Summary for the Data Mining and Machine Learning oral exam (2024-2025)

University of Pisa
Department of Information Engineering
Artificial Intelligence and Data Engineering

Professor: Francesco Marcelloni
Assistant: Alessandro Renda
Assistant: José Luis Corcuera Bárcena
Assistant: Fabrizio Ruffini

Student: Francesco Panattoni

Contents

DISCLAIMER

*This document has been compiled as a comprehensive study aid in preparation for the oral examination of the **Data Mining and Machine Learning** course. The content herein is the result of an extensive review of the topics addressed during the lectures and a search for the most frequently asked questions in oral examinations.*

*While this summary aims to reflect the most important concepts of the course, I make no claims regarding the absolute completeness or accuracy of the information provided. These notes are intended to supplement, not replace, official textbooks and course materials. Furthermore, I **assume no responsibility** for the quality of the content or any results that may arise from using this document as a study resource.*

SOME TIPS

This document is intended strictly as a revisional instrument and should not be treated as a primary source for initial study. It is highly recommended that students engage with this material only after a comprehensive review of the official lecture slides and/or the recommended book (*"**Data Mining Concepts and Techniques**" by Jiawei Han, Micheline Kamber, and Jian Pei*).

The underlined questions indicate the most frequently asked topics, while also taking into account the general popularity of the topic. Even though they are underlined, the Data Stream questions are much less frequently asked than the Classification questions.

Observational evidence suggests that the examination often prioritizes Feature Selection, Statistical Testing, Classification, Clustering, Pattern Mining, and Outlier Detection. Typically, questions concerning Classification, Clustering, and Pattern Mining start with the technical details of a specific algorithm before transitioning into a rigorous discussion of evaluation and the metrics employed to determine the efficacy of the results.

While topics such as Data Streams are less commonly addressed, research into past examination sessions indicates they have been asked on occasion. Other remaining topics, though appearing with less regularity, should still be considered within the potential scope of the oral assessment, as the professor may exercise discretion in exploring any part of the syllabus.

While this document covers the full 12 CFU syllabus, a modified reading path is available for 6 CFU students. The following exclusions apply: chapters from 6 to 10 and sections 4.6, 4.7, 4.8, 5.2, and 5.3. Important: Sections 4.9 and 5.1 are required reading for 6 CFU Students.

Chapter 1: Data

1.1 Topic

1.1.1 What is Machine Learning? What is Data Mining?

Machine Learning is a computer's ability to learn from experience. It is a process that produces a model that can be used to make predictions or decisions from data. There's a paradigm shift: we don't write code to solve problems, but rather we write code to create models that we'll use to solve problems.

Data Mining is the practice of searching for useful information among abundant available data. It's not about extracting data, but about ***extracting knowledge from data***.

Machine Learning is a tool for Data Mining. We use Machine Learning techniques to extract information from data.

1.1.2 What are the mean, median and mode? When is the median preferred over the mean?

- **Mean:** arithmetic average $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$.
- **Weighted Mean:** $\bar{x}_p = \frac{\sum_{i=1}^{N} w_i \cdot x_i}{\sum_{i=1}^{N} w_i}$.
- **Median:** the middle value (or average of middle two) after sorting. If the number of values is even, the median is the mean of the two middle values. It is robust to extreme values and it is a holistic measure.
- **Mode:** most frequent value. The empirical formula is: $\text{mean} - \text{mode} = 3 \cdot (\text{mean} - \text{median})$.

Median is preferred when the distribution is skewed or contains outliers, because the mean can be strongly affected by extreme values.

1.1.3 Describe the five-number summary used in a boxplot and explain the usual rule for flagging outliers

The five-number summary is a set of five key descriptive statistics that summarize a dataset and are used to construct a boxplot. The five numbers are:

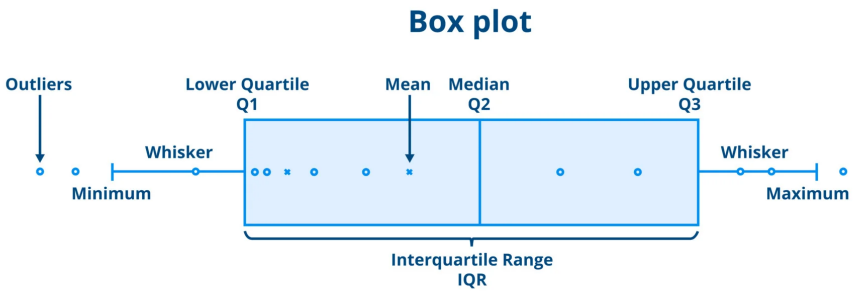
- **Minimum:** The smallest data value (excluding outliers);
- **First Quartile (Q1):** The value below which 25% of the data fall (the 25th percentile);
- **Median (Q2):** The middle value of the data (the 50th percentile);
- **Third Quartile (Q3):** The value below which 75% of the data fall (the 75th percentile);
- **Maximum:** The largest data value (excluding outliers).

A common rule for identifying outliers is based on the **Interquartile Range (IQR)**:

$$IQR = Q3 - Q1$$

Any points below $Q1 - 1.5 \cdot \text{IQR}$ or above $Q3 + 1.5 \cdot \text{IQR}$ are considered outliers.

1.1.4 What is a Boxplot?



A **Boxplot** is a graphical representation of the distribution of a dataset that highlights the median, quartiles, and potential outliers. It is widely used in exploratory data analysis to summarize numerical data.

Components of a Boxplot:

- **Median (Q2):** The middle value of the dataset.
- **First Quartile (Q1):** The 25th percentile, below which 25% of the data lie.
- **Third Quartile (Q3):** The 75th percentile, below which 75% of the data lie.
- **Interquartile Range (IQR):** Measures the spread of the middle 50% of the data.

$$\text{IQR} = Q3 - Q1$$

- **Whiskers:** Lines extending from the box to the smallest and largest values within $1.5 \times$ IQR from the quartiles.
- **Outliers:** Data points outside the whiskers, often plotted as individual points.

Construction Steps:

1. Compute Q1, Q2 (median), and Q3.
2. Calculate IQR: IQR $= Q3 - Q1$.
3. Determine whisker limits:

$$\text{Lower Whisker} = \max(\min(x), Q1 - 1.5 \times \text{IQR})$$

$$\text{Upper Whisker} = \min(\max(x), Q3 + 1.5 \times \text{IQR})$$

4. Plot the box from Q1 to Q3 with a line at the median. Draw whiskers to the limits and mark outliers.

Uses:

- Visualize central tendency, spread, and skewness of data;
- Detect outliers and extreme values;
- Compare distributions across multiple groups.

Positively skewed, where the mode occurs at a value that is smaller than the median or negatively skewed, where the mode occurs at a value greater than the median

1.1.5 What is Jaccard Similarity?

Jaccard Similarity is a proximity measure specifically used for asymmetric binary features.

To compute these measures, a contingency table is first constructed for two objects, i and j, across p binary variables:

	Object $j=1$	Object $j=0$	sum
Object $i=1$	q	r	$q+r$
Object $i=0$	s	t	$s+t$
sum	$q+s$	$r+t$	p

Where q is the number of variables where both are 1, t is the number of variables where both are 0, and r and s represent mismatches.

$$sim_{Jaccard}(i;j) = \frac{q}{q+r+s}$$

We can ignore t because the case with both i and j equal to 0 is uninteresting in asymmetric binary features.

1.1.6 Jaccard coefficient for asymmetric binary attributes: compute the Jaccard dissimilarity

Use the test attributes only (treat Gender as symmetric and ignore it for the Jaccard). The values (Y/P = 1, N = 0) are:

Name	Fever	Cough	Test-1	Test-2	Test-3	Test-4
Jack	1	0	1	0	0	0
Mary	1	0	1	0	1	0
Jim	1	1	0	0	0	0

Compute:

$$D(i;j) = 1 - \frac{q}{q+r+s}$$

where

- q = number of attributes where both i and j are 1;
- r = number of attributes where $i = 1$ and $j = 0$;
- s = number of attributes where $i = 0$ and $j = 1$.

Answer:

- **Jack vs Mary:**

  $$q = \#\{\text{both 1}\} = (\text{Fever}, \text{Test-1}) = 2, \quad r = 0, \quad s = 1 \text{ (Test-3 only)}.$$

  Jaccard similarity $= \frac{q}{q+r+s} = \frac{2}{3}$. Hence dissimilarity

  $$D(\text{Jack};\text{Mary}) = 1 - \frac{2}{3} = \frac{1}{3} \approx 0.333.$$

- **Jack vs Jim:**

  $$q = 1 \text{ (Fever)}, \quad r = 1 \text{ (Test-1: Jack 1, Jim 0)}, \quad s = 1 \text{ (Cough: Jack 0, Jim 1)}.$$

  Similarity $= \frac{1}{3}$, dissimilarity $= 1 - \frac{1}{3} = \frac{2}{3} \approx 0.667$.

- **Jim vs Mary:**

  $$q = 1 \text{ (Fever)}, \quad r = 1 \text{ (Cough: Jim 1, Mary 0)}, \quad s = 2 \text{ (Test-1 and Test-3: Jim 0, Mary 1)}.$$

  Similarity $= \frac{1}{1+1+2} = \frac{1}{4}$, dissimilarity $= 1 - \frac{1}{4} = \frac{3}{4} = 0.75$.

1.1.7 How are variance and standard deviation calculated?

Variance and **Standard Deviation** are measures of **dispersion**. *They quantify how much the data objects in a dataset spread out from the mean.*

- **Variance (σ^2):** The average of the squared differences from the mean.
- **Standard Deviation (σ):** The square root of the variance. It is expressed in the same units as the original data, making it easier to interpret.

$$\text{Variance: } \sigma^2 = \frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}$$

$$\text{Standard Deviation: } \sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}}$$

Where N is the population size and $\bar{x}$ is the population mean.

Use Variance when you need a measure for mathematical or scalable computations. Use Standard Deviation to describe the spread of data about the mean in the same units as your original data.

1.1.8 A dataset contains mixed attribute types (nominal, numeric, and binary). Describe a standard approach to compute a combined dissimilarity between two objects.

To compute the combined dissimilarity between two objects i and j in a dataset with mixed attribute types, a standard approach is to use a weighted formula that aggregates the effects of all p attributes. For numeric values, the dissimilarity is calculated as a normalized distance, typically using the absolute difference of the values divided by the total range of that attribute.

Use a feature-wise (possibly weighted) combination of per-attribute dissimilarities. For attribute f:

$$d_{ij}^{(f)} = \begin{cases} 0 & \text{if } x_{if} = x_{jf} \text{ (for nominal/binary)} \\ 1 & \text{if } x_{if} \neq x_{jf} \text{ (for nominal/binary)} \\ \frac{|x_{if} - x_{jf}|}{\text{max\_range}_f} & \text{for numeric (normalized distance)} \\ z_{if} = \frac{r_{if}-1}{M_f - 1} & \text{for ordinal} \end{cases}$$

(M_f is the maximum rank and r_{if} the actual rank).

Aggregate:

$$d(i;j) = \sum_{f=1}^{p} w_f \, d_{ij}^{(f)}, \qquad \sum_f w_f = 1$$

(choose weights w_f to reflect importance).

1.1.9 Difference between Similarity and Dissimilarity

- **Similarity** measures how alike two objects are. Higher values indicate greater resemblance.

  $$0 \leq s(i;j) \leq 1$$

  where $s(i;j) = 1$ means the objects are identical.

- **Dissimilarity** (or distance) measures how different two objects are. Lower values indicate greater similarity.

  $$0 \leq d(i;j) < \infty$$

  where $d(i;j) = 0$ means the objects are identical.

They are inversely related:

$$s(i;j) = 1 - d'(i;j)$$

where $d'(i;j)$ is a normalized dissimilarity in the range $[0, 1]$.

1.1.10 What is the difference between a data matrix and a dissimilarity matrix? When would you use the latter?

A *Data Matrix* (also called an *object–attribute matrix*) is a rectangular table in which rows represent data objects (or observations) and columns represent attributes (or features).

Formally:

$$X = [x_{ij}]_{n \times p}$$

where:

- n = number of data objects (rows);
- p = number of attributes (columns);
- x_{ij} = value of the j^{th} attribute for the i^{th} data object.

The data matrix represents the raw input used for most analytical tasks such as clustering, classification, and regression.

A *Dissimilarity Matrix* (or *distance matrix*) quantifies the pairwise dissimilarities or distances between all pairs of objects. Each entry measures how different two objects are.

Formally:

$$D = [d(i;j)]_{n \times n}$$

where:

1. $d(i;j) \geq 0$ is the dissimilarity between objects i and j;
2. $d(i;i) = 0$;
3. $d(i;j) = d(j;i)$ for symmetric measures (e.g., Euclidean distance).

In reality we use a Dissimilarity Matrix that is a Triangular Matrix (contrary to property 3).

Relationship Between the two:

- The data matrix contains raw feature values.
- The dissimilarity matrix is derived from the data matrix by applying a distance or similarity function (e.g., Euclidean, Manhattan, Jaccard, Cosine).

You use the dissimilarity matrix when the raw data are not directly comparable, but pairwise relationships are meaningful (e.g., genetic or text similarity).

You can use the dissimilarity matrix in algorithms that operate on distances rather than raw features, such as: Hierarchical clustering, Non-metric methods, or distance-based methods.

1.1.11 Explain Z-score standardization. Why might mean absolute deviation be used instead of standard deviation for scaling?

Z-score Standardization (also called standard score normalization) is a data transformation technique used to rescale numerical features so that they have a mean of zero and a standard deviation of one. It is useful for comparing variables measured on different scales or with different units.

For a variable x with mean $\bar{x}$ and standard deviation σ, the standardized value (z-score) of an observation x_i is given by:

$$z_i = \frac{x_i - \bar{x}}{\sigma}$$

This transformation centers the data around zero and expresses each value in terms of how many standard deviations it lies above or below the mean.

The resulting standardized variable has mean 0 and standard deviation 1. Z-score standardization is sensitive to outliers because both the mean and the standard deviation are affected by extreme values.

Instead of the standard deviation, one can use the **median absolute deviation** for a more robust scaling method:

$$z'_i = \frac{x_i - \text{median}(X)}{\text{median}(|x_i - \text{median}(X)|)}$$

1. The median absolute deviation is less sensitive to outliers than the standard deviation;
2. When data contain extreme values or are not normally distributed, using MAD provides a more stable and representative scaling;
3. It maintains relative relationships between observations while reducing the influence of outliers.

1.1.12 What is Minkowski distance?

The *Minkowski distance* is a generalized metric used to measure the distance between two points in an n-dimensional space. It generalizes several common distance measures such as Euclidean and Manhattan distances.

Definition: Given two points $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ and $\mathbf{u} = (u_1, u_2, \ldots, u_n)$ in n-dimensional space, the Minkowski distance of order h is defined as:

$$D_h(\mathbf{v}; \mathbf{u}) = \left(\sum_{i=1}^{n} |v_i - u_i|^h \right)^{\frac{1}{h}}$$

Special Cases:

- **Manhattan Distance ($h = 1$):**

$$D_1(\mathbf{v}; \mathbf{u}) = \sum_{i=1}^{n} |v_i - u_i|$$

  Measures the sum of absolute differences along each dimension.

- **Euclidean Distance ($h = 2$):**

$$D_2(\mathbf{v}; \mathbf{u}) = \sqrt{\sum_{i=1}^{n} (v_i - u_i)^2}$$

  Measures the straight-line (as-the-crow-flies) distance between points.

- **Chebyshev Distance ($h \to \infty$):**

$$D_\infty(\mathbf{v}; \mathbf{u}) = \max_i |v_i - u_i|$$

  Measures the maximum absolute difference along any dimension.

1.1.13 What is Cosine Similarity?

Cosine Similarity is a metric used to measure how similar two vectors are, regardless of their magnitude. It is widely used in data mining, information retrieval, and text analysis to quantify the similarity between objects represented in a high-dimensional space.

Given two vectors $\mathbf{v}$ and $\mathbf{u}$ in an n-dimensional space, cosine similarity is defined as:

$$\text{Cosine Similarity}(\mathbf{v}; \mathbf{u}) = \cos(\theta) = \frac{\mathbf{v} \cdot \mathbf{u}}{\|\mathbf{v}\| \|\mathbf{u}\|} = \frac{\sum_{i=1}^{n} v_i u_i}{\sqrt{\sum_{i=1}^{n} v_i^2} \sqrt{\sum_{i=1}^{n} u_i^2}}$$

where:

- $\mathbf{v} \cdot \mathbf{u}$ is the dot product of the two vectors;
- $\|\mathbf{v}\|$ and $\|\mathbf{u}\|$ are the Euclidean norms (magnitudes) of the vectors;
- θ is the angle between the two vectors.

The cosine similarity range is typically -1 to 1, where 1 means identical direction (highly similar), 0 means orthogonal (no similarity), and -1 means opposite directions (highly dissimilar). However, for non-negative data like term frequencies (TF-IDF) in text, the range is often 0 to 1, as vectors cannot be opposite.

Cosine similarity measures orientation, not size. This is useful when the magnitude of data (e.g., word frequency counts) varies widely but direction (pattern of occurrence) matters more. In Clustering algorithms (K-Means with cosine distance), cosine similarity helps group items with similar patterns of features.

Documents represented as TF–IDF or word frequency vectors can be compared using cosine similarity to find semantically similar texts.

1.1.14 Difference between Supervised and Unsupervised Learning

Supervised Learning refers to the use of labeled datasets in which each instance is associated with a known class label, enabling the system to learn under supervision. Classification is a supervised learning methodology.

Unsupervised Learning, on the other hand, deals with data that lacks predefined class labels, aiming instead to identify inherent groupings or clusters based on similarity among instances. Clustering is an unsupervised learning methodology.

1.1.15 Similarity Formulas in Binary attributes

	Object $j=1$	Object $j=0$	sum
Object $i=1$	q	r	$q+r$
Object $i=0$	s	t	$s+t$
sum	$q+s$	$r+t$	p

- **Symmetric Binary Distance:** The distance is the ratio of mismatches to the total number of variables:

$$d(i;j) = \frac{r+s}{q+r+s+t}$$

- **Asymmetric Binary Distance:** The distance ignores the unimportant 0-0 matches:

$$d(i;j) = \frac{r+s}{q+r+s}$$

Chapter 2: Data Preprocessing

2.1 Introduction

2.1.1 What is Data Preprocessing?

Data Preprocessing is the stage of preparing raw data for analysis or model training. It improves data quality and model performance. It is divided into:

1. **Data Cleaning**;
2. **Data Integration**;
3. **Data Reduction**;
4. **Data Transformation and Discretization**.

2.2 Data Cleaning

2.2.1 How to deal with missing data?

Data is not always available: many tuples have no recorded value for several attributes, such as customer income in sales data. **Missing data may need to be inferred.**

Therefore we can:

- **Ignore the tuple:** usually done when class label is missing (when doing classification). Not effective when the percentage of missing values per attribute varies considerably;
- **Fill in the missing value manually:** Not the best method. It could be tedious and infeasible;
- **Fill with a global constant:** Not very efficient;
- **Fill with the mean:** As the constant is not very efficient;
- **Fill with the attribute mean for all samples belonging to the same class:** The most commonly used since it is a good compromise between efficiency and complexity;
- **Fill with the most probable value:** inference-based such as K-Nearest Neighbors, Bayesian Formula or Decision Tree.

2.2.2 Talk about Smoothing Algorithms

Noise is random error or variance in a measured variable. Every dataset is effectively a combination of these two elements:

$$\text{Data} = \text{Signal} + \text{Noise}$$

- **Signal:** The underlying trend or relationship you are trying to discover (e.g., "As temperature increases, ice cream sales go up").
- **Noise:** Random fluctuations that don't follow that rule (e.g., a single person buying 50 tubs of ice cream in January, purely by chance for a party).

Think of it like listening to a radio station: the music you want to hear is the signal, while the static and crackling in the background is the noise. If the noise is too loud, you can't understand the music.

The noise can be reduced by a process called **Smoothing**.

In **Smoothing**, the data points of a signal are modified so that individual points that are higher than the immediately adjacent points (presumably because of noise) are reduced, and points that are lower than the adjacent points are increased.

We saw three smoothing algorithms.

Rectangular Smooth

The **Rectangular Smooth** (or **Unweighted Sliding-Average Smooth**) simply replaces each point in the signal with the average of m adjacent points, where m is a positive integer called the smooth width.

If $m = 3$:

$$S_j = \frac{Y_{j-1} + Y_j + Y_{j+1}}{3}$$

If the underlying signal is constant, or is changing linearly with time (increasing or decreasing), then no bias is introduced into the result.

Problems:

- A bias is introduced if the underlying function that represents the signal has a non-zero second derivative. At a local maximum, for example, moving window averaging always reduces the function value.
- Decide m. If m is too small, little smoothing is applied. If m is too large, the signal becomes flat.

Triangular Smooth

The **Triangular Smooth** replaces each point in the signal with the weighted average of m adjacent points. If $m = 5$:

$$S_j = \frac{Y_{j-2} + 2 \cdot Y_{j-1} + 3 \cdot Y_j + 2 \cdot Y_{j+1} + Y_{j+2}}{1 + 2 + 3 + 2 + 1} = \frac{Y_{j-2} + 2 \cdot Y_{j-1} + 3 \cdot Y_j + 2 \cdot Y_{j+1} + Y_{j+2}}{9}$$

The width of the smooth m is an odd integer and the smooth coefficients are symmetrically balanced around the central point. By giving more importance to the central value, it is better able to maintain the height of the peaks and the general shape of the original fluctuations in the data with respect to Rectangular Smoothing.

Problem: Decide m. If m is too small, little smoothing is applied, if m is too large, the signal becomes flat.

Savitzky-Golay Smoothing Filters

The **Savitzky-Golay Smoothing Filters** approximate the underlying signal function of the data within a moving window not by a constant (whose estimate is the average), but by a polynomial of higher order, typically quadratic or quartic. Unlike the moving average (rectangular and triangular smoothing), it does not distort the high frequencies of the signal (the peaks).

$$S_j = \sum_{n=-n_L}^{n_R} c_n \cdot Y_{j+n}$$

- n_L is the number of points used to the **left** of a data point;
- n_R is the number of points used to the **right** of a data point;
- c_n are **filter coefficients**. The sum of all these coefficients is always 1.

Typically, a filter like the moving average approximates the data with a constant: it takes the average of the neighboring points. In this case $c_n = \frac{1}{n_L + n_R + 1}$.

The Savitzky–Golay filter goes further: in each window, it fits a small polynomial (e.g., of degree 2 or 4) to the data, and takes the value of the polynomial in the center of the window as the filtered value (localized regression in a window). This preserves the shape of the curves better, because a polynomial can follow curvatures or slight variations. All these least-squares fits would be laborious if done as described. So the c_n weights can be computed just once using "dummy" data (a window of all zeros except one 1). Then, for real data, you simply apply these weights as a convolution.

M is the degree of the polynomial. With a small M the peaks are "squashed" (loss of amplitude) and the valleys are raised, with an high M the polynomial will start to follow the noise oscillations.

The Savitzky-Golay Smoothing Filters

M	n_L	n_R	Sample Savitzky-Golay Coefficients										
2	2	2				−0.086	0.343	0.486	0.343	−0.086			
2	3	1			−0.143	0.171	0.343	0.371	0.257				
2	4	0		0.086	−0.143	−0.086	0.257	0.886					
2	5	5	−0.084	0.021	0.103	0.161	0.196	0.207	0.196	0.161	0.103	0.021	−0.084
4	4	4		0.035	−0.128	0.070	0.315	0.417	0.315	0.070	−0.128	0.035	
4	5	5	0.042	−0.105	−0.023	0.140	0.280	0.333	0.280	0.140	−0.023	−0.105	0.042

The filter operates through a **localized least-squares regression** in a moving window. The step-by-step process is as follows:

1. **Window Selection:** Define a window of size N around the point j $(Y_{j-n_L}, \ldots, Y_j, \ldots, Y_{j+n_R})$.
2. **Polynomial Assumption:** Assume that the data within this window can be modeled by a polynomial $p(n)$ of degree M:
$$p(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_M n^M = \sum_{i=0}^{M} a_i \cdot n^i$$
where n represents the **relative position** from the center of the window $(n = -n_L, \ldots, 0, \ldots, n_R)$.
3. **Least-Squares Fitting:** Find the coefficients a_i that minimize the sum of squared errors between the polynomial and the actual data:
$$E = \sum_{n=-n_L}^{n_R} (p(n) - Y_{j+n})^2 \Rightarrow \min(E)$$
To find the minimum of the objective function, we need to cancel the partial derivatives with respect to each coefficient a_k:
$$\frac{\partial E}{\partial a_i} = 0 \quad \text{for each } i \in \{0, \ldots, M\} \Rightarrow \frac{\partial E}{\partial \mathbf{a}} = 2\mathbf{X}^T(\mathbf{X}\mathbf{a} - \mathbf{Y}) = 0 \Rightarrow \mathbf{X}^T\mathbf{X}\mathbf{a} - \mathbf{X}^T\mathbf{Y} = 0 \Rightarrow \mathbf{a} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}$$
To solve the system efficiently, we write it in matrix form $\mathbf{X}\mathbf{a} = \mathbf{Y}$. $\mathbf{X} \in \mathbb{R}^{(n_L+n_R+1)\times(M+1)}$:
$$\mathbf{X} = \begin{bmatrix} 1 & (-n_L) & (-n_L)^2 & \ldots & (-n_L)^M \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & n_R & n_R^2 & \ldots & n_R^M \end{bmatrix}$$
The optimal solution for the vector of coefficients $\mathbf{a}$ is obtained by solving:
$$\mathbf{a} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} = \mathbf{C}\mathbf{Y}$$
4. **Central Evaluation:** Once the optimal polynomial is found, the smoothed value S_j is the value of $p(n)$ at the center of the window $(n = 0)$. Since $p(0) = a_0$, we have:
$$S_j = a_0$$
5. **Convolutional Equivalence:** In Savitzky-Golay, the relative coordinates (n_L and n_R) are fixed. Therefore, the matrix $\mathbf{X}$ is constant for each window. This means that the matrix $C = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ (called the pseudoinverse) can only be computed once. Since we're only interested in the clean value of the central point $(S_j = a_0)$, we only need the first row of C. Those values are exactly the convolution coefficients c_n we were talking about earlier. This means a_0 can be pre-calculated as a weighted sum of the data:
$$S_j = \sum_{n=-n_L}^{n_R} c_n \cdot Y_{j+n}$$

The larger the smooth width, the greater the noise reduction, but also the greater the possibility that the signal will be distorted by the smoothing operation.

2.2.3 When should you smooth a signal?

You should do this:

- For cosmetic reasons, to prepare a nicer-looking graphic of a signal for visual inspection or publication;
- If the signal will be subsequently processed by an algorithm that would be adversely affected by the presence of too much high-frequency noise in the signal.

You should NOT smooth prior to statistical procedures such as least-squares curve fitting, because smoothing will not significantly improve the accuracy, all smoothing algorithms are at least slightly "lossy" and smoothing the signal will seriously underestimate the parameter errors predicted by propagation-of-error.

2.2.4 How do you reduce noise from a signal?

- **Smoothing:** Replace each point with a weighted average of its neighbors;
- **Binning:** The data is divided into intervals ("bins") and each bin is represented with a summary value:
  1. Frequency binning: first sort data and partition into (equal-frequency) bins;
  2. Then one can smooth by bin means, smooth by bin median, smooth by bin boundaries, ...
- **Regression:** A function (linear or non-linear) is fitted that approximates the data. Noise is reduced because the function follows only the general trend, not random fluctuations;
- **Clustering:** Similar data is grouped into clusters and outliers are removed;
- **Combining computer and human inspection:** The machine performs automatic cleaning or filtering methods, while the human controls, corrects, or guides the thresholds and decisions.

2.3 Data Integration

2.3.1 What is Data Integration?

Data Integration is the process of combining data from different sources into a single, coherent, compatible, and usable whole.

2.3.2 Correlation between numeric attributes

Correlation measures the linear relationship between two numerical variables A and B. **Pearson**'s formula is used to calculate it and it varies between -1 and +1.

$$r = \frac{\sum_{i=1}^{n}(a_i - \bar{a}) \cdot (b_i - \bar{b})}{n \cdot \sigma_A \cdot \sigma_B}$$

where:

- n is the number of tuples;
- a_i and b_i are respectively the values of the attributes A and B;
- $\bar{a}$ and $\bar{b}$ are respectively the means of the attributes A and B;
- σ_A and σ_B are respectively the standard deviations of the attributes A and B.

If:

- $r > 0$ they are positively correlated;
- $r = 1$ is the perfect positive linear relationship (both grow together);
- $r = 0$ no linear correlation (which does not necessarily mean independence);
- $r < 0$ they are negatively correlated;
- $r = -1$ the linear relationship is perfect negative (one increases, the other decreases).

Covariance is a statistical measure that quantifies how much two numerical variables vary together about their means:

$$\text{COV(A;B)} = \frac{\sum_{i=1}^{n}(a_i - \bar{a}) \cdot (b_i - \bar{b})}{n}$$

So:

$$r = \frac{\text{COV(A;B)}}{\sigma_A \cdot \sigma_B}$$

Standard Deviation σ_X is a measure of the dispersion or variability of a set of numerical data from their mean.

$$\sigma_X = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

The smaller the standard deviation the closer the data is to the mean, the larger the standard deviation the more sparse the data is.

Correlation is measured when you want to compare the strength and direction of the relationship between variables with different units or different datasets. Correlation is preferable when you need to compare attributes with different units of measurement (e.g., age vs. salary), since standardization eliminates the influence of the magnitude of the values, otherwise covariance is better.

2.3.3 What is Chi-Square?

The **Chi-Square test** (denoted as χ^2) is a non-parametric statistical method used to evaluate whether two **nominal** variables are independent or exhibit a relationship. It also measures how much two nominal values are correlated. If two attributes are highly correlated, one adds little new information and can be removed to reduce dimensionality. For classification, use mutual information, otherwise, consider NULL values and interpretability.

The test statistic is defined as:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

where O_i represents the observed frequencies and E_i the expected frequencies.

Let:

$$A = \{a_1, \ldots, a_c\} \qquad B = \{b_1, \ldots, b_r\}$$

where A and B denote the categories of two variables. The data can be organized in a **contingency table** as illustrated in the next image.

	b_1	b_2	...	b_r	
a_1	$o_{1,1}$	$o_{1,2}$		$o_{1,c}$	$o_{1,.}$
a_2	$o_{2,1}$	$o_{2,2}$		$o_{2,c}$	$o_{2,.}$
...					
a_c	$o_{r,1}$	$o_{r,2}$		$o_{r,c}$	$o_{r,.}$
	$o_{.,1}$	$o_{.,2}$		$o_{.,r}$	Tot

For the general case, the Chi-Square statistic is computed as:

$$\chi^2 = \sum_{i=1}^{r}\sum_{j=1}^{c}\frac{(o_{ij} - e_{ij})^2}{e_{ij}},$$

where:

$$e_{ij} = \frac{\text{count}(A = a_i) \cdot \text{count}(B = b_j)}{n}.$$

n is the sum of all the values in the table, the size of the sample analyzed.

The expected frequency e_{ij} represents the number of instances predicted for the combination $(a_i; b_j)$ under the assumption that A and B are independent. A large χ^2 value suggests dependence between the variables, while a value close to zero indicates independence.

Hypothesis testing in the Chi-Square framework is formulated as follows:

Hypothesis	Description
H_0 (Null Hypothesis)	There is no significant relationship between the variables.
H_1 (Alternative Hypothesis)	There is a significant relationship between the variables.

The **degree of freedom** (df) corresponds to the number of independent cells in the contingency table:

$$\text{df} = (r-1)\cdot(c-1).$$

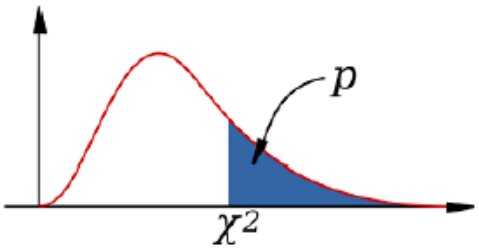
Intuitively, it represents the number of values in a final calculation that are "free to vary".

The X^2 distribution is a type of probability distribution.

Probability distributions provide the probability of every possible value that may occur.

Distributions that are cumulative **give the probability of a random variable being less than or equal to a particular value.**

To find the probability of a particular value, we find the area under the curve before the value. The area that's after the value is called the **p-value**



The **p-value** is the probability of obtaining a result equal to or more extreme than the observed one, assuming the null hypothesis is true. Simply put, it's a measure of how "odd" or "rare" your data is if we assume there's no relationship between the variables. A high p-value confirms H_0 (independence), while a low p-value (usually < 0.05) rejects H_0 (there is a relationship).

The χ^2 distribution depends on the degrees of freedom and is always right-skewed. In hypothesis testing, critical values are often obtained from a Chi-Square table, where each row corresponds to a degree of freedom and columns represent p-values. The null hypothesis H_0 is rejected when the probability of observing a larger χ^2 value is smaller than the chosen significance level α.

Percentage Points of the Chi-Square Distribution

Degrees of Freedom	Probability of a larger value of x^2								
	0.99	0.95	0.90	0.75	0.50	0.25	0.10	0.05	0.01
1	0.000	0.004	0.016	0.102	0.455	1.32	2.71	3.84	6.63
2	0.020	0.103	0.211	0.575	1.386	2.77	4.61	5.99	9.21
3	0.115	0.352	0.584	1.212	2.366	4.11	6.25	7.81	11.34
4	0.297	0.711	1.064	1.923	3.357	5.39	7.78	9.49	13.28
5	0.554	1.145	1.610	2.675	4.351	6.63	9.24	11.07	15.09

In practice:

1. Compute Degree of Freedom;
2. Determine by what percentage I want to reject the null hypothesis;
3. Compute Chi-Square;
4. If the calculated value is greater than the value in the table, then the two nominal variables are correlated, otherwise they are independent. We're talking about correlation, not causation.

2.3.4 Differences between Parametric and Non-Parametric tests

Hypothesis testing in tests is formulated as follows:

Hypothesis	Description
H_0 (Null Hypothesis)	There is no significant relationship between the variables.
H_1 (Alternative Hypothesis)	There is a significant relationship between the variables.

In **Parametric tests**, hypothesis testing is based on the assumption that observed data are distributed according to some distributions of well known form (normal, Bernoulli, and so on) up to some unknown parameter on which we want to make inference (say the mean, or the success probability). Inference is the process of generalizing results from a sample to the total population, often supported by parametric or non-parametric tests.

In **Non-Parametric** tests, in the hypothesis test it is not necessary (or not possible) to specify the parametric form of the distribution(s) of the underlying population(s).

2.4 Data Reduction

2.4.1 Talk about the Curse of Dimensionality

The **Curse of Dimensionality** refers to a set of phenomena that arise when analyzing and organizing data in high-dimensional spaces, often leading to challenges in ***Data Mining and Machine Learning***. As the number of dimensions (features) increases, many common data analysis techniques become less effective.

Some Key Aspects:

1. **Sparsity of Data:** In high-dimensional spaces, data points become increasingly sparse. The volume of the space increases exponentially with the number of dimensions, so even large datasets occupy a tiny fraction of the total space. This sparsity makes it difficult to find meaningful patterns or neighborhoods.

2. **Distance Concentration:** Distances between points in high-dimensional spaces tend to converge, reducing the contrast between nearest and farthest neighbors. Mathematically, for a dataset with n dimensions, the ratio of the difference between the maximum and minimum distances to the minimum distance approaches zero as n increases:
$$\frac{\max(d) - \min(d)}{\min(d)} \to 0 \quad \text{as } n \to \infty$$
This diminishes the effectiveness of distance-based methods such as k-NN, Clustering, and nearest neighbor searches.

3. **Exponential Increase in Sample Size Requirement:** To maintain a given density of points in high-dimensional space, the number of required samples grows exponentially with the number of dimensions. This makes statistical estimation and Machine Learning models prone to overfitting if the dataset is insufficient.

4. **Feature Irrelevance:** In high dimensions, irrelevant or noisy features can dominate distance metrics and statistical measures, making it harder to detect true patterns. Feature selection or dimensionality reduction becomes crucial.

5. **Visualization Difficulty:** Visualizing and interpreting data becomes challenging as humans can only directly perceive 2 or 3 dimensions, limiting intuitive understanding of high-dimensional structures.

The goal is to extract meaningful patterns, relationships, and predictive models from data. However, when dimensionality is high:

- The density of data points becomes insufficient to support statistical inference.
- Algorithms relying on proximity or similarity metrics (e.g., Clustering, Nearest Neighbors, kernel methods) fail to perform effectively.
- The generalization ability of models deteriorates due to overfitting and noise amplification.

Therefore, it is crucial to employ **dimensionality reduction techniques** to mitigate these effects, such as **Feature Reduction** (*Principal Component Analysis (PCA)*, *Feature Selection*, and *Autoencoders*), **Numerosity Reduction**, and **Data Compression**. Reducing dimensionality enhances model interpretability, decreases computational cost, and improves learning performance by focusing on the most informative features.

2.4.2 What is Principal Component Analysis?

Principal Component Analysis (**PCA**) is a technique for **dimensionality reduction**. **It finds a projection that captures the largest amount of variance in the data.** This is because greater variance could mean a greater amount of information. The original data are projected onto a smaller (or equal) space, resulting in a reduced representation.

PCA identifies the **eigenvectors** of the covariance matrix. These eigenvectors define new orthogonal axes (principal components). The projection error is minimized, while the variance in the new space is maximized. *It works only with numerical features.*

Step 1: Adjusted Dataset Given N data vectors from f dimensions, find $k \leq f$ orthogonal vectors (principal components) that best represent the data. D has f rows and N columns (I don't understand why the professor uses a transposed dataset as the basis without explicitly stating it, but I'll accept it.).

- Normalize the input data so that each attribute falls within the same range;
- Compute the mean of each dimension:

$$M_i = \frac{1}{N} \sum_{j=1}^{N} D_{ij}$$

- Subtract the mean from each data value to obtain the **adjusted dataset**:

$$A_{ij} = D_{ij} - M_i \quad \forall i \; \forall j$$

Step 2: Covariance Matrix Compute the covariance matrix C (f rows and f columns) from the adjusted dataset A. Since the means of A are 0, the covariance matrix can be written as:

$$C = \frac{AA^T}{N-1}$$

where $C_{ij} = \text{COV}(i; j)$ measures how dimensions i and j vary together.

Step 3: Eigenvectors and Eigenvalues Compute the eigenvectors and eigenvalues of C and find E (f rows and k columns). The eigenvectors thus obtained define the new space (or new basis) onto which the data will be projected.

- Eigenvectors define the directions of the new axes;
- Eigenvalues measure the amount of variance captured by each axis;
- Eigenvectors corresponding to small or zero eigenvalues can be discarded.

Step 4: Data Transformation Transform the adjusted dataset into the new basis:

$$F = E^T A$$

where:

- F is the transformed dataset (in the new reduced space);
- E^T is the transpose of the matrix of eigenvectors;
- A is the adjusted dataset.

The new dataset F has fewer (or equal) dimensions than A (F has k rows and N columns). Since E is orthogonal ($E^{-1} = E^T$), the original data can be recovered approximately as:

$$A = EF$$

2.4.3 Talk about techniques of Data Reduction

Data Reduction is the process of obtaining a compact representation of a dataset that preserves its essential analytical properties. In other words, it aims to generate a smaller version of the data that yields results comparable to those obtained from the full dataset. This process becomes particularly important when dealing with massive databases or data warehouses, where storage and computational efficiency are critical concerns.

The primary objectives of data reduction are threefold:

- Reduce storage and computational costs;
- Maintain data integrity and analytical usefulness;
- Improve performance of mining algorithms.

Data reduction can be achieved through several complementary strategies, namely **dimensionality reduction**, **numerosity reduction**, and **data compression**.

Dimensionality reduction focuses on eliminating irrelevant or redundant attributes in order to simplify data representation. This approach directly addresses the *Curse of Dimensionality*, a phenomenon where increasing the number of features leads to sparsity in data and weakens the significance of distances among points. Common techniques used in dimensionality reduction include the *Wavelet Transform* which decomposes data into wavelet coefficients for multi-resolution analysis, *Principal Component Analysis (PCA)* which transforms data into orthogonal components that capture maximum variance, and *Feature Selection*, which employs statistical or heuristic methods to retain only the most informative variables. The benefits of these methods are numerous. They reduce noise, facilitate data visualization, and significantly lower computational time and space complexity.

Numerosity reduction aims to reduce the volume of data by representing it through smaller or more compact forms. This can be achieved using two general approaches: parametric and non-parametric methods. Parametric methods assume a specific model structure and store only the parameters of that model. Examples include linear and multiple regression models. Non-parametric methods, on the other hand, do not rely on model assumptions and instead summarize the data through techniques such as histograms, Clustering, or sampling. Both approaches seek to preserve the core informational content of the dataset while discarding redundant or repetitive details.

Data compression represents another important form of data reduction. Its purpose is to encode data more efficiently, thereby reducing the amount of memory or storage required. Compression techniques can be broadly divided into two categories: *lossless* and *lossy*. Lossless compression preserves the original data exactly, making it suitable for applications such as textual or categorical data storage. Lossy compression, in contrast, allows for some approximation and is typically used for complex data types like audio, video, or time-series signals. In practice, both dimensionality and numerosity reduction can be viewed as specialized forms of compression, as they aim to store data in a more concise form without substantially sacrificing analytical accuracy.

2.4.4 Explain the Heuristic Search in Attribute Selection

Heuristic Search is a **greedy approach** *used to identify a near-optimal subset of features when an exhaustive search is computationally infeasible.* Instead of evaluating every possible combination, heuristic methods make the best local choice available at each step.

We need heuristic search because a dataset with d attributes (features) contains 2^d possible attribute combinations. As dimensionality increases, exhaustive search becomes infeasible. Heuristic methods reduce dimensionality and help avoid the **Curse of Dimensionality**. They require ground truth, since they are supervised, but do not transform the data.

Typical heuristic attribute selection methods include:

- **Step-wise Forward Selection**: This approach starts with an empty set and adds the best single attribute at each step until a stopping criterion is met;
- **Step-wise Backward Elimination**: This method begins with the full set of attributes and repeatedly eliminates the least useful attribute at each step;
- **Optimal branch and bound**: This strategy uses attribute elimination combined with backtracking to find the best subset.

The selection of attributes is often guided by **information theory metrics**, particularly when using **Mutual Information** (**MI**):

- **Entropy (Information)**: Measures the uncertainty or impurity of a distribution D. It is higher when the distribution is closer to being uniform. Given D over a finite set, where $< p_1, p_2, \ldots, p_n >$ are the corresponding probabilities, define the Entropy of D by:

$$H(D) = -\sum_i p_i \log_2 p_i$$

- **Mutual Information (MI)**: Quantifies the relationship between two discrete variables, such as an attribute X and a class Y, based on how often they occur together compared to how often they occur on their own.

$$MI(X;Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x;y) \cdot \log \frac{p(x;y)}{p(x)p(y)}$$

  With $\mathcal{X}$ and $\mathcal{Y}$ being respectively the alphabets (all the possible values) of X and Y. Mutual Information can be seen as the reduction of uncertainty (Entropy) of one variable resulting from knowledge of the other. Mutual Information detects any type of relationship, even non-linear ones, unlike correlation;

- **Normalized Mutual Information**: A normalized version of MI that restricts values to the range $[0; 1]$ to compensate for MI's bias toward features with many values.

$$NI(f_i; f_s) = \frac{MI(f_i; f_s)}{\min\{H(f_i); H(f_s)\}}$$

- **Information Gain**: A selection criterion used to maximize the information a candidate feature f_i provides about the class C, while subtracting a penalty for redundancy with the already selected set S.

$$\text{Gain} = MI(C; f_i) - \frac{1}{|S|} \sum_{f_s \in S} NI(f_i; f_s)$$

Mutual Information works only on features that can assume a finite defined set of values, so you have to define a categorical set of values.

The heuristic algorithm (Normalized Mutual Information Feature Selection) follows these procedural steps:

1. **Initialization**: Define F as the initial set of N features and S as an empty set;
2. **Calculate Initial MI**: Compute the MI between each feature and the class labels, $MI(f_i; C)$, for all $f_i \in F$;
3. **First Selection**: Select the feature $\hat{f}_i$ that maximizes $MI(f_i; C)$, then remove it from F and add it to S;
4. **Greedy Selection**: Repeat the following until the selected set S reaches the desired size k:
   (a) Calculate the $MI(f_i; f_s)$ between candidate features $f_i \in F$ and selected features $f_s \in S$;
   (b) Identify the feature $\hat{f}_i \in F$ that maximizes the Information Gain: Gain $= MI(C; f_i) - \frac{1}{|S|} \sum_{f_s \in S} NI(f_i; f_s)$;
   (c) Update the sets by moving the best feature from F to S.
5. **Output**: Return the set S containing the selected attributes.

When $MI(X;Y)$ is low what does it mean? That the attributes X and Y are independent.

Why is the Mutual Information zero when two features independent? The probability $P(x; y)$ is equal to $P(x) \cdot P(y)$ (because Independence is not Mutual Exclusion), then the logarithm $\log \frac{p(x;y)}{p(x)p(y)}$ gives 0.

How can we determine the best value for k? You can test many values of k and understand which gives the best result.

Do you think that we can use Normalized Mutual Information approach to select features only for Classification problems? Since ground truth is needed, we cannot use it for Clustering. Another approach that is good for Clustering or in general for when you do not have ground truth available is Chi-square test.

2.5 Data Transformation and Discretization

2.5.1 What are the Data Normalization techniques?

Data Normalization is a preprocessing technique used to rescale numerical features into a standard range or distribution. It ensures that all variables contribute equally to the analysis, preventing attributes with larger numeric ranges from dominating those with smaller ranges.

There are several common normalization methods, including **Min–Max Normalization**, **Z–Score Normalization**, and **Robust Normalization**.

Min–Max Normalization

Min–Max Normalization, also known as *rescaling*, transforms the original data into a fixed range, typically $[0; 1]$ or $[-1; 1]$. This method linearly scales each feature based on its minimum and maximum values. It is defined as:

$$x'_i = \frac{x_i - \min(x)}{\max(x) - \min(x)} \cdot (b - a) + a$$

where:

- x_i is the original value of the feature,
- $\min(x)$ and $\max(x)$ are the minimum and maximum values of the feature, respectively,
- $[a; b]$ is the desired range for normalization, commonly $[0; 1]$,
- x'_i is the normalized value.

This method preserves the relationships between the original data points but is sensitive to outliers, since extreme values directly affect the scaling range.

Z–Score Normalization

Z–Score Normalization, also known as *standardization*, rescales data based on the mean and standard deviation of the feature. It centers the data around zero with a standard deviation of one, as shown below:

$$x'_i = \frac{x_i - \mu}{\sigma}$$

where:

- μ is the mean of the feature values,
- σ is the standard deviation,
- x'_i represents the standardized value.

This normalization method is particularly useful when data follow (or approximately follow) a Gaussian distribution. It is less sensitive to scaling issues compared to Min–Max normalization but can still be influenced by extreme outliers, which distort the mean and variance.

Robust Normalization

Robust Normalization addresses the sensitivity to outliers by using robust statistical measures (namely, the median and the interquartile range, IQR) instead of the mean and standard deviation. The transformation is expressed as:

$$x'_i = \frac{x_i - \text{median}(x)}{\text{IQR}(x)}$$

where:

- $\text{median}(x)$ is the median of the feature values,
- $\text{IQR}(x) = Q_3 - Q_1$ is the interquartile range, with Q_1 and Q_3 representing the first and third quartiles, respectively.

By scaling data relative to the median and IQR, this approach effectively reduces the influence of outliers, making it ideal for datasets that contain skewed distributions or extreme values.

2.5.2 What are the Data Discretization Techniques?

Data Discretization is the process of transforming continuous numerical attributes into a finite number of intervals or categories. Instead of maintaining raw numeric values, data are replaced by interval labels that represent meaningful ranges. This transformation reduces data size and complexity while enabling efficient analysis, especially for Classification and Pattern Mining tasks that operate at multiple levels of abstraction.

In general, attributes in a dataset can be classified into three main categories:

- **Nominal:** values from an unordered set (e.g., color, profession);
- **Ordinal:** values from an ordered set (e.g., rank);
- **Numeric:** real numbers (e.g., income, age).

The main goals of data discretization are to reduce the volume of data, to improve the interpretability of patterns and models, and to prepare continuous attributes for algorithms that require discrete or categorical inputs. Many data mining algorithms (particularly decision trees, association rule mining, and Bayesian classifiers) operate more effectively on discrete data than on continuous attributes.

Discretization techniques can be broadly categorized based on their use of class information and the direction in which intervals are constructed. In **supervised discretization**, class labels guide the partitioning process. Examples include decision-tree-based and Chi-square-based methods, which create intervals that maximize the separation of classes. In contrast, **unsupervised discretization** does not rely on class information and instead groups values based on data distribution, as seen in binning or Clustering methods. Depending on the direction of processing, techniques may also be classified as **top-down (splitting)**, where large intervals are recursively divided, or **bottom-up (merging)**, where small intervals are successively combined based on similarity criteria.

Among the most widely used discretization methods is **Binning**, in which data values are first sorted and then partitioned into bins or intervals. Two major binning strategies are common. In **equal-width binning**, the entire range of values, from a minimum A to a maximum B, is divided into N intervals of equal size according to:

$$W = \frac{B - A}{N}$$

In **equal-depth binning**, the range is partitioned so that each interval contains approximately the same number of data points. After partitioning, smoothing operations may be applied to reduce noise, such as replacing values within each bin by the bin mean, median, or boundary values.

Another important technique is **histogram analysis**, where the data are divided into a set of buckets, and summary statistics, such as the mean or frequency count, are stored for each bucket. This provides a compact representation of data distributions and supports efficient query processing and mining.

A more adaptive approach is **Clustering-based discretization**, which groups similar data values into clusters using similarity measures. Each cluster then corresponds to one discrete interval. This method captures natural groupings in the data and is particularly useful when data do not follow uniform distributions.

In **decision tree analysis**, discretization is performed as part of the model construction process. The method is supervised and uses entropy-based measures (such as information gain) to determine the best split points for continuous attributes. The data are recursively partitioned according to class information, producing intervals that optimize class separability.

Finally, **correlation analysis**, exemplified by the *ChiMerge* algorithm, performs discretization by statistically analyzing the relationship between intervals and class labels. Using the Chi-square (χ^2) test, ChiMerge identifies and merges adjacent intervals whose class distributions are not significantly different, that is, those with low χ^2 values. The process iteratively merges intervals until no further merges satisfy the criterion, yielding a compact and class-relevant discretization.

A related concept in data reduction and abstraction is **concept hierarchy generation**, where low-level numeric or categorical values are replaced by higher-level categories. For example, the attribute *age* may be discretized into conceptual categories such as *youth*, *adult*, and *senior*. Such hierarchies facilitate multilevel data mining, allowing analyses to be conducted at varying levels of generalization. Concept hierarchies can be defined manually by domain experts or automatically derived based on the number of distinct attribute values or their semantic relationships.

Chapter 3: Classification

3.1 Introduction

3.1.1 How does Classification work?

The process of **Classification** is described as a two-step procedure.

1. **Model Construction** involves building a classifier that captures the relationships between input attributes and the target class label. Each instance in the training set is represented as a vector of attribute values, and the resulting classifier can take the form of rules, decision trees, or mathematical equations.
2. **Model Usage** involves applying the trained model to new data in order to predict their class labels. The quality of the classifier is evaluated by comparing its predicted outputs with the true class labels of a test set. The resulting accuracy rate, defined as the percentage of correctly classified instances, serves as a key indicator of model performance. To prevent overfitting, it is essential that the test set remains independent from the training data.

3.1.2 What is the difference between Classification and Regression?

Classification predicts discrete categories. Regression predicts continuous values.
In short: Classification = labels, Regression = numbers.

3.1.3 What are the evaluation metrics for Classification problems?

Machine Learning models are frequently applied to predict outcomes in Classification problems. Since predictive models rarely achieve perfect accuracy, several performance metrics are used to assess their effectiveness.

When predicting two classes (positive and negative), model outcomes can be categorized as:

- **True Positives (TP):** instances belonging to the positive class that are correctly predicted as positive;
- **True Negatives (TN):** instances belonging to the negative class that are correctly predicted as negative;
- **False Positives (FP):** instances belonging to the negative class that are incorrectly predicted as positive;
- **False Negatives (FN):** instances belonging to the positive class that are incorrectly predicted as negative.

Based on these four quantities, several key metrics can be defined:

Accuracy

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Measures the overall proportion of correctly classified instances.

Error Rate

$$\text{Error Rate} = \frac{FP + FN}{TP + TN + FP + FN} = 1 - \text{Accuracy}$$

Measures the overall proportion of incorrectly classified instances.

Precision

$$\text{Precision} = \frac{TP}{TP + FP}$$

Indicates how many predicted positives are actually positive.

Recall (Sensitivity or True Positive Rate)

$$\text{Recall} = \frac{TP}{TP + FN}$$

Measures the ability of the model to identify positive instances.

F Score

$$F1 \text{ Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Represents the harmonic mean of precision and recall, balancing the two metrics. We use the harmonic mean rather than the arithmetic mean because the denominators differ for Precision and Recall.

It can be a weighted average. It allows you to assign β times more importance (or less if less than 1) to Recall than to Precision:

$$F_\beta \text{ Score} = \frac{(1 + \beta^2) \times \text{Precision} \times \text{Recall}}{(\beta^2 \times \text{Precision}) + \text{Recall}}$$

False Positive Rate (FPR)

$$\text{FPR} = \frac{FP}{FP + TN}$$

Indicates the proportion of negative instances incorrectly classified as positive.

ROC AUC (Area Under the Receiver Operating Characteristic Curve)

$$\text{ROC AUC} = \int_0^1 Recall(FPR)\ d(FPR) = \int_0^1 TPR(FPR)\ d(FPR)$$

The **Receiver Operating Characteristic (ROC)** curve is a graphical representation of a Classification model's performance across different decision thresholds. It is a probability curve that plots the **Recall** (*True Positive Rate*) against the **False Positive Rate (FPR)**.

