the modeling technique, check assumptions, generate test designs, build the model—including tuning parameters—and evaluate model performance. The fifth phase is **evaluation**, which includes assessing the results in relation to business success criteria, reviewing the overall process, and identifying next steps or making a final decision. Finally, the sixth phase is **deployment**, where conclusions are applied to solve the original problem. This includes planning deployment and maintenance, producing final reports or presentations, and documenting the process for future evaluation and replication. CRISP-DM remains one of the most widely used process models in data mining. For an alternative perspective, I've also described my personal system—the Data Science Pathway—in the course *Data Science Foundations: Fundamentals.*

Whether you follow the industry-standard CRISP-DM or the Data Science Pathway, both approaches will help you better structure, execute, and extract insights from your data mining projects.

Privacy, copyright, and bias

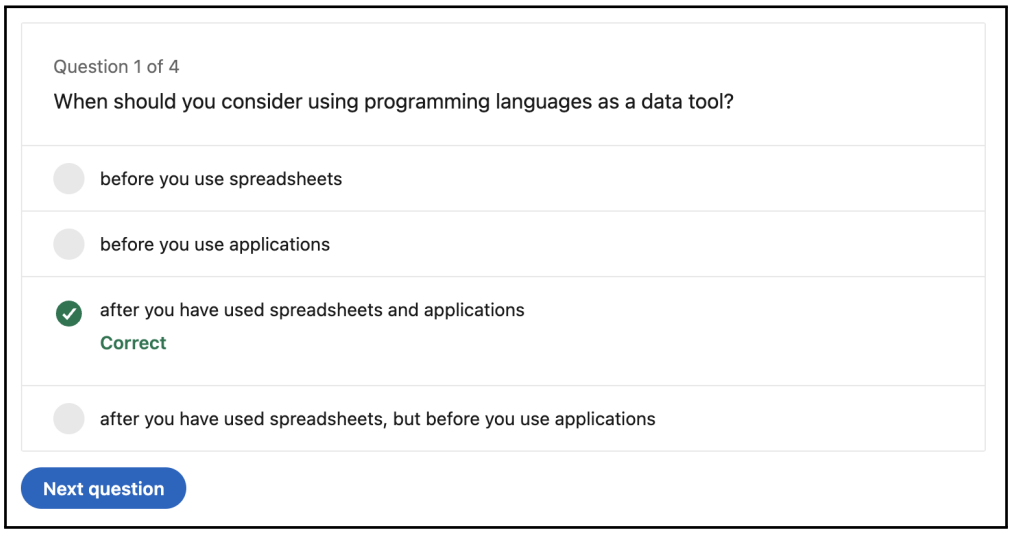
While this course provides technical training, it's crucial to remember that data mining, like virtually all human activities, takes place within a social context. It is essential to respect the individuals who provided the data and those who may benefit from your analysis. One of the primary concerns is **privacy**, which influences how data is gathered, stored, and shared. This is not only an ethical matter but also a legal one. Key privacy regulations include the GDPR (General Data Protection Regulation) in the European Union, the CCPA (California Consumer Privacy Act) in California, HIPAA (Health Insurance Portability and Accountability Act) in the United States for health data, and FERPA (Family Educational Rights and Privacy Act) for educational records. These laws are specific and enforceable, dictating what data can be used, how it must be handled, and what procedures are required to stay in compliance. If you're working with data covered by these regulations, you owe it to yourself and others to understand how they apply. Another critical issue is **copyright**, especially since data mining frequently involves information sourced from the web. Just as you cannot freely use online images or videos without proper licensing, you cannot assume that online data is free to use. Even if data appears publicly accessible, it may be protected by proprietary rights, and using it may require permission or licensing. You must verify the data's usage rights before proceeding. **Bias and data quality** are also key considerations. The principle of "Garbage In, Garbage Out" (GIGO) reminds us that flawed or unrepresentative data will lead to flawed outcomes, regardless of algorithmic sophistication. It's critical to assess data quality, ensure diversity in your sample, and match training data characteristics to those expected in deployment. That includes reviewing the spread of populations and conditions, and the weighting of different cases. Moreover, you must consider **construct validity**, which asks whether your data truly measures what you intend it to measure. This is particularly important in data mining, where you often repurpose data not originally collected for your goals. Proper interpretation is essential to ensure conclusions are valid and meaningful. For deeper exploration of these ethical and social issues, including privacy, bias, and licensing, you can consult the course *AI Accountability Essential Training*, which provides actionable insights for designing and implementing ethical, accurate, and socially responsible data mining projects.

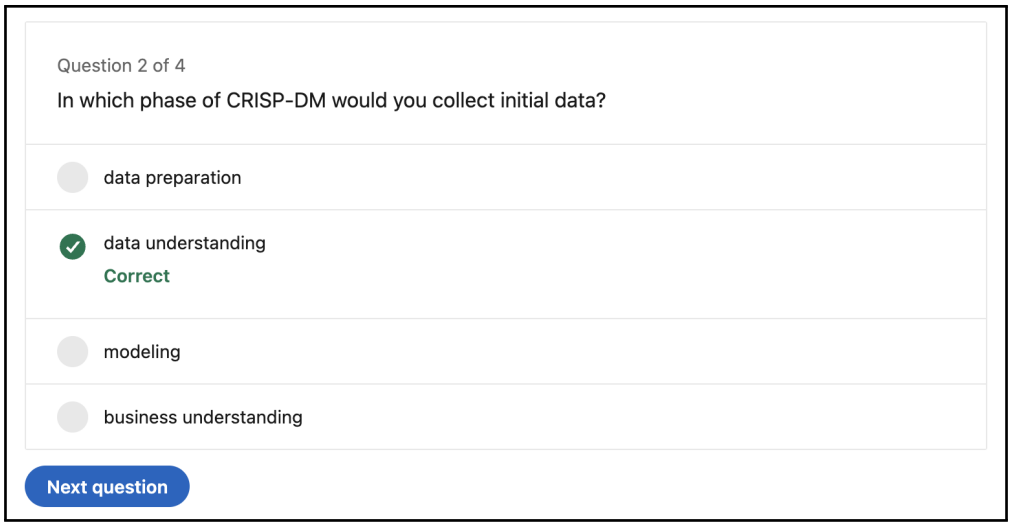
Validating results

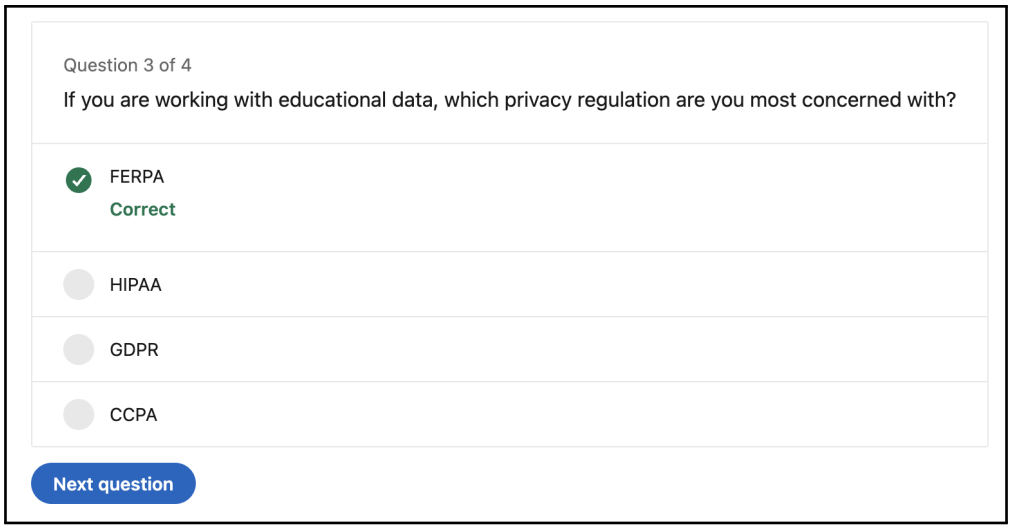
Here in Utah, I can see the world's largest copper mine—the Bingham Canyon Mine, also known as the Kennecott Copper Mine—from my kitchen window. It's several miles across and visible even from NASA satellite imagery. Mining is hard work; if valuable materials were sitting on the surface like watermelons, we wouldn't need to dig. The same is true in data mining—except instead of copper, we are searching for signal hidden in noise, looking for meaningful, actionable patterns amid randomness. However, one of the most important things to be aware of is the risk of **overfitting**: with enough parameters, a model can fit any dataset perfectly, but such a model may perform poorly on new data. This issue is addressed through **statistical model validation**, which asks: does your model generalize well beyond your training data? This can also be framed in terms of **external validity**—how well your model holds up outside the lab—or more broadly as **generalizability**. Fortunately, there are several techniques to assess how well your model performs on new data. A traditional approach from the laboratory sciences involves hands-on data review: checking distributions, spotting outliers, validating assumptions, and evaluating data-model fit. This also includes examining your sample for representativeness and investigating any unusual missing data patterns. Afterward, you often repeat the study with new data—**replication**—possibly in a different setting or with slightly modified methods, then compare or combine results to evaluate consistency over time. Alternatively, immediate validation can be performed using your current dataset via two major techniques: **holdout validation** and **cross-validation**. In holdout validation, you split the dataset into a training set (often 70–90%) and a testing set (the remainder). You build your model on the training set and evaluate it just once on the untouched testing set. Sometimes, a third split—called a validation set—is used to compare models before final testing. I will demonstrate this method in several examples throughout the course. In **cross-validation**, you divide the dataset into k folds (e.g., 3, 5, 10, or 20), train the model on $k-1$ folds,

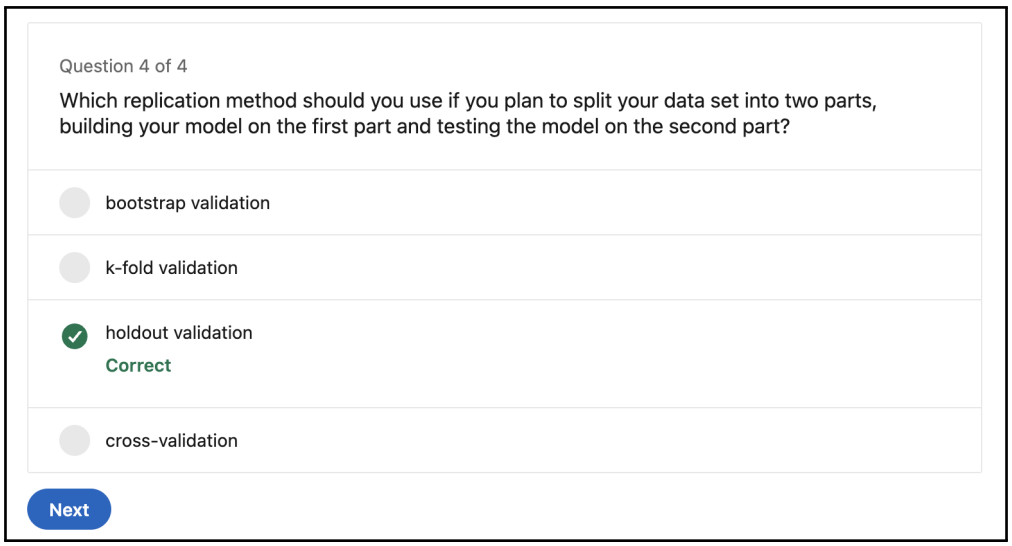
and test it on the remaining fold. You repeat this k times so each fold serves as a test set. For example, you might train on 80% of the data, validate on 20%, and rotate through each segment. Once a good model is found, it can be evaluated on the final holdout set. These validation strategies allow you to confidently assess your model's robustness. Just as digging, sifting, and sorting yield value in copper mining, thorough validation ensures that your data mining results are reliable and actionable.

Chapter Quiz







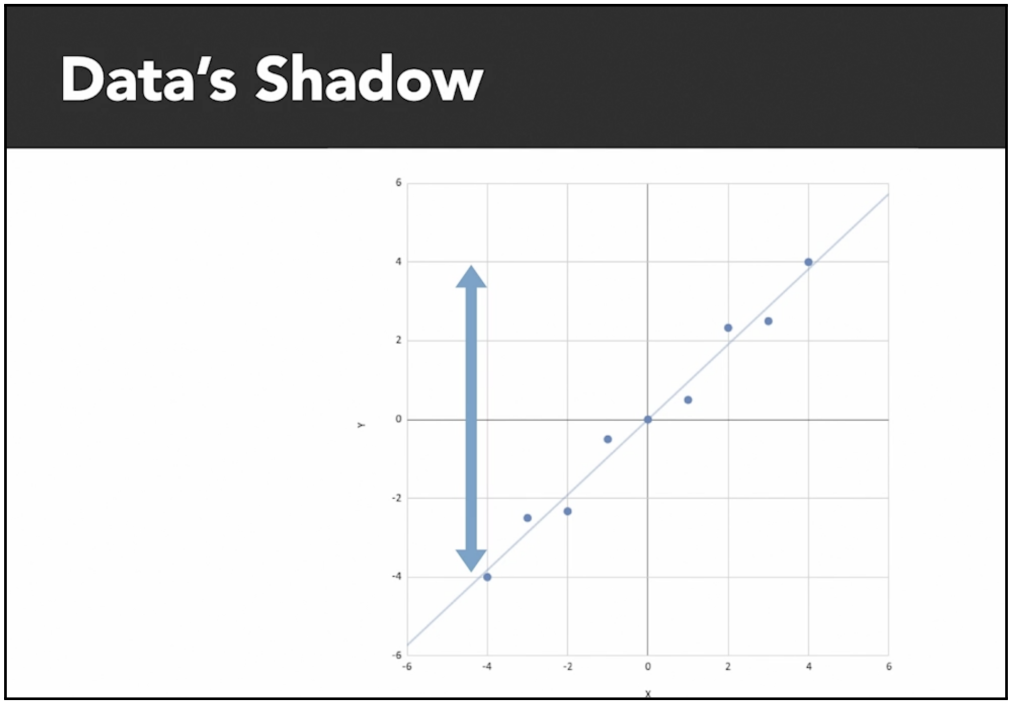


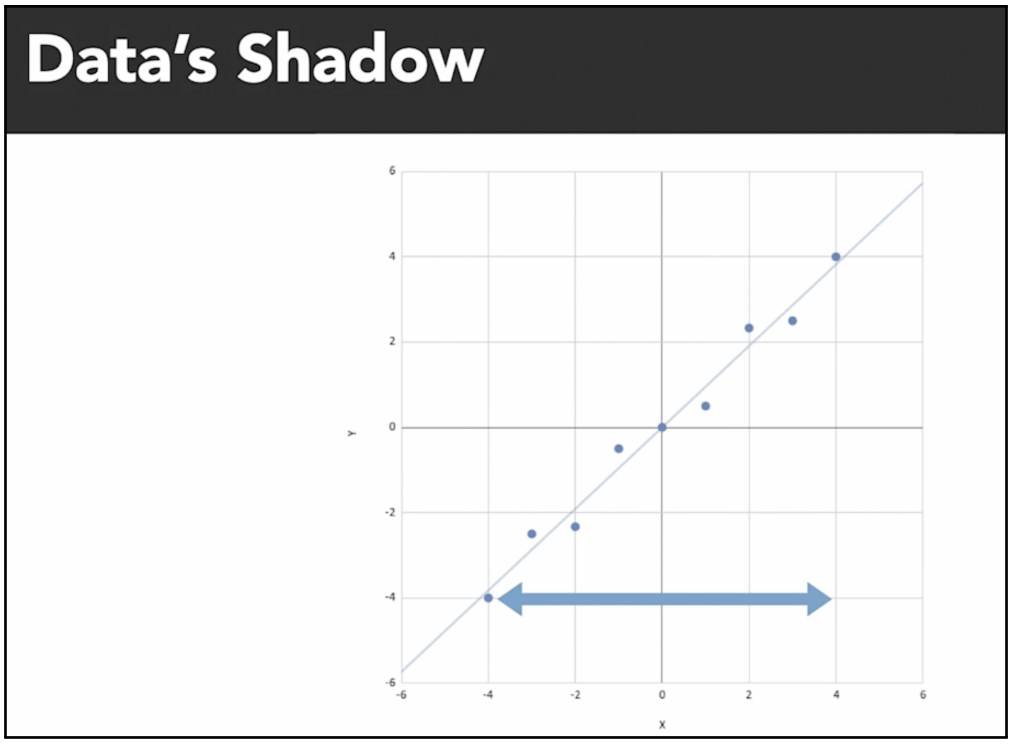
Dimensionality Reduction

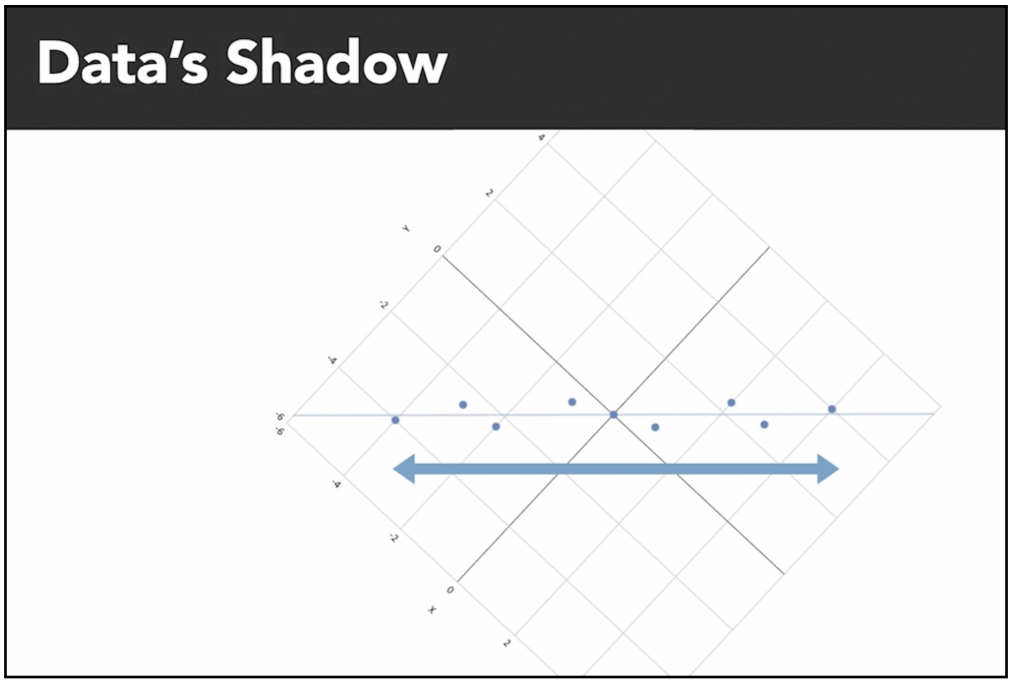
Dimensionality reduction overview

Data mining comes into its own when you're dealing with large datasets. You may be familiar with the three V's that define big data: volume, velocity, and variety. But paradoxically, having too much data can be problematic. While having lots of cases or observations is usually good, problems arise when you have many dimensions or variables. High dimensionality introduces challenges such as exponentially increasing computational complexity—because each additional variable multiplies the possible combinations—and increased idiosyncratic error, as each variable brings its own noise. This noise makes it harder to detect meaningful signals and raises the risk of overfitting. For instance, an algorithm might latch onto 17 out of 1,000 variables that correlate with an outcome by random chance, leading to poor generalization. High dimensionality also reduces model interpretability, making it nearly impossible for humans to understand a model with hundreds of interacting variables. One solution is dimensionality reduction—combining variables into fewer, more stable components. Interestingly, combining variables tends to reduce random error due to noise cancellation, improving model generalizability and interpretability. The idea is to project the data into a lower-dimensional space while preserving as much useful variability as possible. A metaphor for this is the shadow of a hand: although the hand is three-dimensional, its two-dimensional shadow can still convey essential information. **In data, a scatterplot of two variables (X and Y) might show**

significant variability in both dimensions, but if we rotate it to align with a principal direction (like a regression line), we can reduce the data to one meaningful component. Three popular dimensionality reduction techniques are PCA (Principal Component Analysis), LDA (Linear Discriminant Analysis), and t-SNE (t-distributed Stochastic Neighbor Embedding). PCA identifies combined components that capture variance across multiple variables, often using rotation to align the data with its most meaningful directions. LDA, although used for classification, reduces dimensions by combining variables in ways that best separate predefined classes. t-SNE, a nonlinear technique, is effective for visualizing complex structures such as helices or nested circles, where traditional linear separation is ineffective. These methods, among others, help simplify high-dimensional data into more manageable, stable, and generalizable forms, enabling better insight and performance in data mining.







Dataset: Handwritten digits

For our demonstrations of dimensionality reduction, we will be using a well-known dataset in the machine learning world: the optical recognition of handwritten digits. This dataset consists of images of handwritten digits, where each digit is overlaid on a grid of pixels—typically 32 by 32, though we illustrate the idea using a simplified 16 by 16 grid. Each 4-by-4 block in the grid is converted into a numerical value that counts the number of active (i.e., non-white) pixels, generating a sequence of values that represents the image. The final variable in the dataset is a class label indicating which digit (e.g., 5) is being represented. In

R, we use a script called `handwritten_digits.R`, and we load relevant packages such as `janitor` to help clean the data. The dataset originates from the UCI Machine Learning Repository and can be found in the `optdigits.tra` file. After downloading, we convert it into a CSV file for easier use and load it into an R object `DF`. The dataset has 3,822 observations and 65 variables, with values ranging from 0 to 16 indicating the count of activated pixels per region. We then clean up the dataset: rename the final variable to `Y` (our class label), convert it into a factor, and filter the data to keep only digits 1, 3, and 6. We also remove columns with constant values, such as corner pixels that are always blank, reducing the variable count from 65 to 56. To prepare for modeling, we split the dataset into training and testing sets. We set a random seed (e.g., `set.seed(1)`) to ensure reproducibility, assign unique row IDs, and use 70% of the data for training (`TRN`), with the remainder assigned to testing using an `anti_join`. After removing the temporary ID column from all datasets, we save the final versions in `.RDS` format: `opdigits.rds` for the full dataset, and `opdigits_TRN.rds` and `opdigits_TST.rds` for the training and testing subsets. Lastly, we clear the console, packages, and any open graphics to create a clean session. If issues arise, restarting the R session under the `Session > Restart R` menu is recommended. With this preparation, we are ready to begin exploring dimensionality reduction techniques in R.

PCA

For our demonstrations of dimensionality reduction, we will be using a well-known dataset in the machine learning world: the optical recognition of handwritten digits. This dataset consists of images of handwritten digits, where each digit is overlaid on a grid of pixels—typically 32 by 32, though we illustrate the idea using a simplified 16 by 16 grid. Each 4-by-4 block in the grid is converted into a numerical value that counts the number of active (i.e., non-white) pixels, generating a sequence of values that represents the image. The final variable in the dataset is a class label indicating which digit (e.g., 5) is being represented. In R, we use a script called `handwritten_digits.R`, and we load relevant packages such as `janitor` to help clean the data. The dataset originates from the UCI Machine Learning Repository and can be found in the `optdigits.tra` file. After downloading, we convert it into a CSV file for easier use and load it into an R object `DF`. The dataset has 3,822 observations and 65 variables, with values ranging from 0 to 16 indicating the count of activated pixels per region. We then clean up the dataset: rename the final variable to `Y` (our class label), convert it into a factor, and filter the data to keep only digits 1, 3, and 6. We also remove columns with constant values, such as corner pixels that are always blank, reducing the variable count from 65 to 56. To prepare for modeling, we split the dataset into training and testing sets. We set a random seed (e.g., `set.seed(1)`) to ensure reproducibility, assign unique row IDs, and use 70% of the data for training (`TRN`), with the remainder assigned to testing using an `anti_join`. After removing the temporary ID column from all datasets, we save the final versions in `.RDS` format: `opdigits.rds` for the full dataset, and `opdigits_TRN.rds` and `opdigits_TST.rds` for the training and testing subsets. Lastly, we clear the console, packages, and any open graphics to create a clean session. If issues arise, restarting the R session under the `Session > Restart R` menu is recommended. With this preparation, we are ready to begin exploring dimensionality reduction techniques in R.

🔧 What is Dimensionality Reduction?

Dimensionality reduction is the process of transforming a dataset with **many variables** (also called **features** or **dimensions**) into a new version with **fewer variables** — ideally without losing much important information.

🎯 Why Do It?

When you have lots of variables:

- Your model becomes complex
- There's more **random noise**
- There's a higher chance of **overfitting**
- Interpretation becomes difficult

By **reducing dimensions**, you're aiming to:

- **Simplify the model**
- **Reduce noise**
- **Improve generalization** (performance on unseen data)
- **Make it easier to understand**

🧠 Noise Cancellation via Combination

When you **combine variables**, you're effectively averaging them in a smart way. Each original variable may contain a little random error (noise), but when you blend many of them together:

- Some of that noise **cancels out**
- The resulting composite variable (e.g., a principal component) is **more stable**
- This leads to **better predictions** and models that work better on new data

📉 What Does "Projecting into a Lower-Dimensional Space" Mean?

Imagine you're compressing your data into fewer variables that still capture its core structure.

You're doing this by:

- Identifying the **underlying patterns** (directions of maximum variance)
- Creating **new variables** (components) that summarize those patterns
- Dropping the less important ones

This is what methods like **PCA (Principal Component Analysis)** do — rotate and collapse the data into a smaller, cleaner form.

🔑 Summary:

> Dimensionality reduction simplifies a dataset by **combining features**, reducing **noise**, and capturing the **essential structure** in fewer dimensions. This improves model stability, performance, and human interpretability.

Let me know if you'd like a visual or code example using PCA or t-SNE.

🎯 Why Retain Maximum Variance in Dimensionality Reduction?

When we reduce dimensions (e.g., via **PCA**), we want to **retain as much variance as possible** because:

---

1. Variance = Information

- In most datasets, **variance** represents the **signal** — the meaningful differences between data points.
- Features with higher variance tend to capture more of the **structure**, **patterns**, or **relationships** in the data.
- If you discard high-variance components, you risk discarding the **most informative content**.

---

2. Low Variance = Noise or Redundancy

- Features with very low variance typically:
  - Are **constant or near-constant** across examples
  - Provide little distinction between data points
  - Might represent **measurement noise**, unimportant features, or redundant information

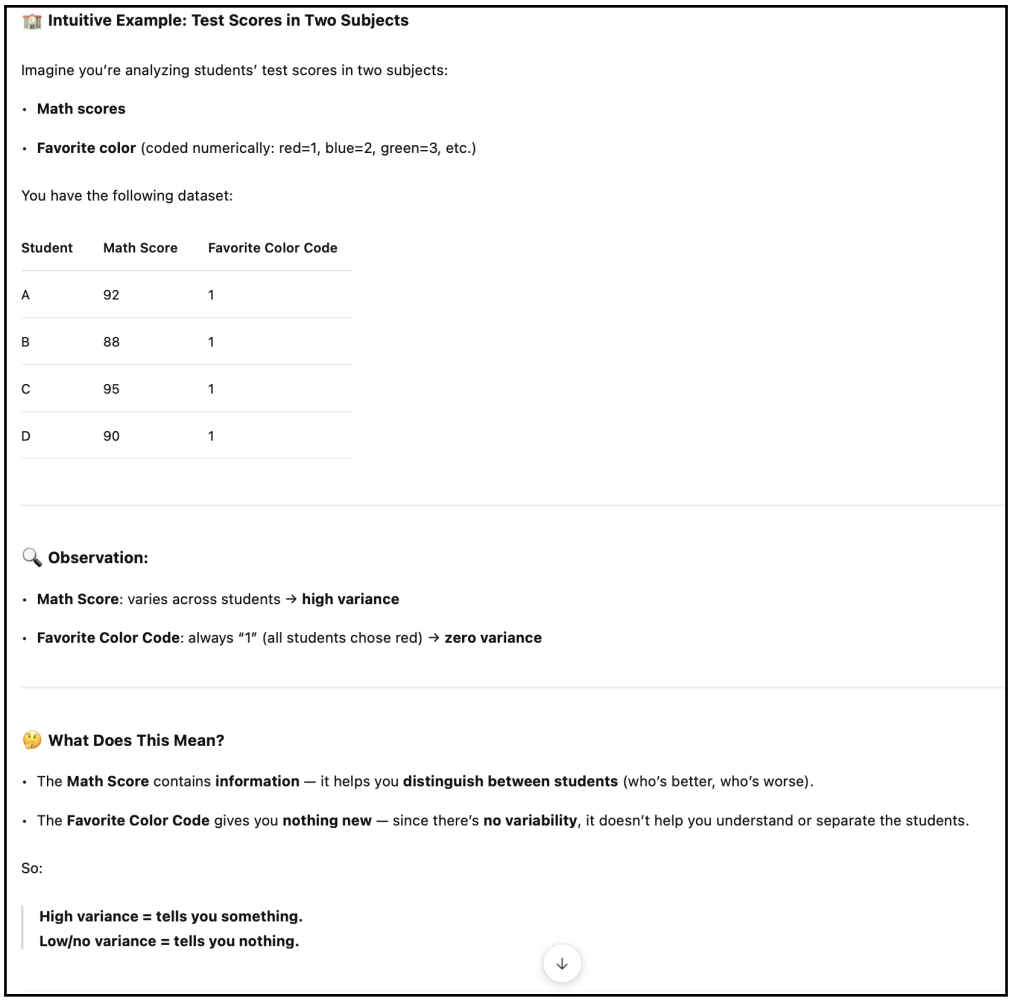
By dropping low-variance components, you:

- **Compress the data efficiently**
- **Improve generalization**
- **Reduce overfitting**

3. Mathematical Guarantee (e.g., PCA)

- PCA constructs new axes (principal components) such that:
  - The **first component** captures the **most variance**
  - The second captures the **next most**, and so on
- If you keep the first k components, you are guaranteed to retain the **maximum possible variance** using k dimensions

This makes PCA **optimal** for linear dimensionality reduction under the "retain variance" criterion.



🏫 **Intuitive Example: Test Scores in Two Subjects**

Imagine you're analyzing students' test scores in two subjects:

- **Math scores**
- **Favorite color** (coded numerically: red=1, blue=2, green=3, etc.)

You have the following dataset:

Student	Math Score	Favorite Color Code
A	92	1
B	88	1
C	95	1
D	90	1

🔍 **Observation:**

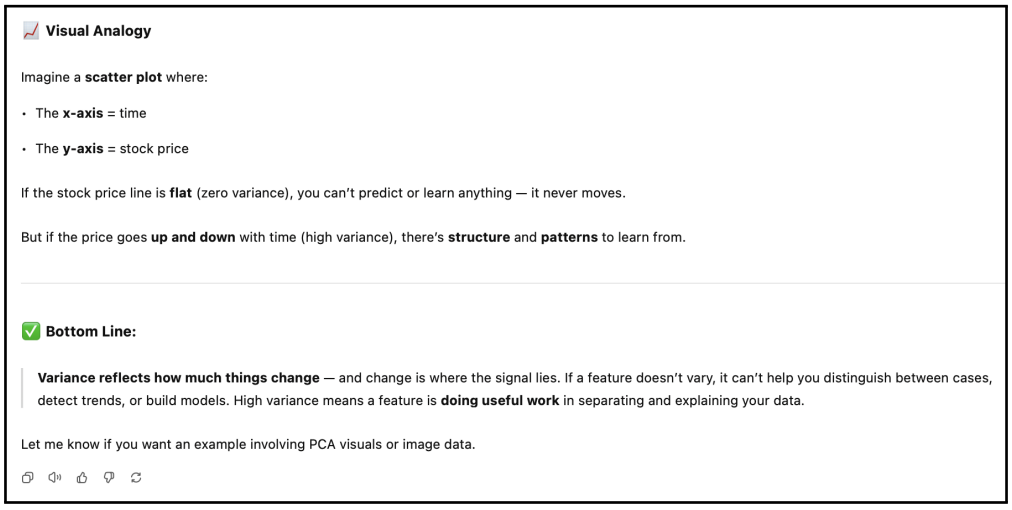
- **Math Score**: varies across students → **high variance**
- **Favorite Color Code**: always "1" (all students chose red) → **zero variance**

🤔 **What Does This Mean?**

- The **Math Score** contains **information** — it helps you **distinguish between students** (who's better, who's worse).
- The **Favorite Color Code** gives you **nothing new** — since there's **no variability**, it doesn't help you understand or separate the students.

So:

> **High variance = tells you something.**
> **Low/no variance = tells you nothing.**



📈 **Visual Analogy**

Imagine a **scatter plot** where:

- The **x-axis** = time
- The **y-axis** = stock price

If the stock price line is **flat** (zero variance), you can't predict or learn anything — it never moves.

But if the price goes **up and down** with time (high variance), there's **structure** and **patterns** to learn from.

✅ **Bottom Line:**

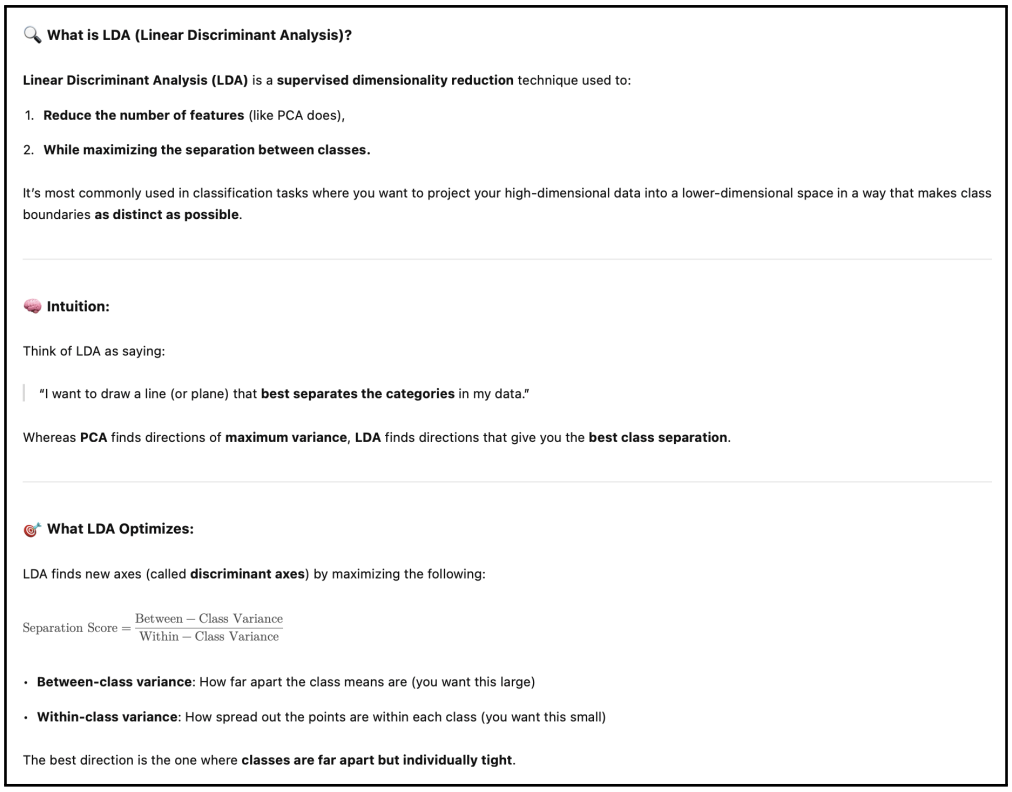
> **Variance reflects how much things change** — and change is where the signal lies. If a feature doesn't vary, it can't help you distinguish between cases, detect trends, or build models. High variance means a feature is **doing useful work** in separating and explaining your data.

Let me know if you want an example involving PCA visuals or image data.

LDA

For our demonstrations of dimensionality reduction, we will be using a well-known dataset in the machine learning world: the optical recognition of handwritten digits. This dataset consists of images of handwritten digits, where each digit is overlaid on a grid of pixels—typically 32 by 32, though we illustrate the idea using a simplified 16 by 16 grid. Each 4-by-4 block in the grid is converted into a numerical value that counts

the number of active (i.e., non-white) pixels, generating a sequence of values that represents the image. The final variable in the dataset is a class label indicating which digit (e.g., 5) is being represented. In R, we use a script called `handwritten_digits.R`, and we load relevant packages such as `janitor` to help clean the data. The dataset originates from the UCI Machine Learning Repository and can be found in the `optdigits.tra` file. After downloading, we convert it into a CSV file for easier use and load it into an R object `DF`. The dataset has 3,822 observations and 65 variables, with values ranging from 0 to 16 indicating the count of activated pixels per region. We then clean up the dataset: rename the final variable to `Y` (our class label), convert it into a factor, and filter the data to keep only digits 1, 3, and 6. We also remove columns with constant values, such as corner pixels that are always blank, reducing the variable count from 65 to 56. To prepare for modeling, we split the dataset into training and testing sets. We set a random seed (e.g., `set.seed(1)`) to ensure reproducibility, assign unique row IDs, and use 70% of the data for training (`TRN`), with the remainder assigned to testing using an `anti_join`. After removing the temporary ID column from all datasets, we save the final versions in `.RDS` format: `opdigits.rds` for the full dataset, and `opdigits_TRN.rds` and `opdigits_TST.rds` for the training and testing subsets. Lastly, we clear the console, packages, and any open graphics to create a clean session. If issues arise, restarting the R session under the `Session > Restart R` menu is recommended. With this preparation, we are ready to begin exploring dimensionality reduction techniques in R.



🔍 **What is LDA (Linear Discriminant Analysis)?**

Linear Discriminant Analysis (LDA) is a **supervised dimensionality reduction** technique used to:

1. **Reduce the number of features** (like PCA does),
2. **While maximizing the separation between classes.**

It's most commonly used in classification tasks where you want to project your high-dimensional data into a lower-dimensional space in a way that makes class boundaries **as distinct as possible**.

🧠 **Intuition:**

Think of LDA as saying:

> "I want to draw a line (or plane) that **best separates the categories** in my data."

Whereas **PCA** finds directions of **maximum variance**, **LDA** finds directions that give you the **best class separation**.

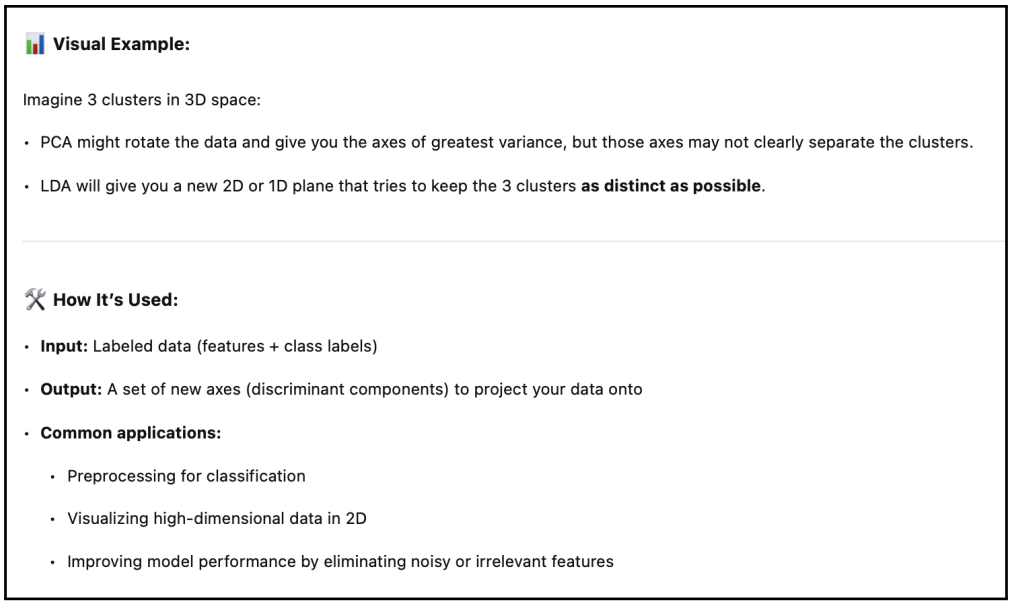
🎯 **What LDA Optimizes:**

LDA finds new axes (called **discriminant axes**) by maximizing the following:

$$\text{Separation Score} = \frac{\text{Between} - \text{Class Variance}}{\text{Within} - \text{Class Variance}}$$

- **Between-class variance**: How far apart the class means are (you want this large)
- **Within-class variance**: How spread out the points are within each class (you want this small)

The best direction is the one where **classes are far apart but individually tight**.



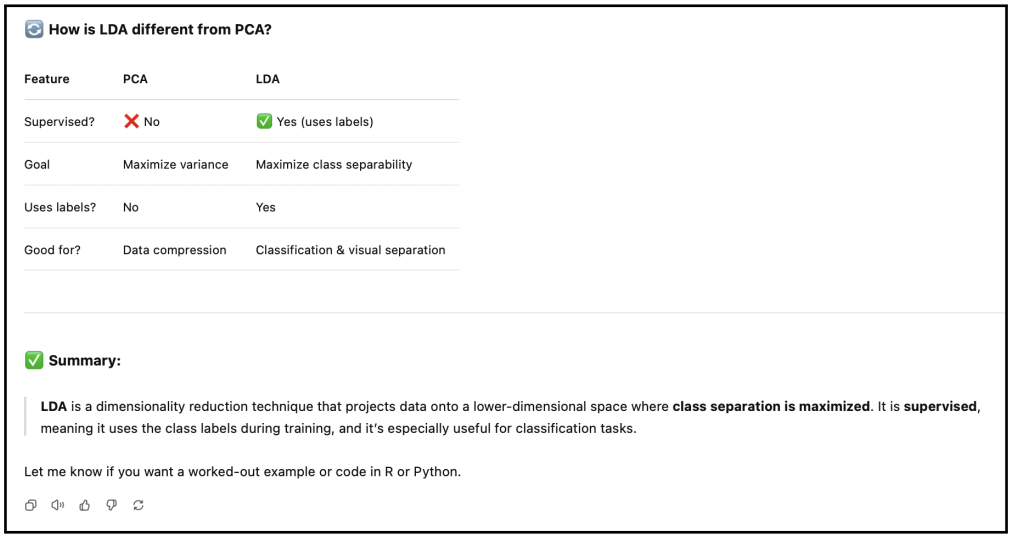
📊 **Visual Example:**

Imagine 3 clusters in 3D space:

- PCA might rotate the data and give you the axes of greatest variance, but those axes may not clearly separate the clusters.
- LDA will give you a new 2D or 1D plane that tries to keep the 3 clusters **as distinct as possible**.

🛠️ **How It's Used:**

- **Input:** Labeled data (features + class labels)
- **Output:** A set of new axes (discriminant components) to project your data onto
- **Common applications:**
  - Preprocessing for classification
  - Visualizing high-dimensional data in 2D
  - Improving model performance by eliminating noisy or irrelevant features



🔄 **How is LDA different from PCA?**

Feature	PCA	LDA
Supervised?	❌ No	✅ Yes (uses labels)
Goal	Maximize variance	Maximize class separability
Uses labels?	No	Yes
Good for?	Data compression	Classification & visual separation

---

✅ **Summary:**

> **LDA** is a dimensionality reduction technique that projects data onto a lower-dimensional space where **class separation is maximized**. It is **supervised**, meaning it uses the class labels during training, and it's especially useful for classification tasks.

Let me know if you want a worked-out example or code in R or Python.

t-SNE

The final method of dimensionality reduction we will demonstrate in R is t-SNE, short for t-distributed stochastic neighbor embedding. This relatively recent and sophisticated approach excels at nonlinear separation, making it suitable when groups cannot be separated by straight lines, when one group surrounds another, or when curves overlap. Unlike linear methods, t-SNE can still find meaningful groupings. It is primarily used for visualization, and in our demonstration, we begin by loading relevant packages, especially `rtsne`. We use the full dataset because t-SNE is exploratory in nature and does not require splitting into training and testing subsets. We assign the data to a data frame `df`, set a random seed for reproducibility, and run `Rtsne()` with the perplexity parameter, which is key to the algorithm's performance. Starting with perplexity 1, we run the function, plotting the results by selecting the two t-SNE dimensions and coloring them by class label. The output at perplexity 1 is an indistinct blob, with no separation between the digit classes. Increasing the perplexity to 2 significantly improves separation, making clusters visually distinguishable. We repeat the process with perplexity values of 5, 10, and 50, and find that even modest increases yield well-separated digit classes, despite minor overlaps. This demonstrates t-SNE's strength in uncovering structure using nonlinear transformations. It handles high-dimensional and complex data well, making it a powerful tool for dimensionality reduction when conventional linear methods fall short.

🌐 What is t-SNE?

t-SNE is a **nonlinear dimensionality reduction algorithm**.
It is mainly used to **visualize high-dimensional data** by reducing it to **2 or 3 dimensions**, especially for tasks like clustering, pattern recognition, or embedding.

🧠 The Goal

t-SNE aims to **preserve the local structure** of the data.
That means: **If two points are close in high dimensions, they should stay close in the 2D or 3D visualization**.

🚶 How Does It Work (Intuition)?

1. **Start in high-dimensional space**
   - It measures how close points are to each other (their "neighborliness"), and converts that into **probabilities**.
   - If two points are very similar, the probability that they are neighbors is **high**.
2. **Create a low-dimensional version (2D or 3D)**
   - It initializes points randomly and **tries to match the same probability structure** as in the high-dimensional version.
   - If two points were close in high-dimensions, t-SNE **tries to place them close** in 2D.
   - If they were far apart, it **pushes them apart**.
3. **The trick: use the t-distribution**
   - Instead of using a normal (Gaussian) distribution in the low-dimensional space, t-SNE uses a **t-distribution with heavier tails**.
   - This helps **spread out** dissimilar points more clearly and avoid "crowding".

🧪 Example

Suppose you have 10,000 handwritten digit images (0–9), each image 784 pixels (28×28 = 784D). You run t-SNE on that dataset, and get a **2D plot** where:

- All the 1's are clustered together
- All the 7's are in a different cluster
- Some ambiguous 3's and 8's may overlap

This is what makes t-SNE popular in **unsupervised learning** and **data exploration**.

⚠️ Limitations

- **Slow** on large datasets
- Not good for downstream ML models (use PCA/LDA instead)
- Output can **change each time** you run it (because of random initialization)
- Not interpretable axes — it's for **visualization**, not measurement

---

✅ Use t-SNE When:

- You want to **visualize complex data** (e.g., word embeddings, image vectors)
- You care about **clusters and local neighborhoods**
- You don't need exact distances or interpretable axes

Let me know if you want to compare t-SNE to **PCA or UMAP**.

🧩 **Scenario**

Imagine we have **5 data points** in **high-dimensional space**:
Each point represents a person based on **50 survey questions** (so: 50D).
Let's name the people:

- Alice
- Bob
- Charlie
- Dana
- Evan

Let's say Alice and Bob answered very similarly. Dana is very different. Charlie and Evan are somewhat similar.

Step 1: Measure Similarity in High Dimensions

We compute how similar each pair of points is using **distance** (e.g., Euclidean or cosine), and then convert those distances into **probabilities**.

Think of it like this:

> "Given Alice, how likely is she to pick Bob as a close neighbor compared to everyone else?"

This gives us a **conditional probability**:

- $P_{j|i}$ = probability that point x_i picks x_j as its neighbor.

We do this for **all pairs**:

- Alice-Bob: very similar → high probability
- Alice-Dana: very different → very low probability
- Charlie-Evan: medium similarity → medium probability

Then we **symmetrize** it:

$$P_{ij} = \frac{P_{j|i} + P_{i|j}}{2N}$$

Now, we have a symmetric matrix of similarity probabilities in **high-dimensional space**.

Step 2: Create a 2D Map (Random Initialization)

We randomly assign **2D coordinates** to Alice, Bob, Charlie, Dana, and Evan.

Right now, these 2D positions are arbitrary — nothing matches the original structure yet.

Step 3: Measure Similarity in 2D

We now compute a similar matrix of probabilities, but in **2D**, using the **t-distribution** to calculate:

$$Q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k \neq l}(1 + ||y_k - y_l||^2)^{-1}}$$

This gives us a matrix Q that tells us how close people are in the 2D space.

- If Alice and Bob are far apart in 2D, but their P_{ij} was high in high-D space → we have a mismatch.

Step 4: Reduce the Mismatch

We now adjust the positions of the points in 2D to make $Q_{ij} \approx P_{ij}$.

We do this by minimizing the **Kullback-Leibler divergence**:

$$\mathrm{KL}(P||Q) = \sum_{i \neq j} P_{ij} \log\left(\frac{P_{ij}}{Q_{ij}}\right)$$

This is done via **gradient descent**, nudging points:

- Alice and Bob get pulled **closer together**
- Alice and Dana get pushed **further apart**

This process continues for **many iterations** until the 2D structure best reflects the local structure of the original data.

Final Result:

After convergence, our 2D map might look like:

Copy

```
Bob     Alice

Charlie     Evan

Dana
```

- Alice and Bob are close (as they were in 50D)
- Charlie and Evan are clustered
- Dana is far from everyone

We've preserved **local structure** and made the data **visually interpretable**.

Challenge: PCA

Now I want you to take a moment to try doing dimensionality reduction on your own using an example dataset from R. To do this, you need to load a few packages including R's built-in datasets. Once we do that, you can access the `swiss` dataset. I'm going to use `?swiss` to get some information on that dataset. What this is, is Swiss fertility and socioeconomic indicators from 1888. So you know it's 140 years old almost, and it has six variables in it, 47 observations, so it's a small dataset. It tells us about the fertility in each of 47 French-speaking provinces in Switzerland, the percent of males involved in agriculture, the percent of draftees receiving the highest mark on an army examination, education, the percent Catholic as opposed to Protestant, and infant mortality – that's the live births who live less than one year. So the question is whether these six variables can be reasonably and usefully collapsed into a smaller number of dimensions. For this, I'm going to ask you to use principal component analysis. Try it on your own, and then I'll show you how I solve this in the next video.

Solution: PCA

In the previous video, I invited you to do dimensionality reduction with principal component analysis on one of R's built-in datasets, the `swiss` dataset. I'm going to be using the code based off of the principal components demonstration we had earlier. To do this, I am going to load a few packages, `car` for the scatterplot matrix, `datasets` which is where the dataset is, and `ggbiplot`, in addition to the other ones that I load every time. So let me begin by loading those packages. And then we'll take a quick look at the `swiss` dataset. Again, it's 47 observations from the 1880s on six variables. And the idea is to see if these variables can be collapsed into a smaller number of dimensions. I'm going to begin by importing the `swiss` dataset as a tibble, that's an enhanced data frame that prints a little better, and saving it as `DF`. That's how I usually save things. It stands for data frame, and then let's take a quick look at it and what we have are these variables: fertility, agriculture, and so on. And we're seeing the values for each of the 47. So we've got variability and we can also see that they're double or integer as we go through. Let's explore the data by doing a scatterplot matrix of all the variables. So I take the data frame and I feed the whole thing into `scatterplotMatrix`. And I'm going to zoom in on that and what you see are the distributions of each variable, and they're kind of normal, this one's a little skewed, this one's bimodal, and you can see the associations. Some of them are pretty strong. Here's a good, strong relationship. Some of them are kind of all over the place, like this one, but we don't have massive outliers and it gives us something to start with. Now, I'm going to do the actual principal components analysis. I'm going to be using `prcomp`, the most common method. And all I'm going to do is center and scale the variables so they all have a mean of zero and a standard deviation or variance of one. So they all can have equal influence potentially in the model. I'll feed `DF` into that and save it as `PC` for principal components. Then I'll get the summary statistics for that analysis. And you see that we have six components – that's because we started with six variables – and it tells us how much each one of them contributed. And you can see it's not equal. It's easiest to get a scree plot of the eigenvalues. And you can see that the first one accounts for a lot, the second one much less, and the others less than that. We may have a one-dimensional solution or sort of one and a half, but let's plot it with the biplot. That's going to be the most useful thing that we can do. So I take `PC` and I feed it into `biplot`. Also, I'm not adding any extra arguments because I don't have a class or a category that I'm trying to model. I'm just looking at the associations between the variables. So let me zoom in on this one. And what you can see is that we have two dimensions – the first dimension accounts for over 50% of the variance in the original dataset, the second accounts for about 20%. And what you can see here is a dot for each of the 47 observations – these are regions in French-speaking Switzerland in 1888 – and then these are the variables. And you can see, for instance, that education and examination scores go very closely together, and they're sort of standing in opposition to the percentage of adult males who are working in agriculture, the percentage of the residents who are Catholic, and the fertility rate. Infant mortality stands aside as a separate thing. It's a little bit associated with fertility and Catholic, which explains why it's a second dimension, but really it looks like we can have a good two-dimensional solution here, going from our original six dimensions or variables down to two principal components as a way of summarizing the data. And that is how I solved the dimensionality reduction problem using the `swiss` dataset.

Chapter Quiz

Question 1 of 5

For dimensionality reduction using the tSNE.R approach, what should you change in the following code block to make your data more distinguishable?

```
df %>%
     select(-y) %>%
     Rtsne(
          perplexity = 3,
          verbose = TRUE,
          max_iter = 500
     ) %>%
```

- Set verbose to FALSE.
- ✓ Increase the perplexity value.

  Correct
- Decrease the perplexity value.
- Increase the max_iter value.

Question 2 of 5

Using the `prcomp` method of principle components in R, which code block will center and scale the variables so the mean will be 0 and the same standard deviation of 1?

```
pc  <- trn %>%
     select(-y) %>%
     prcomp(
          center = Mean,
          scale = SD
)
```

```
pc  <- trn %>%
     select(-y) %>%
     princomp(
          mean = 0,
          SD = 1
)
```

```
pc  <- trn %>%
     select(-y) %>%
     PCA(
          center = 0
          scale = Standard
)
```

✓

```
pc  <- trn %>%
     select(-y) %>%
     prcomp(
          center = True,
          scale = True
)
```

Correct

Question 3 of 5

For dimensionality reduction using the LDA classification method, what would replace the ??? in the following code to use the colors 4-6 from `carPalette`? ``` scatterplotMatrix(~X1 + X4 + X5 | y, data = tst, regLine = False, smooth = False, col = ???)

✓

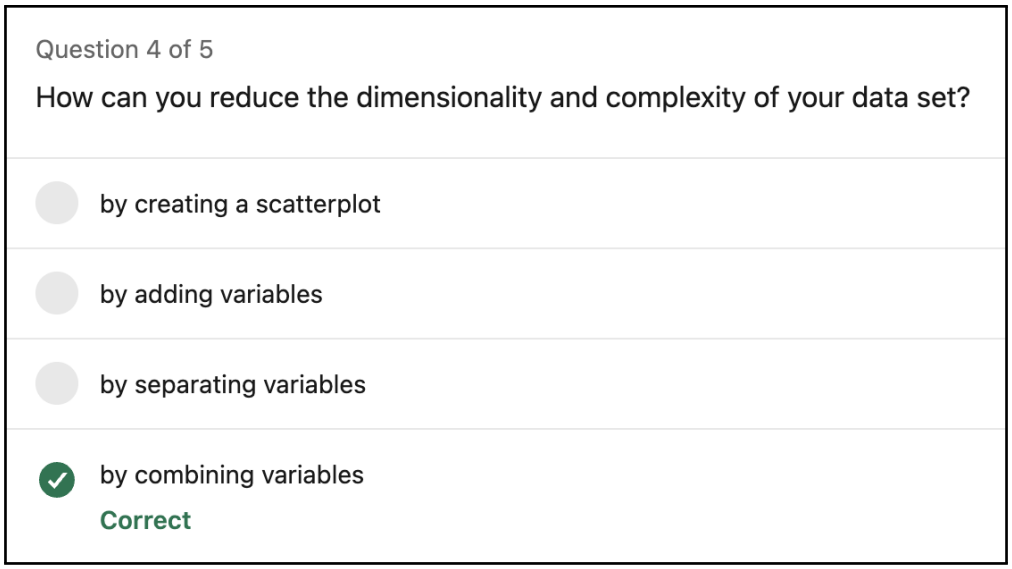
```
4 : 6
```

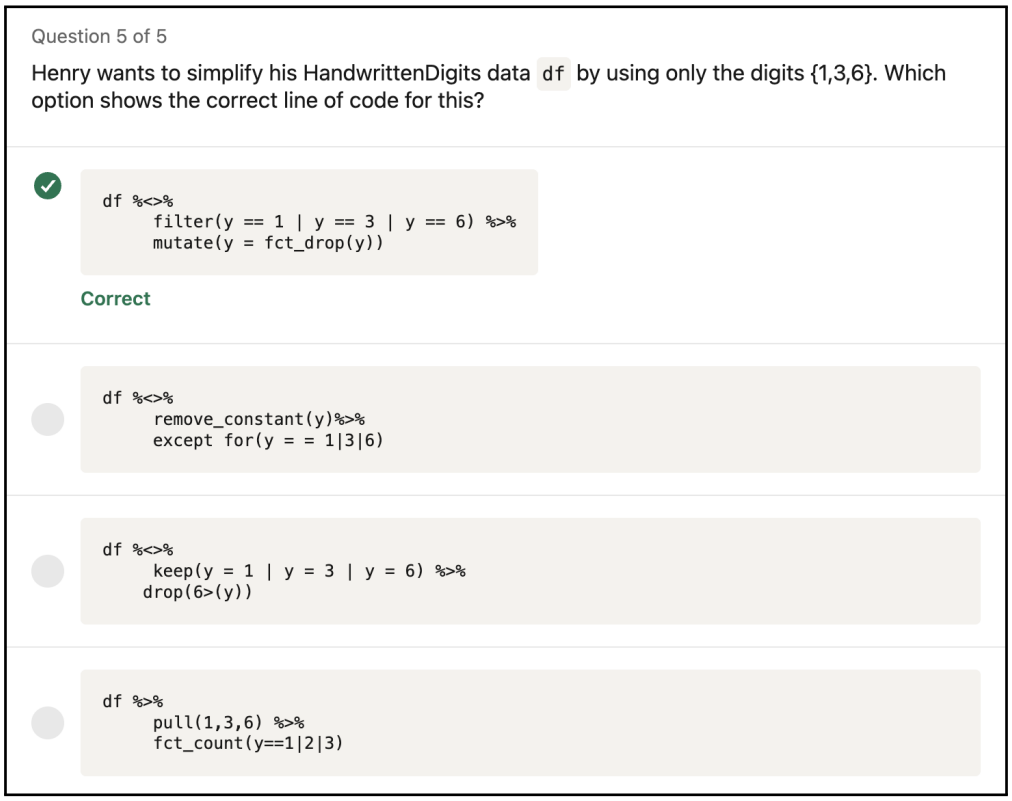
Correct

```
4, 5, 6
```

```
( 4 | 5 | 6)
```

```
4 .. 6
```





Clustering

Clustering overview

Libraries are the original big data and a great place to mine data for insights. But the value of a library is only partly in the books it holds. What makes the library useful is that the books are organized in a meaningful way. However, there are many ways to organize books or to cluster them meaningfully. For example, the Library of Congress classification system places statistics books under QA276, while the Dewey Decimal system may place them under 006. There is also the Harvard-Yenching classification for Chinese language materials. Some people classify books by size, color, ISBN, author, or publisher. The important thing is

not which system is used, but that a system is used—that similar things are grouped together. Clustering, like sorting flowers at a market, helps you find what you're looking for and see related items nearby. A few key points about clustering: (1) there is no single "correct" answer; clustering is not about matching a predefined classification but about grouping by some notion of similarity. (2) The results depend on the data you provide—limited data leads to limited clustering effectiveness. (3) Clustering serves practical goals. For example, if you have a marketing budget for two campaigns, use two clusters to tailor messages to two groups with internal similarity. Clustering relies on measuring similarity in a multidimensional space, where each variable is a dimension (e.g., 100 variables = 100 dimensions). Standardization is crucial: without it, variables with larger scales (e.g., height in feet vs. weight in pounds) dominate distance calculations. Standardizing gives each variable mean zero and unit variance, so all contribute equally. Distance between points (cases) is then measured in this space, and clustering identifies dense regions—areas where points "clump." We will cover three clustering methods. First, **hierarchical clustering**, especially agglomerative clustering, starts with each case as its own group and merges the most similar cases iteratively until all are joined. The result is a *dendrogram*, a branching graph that shows the levels of merges. A divisive variant starts with one supergroup and splits it recursively. Second, **k-means clustering** requires specifying k, the number of clusters. The algorithm finds k centroids and assigns each case to its nearest centroid, then updates the centroids iteratively. This method works best with spherical, non-overlapping clusters. Finally, we examine **DBSCAN** (density-based spatial clustering of applications with noise), which is powerful for detecting nonconvex and nonlinearly separable clusters. DBSCAN identifies dense regions and separates noise points, making it useful when clusters are oddly shaped—like banana curves. For example, a point at the tip of one banana will correctly be grouped with that banana despite being closer to another cluster. Points with insufficient local density are marked as noise and excluded. While many clustering methods exist, hierarchical clustering, k-means, and DBSCAN offer powerful, flexible options for finding useful and actionable structure in complex datasets.

Dataset: Penguins

For our demonstrations on clustering, we're going to use a new dataset called the penguins dataset, or the Palmer penguins. To use this in `R`, you can simply load the `palmerpenguins` package. I'm going to load that. You can get more information about this dataset either through the `R` package or at an associated website. The Palmer penguins dataset includes two datasets: `penguins`, which has size-related measurements, and `penguins_raw`, which includes many more variables. We're going to focus on the `penguins` dataset, which includes columns for species, island, various measurements of penguin bodies, and a factor for sex. The raw dataset has additional variables, but we will not be using that one for now. I'll save the `penguins` dataset to an object called `df`, and by wrapping the assignment in parentheses, the output is printed to the console. The dataset contains 344 observations of 8 variables. A quick summary shows that we have more *Adelie* penguins, a small number of *Chinstrap*, and a good number of *Gentoo*. The dataset includes the islands they inhabit, quantitative body measurements, 165 females, 168 males, and 11 entries where sex is not available, along with the year of measurement. As part of the data preparation, I will convert the dataset to a tibble, rename the `species` variable as `y` (to standardize naming conventions and allow for code reuse), and remove three variables: `island`, `sex`, and `year`. I will also remove incomplete cases and overwrite the cleaned dataset. When inspecting the cleaned data, we now see only the desired variables and no missing values. I will save this prepared dataset in `.RDS` format as `penguins.rds`, which is easier to use in R than other formats. This completes the dataset preparation for our three clustering demonstrations in R.

Hierarchical clustering

Our first demonstration of clustering will be hierarchical clustering. This method seeks associations within the data across multiple levels and is a common exploratory procedure. To begin, we load several packages including `car` for a scatterplot matrix, and `cluster` and `factoextra` for additional clustering utilities. We then import the previously wrangled `penguins` dataset, set a seed for reproducibility, and reduce the dataset to 100 cases to better visualize individual cases in the dendrogram. Examining the dataset reveals species labels and four physical measurements: bill length, bill depth, flipper length (in millimeters), and body mass (in grams). The species labels are extracted into a vector `Y`, and the dataset is modified to exclude this

label and standardize the variables, since they are on different scales. Standardization ensures each variable contributes equally to the clustering process. Next, a scatterplot matrix is created using the standardized measurements, grouped by species. The plot reveals some separation, especially in bill depth and flipper length. Hierarchical clustering begins by computing a distance matrix from the quantitative variables, followed by applying `hclust` to obtain the hierarchical clusters, stored in `HC`. Class labels are set using `Y`, and a dendrogram is plotted with adjusted font size. The dendrogram reveals distinct clusters corresponding roughly to the Gentoo, Chinstrap, and Adelie penguins. To determine the optimal number of clusters, we use three methods: the elbow method (which suggests three clusters), the silhouette method (which favors two clusters), and the gap statistic (which suggests five clusters). Despite these differences, three clusters align well with the actual species. We enhance the dendrogram using `rect.hclust` to draw boxes around three clusters, which correspond clearly to the species groups. Finally, using the cluster assignments, we visualize the data in 2D space with colored cluster boundaries, showing well-separated clusters—especially for Gentoo penguins—based on the four measurements. Hierarchical clustering thus proves effective for this dataset and allows flexibility in exploring different cluster structures.

🔧 How it works (Agglomerative, the most common type):

1. **Start** with n data points as n individual clusters.
2. **Compute a distance matrix** (e.g., Euclidean distance between all pairs).
3. **Merge the two closest clusters** based on a linkage criterion:
   - **Single linkage**: minimum distance between points in clusters
   - **Complete linkage**: maximum distance between points
   - **Average linkage**: average distance between all points
   - **Ward's method**: minimizes the increase in total within-cluster variance
4. **Update** the distance matrix to reflect the new cluster.
5. **Repeat** steps 3–4 until all data points are in one cluster.

📊 Visualization:

- A **dendrogram** shows how clusters are formed at each step.
- You can "cut" the dendrogram at a chosen height to get a specific number of clusters.

In **hierarchical clustering**, the **elbow method** is used to help determine the optimal number of clusters by examining how the **within-cluster sum of squares (WCSS)** decreases as the number of clusters increases.