How to read the curve:

- **The Bottom-Left (0; 0):** The model predicts "Negative" for everything. You have zero false alarms, but you also catch zero positives.
- **The Top-Right (1; 1):** The model predicts "Positive" for everything. You catch every positive, but your false alarm rate is 100%.
- **The "Perfect" Model:** A curve that rushes to the top-left corner (0; 1). This means the model achieved 100% Recall with 0% False Positives.
- **The Diagonal Line:** This represents a model that is essentially guessing randomly. If your curve is on this line, your model is no better than a coin flip.

The **Area Under the ROC Curve (AUC)** provides a single scalar value summarizing the performance of the model. It can be interpreted as the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one.

Interpretation of AUC values:

- 0.5: No discriminative power (random guessing);
- $0.6 - 0.7$: Poor discrimination;
- $0.7 - 0.8$: Acceptable discrimination;
- $0.8 - 0.9$: Excellent discrimination;
- > 0.9: Outstanding discrimination.

Usually in TPR we have just one point in the ROC curve and not a curve. How do we compute the AUC?

Most Machine Learning models actually output a probability score before they convert it into a label. We get a curve instead of a single point by sliding the "cutoff" threshold across every possible value, connecting the point of the model in the graph from (0; 0) to (1; 1).

How to compare two classifiers with the help of ROC?

The simplest way to compare two models is to look at their AUC values. If one classifier has a better AUC it is generally better than the other. But this can be misleading. The relative height of the curves at low FPR reflects performance when false alarms must be minimized, while their behavior near high TPR indicates effectiveness when sensitivity is prioritized. If one curve consistently lies above the other, it stochastically dominates and is superior for any threshold choice.

What is the difference between Precision and Recall?

Precision measures the **quality** of your positive predictions. It answers the question: *Of all the instances the model flagged as positive, how many were actually positive?* **Minimize False Positives**.

Recall measures the **completeness** of your model (ability to not lose real positives) . It answers the question: *Of all the actual positive instances that exist, how many did the model correctly identify?* **Minimize False Negatives**.

What measure do you use?

If the dataset is balanced (same number of classes, evenly distributed), then you use Accuracy.

If the dataset is unbalanced, you use other measures, such as Precision and Recall, or even better, F1-Score. You can also use ROC-AUC.

3.1.4 Talk about Fayyad and Irani Discretization Method

The **Fayyad and Irani method** (1992) is a supervised technique for discretizing continuous attributes in Classification tasks. It aims to divide a numeric attribute into intervals where class distributions are as homogeneous as possible.

The method is grounded in information theory. A key property proved by the authors is that the **optimal cut point** for maximizing Information Gain always lies between two adjacent instances in the sorted sequence that belong to different classes (boundary points).

Entropy measures the uncertainty or impurity of a dataset. In Classification problems, it quantifies how mixed the k class labels are in the dataset S:

$$Ent(S) = -\sum_{i=1}^{k} P(C_i, S) \log_2 P(C_i, S)$$

Where $P(C_i, S)$ is the proportion of instances in S belonging to class C_i. For a potential cut point T that partitions S into subsets S_1 and S_2, the resulting **Class Information Entropy** is:

$$E(A, T; S) = \frac{|S_1|}{|S|} Ent(S_1) + \frac{|S_2|}{|S|} Ent(S_2)$$

The **Information Gain** is then the reduction in Entropy:

$$Gain(A, T; S) = Ent(S) - E(A, T; S)$$

The discretization process follows these recursive steps:

1. **Sorting:** Sort the instances in S according to the values of the continuous attribute A;
2. **Boundary Points:** Identify all potential cut points T located between adjacent instances with different class labels;
3. **Selection:** For every candidate T, calculate $Gain(A, T; S)$. Select the cut point T_{best} that maximizes this gain;
4. **MDL Stopping Criterion:** The split at T_{best} is accepted if and only if:

$$Gain(A, T_{best}; S) > \frac{\log_2(N-1)}{N} + \frac{\Delta(A, T_{best}; S)}{N}$$

where N is the number of instances in S. The value of Δ is calculated as:

$$\Delta(A, T_{best}; S) = \log_2(3^k - 2) - [k \cdot Ent(S) - k_1 \cdot Ent(S_1) - k_2 \cdot Ent(S_2)]$$

In this formula:

- k is the number of classes represented in S;
- k_1 is the number of classes represented in S_1;
- k_2 is the number of classes represented in S_2.

5. **Recursion:** If the criterion is met, the algorithm is applied recursively to the resulting partitions S_1 and S_2. If not, the current interval is considered final and no further splits are made.

This method effectively balances data purity and model simplicity. Its entropy-based criterion and MDL stopping rule make it a robust, because it automatically determines the number of intervals, preventing overfitting by penalizing the creation of too many partitions through the MDL threshold. It is widely used in supervised learning, particularly in decision tree induction.

3.2 Classification Algorithms

3.2.1 Talk about Decision Trees

Decision Tree Induction is a Classification method that builds a hierarchical tree where internal nodes are test attributes, branches represent outcomes, and leaves indicate class labels. Classification is performed by following a path **from root to leaf based on attribute values**.

Decision trees are valued for simplicity and interpretability. They produce models that are easy to visualize and translate into human-readable rules.

The induction of a decision tree is a **recursive, top-down process** known as the **divide-and-conquer** strategy. It is a **greedy algorithm** (at each step choose the best solution available at that moment). Initially, all training examples are placed at the root node. The algorithm then **selects an attribute that best separates the data into distinct classes**. The dataset is partitioned into subsets based on the outcomes of the selected attribute test, and the same process is repeated recursively for each subset. *Attributes are categorical. If continuous-valued, they are discretized in advance.*

The recursion stops when one of the following conditions holds:

1. All instances in the current subset belong to the same class;
2. There are no remaining attributes on which to split, in which case the node is labeled with the majority class of the instances it contains;
3. There are no samples left to split.

The fundamental idea is to create partitions that are as **pure** (one class per branch) as possible, meaning that most instances in a subset belong to a single class.

To decide which attribute should be used for splitting at each node, various attribute selection measures are used. These measures aim to quantify the quality of a split in terms of how well it separates the data into homogeneous class distributions.

Information Gain (ID3)

Information Gain, introduced by Quinlan, is based on the concept of *Entropy* from information theory.

Entropy measures the uncertainty or impurity of a dataset. In Classification problems, it quantifies how mixed the class labels are:

$$Ent(D) = -\sum_{i=1}^{k} P(C_i) \log_2 P(C_i)$$

Given a training set D with m classes $C_1, C_2, \ldots, C_m$, the **expected information** needed to classify a tuple is the Entropy:

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i)$$

where p_i is the non-zero probability that an arbitrary tuple in D belongs to class C_i.

If attribute A has v distinct values $\{a_1, a_2, \ldots, a_v\}$ and divides D into subsets $D_1, D_2, \ldots, D_v$, the **information needed** to classify a tuple in D after using A to split D into v partitions, is:

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} Info(D_j)$$

$Info_A(D)$ tells us how much information would we still need after partitioning to arrive at the exact Classification. $\frac{|D_j|}{|D|}$ is the **cardinality** ratio that acts as a weight to the jth partition.

The information gained by branching the attribute A is then:

$$Gain(A) = Info(D) - Info_A(D)$$

$Gain(A)$ is the expected reduction in the information required, caused by knowing the value of A.

The attribute with the highest Information Gain is selected for splitting. However, this measure tends to be biased toward attributes with many distinct values. This is because it tends to reduce Entropy even without true predictive ability.

To compute **Information Gain** for a **continuous-valued attribute**, it is necessary to determine the **optimal split point**. The process begins by **sorting** the values of the attribute in increasing order. Candidate split points are then defined as the **midpoints** between each pair of adjacent values, computed as

$$\frac{a_i + a_{i+1}}{2}$$

For each candidate split, the dataset is partitioned into two subsets: D_1, containing all tuples such that $A \leq$ **split point**, and D_2, containing all tuples such that $A >$ **split point**. The **expected information requirement** ($Info_A(D)$) is evaluated for each partition, and the split point that **minimizes** this quantity (equivalently, **maximizes Information Gain**) is selected. As an alternative strategy, **discretization** of continuous attributes may be performed **before** applying Decision Tree.

Gain Ratio (C4.5)

To reduce the bias of Information Gain, the C4.5 algorithm uses the **Gain Ratio**, which divides the Information Gain by the intrinsic information (or Entropy) of the split (applies a normalization):

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \log_2 \left(\frac{|D_j|}{|D|} \right)$$

Unlike standard Entropy, $SplitInfo_A(D)$ measures the impurity of the attribute values themselves. It represents the potential information generated by splitting the training data D into v partitions.

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo_A(D)}$$

$GainRatio(A)$ considers for each outcome the number of tuples having a value of the Attribute a_j with respect to the total number of tuples of D.

The attribute with the highest $GainRatio(A)$ is chosen as the splitting attribute, provided its Information Gain is above the average gain value. **It tends to prefer unbalanced splits in which one partition is smaller than the other**.

Gini Index (CART)

Another popular measure, used in the **CART** (Classification and Regression Trees) algorithm, is the **Gini Index**. It measures the degree of impurity in a dataset:

$$Gini(D) = 1 - \sum_{i=1}^{m} p_i^2$$

To determine the best binary split on A, we examine all the possible subsets that can be formed using known values of A. When considering a binary split, we compute a weighted sum of the impurity of each resulting partition. For example, if a binary split on A partitions D into D_1 and D_2, the Gini index of D given that partitioning is:

$$Gini_A(D) = \frac{|D_1|}{|D|} Gini(D_1) + \frac{|D_2|}{|D|} Gini(D_2)$$

For each attribute, each of the possible binary splits is considered.

$$\Delta Gini(A) = Gini(D) - Gini_A(D)$$

The attribute with the minimum $Gini_A(D)$, or equivalently, the maximum reduction in impurity $\Delta Gini(A)$, is selected for splitting.

The Gini index tends to prefer attributes that produce balanced and pure partitions. It has difficulties when the number of classes is large.

Algorithm

The formal process for growing a decision tree from a training set D is described by the following steps:

1. **Create a node** N;
2. **Check Base Cases (Stopping Conditions):**
   - If all instances in D belong to the same class C, return N as a leaf node labeled with C;
   - If the attribute list is empty, return N as a leaf node labeled with the **majority class** in D (majority voting);
   - If there are no instances left to split.
3. **Attribute Selection:** Apply an *Attribute Selection Measure* (such as Information Gain, Gain Ratio or Gini Index) to find the "best" attribute to split the data;
4. **Labeling:** Label node N with the selected splitting attribute;
5. **Partitioning and Branching:** For each unique value a_j of the splitting attribute:
   (a) **Create Branches:** If *categorical*, split into branches for each value a_j. If *numeric*, split at threshold t into $A \leq t$ and $A > t$, chosen to optimize the criterion;
   (b) **Handle Empty Subsets:** Let D_j be the subset of training instances in D that satisfy the condition. If D_j is empty, attach a leaf node labeled with the majority class in the parent set;
   (c) **Recursion:** Else, attach the node returned by recursively calling the algorithm on the subset D_j (point 1 of the algorithm).
6. **Return** N.

Overfitting and Pruning

A major concern in decision tree learning is **overfitting**, which occurs when the tree captures noise or anomalies in the training data rather than the underlying pattern. **An overfitted tree performs well on the training set but poorly on unseen data** (test set).

To address this, **pruning techniques** are applied to simplify the tree and improve generalization. Two main strategies exist: prepruning and postpruning.

In **Prepruning**, a node is not split if the potential division does not yield a statistically significant improvement, as determined by criteria such as max-instances, Chi-Square tests between features or Information Gain thresholds. Selecting suitable thresholds is challenging, and poor choices can lead to the *horizon effect*, where a split that appears unhelpful at an early stage may prevent more informative splits at deeper levels of the tree.

Postpruning first allows the tree to grow fully and then removes branches that do not contribute significantly to accuracy. The following specific methods are widely used:

- **Reduced Error Pruning:** *REP* is a bottom-up approach that requires a separate **validation dataset**. Starting at the leaves, each internal node is temporarily replaced by a leaf labeled with the majority class. If the error rate on the validation set decrease (or not increase) after the replacement, the pruning is made permanent. While simple and effective, its main drawback is the requirement of a dedicated validation set, which reduces the amount of data available for training.
- **Cost-Complexity Pruning (CART):** This bottom-up method balances the error rate against the tree's complexity. It identifies a sequence of progressively larger subtrees by minimizing the cost-complexity function:

  $$CC(T) = Error(T) + \alpha|L(T)|$$

  where $Error(T)$ is the **resubstitution error** (error rate on training data), $|L(T)|$ is the **number of leaf nodes** in the subtree T, and α is a **complexity parameter**. The algorithm calculates a value for α for each internal node and prunes the node with the smallest "cost" of keeping it. The final "best" tree is typically chosen via cross-validation.

- **Pessimistic Pruning (C4.5):** Similar to Cost-Complexity pruning. Since training error is often too low, this method adds a **penalty** (for instance, 0.5 instances per leaf) to create a more "pessimistic" error estimate:

$$e_{pess} = \frac{\sum(errors + penalty)}{N}$$

  It adjusts the error rates obtained by the training set by adding a penalty, computed by adopting a heuristic approach based on statistical theory. If the error rate in the node is lower than the error rate in the subtree originated from the node, then the subtree is pruned.

In the hypothesis that information Entropy is equal to zero, what is the scenario in the training set?

When the information entropy of a training set (or a specific partition) D is equal to zero, it represents a scenario of perfect purity or homogeneity. **It means that all instances in the dataset belong to the same class.**

What are the drawbacks of Decision Tree?

Overfitting is the most significant drawback. Decision trees tend to grow overly complex branches that capture noise and outliers in the training data. While pruning helps, a tree can still overfit, leading to excellent performance on training data but poor generalization on unseen data.

The algorithm uses a **greedy approach**, making the locally optimal choice at each node attribute selection measure. This does not guarantee the discovery of the globally optimal tree structure.

Decision trees are **highly sensitive to small variations** in the training set. A minor change in the data can result in a completely different set of splits, making the model somewhat unreliable.

In the end, each attribute selection metric has its own **bias**.

3.2.2 Tell me about Bayesian Classification and Bayesian Belief Networks

Bayesian Classification is a probabilistic approach to supervised learning that relies on Bayes' theorem to estimate the posterior probability of each class given an input instance. Given a hypothesis H (for example, that a tuple X belongs to class C_i) and observed evidence X (the attribute vector), Bayes' theorem states:

$$P(H \mid X) = \frac{P(X \mid H)\, P(H)}{P(X)}.$$

$P(H \mid X)$ is the probability that the hypothesis H holds given the observed data sample X. It is called **posterior probability**.

In Classification, we choose the class with the **maximum posterior probability**:

$$\hat{C} = \arg\max_i P(C_i \mid X) = \arg\max_i P(C_i)\, P(X \mid C_i).$$

$P(X)$ is ignored because it is a constant for all classes. Informally, this can be written as $posteriori = (likelihood \cdot prior)/evidence$.

Naïve Bayes Classifier

The **Naïve Bayes classifier** simplifies the estimation of $P(X \mid C_i)$ by **assuming that the attributes are conditionally independent** given the class:

$$P(X \mid C_i) = \prod_{k=1}^{n} P(x_k \mid C_i).$$

This allows the model to be trained efficiently using simple frequency counts. A Naïve Bayesian Classifier memorizes the prior probabilities $P(C_i)$ and conditional probabilities $P(x_k \mid C_i)$. $P(X \mid H)$ is the probability of observing those specific values of X while already knowing that the hypothesis H is true. A_k is an attribute of X and x_k is a possible value of A_k and the actual value of A_k in X.

- **Categorical attributes:** estimated by relative frequencies in each class:

$$P(x_k \mid C_i) = \frac{\#(x_k; C_i)}{\#(C_i)}$$

- **Continuous attributes:** often modeled as Gaussian distributions:

$$P(x_k \mid C_i) = \frac{1}{\sqrt{2\pi}\sigma_{ik}} e^{-\frac{(x_k - \mu_{ik})^2}{2\sigma_{ik}^2}}$$

Zero-probability problem arises when not all conditional probabilities are different from 0. Thus the entire computation becomes 0. To solve it, just use Laplace smoothing (add-one):

$$\hat{P}(x_k \mid C_i) = \frac{\#(x_k \text{ in } C_i) + 1}{\#(C_i) + V_k}$$

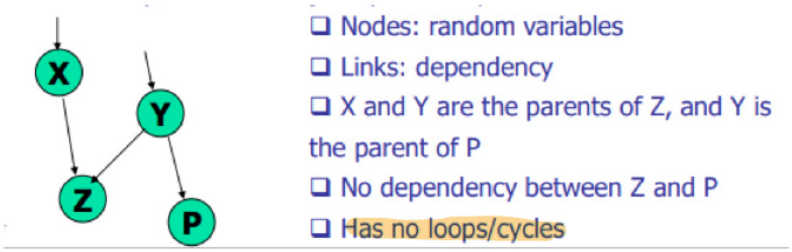
where V_k is the number of distinct values of attribute A_k.

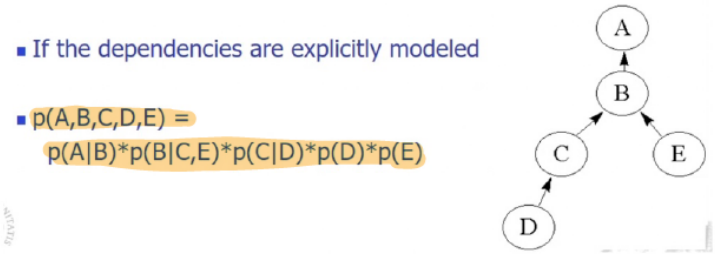
Although **the independence assumption** is rarely true, Naïve Bayes performs surprisingly well in many domains such as text Classification, spam detection, and medical diagnosis. It is simple, robust, fast to train and test, and incremental. Its main limitation is that correlated attributes can violate the independence assumption and reduce accuracy.

Bayesian Belief Networks

A **Bayesian Belief Network** (or **Bayesian Network**) generalizes Naïve Bayes by explicitly representing dependencies among variables. It consists of:

- A **directed acyclic graph (DAG)** where nodes represent attributes and edges represent direct dependencies.
- A set of **conditional probability tables** (**CPTs**) specifying $P(X_i \mid Parents(X_i))$ for each attribute X_i with parents $Parents(X_i)$. There is a table for each variable (node).





The joint probability distribution over all variables factorizes according to the graph structure:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid Parents(X_i))$$

If an arc is drawn from a node Y to a node Z, then Y is a parent or immediate predecessor of Z, and Z is a descendant of Y. **Each variable is conditionally independent of its non-descendants in the graph, given its parents**. Naïve Bayes can be viewed as a special case of a Bayesian Network where the class variable is the single parent of all attributes and there are no edges among attributes.

According to the complexity of the available data and the prior knowledge of the model, training follows one of four scenarios:

1. **Known Structure, All Variables Observable:** The simplest case. Training consists of estimating CPT entries using the frequency of occurrences in the training data (similar to Naïve Bayes).
2. **Known Structure, Hidden Variables:** When some variables are not observed in the training data, methods like **Gradient Descent** or the **Expectation-Maximization (EM)** algorithm must be used to estimate parameters.
3. **Unknown Structure, All Variables Observable:** The algorithm must perform a search through the space of possible DAGs to find the structure that best fits the data dependencies (often using score-based or constraint-based approaches).
4. **Unknown Structure, Hidden Variables:** The most difficult scenario (an active area of research), requiring simultaneous discovery of the network topology and the estimation of hidden parameters.

When the structure is known but variables are hidden, we aim to learn the CPT entries $w_{ijk} = P(Y_i = y_{ij} \mid Parents(Y_i) = u_{ik})$. The objective is to maximize the log-likelihood of the training set S containing D instances:

$$\ln P_w(S) = \sum_{d=1}^{|S|} \ln P_w(X_d)$$

The parameters are updated using the gradient of the log-likelihood:

$$\frac{\partial \ln P_w(S)}{\partial w_{ijk}} = \sum_{d=1}^{|S|} \frac{P(Y_i = y_{ij}, Parents(Y_i) = u_{ik} \mid X_d)}{w_{ijk}}$$

The learning step is performed as:

$$w_{ijk} \leftarrow w_{ijk} + \eta \left(\frac{\partial \ln P_w(S)}{\partial w_{ijk}} \right)$$

where η is the learning rate. After each update, the w_{ijk} values must be **renormalized** so that the probabilities for each parent configuration sum to 1 ($\sum_j w_{ijk} = 1$).

3.2.3 Talk about Rule-Based Classification

Rule-based Classification represents knowledge as a set of **IF–THEN rules** of the form:

$$R: \text{ IF conditions THEN class} = C_i.$$

Each rule consists of an **antecedent** (precondition) and a **consequent** (class assignment).

Two measures quantify a rule's quality:

- **Coverage:** proportion of tuples satisfying the rule's conditions.

  $$\text{coverage}(R) = \frac{n_{\text{covers}}}{|D|}$$

  where n_{covers} is the number of tuples covered by R.

- **Accuracy:** proportion of correctly classified tuples among the covered ones.

  $$\text{accuracy}(R) = \frac{n_{\text{correct}}}{n_{\text{covers}}}$$

When several rules are triggered for the same tuple, **conflict resolution** is required. Common strategies are:

- **Size ordering:** rule with more conditions (more specific antecedent) has priority.
- **Class-based ordering:** rules of more frequent or higher-cost classes first.
- **Rule-based ordering (decision list):** rules sorted by quality (accuracy, coverage, size) or expert priority. Each rule implies the negation of those above it.

Two main approaches exist to obtain rule sets.

1. Rule extraction from decision trees.

- Each path from root to leaf corresponds to one rule.
- The conjunction of attribute tests forms the antecedent. The leaf label is the consequent.
- Rules are initially *mutually exclusive and exhaustive.*
- Example:

$$\text{IF age} = \text{youth AND student} = \text{no} \Rightarrow \text{buys\_computer} = \text{no}$$
$$\text{IF age} = \text{senior AND credit\_rating} = \text{fair} \Rightarrow \text{buys\_computer} = \text{yes}$$

- Pruning: remove conditions that do not increase estimated accuracy. In C4.5, a pessimistic pruning corrects training bias.
- After pruning, rules may overlap or leave gaps.
- C4.5 groups rules per class and orders them to minimize false positives. The class with most uncovered tuples becomes the default.

2. Direct rule induction from data with heuristic approaches.

Sequential Covering Algorithm (separate-and-conquer): learn rules one at a time.

1. Select a target class C_i. Rules are learned one at a time (one for each class, it cycles through the classes).
2. Learn a rule covering many tuples of C_i and few of others. Remove covered positive examples.
3. Repeat until no positive examples remain or rule quality drops below threshold.

Compared to decision trees: rules learned sequentially, not simultaneously. Typical algorithms: **FOIL, AQ, CN2, RIPPER**.

A rule is built using a **Greedy Depth First** strategy:

- Start with the most general rule (empty condition).
- At each step, we add a new condition. We choose the one with the **highest FOIL_gain**:

$$\text{FOIL\_Gain}(R') = \text{pos}' \left[\log_2\left(\frac{\text{pos}'}{\text{pos}' + \text{neg}'}\right) - \log_2\left(\frac{\text{pos}}{\text{pos} + \text{neg}}\right) \right]$$

  where *pos, neg* are positive (TP) and negative (FP) examples covered by R, and *pos', neg'* after adding the new condition.
- If the generated rule has a FOIL_Gain greater than a certain threshold, it is inserted into the ruleset.

FOIL_Gain favors rules that improve accuracy and cover many positives. Using accuracy alone does not take covered tuples into account.

Rules are pruned using a separate validation set to avoid overfitting. In FOIL, if a pruned version of a rule yields higher `FOIL_Prune` score, the condition is removed.

$$\text{FOIL\_Prune}(R) = \frac{pos - neg}{pos + neg}$$

FOIL_Prune(R) > FOIL_Prune(R_{pruned}) remove R.
RIPPER prunes the most recent condition first and continues pruning while performance improves.
An extension of rule-based learning that uses Frequent Pattern Mining.

- Rules have the form:

$$p_1 \wedge p_2 \wedge \cdots \wedge p_l \Rightarrow \text{class} = C$$

  with support and confidence measures.
- Advantages: considers multiple attribute-value combinations simultaneously and can surpass decision-tree accuracy.

3.2.4 Is it possible to classify with Frequent Patterns?

Classification by using frequent patterns, also called **Associative Classification**, combines Association Rule Mining with supervised learning.

It exploits **Frequent Pattern Mining** (chapter 5) to discover strong associations between attribute-value combinations and class labels.

The method searches for patterns of the form:

$$p_1 \wedge p_2 \wedge \cdots \wedge p_l \Rightarrow \text{class} = C \quad (\text{confidence}, \text{support})$$

where p_i are attribute–value pairs (items). Each of these rules links frequent patterns to a class label. The resulting rules are organized to form a rule-based classifier.

It is effective because it explores high-confidence relationships among multiple attributes simultaneously, rather than testing one attribute at a time as in decision trees. It can even overcome constraints of greedy splits and local decisions in Decision Tree Induction.

Empirically, associative classifiers often achieve higher accuracy than some traditional classifiers such as C4.5.

1. **Frequent Pattern Mining** discover all frequent conjunctions of attribute–value pairs that meet minimum support;
2. **Rule Generation:** generate high-confidence rules whose consequent is a class label;
3. **Rule selection and organization:** choose a subset of rules based on quality measures (support, confidence, significance) and organize them for Classification;
4. **Classification:** for a new tuple, find all matching rules and apply an appropriate decision procedure (e.g., first-match, voting, weighted correlation).

CBA (Classification Based on Associations, Liu, Hsu, Ma, KDD'98)

The **Classification Based on Associations Algorithm** integrates Association Rule Mining with Classification tasks. It focuses on generating association rules that directly connect sets of attribute-value pairs with class labels, ranking them according to their reliability and applying them to classify new instances. The algorithm proceeds as follows:

1. **Rule Mining:** CBA first extracts rules of the form *condition set* $\Rightarrow$ *class label*, where the condition set represents a group of attribute-value pairs that occur together in the data and the class label represents the target category;
2. **Rule Ranking:** Once the rules are generated, they are ordered in decreasing order of **confidence**, which measures how often the rule correctly predicts the class. In cases where two or more rules share the same confidence value, the one with higher **support** (frequency in the dataset) is given priority;
3. **Rule Selection:** If multiple rules have identical antecedents (the same condition sets), only the one with the highest confidence is retained to avoid redundancy;
4. **Classification:** When classifying a new tuple, the algorithm scans the ranked rule list and applies the first rule whose condition set matches the tuple's attributes. The class label of that rule becomes the predicted class;
5. **Default Rule:** If none of the rules match a given tuple, a **default rule** (which has the lowest precedence) is applied. This default rule typically assigns the most frequent class in the training data.

CMAR (Classification based on Multiple Association Rules, Li, Han, Pei, ICDM'01)

The **Classification based on Multiple Association Rules Algorithm** extends the ideas of CBA by using multiple rules simultaneously rather than relying on a single matching rule. It uses an efficient Frequent Pattern Mining approach and statistically prunes less significant rules. The main steps of CMAR are:

1. **Rule Generation:** CMAR employs a variant of the **FP-growth** algorithm to efficiently mine all association rules that satisfy predefined minimum support and confidence thresholds. This avoids the need to generate and test all possible itemsets explicitly;
2. **Rule Pruning:** After rule generation, CMAR removes redundant and statistically insignificant rules:

(a) If two rules R_1 and R_2 predict the same class and the antecedent of R_1 is more general than that of R_2, and $\text{conf}(R_1) \geq \text{conf}(R_2)$, then R_2 is pruned since it provides no additional useful information;

(b) CMAR also performs a statistical test based on the χ^2 measure to determine whether the antecedent and the class label of a rule are positively correlated. Rules failing this significance test are eliminated.

3. **Classification Process:** When classifying a new tuple:

   (a) All rules that match the tuple are retrieved. If these rules predict different class labels, they are grouped according to their class;

   (b) For each group of rules corresponding to a given class, CMAR computes a weighted χ^2 correlation value that reflects how strongly those rules support that class;

   (c) The class associated with the group having the highest correlation score is assigned as the predicted class for the tuple.

4. **Performance:** Through the combination of efficient rule mining, pruning, and ensemble-like decision making, CMAR generally achieves slightly higher accuracy and better memory efficiency than CBA.

CPAR (Classification based on Predictive Association Rules, Yin & Han, SDM'03)

The **Classification based on Predictive Association Rules Algorithm** blends the strengths of associative Classification and traditional rule learning. Instead of performing full frequent itemset mining, it uses a FOIL-like approach to generate predictive rules directly. Its main procedure can be described as follows:

1. **Rule Generation:** CPAR adopts a rule learning strategy inspired by the FOIL algorithm. It incrementally constructs predictive rules by adding attribute-value conditions that improve the rule's accuracy;

2. **Weighted Sampling:** Unlike FOIL, which removes all positive samples covered by a rule, CPAR only reduces their weights. This allows multiple rules to overlap in the samples they cover, ensuring that important instances can influence the creation of several rules;

3. **Rule Grouping:** After the rules are generated, they are organized according to their predicted class labels. This grouping facilitates class-specific analysis during prediction;

4. **Rule Selection for Classification:** For each class, the algorithm selects the top-k rules that have the highest **expected accuracy**. When a new tuple is classified, the selected rules collectively contribute to the decision-making process, rather than relying on a single rule;

5. **Efficiency and Accuracy:** CPAR achieves a high level of computational efficiency and produces Classification accuracy that is very close to that of CMAR, making it a practical choice for large datasets.

3.2.5 Talk about Lazy Learners

Lazy learning, also called **instance-based learning**, refers to a family of supervised learning methods that postpone generalization until a query is made. Unlike **eager learners** such as Decision Trees or Naïve Bayes classifiers, which construct a model during the training phase, lazy learners simply store the training data and perform all computation at Classification time. The intuition is that the system keeps the examples "as they are" and reasons locally around a new case rather than forming a global hypothesis in advance.

From a conceptual point of view, eager learners attempt to approximate the entire decision surface during training, while lazy learners perform a local approximation around each query point.

Their advantages include simplicity, the ability to capture complex local patterns, and good performance in domains with irregular decision surfaces. Their disadvantages include high computational cost during Classification, large storage requirements, and reduced effectiveness in high-dimensional spaces.

The k-Nearest Neighbor Method

The k**-Nearest Neighbor** (k-NN) algorithm is one of the most typical examples of a lazy learning approach. Each training instance is represented as a point in an n-dimensional feature space. To classify a new instance, k-NN computes its distance to all stored examples, identifies the k closest ones, and assigns the instance to the most common class among these neighbors. When the target variable is continuous, the predicted value is the average of the neighbors' target values.

The distance between two points X_1 and X_2 with attributes $(x_{1i}; x_{2i})$ is most often measured by the Euclidean metric:

$$\text{dist}(X_1; X_2) = \sqrt{\sum_i (x_{1i} - x_{2i})^2}$$

For categorical attributes, the distance equals 0 if the values are identical and 1 otherwise, while missing values are treated as maximal differences.

An important refinement is the **distance-weighted k-NN**, where each neighbor's influence is inversely proportional to its distance from the query instance. The choice of k critically affects performance: small values make the model sensitive to noise, whereas large values lead to smoother but less flexible decision boundaries. The optimal k is typically determined through cross-validation.

Since k-NN compares each query instance to all training points, it is computationally expensive for large datasets. To improve efficiency, data structures such as **kd-trees** and **ball trees** are commonly used to speed up nearest-neighbor searches, and approximate or partial distance computations can further reduce processing time.

To prevent attributes with large initial ranges from dominating the distance calculation, **Min-Max Normalization** is applied to scale all attributes to a range like $[0; 1]$:

$$x' = \frac{x - \min_A}{\max_A - \min_A}$$

Where $\min_A$ and $\max_A$ are the minimum and maximum values for attribute A.

The value of k is determined experimentally (often via cross-validation).

- If k is too **small** (e.g., $k = 1$), the model is sensitive to **outliers and noise** (overfitting).
- If k is too **large**, the neighbors may include instances from other classes, leading to **oversmoothing** (underfitting).

A limitation of k-NN and related methods is the **curse of dimensionality**. As the number of irrelevant attributes increases, all distances tend to become similar, making the distinction between neighbors meaningless. Therefore, feature selection or weighting is essential to preserve discriminative power.

Editing Algorithm

Since lazy learners retain all training instances, storage requirements can be significant. To reduce redundancy, several **editing techniques** exist.

Wilson editing removes instances that are likely to be noise. For each instance x in the training set, it is classified using its k-Nearest Neighbors. If the class of x does not match the majority class of its neighbors, x is removed. This results in *smoother decision boundaries* and removes overlap between classes.

Multi-edit applies Wilson editing iteratively to stabilize the dataset. The dataset is partitioned into subsets. Wilson editing is applied to one subset using the others as a reference, then the subsets are rotated. The process repeats until no more instances are removed (stability is reached), ensuring a *consistent and representative* training set.

Citation Editing takes into account both the nearest neighbors of a point and the points for which it is a neighbor. An instance is retained only if its class label is consistent with both its references and its citations, providing a **more robust filter** for outliers in high-dimensional spaces.

Supervised Clustering can also compress the data by replacing groups of similar examples with prototypes. This technique eliminates the non representative dataset instances.

These methods help improve both efficiency and generalization.

Case-Based Reasoning

Case-Based Reasoning (CBR) is another major form of lazy learning, based on solving new problems by adapting solutions from similar past cases. Each *case* in the *case base* includes a problem description and its corresponding solution or outcome. The underlying principle is that similar problems tend to have similar solutions.

The reasoning process in CBR is typically structured as a four-step cycle known as the **CBR cycle**:

1. **Retrieve:** identify one or more past cases most similar to the current problem using a similarity metric;
2. **Reuse:** apply or adapt the retrieved solution to the new situation;
3. **Revise:** test the proposed solution and, if necessary, correct it;
4. **Retain:** store the newly confirmed case for future reuse.

Similarity retrieval can be based on numerical distances, symbolic matching, or structural similarity depending on how cases are represented. Efficient retrieval often requires proper indexing through structures such as trees, discrimination networks, or case-retrieval nets. Adaptation can range from simple substitution of attribute

values to more complex transformational procedures. To prevent the uncontrolled growth of the case base, selective retention strategies are employed.

Its main strengths are transparency and the ability to learn incrementally from experience, although its performance depends heavily on the quality of the similarity measure and the efficiency of retrieval mechanisms.

3.3 Evaluation

3.3.1 How does the evaluation of classifiers work?

A single run of different classifiers is not enough to determine which one performs better. In one trial, a classifier might appear superior purely by chance. To draw valid conclusions, we must evaluate performance over multiple trials.

When only one labeled dataset is available, we can use resampling techniques to estimate model performance more reliably. The main methods are the Hold-Out Method, K-Fold Cross Validation, and the Bootstrap Method.

Hold-Out Method

In the **Hold-Out Method**, the dataset (learning set) is divided into two disjoint subsets:

- **Training set**: used to train the model;
- **Test set**: used to evaluate the final model performance.

Typically, the data is split into 70% training and 30% testing (or similar ratios). This approach is simple but may yield high variance in the performance estimate since it depends on a single random split.

K-Fold Cross Validation

K-Fold Cross Validation reduces the variance of the performance estimate by using multiple splits. The dataset is divided into K approximately equal folds:

1. Train the model on $K - 1$ folds;
2. Test it on the remaining fold;
3. Repeat this process K times, each time using a different fold for testing.

A common choice is $K = 10$, known as 10-fold cross validation. In **stratified cross-validation**, each fold preserves approximately the same class distribution as the original dataset.

Bootstrap Method

The **Bootstrap Method** generates multiple training datasets by sampling with replacement from the original dataset. Each bootstrap sample is used to train the model, and the remaining unseen samples (the "out-of-bag" samples) are used for testing. A bootstrap is a new dataset of the same size as the original, obtained through sampling with replacement, used to simulate new realizations of the data, estimate uncertainty or performance, and add variability to the new data. Each bootstrap sample may contain duplicates. Bootstrap provides an estimate of the model's stability and variance. Approximately 63.2% of unique examples end up in training and approximately 36.8% remain out-of-bag.

Estimating Confidence Intervals

- **Hold-Out:** Fast but less reliable due to dependence on a single split.
- **K-Fold Cross Validation:** Balanced trade-off between bias and variance.
- **Bootstrap:** Useful for small datasets and estimating model uncertainty.

Typically, a hybrid between the Holdout Method and K-Fold Cross Validation is used, where you divide the learning set into a training set and a test set, and then perform cross-validation on the training set to choose and optimize the parameters for the Classification algorithm to use, and finally view the statistics (Accuracy, F-Score, Error Rate, ...) on the test set.

Suppose we have 2 classifiers, M1 and M2, which one is better than the other? We use 10-fold cross-validation to obtain $\overline{err(M1)}$ and $\overline{err(M2)}$. These **mean error rates** are just estimates of error on the true population of future data cases.

And what if the difference between the 2 error rates is just attributed to chance? We use a **test of statistical significance**. If we have many classifiers (and not two), we use statistical tests that work on multiple classifiers, or we use one with all the pairs of classifiers. These tests assess whether the observed performance gap reflects a real difference between classifiers or is likely the result of random variation.

Parametric Test (T-Test)

Parametric tests assume that the data follows a specific distribution (usually normal). They are appropriate when this assumption is reasonable and the sample size is sufficiently large. Other assumptions are: the two samples are independent of one another and the two populations have similar variance.

The **paired t-test** is commonly used to compare the mean performance of two classifiers across multiple runs (across k-Folds of Cross Validation). The test statistic is computed as:

$$t = \frac{\overline{err(M_1)} - \overline{err(M_2)}}{\sqrt{\frac{var(M_1 - M_2)}{k}}} = \frac{\overline{err(M_1)} - \overline{err(M_2)}}{\sqrt{\frac{1}{k^2}\sum_{i=1}^{k}\left[(err(M_1)_i - err(M_2)_i) - (\overline{err(M_1)} - \overline{err(M_2)})\right]^2}}$$

where:

- $\overline{err}_i$ is the mean of the errors err_i;
- $var(...)$ is unbiased sample variance formula. The square root of variance is the standard deviation;
- k is the number of paired observations (number of folds).

Choose a significance level α, typically 0.05 is chosen. α is a probability threshold set a priori that controls the risk of committing a Type I error (FP).

Degree of Freedom df is $k-1$. df corresponds to the number of independent cells in the errors table ($k \times 2$ matrix because we have two classifiers).

Hypothesis	Description
H_0 (Null Hypothesis)	The null hypothesis assumes that the mean difference $(\overline{err(M_1)} - \overline{err(M_2)})$ is zero, meaning both classifiers perform equivalently.
H_1 (Alternative Hypothesis)	If the computed t exceeds the critical value t_d from the Student's t distribution (for a chosen significance level α, typically 0.05 and a certain df), we reject H_0 and conclude that the difference is statistically significant.

Let's summarize the test:

1. Set the significance level α and determine degrees of freedom $df = k - 1$;
2. Compute the differences between model errors for each fold: $d_i = err(M_1)_i - err(M_2)_i$;
3. Compute the mean and variance of the differences:

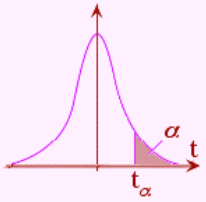
$$\bar{d} = \frac{1}{k}\sum_{i=1}^{k} d_i \qquad var(M_1 - M_2) = \frac{1}{k}\sum_{i=1}^{k}(d_i - \bar{d})^2$$

4. Compute the t-statistic:

$$t = \frac{\bar{d}}{\sqrt{\frac{var(M_1 - M_2)}{k}}}$$

5. Compare $|t|$ with the critical value $t_{\alpha,df}$: If $|t| > t_{\alpha,df}$, reject the null hypothesis and conclude that the performance difference is statistically significant. Otherwise it isn't.

Compare the computed statistic with the critical value from the Student's t distribution with α and $df = k-1$:



T-Distribution Table

df	α = 0.1	0.05	0.025	0.01	0.005	0.001	0.0005
∞	t_α=1.282	1.645	1.960	2.326	2.576	3.091	3.291
1	3.078	6.314	12.706	31.821	63.656	318.289	636.578
2	1.886	2.920	4.303	6.965	9.925	22.328	31.600
3	1.638	2.353	3.182	4.541	5.841	10.214	12.924
4	1.533	2.132	2.776	3.747	4.604	7.173	8.610
5	1.476	2.015	2.571	3.365	4.032	5.894	6.869
6	1.440	1.943	2.447	3.143	3.707	5.208	5.959
7	1.415	1.895	2.365	2.998	3.499	4.785	5.408
8	1.397	1.860	2.306	2.896	3.355	4.501	5.041
9	1.383	1.833	2.262	2.821	3.250	4.297	4.781
10	1.372	1.812	2.228	2.764	3.169	4.144	4.587

Non-Parametric Test (Wilcoxon Signed Rank Test)

When the assumption of normality is not valid or the sample size is small, **non-parametric tests** are preferred. These tests rely on the ranking of differences rather than their actual values. The other two assumptions remain: the two samples are independent of one another and the two populations have similar variance.

The **Wilcoxon Signed Rank Test** is a non-parametric alternative to the paired t-test. It tests whether the median difference between two paired samples is zero.

The main assumptions are: paired observations are independent of other pairs, the paired differences are from a distribution symmetric about the median, and the measurement scale is at least ordinal so that ranking is meaningful.

These are the steps of the test:

1. For each pair $(x_i; y_i)$ (in our case they may be the accuracy pairs) compute the difference $d_i = x_i - y_i$;
2. Remove all pairs with $d_i = 0$ from further calculations. Let n be the number of remaining (non-zero) differences;
3. For the remaining differences form the absolute values $|d_i|$ and sort them in ascending order;
4. Assign ranks to the $|d_i|$ values. If ties occur, assign to each tied value the average of the ranks they would occupy. For example, a tie at ranks 3 and 4 is assigned rank 3.5, and the next distinct value receives rank 5;
5. Reattach the original signs of the d_i to their corresponding ranks, producing signed ranks;
6. Compute W^+, the sum of ranks with positive signs, and W^-, the sum of ranks with negative signs. Verify that $W^+ + W^- = \frac{n(n+1)}{2}$;
7. Define the test statistic W as $W = \min\{W^+; W^-\}$;
8. Decide how to obtain a p-value:
   (a) If n is small, obtain the exact p-value from the Wilcoxon distribution (critical-value tables or exact enumeration);
   (b) If n is large enough (practical rule-of-thumb: the rank-sum total $\frac{n(n+1)}{2}$ sufficiently large), use the normal approximation with tie correction.

9. If $\frac{n(n+1)}{2} \gtrsim 20$ the distribution of the W statistic tends to become a Normal curve, regardless of the shape of the original data. Computing the exact probability of W for $n = 50$ or $n = 100$ requires millions of sign-rank combinations and is computationally prohibitive. In these cases W is well approximated by a normal distribution, so we can use a closed-form formula to obtain the p-value.

    - When using the normal approximation compute the mean and (uncorrected) variance of W:

    $$\mu_W = \frac{n(n+1)}{4} \qquad \sigma_W^2 = \frac{n(n+1)(2n+1)}{24}$$

    - If tied absolute differences exist, correct the variance by subtracting for each tie group of size t the quantity $(t^3 - t)/48$:

    $$\sigma_W^2 \leftarrow \sigma_W^2 - \sum_{\text{tie groups}} \frac{t^3 - t}{48}.$$

    - Compute the standard score

    $$z = \frac{W - \mu_W}{\sigma_W}$$

    taking care to apply continuity-correction if required by convention. z is an approximation of W;

    - Obtain the two-sided (or one-sided, if appropriate) p–value from the standard normal distribution using z, or use the exact p from step 8(a).

10. Compare the p–value to the chosen significance level α. If $p \leq \alpha$, reject the null hypothesis that the median paired difference is zero, otherwise do not reject it.

11. Report results, including n, W^+, W^-, W, whether exact or approximate p was used, any tie corrections applied, the p–value, and the conclusion in context (direction and practical significance).

The test statistic is the smaller of these two sums:

$$W = \min(W^+; W^-)$$

If W is smaller than the critical value from the Wilcoxon distribution, we reject H_0 and infer that one classifier outperforms the other significantly.

Hypothesis	Description
H_0 (Null Hypothesis)	The null hypothesis assumes that the median difference is zero, meaning both classifiers perform equivalently.
H_1 (Alternative Hypothesis)	If W is smaller than the critical value from the Wilcoxon distribution, we infer that one classifier outperforms the other significantly.

3.3.2 How do you evaluate Classifiers with ROC Curves?

The **ROC curves** (Receiver Operating Characteristic curves) are a graphical tool used to evaluate and compare the performance of **binary classifiers**. It illustrates the trade-off between a classifier's **Sensitivity/Recall (True Positive Rate, TPR)**, and **False Positive Rate, FPR** across varying decision thresholds, providing a comprehensive view of a model's discriminative ability independent of class distribution.

For imbalanced datasets, the **Recall-Precision pairing** is often preferred over **Accuracy**. Since statistical tests like the **t-test** and **Wilcoxon test** are designed for single measurements, one can use **F-scores** or **ROC curves** for evaluation.

$$\text{TPR} = \frac{TP}{TP + FN} \qquad \text{FPR} = \frac{FP}{FP + TN}$$

The **TPR** measures how effectively the model identifies positive instances, while the **FPR** quantifies the rate of incorrectly classified negative instances. These values define the **ROC space**, with FPR on the x-axis and TPR on the y-axis. A ROC curve is obtained by varying the **decision threshold** and plotting the corresponding $(FPR; TPR)$ points. Lowering the threshold generally increases both TPR and FPR.

To generate the curve for a specific classifier, follow these steps:

1. **Probability Estimation:** For each test instance, obtain the probability $P(C_{pos} \mid X)$ from the classifier;
2. **Sorting:** Sort all test instances in **descending order** of their predicted probabilities;
3. **Iterative Plotting:** Start at the origin $(0; 0)$. Moving through the sorted list:
   - If the next instance is actually **Positive**, move vertically up by $1/m_{pos}$ (where m_{pos} is the total number of positive instances);
   - If the next instance is actually **Negative**, move horizontally to the right by $1/m_{neg}$ (where m_{neg} is the total number of negative instances).
4. **Termination:** The process concludes at $(1; 1)$ once all instances are processed.

The **ideal classifier** achieves a point near the top-left corner (high TPR, low FPR), while a **random classifier** lies along the diagonal from $(0; 0)$ to $(1; 1)$. Points above the diagonal indicate better-than-random performance, points below indicate worse-than-random performance.

The **Area Under the Curve (AUC)** summarizes classifier performance as a single value, representing the probability that a randomly chosen positive instance scores higher than a randomly chosen negative instance. AUC ranges from 0 to 1:

$$AUC = 1 \text{ for a perfect classifier,} \qquad AUC = 0.5 \text{ for a random classifier}$$

Values below 0.5 suggest systematic misclassification, correctable by inverting predictions. AUC is **threshold-independent** and **invariant under monotonic transformations**, focusing only on the ranking of predictions.

ROC curves allow visual and quantitative comparison of multiple classifiers. A classifier whose ROC curve lies consistently above and to the left of another is generally superior. Intersecting curves require application-specific considerations, balancing TPR and FPR based on context. For example, medical diagnostics prioritize high TPR to avoid missing positive cases, whereas fraud detection prioritizes low FPR to minimize false alarms. In terms of **model selection**, any classifier that lies **below** the convex hull is considered sub-optimal, since superior performance can always be obtained by choosing a classifier located on the hull.

Regarding **interpolation and visualization**, points along the lines connecting vertices of the convex hull can be realized through **interpolation**. This approach effectively transforms the discrete hull into a continuous "best-case" boundary.

ROC analysis is robust to **class imbalance** and provides a complete view across thresholds. However, it does not account for **probability calibration** and may overestimate performance in highly imbalanced data. In such cases, **precision–recall curves** focusing on the positive class are often more informative.

3.3.3 What is a Confusion Matrix?

A **Confusion Matrix** is a table used to evaluate a classification model by comparing **actual** and **predicted** classes. Rows represent actual classes, while columns represent predicted classes. In binary classification, it contains four values:

- **TP**: correctly predicted positives;
- **TN**: correctly predicted negatives;
- **FP**: negatives predicted as positives (Type I error);
- **FN**: positives predicted as negatives (Type II error).

Unlike accuracy, which can be misleading in **imbalanced datasets**, a confusion matrix shows how classes are misclassified and enables the computation of metrics such as **Precision, Recall, Specificity** (True Negative Rate), and **F-measure**.

A **Normalized Confusion Matrix** converts counts into proportions by normalizing each row. The diagonal entries correspond to **Recall** for each class, making it easier to compare performance across datasets and identify poorly predicted classes.

3.3.4 What are the techniques to improve accuracy?

Ensemble methods are strategies designed to improve the accuracy and robustness of Classification models by combining the predictions of multiple classifiers. The fundamental principle is that a group of weak learners, when combined appropriately, can produce a single strong learner with higher predictive performance than any individual model. These methods help reduce variance, bias, and the risk of overfitting, and are particularly effective when individual models are unstable, such as decision trees.

The goal of ensemble learning is to aggregate the outputs of several base classifiers so that the collective decision is more reliable. Each base classifier is trained on the same Classification problem but may differ in training data subsets, feature subsets, or initial conditions. The final decision can be obtained through methods such as majority voting for Classification tasks or averaging for Regression problems.

There are three main types of ensemble approaches: *bagging*, *boosting*, and other combination-based methods such as *stacking*.

Bagging (Bootstrap Aggregating)

Bagging, short for **Bootstrap Aggregating**, is an ensemble method that aims to reduce the variance of a classifier by training multiple models on different random subsets of the training data. It is particularly effective for algorithms that are sensitive to small changes in the training set, such as decision trees.

The process of bagging can be described as follows:

1. Given a training set D of size n, generate k new training sets $D_1, D_2, \ldots, D_k$ by sampling n instances from D uniformly and with replacement (bootstrap sampling).
2. Train a base classifier C_i on each bootstrap sample D_i.
3. For Classification, aggregate the predictions of all classifiers through majority voting. For numeric prediction, take the average of their outputs.

Since each model is trained on a slightly different subset of the data, the resulting ensemble reduces the effect of variance without significantly increasing bias. The overall performance tends to improve, especially when the base classifier is unstable. Random Forests are a notable extension of bagging, where both instance sampling and random feature selection are used to decorrelate individual trees.

The estimated prediction error in bagging can also be computed using the *out-of-bag* (OOB) samples, which are the instances not included in a given bootstrap sample. These OOB instances serve as an internal validation set, providing an unbiased estimate of the model's generalization error.

Boosting

Boosting is another ensemble technique designed to convert weak classifiers into a strong one by focusing successively on the most difficult training examples. Unlike bagging, which builds models independently, boosting constructs the ensemble sequentially, where each new classifier is influenced by the errors of its predecessors.

The general procedure for boosting can be summarized as follows:

1. Assign equal weights to all training instances (the tuples);
2. Train a base classifier on the weighted training data;
3. Evaluate the classifier's performance and increase the weights of instances that were misclassified, while decreasing the weights of those correctly classified;
4. Normalize the weights so that they form a valid probability distribution;
5. Train the next classifier on the updated weighted data;
6. Combine the predictions of all classifiers through a weighted voting mechanism, where classifiers with lower error rates receive higher weights.

The process continues for a predefined number of iterations or until the Classification error reaches a satisfactory level. The final model is a weighted sum of all individual classifiers, where each contributes proportionally to its accuracy.

Comparison Between Bagging and Boosting

Although both bagging and boosting combine multiple base models, they differ in philosophy and effect:

- **Bagging** reduces variance by training models in parallel on random bootstrap samples and averaging their results. It works best with high-variance, low-bias models such as decision trees.
- **Boosting** reduces both bias and variance by training models sequentially, each correcting the mistakes of the previous one. It generally achieves higher accuracy but may be more prone to overfitting and noise sensitivity.

3.3.5 Talk about Random Forest

Random Forest is an **ensemble** learning method and a **bagging method** used for both Classification and Regression tasks. It combines multiple **decision trees** to improve predictive accuracy and reduce overfitting. Each tree is trained on a random subset of the data and features, and the final output is determined by aggregating the predictions of all trees. Random Forest is one of the most widely used Machine Learning algorithms due to its robustness, interpretability, and good performance on a variety of datasets.

The construction of a Random Forest can be summarized in the following steps:

1. Draw B bootstrap samples from the original training data. Each sample is generated by sampling with replacement.
2. For each bootstrap sample, train an unpruned decision tree as follows:
   (a) At each node, randomly select m features from the total of p available features, where $m < p$.
   (b) Choose the best feature and split point among the m selected features based on a chosen impurity measure (e.g., Gini index or Entropy for Classification, variance reduction for Regression).
   (c) Grow the tree to its maximum depth without pruning.
3. Aggregate the predictions from all B trees:
   - For Classification: use majority voting.
   - For Regression: take the average of all tree outputs.

Random Forest is highly accurate and resistant to overfitting due to the averaging of multiple decision trees. It handles high-dimensional data and maintains performance even with missing values or unbalanced features. It can estimate feature importance, which helps identify the most influential variables in the model. Furthermore, it requires minimal parameter tuning and performs well with default settings.

Despite its strengths, Random Forest can be computationally expensive, especially with large datasets or a high number of trees. It requires more memory and time for both training and prediction compared to a single decision tree. Additionally, it loses interpretability because the ensemble structure makes it difficult to understand how specific predictions are made.

3.3.6 Talk about AdaBoost

AdaBoost (Adaptive Boosting) is the most popular and influential implementation of boosting. It was designed to improve the performance of simple base classifiers, often called *weak learners*, such as decision stumps (single-level decision trees).