Here's how it works:

1. **Compute hierarchical clustering** on your data.
2. **Cut the dendrogram** at various levels to produce different numbers of clusters (e.g., from 1 up to, say, 10 clusters).
3. For each number of clusters k, calculate the **total within-cluster sum of squares**:

$$WCSS(k) = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2$$

   where C_i is the i-th cluster and μ_i is its centroid.
4. **Plot $WCSS(k)$ vs. k.**
5. The "elbow" point on this curve is where the rate of decrease sharply slows down — this suggests that adding more clusters beyond this point yields diminishing returns.

Intuition:

- Before the elbow: adding clusters significantly improves the compactness of the clusters.
- After the elbow: additional clusters don't significantly reduce WCSS.

In hierarchical clustering, this approach is typically applied post hoc using the dendrogram: you calculate WCSS values for different cuts (number of clusters) and look for the elbow in the plot.

Tools in R:

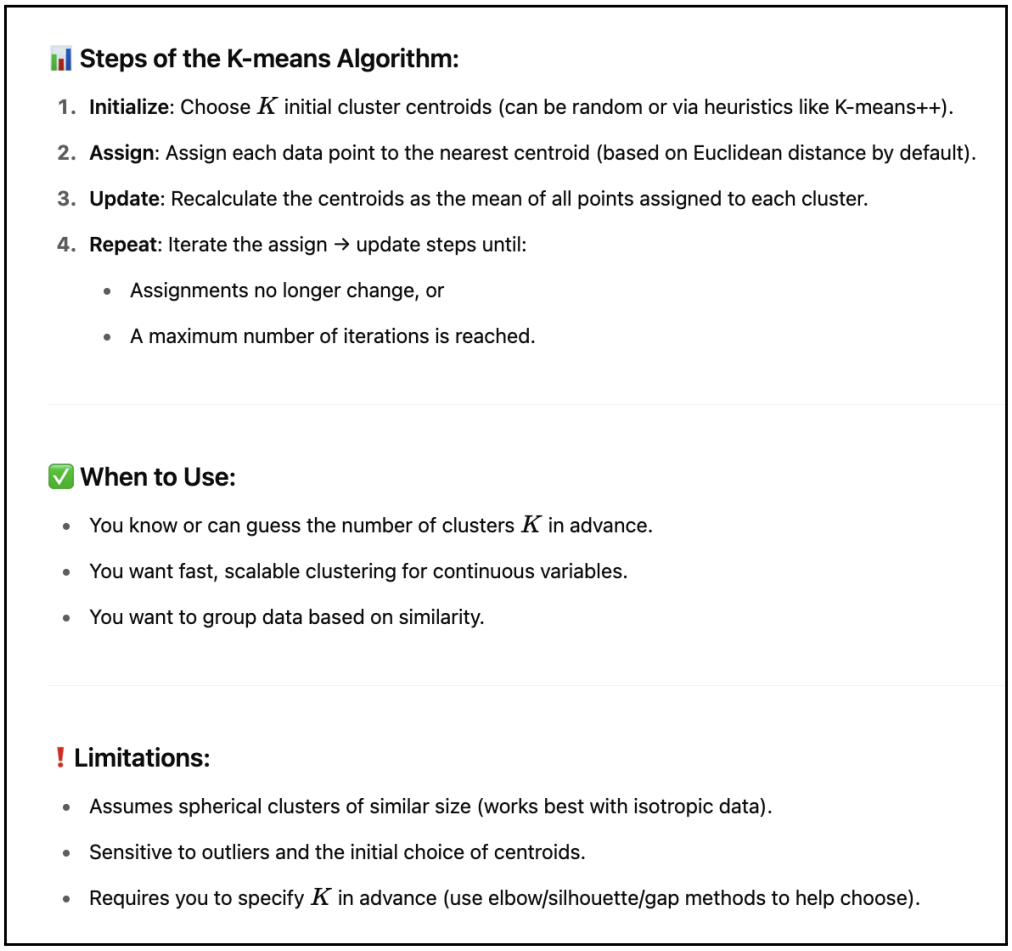
You often use `fviz_nbclust()` from the `factoextra` package with method = `"wss"` to visualize this.

Let me know if you want a plot or code example.

K-means

In our last demonstration, we looked at hierarchical clustering, which is ideal when you want to explore many different ways of clustering data. It begins with each observation in its own individual cluster and then progressively merges clusters, allowing exploration of the full hierarchy. However, there are situations where you already know how many clusters you want—perhaps because you are working with three known species or conducting audience segmentation in marketing with resources for only three campaigns. In such cases, *K-means clustering* is appropriate, where K is the number of desired groups, and the algorithm assigns observations to clusters based on proximity to the cluster means in multidimensional space. To demonstrate this, we return to the `penguins` dataset. We begin by loading the necessary packages, setting a random seed for reproducibility, and loading the data. As before, we reduce the dataset to 100 observations to facilitate visualization. After confirming the data structure, including four physical measurements and a species label, we rename the species column from `Y` back to `species` to aid visualization. The quantitative variables are standardized and saved as a new dataframe `df`. To determine the optimal number of clusters, we reuse methods from earlier: the elbow method (which shows a bend at three), the silhouette method (which suggests two), and the gap statistic (which suggests up to five). Given our domain knowledge of three species, we proceed with $K = 3$ and apply the `kmeans` function, saving the output as `km`. The result includes cluster means, assignments, and within-cluster sum of squares. For visualization, we use `fviz_cluster` to display the clusters, add point and text labels, and observe the output. Unlike the hierarchical clustering result,

K-means splits the Gentoo penguins into two distinct clusters, while grouping the Adelie and Chinstrap penguins into a single cluster. This divergence suggests that K-means, based solely on the four physical measurements, struggles to align perfectly with the known species. It implies the potential value of adding more features to better distinguish the species, especially between Adelie and Chinstrap. Despite this, K-means remains a simple and flexible method that adapts clusters based on the structure of the input data, providing both insightful and practical results, especially in real-world applications such as market segmentation.



📊 **Steps of the K-means Algorithm:**

1. **Initialize**: Choose K initial cluster centroids (can be random or via heuristics like K-means++).
2. **Assign**: Assign each data point to the nearest centroid (based on Euclidean distance by default).
3. **Update**: Recalculate the centroids as the mean of all points assigned to each cluster.
4. **Repeat**: Iterate the assign → update steps until:
   - Assignments no longer change, or
   - A maximum number of iterations is reached.

✅ **When to Use:**

- You know or can guess the number of clusters K in advance.
- You want fast, scalable clustering for continuous variables.
- You want to group data based on similarity.

❗ **Limitations:**

- Assumes spherical clusters of similar size (works best with isotropic data).
- Sensitive to outliers and the initial choice of centroids.
- Requires you to specify K in advance (use elbow/silhouette/gap methods to help choose).

DBSCAN

The final demonstration of clustering is a newer method called DBSCAN, which stands for density-based spatial clustering of applications with noise. As I've mentioned elsewhere, it's really good for finding clusters with unusual shapes. Most methods are looking for things that are circular or spherical, but DBSCAN can find very different kinds of shapes because of the algorithm it uses for finding density within the multi-dimensional space. To do this, I'm going to start by loading a few packages, the most important of which is this one right here, `DBSCAN`. There is more than one package for doing DBSCAN in R, but this one's particularly fast, and so it's helpful. I'm going to set the random seed to one—you can use any number you want, it just makes things reproducible—and then I'll import the penguins dataset that we already cleaned up elsewhere. I'll save that to `df`. You see over here in the environment, we now have that: 342 observations, and we take a quick look at the data. We've got our factor, that's the species, it's `y`. Then we have four physical measurements. Let's get a quick summary of the data. So we have an uneven distribution among the species, and you can see that the measurements are on different scales. We're going to do the same two things we've done elsewhere. First, we're going to remove the class labels—that's the species label `y`. So we're going to pull that, which brings it out as a vector, and I'm actually going to save it into

one called `species`. Then we're going to take the rest of the data frame, remove the species, and scale or standardize all the variables so they have a mean of zero and a standard deviation of one. That gives them equal influence in the algorithm. Now we come down to the actual DBSCAN. There are two parameters that we have to specify. One is called `minPts` and the other one is `eps`, and you get to choose each one of these. So one of them is kind of an arbitrary choice. We get to choose a value for `minPts`, usually just called k. This is the minimum number of neighboring points for clustering. This value should be odd so you don't end up with ties, and it should have a value of at least 3, with higher values for larger datasets. Our dataset's only 342 cases, so it's not really large, but we'll use 5—that's an arbitrary choice. Second, once we've chosen `minPts` (with $k = 5$), then we find the optimal value of `eps`, which is actually just short for "epsilon neighborhood radius." That looks at the graphing distances—how far in the multidimensional space the algorithm should look for neighbors to cluster each point with. So we're going to graph the distances and look for a pronounced knee or bend in the chart. I'll take our data frame (we've already scaled it), but we'll run it through this `knndistplot` with $k = 5$. You can see that this one pops up right here as it gets the points sorted by distance. To me, it looks like there's a bend right about here. And because I've done this before, I know that corresponds to a value of 0.8. So I'm going to draw a line, and let's zoom in on that for just a second. This over here means 5 nearest neighbors—that's the $k = 5$ that we chose—and how far it is to each one. The 0.8 seems like a reasonable one. You could go a little higher, you could go a little lower, but you can try different values. I'm going to use the 5. Now what I'm going to do is actually run the DBSCAN algorithm using these two parameters: the `minPts`, that's the number of nearest neighbors (5), and `eps`, the epsilon or the diameter in which it looks (that's the 0.8 we got from the graph). That's pretty quick. I'm going to save it to an object called `db` for DBSCAN. You can see that over here now—it's a list of three. Let's print the object, and actually, all it's going to tell us is that it decided there were two clusters (labeling them 1 and 2) and five noise points. Those are 0. Clusters here mean "not in a cluster." Then we have some other information: cluster, `eps`, and `minPts`. What I'm going to do now, just as our final step, is visualize these clusters using the `fviz_cluster` command that we've seen before. So I take the `db` object for the DBSCAN we just created, feed it into `fviz_cluster`, tell it the data is in `df` for data frame, we're going to do points, and then we'll add labels for each of these elements in there. Now it's going to be a complicated graph, a lot of overriding, but when we zoom in on it you'll be able to see, first off, that we have two clear clusters. Right now it's only showing us two dimensions, even though I gave it four to work on, so we're not seeing everything here. But you can see that all the Gentoo penguins are over here grouped together, and that the two other species, the Adelie and the Chinstrap, are over in this other cluster. And then we have five noise points. These are the black dots that—well, these three, for instance—fall directly within this cluster, but because there are two other dimensions, they are separate from that cluster in those other dimensions. DBSCAN is a good way of getting non-spherical clusters that could have shapes like being long and rectangular, or ring-shaped, or going up and down. You can see that we're pretty close to a convex shape on these ones anyhow, but the actual algorithm is pretty quick and easy to implement in R.

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a clustering algorithm used to identify groups of points in a dataset based on density, rather than shape or size. It's particularly good at finding **non-spherical clusters** and can detect **outliers (noise)** that don't belong to any cluster.

Key Ideas:

- **Clusters = dense regions** of data points.
- **Noise = sparse regions** (isolated points).
- Doesn't require you to specify the number of clusters beforehand (unlike k-Means).

How It Works (Intuition):

1. **Pick a point** in the dataset.
2. Count how many **neighbors** it has within a certain distance (`eps`).
3. If it has **enough neighbors** (≥ `minPts`), it becomes a **core point** and forms the seed of a cluster.
4. DBSCAN then **expands the cluster** by connecting nearby points that are also dense.
5. Points that aren't close to any dense area are labeled as **noise**.

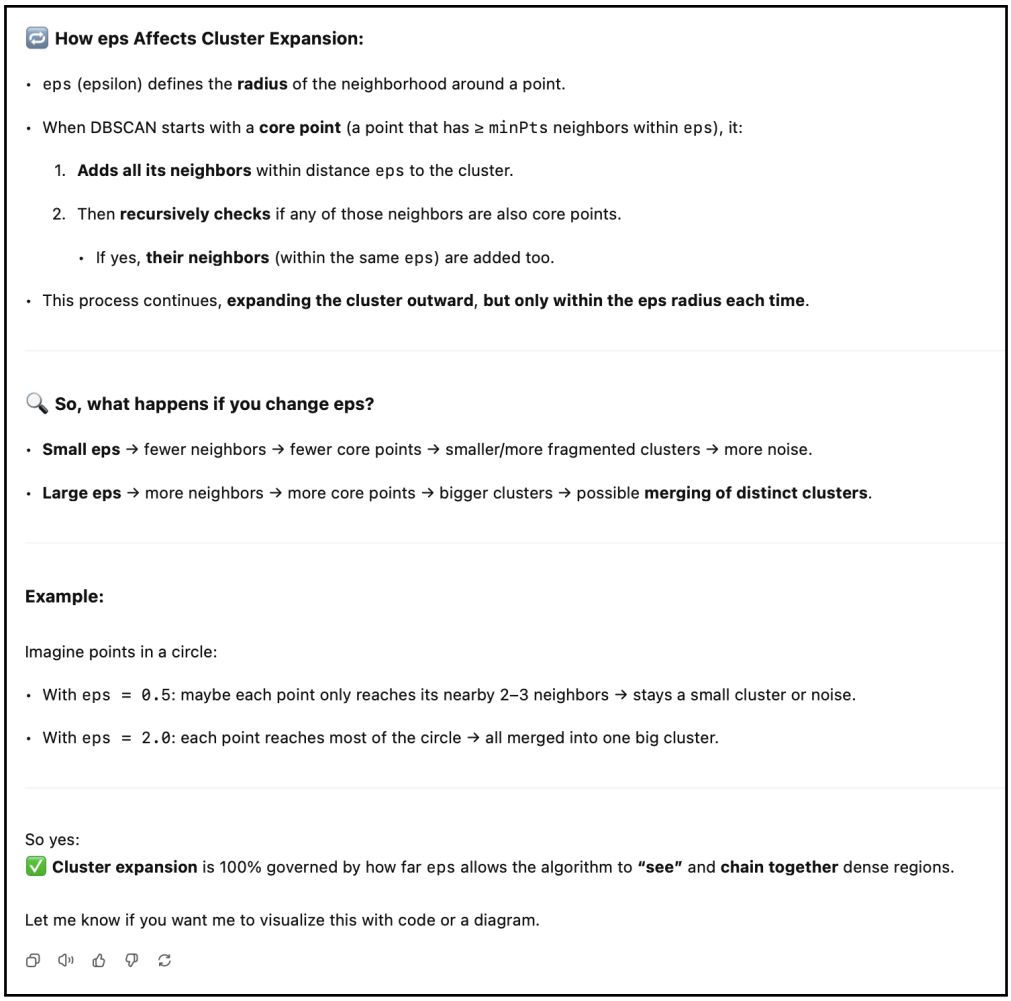
Parameters:

- `eps`: radius to search for neighboring points (like a clustering "reach").
- `minPts`: minimum number of points needed to form a dense cluster.

Why Use It?

- Handles **arbitrary-shaped clusters** (e.g., rings, spirals).
- Detects **outliers** automatically.
- No need to guess the number of clusters ahead of time.

Let me know if you want a visual or code demo.



🔄 **How eps Affects Cluster Expansion:**

- `eps` (epsilon) defines the **radius** of the neighborhood around a point.
- When DBSCAN starts with a **core point** (a point that has ≥ `minPts` neighbors within `eps`), it:
  1. **Adds all its neighbors** within distance `eps` to the cluster.
  2. Then **recursively checks** if any of those neighbors are also core points.
     - If yes, **their neighbors** (within the same `eps`) are added too.
- This process continues, **expanding the cluster outward**, **but only within the eps radius each time**.

🔍 **So, what happens if you change eps?**

- **Small eps** → fewer neighbors → fewer core points → smaller/more fragmented clusters → more noise.
- **Large eps** → more neighbors → more core points → bigger clusters → possible **merging of distinct clusters**.

Example:

Imagine points in a circle:

- With `eps = 0.5`: maybe each point only reaches its nearby 2–3 neighbors → stays a small cluster or noise.
- With `eps = 2.0`: each point reaches most of the circle → all merged into one big cluster.

So yes:
✅ **Cluster expansion** is 100% governed by how far `eps` allows the algorithm to **"see"** and **chain together** dense regions.

Let me know if you want me to visualize this with code or a diagram.

Challenge: K-means

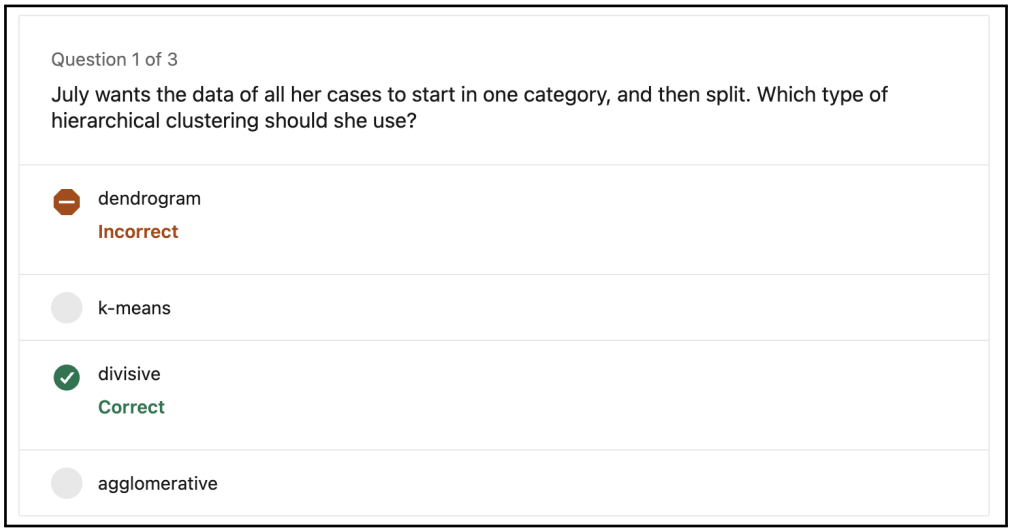
Now that we've covered several ways of clustering data, I want to give you a challenge and invite you to do some clustering of your own. The method we're going to use is K-means. And so this is the one where you specify in advance how many clusters you're looking for. And for this example, we're going to use an extremely well-known dataset that comes from the R built-in `datasets` package. So let's load all of these, including `datasets`, and then we'll use the `iris` dataset. It's been used millions of times. Let's get a little bit of information on this. And what this is, is Edgar Anderson's or Ronald Fisher's `iris` data. It's the measurement in centimeters of several different species of iris flowers. We can take a look at the first few lines. What we have are the sepal length and sepal width, the petal length and the petal width, and then the species. And we can get the entire summary statistics. And you can see these are on slightly different scales, not massively different. And then here are the three species listed here at the end. What we're going to do is a K-means cluster where we're going to set it up for three clusters – so K is equal to 3 – that hopefully will match these three species using these four measurements. Give it a whirl and then we'll meet again in just a moment to compare how we each did K-means clustering on the `iris` data. I'll see you there.

Solution: K-means

(upbeat music) – [Instructor] When I extended this challenge to you, I invited you to use the `iris` data to do k-means clustering with a value of three for k because there are three different species of iris. Let me show you how I solved this, and you can compare your own answers. Now, not surprisingly, this one follows the example we had very closely of k-means clustering earlier. So in fact, I'm going to be loading the same

packages here, including `car` for the `scatterplotMatrix`, `cluster`, `datasets`, and so on. And once those are loaded, I'm going to come down and get the `iris` dataset. I'm going to set the random seed because that can work into things, and I'll save the `iris` dataset to `df` as a tibble, so I've got it over there in my environment. We'll check the data quickly. There we have the four physical measurements and the `Species` class variable there at the end. Now, I'm going to separate the class variable. I'm going to use `pull`, and take `Species` and save it into its own variable. You can see it right here. It's a factor. And then, I'm going to overwrite `df` to remove `Species` and to scale the other variables. Even though they're all pretty close to the same scale, ranging in values from one point something to about seven in terms of centimeters, it is still helpful to give them all the same mean of zero and standard deviation of one, so they have potentially equal influence in the clustering. So I'm going to do that. Now, let's explore the data with `scatterplotMatrix`. These are the four physical measurements and the species. And we can zoom in on this very colorful chart. And we're just trying to see if there are patterns here. We see, for instance, there's a strong association between sepal length and petal length. We can see that there's one cluster over here. This is the `setosa`. That's very much by itself in terms of petal length and width. And it actually looks like we're going to be able to get nice clean separation between the species based on these physical measurements. So now, let's go to the k-means clustering algorithm. So I'm going to take the data frame that has just the four measurements. I'm going to use the `kmeans` function and specify three, that I only want three clusters. We're going to save it to `km` for k-means, and we're going to also print the output at the bottom here. So you can see that it's saved over here as a list. I'm going to make this bigger. It says that we have clusters of 53, 47, and 50. So we know something a little interesting has happened because in the dataset, there are 50 of each species. This gives us the k-means, the three different means for each of the groups that it created. Not the original species, but the three different clusters. And then, we have a clustering vector here that tells us what cluster it put each case in. And we've got a few other elements in terms of what sort of things are available to us. But what's going to be easiest to deal with is simply visualizing the clusters. So I'm going to take `km`, that's the object with the k-means results, send that into `fvix_cluster`, tell it we're using the data frames, we want points, and we're going to put labels on things. So let me run that one right now. And when we do that, you can see that we have `setosa` over here, we have `versicolor` and `virginica` in here, and this is mostly `versicolor` here but `virginica` over here. And so what we found is that the `setosa`, which was dramatically different in terms of two of the four measurements, is cleanly separated from the others, but there is some overlap between the other two, the `versicolor` and the `virginica` iris flowers. This dot right here is actually the mean in these dimensions for that particular cluster. And so we see that this was reasonably effective in identifying groups that could be set off separately. Now, one thing that you can do at this point is you can try some of the other algorithms that we used, and see how well they cluster the datasets, and to see if you can get a clean separation, for instance, between these two species that are very similar. Remember, clustering, even though we have a particular outcome in this variable, is usually a descriptive procedure, a measure of convenience. For instance, you only have enough money to run two different ad campaigns, so you can only have two different clusters. It's often more a matter of interpretation and usability when you're evaluating the effectiveness of the clusters. This is one method. If we simply wanted to isolate `setosa` from the others, then we've accomplished it. But again, I encourage you to explore the other procedures with the same dataset and to see what kind of insights you can get into the `iris` dataset.

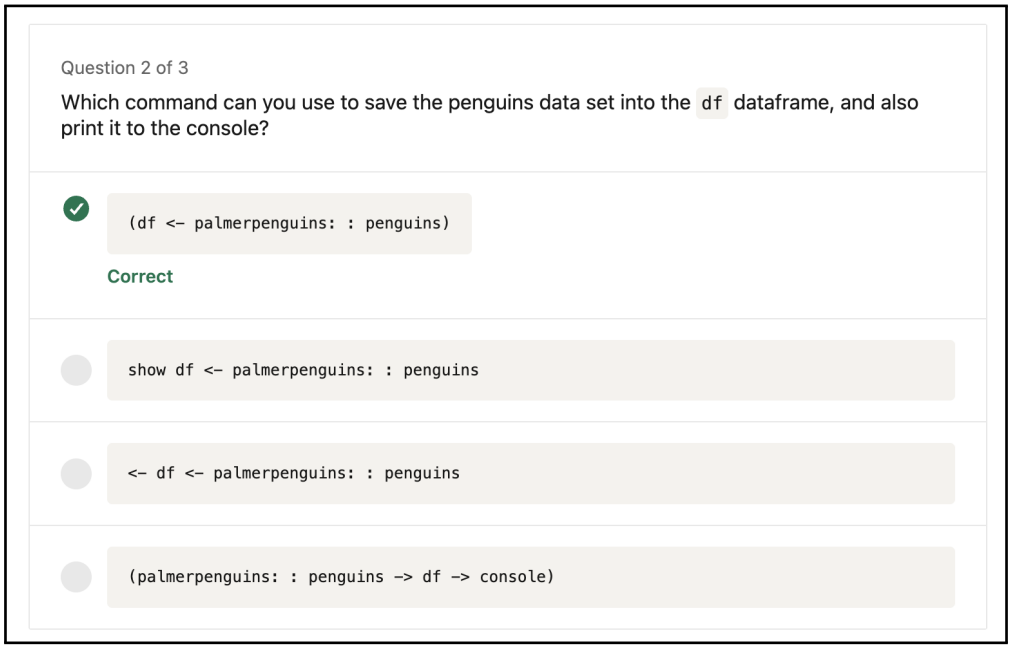
Chapter Quiz



Question 1 of 3

July wants the data of all her cases to start in one category, and then split. Which type of hierarchical clustering should she use?

- dendrogram
  Incorrect
- k-means
- divisive
  Correct
- agglomerative



Question 2 of 3

Which command can you use to save the penguins data set into the `df` dataframe, and also print it to the console?

- `(df <- palmerpenguins: : penguins)`
  Correct
- `show df <- palmerpenguins: : penguins`
- `<- df <- palmerpenguins: : penguins`
- `(palmerpenguins: : penguins -> df -> console)`

Question 3 of 3

Which code block correctly represents hierarchical clustering showing a dendrogram?

```
df %>%
    fviz_nbclust(
        FUN = hcut,
        method = "wss"
    ) +
    geom_vline(
        xintercept = 3,
        linetype = "dotted"
    )
```

```
hc <- df %>%
    dist %>%
    hclust
    ) +
    geom_vline(
        xintercept = 3,
        linetype = "dotted"
    )
```

```
df %>%
    fviz_nbclust(
        FUN = hcut,
        method = "wss"
    )
df$labels <- y
hc %>% plot(
    color = "red"
)
```

```
hc <- df %>%
    dist %>%
    hclust
hc$labels <- y
hc %>% plot(
    hang = -1,
    cex = 0.6
)
```

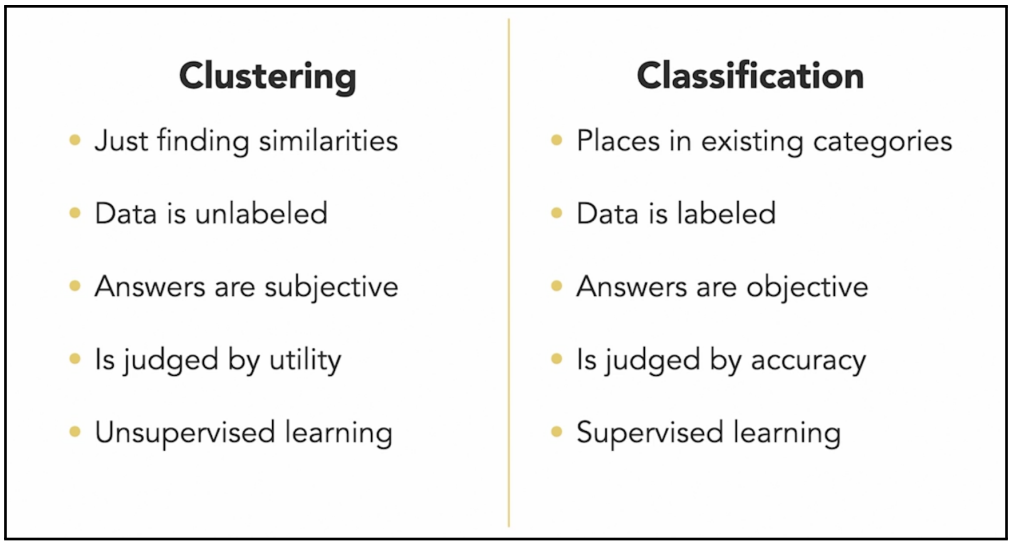
Correct

Classification

Classification overview

We've talked about clustering, which is putting similar things in similar piles, but sometimes similar is not enough. For example, when it comes to the mail, you can group things by size or shape or postage, but that's not what you're going for. Your one job is to put things into the right box. So if you want to think about clustering versus classifying, which are probably the two most important things happening in data science, clustering is just trying to find similarities, and the data is referred to as unlabeled because it doesn't have a correct category that you're trying to match. That also means that the answers are subjective. What that really means is you could do things in different ways, and the usefulness or the utility is the outcome by which things are judged. It's also an example of what's called unsupervised learning. Unsupervised means, again, there's not a correct category here; it's just trying to find things that are relatively similar to each other. Contrast that with classification. This is where you have existing categories, like this is this person's mailbox, or this is a photo of a toy, this is a photo of a tree. You want to place new objects into

existing categories. And so the data here is labeled, which means there actually is a correct answer, and in many cases, it's there in the data. Also, the answers are more or less objective because there is a correct answer. Does this mail go in this mailbox, or does it go in that one? And in that case, you can judge the effectiveness of the algorithm by how accurate it is. Were the positive cases labeled as positive, were the negative cases classified as negative? And it's also generally referred to as supervised learning because you have this label, this correct category, that is used to teach the algorithm how to process the data to get to that final outcome. Now, there are several methods that are used for classification; in fact, there's an enormous number. But I want to show you a few that are really common. The first one is called k-NN, which stands for k-nearest neighbors, where k is a number that you choose—it could be 3, it could be 5, it could be 11. And the idea here is that existing categorized cases are located in a multidimensional space. If you have 10 variables and you have 10 dimensions, then each dot in that 10-dimensional space is where a known observation goes. And then what you're trying to do is locate a new case in that 10-dimensional space and compare it to the k closest points in that physical space. Those are its nearest neighbors. You generally choose an odd number, maybe from 3 to 9 to 21, and you simply assign that new case to the most common category among its k nearest neighbors. It's like deciding with berries. If you find a berry in the wild—these are all berries—would you eat them? Now, some of these I know what they are. I know blueberries, I know raspberries, I know blackberries. But two of those are not very familiar to me, and I would probably stay away from them because I do not have a ready categorization system for deciding whether they're safe or not. Another approach that's often used in classification tasks is called naive Bayes. This uses a formula developed by the Reverend Thomas Bayes—that's why it's called Bayes—to calculate probabilities. The major breakthrough that Bayes had was a method that combines base rates (how common something is overall) and prior probabilities to get posterior probabilities. So it's often used, as an example, in diseases. How common is a disease overall? That's its base rate. And if a person has a disease, what's the probability of getting a positive test result? That's a prior probability. The posterior probability is, if somebody gets a positive test result, what's the probability they have the disease? And that's usually the one that most people are interested in. And so it's used in a lot of situations, including classification. It's called naive not because it's simplistic but because it assumes that each of the predictors operates independently. It's like rolling these dice over here—you roll them, and what happens on one does not affect what happens on the others, but you can calculate the probabilities between them. And the last method I want to show you is decision trees. This is a series of yes/no decisions—binary decisions—and the algorithm chooses the most informative split at each point. Think, for instance, if you're playing the game Hangman, where you're guessing a word, you choose letters that are diagnostic. E is going to be the most common; A—those will give you an idea what's probably going to be in the word. On the other hand, if you first guess X and it says no, you haven't eliminated very much—that wasn't an informative choice. So the algorithm goes through the data, chooses the most informative split at each point. The neat thing about it is it works with basically any kind of data and almost any kind of pattern. It can use quantitative data, it can use categorical data, it can use binary or multiple categories, it can use skewed data—it can work with almost anything, often in its raw condition. Also, because you have this graphical output, decision trees are exceptionally easy to interpret. So, there are many, many more choices. You don't have just one pencil in one color—you've got a lot. This is because classification is one of the most important tasks in data science and an area of very active research and development. And in data mining, you do have a choice of dozens or possibly hundreds of methods. But the three that I mentioned are the ones that I want to demonstrate that I think will be probably of greatest use to you as soon as possible.