The AdaBoost algorithm operates iteratively:

1. Initialize all instance weights equally, $w_j = \frac{1}{n}$ with n the number of tuples;
2. At iteration i, train a weak learner M_i using the weighted data;
3. Compute the Classification error of classifier M_i, that is the sum of the weights of the misclassified tuples:

$$\text{error}(M_i) = \sum_{j=1}^{n} w_j \cdot \text{error}(X_j)$$

where $\text{error}(X_j)$ is the misclassification error of tuple X_j.

4. If a tuple is correctly classified, then the weight is updated as follows:

$$w_j^{new} = w_j^{old} \cdot \frac{\text{error}(M_i)}{1 - \text{error}(M_i)}$$

   Misclassified tuples remain with the same weights;

5. Once all the weights are updated, the weights for all the tuples are normalized (divided by their total sum) so that they form a valid probability distribution;

6. The weight of classifier M_i vote is:

$$\alpha_i = \ln(\frac{1 - \text{error}(M_i)}{\text{error}(M_i)})$$

7. If binary, the final classifier is obtained as the sign of the weighted sum of the weak learners:

$$C(x) = \text{sign}\left(\sum_i \alpha_i M_i(x)\right)$$

   If it is not binary, for each class c we sum the weights of each classifier that assigned class c to X. The class with the highest sum is the "winner" and is returned as the class prediction for tuple X.

AdaBoost places increasing emphasis on examples that are difficult to classify, enabling the ensemble to progressively refine its decision boundaries. However, it can be sensitive to noisy data and outliers because the weights of misclassified samples grow exponentially.

3.3.7 How to deal with imbalanced datasets? (SMOTE)

Imbalanced datasets are a problem because most classifiers assume that classes are roughly equally represented. When one class (the majority) dominates, the model tends to favor it, leading to poor performance on the minority class, which is often the class of real interest.

We can deal with imbalanced datasets in this way:

- **Over-sampling** increases the number of minority class samples to balance the dataset. The simplest approach is **random over-sampling**, which duplicates existing minority instances until both classes are equally represented. Although effective, this method can lead to overfitting because the model may learn redundant patterns from repeated samples. For this we use **SMOTE**.
- **Under-sampling** reduces the number of majority class samples to match the minority class. The simplest method is **random under-sampling**, which removes majority instances at random. This technique decreases training time and balances the dataset but may discard useful information, potentially weakening the model's generalization ability.
- **Threshold moving** adjusts the decision threshold of the classifier rather than modifying the dataset. Instead of predicting the positive class when the probability exceeds 0.5, the threshold is lowered to favor the minority class. This approach changes the trade-off between **Precision** and **Recall** and is often tuned using metrics like the **F1-Score** or the **ROC curve**.
- **Ensemble methods** combine multiple classifiers to improve prediction on imbalanced data. Techniques such as **Bagging**, **Boosting**, and their variants are effective in reducing bias and variance. In particular, **Boosting** algorithms (e.g., AdaBoost, Gradient Boosting, or XGBoost) naturally focus on harder-to-classify samples, often improving minority class detection. **Random Forests** and **EasyEnsemble** are specialized ensemble methods that integrate sampling strategies within the learning process to maintain balance between classes.

SMOTE (**Synthetic Minority Over-sampling Technique**) is a data-level approach designed to address the problem of **imbalanced datasets**, where one class (the minority class) has far fewer instances than the other (the majority class). Instead of simply duplicating existing minority samples, SMOTE generates new, synthetic samples by interpolating between existing ones. This helps the classifier learn a more general decision boundary and prevents bias toward the majority class.

The main idea of SMOTE is to create artificial minority class instances that lie along the line segments connecting existing minority samples and their nearest neighbors in the feature space. By doing this, the algorithm increases the diversity of the minority class and reduces the imbalance ratio, improving Classification performance.

The SMOTE procedure can be described as follows:

1. For each sample x_i in the minority class, identify its k-nearest neighbors, where k is a user-defined parameter (typically $k = 5$);
2. Randomly select one or more of these neighbors, denoted as x_{zi};
3. Generate a new synthetic sample using linear interpolation between x_i and x_{zi}:

$$x_{\text{new}} = x_i + \delta \cdot (x_{zi} - x_i)$$

   where δ is a random number in the range $[0; 1]$;
4. Repeat the process until the desired level of oversampling for the minority class is achieved.

SMOTE helps balance the dataset by introducing synthetic samples rather than mere copies, which reduces overfitting that can occur with simple replication. While SMOTE improves class balance, it does not consider the distribution of the majority class, which can lead to ambiguous synthetic instances near class borders. Proper parameter tuning, such as choosing the number of neighbors k and the oversampling ratio, is essential to maximize performance and maintain realistic data structure.

3.3.8 Why do we need the Naïve assumption in the Naïve Bayesian Classifier?

The **Naïve Bayes classifier** simplifies the estimation of $P(X \mid C_i)$ by **assuming that the attributes are conditionally independent** given the class:

$$P(X \mid C_i) = \prod_{k=1}^{n} P(x_k \mid C_i)$$

Independence between attributes is necessary to make the joint probability estimable and computable, reducing the complexity from exponential to linear. If you have d attributes with n values each, the number of combinations per class is:

$$n^d$$

Without this assumption you have to calculate:

$$P(X \mid C_i) = P(x_1|x_2, x_3, ..., x_n, C_i) \cdot P(x_2|x_3, ..., x_n, C_i) \cdot ... \cdot P(x_n|C_i)$$

Chapter 4: Clustering

4.1 Introduction

4.1.1 What is Clustering?

Clustering is an **unsupervised learning** technique used to group a set of data points into clusters such that samples within the same cluster are more similar to each other than to those in other clusters. Unlike Classification, **Clustering does not rely on labeled data**. Instead, it discovers hidden patterns or structures in the dataset based on similarity or distance measures.

Formally, given a dataset $X = \{x_1, x_2, \ldots, x_n\}$, the goal of Clustering is to partition it into K clusters $C_1, C_2, \ldots, C_K$ such that:

$$C_i \cap C_j = \emptyset \quad \text{for } i \neq j, \qquad \bigcup_{i=1}^{K} C_i = X$$

and the similarity between points within each C_i is maximized, while the similarity between points in different clusters is minimized.

Clustering can be categorized into several main approaches:

- **Partitioning Methods:** These methods divide the data into a predefined number of clusters K. Each cluster is represented by a centroid or medoid, and objects are assigned to the cluster with the closest center according to a distance metric (e.g., Euclidean distance).
- **Hierarchical Methods:** These methods build a hierarchy of clusters either through a *bottom-up* (agglomerative) or *top-down* (divisive) process. In Agglomerative Clustering, each data point starts as its own cluster, and pairs of clusters are successively merged based on a linkage criterion (single, complete, or average linkage). In Divisive Clustering, all data points start in one cluster, which is then recursively split into smaller clusters using the linkage criterion. The result is often represented as a **dendrogram**.
- **Density-Based Methods:** These methods form clusters based on regions of high data density separated by regions of low density. A cluster is viewed as a dense group of points surrounded by areas of lower density.
- **Grid-Based Methods:** These methods divide the data space into a finite number of cells (a grid) and perform Clustering on the grid structure instead of the data directly. The complexity depends on the number of grid cells rather than the number of data points, making these methods computationally efficient for large datasets.

4.1.2 How to assess if a dataset has significant clusters within it?

Before applying any Clustering algorithm, it is important to determine whether the dataset actually contains a meaningful cluster structure. This process is known as **assessing the cluster tendency**. If the data points are uniformly distributed, Clustering methods may produce arbitrary groupings that do not represent real patterns.

A common approach to evaluate the presence of significant clusters is through the **Hopkins Statistic**.

The **Hopkins Statistic** is a numerical measure that quantifies the tendency of a dataset to form clusters. It compares the distances between randomly generated points and their nearest neighbors in the data to the distances between real data points and their nearest neighbors. If the dataset is highly clustered, real data points will be much closer to each other than random points. **Instead of a single value, the average Hopkins statistic H from multiple random samples is used to determine Clustering tendency.**

These are the algorithm steps:

1. Randomly select n data points from the dataset D (n must be much smaller than the cardinality of the dataset);
2. For each selected point d_i, compute the distance x_i to its nearest neighbor in D:
$$x_i = \min_{v \in D, v \neq d_i} (dist(d_i; v))$$
3. Generate a random uniform distribution with n points $q_1, \ldots, q_n$ and the same variation as the original dataset D;
4. For each random point q_i, compute the distance y_i to its nearest neighbor in D:
$$y_i = \min_{v \in D, v \neq q_i} (dist(q_i; v))$$
5. Compute the Hopkins Statistic:
$$H = \frac{\sum_{i=1}^{n} y_i}{\sum_{i=1}^{n} x_i + \sum_{i=1}^{n} y_i}$$

Hypothesis	Description
H_0 (Null Hypothesis)	The null hypothesis assumes that there aren't significant clusters.
H_1 (Alternative Hypothesis)	The alternative hypothesis assumes that there are significant clusters.

The value of H lies in the range $[0; 1]$.

$$\begin{cases} H \approx 0.5 & \text{indicates a uniform random distribution (no cluster structure)} \\ H \to 0 & \text{suggests that data points are regularly spaced (anti-clustered)} \\ H \to 1 & \text{indicates strong Clustering tendency} \end{cases}$$

In practice, a value of $H > 0.75$ it indicates a Clustering tendency at the 90% confidence level (Alternative Hypothesis). Values near 0.5 imply that Clustering algorithms may not yield meaningful results (Null Hypothesis).

4.2 Partitioning Methods

Partitioning-based Clustering algorithms group data into a predefined number of clusters by optimizing a specific criterion, typically based on distances between points and cluster centers. While **K-Means** uses centroids that may not correspond to actual data points, **medoid-based** methods select actual data points as cluster centers, providing greater robustness to noise and outliers. The main medoid-based algorithms are **PAM**, **CLARA**, and **CLARANS**.

4.2.1 Talk about K-Means

The **K-Means** algorithm is a *partitioning method* used in Clustering. Given a dataset D of n objects and a predefined number of clusters k, the goal is to partition the data into k clusters $\{C_1, C_2, \ldots, C_k\}$ such that the **sum of squared distances** between each object and the centroid of its assigned cluster is minimized.

$$E = \sum_{i=1}^{k} \sum_{p \in C_i} \|p - c_i\|^2$$

where c_i is the centroid (mean point) of cluster C_i ($|C_i|$ cardinality of cluster C_i):

$$c_i = \frac{1}{|C_i|} \sum_{p \in C_i} p$$

Each cluster is therefore represented by its centroid, which can be seen as the center of gravity of the cluster. The algorithm follows an iterative refinement procedure that alternates between assigning data points to clusters and updating centroids until convergence.

The process can be summarized in the following steps:

1. Choose the number of clusters k and initialize k centroids, often selected as random points from the dataset;
2. Assign each data point to the nearest centroid according to the chosen distance metric, usually the Euclidean distance;
3. Recompute the centroids by taking the mean of all data points assigned to each cluster;
4. Repeat the assignment and update steps until the centroids no longer change or the changes fall below a predefined threshold.

The algorithm typically **converges quickly**, though it does **not guarantee finding the global minimum** of the objective function. Its computational complexity is $O(tkn)$, where t is the number of iterations, k the number of clusters, and n the number of data points. Since in most practical applications k and t are much smaller than n, K-Means is considered an efficient algorithm suitable for large datasets.

K-Means has several **advantages**:

- Simple and computationally efficient;
- Works well on large, continuous, low-dimensional datasets;
- Produces compact and spherical clusters.

However, it also presents important **limitations**:

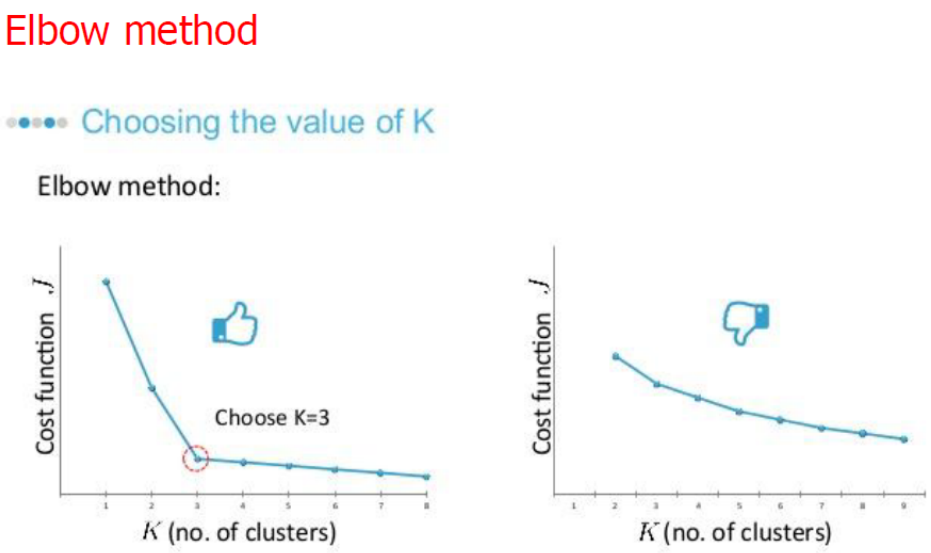
- Requires the number of clusters k to be specified in advance;
- Sensitive to outliers and noise (mean);
- Assumes convex and isotropic cluster shapes;
- Sensitive to the initial choice of centroids.

To select an appropriate number of clusters k, a common heuristic is the **Elbow Method**. This method involves running K-Means for different values of k, computing the total within-cluster variance (*the sum of squared distances*) for each configuration, and plotting the results as a function of k. As k increases, the sum of squared distances decreases, but beyond a certain point, the improvement becomes marginal. The value of k corresponding to this "elbow" point in the curve is considered a reasonable choice.

Several extensions of K-Means have been developed to address some of its limitations:

- **K-Modes**: replaces means with *modes* for categorical data, using frequency-based dissimilarity measures.
- **K-Prototypes**: handles mixed numerical and categorical data by combining distance measures.

A major issue with K-Means is its sensitivity to outliers. Objects with extremely large or small values can significantly distort the position of centroids, leading to poor Clustering results.



4.2.2 Talk about K-Medoids

The **K-Medoids** algorithm is a *partitioning Clustering method* in which each cluster is represented by an actual data object called a medoid. A **medoid** is the most centrally located object of a cluster according to the chosen dissimilarity measure. K-Medoids is introduced as a robust alternative to K-Means because replacing cluster means with medoids reduces sensitivity to outliers and extreme values.

The objective is to find a set of k medoids $\{m_1, \ldots, m_k\}$ and an associated partition $\{C_1, \ldots, C_k\}$ that minimizes the total dissimilarity between objects and their cluster medoid. Unlike K-Means, K-Medoids can be defined purely in terms of a dissimilarity matrix and thus applies to arbitrary distance functions and to non-numeric data if a dissimilarity is available.

Partitioning Around Medoids

The canonical algorithm for K-Medoids is **PAM**. PAM proceeds by choosing k representative objects as initial medoids and then iteratively improving the configuration by swapping medoids with non-medoid objects when a swap reduces the total Clustering cost. PAM relies only on the dissimilarity matrix and, at each iteration, evaluates swap moves of the form "replace current medoid O with non-medoid R" and accepts the best-improving swap. PAM is computationally expensive for large n because it examines many swap candidates.

The PAM algorithm is described by the following steps:

1. Select k objects from the dataset as initial medoids, often chosen randomly or by a heuristic;
2. Assign each object to the cluster whose medoid has minimum dissimilarity to the object;
3. For each medoid O and for each non-medoid object R, compute the total Clustering cost that would result from swapping O with R. If any swap reduces total cost, perform the best swap and go to step 2. Otherwise terminate;
4. Stop when no swap yields improvement. The remaining medoids define the final clusters.

PAM is more robust to noise and outliers than K-Means because medoids are actual data points and are less influenced by extremes. However PAM does not scale well: its worst-case per-iteration cost is high (complexity roughly on the order of $O\big(k(n-k)^2\big)$ per iteration), so PAM is practical only for small to moderate datasets.

Clustering Large Applications

CLARA was introduced to make K-Medoids applicable to larger data collections where the original PAM algorithm becomes computationally infeasible. Instead of applying PAM to the entire dataset, CLARA draws random samples of manageable size and performs Clustering only on each sample. The key idea is that if the sample is representative of the overall data distribution, the medoids obtained from the sample will approximate the medoids that would have been derived from the full dataset.

The method operates as follows:

1. For each of R repetitions, draw a random sample of objects from the dataset;
2. Apply the K-Medoids (PAM) algorithm to the sample to find k medoids;
3. Assign each object in the entire dataset to the closest medoid from this sample;
4. Compute the average dissimilarity of all objects in the full dataset relative to their assigned medoid. If this average is lower than the current minimum, store this configuration as the best one found so far;
5. After completing all repetitions, return the medoids with the lowest overall average dissimilarity as the final Clustering solution.

CLARA thus evaluates multiple candidate solutions and selects the one that minimizes the total cost and it can handle larger datasets. However, CLARA's accuracy depends on the quality and representativeness of the samples: a biased or unrepresentative sample may lead to poor Clustering results. Its computational complexity is approximately $O(ks^2 + k(n-k))$, where s is the sample size, n the number of objects, and k the number of clusters.

CLARA maintains the interpretability and robustness of medoid-based Clustering but trades some precision for efficiency. When the dataset is extremely large or heterogeneous, this method may not always capture small or rare clusters if they are not well represented in the samples.

Clustering Large Applications based on Randomized Search

CLARANS further improves the efficiency of K-Medoids by adopting a randomized graph-search approach. It views the Clustering process as searching for a minimum-cost configuration in a graph, where each node represents a potential solution (a specific set of k medoids) and two nodes are connected if they differ by exactly one medoid.

The algorithm explores this search space in a stochastic rather than exhaustive way. Instead of evaluating all possible neighbor configurations, CLARANS randomly samples a subset of neighbors at each iteration to decide whether to move to a new configuration.

The steps can be summarized as follows:

1. Randomly select an initial node, corresponding to an initial set of k medoids;
2. Randomly sample a subset of the neighboring nodes, each obtained by swapping one medoid with a non-medoid object;
3. For each sampled neighbor, compute the total Clustering cost. If a neighbor offers a lower cost than the current configuration, move to that neighbor and continue the search;
4. If no better neighbor is found after examining the sampled subset, declare the current configuration a local optimum;
5. Restart the search from a new randomly chosen node and repeat the process a predefined number of times to explore multiple local optima.

Each node's cost corresponds to the total dissimilarity between all objects and their nearest medoid. The randomized sampling of neighbors ensures that the algorithm explores the solution space efficiently without exhaustively considering all possible swaps, which would be computationally prohibitive for large datasets.

CLARANS effectively combines the depth of local search with the exploration capability of randomization. The computational complexity of CLARANS is approximately linear $O(n)$ in the number of data objects, making it significantly more scalable than PAM and CLARA. In practice, CLARANS often yields higher-quality clusters than both previous methods while maintaining reasonable computational efficiency.

Summary

- **PAM** performs an exhaustive search for optimal medoids over the entire dataset, making it accurate but computationally expensive for large datasets.
- **CLARA** applies the PAM algorithm on multiple samples of the dataset instead of the entire data. It then selects the best Clustering found among those samples. It is efficient for large datasets but can miss good clusters if samples are not representative.
- **CLARANS** improves over CLARA by performing a randomized search in the space of possible medoid swaps, rather than evaluating all or sampled subsets exhaustively. This makes CLARANS more flexible and usually finds better solutions with less computation.

4.2.3 Why does the curve surely decrease in the Elbow Method?

$$E = \sum_{i=1}^{k} \sum_{p \in C_i} \|p - c_i\|^2$$

By increasing k, it is possible to create smaller clusters. As k approaches the number of points in the dataset, E approaches 0. However, the decrease is subject to diminishing returns: the largest reductions tend to occur for the first increases in k beyond a certain point, so adding clusters brings marginal improvements.

4.3 Hierarchical Methods

Hierarchical Clustering is an unsupervised learning method that builds a hierarchy of clusters without requiring a predefined number of groups. It produces a tree-like structure called a **dendrogram**, which illustrates how data points or clusters are progressively merged or divided based on their similarity. The approach can be:

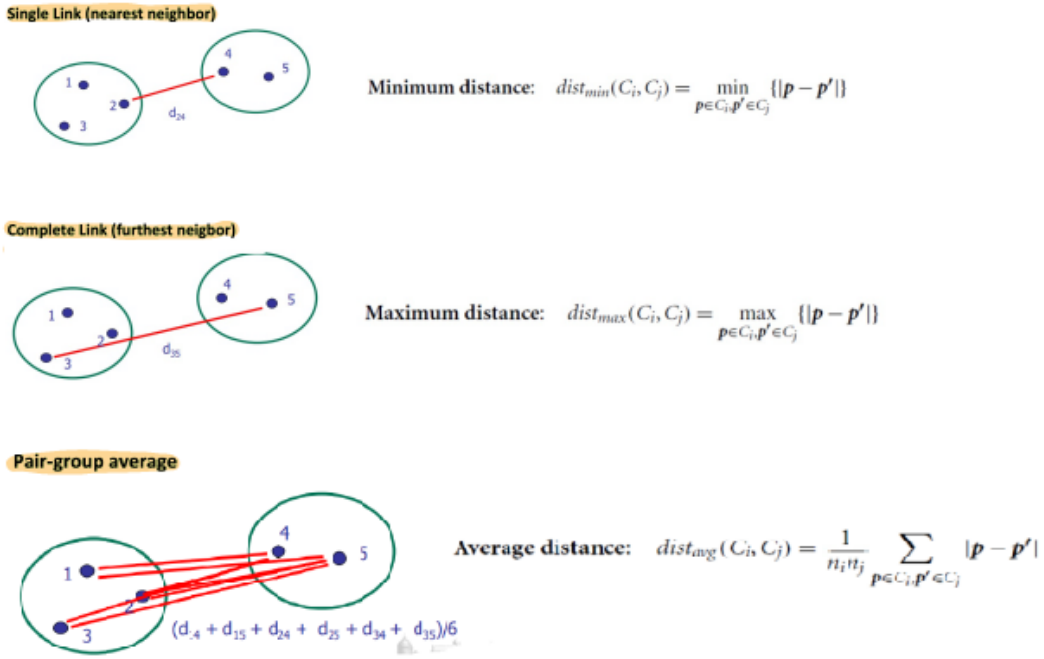
- **Agglomerative (bottom-up)**: starting with individual points and merging them into larger clusters.
- **Divisive (top-down):** starting with one large cluster and recursively splitting it.

Hierarchical Clustering is valued for its interpretability and its ability to reveal nested cluster structures within data.

4.3.1 Talk about single, complete and average linkage

Hierarchical Clustering merges or splits clusters based on distances between them, which are generalized from point-to-point distances using **linkage metrics.** These metrics determine how cluster distances are computed, influencing the hierarchy's structure and quality.

The choice among these linkage criteria significantly influences the resulting hierarchical structure. **Single linkage** prioritizes local connectivity and tends to generate clusters that are loosely connected chains of points. **Complete linkage** focuses on cluster compactness and produces tight, spherical groups. **Average linkage** balances the two, yielding results that are generally stable and representative of the overall data structure.



Single Linkage

Single Linkage (also called *nearest neighbor*) defines the distance between two clusters as the minimum distance between any pair of points belonging to the two clusters. Formally, if C_i and C_j are two clusters, then:

$$d_{SL}(C_i; C_j) = \min_{x \in C_i,\, y \in C_j} d(x; y)$$

This method tends to form long, chain-like clusters because it merges clusters based on the closest pair of elements. Consequently, single linkage is effective at discovering arbitrarily shaped clusters but can lead to poor separation when clusters are connected by narrow bridges of points. It is often referred to as the *minimum distance* criterion.

Complete Linkage

Complete Linkage (also called *farthest neighbor*) defines the distance between two clusters as the maximum distance between any pair of points from the two clusters. It can be expressed as:

$$d_{CL}(C_i; C_j) = \max_{x \in C_i,\, y \in C_j} d(x; y)$$

This approach emphasizes compactness and separation, ensuring that all points in merged clusters are close to one another. Complete linkage produces compact, roughly spherical clusters and avoids the chaining effect of single linkage. It works well when clusters are naturally well-separated and compact, but it may fail for elongated or irregularly shaped clusters.

Average Linkage

Average Linkage (also known as *pair-group average*) computes the distance between two clusters as the average of all pairwise distances between their points. If C_i and C_j contain n_i and n_j points respectively, the average linkage distance is given by:

$$d_{AL}(C_i; C_j) = \frac{1}{n_i n_j} \sum_{x \in C_i} \sum_{y \in C_j} d(x; y)$$

This method provides a compromise between single and complete linkage. It generally performs well both on compact, well-separated clusters and on clusters with more elongated shapes. Average linkage reduces sensitivity to outliers compared to complete linkage while avoiding the chaining effect typical of single linkage.

4.3.2 Talk about AGNES Algorithm

AGNES (**Agglomerative Nesting**) begins by considering each object in the dataset as an individual cluster. Then, step by step, the algorithm merges the two clusters that are most similar according to a chosen distance measure and linkage criterion, until all objects belong to a single cluster. The similarity between clusters is usually based on a **dissimilarity matrix** and on a linkage metric such as **single-linkage** (minimum distance).

The AGNES algorithm can be summarized as follows:

1. Initially, assign each data object to its own cluster, resulting in n singleton clusters;
2. Compute all pairwise distances between clusters using the selected linkage metric;
3. Merge the two clusters that are closest to each other (i.e., have the smallest inter-cluster distance);
4. Update the distance matrix to reflect the new set of clusters after the merge;
5. Repeat steps 2-4 until all objects are merged into a single cluster.

Each merging operation reduces the number of clusters at least by one. The sequence of merges can be recorded and visualized as a **dendrogram**, where the height of each merge corresponds to the distance at which the clusters were joined. Cutting the dendrogram at a specific height allows one to obtain a desired number of clusters.

It typically uses the single-link method, where the similarity between two clusters is determined by the most similar pair of data points belonging to different clusters. This variant tends to create clusters that form long chains of connected points. AGNES continues merging clusters until all data points form one unified cluster, providing a complete hierarchical structure from individual elements up to the entire dataset.

AGNES is more natural when clusters are expected to form from the **bottom up** through the progressive merging of similar items.

AGNES can never undo what was done previously and do not scale well.

4.3.3 Talk about DIANA Algorithm

DIANA (**DIvisive ANAlysis**) algorithm is a divisive hierarchical Clustering method. DIANA starts with all data points in one cluster and repeatedly splits clusters until each object is isolated or a stopping criterion is met. DIANA performs hierarchical Clustering by recursively splitting the most heterogeneous cluster based on dissimilarity (**average linkage**).

The DIANA algorithm can be outlined as follows:

1. Start with a single cluster containing all objects in the dataset;
2. Compute pairwise dissimilarities between all objects in the current clusters;
3. Select the cluster with the largest diameter (the maximum dissimilarity between any two of its objects) as the candidate for division;
4. Within this cluster, identify the object that has the highest average dissimilarity to all other objects. This object initiates a new cluster, known as the *splinter group*;
5. For each remaining object i, compute the difference:

$$D_i = \text{average dissimilarity}(i; \text{old group}) - \text{average dissimilarity}(i; \text{splinter group})$$

   If D_i is positive, object i is closer to the splinter group and is moved there. All points that have already moved to the splinter group are counted for this count;
6. Repeat the previous step from point (2) until all D_i values are negative, meaning no further reassignment is beneficial. At this point, the original cluster has been divided into two new clusters;
7. Choose the cluster with the largest diameter and repeat the procedure until every object forms its own cluster.

This process generates a hierarchical decomposition of the dataset, where the largest clusters are recursively split into smaller and more homogeneous subclusters. As with AGNES, the resulting structure can be represented as a **dendrogram** that reflects the divisions made at each level.

DIANA is useful when starting from an overall view of the data and progressively identifying finer subgroups by splitting large, heterogeneous clusters. While more computationally demanding, it provides a clear **top-down** hierarchical structure that can be useful for exploratory data analysis where the aim is to successively reveal substructures within large groups.

DIANA can never undo what was done previously and do not scale well.

4.3.4 How does BIRCH work?

The **BIRCH** algorithm (**Balanced Iterative Reducing and Clustering using Hierarchies**) is a *hierarchical agglomerative Clustering method* **specifically designed to efficiently handle very large datasets**. Unlike traditional Clustering algorithms that require all data points to be loaded into memory and compared pairwise, BIRCH incrementally and dynamically clusters incoming data points using a compact summary structure known as the **CF Tree** (*Clustering Feature Tree*). This design allows BIRCH to scale well while maintaining good Clustering quality.

BIRCH introduces the concept of a **Clustering Feature (CF)**, which is a concise representation of a subcluster. A CF for a cluster containing N points $\{x_1, x_2, \ldots, x_N\}$ in a d-dimensional space is defined as a triple:

$$CF = (N, LS, SS)$$

where:

$$LS = \sum_{i=1}^{N} x_i \qquad SS = \sum_{i=1}^{N} x_i^2$$

Here, N is the number of data points in the cluster, LS (**linear sum**) is the vector sum of all points, and SS (**square sum**) is the sum of squared values of all points. These three quantities allow efficient computation of essential cluster statistics such as centroid, radius, and diameter:

$$\text{Centroid: } x_0 = \frac{LS}{N} \qquad \text{Radius: } R = \sqrt{\frac{SS}{N} - \left(\frac{LS}{N}\right)^2} \qquad \text{Diameter: } D = \sqrt{\frac{2 \cdot (N \cdot SS - LS^2)}{N \cdot (N-1)}}$$

$$R = \sqrt{\frac{\sum_{i=1}^{N}(x_i - x_0)^2}{N}} = \sqrt{\frac{SS - 2\left(\frac{LS}{N}\right)LS + N\left(\frac{LS}{N}\right)^2}{N}} = \sqrt{\frac{SS - \frac{2LS^2}{N} + \frac{LS^2}{N}}{N}} = \sqrt{\frac{SS - \frac{LS^2}{N}}{N}} = \sqrt{\frac{SS}{N} - \left(\frac{LS}{N}\right)^2}$$

$$\begin{aligned} D &= \sqrt{\frac{\sum_{i=1}^{N}\sum_{j=1}^{N}(x_i - x_j)^2}{N(N-1)}} = \sqrt{\frac{\sum_{i=1}^{N}\sum_{j=1}^{N}(x_i^2 - 2x_ix_j + x_j^2)}{N(N-1)}} = \\ &= \sqrt{\frac{\sum_{i=1}^{N}\sum_{j=1}^{N}x_i^2 - 2\sum_{i=1}^{N}\sum_{j=1}^{N}x_ix_j + \sum_{i=1}^{N}\sum_{j=1}^{N}x_j^2}{N(N-1)}} = \\ &= \sqrt{\frac{N \cdot SS - 2 \cdot LS \cdot LS + N \cdot SS}{N(N-1)}} = \sqrt{\frac{2 \cdot (N \cdot SS - LS^2)}{N(N-1)}} \end{aligned}$$

The CF representation is additive, meaning that merging two clusters can be done simply by summing their CFs:

$$CF_{C1+C2} = (N_1 + N_2, LS_1 + LS_2, SS_1 + SS_2)$$

This property allows BIRCH to incrementally build and update its hierarchical structure without revisiting the raw data points.

The **CF Tree** is a height-balanced tree that stores CF entries in its nodes. Each non-leaf node contains entries that summarize its children, while each leaf node contains CFs that represent clusters of data points. The CF Tree is controlled by three **parameters**:

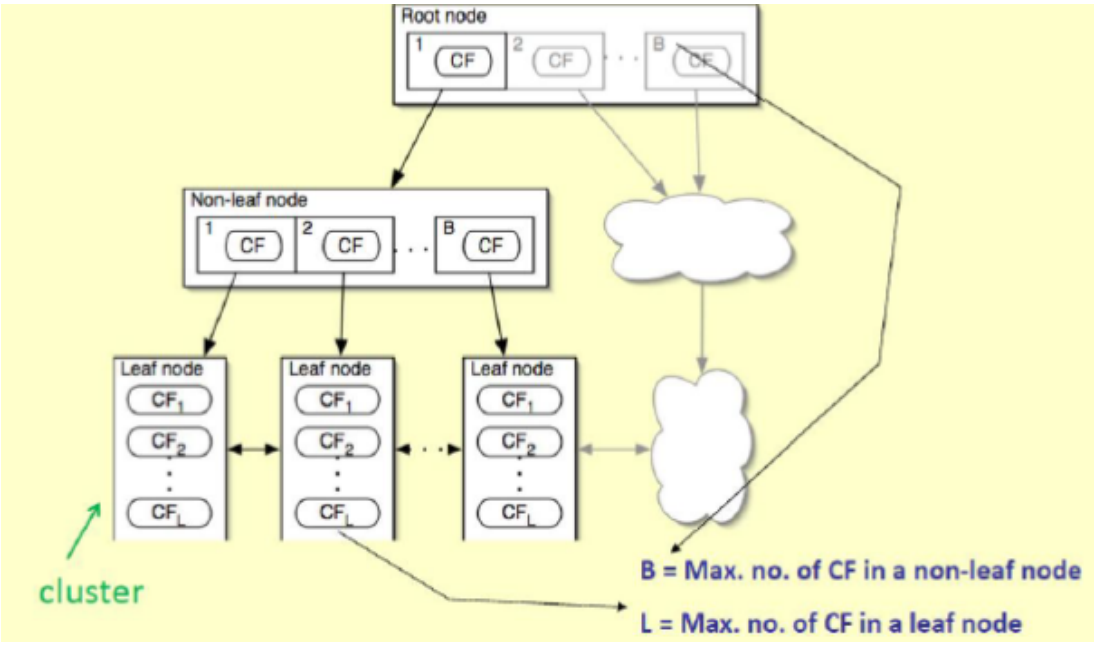
- The **branching factor** B is maximum number of children for a **non**-leaf node;
- The **threshold** T, which determines the maximum diameter D of a subcluster represented by each leaf entry;
- The **maximum** number of entries for a **leaf** node L.

If inserting a new point would cause the diameter of a leaf entry to exceed T, a new entry is created, and node splits may occur, preserving the balanced structure.

The general operation of BIRCH proceeds in four main phases:

- **Phase 1: CF Tree Construction.** Data points are scanned sequentially, and each point is inserted into the appropriate leaf node of the CF Tree based on proximity to existing subclusters. If necessary, the tree is adjusted through node splits. The result is a compact summary of the data stored in the CF Tree. In this fase we can handle outliers (micro-clusters with very small N and very distant from other micro-clusters).
- **Phase 2: Condensation.** If the initial CF Tree is too large to fit in memory, BIRCH can rebuild a smaller tree by condensing existing clusters while retaining the essential structure.
- **Phase 3: Global Clustering.** The leaf entries of the CF Tree (each representing a subcluster) are treated as inputs for a Global Clustering algorithm, such as AGNES or DIANA. This phase refines the cluster structure using the compact summaries. Tree structure depends on insertion order, and Global Clustering merges related micro-clusters to correct this.
- **Phase 4: Optional Refinement.** Finally, an optional pass over the original dataset can be performed to assign each data point to its nearest final cluster centroid (*to label the data points*), improving accuracy. If a point is too far from all macro-cluster centroids (beyond a certain safety distance), it is definitively labeled as an outlier.

BIRCH is designed for large-scale and streaming data scenarios. It only requires a single data scan to produce an initial summary and can improve the result with additional passes if needed. Its computational complexity is approximately linear in the number of data points, $O(n)$, assuming the tree fits in memory.



Some **observations**:

- A leaf node represents a Cluster;
- CF in a father node is the sum of the CF in its child nodes;
- A point is inserted in a leaf node (a cluster) which is close to;
- Inserted items must satisfy T; the cluster updates its CF and all ancestor CFs. If the node is full, it is split;
- If a new entry makes a leaf exceed L, the leaf is split into two using the two most distant entries as seeds. If the parent exceeds B, the split propagates upward to the root.

BIRCH offers several **advantages**:

- Highly efficient for large datasets due to incremental and single-scan Clustering;
- Compact representation of data using the CF Tree, reducing memory usage;
- Supports dynamic data insertion without recomputing previous results;
- Can be combined with other Clustering algorithms for the refinement phase.

However, BIRCH also has some important **limitations**:

- Sensitive to the order of input data, since incremental insertion affects tree structure;
- The effectiveness depends heavily on the choice of threshold T and branching factor B;
- Performs best on numerical data and may not handle categorical attributes directly;
- Less effective when clusters are not compact or have irregular shapes.

4.3.5 Talk about CHAMELEON

The **CHAMELEON** algorithm is a *hierarchical Clustering method* that uses a dynamic model to determine the similarity between clusters. Unlike traditional algorithms that rely on static distance measures or linkage criteria, CHAMELEON adapts the merging strategy based on both the **interconnectivity** and the **relative closeness** of clusters. This dynamic approach allows the algorithm to identify clusters of arbitrary shapes, densities, and sizes, providing a more flexible and realistic Clustering of complex data.

The core idea is that two clusters should be merged only if they are not only close to each other but also well connected relative to their internal connectivity. Therefore, it is not enough for two clusters to be absolutely close, but they must be close in a way that is consistent with their density and structure. This principle enables CHAMELEON to dynamically adapt to the intrinsic characteristics of the dataset.

The algorithm operates in two main phases:

1. **Phase 1: Graph Partitioning.** The dataset is modeled as a *k-Nearest Neighbor (k-NN)* graph, where each vertex represents a data point and edges connect each point to its k nearest neighbors according to a distance measure. Edges are weighted based on the similarity between connected points. The graph is then partitioned into a large number of relatively small subclusters using a graph partitioning algorithm that *minimizes edge weights between partitions*. Each partition represents a group of tightly connected points that can serve as the building blocks for higher-level clusters.

2. **Phase 2: Dynamic Cluster Merging.** In this phase, CHAMELEON iteratively merges subclusters based on a *dynamic modeling* of similarity. The decision to merge two clusters depends on two quantitative criteria:

   - **Relative Interconnectivity (RI):** measures the strength of connections between two clusters compared to the internal connectivity within each cluster. It is defined as the ratio between the total weight of the edges connecting the two clusters and the average internal connectivity of the clusters.
   - **Relative Closeness (RC):** measures how close two clusters are (based on *similarity*), normalized by their internal compactness. It doesn't look at the absolute distance, but rather the distance relative to the cluster density.

   The merging criterion is based on combining these two measures. Two clusters are merged if both the relative interconnectivity and the relative closeness between them are sufficiently high (above T_{RI} and T_{RC} respectively). This ensures that only clusters that are both well connected and close relative to their internal structure are joined.

Mathematically, if $E(C_i; C_j)$ denotes the sum of the edge weights between clusters C_i and C_j, and $E(C_i)$ represents the sum of internal edge weights within cluster C_i. $\overline{S_{EC}}(C_i; C_j)$ is the average weight of the edges that connect vertices in C_i to vertices in C_j, and $\overline{S_{EC}}(C_i)$ is the average weight of the edges that belong to the min-cut bisector of cluster C_i. Then the relative interconnectivity (RI) and relative closeness (RC) can be expressed as:

$$RI(C_i; C_j) = \frac{E(C_i; C_j)}{\frac{E(C_i)+E(C_j)}{2}}$$

$$RC(C_i; C_j) = \frac{\overline{S_{EC}}(C_i; C_j)}{\frac{|C_i|}{|C_i|+|C_j|}\overline{S_{EC}}(C_i) + \frac{|C_j|}{|C_i|+|C_j|}\overline{S_{EC}}(C_j)}$$

These measures are normalized to the internal properties of the clusters, allowing CHAMELEON to adapt its merging behavior to datasets with clusters of varying densities and shapes.

The merging process continues until no pair of clusters satisfies the merging conditions. The final Clustering result captures both the local connectivity of the data and the relative proximity of clusters, yielding a partition that reflects the true structure of the data distribution.

CHAMELEON offers several **advantages**:

- Adapts dynamically to the internal characteristics of clusters, allowing discovery of clusters with varying densities, sizes, and shapes;
- Utilizes both interconnectivity and closeness for merging, producing more accurate and natural groupings;
- Graph-based representation captures local relationships among data points effectively;
- Produces results that are robust to differences in cluster geometry and scale.

However, it also presents some **limitations**:

- Computationally more expensive than traditional hierarchical methods due to graph construction and dynamic similarity calculations $O(n^2)$;
- Requires the number of nearest neighbors k and the partitioning granularity to be appropriately chosen;
- Sensitive to the quality of the initial graph partitioning step;
- Not ideal for extremely large datasets without further optimization or sampling.

4.4 Density-Based Methods

Density-based Clustering methods identify clusters as regions of high data density separated by areas of low density. Unlike partitioning or hierarchical approaches, these methods can discover clusters of arbitrary shape and effectively handle noise or outliers. A cluster is defined as a dense group of points that are reachable from one another according to a specified distance and density threshold. Algorithms such as **DBSCAN**, **OPTICS**, and **DENCLUE** are well-known examples, widely used for spatial data analysis and pattern discovery in complex datasets.

They are based on two parameters:

- ε**:** Maximum radius of the neighbourhood;
- MinPts: Minimum number of points in an ε-neighbourhood of that point.

4.4.1 Talk about relationships in Density-Based Clustering

A central aspect of *Density-Based Clustering* is the definition of relationships among points in terms of **density reachability**. These relationships determine how clusters are formed as connected regions of high density and are based on two fundamental notions: **directly density-reachable** and **density-connected** points. They are defined with respect to two key parameters: the radius ε (epsilon), which determines the neighborhood size, and the minimum number of points *MinPts*, which specifies the minimum density threshold.

Epsilon-Neighborhood

Given a point p, its ε-neighborhood, denoted as $N_\varepsilon(p)$, is the set of all points whose distance from p does not exceed ε:

$$N_\varepsilon(p) = \{q \in D \mid \text{dist}(p; q) \leq \varepsilon\}$$

where D is the dataset and $\text{dist}(p; q)$ is the distance measure, usually the Euclidean distance.

Core, Border and Noise Points

- A point p is a **core point** if $|N_\varepsilon(p)| \geq \mathit{MinPts}$, meaning it lies within a dense region.
- A point q is a **border point** if it is not a core point itself, but lies within the ε-neighborhood of a core point.
- A point is considered **noise** (or outlier) if it is neither a core point nor within the ε-neighborhood of any core point.

Directly Density-Reachable

A point q is said to be *directly density-reachable* from a point p with respect to parameters $(\varepsilon; \mathit{MinPts})$ if:

$$q \in N_\varepsilon(p) \quad \text{and} \quad |N_\varepsilon(p)| \geq \mathit{MinPts}$$

That is, q must lie within the ε-neighborhood of p, and p must be a core point. This relationship is asymmetric, since q may not be a core point and therefore might not directly reach p in return.

Density-Reachable

A point q is said to be *density-reachable* from a point p with respect to $(\varepsilon; \mathit{MinPts})$ if there exists a chain of points $p_1, p_2, \ldots, p_n$ such that:

$$p_1 = p, \quad p_n = q, \quad \text{and} \quad p_{i+1} \text{ is directly density-reachable from } p_i \text{ for all } i = 1, \ldots, n-1$$

In other words, there must exist a sequence of directly density-reachable connections linking p to q. Unlike the "directly density-reachable" relation, the "density-reachable" relation is transitive but not symmetric.

Density-Connected

Two points p and q are said to be *density-connected* with respect to $(\varepsilon; \mathit{MinPts})$ if there exists a point o such that both p and q are density-reachable from o. Formally:

$$\exists o \in D \text{ such that } p \text{ and } q \text{ are density-reachable from } o$$

This relation is symmetric and forms the basis for defining clusters in density-based methods.

Cluster Definition

A cluster in a density-based framework is defined as a non-empty subset of points satisfying two properties:

1. *Maximality:* For any two points p and q, if p belongs to the cluster and q is density-reachable from p, then q also belongs to the cluster.
2. *Connectivity:* Any two points in the cluster are density-connected.

Points that are not part of any cluster are labeled as noise.

4.4.2 Talk about DBSCAN

Algorithm

DBSCAN (**Density-Based Spatial Clustering of Applications with Noise**) is a *density-based Clustering algorithm* that defines clusters as maximal sets of density-connected points and treats low-density points as noise.

DBSCAN uses *two parameters*. The first is ε (epsilon), the radius that defines a point's neighborhood. The second is *MinPts*, the minimum number of points required within an ε-neighborhood for a point to be considered a core point. A point with at least *MinPts* points in its ε-neighborhood is a core point. Points within the ε-neighborhood of a core point but not core themselves are border points. Points that are neither core nor border are noise.

Two fundamental relations underpin DBSCAN. A point q is *directly density-reachable* from point p (w.r.t.[1] $(\varepsilon; \mathit{MinPts})$) if $q \in N_\varepsilon(p)$ and p is a core point. A point q is *density-reachable* from p if there exists a chain of points $p = p_1, p_2, \ldots, p_n = q$ where each p_{i+1} is directly density-reachable from p_i. Two points p and q are *density-connected* if there exists an o such that both p and q are density-reachable from o. These relations are used to grow clusters from core points.

The DBSCAN algorithm proceeds as follows:

1. Marks all objects in D as "*unvisited*";
2. Randomly selects an "*unvisited*" object p and:
   (a) Marks it as "*visited*";
   (b) Checks whether the ε-neighborhood (denoted as $N_\varepsilon(p)$) contains at least *MinPts*. If so, mark p as a *Core Point* and create a new cluster C containing p and then proceed to the next step (3), otherwise it is marked as "*Noise Points*".
3. Expand cluster C by iteratively adding all points that are density-reachable from p. For each point q in $N_\varepsilon(p)$:
   - If q is "*unvisited*", mark it as "*visited*" and compute $N_\varepsilon(q)$;
   - If $|N_\varepsilon(q)| \geq \mathit{MinPts}$, merge its neighborhood into C. If q is not already assigned to any cluster, add it to C;
   - When $N_\varepsilon(p)$ has no object to check, C is completed and returned in output.
4. Repeat the process from point (2) until all points have been "*visited*";
5. If all points have been "*visited*" the algorithm ends.

A cluster in DBSCAN is defined as a maximal set of density-connected points, with sparse regions serving as natural boundaries that prevent the merging of distinct clusters. The algorithm also detects noise points that do not belong to any cluster. Although its worst-case computational complexity is $O(n^2)$, the use of spatial indexing structures such as R*-trees or k-d trees can reduce this to approximately $O(n \log n)$ for low-dimensional data, depending on the efficiency of neighborhood searches and data dimensionality.

[1] with respect to

Advantages and Disadvantages

DBSCAN offers several **advantages**:

- Detects clusters of arbitrary shape, without assuming spherical or convex forms;
- Does not require the number of clusters to be specified in advance;
- Effectively identifies noise and outliers as points not belonging to any dense region;
- Automatically distinguishes dense regions separated by sparse areas, adapting well to irregular data distributions.

However, it also presents some **limitations**:

- Strongly dependent on the selection of parameters ε and *MinPts*, which can be difficult to determine in practice;
- Struggles with datasets containing clusters of varying densities, since a single global ε may not fit all regions;
- Performance and accuracy degrade in high-dimensional data spaces where distance measures become less meaningful;
- Requires efficient spatial indexing to achieve acceptable runtime on large datasets.

How to determine ε and *MinPts*?

Selecting suitable values for ε and *MinPts* is critical for the success of the DBSCAN algorithm. A widely-used *rule of thumb* for *MinPts* is:

$$MinPts \geq M + 1$$

where M is the number of dimensions (features) of the data. In noisier or large datasets it may be more appropriate to use $MinPts = 2\,M$ or higher.

Once *MinPts* has been settled, an effective method for choosing ε is to use a **heuristic approach**. We construct a k-distance graph, where $k = MinPts$. For each point compute the distance to its k-th nearest neighbour, sort the points based on their distances in descending order, and then identify the "elbow", or "knee", or "valley" point in the curve. The corresponding distance value is a good candidate for ε.

The limitation is that this heuristic identifies a global density. In many real-world datasets, different clusters have varying densities. If ε is set too small, many points may be labelled as noise. If it is too large, clusters may merge inappropriately.

What can we guarantee in terms of clusters after applying DBSCAN?

We can guarantee that **each cluster is a maximal set of density-connected points**. For any two points in the same cluster, there exists a chain of points where each consecutive pair is within distance ε and satisfies the density condition. Every cluster contains at least one core point, and around each core point there are at least $MinPts$ points. **In practice all the points inside the cluster are density connected.**

DBSCAN can recover clusters of arbitrary geometry as long as they are density-connected. One of the strengths of DBSCAN is that it is not restricted to convex or spherical shapes. So it **can find concave (arbitrary) shapes**. But that creates a problem for traditional evaluation tools, like **Silhouette Coefficient**, specifically designed to measure the quality of convex clusters.

Points that are not **density-reachable from any core point** are labeled as **noise** (**outliers**). Noise points are guaranteed not to belong to any cluster.

4.4.3 Talk about OPTICS

OPTICS (**Ordering Points To Identify the Clustering Structure**) is a *density-based Clustering algorithm* that extends the principles of DBSCAN to overcome its sensitivity to parameter selection, particularly the choice of ε. While DBSCAN produces a single Clustering result based on a fixed density threshold, OPTICS generates an augmented ordering of the database that captures its intrinsic density-based structure at multiple density levels. This allows OPTICS to identify clusters of varying densities without requiring a global ε parameter.

The main idea of OPTICS is to process the dataset in a way that preserves the order in which points are visited during a density-based exploration. For each point, the algorithm records two key quantities that describe local density relationships: the **core distance** and the **reachability distance**. These two measures allow OPTICS to represent the density structure of the dataset through a visual and analyzable *reachability plot*, from which clusters can be extracted at different density levels.

The **core distance** of a point p is defined as the smallest ε' such that the ε'-neighborhood of p contains at least *MinPts* points. Formally:

$$\text{core\_dist}(p) = \begin{cases} \text{distance to the } \mathit{MinPts}^{th} \text{ nearest neighbor of } p & \text{if } |N_\varepsilon(p)| \geq \mathit{MinPts} \\ \text{undefined} & \text{otherwise} \end{cases}$$

If the core distance is undefined, the point is not dense enough to be considered a core point.

The **reachability distance** quantifies how densely one point is reachable from another core point. For a point p and its neighbor o, the reachability distance is defined as:

$$\text{reach-dist}(p; o) = \max(\text{core\_dist}(p);\ \text{dist}(p; o))$$

This means that the reachability distance reflects both the local density around p (via its core distance) and the distance between p and o.

The OPTICS algorithm works as follows:

1. Initialize all points in the dataset as *unvisited* and set their reachability distance to *undefined*;
2. Select an unprocessed point p and compute its ε-neighborhood;
3. Mark p as *visited* and record its core distance;
4. If p is a core point (its core distance is defined and below $\varepsilon_{\max}$):
   (a) For each unprocessed neighbor o of p, update its reachability distance:
   $$\text{reach-dist}(o; p) = \max(\text{core\_dist}(p);\ \text{dist}(p; o))$$
   (b) Insert all neighbors into a priority queue (the **order seeds**) sorted by their current reachability distances. If it is already in the queue, update the reachability-distance only if it is smaller;
   (c) Extract the point with the smallest reachability distance from the queue and repeat the process (point 2) until the queue is empty.
5. When the queue is empty, select the next unprocessed point and repeat the procedure until all points have been processed.

The output of OPTICS is not an explicit partitioning of the data into clusters but rather an **ordering of points** that reflects their density-based structure. This ordering can be visualized through the **reachability plot**, where the x-axis represents the order in which points were processed and the y-axis shows the reachability distance. Valleys in the reachability plot correspond to dense clusters, while peaks represent sparse regions separating clusters. By selecting appropriate reachability thresholds, different Clusterings can be extracted from the same plot.

OPTICS introduces a key advantage over DBSCAN by avoiding the need for a single global ε. Instead, it uses a **large maximum value** $\varepsilon_{\max}$ that acts as an upper bound for density exploration, while automatically adapting to local density variations. This makes OPTICS capable of revealing clusters with varying densities that DBSCAN would otherwise merge or fragment.

The computational complexity of OPTICS is typically $O(n \log n)$ when spatial indexing structures (e.g., R*-trees or k-d trees) are used, similar to DBSCAN, though with a slightly higher constant factor due to the maintenance of the priority queue. In the worst case the complexity is $O(n^2)$.

OPTICS offers several **advantages**:

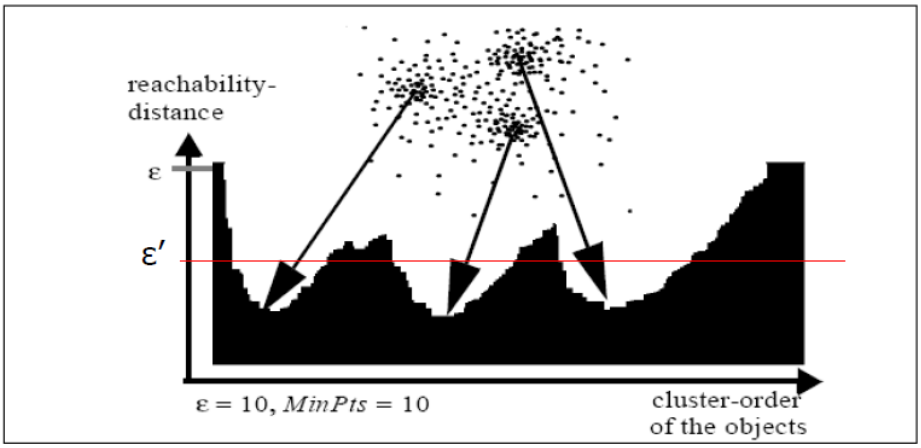
- Identifies clusters of varying densities without requiring a global ε threshold;
- Produces an ordering of points that captures the full density-based structure of the data;
- Automatically detects noise and cluster boundaries through the reachability plot;
- Retains the strengths of DBSCAN in detecting arbitrarily shaped clusters and managing noise.

However, it also presents some **limitations**:

- The interpretation of the reachability plot may require manual analysis to extract meaningful clusters;
- The algorithm introduces additional computational overhead compared to DBSCAN due to the maintenance of the priority queue and ordering structure;
- Parameter *MinPts* still needs to be specified and influences the density estimation;
- The algorithm's performance may decrease in very high-dimensional spaces where distance metrics become less informative.

Draw what we can obtain as output of the OPTICS method

We obtain a **Cluster-Ordering** (a list of objects in the order they were processed) and **two values per object** (each object is assigned a core-distance and a reachability-distance). So we can design a **Reachability Plot**.



The x-axis is the cluster-order and the y-axis is the reachability-distance.

Deep "*valleys*" in the plot represent clusters. The deeper and wider the valley, the more significant the cluster.

High peaks represent regions of low density that separate different clusters.

The *tall and flat regions* are noise or sparse clusters.

4.4.4 *Talk about DENCLUE*

DENCLUE (**DENsity-based Clustering**) is a *density-based Clustering algorithm* that uses mathematical density functions to model the overall distribution of data. Unlike DBSCAN, which relies on discrete neighborhoods and parameter thresholds, DENCLUE represents data density as a continuous function defined by the influence of each data point, allowing it to detect clusters more precisely and handle noise effectively.

DENCLUE assumes that the data space $\mathbb{R}^d$ contains a set of points $\{x_1, x_2, \ldots, x_n\}$. Each data point contributes to the overall density of the space through an **influence function**, which is typically a **kernel function** that decreases with distance. The most common choice is the **Gaussian kernel**, defined as:

$$K(\frac{x - x_i}{h}) = \frac{1}{\sqrt{2\pi}} e^{-\frac{||x-x_j||^2}{2h^2}}$$

where h is the **bandwidth parameter**, controlling the width of the influence region and therefore the smoothness of the density function. A larger h produces smoother density estimates, while smaller h values allow finer resolution of local structures. A kernel function must respect: $K(u) = K(-u)$ and $\int_{-\infty}^{+\infty} K(u)du = 1$.

The **overall density function** is the sum of the influence functions of all data points:

$$f(x) = \frac{1}{nh} \sum_{i=1}^{n} K(\frac{x - x_i}{h}) = \frac{1}{\sqrt{2\pi}h} \sum_{i=1}^{n} e^{-\frac{||x-x_j||^2}{2h^2}}$$

$1/n$ normalizes the n bells to a total probability of 1, while $1/h$ ensures each bell has unit area regardless of its width. This function expresses the estimated density at any point x in the data space. Clusters correspond to regions of high density, typically associated with local maxima of $f(x)$, called **density attractors**. Each data point can be associated with the nearest density attractor by following the gradient of the density function.

The algorithm identifies clusters through an iterative process known as **hill climbing**, in which each point moves iteratively in the direction of the gradient $\nabla f(x)$ until it converges to a local density maximum. Points that converge to the same attractor are assigned to the same cluster. Formally, the gradient ascent step updates the position of each point as follows (δ parameter to control the speed of the convergence):

$$x^{(0)} = x_i \qquad x^{(t+1)} = x^{(t)} + \delta \cdot \frac{\nabla f(x^{(t)})}{|\nabla f(x^{(t)})|}$$

Iterations continue until convergence, that is, until $x^{(t+1)} \approx x^{(t)}$. The resulting stationary points represent the centers of dense regions in the data space.

After identifying all local density attractors, the algorithm groups them into clusters according to their proximity and density values. Two attractors belong to the same cluster if they are connected through a sequence of points whose density values remain above a certain threshold ξ. This threshold determines whether two dense regions should be merged or treated as separate clusters.

The main parameters of DENCLUE are:

- The **influence factor** h, which determines the influence of a data point in its neighborhood (often related to the smoothness of the density estimation);
- The **density threshold** ξ, which describes whether a density-attractor is significant and helps reduce the number of trivial attractors.

The DENCLUE algorithm can be summarized as follows:

1. Use a pre-Clustering grid: accelerate density calculations and hill-climbing by partitioning the space into $2h$ hypercubes, keeping only populated ones. Storing cube means and counts, while connecting adjacent cubes, significantly reduces search costs;
2. Compute the influence function $K(\cdot)$ for each data point in highly populated cubes ($N_c \geq \xi_c$ and those connected to them) using a selected kernel (e.g., Gaussian);
3. Combine all influence functions to obtain the overall density function $f(x)$;
4. For each data point (as above), apply the *hill climbing procedure* to find the local maximum, known as a *density attractor*;
5. Form clusters around density attractors x^* that satisfy $f(x^*) \geq \xi$;
6. For arbitrary shaped clusters, merge density attractors that are connected through paths where every point on the path has a density $\geq \xi$;
7. Mark an object as *noise* if it converges to an attractor x^* with a density value $f(x^*) < \xi$.

The parameter h is commonly selected by varying it over a suitable range and analyzing the number of density attractors $m(h)$. A stable interval $[h_{\min}; h_{\max}]$ where $m(h)$ remains constant indicates a robust choice of scale, and h can be chosen within this range. Once h is fixed, ξ is determined by inspecting the distribution of attractor densities $f(x^*)$ and separating low-density spurious maxima from significant ones. In practice, the two parameters are chosen jointly by first identifying a stable h and then selecting ξ at that scale.

DENCLUE offers several **advantages**:

- Provides a mathematically sound and continuous definition of clusters based on density functions;
- Can identify clusters of arbitrary shape and varying densities;
- Handles noise and outliers effectively by applying a density threshold;
- Scales efficiently to large datasets when implemented with grid-based or tree-based indexing structures;
- Suitable for high-dimensional spaces due to its analytical formulation and kernel-based approach.

However, it also presents some **limitations**:

- Requires careful tuning of parameters h and ξ, which significantly affect Clustering quality;
- Sensitive to the choice of kernel function and smoothing parameter, which can alter the detected density structure;
- Gradient ascent (hill climbing) may converge to local maxima, requiring proper initialization or smoothing;
- Computational cost increases for very large datasets without indexing optimizations.

4.5 Grid-Based Methods

Grid-Based Methods divide the data space into a finite number of cells that form a grid structure.

Clustering operations are then performed on the grid cells rather than on individual data points, significantly reducing computational complexity. Each cell is associated with a density value, typically representing the number of data points it contains, and clusters are formed by grouping adjacent dense cells.

Unlike density-based methods such as DBSCAN, which rely on distance calculations between points, grid-based algorithms work by quantizing the data space into a fixed-resolution grid. This allows them to achieve high efficiency, especially with large datasets and multidimensional data. The performance of these methods depends primarily on the number of grid cells rather than the number of data objects.

4.5.1 Talk about STING

STING (**STatistical INformation Grid**) is a *grid-based Clustering algorithm* that uses a hierarchical structure of rectangular cells to efficiently perform spatial Clustering and analysis. Instead of working directly on the raw data, STING divides the spatial area into a predefined number of grid cells and summarizes the statistical properties of the data within each cell. Clustering and query processing are then performed using these statistical summaries, which makes the algorithm highly scalable and efficient for large spatial datasets. It uses a **top-down approach** to answer spatial data queries, but its structure is built in a **bottom-up** way.

STING represents spatial data hierarchically across multiple levels of resolution. At the lowest level, each cell records statistical information about the data objects it contains, while higher levels aggregate this information to produce coarser summaries of the data distribution. This hierarchical structure enables efficient processing of spatial queries and cluster detection by examining only a subset of grid cells instead of all data points. Each cell stores statistical parameters such as the number of objects, mean, standard deviation, minimum and maximum values, and an estimated distribution type. These parameters are computed during preprocessing and propagated upward through the hierarchy to provide summarized information at higher levels.

The STING algorithm proceeds in several stages:

1. **Grid Construction:** The spatial area is divided into rectangular cells that form the lowest level of the grid hierarchy. Cells at higher levels are created by merging groups of adjacent lower-level cells;
2. **Statistical Computation:** For each cell at every level, statistical parameters such as **count**, **mean**, **standard deviation**, **minimum**, **maximum** and **distribution type** are computed and stored. Higher-level cell statistics are derived from their subcells, ensuring that no full data scan is required once summaries are built;

3. **Top-Down Evaluation:** For a given query or Clustering task, STING starts at a high (coarse) level of the hierarchy and examines the cells to determine which are likely to contain relevant data (e.g., high density). Cells that do not satisfy the desired conditions are pruned early. Cells that meet the conditions are further examined at finer levels until a satisfactory level of detail is reached;

4. **Cluster Formation:** Adjacent dense cells at the appropriate resolution level are combined to form clusters. The boundaries of clusters are aligned with cell borders.

The hierarchical and statistical design of STING enables efficient processing of large spatial datasets by avoiding repeated scans of the raw data. Once grid summaries are built, Clustering and query operations can be executed rapidly using only the stored cell statistics. The algorithm's computational complexity is linear with respect to the number of grid cells, $O(g)$, rather than the number of data points. Because the number of cells is usually much smaller than the total number of objects, STING achieves substantial reductions in computational cost.

STING offers several **advantages**:

- Highly efficient and scalable for large spatial databases due to grid summarization;
- Requires only one data scan during preprocessing, as subsequent computations rely on cell statistics;
- Supports multi-resolution analysis, allowing both coarse and fine-grained exploration;
- Can handle various types of spatial queries such as region queries, range queries, and Clustering queries;
- Robust to noise since decisions are based on aggregated statistical information rather than individual data points.

However, it also presents some **limitations**:

- The quality of clusters depends strongly on the granularity and alignment of the grid structure;
- Cluster boundaries are axis-aligned and may not accurately represent arbitrarily shaped clusters;
- Requires predefined grid parameters, which may not adapt well to datasets with varying density distributions;
- Performs less effectively when clusters are small compared to the grid cell size or when the data distribution is highly irregular.

4.5.2 Talk about CLIQUE

CLIQUE (**Clustering In QUEst**) is a *grid-based and subspace Clustering algorithm* designed to identify clusters in high-dimensional data. Unlike traditional Clustering methods that operate in the full-dimensional space, CLIQUE automatically discovers clusters that exist only in specific subspaces, where subsets of dimensions exhibit dense regions of data. This makes CLIQUE particularly suitable for high-dimensional datasets, where clusters may not be visible in the full space due to the **curse of dimensionality**.

CLIQUE partitions each dimension of the data space into equal-width intervals, creating a multidimensional grid where each data point is assigned to a specific cell based on its attribute values. A unit is classified as dense when the proportion of data points it contains exceeds a predefined threshold τ, and clusters are formed as maximal connected sets of dense units within subspaces. By utilizing the Apriori (monotonicity) property (see chapter 5) of density, which states that if a lower-dimensional unit is not dense then none of its higher-dimensional extensions can be dense, CLIQUE efficiently reduces the search space. This approach enables the algorithm to identify dense subspaces without exhaustively examining the entire multidimensional grid.

The CLIQUE algorithm proceeds as follows:

1. **Discretization of the Data Space:** Each dimension is divided into ξ equal-length intervals, partitioning the data space into non-overlapping rectangular units;

2. **Identification of Dense Units in 1-D Subspaces:** The algorithm scans the data once to count the number of points in each 1-D grid unit. Units whose densities exceed the threshold τ are marked as dense;

3. **Generation of Higher-Dimensional Dense Units:** Using an *Apriori-like approach*, the algorithm generates k-dimensional dense units by joining $(k-1)$-dimensional dense units that share common dimensions. This process leverages the *monotonicity property*: if a unit is dense in k-dimensions, all of its $(k-1)$-dimensional projections must also be dense;
4. **Cluster Formation:** Adjacent dense units in each subspace are connected to form clusters. Connectivity is determined by shared boundaries (common faces) between dense units;
5. **Cluster Description:** Each cluster is represented as a union of dense units (hyper-rectangles), and a concise description of each cluster is generated in the form of logical expressions or ranges of attribute values.

After identifying clusters, CLIQUE automatically determines the most relevant subspaces that describe them and produces interpretable outputs such as rules or hyper-rectangular regions summarizing the data distribution within each cluster. This makes CLIQUE effective not only for Clustering but also for subspace pattern discovery. Its computational complexity depends on the number of dimensions and grid units but remains lower than exhaustive high-dimensional Clustering due to its pruning mechanism. By focusing only on dense regions and their neighborhoods, CLIQUE achieves efficient performance even with large datasets.

CLIQUE offers several **advantages**:

- Automatically identifies clusters in subspaces of high-dimensional data without requiring the dimensionality to be specified in advance;
- Efficiently prunes the search space using the monotonicity property of density, reducing computational cost;
- Works well with large datasets due to its grid-based representation and single data scan for density computation;
- Produces interpretable and compact cluster descriptions based on attribute ranges.

However, it also presents some **limitations**:

- Requires the user to specify the number of intervals per dimension and the density threshold τ, which strongly influence results;
- Sensitive to grid resolution: coarse grids may merge distinct clusters, while fine grids may fragment them;
- Produces axis-aligned clusters, which may not capture clusters with arbitrary orientations in the data space;
- May generate a large number of overlapping subspace clusters when many dense regions exist in different combinations of dimensions.

4.6 High Dimensional Clustering

4.6.1 What is High Dimensional Clustering?

High-dimensional Clustering is the process of grouping data objects that are defined by a large number of attributes, often ranging from dozens to thousands of dimensions. In this context, traditional Clustering algorithms that rely on standard distance measures face the **curse of dimensionality**. As the dimensionality increases, the data becomes increasingly sparse, and the difference between the distance to the *nearest neighbor* and the distance to the *farthest neighbor* of a point tends to decrease. This means that in very high-dimensional spaces, points become nearly *equidistant*, which causes distance-based metrics to lose their **discriminative power**.

A significant challenge in high-dimensional spaces is that many dimensions may be *irrelevant or redundant*. These dimensions act as noise and can mask the actual clusters that exist only in a subset of the total feature space. For example, in *customer purchase data*, two individuals might share a common interest in a specific product category, but this similarity is buried under thousands of other products they have not bought. Therefore, Clustering should not only consider the data points but also the specific **attributes** or **features** involved. To address these problems, two main types of methods are proposed: **Subspace Clustering** and **Dimensionality Reduction Clustering**.

4.6.2 What are the Subspace Clustering Methods?

The fundamental objective of **Subspace Clustering** is to identify groups of objects that are similar within specific subsets of dimensions, acknowledging that in high-dimensional datasets, many attributes may be *irrelevant or noisy* for any given cluster. This paradigm differs from global dimensionality reduction because it allows different clusters to exist in different **subspaces** rather than forcing all data into a single lower-dimensional representation. By focusing on relevant attributes for each group, these methods overcome the *sparsity* of high-dimensional space and the loss of distance contrast.

Bottom-up approaches search for clusters by starting from low-dimensional subspaces and iteratively building up to higher dimensions. These methods rely on the **monotonicity property**, which states that if a collection of points forms a dense cluster in a k-dimensional space, those points must also form a dense cluster in all of its $(k-1)$-dimensional projections. Similarity is based on distance or density. The most prominent example is the **CLIQUE** algorithm, which uses a grid-based approach to identify dense regions. It operates through the following steps:

1. Partition each dimension into a grid of non-overlapping intervals:
2. Use the monotonicity property to intersect dense units from k dimensions and generate candidate units for $(k+1)$ dimensions;
   (a) Generate candidate k-dimensional units from $(k-1)$-dimensional dense units;
   (b) Verify the density of candidates to find true dense units.
3. Merge the resulting adjacent dense units to form maximal *dense regions* which constitute the final clusters.

Top-down approaches begin the Clustering process in the full-dimensional space and then iteratively refine the **subspace** associated with each cluster. These methods typically employ **iterative relocation** strategies where each cluster is assigned a subset of dimensions where its members are most tightly grouped. A representative algorithm is **PROCLUS**, which is based on the *k-medoids* framework. It selects a set of potential medoids and, for each medoid, identifies the dimensions in which the local density of points is highest, effectively finding the optimal subspace for each partition. The similarity is based on distance or density.

Correlation-Based Methods are a class of subspace Clustering techniques that identify patterns in high-dimensional data by using correlation models instead of traditional distance or density measures. They apply feature transformation methods to find subspaces where clusters are more apparent, especially when many dimensions are relevant. A common example is **Principal Component Analysis** (**PCA**), which transforms data into uncorrelated dimensions before Clustering. Another well-known example is the **CASH** algorithm that uses the *Hough transform* to map data points into a parameter space. In this transformed space, clusters appear as *dense intersections* of curves, allowing the algorithm to identify the linear correlations that define the cluster's subspace.

Bi-Clustering Methods focus on the **simultaneous** Clustering of both objects and attributes, creating a *submatrix* of the original data. This technique is widely applied in *bioinformatics* for gene expression analysis, where researchers seek to find genes that show similar expression patterns across only a specific subset of experimental conditions. Bi-clusters are characterized by different patterns, such as *constant values*, *constant rows*, or **coherent values** where the data points follow a consistent additive or multiplicative trend across the selected dimensions. This allows for the discovery of local patterns that are hidden when looking at the entire dataset.

4.6.3 Talk about Bi-Clustering Methods

Bi-Clustering, also known as *Co-Clustering*, is a **Subspace Clustering approach** used to cluster both objects and attributes simultaneously, treating them in a symmetric way. This approach is particularly effective for high-dimensional data, where a cluster might only involve a small subset of objects and a small number of attributes. *In this approach, an object or an attribute may participate in multiple clusters or none at all.*

There are four "ideal" types of bi-clusters based on the values within the submatrix:

- **Constant Values:** For any row i and column j in the submatrix, the value $e_{ij} = c$;
- **Constant Values on Rows/Columns:** Values are constant across rows ($e_{ij} = c+\alpha_i$) or across columns ($e_{ij} = c + \beta_j$);
- **Coherent Values (Pattern-Based):** Rows and columns change in a synchronized way. This is defined by the condition $e_{ij} = c + \alpha_i + \beta_j$, or more formally, for any two rows i_1, i_2 and two columns j_1, j_2:

$$e_{i_1 j_1} - e_{i_2 j_1} = e_{i_1 j_2} - e_{i_2 j_2}$$

- **Coherent Evolutions:** The algorithm only cares if values increase or decrease in a synchronized way, without requiring exact numerical coherence.

The δ**-Bi-Cluster** is a sophisticated model designed to identify submatrices that exhibit **coherent values**. While simpler models might search for submatrices with constant values or constant rows, the δ-bi-cluster model accounts for the fact that in real-world data, such as *gene expression profiles*, values often shift consistently across rows and columns. This means that the relative difference between two attributes remains stable across all objects in the cluster. This relationship is often referred to as an *additive model*, where each entry in the submatrix is the sum of a base value, a row-specific effect, and a column-specific effect.

To identify these patterns, the concept of the **residue** is used to measure how well an individual entry fits into the overall coherence of a submatrix. For a submatrix defined by a set of rows I and columns J, the residue of an element e_{ij} is calculated by taking its value and subtracting the mean of its row within the submatrix and the mean of its column within the submatrix, then adding the overall mean of the entire submatrix. This calculation effectively isolates the *local noise* or deviation from the expected additive pattern.

The quality of a submatrix $I \times J$ as a bi-cluster can be measured by its **mean squared residue value**. Given a gene expression matrix $E = [e_{ij}]$, we define the following means:

- Mean of the i-th row: $e_{i,J} = \frac{1}{|J|} \sum_{j \in J} e_{ij}$
- Mean of the j-th column: $e_{I,j} = \frac{1}{|I|} \sum_{i \in I} e_{ij}$
- Mean of all elements in the submatrix: $e_{IJ} = \frac{1}{|I||J|} \sum_{i \in I, j \in J} e_{ij}$

The mean squared residue is defined as:

$$H(I \times J) = \frac{1}{|I||J|} \sum_{i \in I, j \in J} (e_{ij} - e_{iJ} - e_{Ij} + e_{IJ})^2$$

A submatrix is a δ**-bi-cluster** if $H(I \times J) \leq \delta$, where $\delta \geq 0$ is a user-defined noise tolerance threshold.

Due to the high cost of computing maximal bi-clusters, a **heuristic greedy search** is employed. This search is called δ-Cluster Algorithm and it is divided in two phases:

1. **Deletion Phase**: Start with the entire matrix. Iteratively calculate the mean squared residue for each row and column. Remove the row or column that has the largest mean squared residue until the residue of the remaining matrix is less than or equal to δ;
2. **Addition Phase**: Iteratively expand the δ-bi-cluster obtained in the deletion phase. Calculate the mean squared residues for all rows and columns not currently involved. Add the row or column with the smallest residue into the cluster as long as the δ-bi-cluster requirement $H \leq \delta$ is maintained.

While the mean squared residue captures average noise, the **δ-pCluster** model focuses on pattern similarity within every 2×2 submatrix. A submatrix $I \times J$ is a δ-pCluster if, for every 2×2 submatrix, the p-score is at most δ.

The p-score is defined as:

$$p\text{-score}\begin{pmatrix} e_{i_1 j_1} & e_{i_1 j_2} \\ e_{i_2 j_1} & e_{i_2 j_2} \end{pmatrix} = |(e_{i_1 j_1} - e_{i_2 j_1}) - (e_{i_1 j_2} - e_{i_2 j_2})| \leq \delta$$

The **algorithm properties** are:

1. **Monotonicity**: If $I \times J$ is a δ-pCluster, every $x \times y$ submatrix $(x, y \geq 2)$ of it is also a δ-pCluster;
2. **Maximality**: The search focuses on finding maximal δ-pClusters, where no more rows or columns can be added without violating the threshold.

In the end, **MaPle** is an efficient algorithm designed to enumerate all maximal δ-pClusters. It utilizes a pattern-growth framework based on the downward closure property (monotonicity). For each combination of conditions J, it identifies the maximal subset of genes I such that $I \times J$ forms a δ-pCluster, a process highly similar to mining frequent closed itemsets.

4.6.4 What are the Dimensionality-Reduction Methods?

Dimensionality-Reduction methods are fundamental techniques used to manage the challenges posed by the **curse of dimensionality** by transforming high-dimensional data into a lower-dimensional space. The primary goal is to retain the most significant structural information while removing **noise** and **irrelevant attributes** that can obscure the true underlying clusters. These methods are typically categorized into two paradigms: **feature selection**, which involves choosing a subset of the original dimensions, and **feature transformation**, which creates entirely new variables through mathematical operations on the original set.

In the context of linear transformations, **Principal Component Analysis** (PCA) and **Singular Value Decomposition** (SVD) are the most widely used methods. These techniques identify *principal components*, which are new, orthogonal dimensions along which the data exhibits the highest variance. By projecting the dataset onto the first few components, we can achieve a compressed representation that simplifies the Clustering process. These methods are particularly effective when dimensions are *highly correlated* or redundant, allowing the algorithm to focus on the core information.

A more sophisticated approach is **Spectral Clustering**, which performs non-linear dimensionality reduction. Unlike PCA, which looks for global variance, Spectral Clustering focuses on the **connectivity** and similarity between data points. It maps the data into a new space where clusters that were originally *non-convex* or intertwined become linearly separable. This process is highly effective for complex datasets such as image segments or protein networks.

The algorithm operates through the following sequence:

1. Create an **affinity matrix** where each entry represents the similarity between a pair of data points, often using a Gaussian kernel;
2. Construct a **Laplacian matrix** from the affinity matrix, which captures the geometric structure of the data manifold;
3. Compute the k smallest **eigenvectors** of the Laplacian matrix to define a new k-dimensional subspace;
4. Map the original data points into this lower-dimensional *eigenspace*;
5. Cluster the transformed points using a standard algorithm like **K-Means** to produce the final groups.

The main advantage of these methods is their ability to reveal *latent structures* that are invisible in the original high-dimensional space. However, they also introduce a **scalability challenge**, as the computation of eigenvectors for large affinity matrices can be extremely resource-intensive.

4.7 Constrained Clustering

4.7.1 What is the problem with constraints in Clustering and what is the solution?

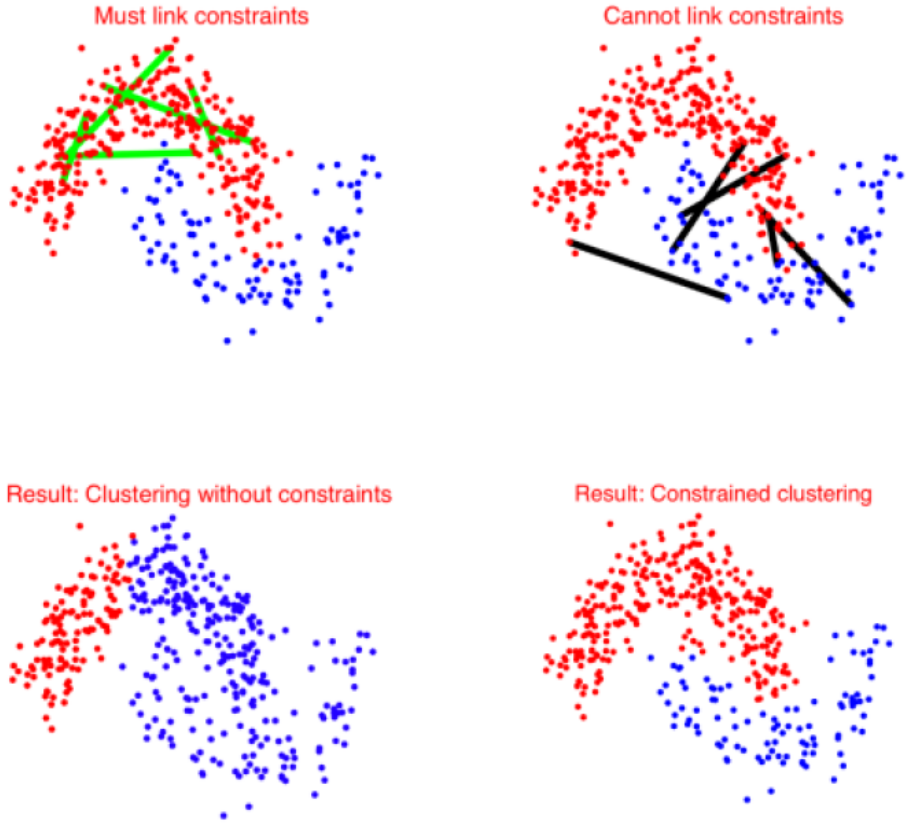
Introduction

The primary **problem** in traditional Clustering is that it is a purely *unsupervised process* that ignores domain-specific knowledge or physical requirements. Algorithms like K-Means or Hierarchical Clustering group data based solely on mathematical similarity, which can lead to results that are theoretically sound but practically *unusable* or *meaningless* for a specific application. For example, when planning the location of service centers in a city, a standard algorithm might group points across a river where no bridge exists, effectively ignoring *physical obstacles*. To solve this, **constraint-based cluster analysis** incorporates user-defined rules and background knowledge into the Clustering process to ensure that the resulting groups satisfy real-world requirements.

The **solution** involves defining and enforcing different categories of constraints that guide the algorithm toward a *user-desired* partition. These are typically divided into **constraints on instances** and **constraints on clusters**.

Constraints on Instances

Must-link constraints specify that two specific objects, x and y, must be placed in the same cluster because they are known to be related. Conversely, **cannot-link constraints** dictate that two objects must be assigned to different clusters. These instance-level constraints are essential when certain data points have *pre-existing relationships* that the distance metric cannot capture.



Constraints on Clusters

Beyond individual instances, constraints can also be applied to the *geometric properties* of the clusters themselves. The ε**-constraint** is used to control the **tightness** or **density** of a cluster, requiring that for any point x in a cluster C, the distance to the cluster center c satisfies the following condition:

$$dist(x; c) \leq \varepsilon$$

This prevents clusters from becoming too geographically dispersed.

On the other hand, the δ**-constraint** (or *minimum separation constraint*) ensures that the clusters are well-separated from each other. It requires that for any two points x and y belonging to different clusters C_i and C_j, the distance must satisfy:

$$dist(x; y) \geq \delta$$

This formula ensures that the space between different groups is large enough to maintain a **clear distinction** between the identified patterns.

Constraints Enforcement

The way these constraints are enforced depends on whether they are treated as **hard constraints** or **soft constraints**. Hard constraints are mandatory requirements that must be strictly satisfied. If a candidate Clustering violates even one *must-link* or *cannot-link* rule, it is considered invalid and rejected. Soft constraints are more flexible and are used when the user wants the algorithm to "try its best" to satisfy the rules. In this case, the Clustering problem is framed as an **optimization task** where the objective function includes a **penalty factor**. The algorithm seeks to minimize a total cost V, which is the sum of the standard Clustering error (like Sum of Squared Errors) and the weighted violations:

$$V = \text{Error}(C) + w \cdot \text{Penalty}(C)$$

Here, w represents the *penalty weight* that determines how strictly the algorithm should prioritize satisfying the constraints over achieving the best mathematical fit.

Speeding Up Constrained Clustering

The primary challenge in Constrained Clustering with obstacles is that the shortest path between two points is no longer a straight line, but a trajectory that must navigate around physical barriers. Calculating these "*Geodesic Distances*" for every pair of objects is computationally prohibitive. To mitigate this, three main optimization strategies are employed:

1. **Visibility Graphs (VG):** To determine the valid path between points, the system constructs a **Visibility Graph** $VG = (V; E)$. In this graph, every vertex of every obstacle is represented as a node. An edge exists between two nodes if, and only if, the vertices they represent are "visible" to each other, meaning the straight line joining them does not intersect any obstacle. The shortest path between any two points p and q in the dataset is then guaranteed to be a subpath within an expanded visibility graph (VG'), transforming a continuous geometric problem into a discrete graph-search problem;

2. **Spatial Triangulation and Microclustering:** Rather than processing every individual data point for distances, which could number in the millions, the algorithm reduces the "granularity" of the dataset. The region R containing the data is first partitioned through *triangulation*. Nearby points within the same triangular sector are then grouped into *microclusters* using density-based or hierarchical techniques similar to those found in *BIRCH* or *DBSCAN*. By performing the heavy computations on a few thousand microclusters instead of millions of individual points, the overall overhead is drastically reduced;

3. **VV and MV Join Indices**: To further accelerate distance lookups during the iterative Clustering process, the system utilizes pre-computed *join indices* based on shortest-path calculations:

   - **VV Indices**: These store the shortest paths between all pairs of **obstacle vertices**;
   - **MV Indices**: These store the shortest paths between each **microcluster** and the relevant **obstacle vertices**.

All this allows the constrained Clustering process to scale to large datasets while maintaining a performance level similar to high-efficiency algorithms like *CLARANS*.

4.7.2 Talk about COP-K Means

The **COP-K Means** algorithm is an extension of the traditional K-Means partitioning method designed to incorporate *background knowledge* in the form of **hard constraints**. While standard Clustering is purely unsupervised, COP-K Means allows users to specify mandatory relationships between data points, specifically through **must-link** and **cannot-link** constraints. This ensures that the resulting clusters are not only mathematically coherent based on distance metrics but also practically relevant to the specific domain application.

The standard cost function for the K-Means algorithm is the **Sum of Squared Error** (**SSE**), also referred to as the *Total Within-Cluster Variance*. It is defined as:

$$E = \sum_{i=1}^{k} \sum_{p \in C_i} \|p - c_i\|^2$$

where c_i is the centroid (mean point) of cluster C_i:

$$c_i = \frac{1}{|C_i|} \sum_{p \in C_i} p$$

The primary aim of minimizing the SSE is to maximize **cluster cohesion** or compactness. By reducing the squared error, the algorithm ensures that objects within each group are as similar as possible to one another (high *intra-cluster similarity*). Geometrically, this results in clusters where points are tightly packed around their respective centroids.

A relation is transitive if $A \to B$ and $B \to C$ imply $A \to C$. The **transitive closure** of a relation ($\to$) is the smallest possible relation that contains all the original pairs and satisfies the transitive property.

The algorithm modifies the standard K-Means logic by introducing a *feasibility check* during the assignment phase and simplifying the dataset through the creation of **super-instances**.

The execution of COP-K Means follows these specific steps:

1. Identify all **must-link constraints** and compute their *transitive closure* to find all sets of points that are required to be in the same cluster;
2. Replace each connected component found in the transitive closure with a single **super-instance** represented by the *mean value* of the points in that set;
3. Assign a *weight* to each super-instance equal to the total number of original points it contains;
4. Randomly initialize k cluster centroids;
5. For each data point or super-instance, find the *nearest centroid* that does not violate any **cannot-link constraints**;
6. If the nearest centroid causes a violation, attempt to assign the point to the *next nearest* feasible centroid;
7. Terminate the process and report a *failure* if a point exists that cannot be assigned to any cluster without violating a hard constraint;
8. Update the cluster centroids by calculating the mean of all points and super-instances (accounting for their weights) assigned to each cluster;
9. Repeat the assignment and update phases until the centroids **converge** and no further changes occur.

COP K-Means has several **advantages**:

- It provides a direct mechanism to enforce **mandatory domain rules** that a purely mathematical distance-based approach would ignore;
- The use of *super-instances* reduces the effective size of the dataset, which can decrease the computational burden during the iterative phases;
- It guarantees that the final output will be a **feasible solution** that respects every user-provided constraint;
- It effectively handles *complex spatial requirements*, such as ensuring points known to be related are never separated.

However, it also presents important **limitations**:

- The algorithm is highly sensitive to the **order of points** and the initial selection of centroids, which can frequently lead to a state where no feasible assignment is possible;
- It cannot handle *conflicting constraints*, and any contradiction in the input rules will prevent the algorithm from finding a valid partition;
- Because it treats constraints as **hard requirements**, it may produce clusters that have a much higher *Sum of Squared Errors (SSE)* than a standard unconstrained approach;
- It lacks the *flexibility* of soft-constraint methods that allow for occasional violations in exchange for better overall cluster quality.

4.7.3 Talk about CVQE

The **CVQE (Constrained Vector Quantization Error)** algorithm is a *partitioning Clustering approach* designed to handle **soft constraints** by treating the Clustering task as an optimization problem. Unlike hard-constraint methods that reject any invalid partition, CVQE allows for the violation of constraints if no better solution can be found, imposing a **penalty** on the Clustering quality instead. The algorithm aims to find a balance by simultaneously optimizing Clustering quality (minimizing distance) and minimizing the total penalty incurred by constraint violations.

The objective function V is defined as the sum of distances used in standard K-Means, adjusted by specific violation penalties:

$$V = \sum_{x \in D} dist(x; c(x))^2 + \text{Penalty}(C)$$

The penalties for violating **must-link** and **cannot-link** constraints are calculated as follows:

- **Must-link violation**: If objects x and y must be linked but are assigned to different centers c_1 and c_2, the penalty $dist(c_1; c_2)$ is added to the objective function.
- **Cannot-link violation**: If objects x and y cannot be linked but are assigned to a common center c, the penalty $dist(c; c')$ is added, where c' is the closest cluster to c that can accommodate x or y.

The execution of the CVQE algorithm follows these structured steps:

1. Initialize k cluster centroids randomly or using a specific heuristic;
2. For each object in the dataset, evaluate its distance to each centroid along with any potential penalties incurred by must-link or cannot-link violations;
3. Assign each object to the cluster center that minimizes the combined value of the distance and the constraint violation penalties;
4. Recalculate the centroids as the mean of the objects currently assigned to each cluster;
5. Repeat the assignment and update steps until the centroids converge or a maximum number of iterations is reached.

CVQE has several **advantages**:

- It provides a flexible framework that can accommodate contradictory or noisy constraints through a "soft" preference system;
- It integrates domain-specific background knowledge into a mathematically rigorous optimization task;
- It ensures that constraints are maximally respected while still prioritizing the inherent similarity between data objects.

However, it also presents important **limitations**:

- The algorithm incurs significant computational costs, particularly when identifying the closest alternative cluster c' for cannot-link violations;
- Like standard K-Means, it is a heuristic approach that may converge to local optima rather than the global minimum of the objective function;
- Its performance is heavily dependent on the chosen distance metric and the initial placement of centroids.

4.8 Graph Clustering

4.8.1 How does Graph Clustering work?

A **Network** is defined as a collection of entities, such as people, proteins, or web pages, that are interconnected through various types of links. In our modern world, networks have grown exponentially in size and complexity, spanning from social media connections to metabolic pathways in biological systems. Studying these structures is essential because individual entities only provide a partial view of a complex system. We must understand the underlying network to grasp how the system functions as a whole.

In mathematical terms, a network is represented as a **Graph** denoted by $G(N; E)$. In this notation, N represents the set of **nodes** (or vertices) which are the entities, and E represents the set of **edges** which are the links (interactions) between them. Graphs provide a rigorous *mathematical framework* to study real-world systems, whether the edges are directed, like a hyperlink pointing from one page to another, or weighted, representing the *strength of collaboration* between researchers.

Currently, several **problems** drive the need for advanced graph analysis. These include *ranking nodes* to determine their importance on the web, predicting *information cascades* where viruses or news propagate through a population, and *link prediction* to identify relationships that are likely to form in the future. **Graph Clustering**, also known as *community detection*, addresses these problems by seeking to identify **cohesive subgroups** where nodes have strong, frequent, or positive ties to each other while remaining relatively isolated from the rest of the network.

To perform cluster analysis on graph data, researchers generally follow one of two primary *methodological paradigms*:

- Transform the graph into a high-dimensional dataset by extracting a **similarity matrix** based on distance measures such as the *Geodesic Distance* or *SimRank*. Apply standard partitioning or hierarchical algorithms to this matrix to group the objects based on their calculated proximity;
- Use methods designed specifically for graphs that exploit their *topological peculiarities* to identify clusters directly. Optimize a quality metric, such as **modularity**, which measures the difference between the actual edges in a cluster and the expected edges in a random connection model. Then partition the graph by finding *sparsest cuts* or dense regions where the connectivity within the group is maximized.

One of the most effective graph-specific approaches is **density-based Clustering**, exemplified by the *SCAN algorithm*. Unlike global partitioning methods, these algorithms look for **structural similarity** between neighboring nodes. They distinguish between *core nodes* that form the heart of a community, *hubs* that bridge multiple groups without belonging to any, and *outliers* that sit at the margins of the network. This allows for a more robust discovery of communities that reflects the **natural connectivity** and hierarchy found in real-world data.

4.8.2 What is the Geodesic Distance?

The **Geodesic Distance** between two vertices in a graph is the length of the *shortest path* connecting them, usually measured by the number of **edges**. If no path exists, the distance is defined as **infinite**. This metric is a basic tool for describing the *topological structure* of a graph and underlies many classical network analysis methods.

Several global graph properties are derived from geodesic distances. The **eccentricity** of a vertex v are the maximum geodesic distances from v to any other vertex. The **radius** of the graph is the minimum eccentricity, while the **diameter** is the maximum eccentricity. Vertices attaining the diameter are called *peripheral*.

In *social networks*, geodesic distance represents the most efficient route for information flow between individuals. However, it is limited as a similarity measure since it only accounts for shortest paths and ignores broader *structural context*, such as shared neighbors.

Due to these limitations, modern graph Clustering often moves beyond simple geodesic distances to more robust measures. These include **SimRank**, which is based on *random walks* and structural context, and *density-based measures* like structural similarity. These advanced approaches capture the *likelihood* that two nodes are related based on the entire network structure rather than just a single shortest path. Nevertheless, the geodesic distance remains an essential **topological descriptor** for calculating network diameter and identifying central hubs that facilitate communication across different communities.

4.8.3 What is SimRank?

SimRank is a general *similarity measure* that expresses the similarity between two nodes in a graph based on a **structural context** intuition. The core principle of this approach is that "two objects are similar if they are related to similar objects". In a directed graph, this means that two nodes are considered similar if their *in-neighbors* are themselves similar. This definition is inherently **recursive** because the similarity of the current pair of nodes depends on the similarity of the pairs in their respective neighborhoods.

Mathematically, for two distinct nodes u and v, the SimRank score $s(u;v)$ is calculated by averaging the similarities of all pairs of their **in-neighbors** and then multiplying by a **decay factor** C, which is a constant between zero and one. This decay factor ensures that similarity decreases as we move further away in the graph.

$$s(u;v) = \frac{C}{|I(u)||I(v)|} \sum_{x \in I(u)} \sum_{y \in I(v)} s(x;y)$$

$I(u)$ is the individual in-neighborhood of u (links that enter in u).

$$I(u) = v \mid (v;u) \in E$$

The base cases for this recursion are straightforward: the similarity of a node to itself is defined as one, and if two distinct nodes have no in-neighbors, their similarity is zero.

$$s_0(u;v) = \begin{cases} 1 & \text{if } u = v \\ 0 & \text{if } u \neq v \end{cases}$$

To compute the actual scores, the equation is solved iteratively until a fixed point is reached.

$$s_{i+1}(u;v) = \begin{cases} \frac{C}{|I(u)||I(v)|} \sum_{x \in I(u)} \sum_{y \in I(v)} s_i(x;y) & \text{if } u \neq v \\ 1 & \text{if } u = v \end{cases}$$

This model is particularly effective in *web mining* and *recommender systems* because it identifies nodes that serve similar roles even if they are not directly connected.

The SimRank measure can also be understood through the lens of a **random walk**. Specifically, the similarity score is related to the **expected meeting distance** (EMD), which represents the expected number of steps required for two *random surfers* starting at nodes u and v to meet at the same node if they move in lock-step. Under this interpretation, nodes are highly similar if surfers starting from them are likely to meet quickly. SimRank effectively maps these meeting distances to a finite interval where a score of one represents an immediate meeting (the nodes are the same) and a score of zero indicates that the surfers can never meet.

The calculation of these scores is performed through an **iterative process** that refines the similarity values until they reach a *fixed point*. The algorithm for computing SimRank follows these steps:

1. Initialize the similarity matrix S_0 such that $s_0(u;v)$ is equal to one if u is the same as v and zero otherwise;
2. For each pair of nodes in the next iteration, apply the recursive formula using the similarity values from the previous step;
3. Update the entries of the matrix while ensuring that self-similarity remains at the constant value of one;
4. Repeat the calculation for a fixed number of iterations K, typically five, or until the values in the matrix **converge** to a stable state;
5. Output the final similarity matrix which contains the *discriminative scores* for all node pairs in the graph.

One of the main *challenges* of SimRank is its computational **complexity**, which is $O(Kn^2d_2)$, where n is the number of nodes, K is the number of iterations, and d_2 represents the average product of the in-degrees of the node pairs. Because the algorithm must maintain a score for every possible pair of vertices, it requires $O(n^2)$ **memory**, which can be prohibitive for very large *social networks* or web graphs. Despite this, it remains a fundamental tool for **Graph Clustering** because it combines both local connectivity and global structural properties into a single, intuitive metric.

4.8.4 What is the Sparsest Cut and how can we find good cuts?

The **sparsest cut** is a criterion used in *Graph Clustering* to partition a set of vertices into two subsets, S and T, in a way that minimizes the ratio of edges crossing the cut relative to the number of nodes in the smaller side. Intuitively, Clustering should cut the graph into pieces where the vertices within a cluster are *well-connected* and those in different clusters are connected in a much **weaker way**. While a simple **minimum cut** strategy aims to minimize the total number of edges removed, it often produces **unbalanced partitions** where a single node is separated from the rest of the graph. The sparsest cut improves upon this by introducing a *balancing factor* into the denominator.

The **sparsity** of a cut $C = (S; T)$ is formally defined by the formula:

$$\Phi = \frac{\text{cut\_size}}{\min(|S|; |T|)}$$

where the cut size is the number of edges crossing the boundary. $|S|$ and $|T|$ are the number of nodes (the cardinality) in the set S and in the set T respectively, sets obtained **after** the cut.

A cut is considered the **sparsest** if its value Φ is not greater than that of any other possible cut. This measure favors solutions that are both *sparse* (few edges crossing the cut) and **balanced** (the sizes of S and T are relatively similar). Finding the absolute sparsest cut is known to be an **NP-hard problem**, so researchers rely on approximation algorithms. One of the best-known theoretical results is an $O(\sqrt{\log n})$ approximation in the *Arora, Rao, and Vazirani algorithm*, which allows for finding reasonably good clusters even in massive networks.

To determine what constitutes a **good cut** for high-quality Clustering, we often use **modularity** as a quality assessment metric. Modularity, denoted as Q, measures the difference between the fraction of edges that fall within the identified clusters and the expected fraction of edges if the graph vertices were **randomly connected**.

$$Q = \sum_{i=1}^{k} \left(\frac{l_i}{|E|} - \left(\frac{d_i}{2|E|} \right)^2 \right)$$

where:

- l_i: The number of edges between vertices within the i-th cluster;
- d_i: The sum of the degrees of the vertices (sum of internal and external edges) in the i-th cluster;
- $|E|$: The total number of edges in the graph.

What $\frac{l_i}{|E|}$ and $\frac{d_i}{2|E|}$ represents in brief?

- $\frac{l_i}{|E|}$ (**Observed Probability**): This represents the actual fraction of all edges in the graph that are contained within cluster i. It is the probability that a randomly selected edge from the real graph connects two nodes that both belong to that specific cluster.
- $\frac{d_i}{2|E|}$ (**Expected Probability Base**): This represents the probability that a single end of a randomly chosen edge is attached to a vertex in cluster i. In the modularity formula, this term is squared to represent the expected fraction of edges that would fall within cluster i if the graph's vertices were randomly connected while maintaining their original degrees.

Modularity is the index we use to understand whether the division of a graph into groups (clusters) is "valuable" or if it occurred by pure chance. A good Clustering result will maximize this value, indicating a strong *community structure* that is not the result of random chance.

Despite its theoretical elegance, finding good cuts in real-world data presents several **challenges**. First, the *computational cost* is extremely high for large-scale graphs because the search space for potential partitions is enormous. Second, most modern networks exhibit **high sparsity**, meaning each vertex connects to only a small fraction of other vertices, which makes it harder to distinguish between *structural gaps* and actual cluster boundaries. Finally, high-dimensional similarity matrices derived from these graphs can be *computationally expensive* to process. Therefore, effective Graph Clustering requires a trade-off between **scalability** and the **quality** of the identified cuts to ensure the discovered communities are both statistically significant and practically useful.

4.8.5 Talk about SCAN

The **SCAN** (**Structural Clustering Algorithm for Networks**) is a *density-based Clustering* algorithm specifically designed to identify clusters in large-scale network data. Unlike traditional partitioning methods that rely solely on distance, SCAN focuses on **structural similarity**, which is based on the intuition that two nodes are similar if they share many common neighbors. This approach allows the algorithm to discover *cohesive subgroups* regardless of their size or shape while effectively segregating nodes that serve as bridges or noise.

Nodes for SCAN are divided in:

- **Cliques**: These are individuals belonging to a tight social group. Members of a clique typically know many of the same people, regardless of the overall size of the group. In the context of the SCAN algorithm, they exhibit high structural similarity with other members of their cluster.
- **Hubs**: These are individuals who act as connectors or bridges between multiple groups but do not belong to any single cluster themselves. A classic example is a politician who bridges various social or professional circles. From a structural standpoint, they are not part of a cluster but have edges connecting them to many different clusters.
- **Outliers**: These individuals reside at the margins of the network and do not belong to any cluster. An example provided is a "hermit" who knows very few people. Unlike hubs, outliers connect to significantly fewer clusters or nodes, remaining isolated from the primary dense regions of the graph.

The immediate neighborhood of a vertex u includes the vertex itself and all vertices it is connected to:

$$\Gamma(u) = \{v \mid (u; v) \in E\} \cup \{u\}$$

At the core of the algorithm is the **structural similarity** metric, denoted as $\sigma(v; w)$, which is calculated as the number of common neighbors between two adjacent vertices divided by the geometric mean of their neighborhood sizes.

$$\sigma(u; v) = \frac{|\Gamma(u) \cap \Gamma(v)|}{\sqrt{|\Gamma(u)| \cdot |\Gamma(v)|}}$$

Using two user-defined parameters, the similarity threshold ε and the popularity threshold μ, the algorithm identifies **core vertices**.

For a vertex u, the ε-Neighborhood is the set of neighbors whose structural similarity to u is at least ε:

$$N_\varepsilon(u) = \{w \in \Gamma(v) \mid \sigma(u; v) \geq \varepsilon\}$$

A vertex u is considered a **core vertex** if its ε-Neighborhood contains at least μ vertices:

$$CORE_{\varepsilon,\mu}(u) \Leftrightarrow |N_\varepsilon(u)| \geq \mu$$

A vertex v is **directly structure-reachable** from a core vertex u if it belongs to u's ε-Neighborhood:

$$DirRECH_{\varepsilon,\mu}(u; v) \Leftrightarrow CORE_{\varepsilon,\mu}(u) \wedge v \in N_\varepsilon(u)$$

Two vertices u and v are **structure-connected** if there exists a core vertex w from which both u and v are structure-reachable (list of $DirRECH$ nodes between them):

$$CONNECT_{\varepsilon,\mu}(u; v) \Leftrightarrow \exists u \in V : RECH_{\varepsilon,\mu}(w; u) \wedge RECH_{\varepsilon,\mu}(w; v)$$

Clusters are then formed by finding the **maximal sets** of nodes that are **structure-connected** through these core vertices. So, *Hubs* are bridge nodes that do not belong to any specific cluster but connect multiple different clusters together, and *Outliers* are isolated nodes that do not belong to any cluster and connect to very few or no other groups, acting as noise in the network.

The SCAN algorithm proceeds through the following sequence of steps:

1. Initialize all vertices in the graph $G(V; E)$ as unlabeled;
2. For each unlabeled vertex u in the set u, check if it meets the criteria to be a **core vertex** based on ε and μ;
3. If u is a core, generate a new **cluster-id** and initialize a queue Q containing all nodes in the ε-neighborhood of u. If u is not a core, label it as a **non-member** and proceed to the next unlabeled vertex until it is a core;
4. While the queue is not empty, extract the first vertex v and identify all nodes that are **directly structure-reachable** from it;
5. For each reachable node, if it is unlabeled or previously marked as a non-member, assign it the current *cluster-id*;
6. If the reachable node is itself a core, insert it into the queue to continue **expanding the cluster** through its neighbors;
7. When $N_\varepsilon(u)$ has no object to check, cluster C is completed and returned in output. Go to the next unlabeled point and re-start from point (2);
8. Once all clusters are formed, examine each non-member to determine if it connects to multiple clusters, in which case it is labeled as a **hub**, or otherwise as an **outlier**.

SCAN has several **advantages**:

- It can discover clusters of **arbitrary shapes** and sizes because it does not assume a spherical or balanced distribution of nodes;
- The algorithm provides a *robust classification* of non-cluster nodes, making it possible to identify critical **hubs** in social or communication networks;
- It does not require the user to pre-specify the **number of clusters**, as it naturally finds communities based on local density;
- The method is inherently **local**, meaning it only needs information about immediate neighbors to perform the Clustering process.

However, it also presents important **limitations**:

- The results are **highly sensitive** to the choice of the parameters ε and μ, which can be difficult for a user to estimate without prior domain knowledge;
- It primarily focuses on the *topology of the graph* and may ignore additional attribute information if not properly integrated;
- In networks with **varying densities**, a single set of parameters might fail to capture all meaningful clusters simultaneously;
- The *greedy expansion* strategy can sometimes be affected by noise if the similarity threshold is set too low.

In terms of **complexity**, the SCAN algorithm is exceptionally efficient and scalable for massive datasets. The general **running time** is $O(|E|)$, where $|E|$ represents the number of edges in the network. This efficiency arises because the algorithm primarily performs local neighborhood searches and processes each edge a constant number of times. For **sparse networks**, which characterize most real-world social and biological graphs, the complexity simplifies to $O(|V|)$, where $|V|$ is the number of vertices. This *linear scalability* makes SCAN a superior choice compared to global optimization algorithms when dealing with networks containing millions or billions of nodes.

4.9 Evaluation of Clustering

4.9.1 How to Evaluate the Clusters?

Extrinsic and Intrinsic methods

Clustering Evaluation addresses three related problems:

1. Deciding whether a dataset exhibits a non-random structure that merits Clustering;
2. Determining a reasonable number of clusters for a given dataset;
3. Measuring how well a produced partition fits the data and how Clusterings can be compared.

Assessing Clustering tendency asks whether the data are significantly non-uniform (i.e., contain meaningful groups) before applying a Clustering algorithm. A commonly used spatial statistic for this purpose is the **Hopkins statistic**. The Hopkins test compares distances from a small sample of real points to their nearest neighbors in the dataset with distances from uniformly generated random points (in the same region) to their nearest neighbors in the dataset. If the dataset were uniformly random the two sets of distances should be similar, whereas clustered data produce systematically smaller nearest-neighbor distances for the sample drawn from the data than for the uniformly generated points.

The Hopkins procedure can be described operationally as follows:

1. Sample $n \ll N$ data points $p_1, \ldots, p_n$ uniformly from the dataset D. For each p_i compute x_i, the distance to its nearest neighbor in D;
2. Generate n uniformly random points $q_1, \ldots, q_n$ in the same data range (a simulated random dataset) and for each q_i compute y_i, the distance from q_i to its nearest neighbor in D;
3. Combine the distances into the Hopkins statistic H: intuitively, if D is uniform then $H \approx 0.5$, if D has Clustering tendency the numerator distances from real points are smaller and H grows above 0.5. Repeating the experiment and averaging gives a robust decision rule. Values $H > 0.75$ are typically taken to indicate Clustering tendency with high confidence.

Determining the number of clusters is often approached with heuristics that trade model complexity and within-cluster compactness.

The **Elbow method** is described operationally as:

1. For each candidate $k > 0$, run a Clustering algorithm (for example K-Means) to produce k clusters;
2. Compute var(k), the total within-cluster variance (sum of squared distances of points to their cluster centroid);
3. Plot var(k) versus k and look for the first marked turning point (the "elbow") where increasing k yields diminishing returns in variance reduction. Choose that k. The idea is that splitting a truly cohesive cluster yields only a small decrease in var(k), so the elbow marks an appropriate tradeoff.

The **cross-validation approach** for selecting k works analogously to supervised model selection:

1. Partition the dataset into m folds;
2. For each candidate k, fit the Clustering (e.g., compute centroids) on $m-1$ folds and evaluate on the held-out fold by assigning each test point to its closest centroid and measuring the sum of squared distances (or other fit metric);
3. Repeat across folds and average the test score. Then compare averaged scores across k and pick the number of clusters that yields the best cross-validated fit. This approach gives a more empirical, data-driven choice of k.

Measuring Clustering quality divides into extrinsic (external) and intrinsic (internal) methods.

BCubed

Extrinsic methods assume a **ground truth labeling** exists and measure how well the Clustering recovers that labeling.