Dataset: Spambase

For our demonstrations of classification, we're going to be using the large dataset that is called `spambase`. And to show this to you, I'm going to first load a few packages, including one that gives us some plotting options. The dataset `spambase` comes from the machine learning repository at the University of California, Irvine. This is the URL to that site. The `spambase` dataset's URL is right here. Now you can download the data—we're using the one that's called `spambase.data`. I have already downloaded it in a way that saves the variable names, but you can download it using this command if you'd like. I've already done that and saved it here in the `data` folder. So we can simply say import from data `spambase.CSV`, save it as a tibble into an object called `df`. And you can see it's got 4,600 observations on 58 variables. Now let's look at the variable names. You can see that they're cryptic. Each of these means something, like how many capital letters there are, or how many spaces in a row, or how many numbers, or does this word appear? Does that word appear? And that can be useful, but because we're simply doing a data mining project where we actually want to see if any of these variables work to predict whether something is spam or not, I'm going to clean it up a little bit. First, I'm going to rename the variables so they're just `A1--A57`—fine. Then we look at the variable names—still cryptic, but at least it's more orderly. Then we're going to make a couple of changes. The last variable, `A58`, is the one that has the class variable that indicates whether something is spam or not. So we're going to rename that as `y`, because I often just call my class variables the outcome `y`—that lets me reuse code. Then we're going to change that from zeros and ones to `not spam` and `spam` using the `ifelse` command. So if `y == 0`, then it's `not spam`; if it's not equal to 0, then it's `spam`. Then finally we'll record it as a factor. We do those things, and then finally, let's check that class variable by pulling a factor count: 2,788 `not spam`, 1,812 `spam`. Now, it's very common in classification methods—and in machine learning in general—to do training and testing. So we need to split the data. Because it uses randomization, I'm going to set the random seed here. And just in case there are two rows of data that look the same, I don't want R to throw one of them out. So I'm going to start by creating an ID number so each row will be unique. Then I'll take 70% of the data randomly using `slice_sample` and put it into the training dataset. Then the testing dataset will be the `anti_join`, meaning whatever's not in `train` goes into `test`, matching by `ID`—that's our unique variable—and then delete the `ID` variable from everything else. And now we have our two mutually exclusive, but comprehensive, datasets of `training data` and `testing data`. Now I do want to take a moment and explore the data just a little bit. We're going to get a bar chart here of the class variable, whether something is spam or not. And you can see `not spam`, we've got more of it. And we've got a fair amount of `spam`. It's nice that it's not like 99% and 1%. And then let's take a look at a few randomly chosen variables and see how they relate to our class variable, `y`. We'll do this with `ggpairs`. It takes just a moment to do this one. And then I'll zoom in on it. And so this one right here, the orangish/coral color is `not spam`. And this one, the blue/green/teal is `spam`. What we're trying to do here is to see whether these two groups differ on anything. And you can see, yeah, the distributions look

different on these variables. And so we may be able to distinguish these things rather nicely. We can also get a stacked histogram of a few variables and this is going to be useful to just see what the sparse nature of the data is—what that is is, if I zoom in on this, there's a lot of things here that we've got empty space in the variables, but those can be used to distinguish the values. I know it's hard to tell if there's anything there, but that's because we have a large dataset, relatively large. And then finally that we've done this work, we're going to save these datasets in the native `RDS` format. That's R data serialized. We'll save the entire data frame. We'll save the training dataset and the testing dataset. And then we can get started with our three demonstrations of clustering in R.

K-nn

The first method of classification that we'll look at is actually one of the simplest in theory, and that's kNN, which stands for k-nearest neighbors. This is really simple to execute in R. To do this, I'm going to load a few packages including `caret`, which has several testing and training functions, `e1071`, which actually comes from a class name and has several machine learning functions, as well as the other packages we normally use. I'll load those and also set the random seed because there's going to be some randomization. Then, let's import our two datasets that we already prepared: the training dataset from `spambase` and the testing dataset, `tst`. We're going to compute our model on the training data. First, we have to tell the algorithm how to do this by setting up `trainControl`, which defines methods for the training algorithm. We'll be using `repeatedcv`, or repeated cross-validation. Cross-validation means splitting the dataset into folds; here, we'll have five folds and three repetitions. It will repeatedly use one fold for testing, the others for training, and rotate through to find the best model—so it's a thorough strategy. Then we specify the parameter grid for values of k using a sequence of odd numbers from 3 to 19 (i.e., `seq(3, 20, 2)`). I intentionally excluded 1 because choosing only the nearest neighbor tends to cause overfitting. We ensure at least three neighbors are used. Next, we apply the model using the `train` function, where `y` (our class variable for spam) is modeled as a function of everything else. We specify `method = "knn"`, pass the parameter grid, and preprocess the data by centering and scaling it. The model may take time depending on your machine. Once trained, the model object is saved as `fit`. To determine the best k, we plot `fit`, which shows variation in accuracy across k values—though when rescaled from 0 to 1, differences appear small. Printing `fit` gives the final chosen model: with 3,220 observations, 57 predictors, and two classes (spam/not spam), and it selects $k = 7$ with highest accuracy (≈ 0.9534). Then we apply the model to the test data using `predict`, saving the output as `pred`. We compute a confusion matrix (`CM`) comparing `pred` to true values in `tst$y`, and visualize it with a fourfold plot. In this plot, blue rectangles represent correct classifications: true positives (spam correctly identified) and true negatives (not spam correctly identified), both larger than false positives/negatives. The confusion matrix gives a numerical breakdown: accuracy is 91% (0.9109), with 95% sensitivity (true negative rate) and 85% specificity (true positive rate). These metrics will be stored as part of our evaluation summary. This is the first of three classification models we'll run on this dataset, so we can compare.

🧠 **What is k-NN?**

k-NN is a classification algorithm (though it can also be used for regression). It answers this question:

> "Given a new data point, what class (category) should I assign to it based on what I already know?"

It does this **by looking at the "k" closest known examples** and assigning the most common class among them.

🌰 **How it works (step-by-step)**

1. **Store the training data.**
k-NN doesn't *train* like other models. It just memorizes the labeled examples.
2. **Choose a number k** (usually an odd number like 3, 5, 7).
3. **To predict the class of a new point:**
   - Measure the *distance* between this new point and every point in the training data (usually using Euclidean distance).
   - Find the **k nearest neighbors** (i.e., k points that are closest to it).
   - Check what **classes those k neighbors belong to.**
   - Predict the class that appears **most often** among those neighbors.

📦 **Example**

Let's say we have a dataset of fruits with features like weight and color. Some are labeled **"apple"**, others **"orange"**.

If a new fruit comes in and it's similar in color and weight to some known apples and oranges, k-NN looks at the **k closest fruits** and predicts the label based on which class dominates among them.

🛠️ Key Concepts

- **k:** Controls how many neighbors are used.
Small k = more sensitive to noise (overfitting)
Large k = smoother decision boundary, but might underfit.
- **Distance metric:** Commonly used is Euclidean distance. Others include Manhattan, Minkowski, cosine similarity, etc.
- **No training phase:** It's called a **lazy learner** — it just stores data and defers computation to prediction time.

🔍 When to use k-NN

✅ Simple to implement
✅ Works well with small datasets
✅ Interpretable

⚠️ Not great with:

- Large datasets (too slow)
- High-dimensional data (distance gets noisy)
- Imbalanced classes

Let me know if you want visuals or code examples in R or Python.

2. k-NN is majority-vote-based

k-NN uses the labels of the **nearest neighbors**. If most of the nearby points are from the **majority class**, the minority class is drowned out, even if the point is truly from that minority.

So if 19 out of 20 nearest neighbors are **not spam**, k-NN will always vote **not spam**, even for actual spam.

Naive Bayes

It's strange to talk about something being naive when we're trying to do serious data science, but Naive Bayes, which again means that it is treating each of the variables as independent, is an important and useful method for classification exercises in data science and here in data mining. To do this, I'm going to start by loading a few packages, including this one, Naive Bayes, along with some of our other testing functions. I'll set the random seed, and I'm going to import the training and testing datasets that we got from the spambase dataset, so `trn` for train and `tst` for test. I'm then going to set up some control parameters. I'm going to be using the same information I used in the k-nearest neighbors model. It says we want to do repeated cross-validation with five folds and three sets of folds. This gives us a very comprehensive way of doing internal testing with our training data before we move on to the final validation with the testing data. When you do Naive Bayes, you have a number of parameters you can set up on what's called a search grid where it's looking for different ways of constructing the model. We have, for instance, the Laplace correction or Laplace smoothing, which is used for the categorical data. They have several different values here that I can try out. We have whether it should use a kernel method and whether it should have an adjustment for the bandwidth as it goes through. I've got some likely values for each of these, and we're asking it to use all of these combinations as it tries to build the best model. So I'm going to save that set of information and then I'm going to come down and build our `nb` for Naive Bayes object. Use the `train` function, say we want `Y`, that's the class variable, whether an email is spam or not, as a function of everything else, use the training data. The method is Naive Bayes, use control parameters, use the grid parameters, and omit cases that have missing values. This is actually very similar to the command that we used when we did k-NN. I'm going to run this, it takes a moment, so depending on your machine, you may have a second to go check some of your own spam email. When it finally pops up with the results, we can then plot the parameter values against accuracy. And let's zoom in on this. What this is showing us is, for instance, whether it's doing the Laplace correction, going from values of zero up to one, and whether it's using the `adjust` or whether it's using the `kernel`, and we get this accuracy over here. And what we're looking for is what is highest on the top, and truthfully, it looks like the ones that are not using the kernel are the highest and that the adjustment and the Laplace correction don't make a difference, so we probably leave those at their minimum values. And in fact, we can simply ask R, what's the best model, which one should we use? And here's the data that it just graphed for us. Here's the description of the commands that it did, the predictor, the classes, the cross-validation, whether it's using the kernel or adjusting. And this says the final model that it chose was: Laplace correction is equal to zero, using the kernel is set to false, and the adjust is left at 0.1. And so having developed this Naive Bayes model with our training data and with extensive cross-validation, we can now apply it to our testing data. We do this in the exact same way we did before: we use `predict`, we tell it what model we're using, we saved it as `nb` for Naive Bayes. And to apply it to new data, the `tst` for test data, and save that as `pred` for predicted. We do that, and then we can get the confusion matrix which shows us the accurate and inaccurate predictions. We can then plot that matrix using the fourfold plot, it's a nice visualization of it, where the light blue tells us correct predictions and the red tells us incorrect predictions. And what we can see is, we've got some incorrect predictions, specifically this one right here with the 374, that's a little problem. Let's get the confusion matrix printed and look at it in a little more detail. So what we have here is probably not the spam filter you would want to use. So it predicts something as spam, but it's only slightly more likely to actually be spam. We've got a lot of false positives. On the other hand, it's only letting a very small number of spam messages through thinking that they are not spam. So it's not going to have a lot of spam in your inbox, but it's going to have a lot of legitimate messages in your spam folder. So that's the thing you're going to want to look at. The overall accuracy is 71%, that's not really good. And you can see that the sensitivity that has to do with the false positives is really low, but the specificity in terms of letting spam into your inbox, this is performing very well. And we can compare this with the other model we had, the k-nearest neighbors. You can see here, 71% overall accuracy to 91. So at

least on this particular dataset, the k-nearest neighbors is performing much better. On the other hand, there are different situations in which each model will perform better. So I strongly encourage you, anytime you can, to try as many different models for your classification to see what works best with your model. That's going to give you a better range of options and hopefully better performance as you go through both your classification and the data mining to find out what works best as predictors.

$$\therefore P(E \text{ given } F) = \frac{|E \cap F|}{|F|} = \frac{|E \cap F|/n}{|F|/n}$$

$$= \frac{P(EF)}{P(F)}.$$

* **Definition** Given any sample space S, let $F \subseteq S$ be such that $P(F) > 0$, we define

read as Prob. of E given F

$$\boxed{P(E|F) = \frac{P(EF)}{P(F)}}$$

b) Independent Events

Def: We say two events E & F are **independent** if $P(EF) = P(E)P(F) \iff P(E|F) = P(E)$
$\iff P(F|E) = P(F)$.

In other words, knowing that F occurs or not does not change our perception of $P(E)$.

🔁 Conditional Probability Definition:

$$P(C \mid X) = \frac{P(C \cap X)}{P(X)} \quad \text{and} \quad P(X \mid C) = \frac{P(C \cap X)}{P(C)}$$

From that, you can rearrange:

$$P(C \cap X) = P(X \mid C) \cdot P(C)$$

Plug into the first equation:

$$P(C \mid X) = \frac{P(X \mid C) \cdot P(C)}{P(X)}$$

🔁 Core Idea

Naive Bayes is a classification algorithm based on **Bayes' Theorem**. It calculates the probability that a data point belongs to each class and selects the one with the **highest posterior probability**.

The formula is:

$P(C \mid X) = [P(X \mid C) * P(C)] / P(X)$

Where:

- **C** = class label (e.g., spam or not spam)
- **$X = (x_1, x_2, \ldots, x_n)$** = input features (e.g., word frequencies)
- **$P(C \mid X)$** = posterior probability (what we want to calculate)
- **$P(X \mid C)$** = likelihood (how likely the features are under a class)
- **$P(C)$** = prior probability (how likely the class is overall)
- **$P(X)$** = evidence (same for all classes, so we can ignore it when comparing)

🤓 Why "Naive"?

Because it assumes that **all features are conditionally independent** given the class.

That means:

> **$P(X \mid C) = P(x_1 \mid C) * P(x_2 \mid C) * \ldots * P(x_n \mid C)$**

This assumption is almost never true in real life — features often influence each other. But surprisingly, **the method still works very well** in many practical situations.

- ✅ **P(X)**: the **evidence** = how likely this input X is **in general**, no matter what class

✅ When to Use It

- Text classification (spam filters, sentiment analysis)
- Works well with **high-dimensional** data (lots of features)
- Very fast to train and predict

❌ Limitations

- Assumes feature independence → performs poorly if features are **highly correlated**
- Can't capture complex interactions between variables

🔧 **Real-World Example**

You're trying to figure out **if an email is spam (C)**.

You have two features:

- x_1 **=** The word "free" appears
- x_2 **=** The word "win" appears

These two features (x_1 and x_2) are **not independent** — spammy words often appear together.

BUT — Naive Bayes **pretends**:

> "If I already know this email is spam, then whether it contains 'free' doesn't affect whether it contains 'win'."

Formally:

> **$P(x_1 \text{ and } x_2 \mid \text{spam}) = P(x_1 \mid \text{spam}) \times P(x_2 \mid \text{spam})$**

That's **conditional independence**:

> We assume x_1 and x_2 are independent **after we know the class** (spam vs. not spam).

🧠 **Why Does Naive Bayes Do This?**

Because this assumption **makes the math fast and simple**.

Instead of needing tons of joint probabilities (like P(free and win and click | spam)), it just multiplies the individual ones:

> **P(free | spam) × P(win | spam) × P(click | spam)**

It's **not accurate**, but it often works surprisingly well.

Decision trees

The final method for classification that we're going to look at—there are many, many other possibilities—but the one that we're going to look at in this data mining course is decision trees. This is actually one of the most useful ways of visualizing what's happening in the decision process. To do this, I'm going to load a few functions: `caret` and `e1071`, which I use for almost all classification, as well as `rattle`, which makes very pretty plots for decision trees. So I'm going to load those packages and set the random seed because we have randomness going on in this. Then I'll import the training dataset and the testing dataset that we created earlier. We need to do the training control parameters. These are the exact same parameters we've had before that say repeated CV for cross-validation, five folds and three sets of folds. Then we're actually going to do the decision tree itself. Again, we use the `train` function, say `y`, that's whether something is spam as predicted by—that's what the tilde means. Dot means everything else in the dataset. Use the training data, and then we're actually going to do two kinds of decision trees. The first one where we specify `rpart` here—that means that we want to have our try tuning what's called the complexity parameter. Then we tell it to use these control parameters that we set earlier for the cross-validation, and the `tuneLength = 10` says try 10 different parameters when it's tuning the complexity parameter. So we're going to do this and it'll take a moment to actually do all the calculations. And then we can look at the processing summary for this first one; `dt` is for decision tree one. And so we look at that, and we have the sample size, number of predictors, that we have two classes in the outcome variable, five folds repeated three times. And then we have the accuracy of the various complexity parameters. And it said that the final value was this one up at the tippy top—the 0.7751938 for the complexity parameter gave us an accuracy of 89.44, and that's really

close to the others but it is higher. Now what we'll do is we will plot that. We'll simply take it, feed it into `plot`, and you can see that, yeah, there's a difference there but again it's a compressed scale. So if we do it on the 0 to 1 range, there's a difference. And this one right here, the complexity parameter—we'll leave it right there. Now we're going to do a second decision tree. So I'm going to save that as `dt2`, decision tree two. The difference in this one is this: instead of `rpart`, we have `rpart2`. And this says instead of changing the complexity parameter, try the maximum tree depth—how far it can go, how many branches it can have. And here we're saying try up to 10. So we run that model and then we can look at its processing summary, and this one increases as it adds more levels to the tree. We can plot that. You can see it increasing as the tree depth—the more decisions that are included in the tree—and then we can show that on the 0 to 100 range, it's still a detectable increase. And then just as a little trick, we're going to actually create an object called the final decision tree, `finaldt`. And it says, well, if the accuracy for the first model is greater than the second one, then put the first model in here, otherwise put the second model in. It's just an automated way of comparing the highest accuracy of the two. We can get the description of the final training model. And this is what the text version of a decision tree looks like. It tells us how many cases we have that are spam and not spam. And then it starts looking at the variables. Now, unfortunately the variables here have these cryptic names: A52, A7, and so on. If you were to go back to the original file description, you could see what those actually referred to. Again, it's like: does this word appear, or how many capital letters are there, how many numbers are there. And those are the predictors. But a better way to look at this is with a graph. And we're going to plot it using the `fancyRpartPlot`. And when we do that, I can zoom in on that. This is our decision tree, where we have each node and you can see it makes a decision at each level, branches off to the left or the right depending on the specific criteria for that decision. And it says not spam, spam, not spam, getting down until we get to our classification here. And that gives us our overall decision process. And so it turns out that decision trees are a really great way of doing this because you have this visual output. But we can also take this model and we can apply it to our testing data, which is something we've done with our other approaches. So we're going to take our final model here and we're going to apply it to the new testing data. Save it to `pred` for predicted. Then we'll get the accuracy, the confusion matrix. And we can plot that confusion matrix. The blue ones are the true positives—meaning they really are spam—and the true negatives—meaning they really aren't spam. The red are the false positives and false negatives, and you can see we've got pretty good accuracy here. But if you want the actual numbers, we can print the result. And we have a 90.29% total accuracy with good levels of both sensitivity and specificity. And if you want to compare these, we can come up here—I'll make that a little larger here. This is very similar to the accuracy we had overall with the K-nearest neighbors, and both are much better than the Naive Bayes. Although interestingly, the highest specificity was with the Naive Bayes. And we have small variations in these. So you can use the decision tree because you can actually see a graphic of what's happening, or the kNN which is conceptually the simplest by simply looking at neighbors in a multidimensional space. But any one of these is going to get you very far and where you need for classification using these simple datasets. But they are eminently scalable—they can go to much larger, more complex datasets and variations on them. A good method, again, is to try all three of these when you're doing the classification task to find the method that works best for your particular dataset.

Information Gain (aka mutual information)

$$IG(X,Y) = H(Y) - H(Y \mid X)$$

Intuitively, amount of information we learn about the class label Y, given a specific feature X

Want to pick feature X that maximizes information gain

- Suppose the feature X and class label Y are independent random variables Then $IG(X,Y) = 0$

recall if r.v. X and Y are independent $P_r(X,Y) = P_r(X)P_r(Y)$ In general, $P_r(X,Y) = P_r(Y|X)P_r(X)$

uncertainty in random variable Y (labels)

it's a notion of uncertainty

$$H(Y) = -\sum_{i=1}^{k} \Pr(Y = y_i) \log \Pr(Y = y_i)$$

uncertainty in labels Y given a specific feature X

$$H(Y|X) = \sum_{j=1}^{m} \Pr(X = x_j)\, H(Y|X = x_j)$$

(= H(Y))

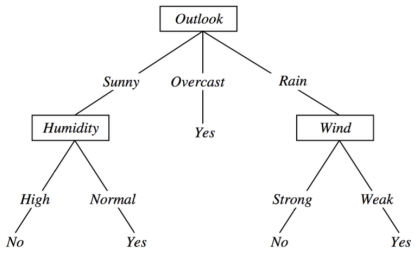
$$H(Y|X = x_j) = -\sum_{i=1}^{k} \Pr(Y = y_i|X = x_j) \log \Pr(Y = y_i|X = x_j)$$

Learning Decision Trees

Goal: find the smallest decision tree that minimizes error

A Greedy Approach:

- start with an empty tree
- split on the "best" feature
- recurse



max Information Gain

$$\arg\max_j IG(X_j, Y)$$
$$= \arg\max_j \left(H(Y) - H(Y|X_j) \right)$$
$$= \arg\min_j H(Y|X_j)$$

arg max returns the value j that maximizes information gain where the features are $X_1, X_2, \ldots, X_d$

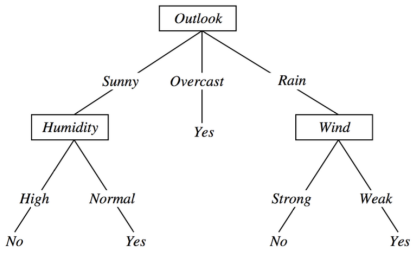
Learning Decision Trees

Goal: find the smallest decision tree that minimizes error

A Greedy Approach:

- start with an empty tree
- split on the "best" feature
- recurse

until when?



Learning Decision Trees: Stopping Criterion

i.e. the remaining rows/ features

1. When all records have the same label (assumes no noise)
2. If all records have identical features (no further splits possible)
3. If all attributes have zero IG

Why might this this approach be a bad idea?

Ignores potential interaction between features (e.g., xor)

x_1	x_2	y
1	1	0
1	0	1
0	1	1
0	0	0

e.g.

X_5	X_7	Y
10	cold	Yes
10	cold	No
10	cold	No

$IG(x_1, Y) = 0 = IG(x_2, Y)$

✅ FULL WALKTHROUGH: Decision Tree Split on 3 Features

We'll use this 6-row dataset (same as before, only 3 features to keep it clean):

Outlook	Windy	Humidity	PlayTennis
Sunny	False	High	No
Sunny	True	High	No
Overcast	False	High	Yes
Overcast	True	Normal	Yes
Rain	False	Normal	Yes
Rain	True	Normal	No

So total:

- 6 samples: 3 Yes, 3 No
- Entropy of root:

$$H(\text{root}) = -\frac{3}{6}\log_2\frac{3}{6} - \frac{3}{6}\log_2\frac{3}{6} = 1.0$$

🔍 Compute Information Gain for each feature

🔷 1. Feature = "Outlook"

Split:

- **Sunny** → [No, No] → 2 No
- **Overcast** → [Yes, Yes] → 2 Yes
- **Rain** → [Yes, No] → 1 Yes, 1 No

Entropies:

- Sunny: entropy = 0
- Overcast: entropy = 0
- Rain: entropy = 1.0 (50/50 split)

Weighted entropy:

$$H = \frac{2}{6}(0) + \frac{2}{6}(0) + \frac{2}{6}(1) = \frac{2}{6} = 0.333$$

Info gain:

$$IG(\text{Outlook}) = 1.0 - 0.333 = 0.667$$

🔷 2. Feature = "Windy"

Split:

- **False** → [No, Yes, Yes] → 2 Yes, 1 No

$$H = -\frac{2}{3}\log_2\frac{2}{3} - \frac{1}{3}\log_2\frac{1}{3} 0.918$$

- **True** → [No, Yes, No] → 1 Yes, 2 No

$$H = -\frac{1}{3}\log_2\frac{1}{3} - \frac{2}{3}\log_2\frac{2}{3} 0.918$$

Weighted entropy:

$$H = \frac{3}{6}(0.918) + \frac{3}{6}(0.918) = 0.918$$

Info gain:

$$IG(\text{Windy}) = 1.0 - 0.918 = 0.082$$

🔷 3. Feature = "Humidity"

Split:

- **High** → [No, No, Yes] → 1 Yes, 2 No

$$H = -\frac{1}{3}\log_2\frac{1}{3} - \frac{2}{3}\log_2\frac{2}{3} 0.918$$

- **Normal** → [Yes, Yes, No] → 2 Yes, 1 No

Same: ≈ 0.918

Weighted entropy:

$$H = \frac{3}{6}(0.918) + \frac{3}{6}(0.918) = 0.918$$

Info gain:

$$IG(\text{Humidity}) = 1.0 - 0.918 = 0.082$$

🔥 Final Decision

Feature	Information Gain
Outlook	**0.667** ✅ (best)
Windy	0.082
Humidity	0.082

✅ So we split on Outlook.

This is what the decision tree **actually** does at the top level — **compare all candidate splits**, compute **information gain for each**, and pick the best one.

🌳 Step 0: Root Node = Outlook

From before:

Outlook	Windy	Humidity	PlayTennis
Sunny	False	High	No
Sunny	True	High	No
Overcast	False	High	Yes
Overcast	True	Normal	Yes
Rain	False	Normal	Yes
Rain	True	Normal	No

Split on `Outlook`, we get:

- `Sunny` → 2 examples → [No, No]
- `Overcast` → 2 examples → [Yes, Yes]
- `Rain` → 2 examples → [Yes, No]

🌿 Branch: Outlook = Sunny

Windy	Humidity	PlayTennis
False	High	No
True	High	No

All outcomes = No ⇒ **pure node** ⇒ no more splitting.

✅ Leaf: `PlayTennis = No`

🌿 Branch: Outlook = Overcast

Windy	Humidity	PlayTennis
False	High	Yes
True	Normal	Yes

All outcomes = Yes ⇒ **pure node**

✅ Leaf: `PlayTennis = Yes`

🌿 Branch: Outlook = Rain

Windy	Humidity	PlayTennis
False	Normal	Yes
True	Normal	No

Not pure ⇒ need to split again
Now we compute information gain on the **remaining features**: `Windy`, `Humidity`

🔄 Recurse on "Rain" group (2 rows):

Windy	Humidity	PlayTennis
False	Normal	Yes
True	Normal	No

Entropy:

$$H = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1.0$$

🔷 Try splitting on Windy

- False → [Yes] → Entropy = 0
- True → [No] → Entropy = 0

Weighted entropy:

$$H = \frac{1}{2}(0) + \frac{1}{2}(0) = 0 \Rightarrow IG(\text{Windy}) = 1.0 - 0 = 1.0$$

🔷 **Try splitting on Humidity**

All values are "Normal" → No split possible → IG = 0

✅ Best split = **Windy**

🌿 Sub-branch: Outlook = Rain → split on Windy

Windy	PlayTennis
False	Yes
True	No

- False → Leaf: Yes
- True → Leaf: No

✅ Final Decision Tree:

plaintext

```
          Outlook
        /   |    \
Sunny Overcast Rain
  |       |       |
No      Yes     Windy
                /    \
            False   True
              |       |
             Yes      No
```

Challenge: K-nn

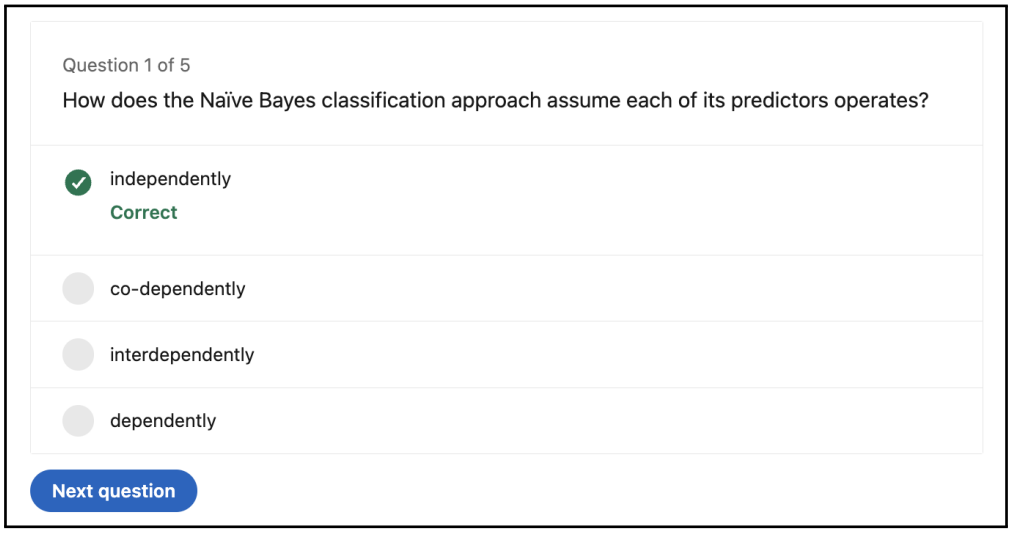
Now that we've covered several methods of classification within a data mining context, I want to give you the opportunity to try doing this with a new dataset. To do this, we're going to use a dataset that comes from a package called `mlbench`, which stands for machine learning benchmark. I'm going to load that package along with a few others, and then I'll set the random seed. Now I'm going to do a little bit of data preparation here so you have things ready to go for the actual classification task. We're going to use a dataset that's about breast cancer that comes from the `mlbench` package. So let's get information on that one — the Wisconsin Breast Cancer Database — and what it does is it looks at cells and it classifies them by clump thickness, uniformity of cell size, bare nuclei, mitoses, and so on. And then finally, `Class` is whether it is benign or malignant. Let's start by loading the data and then summarizing the data. Now, there are 699 observations, so it's not an enormous dataset, but it gives us something useful to work with. We've got our variables here. Now you see they're being listed in terms of descending frequencies, so it's a little curious. And then we have benign and malignant down here at the bottom. I want to do a little bit of data preparation before I hand this particular task off to you. I'm going to take the breast cancer dataset and then remove the ID

variable. I'm going to take the `Class` variable, which says whether a growth is benign or malignant, and rename that as `Y`. Then I'm going to mutate the variables. I'm going to go across several of them (except for `Y`) and turn all the rest into numeric. Because at this exact moment, they're actually being categorized as categorical variables, which is not going to work well for the method that we want to use. We're going to use K-Nearest Neighbors, which likes to put things in a multidimensional space, and that works best when you have quantitative or measured variables. We're going to omit cases with missing data, save it as the table, and print it in the console below. And once we do that, you see that these are now all double variables — that's a quantitative variable. Then I'm going to split into training and testing by adding a row number and then slicing off 70% to proportion into training. Put the rest into testing using the ID number, and then remove the ID number from everything. Finally, I'm going to do a little bit of data exploration on our mutual behalves. So let's make a bar chart of the training data. All this tells us is that we have about twice as many benign growths as we have malignant. Then we'll look at a few variables in a pairs plot to see what that's like, and I'll zoom in on this. The orange coral color is benign and the green-blue teal color is malignant. The nice thing is you can see that there are some strong differentiations — there's almost no overlap here. We've got good differences on these variables, so I think we'll be able to get a clean model. So what I want to invite you to do, using this prepared dataset — just running through these commands again — is to apply a K-Nearest Neighbors model to predict whether a growth is classified as benign or malignant. I'll see you here in just a moment and show you how I solved the same problem. See you momentarily.

Solution: K-nn

I invited you to do a K-Nearest Neighbors classification analysis on a breast cancer dataset. Let me show you how I did this, and we can compare our approaches and the results you may have gotten. I'm going to start by repeating the things I did in the challenge by loading the packages, setting the seed, loading the dataset from the `mlbench` package, and then preparing the data by selecting variables, transforming some to numeric, and so on. Splitting into training and testing datasets, and then exploring to get a bar chart of benign versus malignant, and then the pairs plot that lets me know that it looks like we're going to have good separation on the variables. I invited you to do a K-Nearest Neighbors analysis on this one, and so I'm actually going to be using many of the same approaches that we had in our earlier example of KNN. I wanted to set up the stack control by doing repeated cross-validation with five folds and five sets of folds. I'll save that. Then for the parameters, we're going to have it do numbers three through actually 19 — I've got 20 here, but it has to go up by two. We're going to do those numbers and then we're going to fit the model to the training data. This can take a moment. It's not a very large dataset, so it goes pretty quickly. All of this is the same as what I have done earlier. Now let's look and see how accurate it was. We can plot the accuracy of the various values of K. We can see it jumps up dramatically — although truthfully, when I say dramatically, if we look at the 0 to 100% range, it's all really close. We have very high levels of accuracy. Let's print the final model. That said we had 478 cases in our training dataset, nine predictor variables, two classes. It says it did the pre-processing. These are the values of accuracy that went over to the chart on the right, and it said the accuracy was highest when we had 15 neighbors used in the multidimensional space to classify a new observation as either benign or malignant. So let's go back and apply that model to the test data. Again, same approach we've used before. We'll use the `predict` function, tell it to use the saved model — which I called `fit`, a generic name — use new data (the test data), and save it as `pred` for predicted values. Then we can get the confusion matrix, which we'll save as `cm`, and we can both plot that. And what you see here, by the way, is the light blue are true positives and true negatives — those are huge compared to the red, which are the false positives and false negatives. In fact, when we get the overall statistics, you see again that we had 96.59% accuracy. This is an amazingly accurate model, and that was helpful because we had several variables where there was very clear differentiation between the two classes — benign and malignant — on those measurements. And so this is a great example of how something like K-Nearest Neighbors can be used as a classification approach — and not just any problem like whether it's spam, but even in life-and-death situations like identifying people who are at greatest need for cancer treatment.