Extrinsic evaluation should satisfy several criteria:

- **Cluster Homogeneity:** clusters should be pure relative to ground truth;
- **Cluster Completeness:** objects from the same ground-truth class should be assigned to the same cluster;
- **Sensitivity to "rag bag" effects:** putting a heterogeneous object into a pure cluster is penalized more than putting it into an already mixed cluster;
- **Small Cluster Preservation:** splitting a small true category is more harmful than splitting a large one.

A concrete extrinsic measure is the **Bcubed precision/recall framework**. The Bcubed idea is object-centric: for each object you compute a **Precision** (the fraction of objects in its assigned cluster that share its ground-truth label) and a **Recall** (the fraction of objects with the same ground-truth label that appear in its cluster), then average these per-object precision and recall values across the dataset to obtain global precision and recall scores. *These can be combined (via F-measure) to produce a single quality statistic.*

Let o_i be an object in the dataset, $C(o_i)$ be the cluster containing o_i, and $L(o_i)$ be the set of objects sharing the same category as o_i in the ground truth, in a dataset with N objects. Then the correctness of the relation between two different objects ($i \neq j$) o_i and o_j in Clustering C is given by:

$$\text{Correctness}(o_i; o_j) = \begin{cases} 1 & \text{if } L(o_i) = L(o_j) \iff C(o_i) = C(o_j) \\ 0 & \text{otherwise} \end{cases}$$

The *Precision* of an object o_i indicates how many other objects in the same cluster belong to the same category as o_i. The overall Precision is:

$$Precision_{BCubed} = \frac{\sum_{i=1}^{N} \frac{\sum_{o_j;\, C(o_i)=C(o_j)} \text{Correctness}(o_i;o_j)}{|\{o_j | C(o_j)=C(o_i)\}|}}{N}$$

with $i \neq j$.

The *Recall* of an object o_i reflects how many objects of the same category are assigned to the same cluster as o_i. The overall Recall is:

$$Recall_{BCubed} = \frac{\sum_{i=1}^{N} \frac{\sum_{o_j;\, L(o_i)=L(o_j)} \text{Correctness}(o_i;o_j)}{|\{o_j | L(o_j)=L(o_i)\}|}}{N}$$

with $i \neq j$.

Silhouette-based diagnostic

Intrinsic methods (unsupervised, without ground truth) focus on within-cluster cohesion and between-cluster separation. The most prominent index is the **Silhouette Coefficient**. For a point o define:

$$a(o) = \text{average distance from } o \text{ to all other points in its cluster}$$

$$b(o) = \min_{C' \neq C(o)} \text{average distance from } o \text{ to all points in cluster } C' \quad \text{for all Clusters}$$

that is, $b(o)$ is the smallest mean distance from o to any other cluster. The silhouette coefficient of o is then:

$$a(o) = \frac{\sum_{\substack{o' \in C_i \\ o \neq o'}} \text{dist}(o; o')}{|C_i| - 1}$$

$$b(o) = \min_{j;\, C_j;\, j \neq i} \left(\frac{\sum_{o' \in C_j} \text{dist}(o; o')}{|C_j|} \right)$$

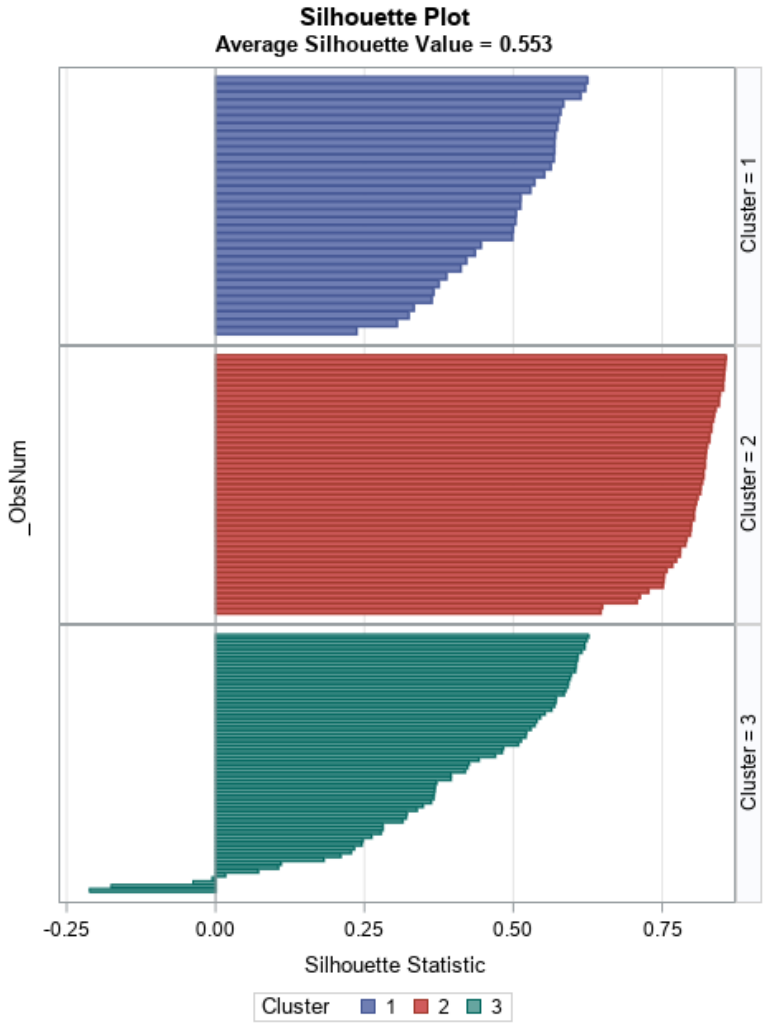
$$s(o) = \frac{b(o) - a(o)}{\max\{a(o);\, b(o)\}}$$

which yields $-1 \leq s(o) \leq 1$.

Interpretation: values near $+1$ indicate that o is well matched to its own cluster and far from others. Values near 0 indicate that o lies between two clusters. Negative values indicate misclassification (on average o is closer to another cluster than to its own). We recommend using the average silhouette over all objects as an intrinsic index of overall Clustering quality and the average silhouette per cluster to inspect cluster fitness.

Operational steps to compute silhouette-based diagnostics are:

1. For each point o compute $a(o)$, the mean intra-cluster distance;
2. For each other cluster compute the mean distance from o to that cluster and take the minimum as $b(o)$;
3. Compute $s(o) = (b(o) - a(o))/\max\{a(o); b(o)\}$;
4. Summarize by averaging $s(o)$ over points in a cluster (cluster quality) and over the whole dataset (global quality). Use silhouette plots to inspect cluster separation and to spot poorly assigned points.



Final Advices

The Elbow method is heuristic and sometimes ambiguous because the "elbow" is not always clear, cross-validation for Clustering requires a sensible fit metric (e.g., Sum of Squared Errors for centroid models) and can be computationally costly, extrinsic measures are only available when a reliable ground truth exists and must be chosen to reflect the application's tradeoffs (e.g., whether small cluster preservation is important). Intrinsic indices such as silhouette provide useful guidance but can be misleading in high dimensions or when cluster shapes deviate strongly from the distance metric's assumptions. The Hopkins test helps avoid wasting effort on uniformly random data but itself requires careful sampling and multiple repetitions for reliability.

4.9.2 Is Silhouette useful for evaluating clusters found by DENCLUE?

Generally, **no**. While the silhouette coefficient can be calculated for any Clustering, it is often misleading for **DENCLUE**.

DENCLUE is a Density-Based algorithm designed to find clusters of *arbitrary shapes* (non-convex, elongated, or nested) based on density attractors. Silhouette relies on Euclidean distances to measure cohesion and separation. It assumes that a "good" cluster is spherical or convex, where every point is close to the center. In a non-convex cluster found by DENCLUE (e.g., a "U" shape), points at opposite ends are density-connected, but geographically far apart. As a result, the value of $a(o)$ (internal cohesion) becomes very high, making the object appear to be poorly integrated into its cluster. In the presence of concave shapes, a point o may be physically very close to a point in another cluster, even though it is separated from it by a "gap". In this case, $b(o)$ becomes small. If $a(o)$ is large and $b(o)$ is small, the silhouette coefficient will be close to 0 or even negative, falsely signaling a Clustering error when, in fact, the algorithm correctly identified the complex shape. Silhouette would assign these points a low score due to poor cohesion, wrongly suggesting the cluster is invalid.

4.9.3 Why were BCubed Precision and Recall introduced if they can't be used in real applications without known class labels?

The primary reason these indices exist is for **Benchmarking**. When a researcher proposes a new Clustering algorithm, they must prove it works. They apply the algorithm to a dataset where the ground truth is already known, and if the algorithm achieves a high BCubed scores on these known datasets, it provides scientific evidence that the algorithm is capable of discovering meaningful patterns. Without ground truth for comparison, it would be impossible to objectively state that Algorithm A is "better" than Algorithm B.

4.9.4 How to compare the performance of two Clustering methods?

Compare the two Clustering methods by evaluating each against a reference (*ground-truth*) labeling using BCubed Scores (**BCubed Precision** and **BCubed Recall**).

BCubed Precision measures cluster purity.

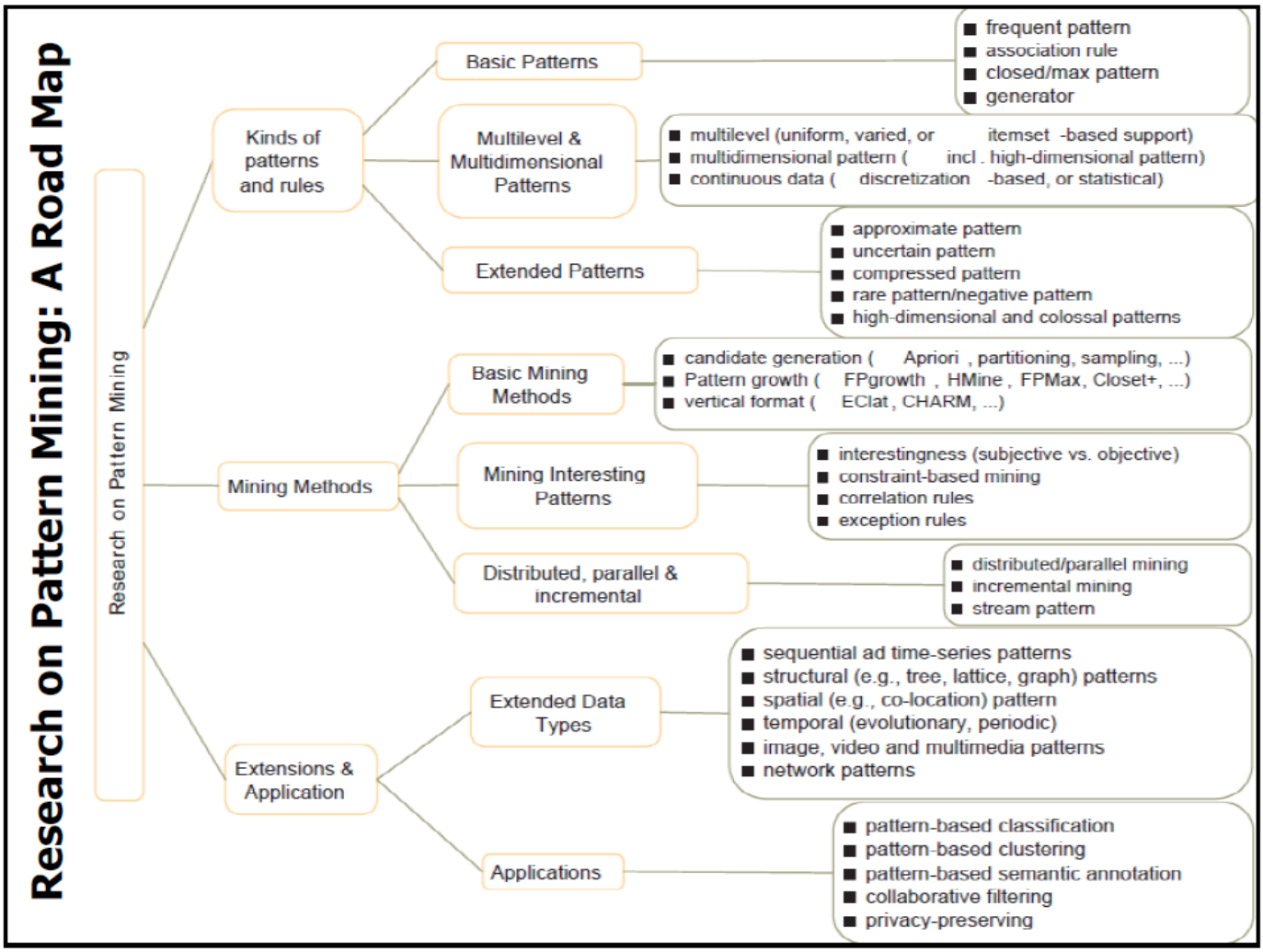
BCubed Recall measures cluster (label) completeness.

Compute BCubed Precision and Recall for both methods and compare the resulting scores (or their harmonic mean, **BCubed F-score**). Higher values indicate better Clustering performance.

4.9.5 What are the advantages of the K-Medoids algorithm over K-Means?

K-Medoids is more robust to noise and outliers because it uses real objects from the dataset (medoids) as cluster centers instead of the arithmetic mean (centroids), which can be strongly influenced by extreme values. Furthermore, it can be applied to non-numeric data types if a dissimilarity matrix is available.

Chapter 5: Pattern Mining



5.1 Mining Frequent Patterns

5.1.1 What is the Support-Confidence framework for Association Rules?

The *Support-Confidence framework* is the fundamental approach used in **Pattern Mining** to identify strong relationships (*pattern*) between *items* in large databases. An **Itemset** I is a set of one or more items. A **Transaction** T is a set of items such that $T \subset I$. An itemset containing k elements is formally referred to as a k-itemset. An **Association Rule** is typically expressed in the form $X \Rightarrow Y$, where X and Y are disjoint itemsets. The framework relies on two primary metrics to evaluate the "*strength*" and "*interestingness*" of these rules: **Support** and **Confidence**.

Support represents the frequency or popularity of an itemset within the dataset. Formally, the Support of a rule $X \Rightarrow Y$ is the probability that a transaction contains both X and Y, calculated as:

$$supp(X \Rightarrow Y) = P(X \cup Y)$$

It serves as a measure of significance. Rules with very low support occur too infrequently to be useful for most business or analytical decisions. Generally $P(X \cup Y) = \frac{\text{number of transactions with } X \text{ and } Y}{\text{number of transactions}}$.

Confidence measures the reliability of the inference (relation) made by the rule. It is defined as the conditional probability that a transaction contains itemset Y given that it contains itemset X, expressed as:

$$conf(X \Rightarrow Y) = P(Y|X) = \frac{supp(X \Rightarrow Y)}{supp(X)}$$

A high confidence value indicates that the rule is highly predictive within the given dataset.

In practice, **the mining process is controlled by user-defined thresholds**: **Minimum Support** (min_sup) and **Minimum Confidence** (min_conf). A rule is considered "strong" if it satisfies both thresholds simultaneously. The discovery of these rules usually follows a two-step process:

1. **Frequent Itemset Generation**: Identify all sets of items that satisfy the minimum support threshold min_sup. This step is computationally the most expensive part of the process because the number of potential itemsets grows exponentially with the number of items;
2. **Rule Generation**: From the frequent itemsets, extract high-confidence rules that meet the minimum confidence threshold min_conf. Since the support values are already known from the first step, this stage is relatively straightforward.

While effective for many applications, the framework has limitations, such as the potential for generating misleading rules when certain items are very frequent, or failing to capture interesting relationships between rare items. To address these issues, additional measures like **Lift** or the χ^2 test are often used alongside support and confidence to evaluate correlation.

5.1.2 What are Closed Itemset and Max-Itemset?

A single **frequent itemset** of length k contains $2^k - 1$ frequent sub-itemsets, which can lead to a prohibitively large number of patterns in datasets with long transactions, making it impossible to store all frequent itemsets (**combinatory explosion**). To manage this complexity, researchers use condensed representations.

An itemset X is considered a **closed itemset** in a dataset D, if it is frequent and there exists no proper super-itemset $Y \supset X$ that possesses the same support count as X. Essentially, **a closed itemset is the largest set of items common to a specific set of transactions**. The collection of closed frequent itemsets is particularly valuable because it provides a **lossless compression** of all frequent patterns. This means that by knowing only the closed itemsets and their support counts, it is possible to derive the complete set of frequent itemsets and their exact support levels without re-scanning the original database. *They are used to reduce the number of patterns while keeping exact subset support.*

A **max-itemset**, or maximal frequent itemset, is a frequent itemset X in a dataset D such that there exists no frequent super-itemset $Y \supset X$ (frequent means that it exceeds the min_sup threshold). This definition implies that **any itemset that is a superset of a max-itemset must be infrequent according to the downward closure property**. While max-itemsets provide an even more compact representation than closed itemsets, they are considered a *lossy compression*. Although they identify which sets are frequent, they do not contain sufficient information to reconstruct the exact support counts of their subsets. They only guarantee that the subsets are frequent. *They are used to identify frequent subsets without exact support details.*

The relationship between these concepts is **hierarchical**. **Every maximal frequent itemset is a closed frequent itemset, but the inverse is not necessarily true.** Mining these condensed patterns is essential for improving the scalability of association rule discovery in large, high-dimensional databases, as it significantly reduces the number of patterns and rules that must be generated and analyzed.

Which one to offer to users?

Neither, because they only indicate co-occurrence (i.e., which elements appear together). **Association Rules** are offered because they transform a simple statistical pattern (itemset) into logical, actionable information.

5.1.3 What is the Apriori Property?

The **Apriori Property**, also known as the *downward closure property of frequent patterns*, is the fundamental principle used to reduce the search space in Association Rule Mining. **This property states that any non-empty subset of a frequent itemset must also be frequent. The logical basis for this principle is that if an itemset I does not appear in at least a minimum number of transactions, then no larger itemset containing I can possibly appear more frequently than I itself.**

This property belongs to a category of mathematical properties called **antimonotonicity**. In practical terms, it means that if a set cannot pass a specific test (such as meeting a minimum support threshold), then all of its supersets will also fail that same test. For example, if the itemset $\{beer, diaper, nuts\}$ is found to be frequent, it is guaranteed that $\{beer, diaper\}$, $\{beer, nuts\}$, and $\{diaper, nuts\}$ are also frequent because every transaction containing the triplet must, by definition, contain its pairs.

The importance of the Apriori property lies in its power to prune the exponential search space of candidate itemsets. Without this property, a database with d items would require checking 2^d possible combinations. This "pruning" ensures that the algorithm only generates and tests candidate itemsets whose subsets are already known to be frequent, significantly improving computational efficiency and scalability for large datasets.

5.1.4 Talk about Apriori Algorithm

The **Apriori Algorithm** is the pivotal approach for frequent itemset mining and serves as the foundation for modern association rule discovery. **Its primary objective is to identify sets of items that appear together in a database more frequently than a user-specified threshold.** The algorithm is characterized by an iterative level-wise search strategy where frequent k-itemsets are used to explore and generate $(k+1)$-itemsets. This process ensures that the search space is navigated systematically from the most basic patterns to the most complex ones. Apriori algorithm follows a **candidate-generation-and-test paradigm**.

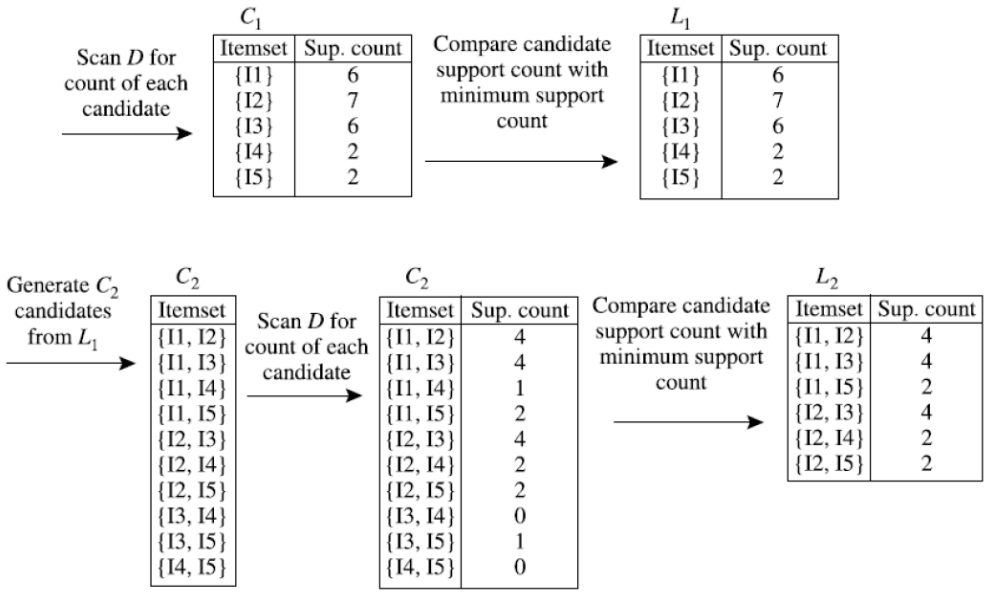
The **database** can be considered as a **parameter**, but the **main parameter** controlling the algorithm is the **Minimum Support threshold** (min_sup), that defines the minimum frequency or probability required for an itemset to be considered "frequent".

The functionality of the algorithm follows a structured candidate generation-and-test approach described by the following steps:

1. **Initial Scan (L_1):** Scan the database once to count occurrences and identify itemsets that satisfy the min_sup threshold to form the frequent 1-itemsets, L_1;
2. **Join Step (C_k):** For $k \geq 2$, generate a set of candidate k-itemsets C_k by joining L_{k-1} with itself ($L_{k-1} \bowtie L_{k-1}$). Two $(k-1)$-itemsets are joinable only if their first $(k-2)$ items are identical and they are ordered lexicographically;
3. **Prune Step:** Apply the **Apriori pruning principle** (based on the downward closure property) to immediately remove any candidate $c \in C_k$ if any of its $(k-1)$-subsets are not frequent (i.e., not present in L_{k-1});
4. **Candidate Testing (Scanning):** Perform a database scan to calculate the actual support count for each remaining candidate in C_k;
5. **Formation of L_k:** Retain only those candidates that meet the *minimum support threshold* to form the frequent k-itemsets, L_k. Increment k and repeat the iterative cycle starting from the *Join Step* at point (2);
6. **Termination:** Conclude the execution when no further frequent itemsets ($L_k = \emptyset$) or candidate sets can't be generated, and return the union of all discovered frequent patterns ($L = \bigcup_k L_k$).

TID	List of item_IDs
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

Let us suppose that the minimum support is 2



(a) Join: $C_3 = L_2 \bowtie L_2 = \{\{I1, I2\}, \{I1, I3\}, \{I1, I5\}, \{I2, I3\}, \{I2, I4\}, \{I2, I5\}\}$
$\bowtie \{\{I1, I2\}, \{I1, I3\}, \{I1, I5\}, \{I2, I3\}, \{I2, I4\}, \{I2, I5\}\}$
$= \{\{I1, I2, I3\}, \{I1, I2, I5\}, \{I1, I3, I5\}, \{I2, I3, I4\}, \{I2, I3, I5\}, \{I2, I4, I5\}\}$.

(b) Prune using the Apriori property: All nonempty subsets of a frequent itemset must also be frequent. Do any of the candidates have a subset that is not frequent?

- The 2-item subsets of {I1, I2, I3} are {I1, I2}, {I1, I3}, and {I2, I3}. All 2-item subsets of {I1, I2, I3} are members of L_2. Therefore, keep {I1, I2, I3} in C_3.
- The 2-item subsets of {I1, I2, I5} are {I1, I2}, {I1, I5}, and {I2, I5}. All 2-item subsets of {I1, I2, I5} are members of L_2. Therefore, keep {I1, I2, I5} in C_3.
- The 2-item subsets of {I1, I3, I5} are {I1, I3}, {I1, I5}, and {I3, I5}. {I3, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I1, I3, I5} from C_3.
- The 2-item subsets of {I2, I3, I4} are {I2, I3}, {I2, I4}, and {I3, I4}. {I3, I4} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I3, I4} from C_3.
- The 2-item subsets of {I2, I3, I5} are {I2, I3}, {I2, I5}, and {I3, I5}. {I3, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I3, I5} from C_3.
- The 2-item subsets of {I2, I4, I5} are {I2, I4}, {I2, I5}, and {I4, I5}. {I4, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I4, I5} from C_3.

(c) Therefore, $C_3 = \{\{I1, I2, I3\}, \{I1, I2, I5\}\}$ after pruning.

Phase 1 partitions the database into n disjoint segments that fit in memory and mines each partition independently to find local frequent itemsets. Any globally frequent itemset must appear as frequent in at least one partition, so all local frequent itemsets are merged to form the global candidate set.

Phase 2 performs a single additional scan of the entire database to compute the true global support of these candidates. Itemsets that do not satisfy the global minimum support threshold are discarded, leaving only the globally frequent itemsets.

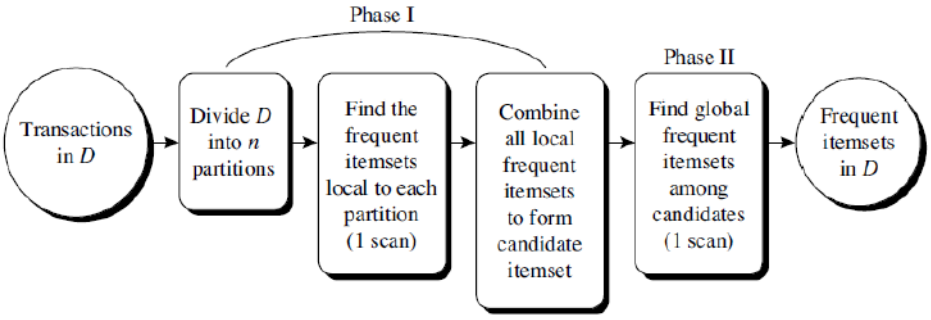
The Apriori Algorithm offers several **advantages**:

- Simple and intuitive algorithm, making it easy to understand and implement;
- Effectively exploits the *downward closure (Apriori) property*, ensuring that only itemsets whose subsets are frequent are considered;
- Significantly reduces the search space by pruning infrequent candidate itemsets early.

However it also presents some **limitations**:

- Requires multiple full database scans (one for each itemset size), leading to high I/O overhead;
- Candidate itemset generation is computationally expensive in both time and memory;
- Performs poorly when the number of items is large or when the minimum support threshold is low;
- In extreme cases, such as discovering a frequent pattern of size 100, it may theoretically require generating $2^{100} - 1$ candidate itemsets.

To address these limitations, several improvements have been developed. The **partitioning method** divides the database into n segments and assumes that any **globally frequent itemset must be locally frequent in at least one partition**. This approach **reduces the total number of database scans to just two**.



Another technique is the **Direct Hashing and Pruning** (**DHP**) **method**, which uses a hash table to filter out candidate 2-itemsets during the initial scan. DHP first scans the database to find frequent 1-itemsets while hashing all 2-itemsets. Buckets below the support threshold are discarded, producing a smaller C_2. For larger itemsets, it continues candidate generation with database pruning and iterative hashing until no frequent itemsets remain. Its main strength is the substantial reduction of candidate sets and scans, though its performance depends on hash quality and incurs additional overhead from managing hash tables.

The complexity of this mining process is sensitive to the support threshold. At low thresholds, the number of frequent itemsets can grow exponentially toward the worst-case scenario of M^N, where M is the number of distinct items and N is the maximum transaction length.

5.1.5 How can you generate Association Rules from Frequent Itemsets?

Association Rules can be generated from **Frequent Itemsets** as follows:

1. For each frequent itemset I, generate all non-empty subsets of I;
2. For every non-empty subset s of I, output the rule $s \Rightarrow (I - s)$ if $conf(s \Rightarrow (I - s)) = \frac{supp(I)}{supp(s)} \geq min\_conf$, where min_conf is the *Minimum Confidence threshold.*

Support suffers from the **rare item problem**, where items that occur infrequently in the dataset are pruned, even though they may lead to interesting and potentially valuable association rules. This issue is particularly significant in transaction databases, which often exhibit highly uneven support distributions across items.

Confidence also has inherent limitations, as it is sensitive to the **frequency of the consequent** in the database. Due to its definition, rules with highly frequent consequents tend to achieve higher confidence values, even when no true association exists between the antecedent and the consequent.

5.1.6 Talk about FP-Growth Algorithm

The **FP-Growth** (**Frequent Pattern Growth**) **algorithm** is an efficient and scalable method for mining the complete set of frequent itemsets without the need for candidate generation. Unlike the Apriori algorithm, which follows a *candidate-generation-and-test paradigm*, FP-Growth adopts a **divide-and-conquer** strategy that *compresses the database into a specialized data structure and then decomposes the mining task into a set of smaller, conditional mining tasks.* This approach significantly reduces the computational overhead by avoiding the exponential number of candidate sets that often plague level-wise algorithms.

The primary parameter required by the FP-Growth algorithm is the **Minimum Support threshold** (*min_sup*), which determines whether an itemset is frequent enough to be extracted. Similar to other Association Rule Mining techniques, this threshold is used to filter out noise and focus on statistically significant patterns within the transaction database.

The core of the algorithm is the **Frequent-Pattern Tree** (**FP-Tree**), a highly compressed representation of the transaction database. The tree is constructed such that frequent items are stored in nodes and overlapping transactions share common prefixes in the tree branches. A **Header Table** is maintained alongside the tree to store the total support count of each frequent item and to provide links to the first occurrence of that item in the tree, allowing for efficient traversal.

The functionality of the FP-Growth algorithm is divided into two main phases, described by the following steps:

1. **FP-Tree Construction:**
   (a) Scan the database once to identify frequent 1-itemsets (L_1) and sort them in descending order of their support frequency;
   (b) Perform a second database scan to construct the FP-Tree by inserting each transaction into the tree, following the sorted order of frequent items and incrementing node counts where paths overlap;
   (c) For each frequent item in the *Header Table*, starting from the one with the lowest frequency, extract its Conditional Pattern Base, which consists of all prefix paths in the FP-Tree leading to that item.
2. **Frequent Itemset Generation:**
   (a) Construct a Conditional FP-Tree from the accumulated patterns in the Conditional Pattern Base (in the Header Table), treating the current item as a suffix (last one);
   (b) Recursively mine the Conditional FP-Tree to identify frequent patterns and terminate the recursion when the tree contains only a single path or becomes empty.

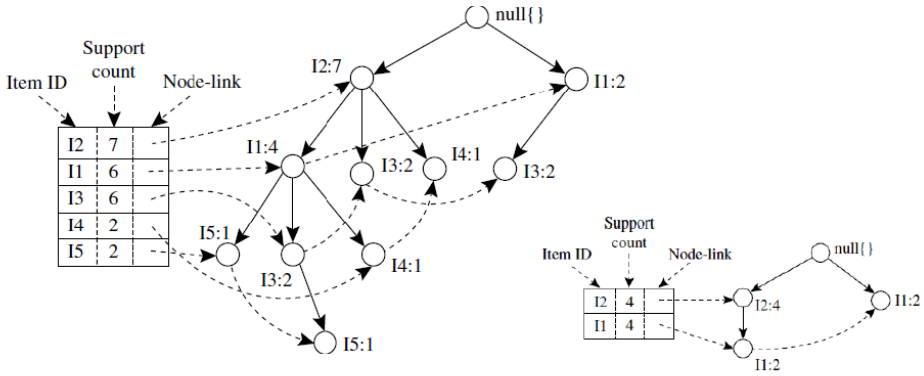
A critical component of the second phase (*Frequent Itemset Generation*) is the **Conditional FP-Tree**. This is a reduced version of the original FP-Tree that only contains transactions (or parts of transactions) that co-occur with a specific suffix item. **By building these local trees, the algorithm can find frequent patterns containing that suffix without having to look at the entire global tree again**. This localized search is what makes the algorithm exceptionally fast for dense datasets.

The FP-Growth Algorithm offers several **advantages**:

- It eliminates the need for candidate generation, which is the most expensive part of the Apriori algorithm;
- It requires only two full scans of the transaction database, significantly reducing I/O operations compared to level-wise approaches;
- The compressed FP-Tree representation often fits in the main memory, allowing for high-performance processing;
- It is generally several orders of magnitude faster than Apriori, especially as the Support threshold decreases.

However it also presents some **limitations**:

- The FP-Tree can still become very large and may not fit in memory if the database is extremely large or the data is highly sparse;
- The construction and recursive mining of the tree are more complex to implement and debug than simpler candidate-based methods;
- The overhead of building and managing conditional trees can be significant when there are a vast number of unique items.



Item	Conditional Pattern Base	Conditional FP-tree	Frequent Patterns Generated
I5	{{I2, I1: 1}, {I2, I1, I3: 1}}	⟨I2: 2, I1: 2⟩	{I2, I5: 2}, {I1, I5: 2}, {I2, I1, I5: 2}
I4	{{I2, I1: 1}, {I2: 1}}	⟨I2: 2⟩	{I2, I4: 2}
I3	{{I2, I1: 2}, {I2: 2}, {I1: 2}}	⟨I2: 4, I1: 2⟩, ⟨I1: 2⟩	{I2, I3: 4}, {I1, I3: 4}, {I2, I1, I3: 2}
I1	{{I2: 4}}	⟨I2: 4⟩	{I2, I1: 4}

FP-Growth is a fundamental improvement in Frequent Pattern Mining because it retains the complete information of frequent patterns while achieving a high level of compression. By transforming the mining problem into a recursive tree-traversal problem, it avoids the combinatorial explosion of candidates and scales linearly with the number of transactions.

5.1.7 Talk about ECLAT Algorithm

The **ECLAT algorithm** (**Equivalence CLAss Transformation**) is a frequent itemset mining method that departs from the traditional horizontal data scanning approach used by Apriori. While Apriori explores the search space using a *breadth-first search* and multiple database passes, ECLAT utilizes a **vertical data format** and a **depth-first search** strategy to discover patterns. This transformation allows the algorithm to determine the Support of an itemset through a simple **intersection** of transaction identifiers rather than repeated scans of the entire database.

In a standard transaction database, data is usually stored in a *horizontal format* where each record consists of a **Transaction ID** (TID) followed by a list of items. ECLAT transforms this into a **vertical format** where each item is associated with a **TID-set**, which is a list of all transactions in which that specific item appears. The *cardinality* of this TID-set represents the exact **support count** of the item. To find the support of a candidate pair like $\{A, B\}$, the algorithm simply intersects the TID-set of A with the TID-set of B. The resulting intersection contains only the TIDs where both items co-occur.

The functionality of the ECLAT algorithm is defined by the following procedural steps:

1. Transform the transaction database from the traditional *horizontal data format* into the **vertical data format** by creating TID-sets for every individual item;
2. Identify frequent 1-itemsets by checking if the size of each **TID-set** meets min_sup;
3. Generate candidate $(k+1)$-itemsets by combining frequent k-itemsets that share a common *prefix*;
4. Calculate the support of the new candidates by performing a **set intersection** of the TID-sets of the generating items;
5. Recursively repeat the process in a *depth-first* manner within each *equivalence class* until no further frequent itemsets can be found.

The ECLAT Algorithm offers several **advantages**:

- It is generally faster than the Apriori algorithm because it avoids the heavy computational cost of repeated database scans;
- The **intersection** of sets is a very efficient operation that can be further optimized using *bit-vectors* to speed up processing;
- It naturally supports a **parallel implementation** because the search space is divided into independent *sub-lattices* based on prefixes;
- There is no need for complex *candidate generation* or testing against the entire database after the initial transformation.

However it also presents some **limitations**:

- The primary drawback is the **memory consumption** required to store the TID-sets, especially for very large databases with many transactions;
- If the TID-sets are too large to fit into the *main memory*, the performance of the algorithm degrades significantly due to swapping;
- While highly effective for *dense datasets*, it may be less efficient than other methods for extremely *sparse data* where intersections yield very few results.

5.1.8 Which Patterns Are Interesting?

The evaluation of pattern interestingness is a critical step in ***Data Mining and Machine Learning***, as the **Support-Confidence framework** often generates a vast number of rules, many of which are redundant or even misleading. A pattern is generally considered **interesting** if it is easily understood by humans, valid on new data with some degree of certainty, potentially useful, and novel. While *objective measures* like **Support** and **Confidence** provide a **baseline for filtering**, they are frequently *insufficient to identify truly significant relationships.*

A major limitation of the Support-Confidence framework is that it ignores the *underlying frequency* of the consequent in a rule. For instance, a rule $A \Rightarrow B$ may have very high confidence simply because B is an extremely frequent item, even if the presence of A actually decreases the likelihood of B occurring. To address this, **correlation analysis** is employed using measures such as **Lift**. Lift calculates the ratio of the observed support of $A \cup B$ to the expected support if A and B were independent.

$$\text{lift}(A \Rightarrow B) = \frac{\text{conf}(A \Rightarrow B)}{\text{supp}(B)} = \frac{\text{sup}(A \Rightarrow B)}{\text{supp}(A) \cdot \text{supp}(B)} = \frac{P(A \cup B)}{P(A)P(B)}$$

A lift value greater than 1 indicates a **positive correlation**, a value less than 1 indicates a **negative correlation**, and a value of 1 suggests that the two itemsets are **independent**.

Similarly, the χ^2 **(chi-square) analysis** can be used to statistically test the hypothesis of independence between items by comparing observed frequencies against expected frequencies in a *contingency table.*

In large-scale databases, many measures are sensitive to the total number of transactions that do not contain any of the items being studied, known as **null transactions**. This leads to the concept of **null-invariance**, which is a highly desirable property for interestingness measures in *sparse datasets.* A measure is null-invariant if its value does not change when the number of null transactions increases. While *Lift* and χ^2 are not null-invariant, several other measures (with values between 0 and 1) satisfy this property:

- **all_confidence**: $\text{all\_conf}(A;B) = \frac{\text{supp}(A \cup B)}{\max\{\text{supp}(A);\text{supp}(B)\}} = \min\{P(A|B); P(B|A)\}$
- **max_confidence**: $\text{max\_conf}(A;B) = \max\{P(A|B); P(B|A)\}$
- **Kulczynski (Kulc)**: $\text{Kulc}(A;B) = \frac{1}{2}(P(A|B) + P(B|A))$
- **Cosine Similarity**: $\text{cosine}(A;B) = \frac{P(A \cup B)}{\sqrt{P(A) \cdot P(B)}} = \frac{\text{supp}(A \cup B)}{\sqrt{\text{supp}(A) \cdot \text{supp}(B)}} = \sqrt{P(A|B) \cdot P(B|A)}$
- **Imbalance Ratio:** $\text{IR}(A;B) = \frac{|\text{supp}(A) - \text{supp}(B)|}{\text{supp}(A) + \text{supp}(B) - \text{supp}(A \cup B)}$

Furthermore, the IR is used to measure the skewness between two itemsets. A high IR indicates that one itemset appears much more frequently than the other, which can make the *Kulczynski measure* more informative than others. By combining these **null-invariant measures** with correlation analysis, practitioners can filter out "pseudo-frequent" patterns and focus on the truly *strong association patterns* that represent genuine regularities in the data.

5.2 Advanced Pattern Mining

5.2.1 How does Pattern Mining in Multi-Level Association work?

In many applications, items are organized into *concept hierarchies* or taxonomies, such as a product catalog where "*Milk*" is a parent of "*Skim Milk*". **Multi-level association mining** is the process of discovering frequent patterns at different levels of abstraction. This approach allows users to find general trends at higher levels and more specific, actionable insights at lower levels of the hierarchy.

The primary challenge in this type of mining is setting the **Minimum Support threshold** (min_sup) correctly across different levels. There are two main strategies for handling support:

- **Uniform Support**: The same min_sup is used for all levels. While simple to implement, it often leads to a "bottleneck" where top-level items are overwhelmingly frequent while bottom-level items fail to meet the threshold, resulting in the loss of detailed information;
- **Reduced Support**: Also referred to as *varied support*, this strategy applies lower min_sup thresholds at lower levels of the hierarchy. This ensures that specific items, which naturally appear less frequently than their general categories, can still be identified as part of a frequent pattern.

The mining process typically follows a top-down approach: frequent items are identified at higher levels, expanded to their descendants, and pruned when parent items are infrequent. Finally, redundancy filtering removes lower-level rules that do not provide additional information beyond their ancestors, ensuring that only refined and informative patterns are retained.

A pattern is considered *ancestor-redundant* if its support and confidence are close to the "expected" values derived from its parent. For example, if "Milk ⇒ Bread" is a strong rule, then "Low-fat Milk ⇒ Wheat Bread" might be considered redundant unless it shows a significantly different correlation. By filtering these, the algorithm focuses only on **refined patterns** that offer unique value.

5.2.2 How does Pattern Mining in Multi-Dimensional Association work?

Multi-dimensional association mining extends single-dimensional mining by discovering relationships across multiple predicates or dimensions, such as demographic, geographic, and transactional attributes. The resulting rules are typically classified as **inter-dimension rules**, where each predicate appears only once, or **hybrid-dimension rules**, which allow repeated predicates within a rule.

A key challenge is handling **quantitative attributes** with continuous values. These are managed through **discretization**, which converts numerical data into intervals using either predefined (static) ranges or data-driven (dynamic) methods.

To efficiently explore patterns across multiple dimensions and levels of detail, multi-dimensional mining often relies on **data cube** technology. Pre-computed aggregates, combined with techniques such as binning or Clustering, enable efficient discovery of frequent, meaningful, and interpretable association rules.

5.2.3 How does Pattern Mining in Quantitative Associations work?

Quantitative Association Rules describe relationships between attributes that contain **numerical values**, such as *age*, *income*, or *price*. Unlike categorical data, these continuous values must be transformed into *discrete intervals* so that standard Frequent Pattern Mining algorithms can process them. This process is essentially a way of mapping a *quantitative problem* onto a *boolean association rule* framework.

The most common approach to handling these attributes is **discretization**. This can be performed in three primary ways:

- **Static Discretization**: Numeric ranges are predefined based on *expert knowledge* or simple mathematical splits like *equi-width* or *equi-depth* binning;
- **Dynamic Discretization**: The algorithm determines the optimal *interval boundaries* by analyzing the **data distribution** to identify the most statistically significant or "interesting" ranges during the mining process;
- **Clustering-based Mining: where numerical data points are grouped into *clusters*. These clusters then serve as the discrete items used to generate rules.**

The final output usually takes the form of an association like "$Age[30-35] \cap Income[50k-70k] \Rightarrow Buy(Luxury\_Car)$," where each range represents a *frequent itemset* discovered in the transformed space.

5.2.4 How does Pattern Mining find rare and negative patterns?

Standard Frequent Pattern Mining focuses on itemsets that exceed a global minimum support threshold, but this often overlooks **rare patterns** and **negative patterns**, which can be highly informative in specialized domains like medical diagnosis or fraud detection.

Rare patterns consist of items that occur infrequently but are strongly associated when they do appear. The primary challenge is the *rare item problem*, where a high *min_sup* filters out rare items, while a low *min_sup* causes a combinatorial explosion of trivial patterns. To solve this, researchers use a **multiple minimum support (MMS)** model, allowing users to specify different support thresholds for different items. This ensures that *infrequent but significant* items, such as expensive luxury goods, are not discarded during the pruning phase.

Negative patterns describe the *absence* of items or a *negative correlation* between them, typically expressed in the form $X \Rightarrow \neg Y$. A common approach to identify these is to find itemsets X and Y that are individually frequent but whose combined support $\sup(X \cup Y)$ is significantly lower than the expected support if they were independent. To distinguish *genuine negative associations* from random infrequent occurrences, the following criteria are often applied:

- Both itemsets X and Y must satisfy the *minimum support* threshold individually;
- The support of the combined set must be less than the product of their individual supports: $\sup(X \cup Y) < \sup(X) \cdot \sup(Y)$;
- A **correlation measure**, such as *Lift* or *Pearson's correlation*, must fall below a specific negative threshold to confirm a strong **inhibitory relationship**.

By moving beyond simple frequency, these methods allow for the discovery of **contradictory relationships** and *niche regularities* that are invisible to standard Association Rule Mining algorithms.

5.2.5 How does Constrained Pattern Mining work?

Constrained Pattern Mining is a specialized branch of data mining that allows users to guide the discovery process by specifying *user-defined constraints*. In standard Frequent Pattern Mining, the algorithm generates all patterns meeting a minimum support threshold, which often results in a massive number of uninteresting results. By incorporating constraints such as specific attribute ranges, price limits, or item categories, the system can focus on the **semantic interest** of the user while significantly improving *computational efficiency* through early *search space pruning*.

The effectiveness of constrained mining depends on the mathematical properties of the constraints, which determine how they can be "pushed" into the mining algorithm. Constraints are primarily categorized into four classes:

- **Anti-monotone constraints** are those where, if an itemset S violates the constraint, then every superset of S will also violate it. The most common example is the *support threshold* itself. If $\{beer\}$ is infrequent, any set containing beer is also infrequent. In a *bottom-up* search like Apriori, if a candidate fails an anti-monotone constraint, the algorithm can safely discard all of its descendants in the search lattice. Other examples include $sum(S.price) \leq v$ (assuming non-negative prices) and $count(S) \leq v$.
- **Monotone constraints** exhibit the opposite behavior: if an itemset S satisfies the constraint, then every superset of S also satisfies it. For example, $sum(S.price) \geq v$ is monotone. While these constraints cannot prune the search space as aggressively as anti-monotone ones, they allow the algorithm to stop checking the constraint for any superset once it has been satisfied by a prefix, thereby reducing the *computational load*.
- **Succinct constraints** are those where all the itemsets that satisfy the constraint can be generated *directly* without the need for a "generate-and-test" approach. This is achieved by defining a *precise set of items* that can possibly form valid itemsets. For instance, a constraint like $min(S.price) > 50$ is succinct because we can simply remove all items with a price of 50 or less from the database before the mining begins. This ensures that every generated candidate is **guaranteed** to satisfy the constraint.
- **Convertible constraints** are more complex because they do not naturally fit into the anti-monotone or monotone categories. However, they can be *converted* into one of these types by imposing a specific **lexicographic or value-based order** on the items. A typical example is $avg(S.price) \geq v$. If items are sorted in *descending order* of price, the constraint becomes *prefix-anti-monotone*, meaning that if a prefix does not satisfy the average, no further expansion of that prefix can satisfy it. This property is particularly useful in **FP-Growth** and other *pattern-growth* methods.

By utilizing these properties, constrained mining transforms a potentially *intractable search* into a directed and efficient process, ensuring that the discovered **knowledge** is both relevant to the user's specific goals and computationally feasible to obtain from high-dimensional datasets. A constraint is considered **strongly convertible** if it is both convertible anti-monotone and convertible monotone.

Different constraints may require conflicting item orderings. If an order R exists such that both C_1 and C_2 are convertible with respect to R, then no conflict occurs. Otherwise, one constraint is satisfied first, and the induced order is used to mine frequent itemsets from the projected database.

Constraint	Anti-monotone	Monotone	Succinct
$v \in S$	no	yes	yes
$S \supseteq V$	no	yes	yes
$S \subseteq V$	yes	no	yes
$\min(S) \leq v$	no	yes	yes
$\min(S) \geq v$	yes	no	yes
$\max(S) \leq v$	yes	no	yes
$\max(S) \geq v$	no	yes	yes
$\text{count}(S) \leq v$	yes	no	weakly
$\text{count}(S) \geq v$	no	yes	weakly
$\text{sum}(S) \leq v\ (a \in S, a \geq 0)$	yes	no	no
$\text{sum}(S) \geq v\ (a \in S, a \geq 0)$	no	yes	no
$\text{range}(S) \leq v$	yes	no	no
$\text{range}(S) \geq v$	no	yes	no
$\text{avg}(S)\ \theta\ v, \theta \in \{=, \leq, \geq\}$	convertible	convertible	no
$\text{support}(S) \geq \xi$	yes	no	no
$\text{support}(S) \leq \xi$	no	yes	no

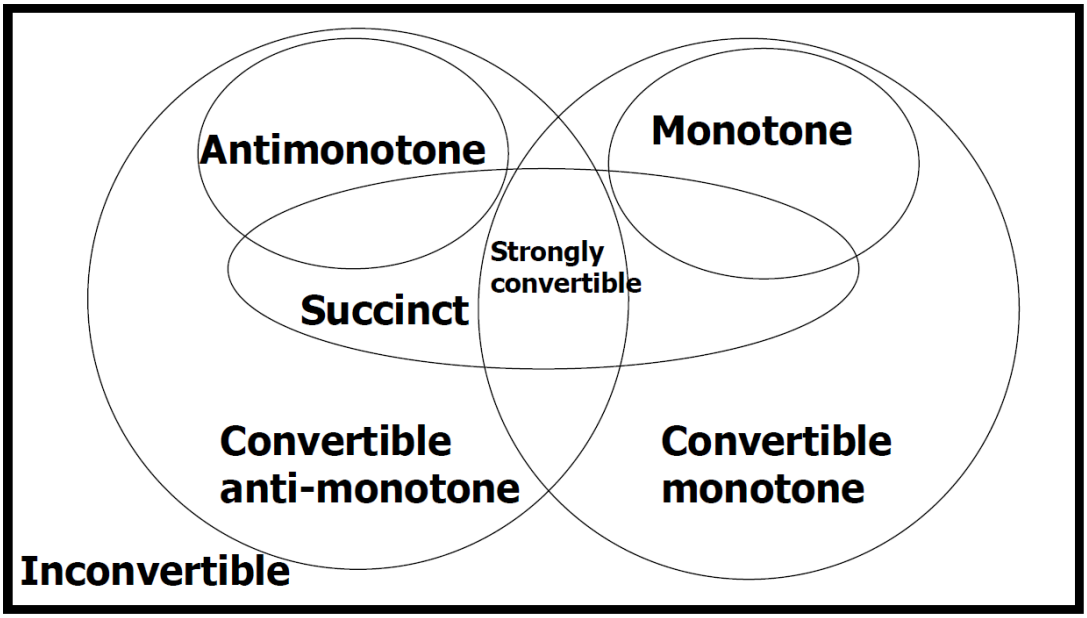
5.2.6 Can Apriori deal with Constraints?

The **Apriori Algorithm** can be effectively extended to handle various types of *user-defined constraints*, allowing the mining process to be more efficient and focused on relevant patterns. Instead of performing a post-processing filter on all discovered frequent itemsets, the **constraint-pushing** strategy integrates these rules directly into the search process. This ensures that the algorithm prunes the search space as early as possible, avoiding the generation of candidate itemsets that are guaranteed to violate the constraints.

The ability of Apriori to handle a constraint depends largely on its mathematical properties, particularly **anti-monotonicity** and **succinctness**:

- **Anti-monotone constraints** are the most compatible with Apriori. Because the algorithm already uses the *support-based pruning* (which is itself anti-monotone), any other anti-monotone constraint, such as $sum(S.price) \leq 100$, can be checked during the **candidate generation** phase. If a candidate k-itemset violates the constraint, it is discarded immediately, and none of its supersets are ever generated or tested;
- **Monotone constraints**, like $sum(S.price) \geq 500$, behave differently. If an itemset fails the constraint, its supersets might still satisfy it, so they cannot be pruned immediately based on the constraint alone. However, once an itemset **satisfies** a monotone constraint, all of its frequent supersets are guaranteed to satisfy it, allowing the algorithm to bypass further constraint checks for that specific branch. In essence, Apriori can implement monotone constraints as a "test-saving" optimization, but fails to use them for "candidate-saving". A monotone constraint can be pushed into FP-Growth;
- **Succinct constraints** are handled even earlier, often during the initial *item filtering* pass. By identifying which items can possibly satisfy the constraint (e.g., only items with $color = red$), the algorithm can restrict the initial set of frequent 1-itemsets. This drastically reduces the size of the initial search space and ensures that all subsequent candidates are "born" satisfying the constraint.

Constraint	Convertible anti-monotone	Convertible monotone	Strongly convertible
avg(S) ≤ , ≥ v	Yes	Yes	Yes
median(S) ≤ , ≥ v	Yes	Yes	Yes
sum(S) ≤ v (items could be of any value, v ≥ 0)	Yes	No	No
sum(S) ≤ v (items could be of any value, v ≤ 0)	No	Yes	No
sum(S) ≥ v (items could be of any value, v ≥ 0)	No	Yes	No
sum(S) ≥ v (items could be of any value, v ≤ 0)	Yes	No	No
.....			



For constraints that are neither anti-monotone nor monotone, such as **convertible constraints** (e.g., $avg(S.price) \geq 50$), Apriori can still adapt by imposing a specific **sorting order** on the items. By processing items in descending order of price, the *average* constraint behaves like an anti-monotone constraint, enabling early pruning.

5.2.7 Talk about Naïve Algorithm

The **Naïve Algorithm** for *Constrained Pattern Mining* uses a simple generate-and-filter approach. It first applies a standard Frequent Pattern Mining method, such as **Apriori**, to find all patterns that satisfy the minimum support threshold, ignoring user constraints. The constraints are then applied in a post-processing step to remove irrelevant patterns.

The main parameters involved in this algorithm are the **minimum support** (min_sup), which determines the frequency requirement for an itemset, and the **Constraint Set** (C), which contains the *user-defined conditions* such as price ranges, item categories, or count limits.

The functionality and steps of the Naïve Algorithm are as follows:

1. Define the **minimum support** threshold and the list of *mathematical or categorical constraints* that the patterns must satisfy;
2. Execute the standard **Apriori algorithm** to discover the complete set of frequent itemsets within the database;
3. Store the entire collection of identified frequent patterns in a *temporary memory buffer*;
4. Iterate through each discovered frequent itemset and check if it fulfills the criteria specified in the **Constraint Set**;
5. Retain only the itemsets that pass the filtering test and output them as the final results.

The Naïve Algorithm offers several **advantages**:

- It is **easy to implement** because it does not require any internal modifications to the underlying mining engine or the pruning logic;
- The algorithm is *highly flexible* and can be used with any type of constraint regardless of whether it is **anti-monotone**, **monotone**, or **succinct**;
- It provides a *useful baseline* for measuring the performance gains of more sophisticated **constraint-pushing** techniques.

However it also presents some **limitations**:

- It is **computationally inefficient** because it wastes time and memory generating a massive number of patterns that are ultimately discarded;
- The algorithm fails to perform **early pruning**, meaning it explores branches of the search space that a *constraint-aware* algorithm would have ignored immediately;
- It is often *unscalable* for large datasets or low support thresholds where the number of frequent itemsets can grow **exponentially**, potentially leading to system crashes or timeouts.

While the Naïve approach is conceptually simple, it represents a *non-optimized* path. Modern data mining systems prefer **integrating constraints** directly into the mining process to transform the task into a **directed search**, thereby avoiding the *redundant computations* that characterize the Naïve method.

5.2.8 How to use Pattern Mining on High-Dimensional Data?

Mining frequent patterns in **high-dimensional data** presents a unique set of challenges that traditional algorithms like Apriori or FP-Growth cannot easily overcome. When a dataset contains thousands of dimensions or transactions with hundreds of items, the search space becomes *astronomical* because any frequent pattern of length k contains $2^k - 1$ frequent sub-patterns. This phenomenon leads to the discovery of a **pattern swamp**, a situation where the algorithm becomes trapped in a mid-sized search space, generating billions of redundant sub-patterns before ever reaching the truly significant *long patterns*.

To effectively handle high-dimensional spaces, the mining objective shifts from finding all frequent patterns to focusing specifically on **Colossal Patterns**. These are exceptionally long patterns that characterize the deep regularities of the data. Traditional bottom-up or top-down approaches are *computationally intractable* for colossal patterns because they must traverse every single subset. Instead, a strategy known as **Pattern-Fusion** is employed to "jump" across the search space. This methodology identifies small *representative core patterns* and merges them into larger structures, effectively bypassing the dense layers of the search lattice.

Another critical aspect of mining in high dimensions is the use of **condensed representations** such as *closed* and *maximal itemsets*. By focusing only on these sets, the algorithm reduces the size of the output without losing the essential **support information** required for analysis. In extremely dense datasets, *row-wise mining* may also be used as an alternative to column-wise approaches, as it scales better when the number of items is much larger than the number of transactions.

Furthermore, **stochastic search** and **bounded-error approximations** are often utilized to navigate high-dimensional data. Rather than guaranteeing a complete set of all possible patterns, these methods focus on **robustness** and **efficiency**, ensuring that the most dominant colossal patterns are identified within a reasonable timeframe. This paradigm shift from *exhaustive search* to **strategic fusion** is what allows modern data mining to remain functional in the era of "big data" and high-dimensional genomic or bio-inspired datasets.

5.2.9 Talk about Pattern-Fusion

The **Pattern-Fusion Algorithm** targets the discovery of **colossal patterns** in high-dimensional datasets. Unlike bottom-up methods such as Apriori or FP-Growth, which suffer from **pattern swamp** when patterns are very long, Pattern-Fusion avoids enumerating all sub-patterns by directly fusing smaller representative patterns to form larger ones, enabling efficient exploration of the search space.

The main parameters used in this algorithm include the **minimum support threshold** (min_sup), which defines the frequency required for a pattern to be interesting, and a **robustness parameter** (τ), which is used to define the **core patterns** that are likely to be part of a larger colossal structure.

A subpattern β is defined as a τ-core pattern of a frequent pattern α if it shares a similar support set such that the ratio of their support sizes satisfies $\frac{|D_\alpha|}{|D_\beta|} \geq \tau$ for $0 < \tau \leq 1$, where $|D_\alpha|$ represents the number of transactions containing α and τ is denoted as the core ratio. In terms of robustness, a pattern α is considered $(d;\tau)$-robust if d is the maximum number of items that can be removed from α such that the resulting pattern remains a τ-core pattern of α. This property implies that a $(d;\tau)$-robust pattern α possesses $\Omega(2^d)$ core patterns, confirming that colossal patterns are significantly more robust than smaller patterns. The distance between patterns α and β is defined as:

$$Dist(\alpha;\beta) = 1 - \frac{|D_\alpha \cap D_\beta|}{|D_\alpha \cup D_\beta|}.$$

This measure defines a bounding ball of radius $r(\tau) = 1 - \frac{1}{\frac{2}{\tau}-1}$, within which patterns are treated as core patterns and merged.

The functionality and steps of the Pattern-Fusion algorithm are as follows:

1. **Initialization:** Use an existing mining algorithm (like Apriori or FP-Growth) to discover the complete set of frequent patterns up to a small maximum size, typically 3, and build a initial pool;
2. **Seed Selection:** Randomly select K patterns as seeds, which are likely to be *core-descendants* of colossal patterns due to their robustness;
3. **Core Pattern Identification:** For each seed pattern, identify all patterns in the pool that fall within a *bounding ball* of radius $r(\tau)$. These are considered the **core patterns** of the same potential colossal pattern;
4. **Fusion Step:** Perform a "leap" by taking the union (fusion) of all patterns identified within each bounding ball. This agglomeration generates a set of much larger super-patterns in a single step;
5. **Refinement and Verification:** Since the fusion step is a shortcut, the resulting super-patterns are refined by checking their actual support in the database and keeping only the most representative;
6. **Termination:** Return the discovered colossal patterns that satisfy the min_sup and the representative criteria.

The Pattern-Fusion Algorithm offers several **advantages**:

- It is exceptionally efficient for **high-dimensional data** where traditional algorithms fail due to the exponential growth of sub-patterns;
- The algorithm successfully "jumps" over the *pattern swamp*, drastically reducing the number of intermediate patterns that need to be processed;
- The methodology is *robust* against minor noise in the data because it relies on the fusion of multiple representative fragments.

However it also presents some **limitations**:

- It is an **approximate mining method**, meaning it may not guarantee the discovery of every single frequent pattern if they do not contain identifiable core structures;
- It is less effective for **low-dimensional data** or datasets where the frequent patterns are short, as the overhead of fusion exceeds the benefits of skipping the search space;
- The *verification step* can still be costly if the fusion process generates a large number of false-positive candidate patterns.

Pattern-Fusion represents a paradigm shift in data mining. By utilizing the **density-based** relationships between patterns in a high-dimensional space, it transforms a previously *intractable problem* into a manageable task, allowing for the discovery of the most dominant and structurally significant regularities in massive datasets.

5.2.10 How to mine Compressed or Approximate Patterns?

Frequent Pattern Mining often produces an overwhelming number of redundant patterns. To address this, **Compressed and Approximate Pattern Mining** aim to extract smaller, more representative pattern sets that summarize the data.

Compressed mining relies on **closed itemsets** for lossless compression and **maximal itemsets** for lossy compression, while **pattern summarization** clusters similar patterns and selects representative ones based on significance. **Approximate mining** handles noise and near-frequent patterns, commonly through **top-k mining**. To avoid redundancy, **redundancy-aware mining** selects patterns that maximize marginal information gain.

The functionality of these advanced methods involves:

1. Applying *similarity thresholds* to identify groups of patterns that are semantically or structurally close;
2. Utilizing **top-k strategies** to limit the output to a user-defined number of results;
3. Incorporating **redundancy measures** to filter out overlapping information;
4. Using *sampling or randomized algorithms* to estimate frequent patterns in extremely large datasets where exact computation is impossible.

By focusing on **representativeness** and *diversity*, these methods transform the "pattern swamp" into a concise and actionable collection of knowledge. These techniques are essential for making data mining results interpretable and useful for **decision-making** in complex environments.

5.2.11 What is the Imbalance Ratio?

The **Imbalance Ratio** (IR) is a measure of interest used in Pattern Mining to evaluate how "balanced" the relationship between two itemsets (A and B) is within an association rule or correlation.

$$\mathrm{IR}(A;B) = \frac{|\sup(A) - \sup(B)|}{\sup(A) + \sup(B) - \sup(A \cup B)}$$

Where:

- $supp(A)$ and $supp(B)$ are the frequencies of the two itemsets;
- $supp(A \cup B)$ is the support of the union of the two itemsets (i.e., the frequency with which they appear together).

The denominator actually represents the support that at least one of the two itemsets is present ($P(A \cup B) = P(A) + P(B) - P(A \cap B)$).

Its main function is to measure the popularity discrepancy between two itemsets that appear related. If it is 0, it means that A and B have equal support (perfectly balanced), while if its value is close to 1, it means that one of the two itemsets is much more frequent than the other (very unbalanced).

In massive databases, the vast majority of transactions contain neither A nor B (null transactions). Measures like *Lift* are influenced by these null transactions, leading to misleading results. The IR measure, however, is null-invariant: its value does not change if we add billions of transactions that do not contain the objects in question. It is used to see if a pattern is frequent just because an itemset is frequent.

5.3 Sequential Pattern Mining

5.3.1 Can you talk about Sequential Patterns?

Sequential Pattern Mining is a specialized field of ***Data Mining and Machine Learning*** focused on discovering **frequent subsequences** within a *sequence database* (database of ordered events with or without concrete notions of time). While standard Association Rule Mining looks for items that occur together in a single transaction, Sequential Pattern Mining identifies patterns that emerge across **multiple transactions** over time. *Even a sequence can be a pattern.*

A **sequence** S is an ordered list of elements, denoted as $\langle e_1 e_2 e_3 \dots e_q \rangle$.

- Each element e_j in the sequence is itself an itemset;
- An item can occur only once within a single itemset, but it can appear multiple times in different itemsets throughout the same sequence;
- A sequence containing k elements is formally referred to as a k-sequence.

To ensure a unique and consistent representation of sequences, the items within each itemset are typically maintained in **lexicographical ordering**, arranged in dictionary-like order. This ordering simplifies sequence comparison and is essential when determining whether one sequence is a subsequence of another.

A sequence $\alpha = \langle a_1 a_2 \dots a_m \rangle$ is said to be a **subsequence** of a sequence $\beta = \langle b_1 b_2 \dots b_n \rangle$ (denoted $\alpha \leq \beta$) if there exists a mapping from the elements of α to those of β that preserves their relative order. Lexicographical ordering within itemsets ensures that such mappings are well-defined and unambiguous.

For example, the sequence $\langle (bd)cb(ac) \rangle$ consists of four elements: the itemsets (bd), (c), (b), and (ac). Under this representation, sequences like $\langle ((bd)cb \rangle$, or $\langle ((bd)(ac) \rangle$, are valid subsequences and are considered sequentials patterns if their frequency in the database satisfy the minimum support threshold.

The discovery of these patterns typically follows an **Apriori-based** Sequential Pattern Mining framework, which decomposes the task into five phases:

1. **Sort Phase**: The original database is sorted primarily by the *Customer ID* and secondarily by the *Transaction Time*, effectively transforming the flat transaction list into a set of distinct customer sequences;
2. **L-itemset Phase**: The algorithm identifies the set of all **frequent l-itemsets** (frequent large items). This is similar to the first step of the Apriori algorithm, where only items meeting the *minimum support threshold min_sup* are retained. Finally, each *l-itemset* is mapped to a *contiguous integer* (ID) to allow for constant-time comparisons during mining;
3. **Transformation Phase:** This step prepares the data for fast sequence counting by replacing each transaction with the IDs of the *l-itemsets*. Transactions without *l-itemsets* are removed, but still counted when computing support;
4. **Sequence Phase**: This is the core iterative process where the algorithm identifies frequent sequences of length 2, 3, and so on. It uses a **candidate generation-and-test** approach, where frequent sequences of length k are joined to form candidates of length $k + 1$, which are then verified against the database;
5. **Maximal Phase:** This final step extracts the **maximal sequences** from the set of all large sequences, S. Starting with the longest discovered patterns (length $k = n$) and moving downward, the algorithm deletes any sequence that is already a subsequence of a longer one.

5.3.2 How to mine Sequences in Web Logs?

Web Usage Mining applies Sequential Pattern Mining to web logs to identify user navigation patterns, modeling user behavior as ordered sequences of individual web pages accessed one at a time.

The mining process generally involves collecting data from one of three locations:

- **Server-side:** Reflects the activity of multiple users and is ideal for web recommender systems, though it may be unreliable due to browser caching;
- **Client-side:** Uses remote agents or modified browsers to collect precise data, eliminating caching and session identification problems;
- **Proxy-side:** Reveals HTTP requests from multiple clients to multiple servers, characterizing the behavior of groups sharing a common server.

Before mining, researchers typically preprocess web server logs (which include IP addresses, timestamps, and requested URLs) to transform them into a transaction database suitable for sequence discovery.

5.3.3 Talk about AprioriAll

The **AprioriAll** Algorithm is a pioneering method for *Sequential Pattern Mining*, designed to discover frequent ordered sequences of itemsets across a transaction database. It extends the logic of the original Apriori algorithm from the domain of static sets to the domain of **time-ordered data**. The algorithm views the problem as finding all sequences whose **Support** (the percentage of total customer sequences that contain the pattern) is greater than or equal to a user-defined threshold. Unlike basic Association Rule Mining, AprioriAll must account for the temporal order.

The primary parameter for AprioriAll are the **minimum support** (*min_sup*), which filters out infrequent sequences. To ensure consistency, items within each element are themselves maintained in a lexicographical order. A sequence S is an ordered list of itemsets $\langle e_1, e_2, \ldots, e_q \rangle$, where each e_j may contain one or more items. For itemsets $t = i_1, \ldots, i_k$ and $t' = j_1, \ldots, j_l$, we define $t < t'$ if they first differ at position h with $i_h < j_h$, or if $k < l$ and t matches the first k items of t'. For example, $(abc) < (abec)$ and $(ab) < (abc)$.

The functionality and steps of the AprioriAll algorithm are defined by the following sequence:

1. **Sort Phase:** The database is sorted by *Customer ID* and then by *Transaction Time* to group all activities belonging to a single individual into a coherent sequence;
2. **L-itemset Phase:** The algorithm finds all **frequent l-itemsets** by scanning the database and counting occurrences, similar to the first pass of the standard Apriori algorithm;
3. **Transformation Phase:** This step prepares the data for fast sequence counting by replacing each transaction with the *l-itemsets* it contains. Transactions without *l-itemsets* are removed, but still counted when computing support. Finally, each *l-itemset* is mapped to a *contiguous integer* to allow for constant-time comparisons during mining;
4. **Sequence Phase:** This phase identifies potential frequent sequences through a join and prune mechanism:
   - **Step 1 (Join):** New candidate sequences are generated by joining two sequences from L_{k-1} that share the same first $(k-2)$ itemsets. The resulting candidate C_k is formed by taking the first $(k-1)$ itemsets from the first sequence and appending the last itemset from the second sequence, and vice versa.
     Example: (k=4)
     $L_3 = \{1\ 2\ 3\}\{2\ 3\ 4\}\{1\ 2\ 4\}\{1\ 3\ 4\}\{1\ 3\ 5\}$
     $C_4 = L_3 \bowtie L_3 = \{1\ 2\ 3\ 4\}\{1\ 2\ 4\ 3\}\{1\ 3\ 4\ 5\}\{1\ 3\ 5\ 4\}$
   - **Step 2 (Prune):** Any candidate sequence in C_k is immediately discarded if any of its subsequences are not present in the set of frequent sequences L_{k-1}. This ensures that only sequences whose components are already known to be frequent are tested against the database.
     Example: $C4 = \{1\ 2\ 3\ 4\}$
5. **Maximal Phase:** After all frequent sequences are found, the algorithm performs a final pass to prune those sequences that are *contained* within a longer frequent sequence, leaving only the **maximal frequent sequences**.

The AprioriAll Algorithm offers several **advantages**:

- It provides a **complete and systematic** way to find all frequent sequences meeting the support threshold;
- It effectively uses the *Apriori property* (downward closure) to prune the search space, ensuring that only potentially frequent sequences are tested;
- The **maximal phase** ensures that the final output is not cluttered with redundant sub-patterns, providing a clearer summary of the data.

However it also presents some **limitations**:

- The **transformation phase** can be extremely expensive in terms of time and storage, as it creates a new version of the entire database;
- It requires **multiple database scans**, once for every level of sequence length, which leads to high I/O overhead on large datasets;
- Like the original Apriori, it can generate a *massive number of candidates*, especially when the frequent sequences are long or the support threshold is low;
- It is often outperformed by more modern **pattern-growth** methods like FreeSpan or PrefixSpan that avoid the heavy transformation and candidate generation steps.

5.3.4 Talk about AprioriSome

The **AprioriSome** Algorithm is an alternative approach to Sequential Pattern Mining that focuses on identifying **maximal frequent sequences** while attempting to minimize the number of candidate sequences evaluated. Unlike the *AprioriAll* algorithm, which systematically finds all frequent sequences of every possible length before filtering for maximality, AprioriSome is designed to skip the counting of many intermediate-length sequences. It is based on the intuition that if we are primarily interested in the longest frequent patterns, we can optimize the search by focusing on longer candidates first and only "filling in the gaps" when necessary.

The primary parameters of AprioriSome include the **minimum support** (min_sup), which sets the threshold for a sequence to be considered frequent, and the **maximum sequence length**, which influences how many levels the algorithm may attempt to skip during candidate generation.

The functionality and steps of the AprioriSome algorithm are structured as follows:

1. **Sort Phase:** Organize the database by *Customer ID* and *Transaction Time* to create a continuous sequence of events for each individual;
2. **L-itemset Phase:** Identify all **frequent l-itemsets** to establish the building blocks for all subsequent sequences;
3. **Transformation Phase:** Map the transactions to the *frequent itemsets* they contain, simplifying the sequence representation;
4. **Forward Phase:** The algorithm finds frequent sequences (L_k) only for specific lengths k determined by a `next(k)` function. If a length is skipped, the algorithm still generates a candidate set C_k using the previous candidate set C_{k-1} (since L_{k-1} it is possible that it is not known) to maintain the chain. This is valid because $L_{k-1} \subseteq C_{k-1}$;
5. **Backward Phase:** Once potential maximal sequences are identified, the algorithm fills in the "*gaps*" by counting the support for the skipped lengths in reverse order, and eliminating those already covered by the **maximal sequence**. The key optimization is that any candidate in C_k already contained in a known frequent super-sequence L_i (where $i > k$) is deleted before counting, as it cannot be a maximal sequence.

The AprioriSome Algorithm offers several **advantages**:

- It can be **more efficient** than AprioriAll in scenarios where there are many long frequent sequences, as it avoids counting many short sub-sequences;
- The algorithm provides a *targeted search* for maximal patterns, which are often the most valuable for decision-making;
- It reduces the **candidate management overhead** by avoiding the exhaustive generation of every frequent level.

However it also presents some **limitations**:

- The **Backward Phase** can introduce redundant work if the initial jumps in the **Forward Phase** were too aggressive or poorly estimated;
- Like all Apriori-based sequential methods, it still relies on the *expensive transformation* of the database;
- It may generate **too many non-maximal candidates** during its forward leaps, leading to memory pressure and increased support-counting time;
- It is generally less flexible than **pattern-growth** algorithms that do not rely on level-wise candidate generation.

AprioriSome is more efficient than AprioriAll because it avoids unnecessary counting: when it finds a frequent long sequence, it skips analyzing its subsequences. This reduces database scans and memory usage by focusing directly on the most informative maximal patterns.

5.3.5 Talk about AprioriDynamicSome

The **AprioriDynamicSome** Algorithm is an advanced optimization of the AprioriSome approach for *Sequential Pattern Mining*. While AprioriSome utilizes fixed steps to "jump" over intermediate sequence lengths, AprioriDynamicSome introduces a **dynamic strategy** to determine which lengths to evaluate during the forward phase. The primary goal is to reach **maximal frequent sequences** as quickly as possible by adapting to the specific distribution of frequent patterns within the dataset. By avoiding the exhaustive level-by-level candidate generation and counting of the standard AprioriAll, this algorithm aims to significantly reduce the computational cost associated with large-scale temporal data.

The main parameters involved include the **minimum support** (min_sup), which serves as the fundamental frequency threshold, and a **step size controller** that manages how many levels the algorithm skips based on the number of frequent sequences discovered in the previous steps.

The functionality and steps of the AprioriDynamicSome algorithm are defined by the following process:

1. **Sort Phase:** Organize the database by *Customer ID* and *Transaction Time* to create a continuous sequence of events for each individual;
2. **L-itemset Phase:** Identify all **frequent l-itemsets** to establish the building blocks for all subsequent sequences;
3. **Transformation Phase:** Map the transactions to the *frequent itemsets* they contain, simplifying the sequence representation;
4. **Initialization Phase:** Count all candidates up to the length of the step (if $step = 3$, then count L_1, L_2, L_3 as if it were AprioriAll);
5. **Forward Dynamic Phase:** Instead of counting every length, the algorithm only counts sequences whose lengths are multiples of a defined *step* (e.g., if $step = 3$, it finds $L_3, L_6, L_9, \dots$). To find L_{k+step}, it uses an *otf*[1]*-generate* procedure that joins L_k with L_{step} to find C_{k+step} during a database scan, significantly reducing the number of candidates compared to standard Apriori generation;
6. **Intermediate Phase:** Because skipped lengths were never even generated in the forward pass, an *Intermediate Phase* first creates candidate sets C_k for all skipped k using the nearest known L_{k-1} or C_{k-1};
7. **Backward Dynamic Phase:** Then, similar to *AprioriSome*, the *Backward Phase* counts these candidates in reverse order, immediately pruning any that are subsequences of already identified **maximal sequences**.

The AprioriDynamicSome Algorithm offers several **advantages**:

- It is **highly efficient** for datasets containing very long frequent sequences because it can "jump" directly to the most significant levels;
- The **adaptive nature** of the algorithm makes it more flexible than the static AprioriSome, as it tunes its search behavior to the underlying data characteristics;
- It reduces the *total number of candidates* that must be stored and counted in memory compared to exhaustive breadth-first searches;
- It focuses on **maximal patterns**, which are typically the most useful for high-level business intelligence and sequence analysis.

However it also presents some **limitations**:

- The **complexity of implementation** is significantly higher than that of AprioriAll due to the dynamic logic and the coordination between forward and backward phases;
- Like other Apriori-based methods, it still suffers from **multiple database scans** and the overhead of the *transformation phase*;
- In datasets with mostly *short patterns*, the overhead of the dynamic jumping logic and the backward verification can make it slower than simple level-wise algorithms;
- It remains **memory-intensive** when dealing with a high number of candidate sequences generated during large leaps.

[1] on-the-fly

5.3.6 Talk about FreeSpan

The **FreeSpan** (**Frequent Pattern-projected Sequential Pattern Mining**) Algorithm is a breakthrough approach in the field of sequence analysis that moves away from the traditional *candidate-generation-and-test* paradigm. Instead of building and checking millions of candidate sequences like Apriori-based methods, FreeSpan utilizes a **pattern-growth** strategy. It is designed to reduce the search space by partitioning the database into smaller, manageable fragments based on the frequent patterns discovered so far. This *divide-and-conquer* methodology allows the algorithm to focus only on the relevant parts of the database, significantly improving performance on large and dense datasets.

The primary parameters for FreeSpan include the **Minimum Support** (min_sup), which filters out infrequent items and sequences, and the **frequent item list** (f_list), which contains all individual items that satisfy the support threshold sorted in descending order of frequency.

The core of FreeSpan lies in how it recursively partitions the database. The process is governed by the **f-list** (Frequent Item List), which ranks items by descending frequency (for example: $\{a:5, b:4, c:4, d:3, e:3, f:2\}$). The database S is partitioned into n subsets. All sequential patterns containing item i (we start from the last) and no item following i (infrequents for the last) in f_list are found in the i-projected database. A sequence s is projected to s_i if it contains i, where s_i is obtained by removing all infrequent items and any frequent item j that follows i in f_list.

For example: if the f_list is $\{a, b, c\}$, the c-projected database ($S|_c$) contains all sequences where c occurs. Patterns are then grown within $S|_c$ using only items $\{a, b, c\}$.

The functionality and steps of the FreeSpan Algorithm are as follows:

1. **Scan and Sort:** Perform an initial scan of the database to identify all frequent items and sort them in descending order of frequency to form the f_list. We start from the last;
2. **Database Partitioning:** Use the frequent items to partition the sequence database into several **projected databases**. Each projected database contains only the sequences that include a specific frequent item;
3. **Recursive Mining:** For each projected database, identify local frequent items and grow the patterns by appending these items to the current **prefix**;
4. **Sequence Refinement:** Refine the sequences within the projected databases by removing items that are no longer frequent in that specific local context;
5. **Termination:** Continue the recursive projection and mining process until no more frequent items can be found in the projected databases.

The FreeSpan Algorithm offers several **advantages**:

- It completely **eliminates candidate generation**, which is the most significant bottleneck in level-wise sequential mining;
- The use of **projected databases** ensures that the algorithm only processes a small fraction of the data in each recursive step;
- It is **substantially faster** and more scalable than AprioriAll, AprioriSome and AprioriDynamicSome, especially when the database contains many long frequent sequences;
- The algorithm reduces *I/O overhead* because the projected databases rapidly shrink in size as the prefix length increases.

However it also presents some **limitations**:

- The process of **generating projected databases** can be memory-intensive if not managed carefully, as multiple versions of the data may exist simultaneously;
- It still involves a significant amount of *data movement* and duplication when creating the physical projections;
- While more efficient than Apriori, it was eventually surpassed by **PrefixSpan**, which uses a more optimized pseudo-projection technique to further reduce memory consumption;
- The implementation is considerably *more complex* than candidate-based algorithms due to the recursive partitioning logic.

While FreeSpan is efficient, its performance is heavily influenced by how it manages projected databases. Two main approaches are used:

- **Parallel Projection:** A sequence is projected simultaneously into the projected databases of all its frequent items. For instance, a sequence containing $\{a, b, c\}$ is duplicated into the a-, b-, and c-projected databases. Its main drawback is the **high memory and disk overhead**, since sequences may be replicated many times, greatly increasing the total projection size.
- **Partition Projection:** As an optimization over parallel projection, each sequence is projected only into the database of its **last** frequent item in f_list order. After mining that item, the sequence is moved to the next relevant projection. This on-the-fly reassignment ensures each sequence appears in only **one** projected database at a time, greatly reducing projection size and improving scalability.

5.3.7 What are the differences between Count-All and Count-Some?

The primary distinction between **Count-All** and **Count-Some** lies in their reckoning strategy and how they navigate the search space for frequent sequences.

AprioriAll belongs to the "*Count-All*" family, meaning it systematically counts all frequent sequences, including those that are non-maximal. It operates in a level-wise manner similar to the standard Apriori algorithm, using frequent $(k-1)$-sequences to generate candidate k-sequences. Because it discovers every frequent subsequence, it requires a separate *Maximal Phase* at the end of execution to filter out redundant patterns and identify the final set of maximal sequences.

In contrast, **AprioriSome** and **AprioriDynamicSome** are "*Count-Some*" algorithms designed to avoid the computational cost of counting non-maximal sequences by jumping through sequence lengths. **AprioriSome** utilizes a *Forward Phase* to find frequent sequences of specific lengths (e.g., lengths 1, 2, 4, and 6, or every two) and a *Backward Phase* to fill the gaps. During its Backward Phase, it significantly optimizes performance by deleting any candidate sequence that is already a subsequence of a known frequent sequence discovered in the Forward Phase, thus ensuring that non-maximal patterns are never counted.

AprioriDynamicSome further refines this jumping strategy by introducing a *step* variable to determine which lengths are counted in the Forward Phase. While AprioriSome generates all intermediate candidate sets (C_k) during its Forward Phase, AprioriDynamicSome skips this generation for intermediate lengths entirely. Instead, it uses a specialized *otf-generate* (on-the-fly) procedure to jump from L_k to L_{k+step} by joining patterns within the context of a specific customer sequence during the database scan. This "on-the-fly" approach can lead to a much smaller candidate set compared to the standard join-step used in AprioriAll and AprioriSome. However, while AprioriSome typically performs better than AprioriAll because it avoids non-maximal counts, AprioriDynamicSome can occasionally generate an excessive number of candidates depending on the dataset characteristics.

5.3.8 Why is *otf-generate* needed in AprioriDynamicSome?

The *on-the-fly-generate* procedure is essential for the efficiency of **AprioriDynamicSome** due to the following reasons:

- **Candidate Space Optimization:** The standard *apriori-generate* procedure used in AprioriSome often generates a much larger set of candidate sequences (C_{k+step}) than necessary. By generating candidates only from patterns that actually appear together in the same customer sequence, *otf-generate* maintains a more manageable search space;
- **Computational Efficiency:** It is often faster to find all frequent members of L_k and L_{step} within a specific customer sequence c than to check that sequence against a massive, globally-generated candidate set C_{k+step}, particularly when the sum of the sizes $|L_k| + |L_{step}|$ is smaller than the size of the traditional candidate set;
- **Join Condition Simplification:** Traditional Apriori joining requires itemsets to match on their first $(k-2)$ terms to increment length by one. Since AprioriDynamicSome jumps by a larger *step*, *otf-generate* provides a generalized way to join a k-sequence and a j-sequence without needing complex prefix-matching;
- **Ensuring Relative Order:** The procedure uses the intuition that if a k-sequence and a j-sequence appear in a record without overlapping, they form a valid candidate $(k+j)$-sequence. It specifically checks that the end position of the first sequence ($X_k.end$) is strictly less than the start position of the second ($X_j.start$) to ensure the temporal order is preserved.

Chapter 6: Outlier

6.1 Outlier Detection and Outlier Analysis

6.1.1 What is an Outlier? What are the types of Outliers?

An **Outlier** is a data object that deviates significantly from the normal objects in a dataset, as if it had been generated by a different mechanism. **Outliers are not mere random measurement Noise**. Noise is random error or variance in a measured variable and is normally removed or reduced before outlier analysis. By contrast, outliers are potentially interesting because they indicate events or behaviors that violate the mechanism that generates the normal data. Typical application domains where outliers matter include fraud detection (credit card or telecom fraud), intrusion detection in networks, fault detection in industry, medical anomaly discovery, and discovery of exceptional performers in sports or finance.

Modeling normal and abnormal behavior accurately is difficult because normal behavior can vary across applications. The boundary between normal and outlier objects is often not sharp. The presence of noise may blur this distinction and hide true outliers. Effective outlier detection requires methods that are domain-specific, explainable, and able to quantify how unlikely a data object is according to the normal data generation process.

There are three principal types of outliers: *global outliers*, *contextual outliers*, and *collective outliers*.

- **Global Outliers (point anomalies):** Objects that significantly deviate from the entire dataset. They can be detected using deviation measures such as distance or probability scores. Common in intrusion or fault detection tasks.
- **Contextual Outliers (conditional anomalies):** Objects that are anomalous only within a specific context. To evaluate such outliers, the attributes of each object are divided into two sets:
  - *Contextual attributes:* which define the context (time, location);
  - *Behavioral attributes:* which describe the object's measurable behavior.

  Example: 35°C in Pisa is normal in summer but abnormal in winter. The main challenge is defining the proper context.
- **Collective Outliers:** Groups of objects that deviate collectively, though individual members may appear normal. Examples include coordinated network attacks or abnormal event sequences. Detection requires modeling relationships (distance, temporal, or structural links).

6.1.2 Make an overview of Outlier Detection

Outlier detection methods can be categorized according to available information and model assumptions.

From the information viewpoint, methods can be:

- **Supervised:** when labeled examples of both normal and outlier objects are available;
- **Semi-supervised:** when only normal examples (or only few labeled outliers) are known;
- **Unsupervised:** when no labels are given and structure is inferred directly from data.

From the modeling viewpoint, methods can be:

- **Statistical approaches:** assume a probabilistic model for normal data and flag points in low-probability regions;
- **Proximity-based approaches:** detect objects far from their nearest neighbors;
- **Clustering-based approaches:** consider points not belonging to large dense clusters as outliers;
- **Classification approaches:** learn a boundary for normal data and treat points outside it as anomalies.

6.2 Statistical (Model-Based) Methods

Statistical approaches, or **model-based methods**, assume that data are generated by a stochastic process following a probability distribution. The goal is to learn a generative model that fits the data and to identify as **outliers** those objects lying in low-probability regions. These methods estimate the parameters of the normal-data distribution and measure how unlikely each observation is.

The **effectiveness** of statistical approaches depends on how well the assumed model matches the true data distribution. When the model fits well, these methods provide interpretable and principled detection, otherwise they may yield misleading results. Two main categories exist: **parametric** and **non-parametric** methods.

Their main **advantages** are a clear probabilistic interpretation of outliers, computational efficiency for simple models (such as Gaussian ones), and the ability to quantify Outlierness through probabilities. However, they also have notable **disadvantages**: strong assumptions about data distribution, sensitivity to noise and model deviations, and limited applicability to complex, high-dimensional, or multimodal data.

6.2.1 What are the Parametric Approaches to Outlier Detection?

Parametric approaches assume that the normal data are generated from a specific parametric probability distribution defined by a set of parameters θ. The probability density function $f(x;\theta)$ gives the likelihood that an object x was generated by the model. The smaller this probability, the more likely the object is an outlier. The parameters of the model are estimated from the data, usually via **Maximum Likelihood Estimation (MLE)**.

Parametric methods can handle both univariate and multivariate data.

Univariate Outlier Detection Based on the Normal Distribution

When the data consist of a single attribute, it is often assumed that they follow a normal (Gaussian) distribution. The parameters μ (mean) and σ (standard deviation) are estimated from the data, and objects with very low probability under this distribution are considered outliers.

To verify if the normality assumption holds, tests such as the **Shapiro–Wilk Test** or visual tools such as the **Q–Q plot** (Quantile–Quantile plot) can be used. In a Q–Q plot, if the data points approximately follow a straight line when plotted against the quantiles of a theoretical normal distribution, the assumption of normality is reasonable.

To find the optimal parameters $\hat{\mu}$ and $\hat{\sigma}^2$ that maximize likelihood, we simplify the density function using the natural logarithm:

$$\ln L(\mu, \sigma^2) = -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln\sigma^2 - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2$$

Setting the partial derivatives with respect to μ and σ^2 to zero yields the MLE:

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})^2$$

Data points with low probability, typically those outside the $\hat{\mu} \pm 3\hat{\sigma}$ range, are classified as outliers.

The Grubbs Test

The **Grubbs Test** (also called the *Maximum Normed Residual Test*) is another classical *univariate* parametric method assuming normality. It identifies one outlier at a time by evaluating how far the most extreme value is from the mean relative to the standard deviation.

The procedure is:

1. The data is sorted from smallest to largest. Compute the test statistic:

$$G = \frac{\max |x_i - \bar{x}|}{s}$$

   where $\bar{x}$ is the sample mean and s the sample standard deviation;

2. Determine the critical value G_{crit} from Grubbs' distribution table using the desired significance level α;

3. If $G > G_{crit}$, the corresponding observation is rejected as an outlier.

This test can be applied iteratively, removing one outlier at a time, but assumes the data follow a normal distribution. The test is looking only at the point that is furthest from the mean (the most extreme value, look at the numerator of the formula).

Multivariate Outlier Detection

For multivariate data (two or more variables), univariate methods are insufficient because they ignore correlations among variables. Parametric multivariate methods address this by transforming the multivariate problem into a univariate one through a distance measure.

Mahalanobis Distance

The **Mahalanobis distance** measures the distance of an observation $\mathbf{o}$ from the mean vector $\bar{\mathbf{o}}$ considering the covariance among variables:

$$\text{MD}^2(\mathbf{o}; \bar{\mathbf{o}}) = (\mathbf{o} - \bar{\mathbf{o}})^T S^{-1} (\mathbf{o} - \bar{\mathbf{o}})$$

where S is the covariance matrix. This metric standardizes the scale of variables and accounts for correlations. Euclidean distance can be misleading because it is scale-sensitive and ignores correlations, while Mahalanobis distance accounts for both by standardizing and decorrelating the data. A large Mahalanobis distance indicates that the observation lies far from the multivariate mean, suggesting an outlier. However, computing and inverting the covariance matrix can be computationally expensive, especially for high-dimensional data. Use the Grubb's test on these distances to detect outliers.

Chi-Square Statistic

Assuming that the data follow a multivariate normal distribution, the squared Mahalanobis distance follows approximately a χ^2 distribution with p degrees of freedom (where p is the number of variables):

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

If the computed χ^2 value for an observation exceeds a threshold (e.g., mean $+3\sigma$ of χ^2), the observation is flagged as an outlier. This approach works well when the number of variables is large enough (typically > 30) so that the Central Limit Theorem ensures approximate normality.

6.2.2 What are the Non-Parametric Approaches to Outlier Detection?

Non-parametric approaches do not assume any specific statistical distribution for the data. Instead, they infer the underlying structure directly from the data itself. These methods are more flexible and can handle data that deviate from known distributions, but they can be computationally more expensive.

Histogram-Based Detection

In histogram-based methods, the data space is divided into bins, and the frequency of observations in each bin is computed. The probability of observing a given value is proportional to its bin height. An object is considered an outlier if it falls into a bin with very low frequency.

For example, in a histogram of transaction amounts, if only 0.2% of transactions exceed \$5,000, a transaction of \$7,500 can be identified as an outlier. The main drawback of this method lies in the choice of bin width:

- Too small bins may isolate normal objects into rare bins, leading to false positives;
- Too large bins may group true outliers with normal data, leading to false negatives.

Kernel Density Estimation (KDE)

Kernel Density Estimation overcomes the bin-size problem by estimating a smooth probability density function:

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

where $K(\cdot)$ is a kernel function (typically Gaussian), h is the bandwidth (smoothing parameter), and n is the number of data points. The bandwidth controls the smoothness of the density estimate: smaller h produces more detail but more noise. Larger h yields smoother estimates. Points located in regions of very low estimated density are identified as outliers.

Computational Considerations

The computational cost of statistical methods varies with model complexity:

- Simple parametric models, such as a single Gaussian, are linear in time;
- Mixture models, estimated via Expectation–Maximization (EM), require multiple iterations but each iteration is linear in dataset size;
- Non-parametric KDE can be quadratic in time with respect to the number of data points.

Once the model is learned, outlier detection itself is typically fast, since it requires only evaluating the density or probability for each object.

6.3 Proximity-Based Methods

Proximity-based approaches identify outliers by analyzing the **distances** or **proximities** among data objects. The central idea is that **normal objects** typically lie close to their neighbors, forming dense regions, while **outliers** are located far away or in sparse regions of the data space. These methods assume that an object whose proximity pattern deviates significantly from that of the majority is anomalous.

Two main categories exist:

- **Distance-based methods:** An object is an outlier if it has too few neighboring points within a specified radius.
- **Density-based methods:** Outliers are objects whose local density is significantly lower than that of their neighbors.

The effectiveness of proximity-based methods depends on the choice of the distance or similarity measure, which must be meaningful for the domain and data type. A common challenge arises when different regions of the dataset have varying densities, or when the notion of proximity is distorted by irrelevant or correlated features. Despite these challenges, proximity-based methods remain widely used due to their **intuitive interpretation**, **unsupervised nature**, and ability to handle datasets without predefined probabilistic models.

6.3.1 How does Distance-Based Outlier Detection work?

Global Distance-Based Outlier Detection

Distance-based outlier detection defines outliers in terms of the **distance** between a data object and the other objects in the dataset. The main intuition is that normal objects occur in dense neighborhoods, whereas outliers are separated from others by large distances. Thus, an object is considered an outlier if it has too few neighbors within a specified distance threshold.

Formally, for a dataset D containing n objects, an object o is a **global distance-based outlier** if at least a fraction π of the objects in D lie farther than a given distance r from o. Equivalently, one can check the distance between o and its k-th nearest neighbor o_k: if $\text{dist}(o; o_k) > r$, then o is an outlier. These definitions depend on two parameters, the distance threshold r and the fraction (or neighbor count) that defines what constitutes a dense region.

An object o is a *global distance-based outlier* if:

$$\frac{|\{o' \mid dist(o; o') \leq r\}|}{|D|} \leq \pi$$

with o' that represents a generic object belonging to the dataset D. The formula uses o' to count how many objects in the entire dataset are located within the specified distance threshold r of the object o.

π is the **Fraction Threshold**. This is a value between 0 and 1 that represents the maximum allowable fraction of the total dataset D that can exist within the r-neighborhood of an outlier. If the fraction of neighbors is less than or equal to π, it means object o is too isolated to be considered "normal".

The **intuition** behind this approach is simple: in regions of high data density, most points have many close neighbors, in regions of low density, outliers have only a few nearby points. The challenge lies in choosing suitable parameter values for r and π that reflect the true structure of the data. This is a **global distance metric**.

The standard distance-based detection algorithm proceeds as follows:

1. For each object o_i in the dataset, compute the distance between o_i and all other objects in D;
2. Count the number of objects that fall within the distance r from o_i;
3. If the number of such neighbors is less than $\pi \cdot |D|$, declare o_i as a distance-based outlier.

Although this naive approach requires comparing each object to all others, making it $O(n^2)$ in the worst case, in practice the computation is faster because most objects quickly meet the neighbor threshold and can terminate early.

The primary flaw in the distance-based approach is its reliance on a global threshold (r and π). Because it uses a single fixed distance to define the neighborhood, it cannot account for datasets where density varies between clusters.

In dense clusters, an outlier might be physically close to the cluster but have a significantly lower density than its members. The global distance r might still find enough neighbors to label it "normal".

In sparse clusters, perfectly normal points might be flagged as outliers because their neighbors are naturally further apart than the global r threshold allows.

Grid-Based Method

When datasets are too large to fit entirely in memory, grid-based strategies can be applied to reduce computation. The data space is divided into a multi-dimensional grid of cells, where each cell is a hypercube with diagonal length $r/2$. Only neighboring cells need to be compared, allowing efficient pruning.

Two main pruning rules are used:

- **Level-1 cell rule:** If the number of objects in a cell and its immediate neighboring cells (within distance r) exceeds $\pi|D|$, then all objects in that cell are normal, since they have enough neighbors nearby.
- **Level-2 cell rule:** If the number of objects in a cell and in the cells up to two levels away (beyond distance r) is smaller than $\pi|D| + 1$, then all objects in that cell are outliers, because even their extended neighborhood is too sparse.

This approach greatly reduces the number of pairwise distance computations by checking only the cells that cannot be pruned by these rules.

Local Distance-Based Outlier Detection (Outlierness)

A limitation of global distance-based detection is that it may fail in datasets with varying density. In such cases, an object might appear far from some clusters but still have similar density to nearby points. To handle this, **local distance-based methods** are introduced, which evaluate each object relative to its local neighborhood.

For an object x_i, let $N_k(x_i)$ be its k nearest neighbors and $D_k(x_i)$ the average distance to them:

$$D_k(x_i) = \frac{1}{k} \cdot \sum_{x_j \in N_k(x_i)} \|x_i - x_j\|$$

The **Outlierness** of x_i is then defined as the ratio between $D_k(x_i)$ and the average D_k of its neighbors:

$$O_k(x_i) = \frac{D_k(x_i)}{\frac{1}{k}\sum_{x_j \in N_k(x_i)} D_k(x_j)}$$

If $O_k(x_i) > 1$, the object is farther from its neighbors than expected, and thus locally anomalous. This relative formulation adapts to regions of different density, although it can produce unintuitive results when clusters are close or overlapping. This is a **local distance metric**.

This formula works because it is relative (local) rather than absolute (global). If an object x_i belongs to a cluster (whether sparse or dense) its average distance to its k nearest neighbors ($D_k(x_i)$) will be comparable to the average distances of those neighbors to their own k nearest neighbors. Consequently, this ratio will be close to 1.

If x_i is an outlier, it is significantly further from its neighbors than those neighbors are from each other. This makes $D_k(x_i)$ much larger than the average D_k of its neighbors, resulting in an Outlierness score > 1.

They offer several **advantages**:

- Conceptually simple and intuitive;
- Requires no assumptions about the data distribution;
- Works well for unsupervised detection of global anomalies.

However, they also present some **limitations**:

- Sensitive to the choice of parameters r and π (or k);
- Computationally expensive for large datasets without optimization;
- Performance degrades when data have non-uniform density.

While this approach handles varying densities, it can produce unintuitive results when clusters are physically close to each other.

For **example**, in a scenario with two close clusters (C_1 and C_2), a point p located on the edge of C_1 (facing C_2) might be assigned a higher Outlierness score than a point q that is clearly further away from any cluster.

The **reason** is that for small values of k, the k nearest neighbors of point p might only include points from the dense cluster C_1, while the nearby points in cluster C_2 are ignored. This makes p appear "isolated" relative to the high density of its C_1 neighbors, even though it is physically surrounded by other points. The biggest practical limitation is that Outlierness doesn't tell you how isolated a point is in absolute terms, but only how isolated it is relative to its neighbors.

6.3.2 Talk about the Density-Based Outlier Detection and more in particular LOF

Density-Based Outlier Detection

Density-based outlier detection refines the idea of distance-based methods by comparing the **local density** of each object to the density of its neighbors. The key intuition is that an object is considered an outlier if it resides in a region significantly less dense than that of its surrounding neighborhood. This approach overcomes the limitations of global distance-based detection, which may fail when datasets contain regions of varying density.

Unlike distance-based approaches, which use a global threshold on distances, density-based methods define the degree of anomaly relative to the local neighborhood, making them more adaptive to non-uniform data distributions.

The **local reachability density** quantifies how densely an object is surrounded by its neighbors. For each object p, let $N_k(p)$ be the set of its k nearest neighbors. The *reachability distance* between p and a neighbor o is defined as:

$$\text{reach-dist}_k(p; o) = \max\{k\text{-distance}(p); \text{dist}(p; o)\}$$

where k-distance(p) is the distance from p to its k-th nearest neighbor. This ensures that very close points do not dominate the measure by enforcing a lower bound on distances. *This is the same that we see in OPTICS algorithm.*

The **local reachability density** of p is then computed as the inverse of the average reachability distance from p to its k neighbors:

$$\text{lrd}_k(p) = \frac{|N_k(p)|}{\sum_{o \in N_k(p)} \text{reach-dist}_k(p; o)}$$

A high $\text{lrd}_k(p)$ indicates that p lies in a dense neighborhood, a low value indicates that p is relatively isolated.

LOF

The **Local Outlier Factor (LOF)** measures how much the density around a point differs from the density around its neighbors. It is defined as the average ratio of the local reachability densities of the neighbors of p to the local reachability density of p itself:

$$\text{LOF}_k(p) = \frac{1}{|N_k(p)|} \sum_{o \in N_k(p)} \frac{\text{lrd}_k(o)}{\text{lrd}_k(p)} = \sum_{o \in N_k(p)} \text{lrd}_k(o) \cdot \sum_{o \in N_k(p)} \text{reach-dist}_k(p; o)$$

A value of $\text{LOF}_k(p) \approx 1$ indicates that the point has a density similar to its neighbors and is therefore normal. If $\text{LOF}_k(p) > 1$, the point has a lower density than its neighbors and is considered an outlier. The larger the LOF value, the stronger the Outlierness.

Interpretation of LOF Values:

- LOF ≈ 1: object is in a region of similar density to its neighbors (normal);
- LOF > 1: object is in a sparser region than its neighbors (potential outlier);
- LOF $\gg 1$: object is in a much less dense region (strong outlier).

This are the algorithmic steps for LOF Computation:

1. For each object p, determine the set of its k nearest neighbors $N_k(p)$;
2. Compute the k-distance for every object and the reachability distance for each pair $(p; o)$;
3. Compute the local reachability density $\text{lrd}_k(p)$ for every object;
4. Compute $\text{LOF}_k(p)$ by comparing $\text{lrd}_k(p)$ with the densities of its neighbors;
5. Rank objects by their LOF values to identify outliers.

The reachability distance imposes a minimum value (the distance to the k-th neighbor) in the calculation. This "smooths" statistical fluctuations and makes the algorithm less sensitive to small groups of nearby anomalies. Imagine a point x_i at the edge of a dense cluster. Its distance from its neighbors (D_k) could be large, causing O_k to skyrocket and flagging a false outlier. LOF is more sophisticated: it understands that if x_i's neighbors also have slightly different densities, the relationship balances out, avoiding flagging points that are simply "on the edges" of a cluster as outliers. LOF analyzes the "rhythm" of local density, making it much more accurate in identifying who is truly out of context.

If a cluster has locally varying density, the LOF method adjusts to the relative scale of that region. For instance, points in a sparse cluster are not flagged as outliers if they have a density consistent with other points in the same cluster. Conversely, a single point lying between two clusters or in a low-density boundary area would obtain a high LOF score and be considered an outlier. So, even if it is a better method than distance-based ones, **it has the same flaw as Local Distance-Based Outlier Detection**.

LOF offers several **advantages**:

- Adapts to local variations in data density;
- Does not require global distance or density thresholds;
- Suitable for complex and non-uniform datasets.

However, they also present some **limitations**:

- Sensitive to the choice of the parameter k (neighborhood size);
- Computationally intensive for large datasets due to neighbor searches;
- LOF values can be difficult to interpret in very high-dimensional data.

6.4 Clustering-Based Methods

Clustering-based approaches rely on the assumption that normal data objects form large and dense clusters, while outliers do not belong to any cluster or belong to small, sparse clusters. Instead of explicitly modeling the distribution of normal data, these methods infer the structure of the dataset through Clustering and then measure how well each object fits within that structure.

After the data are partitioned into clusters, an object is considered an outlier if it satisfies one of the following conditions: it is not assigned to any cluster, it is located far from the centroid of the nearest cluster, or it belongs to a very small or low-density cluster.

These methods are flexible and intuitive because they naturally capture data structure. However, their performance depends on the Clustering algorithm used (e.g., K-Means, DBSCAN, Hierarchical Clustering) and the choice of parameters such as the number of clusters or density thresholds.

They offer several **advantages**:

- Do not require labeled data;
- Capture both global and local data structure;
- Provide a natural interpretation of normality as cluster membership.

However, they also present some **limitations**:

- Sensitive to Clustering parameters and distance metrics;
- Ineffective if clusters overlap or have irregular shapes;
- Computationally expensive for large or high-dimensional data.

6.4.1 Talk about CBLOF

The **CBLOF** (**Cluster-Based Local Outlier Factor**) algorithm is a Clustering-based approach to outlier detection that quantifies the degree of Outlierness of each object according to the cluster structure of the data. It assumes that **normal objects belong to large and dense clusters**, while **outliers either belong to small, sparse clusters or are far from large clusters**. The CBLOF approach extends the general idea of Clustering-based detection by combining information about both the *size* of the cluster to which a data point belongs and its *distance* to other clusters. It therefore captures both global and local deviations in the dataset.

We seek an index b (the boundary) that splits the clusters into $C_{\text{large}} = C_1, \ldots, C_b$ and $C_{\text{small}} = C_{b+1}, \ldots, C_k$.

$$\begin{cases} (|C_1| + |C_2| + \cdots + |C_b|) \geq |D| \cdot \alpha \\ \frac{|C_b|}{|C_{b+1}|} \geq \beta \end{cases}$$

The first b clusters must contain at least an α fraction of the dataset D. A boundary is also triggered when a cluster is at least β times larger than the next. We use $\alpha = 0.9$ and $\beta = 5$ by default. $C_{large} = \{C_i | i \leq b\}$ and $C_{small} = \{C_j | j > b\}$.

The CBLOF algorithm proceeds as follows:

1. Apply a Clustering algorithm (such as K-Means or any partition-based method) to divide the dataset into k clusters;
2. Rank the clusters by their size (number of objects). Let C_{large} be the set of **large clusters** and C_{small} be the set of **small clusters**;
3. For each data object p in a cluster C_i, compute its **Cluster-Based Local Outlier Factor** as:

$$CBLOF(p) = \begin{cases} |C_i| \cdot \text{dist}(p; \text{Centroid}(C_i)) & \text{if } C_i \in C_{large} \\ |C_i| \cdot \text{dist}(p; \text{Centroid}(C_{nearest\_large})) & \text{if } C_i \in C_{small} \end{cases}$$

   where $|C_i|$ is the size of the cluster to which p belongs, $\text{Centroid}(C_i)$ is the centroid of that cluster, and $\text{dist}(\cdot)$ is a distance function such as Euclidean distance;
4. Rank all data points by their $CBLOF(p)$ values in descending order. The higher the CBLOF value, the stronger the Outlierness.

The CBLOF value reflects both cluster membership and distance. Objects in large clusters are expected to be normal and thus have small CBLOF values, while objects in small clusters or those lying far from any large cluster will have high CBLOF values, identifying them as potential outliers.

For points in large clusters, the Outlierness depends on their deviation from the cluster center. For points in small clusters, the distance to the nearest large cluster dominates, since small clusters are considered likely to contain anomalies. The use of cluster size as a weight ensures that isolated points or small groups of unusual observations receive proportionally higher scores.

A normalized variant, called **Local CBLOF (LCBLOF)**, uses normalized cluster sizes to avoid overemphasizing differences between extremely large and small clusters:

$$LCBLOF(p) = \frac{|C_i|}{|C_{max}|} \cdot \text{dist}(p; \text{Centroid}(C_{ref}))$$

where C_{ref} is the relevant comparison cluster (the same or the nearest large cluster). This modification reduces the bias caused by disproportionate cluster sizes and stabilizes the scoring scale.

It offers several **advantages**:

- Combines cluster structure and distance information into a single score;
- Works in unsupervised settings with no labeled data;
- Adapts well to data containing both dense and sparse regions.

However, it also presents some **limitations**:

- Requires an effective Clustering step. Performance depends heavily on the chosen Clustering algorithm and parameters;
- Assumes that large clusters always represent normal data, which may not hold in some domains;
- Sensitive to the scale of cluster sizes and distance metrics used.

6.4.2 Talk about DBSCAN for Outlier Detection

The **Density-Based Spatial Clustering of Applications with Noise** (**DBSCAN**) algorithm is a density-based Clustering method that naturally identifies outliers as those data points that do not belong to any cluster. It assumes that **normal data lie in dense regions** (clusters), while **outliers reside in low-density zones** and hence are labelled as "noise" or anomalies.