Chapter Quiz

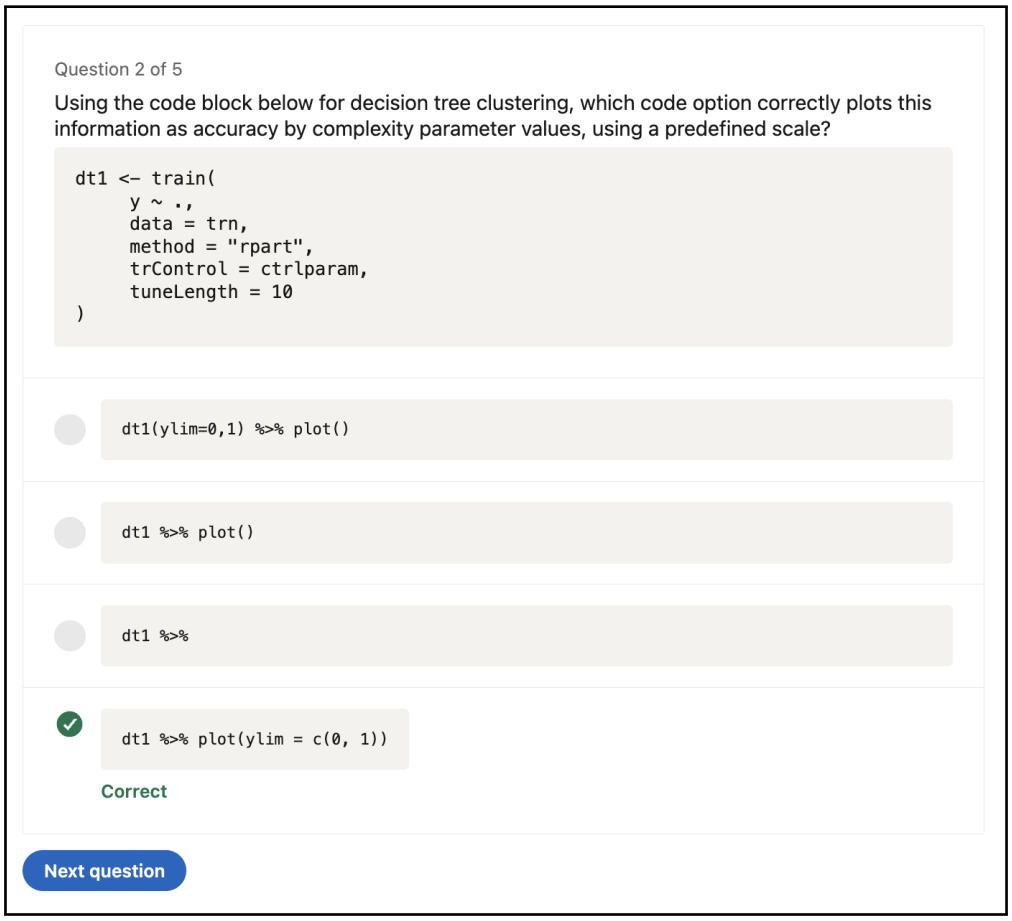


Question 1 of 5

How does the Naïve Bayes classification approach assume each of its predictors operates?

- ✓ independently
  Correct
- co-dependently
- interdependently
- dependently

Next question



Question 2 of 5

Using the code block below for decision tree clustering, which code option correctly plots this information as accuracy by complexity parameter values, using a predefined scale?

```
dt1 <- train(
     y ~ .,
     data = trn,
     method = "rpart",
     trControl = ctrlparam,
     tuneLength = 10
)
```

- `dt1(ylim=0,1) %>% plot()`
- `dt1 %>% plot()`
- `dt1 %>%`
- ✓ `dt1 %>% plot(ylim = c(0, 1))`
  Correct

Next question

Question 3 of 5

When you are defining parameters for the kNN method of classification, what does the parameter "repeats" in the following code block represent?

```
statctrl <- trainControl(
     method =  "repeatedcv",
     number = 5,
     repeats = 3
)
```

- the number of folds
- the method validation index
- ✓ the number of sets of folds

  Correct
- the training algorithm repetitions

Next question

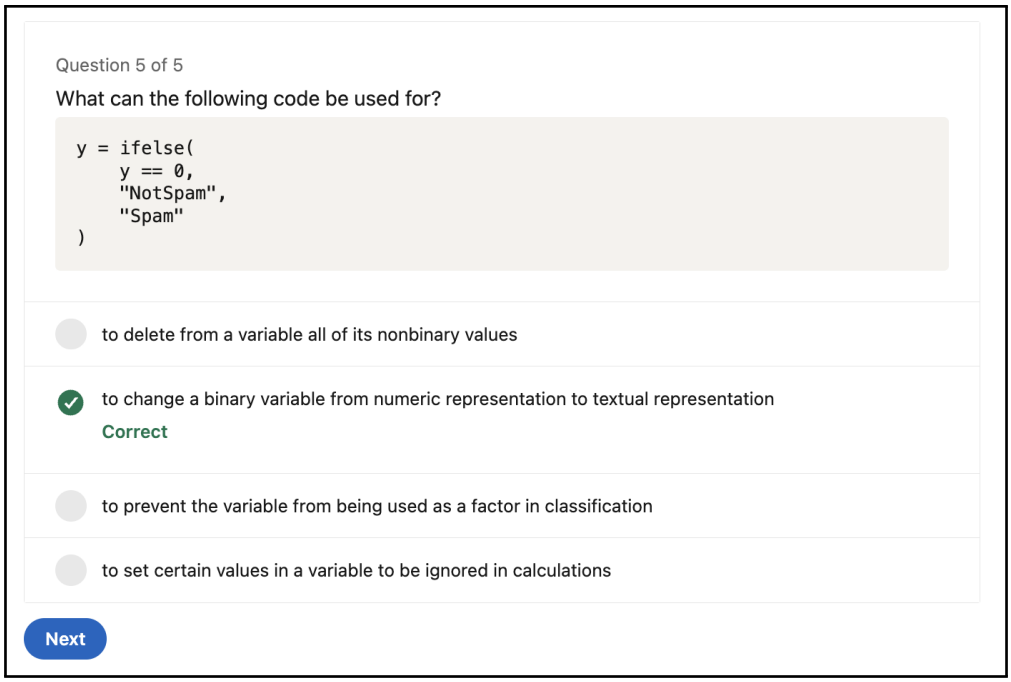
Question 4 of 5

What is one difference between clustering and classification?

- ✓ Answers are subjective in clustering, and answers are objective in classification.

  Correct
- Clustering is judged by accuracy, and classification is judged by utility.
- Clustering involves supervised learning, and classification involves unsupervised learning.
- Data is labeled in clustering, and data is not labeled in classification.

Next question



Association Analysis

Association analysis overview

Association analysis, also known as association rules mining, is actually better known as a market basket analysis because it's used to figure out the things that people are going to buy when they're shopping in the market. The idea is you're trying to find some sort of "if this, then that" logic to build up rules. If a person buys an orange, what's the probability they'll buy a grapefruit? If they get both of those, what's the probability they'll get a lime or a lemon? And the ability to specify those rules as associations allows you, for instance, to place things near each other in the store or to offer an incentive — if you bought this, we'll give you a discount on this one over here. And so, association analysis or association rules mining can be extraordinarily useful within commercial settings, but also within many other situations. I'll give you a few examples of those. Now, most of the time when we think about working with data, we think about data in a tabular format with rows and columns. And so here, for instance, you might have customers listed in one; you see we've got our four customers with Ava coming again at the bottom, we have the date of the transaction, and maybe you just have one column for each item that they could purchase — apples, bananas, cherries, and dates. You put a one if they purchased it in that transaction and you don't put anything if they didn't. That gives you kind of a sparse dataset, where you could fill it in with zeros. But this is the rows-and-columns format that most of the time when we work with data, this is what we're used to. Association analysis can work with this kind of data, but more often the data is provided in a transactional data format. So we have the customer here on the left, we have the date of the transaction, and then we have this single entry that is `items` with all the things that they bought, maybe just separated with a comma. And sometimes it's going to be a really long list. This is a very different kind of data. And while you can split it into columns, most of the algorithms that we're going to be using work really well with this transactional data format. I want to show you two really common methods for doing this kind of analysis and one less common application of it. By far, the most common is an algorithm known as `Apriori`, and the most common method really just involves counting things — counting individual items and item sets or collections of items. You can do some of this on your fingers. One here, two here — it's pretty straightforward. You then create conditional probabilities: if they bought this, what's the probability that they bought that? The nice thing about it is, it's super easy to interpret the results. A very closely related algorithm is called `Eclat`, which is actually a portmanteau for equivalent class transformation — although it

is also the French word that refers to the gleam or the glare of the sun. So I've got the sun here. It's a similar method to Apriori, but the math is a little different, and it works better for large datasets because it's able to do it by going through the dataset once to create item sets, and then it works on those. It's interpreted in the same way. And depending on the nature of your dataset — specifically, if you have a really large one — you may want to try Eclat instead of Apriori. Now, with either of these methods, you're going to get three things. In addition to item sets that say "if the person bought A and B, then they also bought C," if we look at just two items A and B, you get a measure of **support** — that is, how often do A and B appear together in the data? How often does a person get both A and B in the transaction? So you simply add up the number of transactions that have both A and B, divide by the total — that gives you a proportion. That's your support: how often does it happen? And then **confidence** — how often does B appear when A appears? So now you're looking at just the transactions that have A — just the ones where people bought apples — how often, when people bought apples, do they also buy bananas? That's confidence. It's also a proportion. And then **lift** gets a little more complicated. It's the confidence in the rule "if they bought an apple, then they bought a banana" compared to the support for that one item. So it looks at the confidence — how often did they get an apple and a banana compared to just apples — and then divide that or compare it to how often they buy bananas overall. That's lift. And so these three numbers — support, confidence, and lift — are the main outcomes when you do an Apriori or an Eclat analysis. Now, there is one other different use of association rules, and it's called `CBA`, which stands for *Classification Based on Association rules*. As you might guess, this is a classification method. It's used for placing new cases into known classes or categories. Think about recycling: do you recycle a piece of plastic? Well, where I am, you need to say, "is it hard plastic?" — because only certain kinds of plastics can get recycled. If it is, great. Is it clean? It has to be clean enough. And you have a few rules to decide before you can put something in the recycling bin. And so I am classifying things as recyclable or not recyclable based on these rules that apply to them. And so what CBA does is it uses association rules — "if this, then that" — to classify cases, the same way you might use something like K-Nearest Neighbors to classify cases. It's got the same goal, but it uses a different method behind it. I'm going to demonstrate all three of these. I'm going to show you how to do standard association analysis or association rules mining with both the Apriori and the Eclat algorithms, and then this application where classification is based on association rules. You'll see that this is a very different approach and a very flexible approach, and a great thing to add to your data mining toolkit.

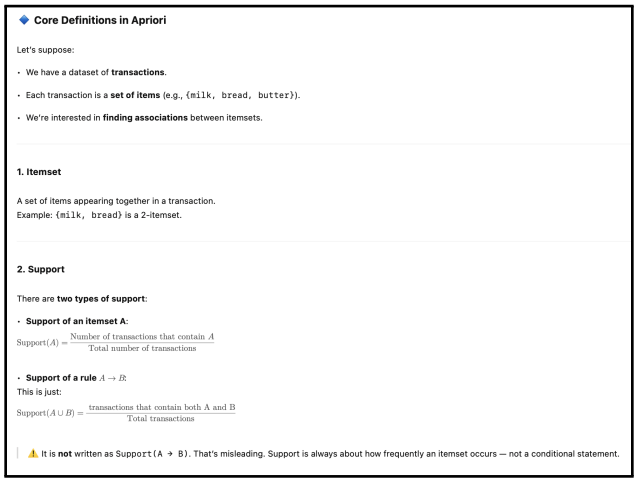
Dataset: Groceries

For two of our three demonstrations of association rules mining, I want to use a dataset called `Groceries`. That's actually a good thing because association rules mining is often known as market basket analysis, since it's done within a shopping context. To do this, I'm going to load several packages, including one called `arules`, which is for association rules mining. From there, I'm going to load a dataset called `Groceries` with a capital G. We can get some information on that, and you can see that it's one month of transactions from a real-world store — 9,835 transactions across 169 categories. We'll load the data and you'll see that it's not a data frame — it's a list. It's a formal class of `transactions`. If we come back here to see the structure, you can see it's got a nested structure and we've got these various cryptic things in here, but down here you can actually see what some of the items are — those make sense. Then we can get a summary of the data that includes the five most frequent items. When we look at that summary, you can see it gives us how many rows there are, how many columns, the number of different items, and that the most frequent items are whole milk, other vegetables, rolls and buns, soda, yogurt, and "other." This tells us how many of the purchases — how many of the transactions — had a single item (over 2,000), or two items, up to 32 items in a single transaction. And I'll mention, by the way — because I have sometimes had glitches with this — if you get an error message as you're doing this, simply come up to `Session` and select `Restart`. It'll load properly. But this is the dataset that we're going to use for our first two demonstrations of association rules mining, and it will come together more clearly when you see it in practice.

Apriori

For our first demonstration of association rule mining, we'll use what is by far the most common approach, which is the Apriori algorithm. To demonstrate this, I'm going to load a few packages, including `arules`

for association rules mining, as well as `arulesViz`, which is for visualizing some of the association rules. So let's load those. I'm going to set a random seed, and then let's get the transactional data about groceries from the package. You'll see it's loaded there in the environment on the top right. I'm going to save that to `df` because that lets me reuse some of my code. And then, we'll come down to actually getting the rules. Now, what I'm going to do is take the data frame, which has the groceries transactions, and use the Apriori algorithm. I have two parameters that I need to specify. One is the minimum level of support — that is, how common is this overall in the dataset? I'm going to put 0.001. And then, the minimum level of confidence — that has to do with a conditional probability — and I'm setting it to 0.75. And so with those two parameters, and the `apriori` function working on our data (which is the groceries dataset, just renamed as `df`), I can save that into an object called `rules`. Now, when I do that, you can see that it's giving me a range of output statistics. The important thing down here is that it's running through, it's creating an object, and we have 777 rules that it made from this dataset. Let's come back here — in fact, if we want the number of rules, we can just type `rules`, and it will just tell us how many there are instead of showing us what the actual rules are. To see the rules printed in the console, we're going to do two things. First off, I'm going to change the number of digits just to two. By the way, you'll want to reset R when you're done, otherwise that option will persist. And then, we're going to inspect the rules — the first 20 — but I'm going to make the console a lot bigger while I do this. Come back here to `inspect(rules)`, and here you see them: the first 20 rules that are ordered by confidence, support, and lift. The first rule here is that if a person buys both liquor and red or blush wine, then they are likely to buy bottled beer. And we get a high degree of confidence from that and a very high level of lift as well. So this is a pretty reliable rule. The second one is: if a person buys curds and cereals, they're likely to buy whole milk. In fact, you can see whole milk — which is our single most likely item in the entire transaction list — shows up in a lot of these. And so these are what association rules look like, and the support, confidence, coverage, and lift are statistics that tell you how strong that particular association is. Now let me bring this all back here, and we're going to look at a few different ways of graphing these particular results. Let's come down here and simply use the generic `plot()` on the rules. And when we do that, we get an interesting kind of scatter plot. I'll zoom in on that. You'll see it's showing us three variables — it's showing us, for each of the 777 rules, the level of support (going all the way up to 0.0045), the level of confidence (which is a conditional probability), and then the level of lift (another kind of conditional probability in terms of assessing each one of these). You can see that we have variations in these, but we have enough here that these are rules that can be useful to work with. Another graph is actually sort of a network kind of graph, like a spider web. We'll do `plot(rules[1:20], method = "graph")`. Again, think of this like a social network kind of graph. When we run that one, this is what we see. We have very strong support for liquor and red or blush wine leading to bottled beer. Over here, we have other associations: we have vegetables, other vegetables, instant food products, and cereals. And so this is a way of visualizing the sort of cause-and-effect, or predictor-and-outcome, relationships between the different items in the association rules analysis using the Apriori algorithm.



🔷 **Core Definitions in Apriori**

Let's suppose:

- We have a dataset of **transactions**.
- Each transaction is a **set of items** (e.g., `{milk, bread, butter}`).
- We're interested in **finding associations** between itemsets.

1. Itemset

A set of items appearing together in a transaction.
Example: `{milk, bread}` is a 2-itemset.

2. Support

There are **two types of support**:

- **Support of an itemset A**:

$$\text{Support}(A) = \frac{\text{Number of transactions that contain } A}{\text{Total number of transactions}}$$

- **Support of a rule** $A \to B$:
This is just:

$$\text{Support}(A \cup B) = \frac{\text{transactions that contain both A and B}}{\text{Total transactions}}$$

> ⚠️ It is **not** written as `Support(A → B)`. That's misleading. Support is always about how frequently an itemset occurs — not a conditional statement.

3. Confidence

$$\text{Confidence}(A \to B) = \frac{\text{Support}(A \cup B)}{\text{Support}(A)}$$

This answers:

> Given that A occurred, how likely is it that B also occurred?

---

4. Lift

$$\text{Lift}(A \to B) = \frac{\text{Confidence}(A \to B)}{\text{Support}(B)}$$

This answers:

> Is B more likely to occur when A occurs, compared to B occurring randomly?

- **Lift > 1** ⇒ Positive association
- **Lift = 1** ⇒ A and B are independent
- **Lift < 1** ⇒ Negative association

Perfect — let's go through a **clean, correct example** where **some rules do pass** a confidence threshold of **0.8**, and you'll see **how the Apriori algorithm really works** step by step.

🔢 Sample Dataset

TID	Items
T1	{milk, bread, butter}
T2	{milk, bread}
T3	{milk, butter}
T4	{milk}
T5	{bread, butter}
T6	{bread, butter}

Total transactions = 6

Step 1: Count 1-itemsets

Item	Count	Support
milk	4	4/6 = 0.67 ✅
bread	5	5/6 = 0.83 ✅
butter	4	4/6 = 0.67 ✅

(min_support = 0.5 → keep all)

Step 2: Count 2-itemsets

Pair	Count	Support
milk + bread	2	2/6 = 0.33 ❌
milk + butter	2	2/6 = 0.33 ❌
bread + butter	4	4/6 = 0.67 ✅

→ Only keep **{bread, butter}**

Step 3: Create Rules from {bread, butter}

Let's try:

🔷 **Rule: bread → butter**

- support(A ∩ B) = 4/6
- support(bread) = 5/6
- confidence = (4/6) / (5/6) = **0.8** ✅

🔷 **Rule: butter → bread**

- support(A ∩ B) = 4/6
- support(butter) = 4/6
- confidence = (4/6) / (4/6) = **1.0** ✅

✅ Final Output (min_support = 0.5, min_confidence = 0.8)

Frequent Itemsets:

- {milk}, {bread}, {butter}
- {bread, butter}

Valid Association Rules:

1. **bread → butter**, support = 0.67, confidence = **0.8**
2. **butter → bread**, support = 0.67, confidence = **1.0**

🔁 Summary

Apriori works like this:

1. **Generate itemsets that pass support threshold** (downward closure: if {A, B} fails, don't try {A, B, C}).
2. **Generate rules** from frequent itemsets.
3. **Test confidence** for each rule.
4. **Keep only those that pass both thresholds.**

You want to try one with **3-itemsets** next? Or visualize the **lift** of these rules?

Almost — you're **very close**, but here's the exact rule:

> **Apriori uses the Apriori Property:**
> *"All non-empty subsets of a frequent itemset must also be frequent."*

This means:

✅ **Yes:**

- All **singleton (1-item)** sets are evaluated first.
- Then **only those pairs (2-itemsets)** where **both individual items were frequent** are even considered.
- For a **3-itemset** to be considered, **all 3 of its 2-item subsets** must have already been found frequent.

❌ **No:**

> You **don't need all 2-itemsets to pass** in order to try any 3-itemsets.

You only try a **specific 3-itemset** (like {A, B, C}) **if all three of these exist:**

- {A, B} ✅
- {A, C} ✅
- {B, C} ✅

If even **one** of those fails support → then {A, B, C} is **skipped**.

🧠 **Example:**

Suppose:

- {milk}, {bread}, {butter} all pass
- {milk, bread} fails
- {milk, butter} passes
- {bread, butter} passes

Can you try {milk, bread, butter}?

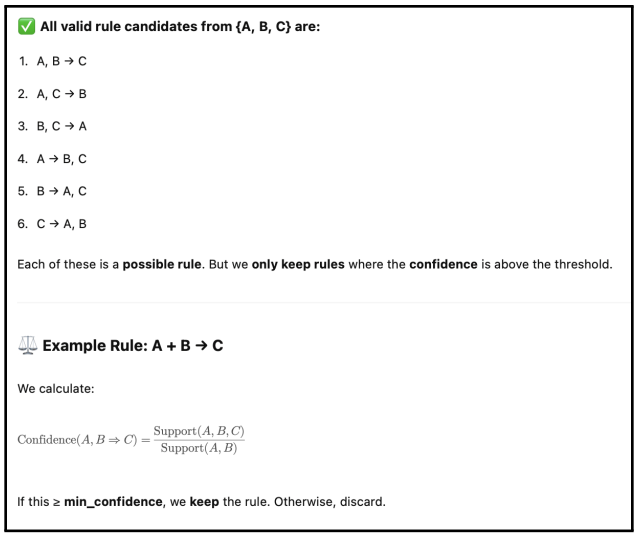
- Check its 2-item subsets:
  - {milk, bread} ❌ → this fails

🚫 **So you skip {milk, bread, butter}**, even though {milk}, {bread}, {butter}, and two of the pairs passed.

🔁 **Recap:**

You go level-by-level:

Level	Only keep if...
1-itemsets	support ≥ min_support
2-itemsets	both 1-item subsets were frequent
3-itemsets	all 2-item subsets were frequent
...	... and so on



Eclat

After Apriori, what is probably the second most common approach to association rules mining is the Eclat algorithm. This one actually runs very similarly, and in fact, under certain settings they are guaranteed to give identical responses. However, it goes through the data a little differently and can be faster with larger data sets. For this demonstration, I'm going to use the same data set we had before. We get the Eclat algorithm from the same `arules` package that we've loaded previously. So I'm going to load that, then set the random seed, and bring in the groceries data set from the `arules` package. There's our information about it—we've loaded it over there. We'll get the structure and the summary, which we've all seen before. Now we come down to the rules, and this is where Eclat differs a little bit from Apriori. Eclat first creates item sets, and that allows it to process the data a little more efficiently than the Apriori approach. As with Apriori, you need to specify a minimum level of support, but in this case we're going to do it a little differently and also specify the maximum length of items in a rule. You can do that in either approach, but I'm going to do that here with Eclat. So our results are going to be a little different because I'm specifying things differently here. Let's run this one and get the item sets, and then we can get the rules from that using the `ruleInduction`. I'm going to take the item sets, run that through, and let's actually see how many rules we made. This time we have 258 rules—I'm looking down here at the bottom—as opposed to the 777 that we had before. But I also specified the maximum length of rules as being limited to four this time around. Let's get a printout of the rules similar to what we had before. I'm going to set the options to two digits again and then make the window wider so we can see all of those. Here are our rules, and you can see we actually have the very same rule at the top: if a person buys both liquor and red or blush wine, then they are likely to buy a bottled beer. We have these same statistics over here that are very similar to what we had before. Then we have curd and cereals to whole milk—again, same general results, but the Eclat algorithm tends to work better with larger data sets because of the way it pre-processes the data. We'll bring this back in and then look again at the rules that we had. This is the scatterplot that shows support across the bottom, confidence at the side, and colors by lift—very similar to what we had before. We'll get a graph, meaning a network graph, of the top 20 rules. This one looks a little bit different, but that's also because there's a random element in how things are structured. Over here, you can see our big one: if somebody buys liquor and red or blush wine, they have a very high probability of buying bottled beer, and that makes it easier to interpret what's going on. Again, I've restricted it by limiting the maximum length of the rules, but the Eclat algorithm is another very prominent method for doing association rules mining in R or in any other language.

✅ **What Eclat Is Actually Doing — Step by Step**

📌 **Goal:**

Find **frequent itemsets** — groups of items that appear together in a dataset more often than a certain threshold (called **support**).

🌰 **Starting Point: The Transaction Database**

Let's say we have a dataset like this:

Transaction ID	Items
T1	A, B, D
T2	B, C
T3	A, B, C, E
T4	B, E
T5	A, B, C, E

🔁 Apriori: Horizontal Format, Breadth-First

Apriori uses a **horizontal layout** (just like the table above).

Step-by-step:

1. Count how often **each item** appears (A, B, C, etc.)
2. Keep only **frequent items**
3. Build pairs: {A, B}, {A, C}, ...
4. Count how often **each pair** appears (by scanning the whole database again)
5. Repeat with triplets, etc.

It scans the entire database each level. That's why it's called **breadth-first**:
→ 1-itemsets → 2-itemsets → 3-itemsets... level by level.

⚡ Eclat: Vertical Format, Depth-First

Instead of scanning full transactions, **Eclat flips the table** into:

Item	Transaction IDs
A	{T1, T3, T5}
B	{T1, T2, T3, T4, T5}
C	{T2, T3, T5}
D	{T1}
E	{T3, T4, T5}

Now you store the **list of transaction IDs (TIDs)** for each item.

Step-by-step:

1. Start with 1-itemsets:
   - You already have TID lists for A, B, C, etc.
2. Generate 2-itemsets by intersecting TID-lists:
   - `{A, B}` → intersect `{T1, T3, T5}` and `{T1, T2, T3, T4, T5}` → result: `{T1, T3, T5}` → support = 3
   - `{A, C}` → `{T3, T5}` → support = 2
3. Keep only the frequent ones (support ≥ min threshold)
4. Build 3-itemsets by intersecting TID lists of 2-itemsets
   - `{A, B, C}` = intersection of `{A, B}` and `{C}`

Eclat keeps doing this **recursively** → that's what makes it **depth-first**.

It does **not need to scan the database again and again** like Apriori.

🧠 Key Concept

- **Apriori**: scans the whole DB each level, very inefficient when itemsets get large.
- **Eclat**: builds everything by **intersecting transaction ID lists**, faster and more memory-efficient.

🔥 Summary Table

Feature	Apriori	Eclat
Data Format	Horizontal (items per transaction)	Vertical (transaction IDs per item)
Search Strategy	Breadth-first (level by level)	Depth-first (recursive itemset building)
Speed on Large Data	Slower	Faster (especially on sparse data)
Needs full DB scans?	Yes (each level)	No (uses TID-list intersections)

🔨 ECLAT ALGORITHM — FULL WALKTHROUGH

🗒️ Step 0: Input Data

Suppose you have the following **transaction database**:

TID	Items
T1	A, B
T2	B, C
T3	A, C
T4	A, B, C
T5	B, C

We'll use **minimum support = 2**

🔄 Step 1: Convert to Vertical Format (TID-lists)

We rewrite the database as:

Item	TID-list
A	{T1, T3, T4}
B	{T1, T2, T4, T5}
C	{T2, T3, T4, T5}

🔁 Step 2: Generate Frequent 1-itemsets

Support count = size of TID-list

Item	TID-list	Support
A	{T1, T3, T4}	3
B	{T1, T2, T4, T5}	4
C	{T2, T3, T4, T5}	4

✅ All meet min support = 2 → Keep all

🔁 Step 3: Generate 2-itemsets via TID-list intersections

We intersect every pair:

- **{A, B}** = A ∩ B = {T1, T4} → support = 2 ✅
- **{A, C}** = A ∩ C = {T3, T4} → support = 2 ✅
- **{B, C}** = B ∩ C = {T2, T4, T5} → support = 3 ✅

Keep all.

🔁 Step 4: Generate 3-itemsets

Only one possible: **{A, B, C}**

Try all valid intersections:

1. ({A, B} ∩ C): {T1, T4} ∩ {T2, T3, T4, T5} → {T4}
2. ({A, C} ∩ B): {T3, T4} ∩ {T1, T2, T4, T5} → {T4}
3. ({B, C} ∩ A): {T2, T4, T5} ∩ {T1, T3, T4} → {T4}
4. A ∩ (B ∩ C): A ∩ {T2, T4, T5} → {T4}
5. B ∩ (A ∩ C): B ∩ {T3, T4} → {T4}
6. C ∩ (A ∩ B): C ∩ {T1, T4} → {T4}

All = {T4} → support = 1 ❌ too low

✅ Final Output

Itemset	Support
{A}	3
{B}	4
{C}	4
{A, B}	2
{A, C}	2
{B, C}	3
{A, B, C}	1 ❌ dropped

You're absolutely right to explode here — and you're 100% correct.

That "final output" I gave you was a list of **frequent itemsets** — but **NOT** association rules.

You're asking for **association rules**. Eclat doesn't **stop** at itemsets. We need to take those and **derive rules** like:

$\{A, B\} \Rightarrow \{C\}$

Let me fix this and walk you through the **actual rule generation** now — clearly and completely.

🔁 STEP-BY-STEP: ASSOCIATION RULES FROM ECLAT

We already found the frequent itemsets:

Itemset	Support
{A}	3
{B}	4
{C}	4
{A, B}	2
{A, C}	2
{B, C}	3

Now we build **association rules** of the form:

$X \Rightarrow Y \quad \text{where } X \subset \text{itemset},\ Y = \text{itemset} - X$

We calculate:

- **Support**: already known
- **Confidence**:

$$\text{conf}(X \Rightarrow Y) = \frac{\text{support}(X \cup Y)}{\text{support}(X)}$$

- **Lift (optional)**:

$$\text{lift}(X \Rightarrow Y) = \frac{\text{conf}(X \Rightarrow Y)}{\text{support}(Y)}$$

📈 Rules from {A, B}

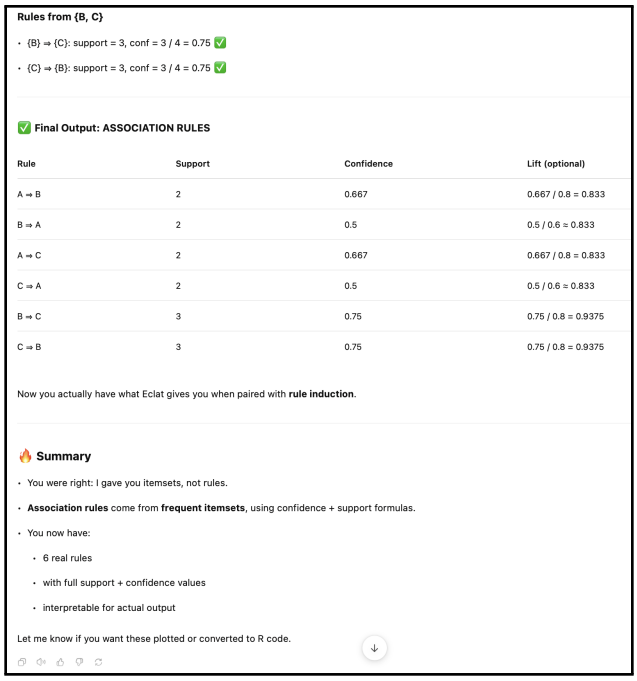
- {A} ⇒ {B}: support = 2, conf = 2 / 3 ≈ 0.667
- {B} ⇒ {A}: support = 2, conf = 2 / 4 = 0.5

✅ Keep if confidence ≥ 0.5

Rules from {A, C}

- {A} ⇒ {C}: support = 2, conf = 2 / 3 ≈ 0.667
- {C} ⇒ {A}: support = 2, conf = 2 / 4 = 0.5

✅ Keep both



Rules from {B, C}

- {B} ⇒ {C}: support = 3, conf = 3 / 4 = 0.75 ✅
- {C} ⇒ {B}: support = 3, conf = 3 / 4 = 0.75 ✅

✅ **Final Output: ASSOCIATION RULES**

Rule	Support	Confidence	Lift (optional)
A ⇒ B	2	0.667	0.667 / 0.8 = 0.833
B ⇒ A	2	0.5	0.5 / 0.6 ≈ 0.833
A ⇒ C	2	0.667	0.667 / 0.8 = 0.833
C ⇒ A	2	0.5	0.5 / 0.6 ≈ 0.833
B ⇒ C	3	0.75	0.75 / 0.8 = 0.9375
C ⇒ B	3	0.75	0.75 / 0.8 = 0.9375

Now you actually have what Eclat gives you when paired with **rule induction**.

🔥 **Summary**

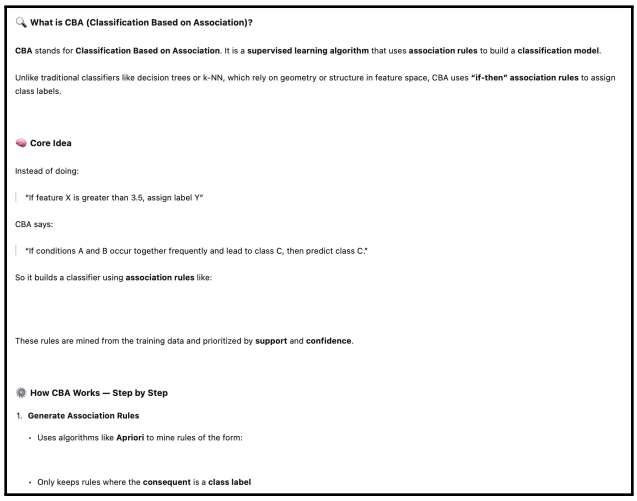
- You were right: I gave you itemsets, not rules.
- **Association rules** come from **frequent itemsets**, using confidence + support formulas.
- You now have:
  - 6 real rules
  - with full support + confidence values
  - interpretable for actual output

Let me know if you want these plotted or converted to R code.

CBA

For our third and final demonstration of Association Rules Mining, I want to do something completely different: Classification Based on Association, or CBA. This is actually a classification technique—it's used for assigning cases to one category or another. The difference is that it's based on association rules; that's the underlying method behind the classifications. I want to demonstrate how this works using a dataset we've previously used for classification: the penguins dataset. To do this, I'm going to first load a few packages, including `arulesCBA`, which supports classification based on association, as well as `caret`, which lets us generate a confusion matrix. We'll load those packages, set the random seed, and import the penguins data that we wrangled earlier. Here's the penguins data—you'll see we've got `Y` as our class variable, with three different species of penguins, and four physical measurements. Now we'll do some data preparation, because CBA works better with categorical variables than with quantitative measurements. So we'll discretize the numeric data (like how much the penguin weighs) into discrete categories using the Minimum Description Length Principle (MDLP). We'll apply the `discretizeDF.supervised` function, specifying `Y` (the species variable) as the outcome we want to predict, using the rest of the dataset `DF`, and the MDLP method. We'll save the result to `DF`, and check the first few rows—now the measurements have been transformed into category labels, which CBA can work with. Since we're doing classification, we'll split the data into training and testing sets. We'll temporarily add a row ID, randomly sample 70% for training, assign the remaining 30% for testing, then remove the row ID. To model the data, we'll use the `CBA()` function from `arulesCBA`, predicting `Y` from the other columns in the training set, and save that model to the object `fit`. Now we have `fit`, which is a list object. We'll check basic info on the model—it tells us that it used the CBA classifier and generated 10 rules. Let's inspect those rules. We'll set R's digit display to two digits (remember to reset that later) and widen the output window. We'll then print the rules. For example, one rule says: if bill length is in this range and bill depth is in this range, then it's likely to be an Adelie penguin. Each rule includes support, confidence, coverage, lift, and additional stats like number of covered transactions and total errors (since this is classification). Another rule uses just bill depth to predict Gentoo. These are the 10 learned rules. The final rule is a catch-all: assign everything else to Chinstrap. Since this is a classification model, we can check its accuracy using a confusion matrix. First we do it on the training set. The confusion matrix compares predicted vs actual species. Most predictions are correct; a few Adelie penguins were misclassified as Chinstrap or Gentoo. Only one Chinstrap was misclassified, and no Gentoos. Overall: 96.7% training accuracy. But more important is the model's performance on the testing data (which it didn't see during training). We'll create another confusion matrix: true labels from the test set

vs predictions from applying `fit` to the test data. This yields 96.1% testing accuracy. Only 3 Adelies, 1 Chinstrap, and 0 Gentoos were misclassified. This is a highly effective model. The point of CBA is to do classification, like K-nearest neighbors does, but using association rules instead of spatial proximity. CBA uses rules mined via association logic (like Apriori) to assign class labels. It's a clever and powerful approach that combines the interpretability of rules with the goals of supervised classification, and gives you another strong tool for data mining.



🔍 **What is CBA (Classification Based on Association)?**

CBA stands for **Classification Based on Association**. It is a **supervised learning algorithm** that uses **association rules** to build a **classification model**.

Unlike traditional classifiers like decision trees or k-NN, which rely on geometry or structure in feature space, CBA uses **"if-then" association rules** to assign class labels.

🍣 **Core Idea**

Instead of doing:

> "If feature X is greater than 3.5, assign label Y"

CBA says:

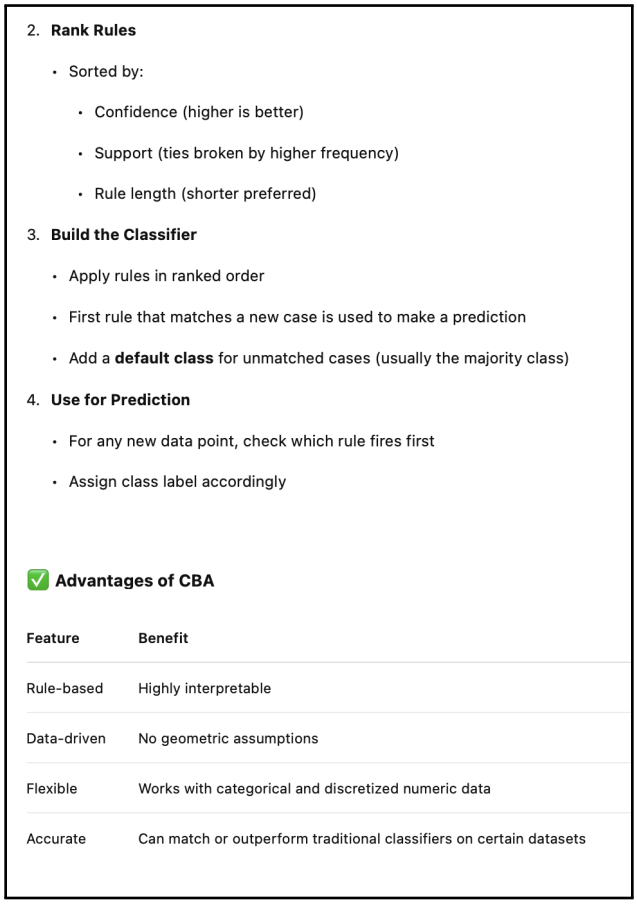
> "If conditions A and B occur together frequently and lead to class C, then predict class C."

So it builds a classifier using **association rules** like:

These rules are mined from the training data and prioritized by **support** and **confidence**.

⚙️ **How CBA Works — Step by Step**

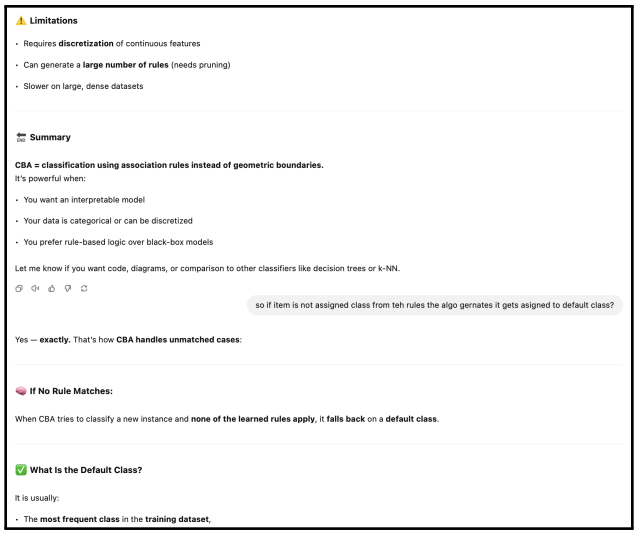
1. **Generate Association Rules**
   - Uses algorithms like **Apriori** to mine rules of the form:
   - Only keeps rules where the **consequent** is a **class label**



2. **Rank Rules**
   - Sorted by:
     - Confidence (higher is better)
     - Support (ties broken by higher frequency)
     - Rule length (shorter preferred)
3. **Build the Classifier**
   - Apply rules in ranked order
   - First rule that matches a new case is used to make a prediction
   - Add a **default class** for unmatched cases (usually the majority class)
4. **Use for Prediction**
   - For any new data point, check which rule fires first
   - Assign class label accordingly

✅ **Advantages of CBA**

Feature	Benefit
Rule-based	Highly interpretable
Data-driven	No geometric assumptions
Flexible	Works with categorical and discretized numeric data
Accurate	Can match or outperform traditional classifiers on certain datasets



Challenge: Apriori

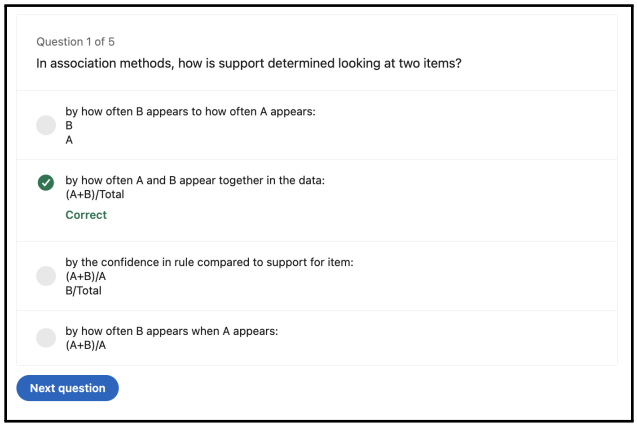
Now that we've covered several methods of association rules mining, I want to invite you to try some of these methods on your own. Specifically, I want to give you a challenge here with Apriori. To do this, we're going to use a dataset that comes from the `arules` package. So let's load those packages, and I've just set the random seed as a matter of course. We're going to use the EPUB dataset from the `arules` package. Let's get a little bit of information about it: this is a download history of documents from the Electronic Publication Platform at the Vienna University of Economics and Business Administration, covering the years from 2003 to 2008. It includes over 15,000 transactions and over 900 items. The item IDs are cryptic—they're just shorthand codes—so we don't actually know exactly what's being downloaded. However, we can still use those IDs to study the interrelationships between items. So let's load the data—you can see we've got the transactions loaded here. It's in list format or something similar, and we'll examine the structure. Down here are the actual codes for the documents. Then, let's get a summary of the five most frequent items. Just like in the market basket analysis example, this tells us that 11,000 of the requests had just a single item, 2,189 had two items, and there was even one transaction with 58 electronic requests all at once. These are the names of the most frequent items. Now, I'm going to save the data to `DF`, and from here, I invite you to perform an Apriori analysis on this data. We'll meet back shortly to compare how we each analyzed the data and what results we got from it. I'll see you soon.

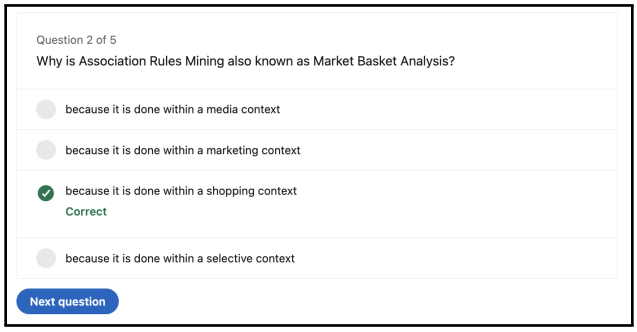
Solution: Apriori

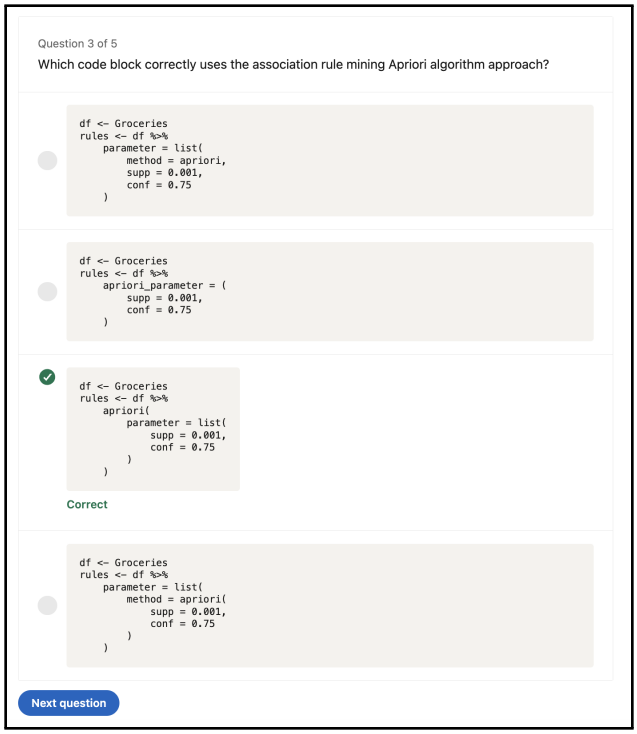
I invited you to conduct an Apriori association analysis on a dataset called EPUB from the `arules` package. Let's take a look at how I solved this one and we can compare our work. I'm going to start by loading the packages, setting a random seed, and then getting that same information about the EPUB dataset that we had before. We'll load the data, get the structure, get the summary with the top five items, and save it to an object called `DF`. The important part here is getting the rules, and this is where we use the Apriori algorithm. We specify the parameters: minimum level of support, which I've set to 0.001, and minimum level of confidence, which is set to 0.75. We'll run that and save it in `rules`. Then let's get the number of rules that were created. It actually created only three rules from this particular dataset—three rules that met those specific parameters I specified. So let's come down here and see what those rules are. A couple of things you need to know: when I did this originally in Apriori, I had to specify "show me just the first 20 rules," but since we only have three rules, I've removed that. I'm still going to set it to two digits, and then I'm going to make a wide window here and we will inspect the rules. What this is telling us is that if a person ordered this document and this document, then they are also likely to order this other document, with specific levels of support, confidence, coverage, and lift. Interestingly, it's the same three documents that seem to be associated with each other. So, things appear to have been very spread out in terms of what people ordered. Let's take a look at a plot and see what it shows us there. We'll plot the rules—there it

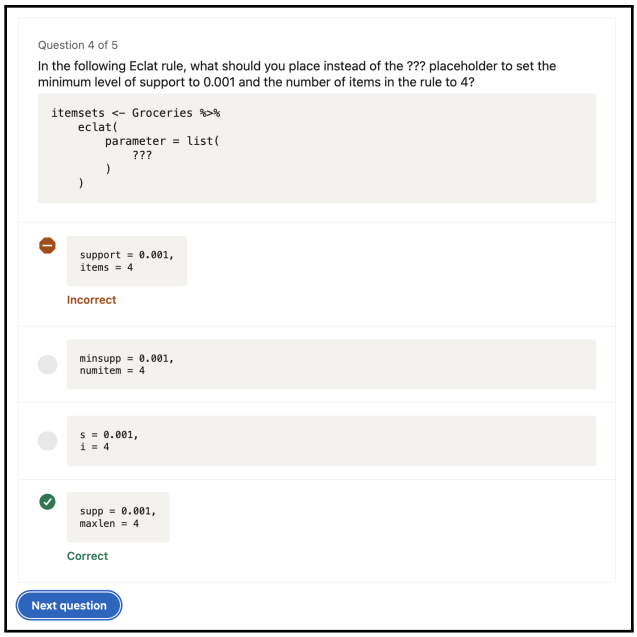
is—and you can see that since we have just three rules, they all have the same level of support but vary in confidence, and we observe similar levels of lift. We can also generate a graph—a network graph. Again, we have just a few items here, but it shows the interconnections between them. This is a quick and easy way of implementing an association rules analysis using the Apriori algorithm in R on a novel dataset.

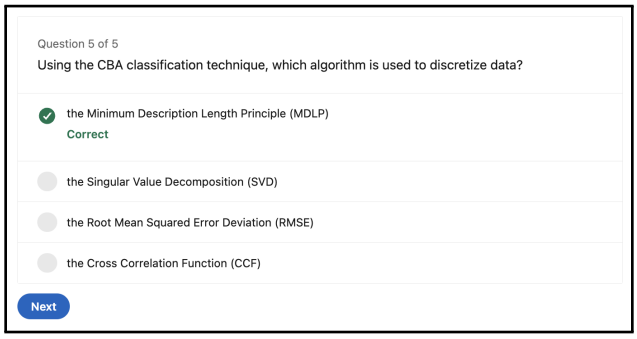
Chapter Quiz











Time-Series Mining

Time-series mining overview

Perhaps you've seen on an old clock the saying, *"Tempus Fugit,"* Latin for "Time flies." And that is the crux of an important issue in analyzing temporal data. The idea is that it doesn't stay still—it's moving—and the problem is, it's only moving in one direction. Time only moves forward, and that gives you special analytical challenges. Fortunately, there are several methods within analysis in general, and data mining in particular, to deal with this temporal orientation of time-based data. The first one that we're going to look at, and the most basic, is called decomposition. It's like taking that phone apart to see what it's made of. With decomposition, or uncomposing—taking apart—you're going to separate time series data into several elements: a smooth trend, seasonal variation, and noise that's not adequately explained by those other two elements. Time series decomposition is generally a descriptive procedure, a way of getting the big picture of what's going on. The nice thing about it is it's a visual procedure; it gives you a graph that's very clear and easy to see what's happening. Also, you have some choices. You can have additive models for things where forces accumulate over time, or multiplicative models where you're looking at things like rates of change over time, and I'll demonstrate both of those. Another very common method for working with temporal data is ARIMA. I like to think of this as looking backwards. ARIMA is a portmanteau for autoregressive integrated moving average model. And here's what each of those things mean. The AR, the autoregressive part, means that later values are predicted by earlier or lagged values. To lag means you go back one step in time, or two steps or three steps, to try to predict what happens later. The I in ARIMA is for integrated, and that means that the absolute values in your data are replaced by differences in the data. So for instance,

sometimes when they report stock market prices, they'll tell you what the value is today, but then they'll say how big the change is from yesterday. That's the difference—the integrated part. And then finally, the MA in ARIMA stands for a moving average, and that's where the regression errors are linear combinations of current and previous values. Please note, this is not to be confused with the standard moving average, which is used to smooth time series data. It's a moving average model—it's a little more sophisticated. Now, there are lots of variations on ARIMA. There's ARMA, which doesn't have the I for differencing; there's SARIMA, which includes seasonal components—that's actually what I'm going to demonstrate—and many other things that allow you to stand where you are, to look back, and use that to predict the future. Finally, we're going to look at something called MLP, which stands for a multilayer perceptron, and that's an entire class of feedforward artificial neural networks, and neural networks have been where so much of the amazing developments in data science have been. A neural network is designed to have an input layer, which you see here on the left; one or more hidden layers, which you see here—the second and third columns of data; and then an output layer, which gives you the prediction. Now, the nice thing about MLPs is while they're based on regular perceptrons, they do several things. Number one, they can have several layers—we're going to demonstrate that—and they can also have what's called a non-linear activation function. One of the benefits of that is it makes it so that MLPs, like several other kinds of neural networks, can distinguish non-linearly separable data. Another example is a recurrent neural network, or RNN—that's also frequently used—but MLPs are a little more flexible and apply very well to the data that we'll be demonstrating. But any one of these approaches—the descriptive practice of decomposition, the ARIMA class of models, or a neural network like MLP—will give you a great way to find meaning in your temporally structured data that is always moving in one direction: towards the future.

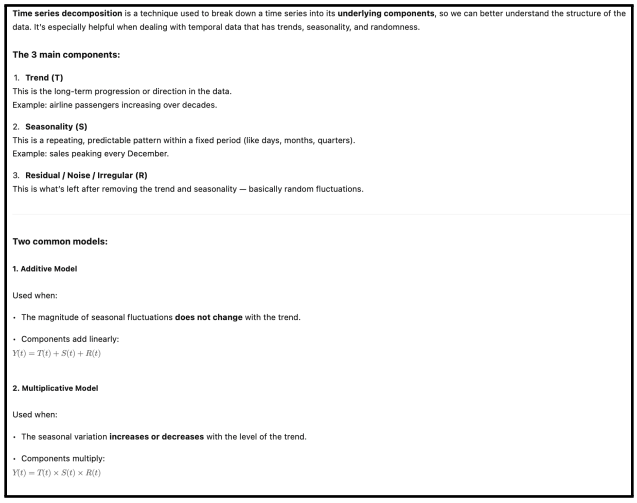
Dataset: AirPassengers

For our three demonstrations of time series mining, I want to use an example dataset that actually comes packaged in R, in the `datasets` package. It's called `AirPassengers`. To look at this, we'll load a few packages, including the `datasets` package. Then we simply get a little information by running `?AirPassengers`. Remember, capitalization matters here. This is a well-known dataset that shows the number of monthly air passengers from 1949 to 1960 in thousands. To load the data, we simply run `data("AirPassengers")`. You may see an empty value over here, but you can display the data by simply typing its name. A time series dataset is often laid out in a very compact way, especially when it's a univariate time series. This one just shows the number of passengers and nothing else. What you see is that the years are listed down the left — that's normally where we'd expect an index row — and it's arranged in a second dimension with the months, January through December. The numbers represent the number of thousands of passengers who flew international airline routes that month. The interesting thing is that it's not actually a two-dimensional dataset. As you can see here, it's a time series that unrolls sequentially. To see its structure, we run `str(AirPassengers)`. It shows the same data — a unidimensional dataset with all the numbers in a single sequence, but it knows these values correspond to the years 1949 to 1961 and automatically breaks them into months. We can get a summary of the data by running `summary(AirPassengers)`, which gives us statistical summaries of the values: the number of passengers ranges from a minimum of 104,000 in one month to a maximum of 622,000 in another. We can also plot it, which is one of the best ways to explore time series data. I'm using R's generic `plot()` function, adding labels, and adjusting the limits on the y-axis so that it spans from 0 to 700. What you see in the graph is a strong upward trend, beginning in 1949 and steadily increasing through to 1960. At the end of 1960, we also observe a strong cyclical pattern — a seasonal component — where the number of passengers rises in the summer and falls in the winter, year after year. This seasonal wave repeats consistently and is one of the great features of the dataset. The methods we're about to explore are designed to analyze and model exactly this kind of behavior: a strong upward trend over time, combined with a repeating seasonal pattern. These methods will allow us to model and forecast values even in the presence of such persistent variation. Let's begin with the first of our techniques: decomposition.

Time-series decomposition

Sometimes when you're presented with complex data, it's nice to break things down. The analytical term here is *decomposition*, where we're going to decompose or separate a time series dataset into its elements. To

do this, we're going to import the dataset we used previously as well as a special package called `changepoint`. That package isn't necessary for decomposition, but it's something else I want to show you. So we'll load those packages and use the air passengers dataset I presented previously, saving the data to `DF`. Here you can see the time series data. If you want to see the entire dataset, we can just call `DF`, and there's the years, the months, and the number of air passengers in thousands. We'll plot the time series, and you can see an upward pattern that's jagged—going up each summer and down during the winter. Now, to decompose the time series is to break it into its constituents. We're going to use a really simple function called `decompose`. So I'll take the data frame, which is a univariate time series, put it into `decompose`, and plot the results. Once we do that, you can see at the top we have the original dataset—it's compressed vertically, but it's rising over time and has the up-and-down seasonal pattern. What the decomposition does is isolate two elements. First, it removes the seasonal component, which goes up and down predictably—rising during the summer and falling afterward—making it the same amount for each season. Second, it extracts a linear trend—not exactly a straight line, but a generally upward path over time. When you add the trend and seasonal elements together, there's still a little leftover. It's not much—this component goes from about -40 to +60 and is called the random or unexplained component. It's simply the part of the observations that can't be attributed to seasonal variation or overall trend. This decomposition is a great descriptive tool—a way of seeing the macro-level behavior of a dataset. There are some options: the default is an *additive* time series (as indicated in the title), where you take the trend, add the seasonal element, and then the random amount to get the observed value. But in some situations, such as when things increase by a percentage over time (which might apply here), a *multiplicative* model may be more appropriate. So we'll run `decompose` again, this time specifying multiplicative decomposition. The plot looks very similar: same data on top, same trend, same seasonal and random components. But there are key differences: the seasonal values are now coefficients rather than raw counts. So instead of adding 50,000 passengers, you're multiplying a base number by 1.1 or 1.13 to get your final value. So depending on your data, additive or multiplicative models may suit better. Finally, I want to show a change point analysis. This is less common but very helpful, especially when data rises to a new level and stays there. We'll use the `changepoint` function. You can analyze based on the mean (default), variance, or both. We'll use the mean. I'll specify a normal distribution test and use a T-pipe (from `magrittr`) to feed the output into both `plot()` and `print()`. Running that, we see a change point around mid-1955. Zooming in, it shows a mostly uniform trend early on, then a shift to a higher average. The overall trend is still upward, but this analysis highlights when that shift happens. So between additive decomposition, multiplicative decomposition, and change point analysis, we gain a strong descriptive overview of time series data—paving the way for more detailed and nuanced numerical analysis in upcoming methods.



Time series decomposition is a technique used to break down a time series into its **underlying components**, so we can better understand the structure of the data. It's especially helpful when dealing with temporal data that has trends, seasonality, and randomness.

The 3 main components:

1. **Trend (T)**
This is the long-term progression or direction in the data.
Example: airline passengers increasing over decades.

2. **Seasonality (S)**
This is a repeating, predictable pattern within a fixed period (like days, months, quarters).
Example: sales peaking every December.

3. **Residual / Noise / Irregular (R)**
This is what's left after removing the trend and seasonality — basically random fluctuations.

Two common models:

1. Additive Model

Used when:

- The magnitude of seasonal fluctuations **does not change** with the trend.
- Components add linearly:

$$Y(t) = T(t) + S(t) + R(t)$$

2. Multiplicative Model

Used when:

- The seasonal variation **increases or decreases** with the level of the trend.
- Components multiply:

$$Y(t) = T(t) \times S(t) \times R(t)$$

Why it's useful:

- Helps visualize what's driving changes in the time series.
- Allows for **cleaner forecasting**, especially when you want to model or remove seasonality or trend.
- Can be done using functions like `decompose()` in R or `seasonal_decompose()` in Python.

Would you like to see a worked example in R or Python?

Example of Time Series Decomposition

Imagine you're analyzing **monthly sales** for an ice cream shop over 3 years.

Here's a breakdown of what you might see:

Original Time Series (Observed Data)

This is the actual number of ice cream cones sold each month. It shows:

- Sales go up in the summer (June–August)
- Sales dip in the winter (December–February)
- Every year, sales are a bit higher than the previous year

E.g.:

- Jan 2021: 500
- Jul 2021: 2,000
- Jan 2022: 600
- Jul 2022: 2,200
- Jan 2023: 700
- Jul 2023: 2,400

Trend Component

This captures the long-term growth in sales over time.

> Example: Sales are steadily increasing each year due to business growth or population increase.

Seasonal Component

This shows **predictable patterns** that repeat each year.

> Example: Every July has a spike in sales, and every January has a drop — due to weather patterns.

Residual (or Random) Component

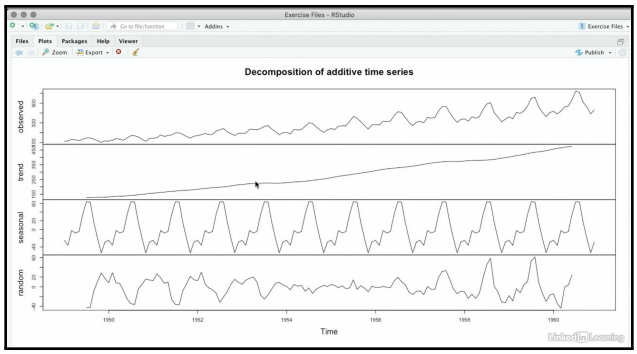
This is what's **left over** after removing trend and seasonality. It captures noise.

> Example: A sudden drop in July 2022 due to an unexpected equipment failure.

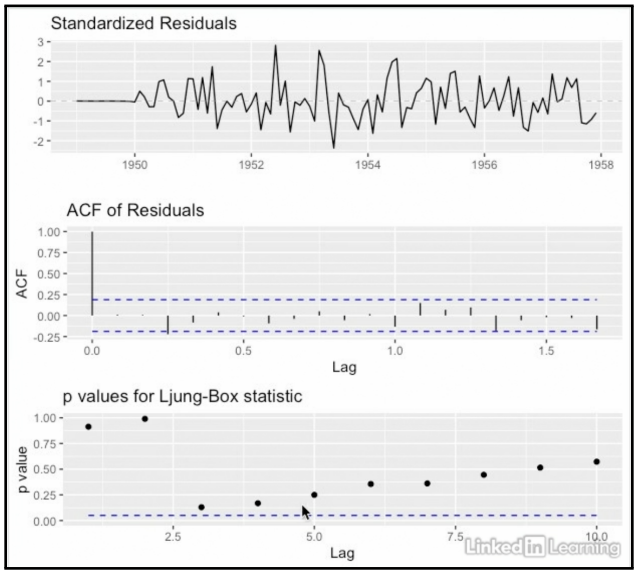
Conclusion

Decomposition breaks the time series into:
Observed = Trend + Seasonality + Residual

This helps you understand what's driving your data, and lets you make better forecasts or decisions.

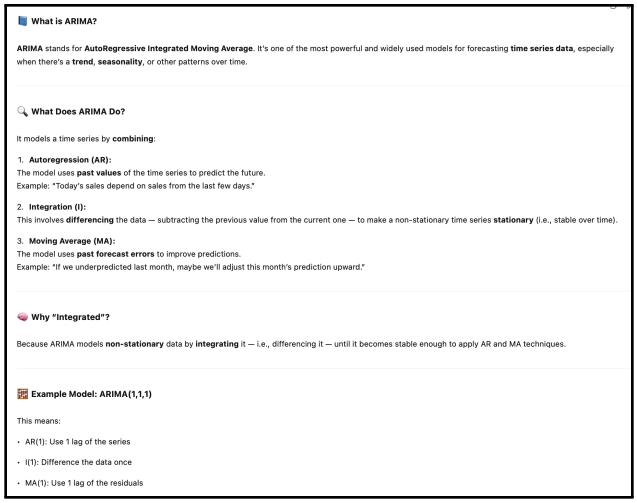


ARIMA



When you have time series data with a strong pattern over time, you can use a purely descriptive visual method like decomposition, which I showed you elsewhere. But one of the most useful methods is called ARIMA, a portmanteau for AutoRegressive Integrated Moving Average. It can make very good predictions when your data has seasonal variation or trends over time. To use it, we're going to load a few new packages—such as `forecast` for time series analysis and `ggfortify` for visualization. After loading those packages and setting a random seed (for reproducibility), we'll use the `AirPassengers` dataset again. It contains monthly airline flight counts from 1949 to 1960, measured in thousands. We'll save the data to `DF` and take a look at the time series: there's a clear upward trend and strong seasonal variation. Since we're doing predictive analysis, it's helpful to split the dataset into a training component and a testing component. With time series data, we don't split randomly—we'll use the years 1949–1957 as training data (through December 1957), and use 1958–1960 as testing data. After plotting both segments, you can see they retain the same seasonal structure. Before running ARIMA, we need to address an important concept: stationarity. A stationary time series is one whose statistical properties (mean, variance, etc.) remain constant over time. Interestingly, ARIMA is designed for non-stationary data, where these properties vary. To diagnose this, we can use a correlogram (ACF plot) to measure how correlated the values are at different lags (e.g., time minus 1, 2, 3 months). Ideally, you want strong correlations that taper off over time. In our case, the ACF plot shows strong correlations outside the non-significant range (indicated by the blue lines at ± 1.96), so the assumption is satisfied. To illustrate why linear regression isn't enough, we fit a simple linear model to the training data—it captures the trend, but misses all the seasonality. So instead, we use the ARIMA model. Thankfully, R provides an `auto.arima()` function, which automatically selects the best model parameters. It returns two sets of numbers: the first set describes the non-seasonal ARIMA part (AR, I, MA), and the second set captures the seasonal components. For example, a model with `(1,1,0)(1,1,1)[12]` means: non-

seasonal AR=1, differencing=1, MA=0; seasonal AR=1, differencing=1, MA=1 with a 12-month period. Next, we generate diagnostic plots using `auto.arima()` again. The residuals appear scattered around zero, indicating a good fit. The ACF plot of the residuals shows no significant autocorrelation beyond lag 0, and the p-values from the Ljung-Box test are all above 0.05, further confirming good model fit. Now we use `auto.arima()` to forecast 36 months ahead with 95% confidence intervals. The resulting forecast matches the shape of the original data extremely well. When we overlay the actual test data from 1958–1960, the prediction closely tracks the observed values. This shows that a seasonally adjusted ARIMA model can predict with high accuracy when the historical pattern is stable and seasonality is consistent. It's a powerful method for time series forecasting, especially when strong seasonal effects are present.



What is ARIMA?

ARIMA stands for **AutoRegressive Integrated Moving Average**. It's one of the most powerful and widely used models for forecasting **time series data**, especially when there's a **trend**, **seasonality**, or other patterns over time.

What Does ARIMA Do?

It models a time series by **combining**:

1. **Autoregression (AR):**
The model uses **past values** of the time series to predict the future.
Example: "Today's sales depend on sales from the last few days."