DBSCAN works as follows:

1. Specify two key parameters: ε (eps), the maximum distance within which neighbouring points are considered part of the same neighbourhood, and MinPts, the minimum number of points required to form a dense region;
2. For each unvisited point p:
   - Retrieve all points within distance ε of p. If the number of such points is $\geq$ MinPts, then p is a "core point" and a new cluster is begun (or expanded);
   - Else if p is within ε of a core point but has fewer than MinPts neighbours, then p is a "border point";
   - Otherwise p is labelled as a "noise point", an outlier.
3. Expand each cluster by recursively including all points density-reachable from core points, that is, reachable via a chain of core points each within ε;
4. After the Clustering completes, all points labelled as noise are interpreted as potential outliers, they lie in regions of insufficient density to be grouped into any cluster.

The DBSCAN approach therefore flags outliers by virtue of their isolation from dense neighbourhoods: points that are neither core nor border are singled out.

6.5 Classification-Based Methods

Classification-based approaches treat outlier detection as a problem of learning a decision boundary that separates normal data from anomalies. These methods use a model trained from labeled or partially labeled data to distinguish between the *normal class* and the *outlier class*. When only normal examples are available, the task becomes a one-class Classification problem in which the algorithm learns the region of normality and flags any observation outside this region as an outlier.

The central idea is that a classifier implicitly defines a boundary in the feature space that encloses the region containing most of the data. Points lying far from this boundary are considered anomalies. Common techniques include **One-Class Support Vector Machines (OCSVM)** and **Support Vector Data Description (SVDD)**, which construct minimal hyperspheres or separating surfaces around the normal data. Other approaches use nearest-neighbor or ensemble-based models trained to discriminate between synthetic noise samples and the observed data.

Classification-based methods are particularly effective when labeled data are available, as they provide explicit decision boundaries and interpretable results. However, they depend strongly on the quality and representativeness of the training set.

They offer several **advantages**:

- High accuracy when labeled training data are reliable;
- Provide explicit, interpretable decision boundaries;
- Efficient and scalable with modern machine learning models.

However, they also present some **limitations**:

- Require labeled data, which are often scarce or imbalanced;
- Performance degrades if training data contain undetected outliers;
- One-class models may fail in highly multimodal distributions.

6.6 Mining Contextual and Collective Outliers

In many real-world applications, anomalies cannot be accurately identified by examining individual data points in isolation.

A **contextual outlier** (or *conditional outlier*) is an object that is anomalous only within a specific context. Its abnormality depends on contextual attributes (such as time, location, or conditions) relative to behavioral attributes that describe its measurable behavior. For example, a temperature of 35°C may be normal in summer but anomalous in winter.

A **collective outlier**, in contrast, refers to a group of objects that collectively exhibit anomalous behavior, even if individual members appear normal in isolation. Examples include coordinated cyberattacks or unusual temporal patterns in sensor readings. Detecting such outliers requires modeling dependencies such as temporal order, spatial correlation, or structural links between data points.

Mining contextual and collective outliers is thus essential for complex domains where data exhibit strong dependencies and contextual variability, enabling more accurate and meaningful anomaly detection beyond simple point-based analysis.

6.6.1 How to mine Contextual or Collective Outliers?

Contextual Outliers

The mining of contextual outliers generally proceeds in two steps:

1. **Context Determination:** Identify or construct the context for each object using its contextual attributes. This can be done by Clustering the context space or by directly using categorical or numerical attributes such as time, spatial coordinates, or demographic variables;

2. **Behavioral Deviation Analysis:** Within each context group, apply a standard outlier detection method (statistical, distance-based, or density-based) to the behavioral attributes to find objects that deviate from the local norm.

When contextual data are sparse or continuous, probabilistic or regression-based models can be used to predict the expected behavior under a given context. For each object, the deviation between observed and expected behavior determines its degree of Outlierness. A general probabilistic representation is $P(V|U)$, where U represents the contextual attributes and V the behavioral attributes. Objects with low conditional probability under this model are contextual outliers:

$$\text{Outlier\_Score}(o) = \sum_{U_j} p(o \in U_j) \sum_{V_i} p(o \in V_i) P(V_i|U_j)$$

Model-Based Approach

Model-based contextual outlier detection explicitly estimates the dependency between context and behavior. Examples include:

- **Regression models:** predict expected behavioral values given context. Large residuals indicate outliers.
- **Hidden Markov Models (HMM):** capture temporal or sequential dependencies, where deviations in state transitions signal anomalies.

These approaches are effective for spatio-temporal or event-sequence data, where context and behavior evolve over time or space.

Challenges:

- Defining appropriate contextual attributes and modeling their relationship with behavioral attributes.
- Ensuring sufficient data within each context for reliable estimation.
- Handling overlapping or continuous contexts efficiently.

Collective Outliers

The detection of collective outliers typically involves three main steps:

1. **Structure Identification:** Define the structure or grouping of related objects, such as time windows, spatial regions, or subgraphs;
2. **Feature Extraction:** Represent each structured group as a higher-level object using aggregated or derived features that capture its collective behavior;
3. **Outlier Detection:** Apply standard outlier detection methods to these group-level representations to identify those that deviate from normal collective behavior.

Alternatively, model-based approaches can learn the expected dependency pattern among related objects and flag subsets that significantly violate it. Examples include Markov models for time-series, graphical models for spatial data, and subgraph analysis for networked data.

Distinction Between Contextual and Collective Outliers

While both types involve dependency structures, the distinction is that contextual outliers are defined for individual objects whose normality depends on external contextual variables, whereas collective outliers concern groups of objects whose joint behavior diverges from the global pattern, regardless of their individual normality.

Challenges:

- Defining appropriate relationships or group structures among data points.
- Extracting representative features for complex or high-dimensional groups.
- Managing computational complexity in large temporal or spatial datasets.

6.7 High-Dimensional Outlier Detection

High-Dimensional data pose specific challenges for **outlier detection**. Distances concentrate and become less discriminative as dimensionality grows. Data are often sparse across many dimensions and normal/abnormal behavior may manifest only in low-dimensional subspaces. Interpreting why a point is anomalous becomes harder because many features may contribute. Finally, the number of candidate subspaces grows exponentially, so methods must be both selective and scalable.

Three main strategies are applied:

- **Extend conventional methods.** Use rank-based or relative distances instead of absolute ones. Apply dimensionality reduction. Ranking k-nearest distances reduces scale sensitivity.
- **Search in subspaces.** Project data onto low-dimensional subspaces and detect low-density regions. Grid-based methods discretize dimensions and measure a *sparsity coefficient*, showing deviation from expected occupancy. Negative sparsity values indicate possible outliers. Heuristic or evolutionary searches reduce computational cost. Principal Component Analysis (PCA) can reveal outliers on low-variance components.
- **Use angle-based models.** Replace unreliable distance metrics with angular analysis. For a point p, compute angles between vectors $\overrightarrow{px}$ and $\overrightarrow{py}$ for pairs $(x; y)$. The variance of these angles defines an outlier score: small variance ⇒ likely outlier, large variance ⇒ likely normal. The Angle-Based Outlier Factor (ABOF) uses distance-weighted angle variance, efficient approximations exist.

They offer several **advantages**:

- Methods adapted to high-D reduce false signals from distance concentration.
- Subspace and angle-based approaches improve interpretability by localizing the anomalous dimensions or directions.

However, they also present some **limitations**:

- Search over subspaces is combinatorial and requires heuristics.
- Angle-based measures can be computationally heavy and need approximations for large datasets.
- Dimensionality reduction (e.g., PCA) may suppress meaningful anomalies if applied without care.

6.7.1 Talk about ABOF. Why use angles instead of distances?

The **ABOF** (**Angle-Based Outlier Factor**) is a method specifically designed for detecting outliers in **high-dimensional data**. In high-dimensional spaces, traditional distance-based measures lose discriminative power because distances between points tend to become similar due to the **curse of dimensionality**. As dimensionality increases, all pairwise distances converge, making it difficult to distinguish dense regions from sparse ones. The ABOF algorithm addresses this limitation by exploiting the distribution of **angles** between data points rather than relying directly on distances.

In high-dimensional spaces, while distances become less informative, the **variance of angles** between vectors remains meaningful. Intuitively, normal data points tend to be surrounded by other points uniformly distributed in many directions, resulting in a wide spread of angles. In contrast, an outlier typically lies in a region where most other points are located in a similar direction relative to it, resulting in **small angular variance**. Therefore, measuring the spread of angles around a point provides a robust indicator of its isolation in high dimensions.

Angles are less affected by the scale of the data and by distance concentration. Thus, the use of angular variance allows distinguishing isolated points from dense clusters even when distances themselves are nearly uniform.

For a given data object o, consider all other pairs of objects $(x_i; x_j)$ in the dataset. Define the vectors $\overrightarrow{ox_i}$ and $\overrightarrow{ox_j}$, representing the directions from o to those points. The algorithm measures the **variance of the angles** formed by these vectors, weighted by the distances between the objects.

The *ABOF* of o is defined as:

$$\text{ABOF}(o) = \text{Var}_{i;j}\left(\frac{\angle(\overrightarrow{ox_i};\overrightarrow{ox_j})}{\|x_i - o\|^2 \cdot \|x_j - o\|^2}\right) = Var_{i,j}\left(\omega_{i,j} \cdot < \overrightarrow{ox_i};\overrightarrow{ox_j} >\right) =$$

$$= \text{Var}_{i;j}\left(\frac{\|x_i - o\| \cdot \|x_j - o\| \cdot \cos(\theta_{i,j})}{\|x_i - o\|^2 \cdot \|x_j - o\|^2}\right) = \text{Var}_{i;j}\left(\frac{\cos(\theta_{i,j})}{\|x_i - o\| \cdot \|x_j - o\|}\right)$$

where the weighting term $\omega_{i,j}$ is typically defined as:

$$\omega_{i,j} = \frac{1}{\|x_i - o\| \cdot \|x_j - o\|}$$

This weighting emphasizes the contribution of closer neighbors, reducing the influence of faraway points. The variance is computed over all pairs $(x_i; x_j)$ in the dataset (excluding o). The formula includes distances in the denominator to give more weight to nearby points. This helps keep the algorithm effective even on lower-dimensional datasets where angles alone might be less reliable.

Interpretation of ABOF Values:

- Points in dense regions have **large angular variance**, since other points surround them in many directions. They therefore have high ABOF values and are considered normal.
- Points in sparse regions (potential outliers) have **small angular variance**, as most other points lie in similar directions. They thus have low ABOF values and are identified as outliers.

Hence, the Outlierness of an object is *inversely* related to its ABOF value: smaller ABOF values indicate stronger Outlierness. The ABOF is calculated for all objects and the results are sorted in ascending order to find the lowest ones.

ABOF works as follows:

1. For each data object o, compute the vectors $\overrightarrow{ox_i}$ to all other points;
2. For each pair of vectors $(\overrightarrow{ox_i}, \overrightarrow{ox_j})$, compute the cosine of the angle between them:

$$\cos(\theta_{i,j}) = \frac{(\overrightarrow{ox_i} \cdot \overrightarrow{ox_j})}{\|\overrightarrow{ox_i}\| \, \|\overrightarrow{ox_j}\|}$$

3. Compute the variance of these cosine values, weighted by $\omega_{i,j} = 1/(\|o - x_i\| \, \|o - x_j\|)$. Compute ABOF in practice;
4. Assign to o its ABOF score as the resulting weighted variance;
5. Rank objects by increasing ABOF values. Points with the lowest scores are the strongest outliers.

The naive computation of ABOF involves evaluating all pairs of points for each object, resulting in a complexity of $O(n^3)$, which is impractical for large datasets. To improve efficiency, an approximate version called **Fast-ABOF** limits the computation to the k nearest neighbors of each point. This reduces complexity to approximately $O(n \cdot k^2)$ while preserving accuracy, since the local neighborhood provides most of the discriminative information.

ABOF offers several **advantages**:

- Effective for high-dimensional data where distance-based methods fail;
- Robust to scale and data sparsity due to angle-based reasoning;
- Does not require strong assumptions about data distribution or density.

However, they also present some **limitations**:

- Computationally expensive for large datasets without approximation;
- Sensitive to noise and requires careful selection of neighborhood size in Fast-ABOF;
- Less intuitive to interpret compared to distance-based scores.

Chapter 7: Time Series

7.1 Introduction

7.1.1 What is a Time Series?

A **Time Series** is defined as a **set of data points** that are **ordered in time** and recorded at **equally spaced intervals** such as hours, minutes, months, or quarters. This **intrinsic order** is what differentiates time series from other regression tasks, as the *sequential position* of the data is fundamental to understanding the underlying process and because it is **possible to forecast time series without the use of features**. The primary objective of analyzing these datasets is **forecasting**, which is the practice of **predicting the future** by utilizing **historical data** and incorporating knowledge of **future events** that might influence the results.

A critical step in time series analysis is **Decomposition**, a statistical procedure that separates an observed series into interpretable components in order to simplify modeling, interpretation, and forecasting. In its additive form, a time series y_t can be written as:

$$y_t = T_t + S_t + R_t,$$

where each component captures a distinct source of variation.

- **Trend:** The trend component T_t describes the long-term, slow-moving evolution of the time series. It reflects persistent changes in the level of the data, such as gradual growth, decline, or structural shifts over time. In time series modeling, identifying the trend is crucial because it captures low-frequency dynamics that dominate long-horizon behavior and often need to be removed or explicitly modeled to achieve stationarity.
- **Seasonality:** The seasonal component S_t represents systematic, periodic patterns that repeat over a fixed and known time interval, such as daily, weekly, monthly, or yearly cycles. Seasonality arises from recurring external factors (e.g., calendar effects, climate, or business cycles) and introduces regular fluctuations around the trend. Properly modeling seasonality improves forecast accuracy and prevents these predictable patterns from being misinterpreted as random variation.
- **Residuals:** The residual (or irregular) component R_t contains the remaining variation after removing trend and seasonality. It is typically interpreted as random noise, short-term shocks, or irregular behavior that cannot be explained by systematic components. In a well-specified decomposition, residuals should exhibit no clear structure and resemble a stationary noise process, any remaining autocorrelation may indicate that the trend or seasonal components require refinement.

The *methodological approach* used for forecasting usually depends on the *scale of the data* available.

Traditional statistical approaches, such as the **ARIMA** or **SARIMAX** models, are particularly effective for **smaller datasets** containing fewer than 10,000 points. These models rely heavily on the concept of *stationarity*, where the *mean and variance* of the series remain constant over time.

For **larger datasets**, **machine learning approaches** and **deep learning architectures** like the **Long Short-Term Memory** (**LSTM**) network are preferred. These modern models utilize a **data windowing process** to organize the series into *inputs and labels*, allowing the network to capture **complex non-linear relationships** that traditional methods might miss.

Let's imagine a very simple mathematical model called AR(1) (Autoregressive of order 1), where today's value (y_t) depends on yesterday's value (y_{t-1}) plus the noise/residual (ε_t) plus a constant (C):

$$y_t = C + \phi_1 y_{t-1} + \varepsilon_t$$

where ϕ_1 is the unit root.

7.1.2 Why are the mean, the variance and the autocorrelation important for Time Series?

The **mean**, the **variance**, and the **autocorrelation** represent the three fundamental *statistical properties* used to define the behavior and stability of a time series. The **mean** is the *average value* of the series across a specific period. In the context of forecasting, a **constant mean** indicates the absence of a *trend*, ensuring that the series oscillates around a stable level. The **variance** measures the **dispersion of data points** around that mean, representing the *power or volatility* of the signal. For a series to be considered *stationary*, the variance must also remain constant over time, a condition known as **homoscedasticity**. If the variance changes, the model may struggle to produce *reliable prediction intervals*.

The **autocorrelation** is perhaps the most critical property for *model identification*, as it measures the **linear relationship** between the current value of a series and its **past values** at specific *lags*. This is vital because time series data is *self-dependent*, meaning past observations carry information about future ones.

$$r_k = \frac{\sum_{t=k+1}^{T}(y_t - \bar{y})(y_{t-k} - \bar{y})}{\sum_{t=1}^{T}(y_t - \bar{y})^2}$$

Where:

- T is the total number of observations in the time series;
- y_t is the value of the series at time t;
- $\bar{y}$ is the mean value of the time series;
- k is the lag (the number of timesteps separating the values).

Theoretically, autocorrelation r_k is the ratio of the autocovariance at lag k to the variance of the series.

7.1.3 What is a baseline model and why is it essential for evaluation? What are MAE and MAPE and when to use one over the other?

A **baseline model**, often referred to as a *trivial solution*, is a **simplistic forecasting method** used to establish a minimum performance benchmark. In the context of time series, the most common baseline is the **naïve forecast**, which simply predicts that the *next value* will be equal to the **last observed value**. Another frequently used approach is the *seasonal naïve forecast*, where the prediction for the current period is set equal to the value observed in the same period of the *previous cycle*.

Using a baseline is **essential** because it provides a *reference point* for evaluating **complex forecasting models** such as **SARIMAX** or **Long Short-Term Memory (LSTM)** networks. A sophisticated model is only considered **valuable** or *useful* if it can **outperform the baseline**. If a complex deep learning architecture yields an error rate similar to or higher than a simple naïve forecast, the *computational overhead* and *architectural complexity* are not justified, indicating that the model is failing to capture the **underlying temporal patterns** effectively.

To quantify this performance, we use **evaluation metrics** such as the **Mean Absolute Error (MAE)** and the **Mean Absolute Percentage Error (MAPE)**. The **MAE** measures the *average magnitude* of the errors in a set of predictions without considering their direction. It is calculated using the following formula:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

where y_i is the **actual value** and $\hat{y}_i$ is the **forecasted value**. Since the MAE is *scale-dependent*, its result is expressed in the **same units** as the original data, making it intuitive for understanding the *physical error* of the system.

The **MAPE** provides a *relative measure* of error by expressing it as a **percentage** of the actual values. It is defined by the formula:

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100$$

Because the MAPE is *scale-independent*, it allows for the **comparison of forecast accuracy** across different time series that may have **vastly different magnitudes**.

The decision to use one metric over the other depends on the *objective of the analysis*:

- Use **MAE** when the **absolute cost of the error** is the main priority and the data remains within a *stable range*. It is the preferred choice when communicating with stakeholders who need to know the error in *real-world units*;
- Use **MAPE** when you need to **compare model performance** across different datasets or when the *relative impact* of the error is more critical than its absolute value. However, it is important to note that MAPE cannot be used if the dataset contains **zero values**, as the *division by zero* would lead to undefined results;
- In **machine learning approaches**, both metrics are often reported together to provide a *comprehensive view* of how the model handles both the **scale of the data** and the **relative precision** of the forecasts.

7.2 Traditional Approaches

7.2.1 Why is Stationarity important for Time Series?

A **stationary time series** is defined as a process where the *statistical properties*, specifically the **mean**, **variance**, and **autocorrelation**, remain constant over time and do not depend on the *time of observation*. Stationarity is a fundamental requirement for most **traditional statistical models**, such as the **ARIMA** and **SARIMAX** families, because these models are designed to capture *linear dependencies* that are expected to persist. If a process is stationary, we can assume that the **patterns observed in historical data** will remain valid in the **future**, which is the essential logic required for **reliable forecasting**.

When a series is *non-stationary*, it often exhibits a **trend** or **seasonality**, meaning the *mean value* or the *fluctuations* change over time. Modeling such data without transformation leads to **spurious regressions**, where the model captures *coincidental correlations* that do not represent a true underlying relationship. To fix this, mathematical **transformations** are applied to stabilize the data:

- **Differencing**: This involves calculating the change between consecutive observations to remove a **trend** and stabilize the **mean**. In practice, you subtract each row from the previous one;
- **Logarithmic Transformation**: This is used to stabilize the **variance** when the size of the fluctuations is proportional to the level of the series. In practice, you apply directly the logarithm to the raw data.

The **Augmented Dickey-Fuller (ADF) test** is the primary tool used to verify stationarity by checking for the presence of a *unit root*. The **null hypothesis** of the test assumes the series is non-stationary. Only when the resulting **p-value** is low, typically below 0.05, can we reject the null hypothesis and proceed with **model identification** using tools like the *ACF and PACF plots*. Even in **machine learning approaches**, where models like **LSTM** are more flexible in handling *non-linearities*, ensuring the data is stationary or properly scaled remains a **standard best practice** to ensure the *stability and convergence* of the neural network during training.

7.2.2 How does the ADF test work?

The **Augmented Dickey-Fuller (ADF) test** is a *formal statistical procedure* used to detect the presence of a **unit root**, which signifies **non-stationarity**. It operates by fitting an **autoregressive model** and testing the coefficient γ in the equation: $\Delta y_t = \gamma y_{t-1} + \sum \beta_i \Delta y_{t-i} + \varepsilon_t$. Here, $\gamma = \phi - 1$, where ϕ is the first-order lag coefficient. If $\phi = 1$, the series follows a **random walk** where shocks have permanent effects. The "augmented" component incorporates *lagged differences* (Δy_{t-i}) to account for **higher-order serial correlation**. This ensures that the *residuals* (ε_t) behave like **white noise**, eliminating *estimation bias* and making the test robust for complex, real-world data where temporal dependencies extend across multiple lags.

The test evaluates the stochastic properties of the series through two *formal hypotheses*:

- **Null Hypothesis (H_0)**: $\gamma = 0$ ($\phi = 1$). The series **has a unit root**, meaning it is **non-stationary** and possesses a *time-dependent variance* or trend;
- **Alternative Hypothesis (H_1)**: $\gamma < 0$ ($\phi < 1$). The series **is stationary** and *mean-reverting*, indicating that its statistical properties are **time-invariant**.

To *interpret the results*, the analyst examines the **p-value** and the **ADF statistic**. If the **p-value is less than 0.05** (or the statistic is **more negative** than the *critical values*), we **reject the null hypothesis** in favor of stationarity. Conversely, if we **fail to reject** H_0, the series is non-stationary and must undergo **transformations**, typically **differencing** (I) or *log-scaling*, to stabilize the mean and variance. This preprocessing is mandatory before applying traditional forecasting frameworks like **ARIMA**, as these models require stationary inputs to ensure that the estimated correlations are not *spurious* and that the forecasts remain reliable over time.

7.2.3 What are Moving Average and Autoregressive Processes and what is their conceptual difference?

A **Moving Average (MA)** process models the current value of the series as a **linear combination of past forecast errors** or *innovations*. In an $MA(q)$ model, the parameter q defines the **window of past shocks** that influence the current state. Unlike the AR model, which relies on the *observed values* of the variable, the MA model relies on the **unpredictable white noise** from previous steps to explain the current *deviation from the mean*.

$$y_t = \mu + \varepsilon_t + \theta_1\varepsilon_{t-1} + \theta_2\varepsilon_{t-2} + ... + \theta_q\varepsilon_{t-q}$$

An **Autoregressive (AR)** process is a model where the current value of a time series is expressed as a **linear combination of its own past values** and a *random error term*. In an $AR(p)$ model, the parameter p represents the *order*, indicating how many **previous timesteps** are used to predict the current one. Mathematically, it assumes that the *internal momentum* of the series drives its future behavior, meaning that if a value was high in the past, it is likely to remain high unless a **stochastic shock** occurs.

$$y_t = C + \phi_1 y_{t-1} + \phi_2 y_{t-2} + ... + \phi_p y_{t-p} + \varepsilon_t$$

The **conceptual difference** between the two lies in the *nature of the dependency*:

- **Autoregressive Process**: It represents a **system with memory** where past states directly influence the current state, creating a *persistent effect* that decays over time;
- **Moving Average Process**: It represents a **system reacting to shocks**, where a random event (an error) only affects the series for a *finite number of steps* before its influence disappears completely.

In practice, these processes are identified using *correlation plots*. For an **AR process**, the **Partial Autocorrelation Function (PACF)** will *cut off* after lag p, while the **Autocorrelation Function (ACF)** will *tail off* exponentially. For an **MA process**, the opposite occurs: the **ACF** will *cut off* after lag q, while the **PACF** will *tail off*. Understanding these differences is essential for the **Box-Jenkins methodology**, which allows researchers to select the most appropriate **ARIMA parameters** to ensure that the final *residuals* approximate white noise.

In many real-world scenarios, a time series displays characteristics of both *momentum* and *short-term shocks*, making a simple AR or MA model insufficient. To capture these complex dynamics, we combine them into the **Autoregressive Moving Average (ARMA)** model. An $ARMA(p;q)$ model expresses the current value of the series as a **linear combination** of its own **past values** (the AR part) and its **past forecast errors** (the MA part). The general mathematical formula for an $ARMA(p;q)$ process is:

$$y_t = C + \phi_1 y_{t-1} + ... + \phi_p y_{t-p} + \varepsilon_t + \theta_1\varepsilon_{t-1} + ... + \theta_q\varepsilon_{t-q}$$

where p and q represent the *orders* of the AR and MA components, respectively. This combination allows the model to be **parsimonious**, meaning it can describe the data with *fewer parameters* than a high-order AR or MA model used alone.

7.2.4 Describe the Autocorrelation Function and the Partial Autocorrelation Function

The **Autocorrelation Function (ACF)** and the **Partial Autocorrelation Function (PACF)** are the primary *diagnostic tools* used in the **Box-Jenkins methodology** to identify the orders of stationary time series models. The **ACF** measures the **total correlation** between an observation y_t and its previous value y_{t-k} across a range of lags. The mathematical formula for the *sample autocorrelation* at lag k is:

$$r_k = \frac{\sum_{t=k+1}^{T}(y_t - \bar{y})(y_{t-k} - \bar{y})}{\sum_{t=1}^{T}(y_t - \bar{y})^2}$$

where T is the total number of observations and $\bar{y}$ is the **mean of the series**. This function captures both *direct relationships* and *indirect effects* passed through intervening observations.

In contrast, the **PACF** measures the **direct correlation** between y_t and y_{t-k} after **removing the influence** of all intermediate lags. Formally, the partial autocorrelation at lag k, denoted as ϕ_{kk}, can be found by solving the **Yule-Walker equations** or fitting an AR model of order k:

$$y_t = \phi_{k1} y_{t-1} + \phi_{k2} y_{t-2} + ... + \phi_{kk} y_{t-k} + \varepsilon_t$$

where ϕ_{kk} is the coefficient of interest. This isolation is crucial because it allows us to identify the specific *contribution* of a single lag without the *shadowing effect* of more recent observations.

To identify the **order of the process**, we examine the behavior of these plots to see if the coefficients *tail off* (decay toward zero) or **cut off** (abruptly drop to non-significance) after a certain lag:

- **Autoregressive Process ($AR(p)$)**: The **PACF** is the determining plot, as it will **cut off** immediately after lag p. Meanwhile, the **ACF** will *tail off* exponentially or in a sinusoidal fashion, reflecting the *memory effect* inherent in autoregressive systems;
- **Moving Average Process ($MA(q)$)**: The **ACF** is the determining plot, showing a **cut off** after lag q. Conversely, the **PACF** will *tail off* toward zero. This occurs because an MA process only has *finite memory* restricted to the window of past error terms;
- **ARMA Process**: If both the ACF and PACF *tail off* without a clear cut-off point, the series likely requires a **combination** of both AR and MA components.

The *significance* of these coefficients is usually judged against a **confidence interval** (typically 95%) plotted as dashed lines. Any coefficient that stays within these bounds is considered **statistically insignificant**, suggesting that the corresponding lag does not contribute meaningful information to the model. By correctly identifying these "spikes," the researcher can define the initial values for the **p** and **q** parameters in an **ARIMA** model before proceeding to the *estimation and validation* phases.

7.2.5 The Evolution of Traditional Models: From AR and MA to SARIMAX

A **Time Series** is defined as a **set of data points** that are **ordered in time** and recorded at **equally spaced intervals** such as hours, minutes, months, or quarters.

The development of traditional time series models follows a **logical and cumulative progression**, where each model is designed to overcome a specific *shortcoming* of its predecessors. Rather than replacing earlier approaches, newer models extend them, resulting in a flexible family of methods capable of handling increasingly complex data-generating processes. At the core of this evolution lie two fundamental building blocks: the **Autoregressive (AR)** process and the **Moving Average (MA)** process.

An **Autoregressive (AR)** process assumes that the current observation of a time series can be expressed as a *linear combination* of its own **past values**. This structure captures the idea that historical observations contain information about future behavior, allowing the model to represent persistence, inertia, and long-term dependence in the data. An $AR(p)$ model, where p denotes the number of lags considered, is defined as:

$$y_t = C + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + \varepsilon_t$$

Here, C is a constant term, ϕ_i are the autoregressive coefficients, and ε_t is a white-noise error term. The magnitude and sign of the coefficients determine how strongly and in what direction past values influence the present, making AR models particularly suitable for capturing smooth, persistent dynamics.

In contrast, the **Moving Average (MA)** process models the current value of the series as a function of **past forecast errors**, also referred to as random shocks. Rather than relying on the history of the series itself, MA models focus on how unexpected disturbances propagate over time. This makes them well suited for describing *short-term irregularities* that gradually decay. An $MA(q)$ model is expressed as:

$$y_t = \mu + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \cdots + \theta_q \varepsilon_{t-q}$$

where μ is the mean of the series, θ_j are the moving average coefficients, and ε_t is white noise. The MA structure captures the transitory impact of shocks, emphasizing error correction rather than long-term dependence.

To achieve greater **parsimony**—the ability to explain complex behavior using a minimal number of parameters—the AR and MA components are combined into the **Autoregressive Moving Average (ARMA)** model. By jointly modeling dependence on past values and past errors, ARMA models can represent a wide range of autocorrelation structures more efficiently than either AR or MA models alone. The general form of an $ARMA(p, q)$ model is:

$$y_t = C + \phi_1 y_{t-1} + \cdots + \phi_p y_{t-p} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q}$$

This combination allows the model to capture both persistent trends in behavior and short-lived shocks within a single, unified framework.

A key limitation of ARMA models is the requirement that the time series be **stationary**, meaning its statistical properties (such as mean and variance) do not change over time. Since many real-world time series exhibit **trends** or evolving levels, the **Autoregressive Integrated Moving Average (ARIMA)** model was introduced to address this issue. The "Integrated" component, denoted by d, refers to the application of **differencing** operations that remove non-stationary behavior by stabilizing the mean of the series. An $ARIMA(p; d; q)$ model applies an ARMA structure to the differenced series y'_t:

$$y'_t = C + \phi_1 y'_{t-1} + \cdots + \phi_p y'_{t-p} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q}$$

Through differencing, ARIMA models are able to handle trending data while retaining the interpretability and structure of ARMA models.

Despite this improvement, ARIMA models remain inadequate for series exhibiting **seasonality**, that is, patterns that repeat at regular intervals such as monthly, quarterly, or yearly cycles. To capture these periodic behaviors, the **Seasonal ARIMA (SARIMA)** model extends ARIMA by introducing additional seasonal autoregressive, differencing, and moving average terms, denoted as $(P; D; Q)_m$, where m represents the seasonal period. These seasonal components explicitly model correlations at seasonal lags, enabling the representation of recurring cyclical structures. The parameter m denotes the number of observations per seasonal cycle, determined by the data frequency(annual $m = 1$, monthly $m = 12$, etcetera ...).

The culmination of this evolutionary path is the **SARIMAX** model, which further extends SARIMA by incorporating a **linear combination of exogenous variables** (X). These external regressors allow the model to account for influences that originate outside the time series itself, such as economic indicators, weather conditions, or marketing interventions. By integrating both internal dynamics and external drivers, SARIMAX offers a powerful and flexible framework for real-world forecasting. However, this added flexibility comes at the cost of increased complexity, as reliable long-term forecasts require not only modeling the target series but also generating or assuming future values for the exogenous predictors.

7.2.6 How do you choose the "best" model among several candidates?

Selecting the **optimal model** from a pool of candidates is a critical step in the *Box-Jenkins methodology* for time series forecasting. While it is tempting to choose the model that yields the *lowest training error*, this approach often leads to **overfitting**, where the model captures random noise instead of the underlying signal. To avoid this, we use the **Akaike Information Criterion** ($\mathbf{AIC} = 2k - 2\ln(L)$), which provides a quantitative way to compare models by balancing **accuracy** against **simplicity**.

The AIC achieves this balance by considering two *competing components*:

- **Goodness of Fit**: This is represented by the **log-likelihood** ($\ln(L)$), which measures how likely it is that the observed data was generated by the specific model parameters;
- **Model Complexity**: This is represented by a **penalty term** ($2k$) that is proportional to the number of *estimated parameters* ($k = p + q + P + Q + \text{num}(X)$) in the model.

The principle of **parsimony** suggests that, among several models that explain the data equally well, the *simplest one* should be preferred. Mathematically, as the number of parameters increases, the *fit improves* and the log-likelihood term becomes more negative (which would reduce the AIC), but the penalty term simultaneously increases. The **best model** is defined as the one that achieves the **minimum AIC value**. This value represents the point where the *marginal gain in fit* no longer justifies the *cost of adding more complexity.*

In practical workflows, researchers often fit every combination of parameters within a range (for example, searching through various **ARIMA** $(p; d; q)$ and **SARIMA** $(P; D; Q)_m$ orders) and use the AIC to *rank* them. Once the model with the **lowest AIC** is selected, it is essential to perform a final **residual analysis** to ensure the residuals approximate *white noise*, confirming that the chosen model is not only parsimonious but also **statistically sound**.

7.2.7 Once a model is fitted, why is it necessary to perform a residual analysis?

After estimating the parameters of a time series model such as an **ARIMA** or **SARIMAX** process, performing a **residual analysis** is the final and most critical step of the *Box-Jenkins methodology*. The residuals represent the *errors* or the difference between the **observed values** and the **predicted values** produced by the model. The fundamental goal of this analysis is to ensure that these residuals approximate **white noise**, which means they should have a *mean of zero*, a *constant variance*, and **no remaining autocorrelation**. If the residuals still contain *systematic patterns* or *serial correlation*, it indicates that the model has failed to capture all the **available information** in the data, meaning the forecast is **suboptimal** and the model must be refined.

To assess the quality of the residuals, researchers employ **qualitative methods** for visual inspection. The **Normal Q-Q plot** (Quantile-Quantile plot) is used to determine if the residuals follow a **normal distribution**. This tool plots the **theoretical quantiles** of a standard normal distribution against the **sample quantiles** of the residuals:

- If the residuals are normally distributed, the points will fall approximately along a **straight diagonal line**;
- Significant *deviations* from this line at the ends suggest the presence of **heavy tails** or *skewness*, indicating that the model does not account for **extreme values** or **outliers** effectively.

While visual plots are helpful, **quantitative methods** provide a rigorous statistical foundation for model validation. The **Ljung-Box test** is a *portmanteau test* used to examine whether the **autocorrelations** of the residual series are **significantly different from zero** for a group of lags. This test is defined by:

1. A **Null Hypothesis (H_0)** stating that the residuals are **independently distributed**, meaning they represent *white noise*;
2. An **Alternative Hypothesis (H_1)** stating that the residuals exhibit **serial correlation**, meaning the model is *missing temporal structure*.

To interpret the test, we examine the **p-value**. If the **p-value is greater than 0.05**, we **fail to reject the null hypothesis**, concluding that the residuals are *random noise* and the model is **statistically sound**. If the p-value is low, the model is *inadequate* because the residuals are **still predictable**, suggesting that the orders p or q may need to be increased to capture the remaining *lagged dependencies*.

7.2.8 Describe the traditional workflow of the traditional approaches on Time Series Forecasting

The traditional approach to forecasting follows a structured, iterative sequence known as the **Box-Jenkins methodology**, which is summarized by the pipeline: **Check Stationarity** $\rightarrow$ **Transform** $\rightarrow$ **Identify** p, d, q **via ACF/PACF** $\rightarrow$ **Fit** $\rightarrow$ **Validate Residuals** $\rightarrow$ **Forecast**. The process begins with **visualizing the series** to identify *long-term trends* and *seasonal patterns*. If the **variance** is not constant over time, a preliminary **logarithmic transformation** is often applied to stabilize the fluctuations.

Once the data is visually inspected, the researcher must conduct a **stationarity test**, specifically the **Augmented Dickey-Fuller (ADF) test**. If the p-value indicates the presence of a *unit root*, the series is non-stationary and requires **differencing** (d). This step is repeated until the p-value falls below the *significance threshold*, confirming that the *mean and variance* are time-invariant. Following this, **ACF and PACF plots** are generated to hypothesize the initial orders for the **Moving Average (MA)** and **Autoregressive (AR)** components, as well as any *seasonal orders* (P, D, Q).

The modeling phase involves a **grid search** across various candidate models. We select the **optimal model** by minimizing the **Akaike Information Criterion (AIC)**, which ensures a balance between *goodness of fit* and **model parsimony**. After fitting, it is mandatory to **validate the residuals** to ensure they approximate **white noise**.

This diagnostic check is performed through:

- **Qualitative analysis**: Inspecting *standardized residuals* and the **Normal Q-Q plot** to verify that errors are *normally distributed*;
- **Quantitative analysis**: Applying the **Ljung-Box test** where a **p-value greater than 0.05** is ideal, indicating that no *significant autocorrelation* remains in the residuals.

In more complex scenarios involving **SARIMAX**, the researcher must also evaluate the influence of **exogenous variables**. A critical limitation of this approach is that forecasting multiple steps into the future requires having *pre-existing forecasts* for these external predictors. To assess the *real-world reliability* of the model, a **rolling forecast** is used to calculate **out-of-sample performance** metrics such as the **Mean Absolute Percentage Error (MAPE)**.

Finally, several *practical warnings* must be considered during this workflow:

1. If the series behaves like a **random walk**, traditional models face severe limits and often cannot outperform a **naïve forecast** (trivial solution);
2. One should **not trust the AIC alone**, as a low AIC does not guarantee that the residuals are free of *serial correlation*;
3. Performance must always be confirmed through **diagnostic checking** and **out-of-sample validation** to ensure the model generalizes well to unseen data.

7.3 Machine Learning Approaches

7.3.1 When are Machine Learning approaches better?

Machine Learning approaches are generally preferred over traditional statistical methods when the **dataset size** is large, typically exceeding **10,000 data points**. While statistical models like ARIMA are highly effective for small datasets, they become computationally *inefficient and slow* to fit as the number of observations grows. Furthermore, machine learning is the ideal choice when traditional models result in **correlated residuals** that do not approximate *white noise*, indicating that the linear assumptions of classical statistics have failed to capture the **complex non-linear relationships** present in the data.

Beyond the dataset scale, machine learning is often utilized when the problem involves *high-dimensional exogenous variables* or when the *underlying patterns* are too irregular for seasonal differencing to resolve. Regarding the strategies used for forecasting, there are three primary **types of machine learning models**:

- **Single-step model**: This architecture is designed to output a **single value** representing the prediction for the **next immediate timestep**. Although the input window can be of any length to provide context, the model focuses exclusively on the *point estimate* of the very next observation;
- **Multi-step model**: This approach generates a **sequence of values** representing predictions for **many timesteps** into the future. It is particularly useful for *long-term planning*, such as predicting the hourly traffic volume for an entire day based on the previous week's history;
- **Multi-output model**: This model produces predictions for **more than one target variable** simultaneously. An example would be a system that forecasts both **temperature and wind speed** at the same time, allowing the network to learn the *shared correlations* and *physical dependencies* between different sensors.

In modern practice, these models often rely on **Deep Learning** architectures, such as **Dense Networks**, **Recurrent Neural Networks (RNN)**, or **Long Short-Term Memory (LSTM)** units. These models require a **data windowing** process where the time series is restructured into *overlapping windows* of inputs and labels, enabling the model to treat the temporal forecasting task as a **supervised learning** problem.

7.3.2 Talk about the Data Windowing process

Before a time series can be processed by a **machine learning model**, the raw temporal data must undergo a rigorous *preprocessing phase*. Raw timestamps, typically expressed in the format **YYYY-MM-DD HH:MM:SS**, cannot be fed directly into a neural network because they are strings or *unscaled integers*. The first step is to transform these into **numerical features**. While extracting the hour or day as a simple integer is possible, it creates a *continuity problem* where the model perceives 23:59 and 00:01 as being far apart. To solve this, we use **cyclical encoding** by applying **sine and cosine transformations**. This project's time onto a *unit circle*, ensuring that the **temporal proximity** between the end of one cycle and the beginning of the next is mathematically preserved.

Once the time features are encoded, the next phase is the **Data Windowing process**. This is the fundamental technique used to transform a *continuous sequence* of data into a **supervised learning problem** consisting of **inputs (features)** and **labels (targets)**. In this framework, the model learns to map a specific *history of observations* to a future value or sequence. This process is governed by three critical **parameters**:

- **Input Width**: The number of **past timesteps** that the model uses as the "context" or history to make a prediction;
- **Label Width**: The number of **future timesteps** that the model is tasked with predicting, which determines if the model is *single-step* or *multi-step*;
- **Shift**: The **time offset** between the end of the input window and the start of the label window, defining how far into the future we are looking.

The combination of these parameters defines the **total window width**. To maximize the amount of training data available, we apply a **sliding window approach**. Instead of creating non-overlapping blocks, we move the window forward by a **step of 1**, allowing the model to see almost every possible *sub-sequence* in the dataset. This results in a **three-dimensional tensor** shape, usually represented as *(batch, time, features)*, which is the standard input format for architectures like **Recurrent Neural Networks (RNN)** or **Long Short-Term Memory (LSTM)** networks. This structured organization is essential because it allows the model to capture the **temporal dependencies** and *lagged correlations* that are inherent in time series data.

7.3.3 What are the Traditional Machine Learning Approaches for Time Series Forecasting?

The transition from traditional statistical methods to **machine learning approaches** requires a fundamental shift in how temporal data is represented and processed. Unlike classical models that work directly with the raw values of a series, machine learning requires the **engineering of features** to represent the passage of time. Raw timestamps, typically formatted as **YYYY-MM-DD HH:MM:SS**, are converted into *numerical signals* using **cyclical encoding**. By applying **sine and cosine transformations** to time units like hours or months, we map the data onto a *circular coordinate system*. This ensures that the model mathematically understands that the end of one cycle is *temporally adjacent* to the beginning of the next, preventing the **continuity break** that occurs with simple linear integers.

Once the features are encoded, the series is restructured through the **Data Windowing process** to create a **supervised learning dataset**. This process organizes the data into **fixed-size windows** consisting of **inputs (features)** and **labels (targets)**. This transformation is defined by three specific *structural parameters*: the **Input Width**, which is the number of *historical timesteps* provided to the model. The **Label Width**, which is the number of *future timesteps* to be predicted, and the **Shift**, which is the *temporal offset* between the input and the target.

Regarding the **forecasting strategies** available, machine learning approaches are classified into three primary categories based on the structure of their outputs:

- **Single-step model**: This is the most *straightforward approach*, where the model is designed to output a **single value** representing the prediction for the **next immediate timestep**. While the input window can be of any length to provide context, the output remains a **point estimate** for $t + 1$. This is often used as a *baseline* for more complex architectures;
- **Multi-step model**: In this configuration, the model generates a **sequence of values** representing predictions for **multiple timesteps** into the future. This is essential for *planning horizons* where a single value is insufficient, such as forecasting electricity demand for the next 24 hours. These models must capture the **temporal evolution** of the series over the entire predicted range;

- **Multi-output model**: This approach involves generating predictions for **more than one target variable** at the same time. For instance, a model might simultaneously forecast **temperature and wind speed**. This is a *powerful strategy* because it allows the neural network to learn the **inter-dependencies** and **correlations** between different sensors or variables, often leading to better performance than training separate models for each target.

7.3.4 What are the Deep Learning Approaches for Time Series Forecasting?

Deep Learning models offer a powerful alternative to traditional methods when datasets are large and relationships are **highly non-linear**. These approaches treat forecasting as a *supervised learning* problem where a sequence of historical data points is mapped to one or more future values. The complexity of these models ranges from simple *linear regressions* to sophisticated **Recurrent Neural Networks** capable of long-term memory.

The evolution of these models can be categorized based on their architectural complexity and the specific forecasting task they address:

- **Simple linear model**: This is the most basic approach, where a single *dense layer* with a single output unit is used to perform a **single-step forecast**. It acts as a *linear baseline*, mapping the input window directly to the next predicted value without any hidden layers or non-linear activation functions;
- **Multi-step linear model**: This model extends the linear approach to predict a **sequence of future values**. The output layer contains a number of units equal to the **label width**, allowing the model to project a horizon of multiple timesteps in a single forward pass;
- **Multi-output linear model**: In this configuration, the model is designed to forecast **multiple target variables** simultaneously. By having an output unit for each variable at each timestep, the model learns the *shared linear trends* between different features of the dataset;
- **Deep Neural Network (DNN)**: These models introduce **stacked dense layers** between the input and the output. By utilizing *non-linear activation functions* like **ReLU**, a DNN can capture more **complex patterns** than a linear model. Like their linear counterparts, DNNs can be configured for **Simple** (one step), **Multi-Step** (sequence), or **Multi-Output** (multiple variables) forecasting tasks.

While dense networks are effective, they treat each point in the input window independently, failing to inherently account for the **sequential nature** of time. To solve this, **Recurrent Neural Networks (RNNs)** are employed. An RNN processes data step-by-step, maintaining an internal **hidden state** that acts as a *memory*. This state is updated at each timestep, allowing the network to pass information from the beginning of the sequence to the end. However, standard RNNs suffer from the *vanishing gradient problem*, which limits their ability to remember information across long sequences.

To overcome these limitations, the **Long Short-Term Memory (LSTM)** architecture was developed. The LSTM is a specialized type of RNN that introduces a **cell state**, a "conveyor belt" that allows information to flow through the entire sequence with minimal changes. The flow of information is strictly regulated by three **gate mechanisms**:

1. **Forget Gate**: Uses a *sigmoid activation* to decide which information from the previous cell state should be **discarded**;
2. **Input Gate**: Determines which new information from the current timestep is **relevant** enough to be **stored** in the cell state;
3. **Output Gate**: Combines the current input and the filtered cell state to produce the **hidden state**, which serves as the prediction for the current step and the memory for the next.

The use of *tanh activations* within these gates ensures that the values remain scaled, preventing the **gradients from exploding** during training. Because of this sophisticated control over memory, LSTMs are considered the **state-of-the-art** for deep learning in time series, as they can successfully identify **long-term dependencies** that simple dense or recurrent models might miss.

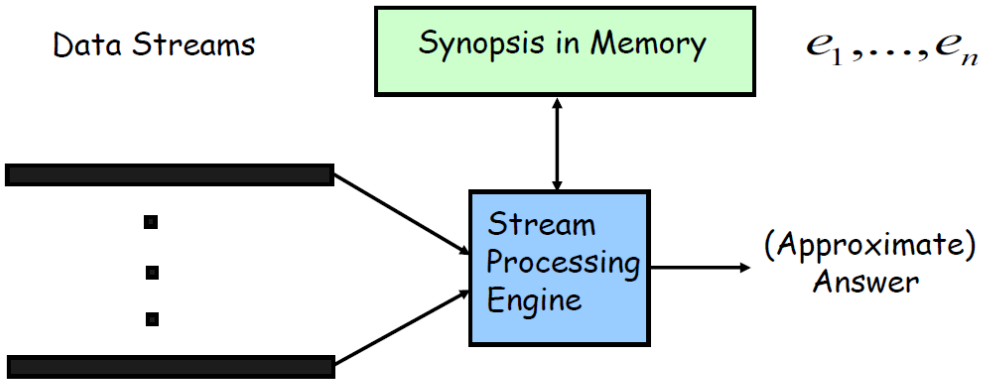
Chapter 8: Data Stream

8.1 Introduction

8.1.1 What is a Data Stream?

A **Data Stream** is defined as a **potentially unbounded** and **ordered sequence** of data instances that arrive continuously at a high velocity. Mathematically, a stream S can be represented as an infinite set $S = \{x_1, x_2, x_3, ..., x_N\}$ where each x_i is a *d-dimensional feature vector* and N tends toward **infinity**. Unlike traditional datasets, which are static and finite, data streams are **dynamic** and characterized by their *steady evolution*, meaning the underlying statistical properties of the data often change over time, a phenomenon known as **concept drift**.

The nature of data streams introduces significant **computational challenges** that render traditional data mining techniques impractical. Standard algorithms typically require the *entire dataset* to be present in memory and involve **multiple scans** of the data to build models. In contrast, Data Stream Mining must operate under **strict constraints**. **Online analysis** occurs when data flows into the system and is used to create a synopsis. **Offline analysis** occurs later, usually at the user's request or at regular intervals, to produce the final result.



- **Stream processing requirements**
  - **Single pass**: Each record is examined at most once
  - **Bounded storage**: Limited Memory (M) for storing synopsis
  - **Real-time**: Per record processing time (to maintain synopsis) must be low

Consequently, the answers provided by stream mining algorithms are generally **approximate** rather than exact. These approximations are often accompanied by *deterministic* or *probabilistic bounds* to ensure the reliability of the results. Practical applications of this technology are found in *network monitoring*, *financial transactions*, *sensor networks*, and *web server logs*, where the volume of data is too massive to be stored for *offline analysis*.

Only a synopsis of the input stream is stored: special data structures enable to incrementally summarize the input stream. Four commonly used data structures:

- **Feature Vectors:** summary of the data instances;
- **Prototype Arrays:** keep only a number of representative instances that exemplify the data;
- **Coreset Trees:** keep the summary in a tree structure;
- **Grids:** keep the data density in the feature space.

8.1.2 What is a Concept Drift?

Concept Drift is defined as an **unforeseen change** in the **statistical properties** of a data stream over time. In the context of machine learning, it represents a *shift in the relationship* between the **input features** and the **target variable**, which causes the patterns learned by a model to become *irrelevant or obsolete*. This phenomenon is a fundamental problem in **stream mining** because it violates the assumption that data distributions are *stationary*. As the underlying *concepts* evolve, the performance of mining techniques, such as **Classification or Clustering**, tends to degrade, leading to a significant drop in **prediction accuracy** and **model decay**.

There are four primary *types of concept drift* that determine how a system must adapt to the incoming data:

- **Sudden concept drift**: This occurs when the change between two consecutive instances happens **at once**. After this *abrupt shift*, only instances of the **new concept** are received, requiring the model to adapt *immediately* to remain valid;
- **Gradual concept drift**: This involves a *slower transition* where the number of instances belonging to the previous class **gradually decreases** while the number of instances from the new class **increases**. During this period, instances of **both classes** are visible simultaneously;
- **Incremental concept drift**: The data instances evolve **step by step** from the previous class to a new one. The instances arriving during this transition are *transitional forms* that do not necessarily belong to either of the established classes;
- **Recurring concept drift**: This occurs when statistical characteristics change between two or more concepts **several times**. Neither concept disappears permanently, and they appear in *turns*, which is often observed in **seasonal or cyclical** data patterns like annual shopping behaviors.

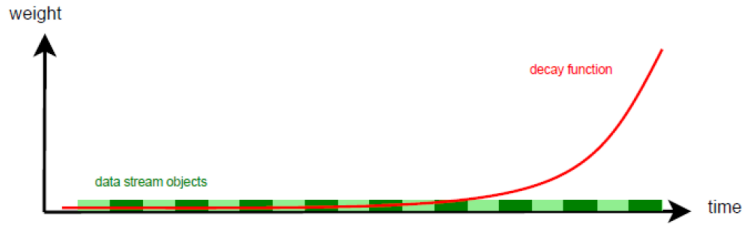
To handle these changes, **stream mining algorithms** must incorporate *adaptive mechanisms*.

8.1.3 Why should you use a Window Model for Data Streams?

A **Window Model** is a computational strategy specifically designed to handle the challenges of processing **Data Streams**, which are characterized as **potentially unbounded** and continuous sequences of data. *It is more efficient to process recent data instead of the whole data.* Because it is physically impossible to store the entire history of an infinite stream due to *bounded memory constraints*, researchers use windowing to focus the system's resources on a **finite subset** of the most recent or relevant information. This approach effectively creates a **moving synopsis** of the data, allowing algorithms to perform **real-time analytics** without needing to perform multiple scans of the overall dataset, which is a major requirement for maintaining low *per-record processing times*.

The primary reason to adopt a window model is the existence of **concept drift**, which is the *unforeseen change* in the statistical properties of the data over time. In most real-world applications, **recent data** is significantly more useful for **forecasting and decision-making** than historical data that may have been generated under different conditions. By applying a window, the system can **automatically forget** outdated patterns that are no longer representative of the current environment. This ensures that the mathematical models remain *adaptive and accurate* even when the underlying data-generating process evolves suddenly or gradually.

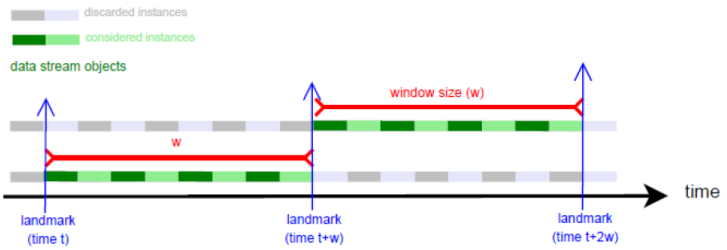
The **Damped Window Model** is one of the more sophisticated windowing techniques because it does not simply discard data but rather assumes that the **importance of instances decreases** as they age. This is typically implemented using a **decay function**, such as $f(t) = 2^{-\lambda t}$, where the parameter λ represents the *decay rate*. In this model, every instance in the stream is assigned a **weight** that is *scaled down* based on the time passed since it was received. This ensures that **recent observations** have the highest influence on the results while the impact of *historical noise* is slowly filtered out over time.