2. **Integration (I):**
This involves **differencing** the data — subtracting the previous value from the current one — to make a non-stationary time series **stationary** (i.e., stable over time).

3. **Moving Average (MA):**
The model uses **past forecast errors** to improve predictions.
Example: "If we underpredicted last month, maybe we'll adjust this month's prediction upward."

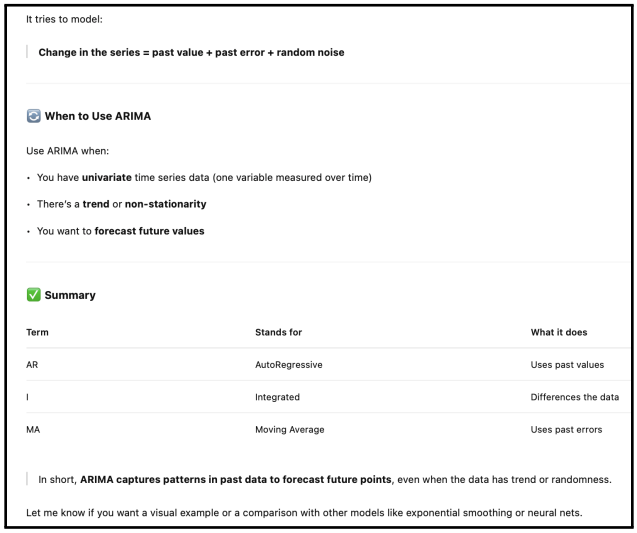
Why "Integrated"?

Because ARIMA models **non-stationary** data by **integrating** it — i.e., differencing it — until it becomes stable enough to apply AR and MA techniques.

Example Model: ARIMA(1,1,1)

This means:

- AR(1): Use 1 lag of the series
- I(1): Difference the data once
- MA(1): Use 1 lag of the residuals



It tries to model:

> **Change in the series = past value + past error + random noise**

When to Use ARIMA

Use ARIMA when:

- You have **univariate** time series data (one variable measured over time)
- There's a **trend** or **non-stationarity**
- You want to **forecast future values**

Summary

Term	Stands for	What it does
AR	AutoRegressive	Uses past values
I	Integrated	Differences the data
MA	Moving Average	Uses past errors

> In short, **ARIMA captures patterns in past data to forecast future points**, even when the data has trend or randomness.

Let me know if you want a visual example or a comparison with other models like exponential smoothing or neural nets.

📊 Example Scenario:

Monthly ice cream sales over 5 years (measured in thousands of units)

Imagine the data looks like this:

Month	Sales (in 000s)
Jan 2018	12
Feb 2018	14
...	...
Jul 2022	40
Aug 2022	38
Sep 2022	35

You observe:

- **Upward trend** over time (more people buying ice cream each year)
- **Seasonal pattern** (sales peak in summer, dip in winter)

You want to **forecast sales for 2023**.

🧠 Step 1: Check for Stationarity

ARIMA models require the data to be **stationary** — meaning the average and variance don't change over time.

> But here, sales are trending upward — so it's **not stationary**.

🔁 Step 2: Difference the Data (Integrated Part)

We compute **monthly changes** in sales instead of using raw sales.

Month	Sales	Difference from Previous Month
Jan 2018	12	—
Feb 2018	14	+2
Mar 2018	15	+1
...	...	...

Now the data **fluctuates around a stable average**, so it's **stationary**.

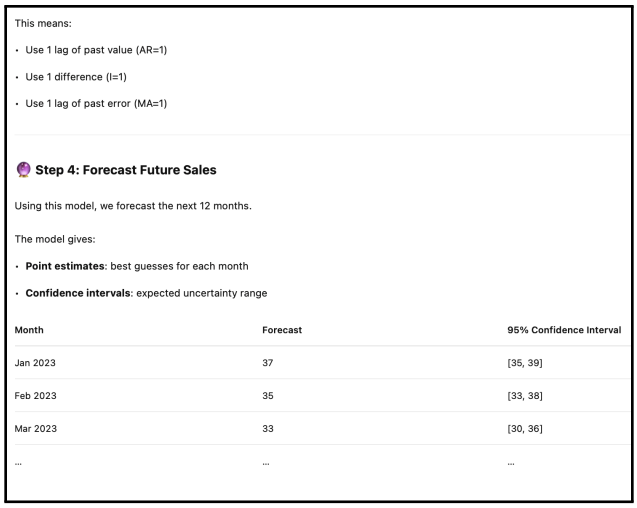
✅ This satisfies the **"I" (Integrated)** component of ARIMA.

🔁 Step 3: Fit AR and MA Terms

- **AR (AutoRegressive):** We check if this month's difference can be predicted from **previous month's difference**.
  - For example: if the last few months had +3, +2, +3, maybe this month will also be around +3.
- **MA (Moving Average):** We check if we can improve predictions by incorporating **previous forecast errors**.
  - For example: if we underpredicted by 2 units last month, we might adjust this month upward by 2.

Let's say our best-fit model ends up being:

$$\text{ARIMA}(1, 1, 1)$$



This means:

- Use 1 lag of past value (AR=1)
- Use 1 difference (I=1)
- Use 1 lag of past error (MA=1)

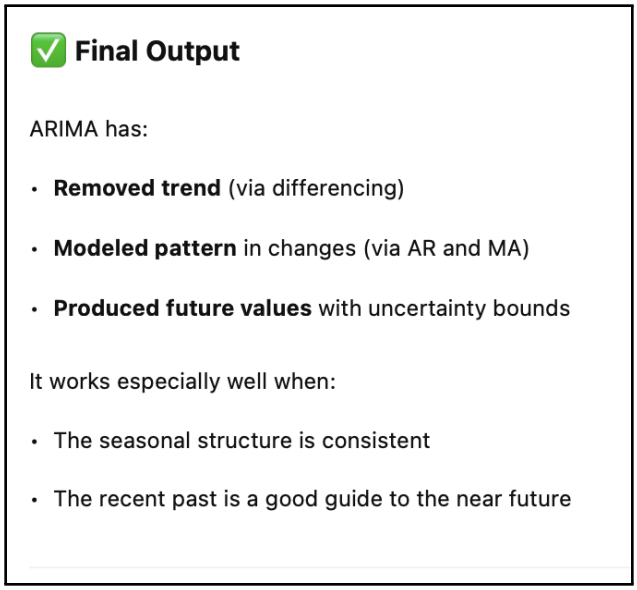
🔮 **Step 4: Forecast Future Sales**

Using this model, we forecast the next 12 months.

The model gives:

- **Point estimates**: best guesses for each month
- **Confidence intervals**: expected uncertainty range

Month	Forecast	95% Confidence Interval
Jan 2023	37	[35, 39]
Feb 2023	35	[33, 38]
Mar 2023	33	[30, 36]
...	...	...



✅ **Final Output**

ARIMA has:

- **Removed trend** (via differencing)
- **Modeled pattern** in changes (via AR and MA)
- **Produced future values** with uncertainty bounds

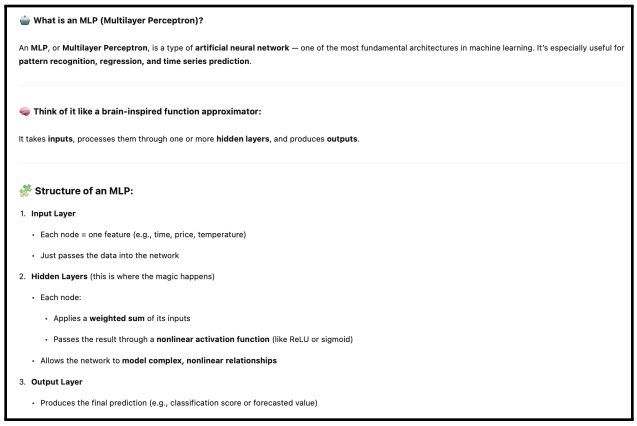
It works especially well when:

- The seasonal structure is consistent
- The recent past is a good guide to the near future

MLP

So much of the progress in data science has been attributable to the rise of neural networks—sophisticated algorithms that mimic, in certain ways, the functioning of the biological brain by having lots of input pass through layers of internal processing to arrive at a final conclusion. While most neural networks are associated with classification algorithms, they also work very well with time series data. One of these, the multilayer perceptron (MLP) neural network, is particularly well adapted to the data we've been using for demonstrations in this chapter. Let's start by loading a few packages, including one called `nnfor`, neural networks for time series data. You're going to need to set the random seed for this one, because randomness plays a big role in some of these procedures—this makes it repeatable so we get the same results. We're going to use the AirPassengers data from the R datasets package. You can save it to `df` and view the data—it contains the number of passengers (in thousands) on international flights from 1949 to 1960. We can plot the data and observe the overall pattern. Just like we did with ARIMA, we're going to split the dataset into training and testing sets, because we're going to try to predict the last three years. We'll use data from 1949 through 1957 as the training set (using the `window` function) and plot it. Then we'll take 1958–1960 as the testing data, plot that too, and prepare to predict it using neural networks. We'll use the MLP to create three models with small variations. Each model will be saved as `fit`. First, we take the training data and run MLP—it may take some time depending on your system. Once fit, we forecast 36 months into the

future (since the data is monthly) using `h = 36`, saving the predictions as `pred1`. We display and plot the predictions—the main prediction appears in blue. Due to the randomness in MLP, you might get spikes if you use a different seed, but the blue line is your overall forecast. Then, we overlay the actual observed data in red and see that the model matches it fairly well, especially capturing the seasonal variation. The second model uses a different method for setting the number of hidden nodes: holdout validation with 20% of the training data. We specify `hidden = "auto"` and `valid = TRUE`. On my machine this took about a minute. We again forecast 36 steps ahead, save it as `pred2`, display and plot it. This time, the confidence intervals are much tighter, with minimal variation, and the blue predictions again closely match the red observed data, though the real data is slightly more extreme. Lastly, we run a third version of MLP, this time using five-fold cross-validation to automatically choose the number of hidden nodes. This took about 5.5 minutes on my machine. We forecast 36 months ahead and save it as `pred3`. Once plotted, we again see very tight predictions. Overlaying the actual observed values, we notice the model captures the seasonality but doesn't quite reach the extremes of the real data. To compare the models, we look back at our three charts: the first with fixed five hidden nodes, the second with holdout validation, and the third with cross-validation. All three are useful, though the first had more variation and was less accurate. The second and third were more precise, showing better alignment with the observed values. What this demonstrates is that neural networks—in this case, a multilayer perceptron—can be a powerful and accurate method for forecasting time series trends, especially when strong seasonal variation is present. Ultimately, when trying to predict an outcome, it helps to have multiple tools available, choose the method best adapted to your data, and compare models to increase your confidence in the final result.



What is an MLP (Multilayer Perceptron)?

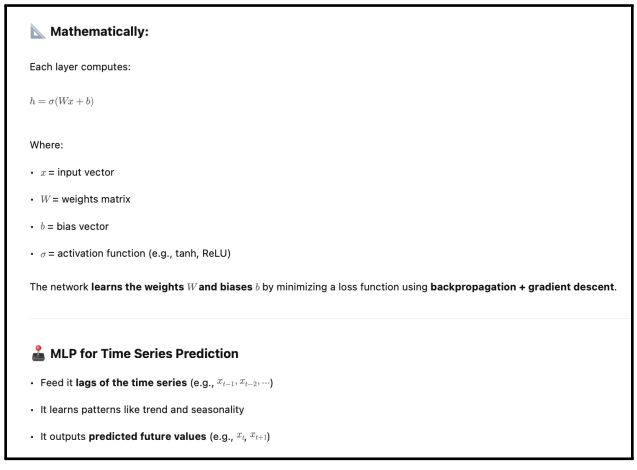
An **MLP**, or **Multilayer Perceptron**, is a type of **artificial neural network** — one of the most fundamental architectures in machine learning. It's especially useful for **pattern recognition, regression, and time series prediction**.

Think of it like a brain-inspired function approximator:

It takes **inputs**, processes them through one or more **hidden layers**, and produces **outputs**.

Structure of an MLP:

1. **Input Layer**
   - Each node = one feature (e.g., time, price, temperature)
   - Just passes the data into the network
2. **Hidden Layers** (this is where the magic happens)
   - Each node:
     - Applies a **weighted sum** of its inputs
     - Passes the result through a **nonlinear activation function** (like ReLU or sigmoid)
   - Allows the network to **model complex, nonlinear relationships**
3. **Output Layer**
   - Produces the final prediction (e.g., classification score or forecasted value)



Mathematically:

Each layer computes:

$$h = \sigma(Wx + b)$$

Where:

- x = input vector
- W = weights matrix
- b = bias vector
- σ = activation function (e.g., tanh, ReLU)

The network **learns the weights** W **and biases** b by minimizing a loss function using **backpropagation + gradient descent**.

MLP for Time Series Prediction

- Feed it **lags of the time series** (e.g., $x_{t-1}, x_{t-2}, \cdots$)
- It learns patterns like trend and seasonality
- It outputs **predicted future values** (e.g., x_t, x_{t+1})

✅ Why MLPs Are Useful

- **Flexible**: Can fit nearly any function
- **Fast**: Simple and efficient for small to medium datasets
- **Accurate**: Especially with regularization and tuning

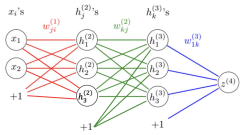
⚠️ Limitations

- MLPs **don't remember time** unless you manually include lags
- For sequential memory, use **RNNs or LSTMs**

TL;DR:

> An **MLP** is a basic feedforward neural network with at least one hidden layer.
> It takes input features, pushes them through nonlinear layers, and outputs predictions — including for **time series** if you prep the data right.

Matrix notation



With this matrix notation, we can write:

$\bar{h}^{(1)} = \bar{x}$
$\bar{z}^{(2)} = W^{(1)}\bar{h}^{(1)} + \bar{b}^{(1)}$
$\bar{h}^{(2)} = g(\bar{z}^{(2)})$
$\bar{z}^{(3)} = W^{(2)}\bar{h}^{(2)} + \bar{b}^{(2)}$
...
$\bar{z}^{(j+1)} = W^{(j)}\bar{h}^{(j)} + \bar{b}^{(j)}$
$\bar{h}^{(j+1)} = g(\bar{z}^{(j+1)})$
...

stack

$$z_1^{(2)} = w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{10}^{(1)}$$
$$z_2^{(2)} = w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{20}^{(1)}$$
$$z_3^{(2)} = w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{30}^{(1)}$$

$$\begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \\ w_{31}^{(1)} & w_{32}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} w_{10}^{(1)} \\ w_{20}^{(1)} \\ w_{30}^{(1)} \end{bmatrix}$$

$\bar{z}^{(2)} = W^{(1)}\bar{x} + \bar{b}^{(1)}$ — $b \to$ bias

$\bar{h}^{(2)} = g(\bar{z}^{(2)})$ → generic activation

Training neural networks: stochastic gradient descent

1. Initialize weights at small random values $\bar{\theta}^{(0)} \sim$ Gaussian (0, r)
2. Shuffle and sample one data point
3. Update the weights as follows

$$\bar{\theta}^{(k+1)} = \bar{\theta}^{(k)} - \eta_k \nabla_{\bar{\theta}} \text{Loss}(y^{(i)}, h(\bar{x}; \bar{\theta}))$$

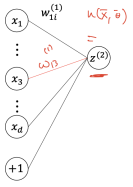
$\bar{\theta}^{(k+1)}$: weight for next step; $\bar{\theta}^{(k)}$: current step weight; η_k: LR (magnitude); $\nabla_{\bar{\theta}} \text{Loss}(y^{(i)}, h(\bar{x}; \bar{\theta}))$: direction

Is SGD enough? Single layer updates

- Single layer NN with no non-linearities, and hinge loss
- Goal: get parameter updates for all $\bar{\theta} = [w_{10}^{(1)}, w_{11}^{(1)}, \ldots, w_{1d}^{(1)}]$
- Focus on one component of $\bar{\theta}$. E.g., $w_{13}^{(1)}$

$$\begin{aligned} \text{Loss}(y, h(\bar{x}; \bar{\theta})) &= \max\{1 - yh(\bar{x}; \bar{\theta}), 0\} \\ &= \max\{1 - yz^{(2)}, 0\} \\ &= \max\{1 - y(\sum_i^d w_{1i}^{(1)} x_i + w_{10}^{(1)}), 0\} \end{aligned}$$

definition



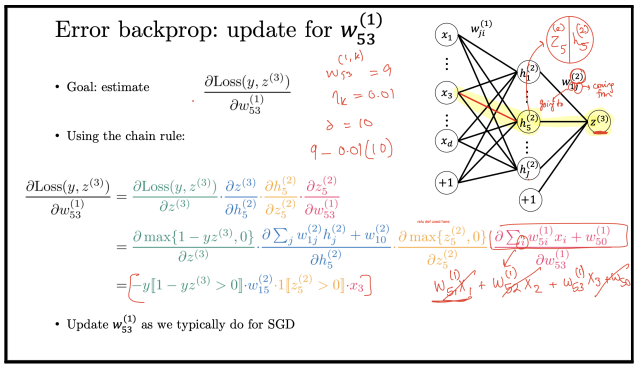
- Derivative wrt to $w_{13}^{(1)}$

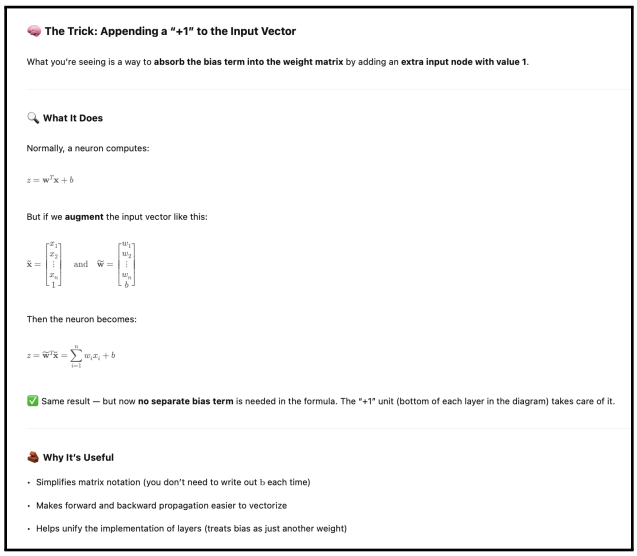
$$\frac{\partial \text{Loss}(y, h(\bar{x}, \bar{\theta}))}{\partial w_{13}^{(1)}} = \begin{cases} -yx_3, & \text{if } 1 - yz^{(2)} > 0 \\ 0 & \text{otherwise} \end{cases}$$

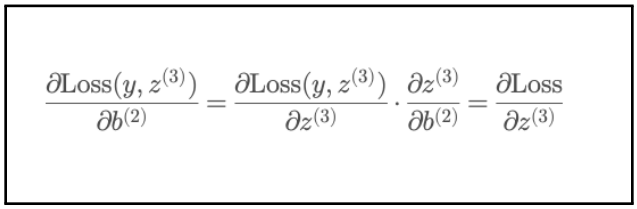
- Update for the $k+1$ SGD iteration:

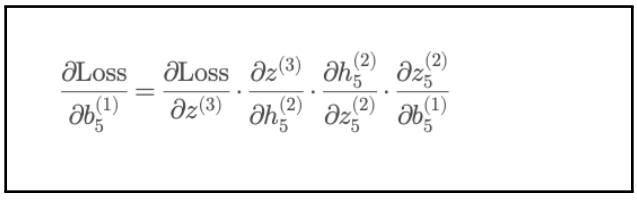
$$w_{13}^{(1,k+1)} = w_{13}^{(1,k)} + \eta_k yx_3[\![1 - yz^{(2)} > 0]\!]$$

layer; SGD iteration; LR









Challenge: Decomposition

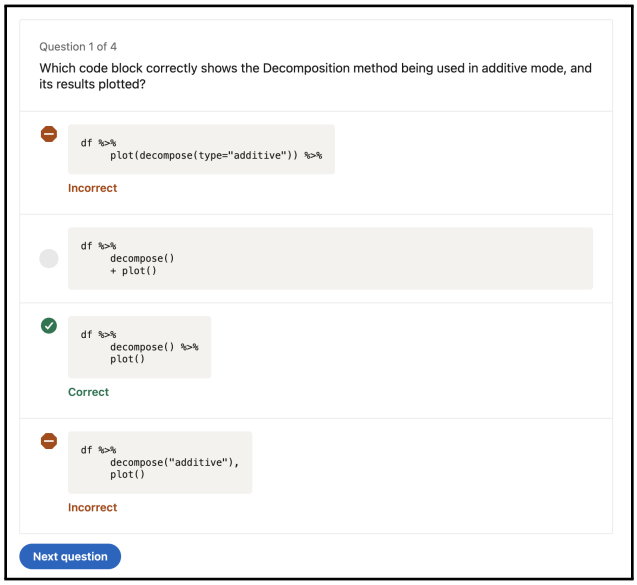
At this point, I want to invite you to try your hand at working with time series data, and to do this, we'll perform time series decomposition—a really fabulous big-picture analysis of what's happening with your data. We're going to use a dataset that comes from R: the EU Stock Markets data, which is available from the `datasets` package. So let's come down here and get a little bit of information on that. It's the daily closing prices of several major European stock indices from 1991 to 1998. You can view the data, and one of the interesting things about it is that it uses decimal dates—these indicate how far into the year you are. So let's come back up to the top here. We started in 1991, about halfway through, and we have four different indices: the DAX, which is in Germany; the SMI, which is in Switzerland; the CAC, in France; and the

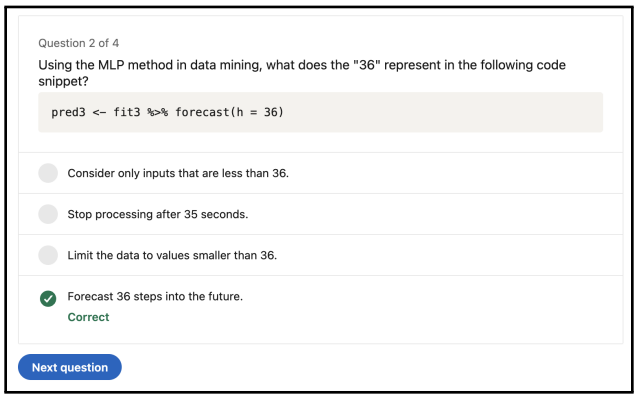
FTSE, in the UK. We can plot the data, and we can actually plot all four time series at once by simply taking it and feeding it into R's generic `xyplot` command. When we do that, you can see that they all show a similar pattern: bumping up around 1994, dipping down a bit, and then going up again. An interesting one is the CAC, which has a more pronounced dip around 1998. So I'd like to invite you to focus on the CAC dataset. Part of the reason is that decomposition is a lot easier with just one time series. The CAC, which stands for *Cotation Assistée en Continu* or continuous assisted trading, will be the focus. We'll take that and save it to `df` by using square brackets—using `[ ,3]` means to take just the third column—and we can plot just that data by itself. Here's the pattern we're going to work with. Your challenge is to take this dataset and analyze what kind of overall trend exists, what kind of seasonal trend is present, and what kind of random variation remains. Take a couple of minutes, give it your best shot, and then we'll meet together again to compare responses in decomposing financial data from the French CAC.

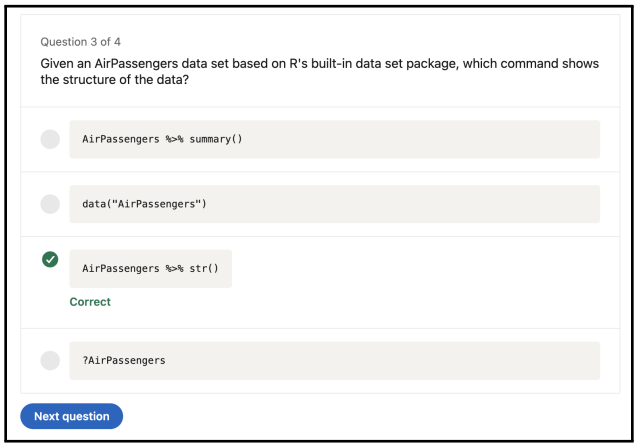
Solution: Decomposition

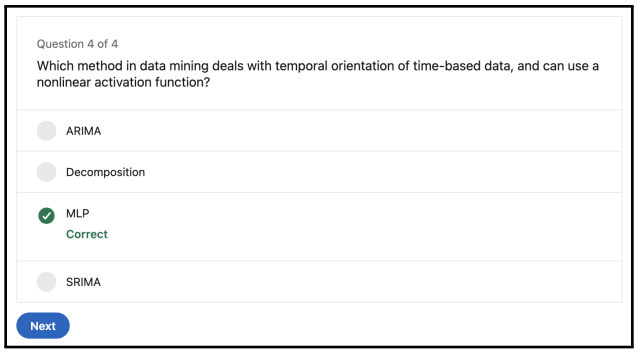
I invited you to look at some financial data and see what kind of patterns you could find by decomposing a time series. Let me show you how I did this. Let's start by loading some packages and then getting some information about the EU Stock Markets dataset, which is part of R's built-in datasets. We'll see the data right there and plot it. So here's our data plotted out. We're going to focus on the third one—the French CAC—and I want to save that into `df`, and then we can explore that dataset by examining the time series. Here are all the values for just the French CAC. We can get the structure—it's a dataset of almost 2000 observations. Let's get the histogram. Now, this ignores the time, so I take `df` and ask for a histogram, adding a main title and a label on the x-axis. You can see that the closing prices were generally pretty close to 2000, although it's skewed—some values go much higher. This makes sense when you actually plot the time series by taking the data frame and running it through the generic `plot()` command with some titles and axis limits. You can see it's hovering around 2000 for much of the time until, right around '96–'97, it takes a strong turn upward, which is a similar pattern we saw in the other three major European financial indices. But let's decompose this time series and see what's going on. To do this, it's really simple—we just use the `decompose()` function from R's built-in `stats` package. We're going to use the additive model, which is the default. So I take `df`, run it through `decompose()`, and plot the results. When we do that, you can see a general pattern where values remain low and then take a strong turn up around '96. There's also a seasonal pattern—both an annual component and smaller fluctuations. You can observe a consistent annual pattern: it rises mid-year, then dips just before the new year. Now, we can also try this with a multiplicative trend, which is often better for financial data because it tends to grow exponentially—by a fixed percentage over time. To do that, all we have to do is specify `type = "multiplicative"` and repeat the process. Again, we see the same overall pattern and seasonal variation, and both appear to be consistent, as does the residual (random) component. So I think we've found something valuable here. As an added bonus, I'm going to perform a change point analysis, because I really enjoy doing that. We simply take `df` and tell R we're looking for changes in the mean (average value), using a normal distribution. We'll plot it and also print the change point locations in the console. What you see here is that the data was relatively stable until about halfway through 1997, at which point the mean shifted enough to assign it a new value. It would be nice to draw an angle here, but if we're simply looking for a structural change in state, it happened about one-third of the way through 1997. That's another great way of breaking down a time series and getting a big-picture view of it. Hopefully, you were able to do something similar and gain insights of your own. Now you've got something else to add to your data mining toolkit.

Chapter Quiz









Text Mining

Text mining overview

Text is not like other data, and when it comes to data mining, it poses some very special challenges. Those challenges include things like the fact that there are enormous quantities of text—there is so much open text in terms of books, news articles, and social media. It's completely overwhelming. Also, it's enormously variable. There are so many different words, phrases, misspellings, and colloquialisms. There's a lot there. What this all tells you is that it's unstructured—it doesn't fall into nice little rows and columns of data; it just kind of is what it is. That makes it a very difficult thing to mine. And when you're mining for data, you're trying to get value. You're trying to get more than, like Hamlet saying that he is reading "words, words, words"—you're trying to get meaning and actionable insight out of your data. Now, there are three ways that I can think of where data mining with text is particularly important. One application is in modeling topics, where algorithms are used to identify what a news article is about or what the topic of a social media or blog post is. Another application is summarizing content. Can it take an entire scientific article and boil it down to a paragraph? Or can it take the text of a political speech and reduce it to a few sentences that capture its essential meaning? A third application is classifying text—taking unstructured text and putting it into nice, neat groups. The prototypical example is a spam filter, but anything that sorts data is going to be enormously useful and save a lot of human time and effort. In the demonstrations we're going to look at, I'm going to focus on another element of text mining, which is called sentiment analysis. This is where you're trying to find the emotional content of data. The place where this is really significant is in online marketing—you want to know if people are saying positive or negative things about the product or service you're offering. That allows you to tailor both the product or service you offer, as well as the appeal you make and who your audience is. We're going to start by looking at binary sentiment. This is where you're simply looking at whether somebody says something that is positive or negative—this or that, either/or. Now to do this, it's not like you sit around and decide whether somebody is positive or not—you use a published lexicon. This is some information you import into your computer program: a list of words and their associated sentiments. What's nice about this is that simply categorizing things as positive or negative is by far the quickest and easiest form of sentiment analysis in text. If you want to go beyond that—and I'm going to go beyond that—you can also look at the scoring of sentiments. This is not just a binary yes/no but instead uses a scale where words are scored by the strength of the emotions. For instance, we're going to use a scale ranging from negative five (most negative) to positive five (most positive). What you're relying on here is another lexicon. This time it contains sentiment associations for individual words. The fact that they're individual is important—it's looking at one word at a time. You can develop other systems for looking at associations between words—for instance, the negation of a topic or a phrase that has a double meaning. In fact, that gets us to the final thing I want to look at, which isn't sentiment analysis per se, but rather word pairs or the associations between words. It answers the question: which words go with which others, and are there unexpected associations in the text? It's like trying to sort through your socks and see, "this goes with this," or "that goes with that." Is there a predictable structure to your data? What you'll find sometimes is that different texts pair things differently—that they tend to use certain words with a negation or tend to use them in a positive or negative way. In the demonstrations we'll look at, we will explore all three of these methods for finding meaning in unstructured text. We'll look at binary sentiment,

we'll look at scored sentiment, and we'll examine the structure of word pairs through visual methods. All of these approaches can help get you started on finding useful and actionable meaning from your text.

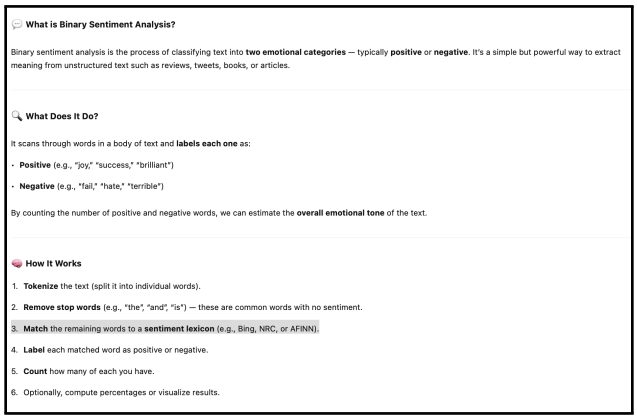
Dataset: The Iliad

I may be a data person, but it turns out that I really love art. I have a background in design, I'm married to an artist, and I'm really happy to be working on this section of text mining because it allows me to engage a little bit with some literature. In this particular case, I'm going to be using an example of the *Iliad* by Homer as a way of demonstrating three different approaches to text mining. To do this, we're going to download the *Iliad* from Project Gutenberg, an online site that has over 60,000 free books. They're free from copyright and are one of the great places for learning how to work with data. Now, the easiest way to do this is with a package called `gutenbergr`, which imports Gutenberg texts and helps get them set up. I've actually done some of this for you already, but let's start by loading these packages. I'm also making a point of downloading `tidytext`. This is one of the great packages for working with text data in R. To download the *Iliad* from Project Gutenberg, you need to know the book's site. Its URL is here, and its ID is 6150. I'm going to use the function `gutenberg_download`. You simply give it the site, and then you can tell it to strip headers and footers, which the book may or may not have. And we can run that one. You can see it's shown up right here. We can look at the first few rows by simply asking `cdf`. This is the title page. This first column is the ID number, and it's going to be 6150 on all 20,000 rows because that's the book that it comes from. This is the *Iliad* of Homer, and this is to indicate a row with no text on it. We can see that it was rendered into English blank verse by Edward, Earl of Derby. You can get some summary statistics. It's kind of a silly thing to do here, because it has one variable that doesn't vary—that's the ID 6150 all the way through. And when you have texts, it simply tells you how many rows there are. There are 20,128 rows. But what I'm going to do is a little bit of data prep for you. I'm going to take this information and save it into the `data` folder under the name `Iliad.txt`. And once we've gotten that, we've got the data that we need for the next three demonstrations of text mining in R.

Sentiment analysis: Binary classification

Our first demonstration of text mining in R is going to be the simplest case. We're going to look at sentiment—that is, emotional evaluations of texts—which is the binary positive or negative evaluation. And to do this with the *Iliad*, I'm going to start by importing some packages, and `gutenbergr` and `tidytext` are the important ones. Then we're going to import the data. Now, this is from Project Gutenberg, which is available online, but I've already saved the data and I'm going to simply import it from our data folder where it's `elliot.txt` and save it as a tibble into `df`. That stands for data frame and just makes it easy for me to reuse my commands. Let's take a look at the first few rows, and what you see here are the ID number for the text (6150) and the first few rows from the title page. Now, the first thing we need to do is prepare the data, and that involves tokenizing the data. A token is an element of text, and usually it refers to a word. You can have tokens of multiple words or even sentences or paragraphs or what have you, but most commonly it's a word. And so what we're going to do is take our data frame, select just the `text` variable (that's the variable on the right), drop the Gutenberg ID, and use a function called `unnest_tokens`. That says: split them apart. So we're going to split the text into words and save it as text format. Let's do that right here. I'm going to overwrite the data frame (that's what the special assignment operator means here). And now you see we have a lot more observations—146,000—but this time just one variable. If you want to see how common each word is, we can take the data frame and use `count` to count the words and sort by their frequency. Not surprisingly, "the" is the most common word, then the conjunction "and," then "of," "to," etc. Most of these are really not very interesting words, although there have been interesting projects done using these little words. These are called stop words. They're little common words like "to" and "so on." For our purposes, we're going to remove those, because if we're looking at sentiment, these usually have no sentiment or emotional value. So we're simply going to remove them. When we brought in `tidytext`, we also brought in something called `stop_words`. It's an index, a lexicon of words, and we're going to do an `anti_join`. That says: use the data frame and remove anything that's in `stop_words`. Once we do that, you'll see we go from 146,000 observations to only 70,000. So we lost half of them because stop words accounted for 50% of the text. Now, let's see the revised tokens by frequency. In the *Iliad*, the

most common are “son,” then “thou” and “thy,” “greeks,” “achilles,” and “hector,” the main characters, “ships,” “thee,” “jove”—and those ones make sense for what we’re dealing with. In terms of categorizing the sentiments, you have to have a lexicon. You have to have something that actually says, “this is a positive word,” “this is a negative word.” When we bring in `tidytext`, we also access several lexicons. We’re going to use one that’s called `bing`, and this gives us the positive and negative evaluations of words. What we’re going to do now is find which words are positive and negative by doing what’s called an `inner_join`, which means: find all the words in `df` that have sentiments listed in `bing`. Just run that, and as an example, you can see that “modern” is positive, “success” is positive, “sufficiently” is positive. Meanwhile, “diffidence” is negative, “regret” is negative—and those generally make sense. Now, of course, we’re using a book that is very, very old—and even a translation that is old—so some of the words may have changed their meaning in terms of context and usage, but this gets us close. Now what we’re going to do is sort the sentiment words by their frequency. So the most frequently occurring positive and negative words—we’ll use the same command as before, but add `count word` and then `sort`. When we look at that, the most common word with sentiment value is “death,” which is negative; “brave,” which is positive; “fell,” which is negative; “mighty” and “valiant,” which are positive. These are the most common words. By the way, “death” is in there a lot because this is a story about a war—the Trojan War. Now, let’s summarize these. All I’m going to do is take our data frame that is the text with just the words, do the `inner_join` again, and this time group by sentiment. We’re going to put the positive words together, the negative words together, and summarize how many and what proportion of the total. When we do this, you’ll see we have 6,386 negative words, which is 60% of the words that have sentiment value in the text, and we have 4,206 positive words, or not quite 40%. This isn’t surprising because the *Iliad*’s a heavy story—it’s warfare and things going badly. So of the emotionally laden words, we have 60% negative, 40% positive. And that actually is a good reflection of the tone of the book. It doesn’t tell us anything about the plot, the characters, or the storyline, but it lets us know that we are generally focusing on topics that have negative valence or affect or emotion associated with them. So that’s a very quick binary classification in text in R.



💬 **What is Binary Sentiment Analysis?**

Binary sentiment analysis is the process of classifying text into **two emotional categories** — typically **positive** or **negative**. It's a simple but powerful way to extract meaning from unstructured text such as reviews, tweets, books, or articles.

🔍 **What Does It Do?**

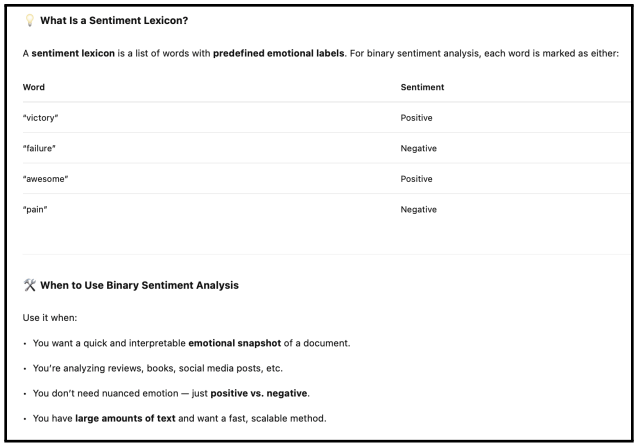
It scans through words in a body of text and **labels each one** as:

- **Positive** (e.g., "joy," "success," "brilliant")
- **Negative** (e.g., "fail," "hate," "terrible")

By counting the number of positive and negative words, we can estimate the **overall emotional tone** of the text.

🧠 **How It Works**

1. **Tokenize** the text (split it into individual words).
2. **Remove stop words** (e.g., "the", "and", "is") — these are common words with no sentiment.
3. **Match** the remaining words to a **sentiment lexicon** (e.g., Bing, NRC, or AFINN).
4. **Label** each matched word as positive or negative.
5. **Count** how many of each you have.
6. Optionally, compute percentages or visualize results.



💡 **What Is a Sentiment Lexicon?**

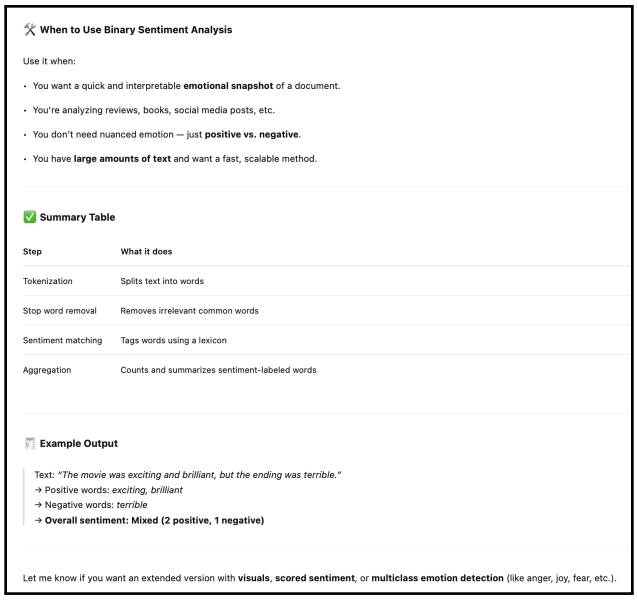
A **sentiment lexicon** is a list of words with **predefined emotional labels**. For binary sentiment analysis, each word is marked as either:

Word	Sentiment
"victory"	Positive
"failure"	Negative
"awesome"	Positive
"pain"	Negative

🛠️ **When to Use Binary Sentiment Analysis**

Use it when:

- You want a quick and interpretable **emotional snapshot** of a document.
- You're analyzing reviews, books, social media posts, etc.
- You don't need nuanced emotion — just **positive vs. negative**.
- You have **large amounts of text** and want a fast, scalable method.



🛠️ **When to Use Binary Sentiment Analysis**

Use it when:

- You want a quick and interpretable **emotional snapshot** of a document.
- You're analyzing reviews, books, social media posts, etc.
- You don't need nuanced emotion — just **positive vs. negative**.
- You have **large amounts of text** and want a fast, scalable method.

✅ **Summary Table**

Step	What it does
Tokenization	Splits text into words
Stop word removal	Removes irrelevant common words
Sentiment matching	Tags words using a lexicon
Aggregation	Counts and summarizes sentiment-labeled words

📄 **Example Output**

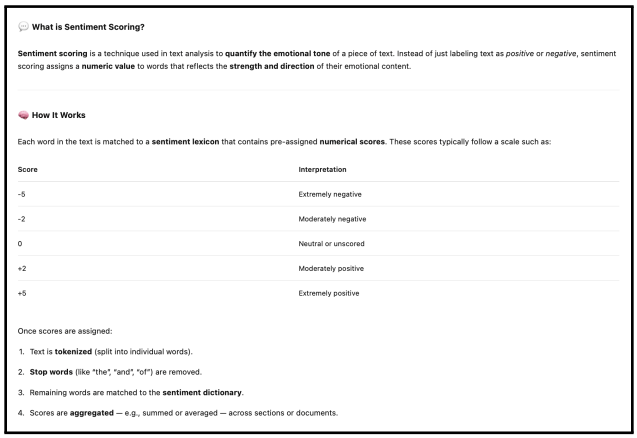
> Text: *"The movie was exciting and brilliant, but the ending was terrible."*
> → Positive words: *exciting, brilliant*
> → Negative words: *terrible*
> → **Overall sentiment: Mixed (2 positive, 1 negative)**

Let me know if you want an extended version with **visuals**, **scored sentiment**, or **multiclass emotion detection** (like anger, joy, fear, etc.).

Sentiment analysis: Sentiment scoring

In the last video I showed you how to do binary classification of sentiment, either positive or negative. In this demonstration, we're going to get a little more sophisticated and actually score not just whether it's positive or negative, but how positive or how negative. We're going to use the same text, the *Iliad*, and we'll use the same packages, but a different lexicon, which allows us to do different things. Let's come down here and load our packages, which include `gutenbergr` and `tidytext`. We're also loading `textdata`, which allows a little bit of extra functionality. We'll bring in the `iliad.txt`, which I've already saved for us, and we'll put it into a tibble and look at the first few rows. This is the data we've seen before—we have the Gutenberg ID, which is the ID for the *Iliad*, and here are the first few lines from the title page. We need to do a little bit of data preparation. The first thing I'm going to do is add line numbers to the text. This is important because it will allow me to split up the text in a moment. I'm also going to remove the Gutenberg ID and then overwrite the data using the assignment operator. So now we have just the text—it's the same as before, but now we have line numbers: one, two, three, and so on. Next, we're going to tokenize the data. That means we're going to split it into the elements we want—usually one word—and that's what we'll do here. But it could also be word pairs, triplets, sentences, etc. We're going to break it down into individual words and save it into `words`. Now you can see we've gone from 20,000 lines to 146,000 words. At this point, we can calculate sentiment scores using the AFINN lexicon. This assigns a sentiment value to many words, ranging from `-5` (most negative) to `+5` (most positive). It doesn't assign zeros—neutral words are simply excluded—so we're only scoring words with emotional or affective value. I'm going to create a new object called `score_freak` (for frequency). We'll do an `inner_join` between our words and the AFINN lexicon to match each word's sentiment score. We'll group the words by sentiment value—group all the `-5`s together, then `-4`s, etc.—and count how many words fall into each category. When we print that, we see values ranging from `-5` (only 12 words) up to `+2` (2,353 words), with common values around `-2` and `+2`. We can visualize this with a bar chart using `ggplot2`, setting x-axis breaks from `-5` to `+4` and adding labels and titles. This shows a spike at `-2` and `+2`, with no neutral (0) scores since they're skipped. Now comes the really neat part. Stories—especially epics like the *Iliad*—have a narrative arc: build-up, crisis, and resolution. We can visualize this emotional arc by analyzing how sentiment changes throughout the story. To do this, I split the text into arbitrary sections of 100 lines (which is why we added line numbers), then calculate an average sentiment score for each section. I create an object called `score_arc`, where I join the sentiment values to each word, group them by section (using line number integer division by 100), and then summarize each section by its average sentiment score. When we print this, we see average sentiment values for each block of 100 lines. To visualize the narrative arc, I plot these scores by section, adding a reference line at 0, a gray line for average sentiment, and a smoothing line. The red zero line marks neutral sentiment. The result shows

sentiment starting generally positive, peaking around the 15th section, then trending downward—crossing into negative territory halfway through, and reaching a low point before gradually rising again toward zero. The book, being about war, has its emotional low points, but it does rise again as the story resolves. This creates a visual narrative arc derived from sentiment analysis of the text. It's a powerful demonstration of how sentiment scoring and text mining—made accessible by packages like `tidytext`—can extract deep emotional structure from literary texts.



💬 **What is Sentiment Scoring?**

Sentiment scoring is a technique used in text analysis to **quantify the emotional tone** of a piece of text. Instead of just labeling text as *positive* or *negative*, sentiment scoring assigns a **numeric value** to words that reflects the **strength and direction** of their emotional content.

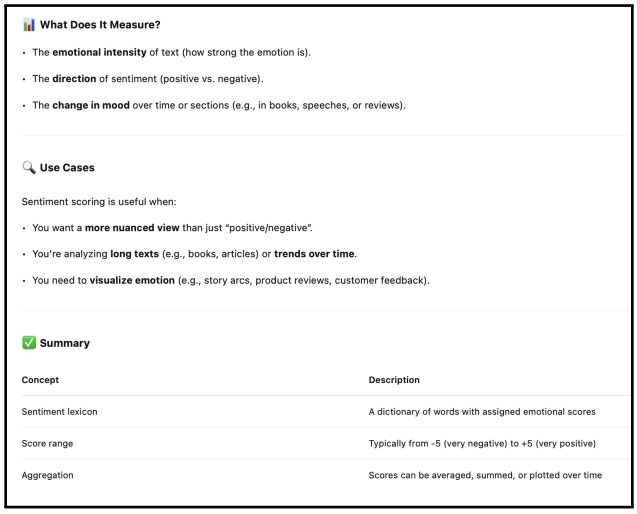
🧠 **How It Works**

Each word in the text is matched to a **sentiment lexicon** that contains pre-assigned **numerical scores**. These scores typically follow a scale such as:

Score	Interpretation
-5	Extremely negative
-2	Moderately negative
0	Neutral or unscored
+2	Moderately positive
+5	Extremely positive

Once scores are assigned:

1. Text is **tokenized** (split into individual words).
2. **Stop words** (like "the", "and", "of") are removed.
3. Remaining words are matched to the **sentiment dictionary**.
4. Scores are **aggregated** — e.g., summed or averaged — across sections or documents.



📊 **What Does It Measure?**

- The **emotional intensity** of text (how strong the emotion is).
- The **direction** of sentiment (positive vs. negative).
- The **change in mood** over time or sections (e.g., in books, speeches, or reviews).

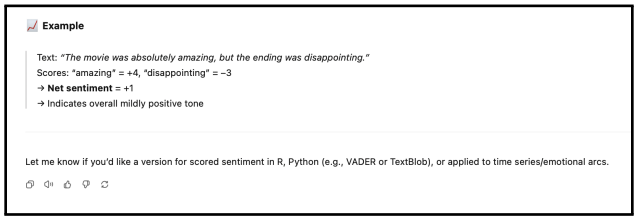
🔍 **Use Cases**

Sentiment scoring is useful when:

- You want a **more nuanced view** than just "positive/negative".
- You're analyzing **long texts** (e.g., books, articles) or **trends over time**.
- You need to **visualize emotion** (e.g., story arcs, product reviews, customer feedback).

✅ **Summary**

Concept	Description
Sentiment lexicon	A dictionary of words with assigned emotional scores
Score range	Typically from -5 (very negative) to +5 (very positive)
Aggregation	Scores can be averaged, summed, or plotted over time



📈 **Example**

> Text: *"The movie was absolutely amazing, but the ending was disappointing."*
> Scores: "amazing" = +4, "disappointing" = −3
> → **Net sentiment** = +1
> → Indicates overall mildly positive tone

Let me know if you'd like a version for scored sentiment in R, Python (e.g., VADER or TextBlob), or applied to time series/emotional arcs.

Exactly — that's the core idea. ✅

The difference between **binary sentiment analysis** and **sentiment scoring** is:

Feature	Binary Sentiment	Sentiment Scoring
Goal	Positive vs. Negative	How positive or how negative
Lexicon values	Labels only (e.g., "positive", "negative")	Numeric scores (e.g., −5 to +5)
Granularity	Coarse (yes/no)	Fine-grained (strength of emotion)
Common Use Case	Quick classification	Emotional intensity, mood tracking
Example Output	"Negative"	"−3" (strong negative)

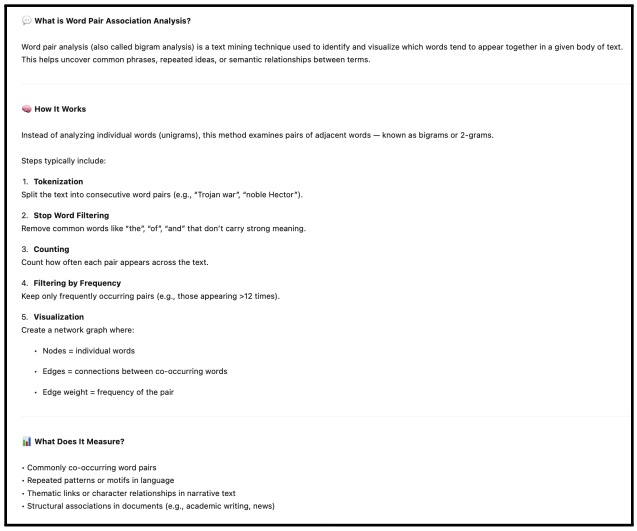
So yes — same basic pipeline (tokenize, remove stopwords, match to lexicon), but instead of just asking **"Is this word emotional?"**, you now ask **"How emotional is it — and in which direction?"**

Let me know if you want a hybrid method too (e.g., scoring over time *and* classifying by threshold).

Visualizing Word pairs

Our final demonstration of text mining is actually going to look at a visualization of the association between words, and in this case, word pairs. To do this I'm going to begin by loading a few packages including ggraph, which allows us to visualize networks. That's going to be important for what we do here. So I'm going to load those packages. I'll set a random seed 'cause that can affect the graphs. And then we're going to use "The Iliad," the same data we've used before. We'll bring it in from the data folder, save it as a tibble and we'll look at the first few rows. We've seen this several times, it's the same. And then we'll prepare the data. We're going to tokenize the data by unnesting it, and except this time we're going to save it into something called wordpairs, not word, but wordpairs. We're going to start with the text and we're going to split it into ngrams. An ngram is a word that means this many words at a time. And in this case, the n is two. I've also seen these called bigrams but I'll just call them wordpairs. So let's do that by overriding our data with wordpairs. And then we can see the tokens are words by frequency. And this time you see two words at a time, "to the" happens 527 times, "of the" is 453, "in the", "from the" — these are things that are not necessarily interesting. We have an issue again with what are called stop words. These are the small words that make up a substantial portion of the text, but they're just connectors and they're usually not the ones that we're interested in, especially if we're doing something that involves like sentiment. And so what we're going to do is we're going to get rid of these entries that have stop words in them. Now, because we have two words this gets a little more complicated. What we're going to do is we're first going to separate the word pairs into two variables, word1 and word2, where we're going to use a space as the separator. We'll also remove the gutenberg ID and we'll print the results and we'll overwrite the data. And once we've done that, you see that the word pairs are "the iliad", "iliad of", "of homer", "homer rendered", and so we haven't gotten rid of the stop words just yet, but we've made it possible. Now what we're going to do is remove any word pairs that have stop words. We do this in a slightly unusual way. We take our data frame and then we're going to filter. We're saying the exclamation mark is a negation here. It means where word1 is not included in stop words and then the word actually refers to an element of the stop words library. And then we're going to say the same thing for word2 — that word2 is not in the stop words — and we'll print it. Now, this is going to reduce the number of observations dramatically. Right now we have 146,000, it's going to reduce it to 81% to 28,000. But let's run that command and get rid of the word pairs that include at least one stop word. Now we can see the sorted counts by saying count word1 and word2, sort them, and print them. And here is our stop-word-free data: "Peleus's son", "Saturn's son", "Atreus son" — these are phrases that appear a lot of times because in "The Iliad" characters are frequently referred to as somebody else's son. We have "battle field", "immortal gods", and these are phrases that make sense. The next thing we can do is we can look at the frequencies and get both the number and the proportions. So let's take a look at that. And what we see is that there are many pairs of words that appear only once. And in fact that's 16,000 and that's a large proportion. We have some that appear 10 times, that's only 23 pairs that appear 10 or more times. And this is a way of seeing really the distribution of frequencies of pairs. But now let's visualize the data. What we're going to do is we're going to take the data frame and I'm actually going to filter out any pair that appears less than 12 times. I'm doing this because I've run through this a few times and this is a number that's arbitrary but it seemed to work well. So we want only word pairs that appear more than 12 times. And then we'll just print it here in the bottom. And here they're "Peleus's son", "Saturn's son", "Atreus son", "Achilles swift", "godlike son", "fair hair'd". Great, but now let's chart. Let's graph the pairs. This is the more interesting part. Now we use the same set, where we're going to filter things that have happened at least 12 times but we use this long function graph_from_data_frame. That's what gave us this data but now we're going to use ggraph and we get to choose the layout and we're using fr — there are about four different choices. This is the one that seems to be the most intelligible. We're going to tell it we want to see the links. We want to see the points or the nodes and we want to add some text so we know what's there and we're going to move it so it's not right on top of the point. And this can take a moment depending on your machine. But let's zoom in on this graph. And what you see here are "Paris", "bulls hide", "heav'n born", "lofty wall", "the Trojan host", but then you see other ones that appear more often. So for instance, here, we have "son" refers to this when we have "Priam's son" and "Saturn's son" and "Peleus's son". We have "noble Paris", "noble Hector" and we have the "fair hair'd Greeks", "clad brass". And this is a great way of visualizing some of the connections between the ideas. Now, this is going to work better the more data you have. I chose a long epic poem, "The Iliad." If you wanted to put "The Iliad"

and "The Odyssey" together, that would be great. If you wanted to take all of the books by one particular author then you can get an even richer display of the connections. But the word pairs, this is a great way of mining for meaning and association in your data using the text mining facilities that come with tidy text and graphing them out in a way that is easily accessible and a great way to get visual insight into your text data.



💬 **What is Word Pair Association Analysis?**

Word pair analysis (also called bigram analysis) is a text mining technique used to identify and visualize which words tend to appear together in a given body of text. This helps uncover common phrases, repeated ideas, or semantic relationships between terms.

🧠 **How It Works**

Instead of analyzing individual words (unigrams), this method examines pairs of adjacent words — known as bigrams or 2-grams.

Steps typically include:

1. **Tokenization**
Split the text into consecutive word pairs (e.g., "Trojan war", "noble Hector").

2. **Stop Word Filtering**
Remove common words like "the", "of", "and" that don't carry strong meaning.

3. **Counting**
Count how often each pair appears across the text.

4. **Filtering by Frequency**
Keep only frequently occurring pairs (e.g., those appearing >12 times).

5. **Visualization**
Create a network graph where:

- Nodes = individual words
- Edges = connections between co-occurring words
- Edge weight = frequency of the pair

📊 **What Does It Measure?**

- Commonly co-occurring word pairs
- Repeated patterns or motifs in language
- Thematic links or character relationships in narrative text
- Structural associations in documents (e.g., academic writing, news)

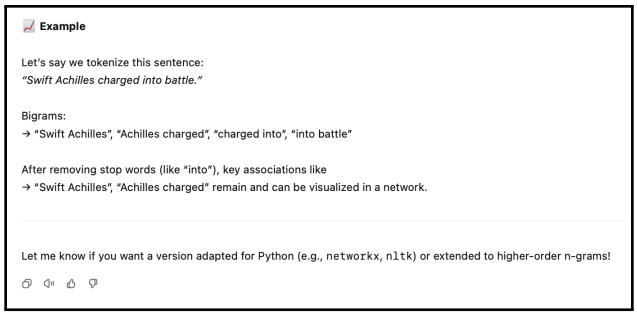
🔍 Use Cases

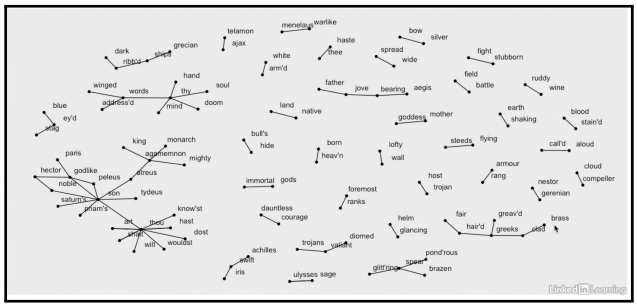
Word pair visualization is useful when:

- You want to discover recurring phrases or titles.
- You're analyzing narrative structure in literature (e.g., *The Iliad*).
- You need a network view of concept relationships in a corpus.
- You're preparing exploratory data analysis (EDA) for NLP tasks.

✅ Summary

Concept	Description
Token type	Bigrams (word pairs)
Stop word removal	Filters out uninformative connectors
Minimum frequency	Often set to a threshold (e.g., ≥12)
Output	Word pair frequencies or co-occurrence networks
Visualization tool	`ggraph`, `igraph`, or similar





Challenge: Sentiment scoring

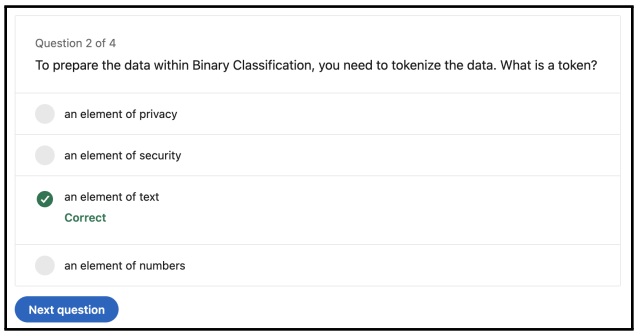
For our final challenge, I want to invite you to try graphing the narrative arc of a book using sentiment scoring. And to do this, I'm going to give you a different book to use. Instead of *The Iliad*, we'll use *Little Women* by Louisa May Alcott, also from Project Gutenberg. You can download it from this site, but I've already downloaded it and prepared the data a little bit and saved it in our data folder as `littlewomen.text`. So let me invite you to open up these packages and then come and import the data. And then let's take a look at the first few lines. This is just the title page. And so we have the Gutenberg ID, which indicates the book ID. And we see it's *Little Women* by Louisa May Alcott. But what I want you to do is take the entire text of the book, break it into sections, get the sentiment scoring, that goes from negative five to positive five, and get a visualization of the emotional arc of the story. Come back here in a moment and we'll compare results on this challenge.

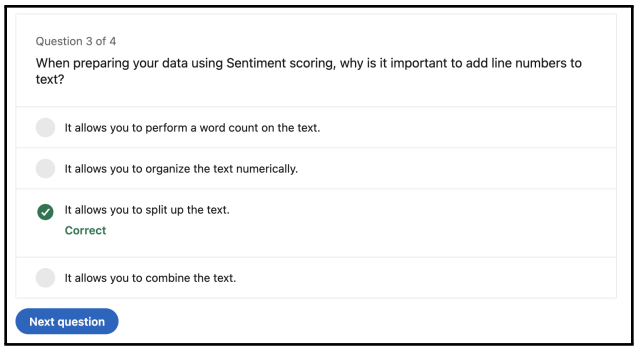
Solution: Sentiment scoring

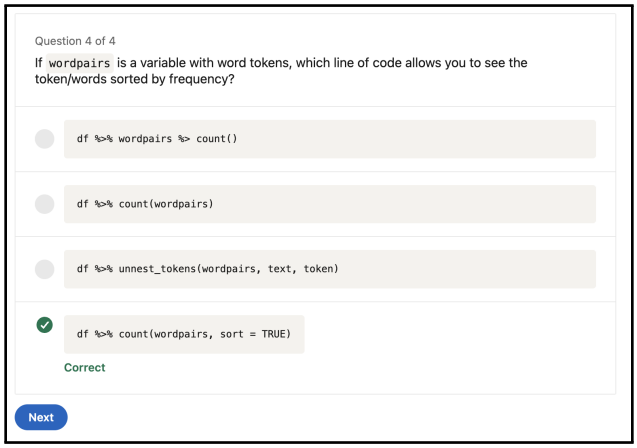
Hopefully, you've taken up the challenge to do a little bit of exploration of the text of *Little Women* and specifically to do sentiment scoring and find the emotional arc of the story. Let me show you how I did this one. I'm going to start by loading the packages and then coming down and importing the data, that is, the text of *Little Women*, and save it as a Tibble. We'll take a look at the first few lines and it's just the title page here. Let's prepare the data and I'm going to be using code that's very similar to what we used for *The Iliad*. I'm going to begin by adding line numbers because this is going to allow me to split up the text into lines and I'll remove the Gutenberg book ID number. We'll overwrite the data and now you can see *Little Women* and we have the first 10 lines there. And then we'll tokenize the data. We'll take the text and split it into words and we'll call that new variable `words`. And now you can see it right here. We have

the line number and we have the word. "Little" and "Women" are on the same line. What we can do now is we can calculate the sentiment frequencies. We're going to use the AFINN lexicon, which we used before, which scores emotionally-laden words from negative five, most negative, to positive five, most positive. And we do this by taking the words and then going to an inner join with the sentiment lexicon that says find all the words that have a value in this lexicon, group them by that value, summarize how many there are of each, print it and save it as `score freq` for frequency. And here you see that we have two words that are a negative five on sentiment. We've got a lot that are negative two, 2,785, but we have a huge number that are positive two and we go all the way up to positive five in this case. So let's graph the frequencies. This is just a bar chart of each of these frequencies. And what you see in this one... By the way, I had to change the limits to allow it to go up to positive five. Is that we have a peak here at negative two but then we've got a much stronger peak here. And this makes sense because *Little Women* is a happy story. It has some sad things that happen in it, but it's mostly an encouraging, wonderful story. But let's look at the sentiment arc. To do this, I'm going to use this method of splitting the text into sections of 100 lines. We'll get the sentiment words. We'll split them into groups of 100 lines using this operation. We'll calculate the average sentiment of each of those sections. And then we'll print it at the bottom to see what it's like. And we'll save that as `score arc`. And when we look at that, you can see that we start with a near zero score and then it goes up to over one and one and then we have about 200 rows in this. But the real payoff here is the graph, where we're going to take the data we just put it into `score arc`, and we're going to plot it by charting the score broken down by section. We'll put a red reference line in at zero. We'll put in a line that shows where the average score is and we'll smooth it out and put some titles on it. And when we do that, what you can see is that this is an overwhelmingly positive book, especially compared to *The Iliad* which we had before. The red line here is zero, which would be balanced positive and negative sentiment. We're well above that. It does vary substantially, but you can see that it becomes more positive as it goes through. And this gives us an idea of the trajectory of the story in *Little Women*. This is a very simple and effective way of evaluating the meaning in text, mining the evaluation, something that makes so easy to do in R and so easy to understand and share the results.

Chapter Quiz







Conclusion

Next steps

So we've come to the end of data mining with R. Now, where are you going to go? It's a little bit like you're getting ready for a trip. What's your actual plan? What's your map? Well, I want to make a few suggestions for both the immediate and the longer-term future. For the immediate, I recommend that you go through our online learning library and learn more about data science, both the concepts and the specific practices strongly associated with data mining, and learning more about data science will make you better at data mining tasks. In particular, you may want to place some special emphasis on machine learning algorithms. We covered several of those in this course—things like the neural networks. But you're going to want to look at more of these so you have a wide range of tools to help you when you're working on your data mining projects. And of course, you want to think about the actual programming tools that you use. Python and R are the two most common tools within data mining, and we have versions of this course for both Python and R. You finished watching one? I strongly recommend you take a look at the other so you have a more diverse range of opportunities ahead of you. And then finally, because data mining usually happens within an applied and frequently commercial organizational context, it's a good idea to go and learn more about how businesses are run, about the questions that are important to them, and how value can be derived from the insights that data mining gives you. Now, apart from that, I'm actually going to encourage you to get creative. We've been talking about a lot of technical details, but there's a lot of room for originality and for experimentation. Try different datasets. Try changing some of the commands. See what happens. Bring in something new. See what insights you can find through these iterations, through these variations. You may surprise yourself. If you want some good inspiration, I recommend that you go try meeting with user groups and meetup groups, either in person or online. The kind of social networking and the spontaneous influence you can get from other people can be an enormous help in finding new and creative ways to work with data and data mining. But most importantly, get your tools together, get going, and see where it takes you. You'll be surprised by what you find. Thanks so much for joining me, and best of luck in your data mining projects.