Damped window model

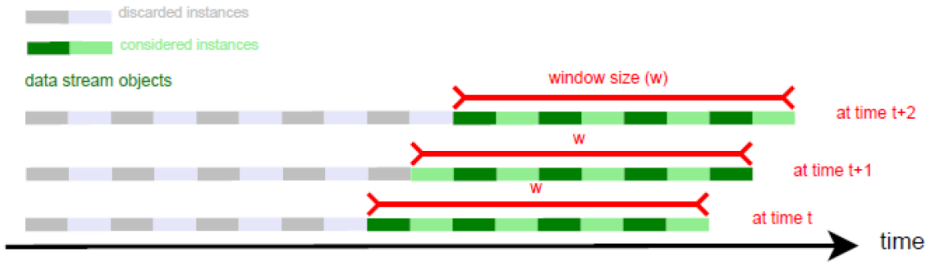
In contrast, the **Landmark Window Model** divides the continuous stream into **discrete and non-intersecting chunks** based on specific temporal markers known as **landmarks**. In this model, all data instances captured between two landmarks are treated with **equal weight**. When the current window reaches its capacity or a new landmark is triggered, the entire contents of the previous window are cleared and a **new processing cycle** begins from zero. This approach is often used when the goal is to analyze data in *logical batches*, such as daily summaries or transaction blocks where **sequential overlaps** are not a requirement.

$$W_m = [x_{m*w}, \ldots, x_{(m+1)*w-1}] \qquad m = \left\lfloor \frac{i}{w} \right\rfloor$$



Finally, the **Sliding Window Model** is the most common approach for tasks that require a **smooth and continuous update** of the data statistics. This model operates on a *First-In-First-Out (FIFO)* basis where the window maintains a **fixed length** and swaps out exactly one instance at each step. As a new instance enters the window, the **oldest instance** moves out and is discarded. Unlike the landmark model, consecutive sliding windows **mostly overlap**, and all instances currently inside the window are considered to have **equal weight**. This is particularly useful for *monitoring systems* that need to provide an **approximate answer** based on a steady, real-time snapshot of the last N observations.

$$W_m = [x_m, \ldots, x_{(m+w-1)}]$$



8.2 Clustering

8.2.1 Why use Clustering for Data Streams?

In most cases, true class labels are not available for stream instances and there is no prior knowledge about the number of classes. Therefore, Clustering, being unsupervised is one of the most suitable data mining and data analysis methods for data streams.

8.2.2 Talk about Adaptive Streaming K-Means

The **Adaptive Streaming K-Means** algorithm is a specialized *partitioning-based Clustering* technique designed to handle the **dynamic nature** of data streams where true class labels are typically unavailable. Traditional K-Means is unsuitable for streams because it requires the entire dataset and a *predefined number of clusters* (k), whereas this adaptive version can **autonomously determine** the optimal k and respond to **concept drift** in real-time. The algorithm operates in two distinct phases: an *initialization phase* (function `determineCentroids()`) to find k and determines candidate centroids and a *continuous Clustering phase* (function `changeDetected()`) to maintain the model as new data arrives. The parameter l defines the number of data instances accumulated in initialization phase.

The execution of the algorithm follows these specific **procedural steps**:

1. **Accumulate Initial Data**: The process begins by collecting a fixed sequence of data instances, denoted by length l, to serve as the basis for the first model;
2. **Estimate Probability Density**: For each feature in the data, the algorithm estimates the **Probability Density Function** (**PDF**) for each feature to understand the data distribution;
3. **Identify Potential Regions**: It detects *directional changes* in the PDF curves, with each change defining a candidate cluster region and suggesting a value for k;
4. **Generate Candidate Centroids**: The PDF curves are split into *equiprobable areas* defined by **beta points**, and the middle points between these betas are saved as **candidate initial centroids**;
5. **Select Best K**: The algorithm runs standard K-Means for different values of $k \in [k_{min}; k_{max}]$ and selects the **best Clustering configuration** by comparing their **Silhouette coefficients**;
6. **Monitor for Drift**: In the online phase, the system continuously tracks the **standard deviation and mean** of the incoming input stream;
7. **Predict and Respond**: If the algorithm detects a significant change in these statistics, it predicts a **concept drift**, declares the current centroids invalid, and triggers a complete **re-initialization** to adapt to the new data patterns.

The Adaptive K-Means Algorithm offers several **advantages**:

- It effectively handles **multi-density clusters** and is **suitable** for high-dimensional data;
- It provides **strong drift adaptation** by automatically re-centering and re-evaluating the number of clusters when the data distribution shifts;
- It does not require **expert knowledge** or prior information about the number of classes to function effectively.

However it also presents some **limitations**:

- The algorithm **lacks built-in outlier detection** capabilities, meaning noise can significantly distort the cluster centroids;
- The **re-initialization phase** is computationally expensive ($O(d \cdot l) + O(l \cdot d \cdot k \cdot cs)$), which can create temporary *bottlenecks* during high-velocity drifts;
- Performance is highly dependent on the quality of the *initial data sequence* used for PDF estimation.

The total worst-case complexity of this algorithm is defined as $O(d \cdot l) + O(l \cdot d \cdot k \cdot cs)$. This complexity is partitioned into the following phases:

- **Initial k-Estimation**: Estimating the number of clusters for a single dimension has a complexity of $O(l)$, where l is the length of the initial sequence. Because this estimation is performed for all d dimensions, the total complexity for this phase is $O(d \cdot l)$;
- **Initial Centroid Fitting**: After determining the initial centroids, running the K-Means algorithm takes $O(l \cdot d \cdot k \cdot cs)$. Here, cs represents the number of different *centroid sets* evaluated, and the efficiency is high because no further iterations of the algorithm are required at this stage;
- **Online Phase**: Assigning a single newly received data instance to the **nearest cluster** during the continuous Clustering phase is extremely efficient, with a complexity of $O(k)$.

8.2.3 Talk about MuDi-Stream

MuDi-Stream is a *hybrid stream Clustering algorithm* that combines **density-based** and **grid-based** approaches to handle evolving data streams. Its primary goal is to provide a robust mechanism for *multi-density Clustering* while simultaneously performing **outlier detection**. The algorithm maintains a *data synopsis* through a specialized structure called **core mini-clusters (cmc)**, which are feature vectors that store the weight, center, radius, and maximum distance of instances from the mean. This allows the system to summarize the stream incrementally without storing individual data points.

The execution of the MuDi-Stream algorithm is divided into an **online phase** for real-time processing and an **offline phase** for final cluster generation:

1. **Initialize Grid Structure**: The process begins by setting up a grid across all dimensions of the feature space to track data density and identify potential outliers. The feature space is divided into cells based on a parameter called `gridGranularity`;
2. **Process New Instance**: For every incoming data instance x, the algorithm performs a *linear search* to find the nearest existing **core mini-cluster** (initially cell exceeding threshold α);
3. **Update or Map**: If the instance falls within the radius of a cmc, it is added to that cluster, otherwise it is mapped to the **grid structure** to check if that region is becoming dense enough to form a new cmc;
4. **Apply Decay and Prune**: Periodically, the algorithm applies a **damped window model** to scale down the weights of older data and removes low-weighted grids and cmcs to maintain *bounded storage*: $w_{cmc}(t_{now}) = w_{cmc}(t_{last}) \cdot 2^{-\lambda(t_{now}-t_{last})}$. If you add a point add 1: $w_{cmc}(t_{now})' = w_{cmc}(t_{now}) + 1$;
5. **Select Seed for Final Clustering**: In the offline phase, an *unvisited* core mini-cluster is randomly selected to potentially start a new final cluster;
6. **Identify Neighbors**: The algorithm checks if the selected cmc has **neighbors** within a specific density threshold. If it has none, it is marked as **noise**;
7. **Expand Clusters**: If neighbors are found, a new **final cluster** is created by recursively merging the cmc with its neighbors until all reachable core mini-clusters are marked as visited.

The MuDi-Stream Algorithm offers several **advantages**:

- It excels at identifying clusters with **multiple densities** and provides **effective outlier detection** using the grid structure;
- It demonstrates **strong drift adaptation** by pruning outdated synopses through the *damped window* mechanism;
- It effectively manages the **infinite nature** of the stream by storing only the essential statistics in *core mini-clusters*.

However it also presents some **limitations**:

- It is **not suitable for high-dimensional data** because the complexity of the grid structure grows *exponentially* with the number of dimensions;
- The quality of the Clustering is highly sensitive to **input parameters** such as the density threshold, decay rate, and *grid granularity*;
- It requires **expert knowledge** to properly tune these parameters for a specific dataset, unlike more autonomous algorithms.

The overall complexity of MuDi-Stream is expressed as $O(c) + O(\log G)$. This model is primarily driven by the number of **core mini-clusters** (c) and the total number of **density grids** (G):

- **Instance Processing**: For each new data instance arriving from the stream, a **linear search** is performed on the existing core mini-clusters, resulting in a complexity of $O(c)$;
- **Grid Mapping**: Mapping an instance to the *grid structure* is very efficient, with a complexity of $O(\log \log G)$. This is achieved because the list of grids is maintained as a **tree structure**;
- **Pruning Phase**: During the pruning period, all core mini-clusters and grids are examined, making the time complexity $O(c)$ for the clusters and $O(\log G)$ for the grids;
- **Space Complexity**: The grid uses $O(\log G)$ space due to pruning, but since G grows exponentially with dimensionality, MuDi-Stream is unsuitable for high-dimensional data.

8.3 Classification

8.3.1 Why can we use Classification for Data Streams?

Classification is used in the context of **Data Streams** to assign incoming instances to *predefined categories* in real-time. Unlike traditional Classification, which assumes that all training data can be simultaneously stored in **main memory** or repeatedly read from a disk, Stream Classification must handle an **infinite sequence** of instances that arrive at a rapid rate.

To implement Classification effectively on streams, a *key observation* is utilized: it may be sufficient to consider only a **small subset** of the training examples to find the best attribute at a node. This is mathematically supported by the **Hoeffding Bound**, which determines the **minimum number of examples needed to choose a split with high confidence**. In short, **the goal is to design Decision Tree learners that read each example at most once, and use a small constant time to process it**.

The **Hoeffding Bound** is a *probabilistic tool* used to estimate the number of observations needed to ensure that a **sample mean** is close to the **true population mean** with a high level of confidence. In the context of **Data Stream Mining**, it is the mathematical foundation of the **Very Fast Decision Tree (VFDT)** algorithm, as it allows the learner to make *statistically sound* decisions about splitting nodes using only a **small subset** of the potentially infinite stream.

The bound states that if we have N independent observations of a *random variable* X that has a **range** R, and $\bar{x}$ is the observed *sample mean*, then with probability $1-\delta$, the **true mean** of the variable is at least $\bar{x}-\varepsilon$, where ε is calculated as:

$$\varepsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2N}}$$

In this formula, R represents the **range of the random variable** (such as the **logarithm of the number of classes** $\ln(K)$), N is the **number of samples** seen so far, and δ is the **confidence parameter** specified by the user.

To use this bound for **decision tree induction**, the algorithm follows these principles:

- **Evaluate Attributes**: The learner calculates a *heuristic measure* $G(\cdot)$, such as **Information Gain** or the **Gini Index**, for all available attributes based on the N examples seen at a node;
- **Compare Best Splits**: Let X_A be the attribute with the **highest evaluation** and X_B be the attribute with the **second-highest** evaluation;
- **Apply the Split Condition**: If the difference between the observed gains $\Delta\bar{G} = \bar{G}(X_A) - \bar{G}(X_B)$ is **greater than** ε, the Hoeffding Bound guarantees with probability $1-\delta$ that X_A is indeed the **superior attribute**;
- **Execute the Split**: Once the condition $\Delta\bar{G} > \varepsilon$ is met, the node is **split permanently** using X_A, and subsequent examples are passed to the new children.

When we compare the best attribute (X_A) with the second-best (X_B), we calculate their observed difference: $\Delta\overline{G} = \overline{G}(X_A) - \overline{G}(X_B)$. The Hoeffding Bound ensures that the true difference (ΔG) between the two attributes follows this relationship:

$$\Delta G \geq \Delta\overline{G} - \varepsilon$$

So, if we observe that the difference is greater than ε ($\Delta\overline{G} > \varepsilon$), then applying the above formula we necessarily obtain that: $\Delta G > 0$ and therefore X_A is better than X_B and consequently than all the others with a probability $1-\delta$.

This approach is highly **efficient** because it avoids the need to store the entire dataset. However, it is important to note that certain split measures like *Information Gain* and *Gini Index* do not strictly satisfy the assumptions of the original Hoeffding Bound because they cannot be expressed as a *sum of independent elements*. To *resolve this problem* and maintain the **theoretical guarantees** of the split, researchers utilize modified versions of the bound, such as those derived from **McDiarmid's inequality**, which generalizes the Hoeffding Bound to functions with bounded differences. The updated calculation for ε is formulated as:

$$\varepsilon = C_{Gain}(K;N)\sqrt{\frac{\ln(1/\delta)}{2N}}$$

where the constant $C_{Gain}(K; N)$ is defined by a **complexity function** that accounts for the number of classes K and the number of examples N. The specific *penalty function* for Information Gain is expressed as:

$$C_{Gain}(K; N) = 6(K \log 2(e \cdot N) + \log_2(2N)) + 2 \log_2(K)$$

By replacing the standard ε with this corrected version, the **Very Fast Decision Tree (VFDT)** can rigorously justify a split even when the heuristic measure is not a simple sum.

The primary objective of these approaches is to **minimize the disagreement** between a tree produced from a stream and a tree that would be produced using *infinite examples*. By focusing on the **most significant attributes** and using *pre-pruning* strategies (such as considering a "**null**" attribute, and if the "don't split" option is not worse than any attribute, the node remains a leaf), these models balance **model complexity** with *computational efficiency*. This makes Classification a viable and **powerful tool** for extracting insights from massive, time-changing datasets where traditional static mining would fail.

The implementation of the **Hoeffding Tree**, specifically through the **Very Fast Decision Tree** (**VFDT**) algorithm, follows a strict *single-pass* computational model where each record is examined at most once. The learning process is characterized by an incremental growth strategy:

- **Initialization**: The algorithm begins with a single leaf (the root) and initializes sufficient statistics n_{ijk} (representing counts for attribute i, value j, and class k) to zero;
- **Statistical Accumulation**: As instances x are streamed, they are filtered down to the appropriate leaf using the current tree structure. The statistics n_{ijk} are updated only at the leaf level to maintain a *bounded storage* synopsis in memory;
- **Efficiency via** n_{min}: To reduce the high computational cost of recomputing the heuristic measure $G(\cdot)$ for every single instance, VFDT introduces a user-defined parameter n_{min}. The algorithm only checks for a split after a leaf has accumulated at least n_{min} new examples since the last evaluation;
- **Growth and Memory Management**: If the Hoeffding condition $\Delta\bar{G} > \varepsilon$ is satisfied, the leaf is replaced by an internal node with new leaves. The required memory for this structure is roughly $O(leaves \times attributes \times values \times classes)$, making it essential to monitor the memory used for the synopsis.

This cycle ensures that the tree remains a *real-time* model where per-record processing time is kept low enough to match the arrival rate of the stream.

8.3.2 Talk about CVFDT

The **Concept-Adapting Very Fast Decision Tree** (**CVFDT**) is an advanced *extension* of the **VFDT** algorithm designed specifically to address the challenge of **concept drift** in non-stationary data streams. Original VFDT lacks a mechanism to update its structure once a split is finalized. **CVFDT** overcomes this by using a **sliding window approach**. It continuously monitors the **splitting attributes** at each node to detect when they are no longer optimal, allowing it to adapt the model **without needing to rebuild** the entire tree from scratch.

The **procedural logic** of the CVFDT algorithm follows these structured *evolutionary steps*:

1. **Initialize with VFDT**: The algorithm starts by building a decision tree using the standard **Hoeffding Bound** logic where instances are processed as they arrive;
2. **Maintain Sliding Window**: As new instances arrive, the **oldest examples** are removed from the internal *sufficient statistics*, ensuring that the counts at each node only reflect the current state of the stream;
3. **Monitor Splitting Quality**: The algorithm periodically evaluates whether the **attribute currently used** for a split is still the best one according to the heuristic measure (e.g., *Information Gain*);
4. **Initiate Alternative Subtrees**: If a different attribute is found to be superior to the existing one, CVFDT begins **growing an alternative subtree** in the background with the new best attribute at its root;
5. **Simultaneous Processing**: New instances are passed down through **both the original and the alternative** subtrees to accumulate statistics and grow the new structure;
6. **Compare Accuracy**: The algorithm periodically uses a *bunch of samples* to evaluate the **Classification accuracy** of the original subtree versus the alternative one;
7. **Replace Outdated Nodes**: If the alternative subtree becomes **more accurate** than the original, the old subtree is discarded and the alternative one is **promoted** to take its place.

The CVFDT offers several **advantages**:

- It provides **continuous adaptation** to time-changing concepts by maintaining multiple versions of tree segments;
- It maintains the **speed and accuracy** of VFDT while ensuring the model does not become *obsolete* as data distributions evolve;
- It is capable of responding to **all types of drift**, including sudden and gradual shifts, by replacing only the **specific parts** of the tree that are no longer valid.

However it also presents some **limitations**:

- It requires **significantly more memory** than VFDT because it must store sufficient statistics for multiple alternative subtrees simultaneously;
- The process of **growing and evaluating** alternative paths adds *computational overhead*, which can impact throughput during periods of high-speed data arrival;
- Setting the **window size** is a critical and difficult task, as a window that is too small leads to *instability*, while one that is too large delays **drift detection**.

Regarding **computational complexity**, CVFDT inherits the basic *per-example processing* efficiency of VFDT but adds a **maintenance cost**. The **complexity** per example remains small and constant, but it is multiplied by the number of **active alternative subtrees**. The **space complexity** is significantly higher, expressed as $O(leaves \times attributes \times values \times classes)$, and is further scaled by the **sliding window size** and the *number of alternative nodes* kept in memory to manage the **concept-adaptation** process.

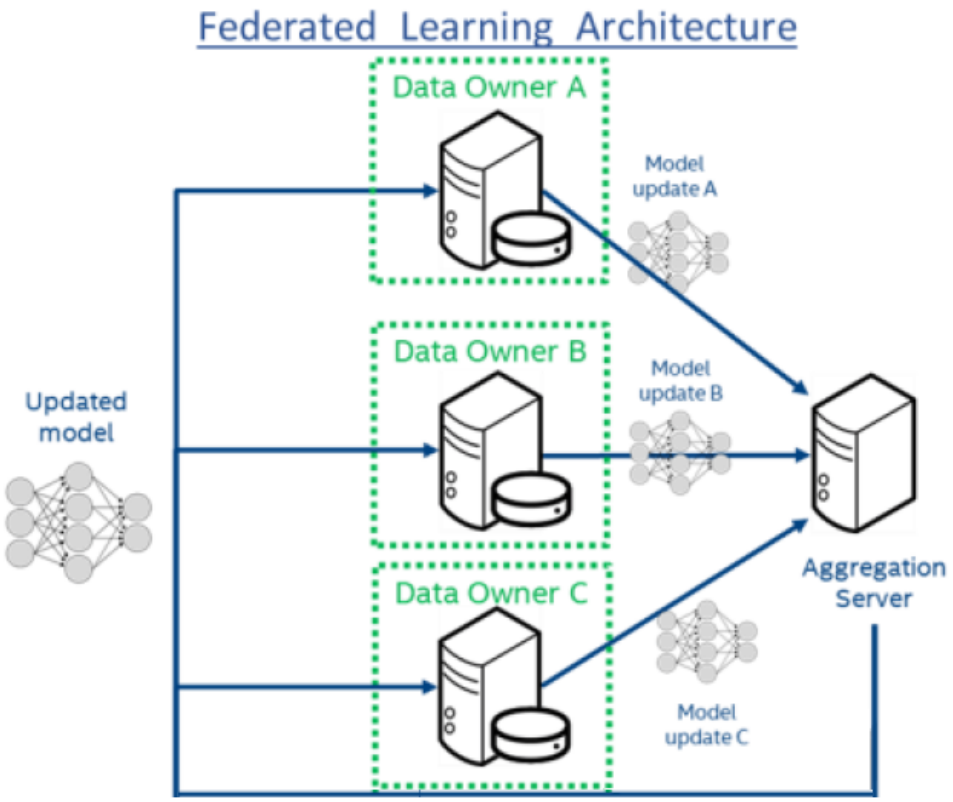
Chapter 9: Federated Learning and XAI

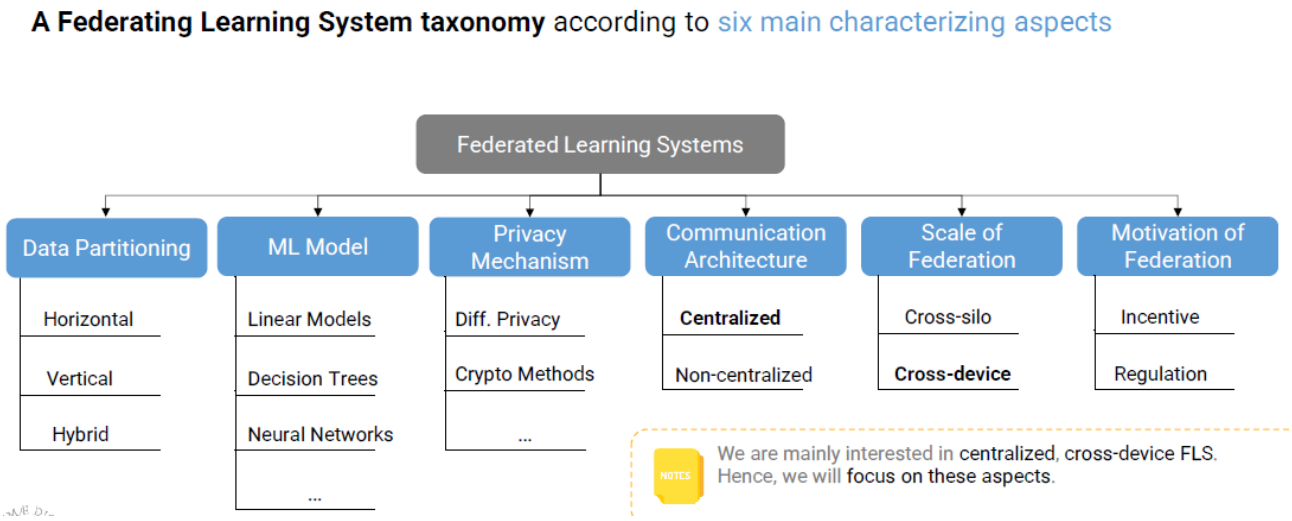
9.1 Federated Learning

9.1.1 Why do we need Federated Learning?

Federated Learning is a *decentralized* Machine Learning paradigm that allows multiple entities, known as **clients**, to collaborate in solving a Machine Learning problem without the need to exchange their *raw data*. In this setting, the data remains stored **locally** on the devices of the owners, and only *focused updates*, such as gradients or model parameters, are sent to a **central aggregation server**. This server aggregates the updates to refine a **global model**, which is then broadcasted back to the edge devices to allow them to continuously improve their individual versions of the model. This iterative process ensures that the model learns from the collective knowledge of all participants while strictly adhering to **privacy-by-design** principles.

We need Federated Learning because modern Machine Learning algorithms, particularly *Deep Learning* models, are **data hungry** and require massive amounts of information to reach high accuracy. However, in reality, data is often spread across various devices and different owners, creating **isolated data islands**. Transferring this data to a central location is frequently impossible due to *strict privacy restrictions* and regulations. Federated Learning resolves the clash between the **need to collect data** for accurate AI and the **need to preserve the privacy** of data owners. By keeping data at its source, it enables the creation of **Trustworthy AI** systems that satisfy requirements for *transparency* and *data governance*.





9.1.2 Talk about Data Partitioning in Federated Learning

Federated Learning Systems are primarily classified by how data is distributed across the *sample space* and *feature space*. This **data partitioning** determines the learning strategy and communication design. The literature commonly identifies three types: **Horizontal**, **Vertical**, and **Hybrid Federated Learning**.

Horizontal Federated Learning (*example-based*) assumes a shared **feature space** with disjoint **samples**, as in cross-device settings where clients train on similar data formats but different users. Model updates are easily aggregated since features are aligned.

Vertical Federated Learning (*feature-based*) involves a shared **sample space** but different **features**, such as multiple organizations holding complementary data about the same individuals. This setting requires advanced *privacy-preserving* techniques to coordinate learning.

Hybrid Federated Learning combines both horizontal and vertical splits, where parties partially overlap in samples and features. These scenarios demand flexible architectures and influence the choice between *centralized* and *decentralized* communication models.

9.1.3 Talk about ML Models in Federated Learning

Federated Learning has evolved from an initial focus on **Artificial Neural Networks** to supporting diverse model families suited to different data types and constraints. Commonly used models include **Neural Networks**, **Decision Trees**, and **Support Vector Machines**.

Neural Networks are the most prevalent due to their flexibility and accuracy. They are typically trained using **federated stochastic gradient descent**, where model parameters are aggregated (e.g., via averaging), and are well suited for tasks such as medical imaging and natural language processing.

Decision Trees, including **Gradient Boosting Decision Trees**, are efficient and effective for **tabular data**. Their main advantage lies in their **interpretability**, making them suitable for applications requiring transparency.

Support Vector Machines are also used in federated settings, often trained via federated optimization methods. They are particularly effective for *Classification* and *anomaly detection* in high-dimensional spaces.

Choosing an appropriate model is crucial in Federated Learning, as it involves balancing **predictive performance** with *explainability*, a key requirement for **Trustworthy AI**.

9.1.4 Talk about Privacy Mechanisms in Federated Learning

Although **Federated Learning** keeps raw data local, shared *model updates* can still leak sensitive information. To protect clients during aggregation, several **privacy mechanisms** are employed, notably **Differential Privacy**, **Secure Multi-Party Computation**, **Homomorphic Encryption**, and **Trusted Execution Environments**. Adversaries may also attack by biasing data to suit their interests.

Differential Privacy adds calibrated noise to local updates, limiting the influence of any single data point and providing formal privacy guarantees at the cost of potential accuracy loss.

Secure Multi-Party Computation enables the server to observe only aggregated updates, typically via *secret sharing*, preventing access to individual contributions.

Homomorphic Encryption allows aggregation directly on encrypted data, ensuring strong confidentiality but incurring significant computational overhead.

Trusted Execution Environments use hardware-protected enclaves to securely aggregate updates, offering efficient and trusted execution even on potentially compromised systems.

9.1.5 Talk about Communication Architecture in Federated Learning

The **communication architecture** defines how model updates are exchanged and aggregated in a **Federated Learning System**. Two main paradigms are studied: **Centralized** and **Non-centralized** architectures.

In a **Centralized Architecture**, a **central server** coordinates training by distributing the global model, collecting local updates from clients, and aggregating them iteratively until convergence. This design is simple and efficient but introduces a **single point of failure** and scalability bottlenecks.

A **Non-centralized Architecture** (*decentralized* or *peer-to-peer*) removes the central server, allowing participants to exchange updates directly with peers. Each node updates its model using both local data and received gradients. While harder to coordinate and more communication-intensive, this approach improves **robustness** and avoids single points of failure.

9.1.6 Talk about Scale of Federation in Federated Learning

Federated Learning systems can be classified by their **scale of federation**, which affects client selection and optimization strategies. The two main settings are **Cross-silo** and **Cross-device Federated Learning**.

Cross-silo Federated Learning involves a small number of reliable participants, typically organizations such as hospitals or research institutions. These clients have strong computational resources, stable connectivity, and usually participate in every training round, enabling efficient convergence on large distributed datasets.

Cross-device Federated Learning operates at massive scale, often with millions of mobile or IoT devices. Clients are unreliable and intermittently available, each holding limited local data. As a result, only a random subset of devices participates in each communication round to ensure scalability and robustness.

Beyond scale, the two settings differ in **governance**: cross-device learning emphasizes user-level privacy and communication efficiency, while cross-silo learning must satisfy strict *legal and regulatory* requirements such as GDPR or HIPAA.

9.1.7 Talk about Motivation of Federation in Federated Learning

The effectiveness of a **Federated Learning System** depends on the **motivation of participants** to join and remain active. Since parties may be both collaborators and competitors, motivations are typically grouped into **regulatory** and **incentive-based** drivers.

Regulations provide a legal motivation, as frameworks such as **GDPR** and **HIPAA** restrict centralized data collection. Federated Learning enables organizations to collaborate, overcome data silos, and build robust models while remaining compliant.

Incentive Mechanisms encourage voluntary participation by rewarding contributions. Common approaches include **reputation systems** (better service quality), **financial rewards**, **social incentives**, and **gamification**.

Balancing these motivations is essential to prevent **free-riding** and to align individual interests with the collective objective of learning an effective global model.

9.1.8 Talk about Algorithms in Federated Learning

The development of robust and efficient **optimization algorithms** is at the heart of Federated Learning, as they must handle *non-IID*[1] *data distribution*, communication constraints, and privacy requirements. These algorithms range from those designed for **Neural Networks** and **Linear Models** to more interpretable structures such as **Decision Trees** and **Unsupervised Clustering** methods.

For models based on *gradient descent*, the two most popular approaches are **Federated Stochastic Gradient Descent** (FedSGD) and **Federated Averaging** (FedAvg).

In **FedSGD**, each client computes the gradient on its local data at the current global model state, and the central server then *aggregates these gradients* to perform a single update step.

Federated Averaging (FedAvg) is a more *generalized version* that is better suited for communication-constrained environments. Its execution follows these steps:

1. The server initializes the global model weights;
2. In each round, the server selects a *random subset of clients* and sends them the current global model;
3. Each selected client performs **multiple local epochs** of gradient descent on its own data;
4. The clients return their *updated local weights* to the server;
5. The server computes a **weighted average** of these weights to produce the new global model.

FedSGD can be seen as a *special case* of FedAvg in which all clients participate, the batch size is infinite, and only one local epoch is performed.

Federated Decision Trees provide an alternative for scenarios requiring **high interpretability** and efficiency on tabular data. One prominent approach is the **ICDTA4FL** process, which focuses on aggregating local models into a *transparent global structure*. The process involves:

1. Each client trains a **local decision tree** using its private data;
2. The clients send these trees to the server, which *filters them* based on specific evaluation metrics;
3. The server **extracts the rules** from the selected trees and aggregates them using a *Cartesian product*;
4. A **Global Decision Tree** is built from these aggregated rules and distributed back to the clients.

Another technical variant is *Federated Fuzzy Regression Trees*, where **variance reduction** is used to determine the best split. In this setting, clients share only *first- and second-order sums* of their data with the server, enabling global variance computation without revealing individual samples.

Finally, **Federated Clustering** adapts unsupervised learning to decentralized environments through algorithms such as **Federated C-Means** (LLF-CM). The implementation varies depending on the **data partitioning** scheme.

In **Horizontal Federated C-Means**, different owners hold different objects with the *same features*. Clients evaluate the *membership* of their local objects to the current cluster centers and compute **summary statistics**. The server aggregates these statistics to *recompute the global centers*.

In **Vertical Federated C-Means**, different owners hold different *features for the same set of objects*. Each client generates *random projections* of the cluster centers over its local feature space. During each round, clients compute the **local squared distances** between objects and these projected centers. These distances are sent to the server, which **sums them** to determine the *global distance* and final cluster assignments. This *asymmetric communication* enables clustering across organizations while preserving feature-level privacy.

[1] Independent and Identically Distributed

9.2 Explainability

9.2.1 What is Explainability?

The research field of **Explainable Artificial Intelligence** (**XAI**) focuses on developing methods that make machine learning models understandable to humans. Although its roots can be traced back to *McCarthy* (1958), who linked explainability to the requirements of a system **able to learn**, the concept has significantly evolved over time.

Today, **explainability** refers to understanding *how a system works* and deciphering why a specific input leads to a particular prediction. The importance of explainability arises from the increasing use of AI in *life-impacting decisions*, where opacity can threaten **fundamental rights** and **societal safety**. Several domains illustrate why understanding the reasoning behind automated decisions is essential:

1. **Healthcare**: In *diagnostic systems*, XAI enables clinicians to trust and validate predictions, while in *drug discovery* it supports efficient and reliable research.
2. **Finance**: Explainability promotes **fairness** in credit scoring, allows individuals to understand adverse decisions, and assists institutions in *risk assessment*.
3. **Autonomous Vehicles**: Interpretable decision-making is critical for **public safety** and for building trust in self-driving technologies.
4. **Criminal Justice**: In applications such as *recidivism prediction*, XAI helps detect and prevent **algorithmic biases** that may lead to unfair outcomes.
5. **Ethical AI Deployment**: Transparent models enable the identification and mitigation of biases, promoting fairness and equity in various AI applications, such as hiring processes and resource allocation.

By fostering **transparency**, XAI supports *bias mitigation* and *error detection*, ensuring compliance with regulations while respecting **human agency**. Ultimately, the goal of explainability is to balance the *performance* of advanced AI models with the *interpretability* necessary to make them truly **trustworthy** in a socio-technical context.

9.2.2 Talk about XAI in Federated Learning

The integration of **Explainable Artificial Intelligence** (**XAI**) with **Federated Learning**, known as **Fed-XAI**, supports the development of **Trustworthy AI** by jointly addressing *privacy* and *transparency*. While Federated Learning enables **decentralized training** to protect sensitive data, XAI ensures that models can *explain their decisions* to users and stakeholders, in line with key European AI requirements.

Explainability in federated settings is commonly divided into two categories:

1. **Interpretable Models** (*glass-box models*), which are transparent by design. Examples include **Decision Trees** and **Rule-Based Systems**, offering simulatability, decomposability, and algorithmic transparency.
2. **Model Interpretability Techniques**, which provide **post-hoc explanations for black-box** models such as Deep Neural Networks, using methods like local explanations and feature relevance analysis.

A central challenge in Fed-XAI is the **interpretability–performance trade-off**. Complex models often achieve higher accuracy but lack transparency, while simpler models offer better explainability at the cost of predictive power. Current research aims to reduce this gap by designing federated approaches that balance **accuracy** and **explainability**.

Federated versions of interpretable models, such as **Federated Decision Trees**, also help meet regulatory requirements like the **GDPR** and its *right to an explanation*, making XAI a key component for the accountability and adoption of federated AI systems.

9.2.3 What types of boxes are there?

In **Explainable Artificial Intelligence**, models are classified by their level of *transparency* into **White Boxes**, **Grey Boxes**, and **Black Boxes**, ranging from fully interpretable to fully opaque. Choosing the right type is essential when *accountability* and *trustworthiness* are as important as predictive accuracy.

White Boxes, also called *glass boxes*, are **interpretable by design**, allowing humans to trace each step of the model's logic. Examples include linear and logistic regression, where coefficients directly show feature impact, decision trees, which follow simple "if-then" reasoning, case-based reasoning models such as k-Nearest Neighbors, which explain predictions by referencing similar historical examples. These models provide maximum transparency but may underperform on highly complex data.

Grey Boxes are partially interpretable, combining human-understandable structures with the ability to capture more complex relationships. Examples include fuzzy rule-based systems, which express rules using linguistic labels and fuzziness, and Bayesian Networks, which model probabilistic relationships between variables using directed acyclic graphs. Grey Boxes strike a compromise between interpretability and predictive power.

Black Boxes are *opaque* models, such as Deep Neural Networks and ensemble methods like Random Forests, which often achieve the highest predictive accuracy but reveal little about their internal logic. Understanding their decisions requires **post-hoc explainability techniques**. Modern AI challenges involve balancing the high performance of these opaque models with the transparency needed to ensure fairness and trust.

9.2.4 What are the Post-hoc Techniques in XAI?

Post-hoc techniques are explainability methods used to interpret the decisions of **opaque models**, or *black boxes*, after they have been fully trained. Unlike *interpretable-by-design* models, these techniques do not modify the model's architecture but provide an external layer of understanding. They are especially important for **Deep Neural Networks** and *Ensemble Models*, where the internal complexity is too high for direct human interpretation. Post-hoc methods are generally classified as **model-specific** (designed for a particular model type) or **model-agnostic** (applicable to any model), and can provide either a **global overview** of the system or a **local explanation** for individual predictions.

One widely used and mathematically grounded method is **SHAP** (Shapley Additive Explanations), based on *cooperative game theory*. Features are treated as players, and the model prediction as the payout. SHAP quantifies the impact of each feature on a model prediction by calculating the **Shapley Values** (ϕ_i), representing the *average marginal contribution* of a feature across all possible **coalitions**. The prediction for an instance x_i is expressed as an additive function of these values:

$$\hat{y}_i = f(\vec{x}_i) = \phi_0 + \sum_{j=1}^{F} \phi_j$$

where ϕ_0 is the **base value** (average prediction over a background dataset) and F is the number of features. This ensures **fair attribution** by simulating what the model would predict without access to certain features, replacing them with reference values from a background dataset.

Another popular approach is **LIME** (Local Interpretable Model-agnostic Explanations), which achieves **local interpretability** by approximating a complex model with a simpler *surrogate model* around a specific instance. While the model may be globally complex, it is often *locally linear*. LIME works by selecting an input instance, generating **perturbed samples** around it, obtaining predictions from the original black-box model, and training a *glass-box model* such as a linear regression or decision tree on these **weighted samples** to explain the *local behavior* of the complex model.

LIME achieves **local interpretability** by following a specific four-step process to mimic a black-box model m around an input instance I:

1. Select the model m and the instance I to be explained;
2. Create **perturbed samples** (synthetic data) "close" to instance I and obtain predictions from model m;
3. Train a simpler, *interpretable-by-design* **surrogate model** on these synthetic inputs, using weights based on their distance from I;
4. Use the surrogate model (a linear model or decision tree) to explain the local behavior of model m.

For **computer vision** tasks, **Saliency Maps** are commonly used. These are images where *pixel brightness* reflects the **importance** of each pixel for the model's decision. Positive values (often red) indicate pixels that contributed to the chosen class, while negative values (blue) indicate pixels that opposed it. **Counterfactual Explanations** answer the question "why not?" by identifying the **smallest change** to input features that would alter the model's output to a desired class. They are particularly useful in *financial and legal contexts*, where users need clear guidance to change an automated decision.

Chapter 10: Hadoop

10.1 Topic

10.1.1 What is MapReduce?

MapReduce is a programming model and framework developed by **Google** to process large volumes of data in a **distributed** and **fault-tolerant** manner on clusters of machines. It is based on two main phases:

1. **Map**
   - Each input data (typically a **key-value pair** $\langle k, v \rangle$) is passed to a **map function** that generates zero or more **intermediate pairs** $\langle k', v' \rangle$.
   - This phase is highly **parallelizable**, as each record can be processed independently.
2. **Reduce**
   - All intermediate pairs are **grouped by key** k' (shuffle & sort) and delivered to a **reduce function**, which aggregates or summarizes the values v' for each k', producing the **final output**.

MapReduce relies on principles from **functional programming**:

1. **Purity**: Always produces the same output given the same input (**idempotence**).
2. **Immutability**: Data structures cannot be modified by other processes, and history is preserved.
3. **Higher-order functions**: Functions that operate on other functions, taking them as arguments and/or returning them as output.

10.1.2 What is Hadoop?

Hadoop is an open-source framework developed by the Apache Software Foundation for distributed processing and storage of large volumes of data (*Big Data*). Hadoop consists of:

- **HDFS (Hadoop Distributed File System)**: A distributed file system that splits files across a cluster of machines, ensuring durability and scalability.
- **MapReduce**: A programming model that divides work into a Map (mapping) phase and a Reduce (aggregation) phase, enabling parallel processing of large datasets.
- **YARN (Yet Another Resource Negotiator)**: In Hadoop 2.0 and 3.0, a resource management system coordinates CPU, memory, and job execution across cluster resources.

10.1.3 What is the core difference between the Hadoop architecture and traditional Parallel Databases?

The fundamental distinction between the **Hadoop architecture** and *traditional Parallel Databases* lies in their design philosophy regarding hardware, data structures, and scalability. Traditional Parallel Databases, often referred to as **Massively Parallel Processing (MPP)** systems, are typically built on specialized, high-end hardware and require a strict *schema-on-write* approach where data must be structured before being stored. In contrast, Hadoop is designed as a **distributed framework** that runs on *commodity hardware* and utilizes a **schema-on-read** model, allowing it to ingest and process massive volumes of *unstructured* or *semi-structured* data.

While traditional databases focus on *ACID compliance* (Atomicity, Consistency, Isolation, Durability) for transactional integrity, Hadoop prioritizes **fault tolerance** and *horizontal scalability*. It achieves this through a

shared-nothing architecture where each node is independent, and the system assumes that hardware failure is a common occurrence rather than an exception.

The "functionality" of the Hadoop framework is governed by its two core components: the **Hadoop Distributed File System (HDFS)** and the **MapReduce** engine. The parameters that define its operation include the *replication factor* (typically set to three), the *block size* (defaulting to 64MB or 128MB to handle large files), and the **node count** within the cluster.

The functionality and operation of this architectural paradigm are described by the following steps:

1. Divide massive datasets into **fixed-size blocks** and distribute them across multiple *DataNodes* in the cluster.
2. Replicate each block across different racks using *rack-aware placement* to ensure high availability and data reliability.
3. Direct the computation to the data by moving the **MapReduce code** to the nodes where the data blocks are physically stored, minimizing network congestion.
4. Execute the *Map phase* to process local data and produce intermediate key-value pairs.
5. Perform the **Shuffle and Sort** phase to group intermediate data by key and transfer it to the appropriate *Reducer*.
6. Aggregate the results in the *Reduce phase* and write the final output back to the distributed file system.

The strengths of the Hadoop architecture include:

- **High Cost-Efficiency**: It runs on inexpensive commodity servers rather than proprietary, high-end hardware.
- **Extreme Scalability**: The system can scale to thousands of nodes and handle petabytes of data without significant performance degradation.
- **Superior Fault Tolerance**: Automatic data replication and task re-execution ensure that the system remains operational even if multiple nodes fail.
- **Data Flexibility**: It can process any type of data, including text, logs, and multimedia, without requiring a predefined relational schema.

However, the architecture also has notable weaknesses:

- **High Latency**: MapReduce is a *batch-processing system* and is not suitable for real-time applications or interactive queries.
- **Heavy I/O Overhead**: The constant writing of intermediate results to disk and the network-intensive *shuffle phase* can slow down processing.
- **Single Point of Failure**: In early versions, the **NameNode** represented a critical vulnerability where its failure could render the entire cluster inaccessible.
- **Not Suitable for Small Data**: The overhead of initializing the distributed framework makes it less efficient than a traditional database for small or medium-sized datasets.

Hadoop represents a shift from *data-centric* to **compute-centric** distribution. By allowing for *linear speedup* and **sizeup**, it provides the necessary infrastructure for *Big Data* applications where traditional parallel relational databases would be cost-prohibitive and technically limited.

10.1.4 How does HDFS work?

HDFS (**Hadoop Distributed File System**) is the distributed file system underlying the entire Hadoop framework. It is designed to run on commodity hardware (highly reliable and low cost) and is highly fault-tolerant, as failures are the norm rather than the exception.

In HDFS, files are split into large *blocks* (typically 64 MB or 128 MB), and each block is replicated across multiple nodes (usually three) to ensure reliability and availability. Blocks are distributed across different racks (groups of nodes) to withstand rack-level failures. Users can configure both the block size and the number of replicas.

Unlike single-disk file systems (where blocks are usually 512 bytes and minimum transfers are a few kilobytes), HDFS uses large blocks to minimize seek overhead on large datasets. Files can exceed the capacity of a single disk, and fixed-size blocks simplify the storage subsystem.

The architecture follows a master/slave model:

- **NameNode (master)**: Maintains the filesystem tree, metadata, the file-to-block mapping, and the locations of replicas.
- **DataNode (slave)**: Store the actual blocks and handle read/write requests as directed by the NameNode.

During read/write operations, the client always contacts the NameNode first, which indicates the DataNodes for reading or writing blocks. When writing, the NameNode follows this replica placement strategy:

1. First copy: on the client's node (or a random node if the client is outside the cluster);
2. Second copy: on a different rack than the first;
3. Third copy: on the same rack as the second, but on a different node;
4. Additional copies: on randomly selected nodes across the cluster.

To further optimize fault tolerance and performance, HDFS calculates the "distance" between nodes based on the network tree structure: nodes on the same rack have shorter distance, while nodes on different racks are farther apart. This allows the NameNode to place replicas so that data remains accessible even if an entire rack fails.

10.1.5 Why is the "Write Once Read Many" data model central to Hadoop's performance?

The **Write Once Read Many** (**WORM**) data model is a fundamental design principle of the **Hadoop Distributed File System (HDFS)** that prioritizes *high-throughput data access* over the ability to perform frequent data modifications. In traditional database systems, managing simultaneous **read and write** operations requires complex *locking mechanisms* and *concurrency control* to maintain data integrity. By enforcing a model where a file, once created and written, is immutable and cannot be changed, Hadoop eliminates the computational overhead of **data coherency** management across thousands of distributed nodes.

This model is particularly central to performance because it aligns with the *batch processing* nature of **MapReduce**. Since data does not change during the execution of a job, the framework can safely move *computation to the data* rather than moving the data to the computation. The primary parameters supporting this model include the **Block Size** (typically 128 MB), which facilitates large sequential reads, and the **Replication Factor**, which ensures that multiple static copies of the *immutable blocks* are available for parallel access.

The functionality and operational flow of the WORM model within HDFS are described by the following steps:

1. A client requests the **NameNode** to create a new file and receives a list of *DataNodes* to store the initial data blocks;
2. Data is streamed directly to the **DataNodes** and replicated across different racks to ensure high availability;
3. Once the write operation is complete, the file is **closed**, and the NameNode updates the *file system namespace* to mark the file as permanent;
4. The **MapReduce engine** identifies the location of these static blocks and schedules tasks to run locally on those specific nodes;
5. Multiple *mappers* read different blocks of the same file simultaneously without the need for **mutexes** or coordination, as the content is guaranteed not to change;
6. The system performs *parallel sequential reads* at massive scale, maximizing the aggregate **I/O bandwidth** of the entire cluster.

The strengths of the Write Once Read Many data model include:

- It enables **simplified data coherency** by removing the need for expensive distributed locks;
- The model is optimized for **high-throughput sequential access**, which is significantly faster than random access for large-scale *Big Data* analysis;
- It facilitates **efficient replication** since the system does not need to propagate updates to existing blocks;

- **Data integrity** is easier to maintain because static blocks can be reliably verified using *checksums* to detect hardware-induced corruption.

Despite its performance benefits, the model has specific weaknesses:

- It is completely **unsuitable for applications** requiring frequent updates or *real-time* random modifications to data;
- The model is not designed for **Online Transaction Processing (OLTP)**, as the cost of deleting or changing a record would require rewriting the entire file;
- It creates *processing latency* for small, interactive queries that do not fit the **batch-oriented** nature of the framework.

The WORM model is the cornerstone that allows Hadoop to achieve **linear speedup** and *scaleup*. By assuming that "a file once created, written, and closed need not be changed," Hadoop simplifies the **distributed computing** problem, allowing for the massive parallelism required to process the unstructured datasets found in *biological data mining* and other scientific fields.

10.1.6 What does the Mapper do in detail?

The **Mapper** is a function that takes a **key/value pair** as input and produces a list of new **intermediate key/value pairs**. It is used to **transform** and **filter** data. Each Mapper works independently on its assigned input. Multiple Mappers run in **parallel**, each processing a portion of the input data (the **input splits**). Mappers should ideally run on nodes that **contain their data locally** to avoid **network traffic**. Mappers are generally **stateless**.

10.1.7 What does the Reducer do in detail?

The **Reducer** is a function that takes a **key** and all associated **values** (aggregated from all Mappers) as input. These lists of values for each key are created through the **shuffle and sort** process. The Reducer **aggregates** these values to produce a **final output**, such as a sum, average, or other summary operation. Reducers are generally **stateless**.

10.1.8 What does the Combiner do in detail?

The **Combiner** is an optional function that performs **local aggregation** on the same node as the Mapper, reducing the amount of data transferred to Reducers. It can be seen as a **mini-Reducer**. A Combiner can only be used if the **reduce function** is **associative** and **commutative** (e.g., sum). If used, the Combiner must have the **same form** as the Reduce function.

10.1.9 What does the Partitioner do in detail?

The **Partitioner** is a function that determines which **Reducer** will receive each key/value pair emitted by the Mappers. It is used to **balance the load** among Reducers. By default, it uses a **hash function** on the key to distribute pairs and is applied **before the shuffle and sort** phase. The Partitioner uses only the **key** and ignores the value.

10.1.10 Describe the Shuffle and Sort Phase

The **Shuffle and Sort** process in **MapReduce** occurs in two distinct phases after the **Partitioner**, ensuring that all **intermediate key/value pairs** produced by Mappers are correctly delivered and ordered before the **reduce phase**.

During the **Shuffle** phase, once each Mapper has generated intermediate key/value pairs, these are not sent immediately to the Reducers. Instead, they are initially stored in an **in-memory buffer**. When the buffer reaches a predefined threshold (e.g., 100 MB), Hadoop triggers a **spill** process. The key/value pairs are divided into **partitions** by the Partitioner, which determines which **Reducer** will process each key. Data is then **locally sorted** by key and partition, written to a **temporary file** on disk, and accompanied by an **index** for efficient access. After the Mapper finishes, all temporary files from spills are **merged locally** into a single **intermediate output file** for that Mapper, still internally **partitioned**.

At the start of the reduce phase, each Reducer **pulls** the corresponding segments from all Mappers producing data for its partition. Segments are retrieved efficiently using the **indexing**.

During the **Sort** phase, the first operation is an **external merge**. The Reducer receives streams of pairs from each Mapper, already **sorted by key** but separate per Mapper, and merges them into a single **globally ordered sequence**. For each key, a **list of values** sharing that key is constructed, which is then processed by the **reduce function**.

10.1.11 What are some common limitations of the MapReduce model?

The **MapReduce** programming model is a core pillar of the *Hadoop ecosystem*, designed to process vast amounts of data in parallel across a distributed cluster. While it revolutionized **Big Data** processing by offering a simple abstraction for massive parallelism, it is characterized by several inherent *limitational constraints* that stem from its batch-oriented design and its reliance on disk-based storage. MapReduce operates by breaking a job into independent **Map** and **Reduce** tasks, but this rigid structure can become a bottleneck for modern, complex analytical workloads.

The strengths of the MapReduce model include:

- **Extreme Scalability**: It allows for the processing of petabytes of data by simply adding more commodity nodes to the cluster;
- **Built-in Fault Tolerance**: The framework automatically detects failed tasks and re-executes them on healthy nodes without requiring user intervention;
- **Programming Simplicity**: It hides the complexities of *network communication*, *data partitioning*, and *concurrency control* from the developer;
- **Data Locality**: By moving the **computation to the data**, it significantly reduces the need to move large datasets across the network.

Despite these advantages, the common **limitations** of MapReduce are significant:

- **High Latency**: It is strictly a *batch-processing* model and is not suitable for **real-time** or *interactive* data analysis;
- **Inefficiency for Iterative Algorithms**: Algorithms that require multiple passes over the data, such as **K-Means** or *PageRank*, suffer because MapReduce must write and read from HDFS at every iteration;
- **Heavy Disk I/O**: The requirement to write intermediate results to local disks and final results to HDFS creates a massive **performance bottleneck** compared to in-memory systems;
- **Rigid Structure**: The two-stage *Map-then-Reduce* paradigm is often too restrictive for complex workflows that require more flexible **Directed Acyclic Graph (DAG)** execution;
- **Small File Problem**: Hadoop is optimized for large files, and MapReduce performs poorly when dealing with a high volume of *small files* due to the overhead of the **NameNode** and task initialization.

These limitations led to the development of more advanced frameworks like **Apache Spark**, which addresses the *iterative inefficiency* and *latency issues* by keeping data in **RAM** whenever possible. Nonetheless, MapReduce remains a fundamental tool for extremely large-scale, one-off *data transformation* tasks where robustness and disk-based capacity are more critical than raw execution speed.

10.1.12 How is the K-Means algorithm parallelized using MapReduce?

The **K-Means algorithm** is one of the most popular iterative *Clustering techniques*, and its parallelization within the **MapReduce** framework is a classic example of how to handle large-scale *unsupervised learning*. Since the standard algorithm requires calculating the distance between every data point and every cluster center, it becomes computationally expensive as the dataset grows. By utilizing the **distributed computing** power of Hadoop, the workload of distance calculation and cluster assignment can be spread across hundreds of *DataNodes*, allowing the algorithm to scale to massive datasets that would not fit in the memory of a single machine.

The primary parameters for the parallelized K-Means include the **number of clusters** (K), the *distance metric* (usually Euclidean distance), the **maximum number of iterations**, and the *convergence threshold* (ϵ), which determines when the centroids have shifted little enough to stop the process. Because MapReduce is inherently *stateless*, a driver program is typically used to manage the **iterative loop**, checking for convergence after each job and redistributing the updated *centroids* for the next pass.

The functionality and operational steps of the Parallel K-Means algorithm are as follows:

1. Initialize K **cluster centroids** and store them in a file accessible to all nodes, typically using the *Distributed Cache* for efficiency;
2. Start the **Map Phase**, where each mapper reads a portion of the data points and calculates the distance from each point to all K centroids;
3. Assign each point to the **nearest centroid** and emit a key-value pair where the *key* is the Centroid ID and the *value* is the coordinates of the data point;
4. Utilize an optional **Combiner** to partially aggregate the points belonging to the same centroid on the local node to reduce network traffic during the shuffle;
5. Execute the **Reduce Phase**, where each reducer receives a Centroid ID and the list of all points assigned to it across the entire cluster;
6. Calculate the **new centroid** by computing the arithmetic mean of all assigned points and write the updated coordinates back to *HDFS*;
7. The driver program compares the new centroids with the old ones to determine if the **convergence criteria** are met or if another iteration is required.

The strengths of parallelizing K-Means with MapReduce include:

- It provides **high scalability**, enabling the processing of billions of points and high-dimensional *feature vectors*;
- The framework offers **automatic fault tolerance**, as the failure of a single mapper or reducer does not require restarting the entire Clustering iteration;
- It leverages **data locality**, ensuring that the heavy distance calculations are performed on the nodes where the data blocks are physically stored;
- The architecture allows for the *easy integration* of different distance metrics and data formats without changing the core parallel logic.

The weaknesses of this approach are primarily related to the MapReduce execution model:

- The algorithm suffers from **high I/O overhead** because the centroids and data must be read from and written to *HDFS* at every single iteration;
- There is significant **network latency** during the shuffle phase, especially when the number of clusters K or the *dimensionality* of the data is very high;
- The *startup time* for a MapReduce job can be significant, making small iterations feel sluggish compared to in-memory systems;
- It is less efficient than **Apache Spark** for this specific task, as Spark can keep the data in *RAM* between iterations instead of hitting the disk.

The key to performance in Parallel K-Means is minimizing the amount of data transferred over the **network**. By using a *Combiner* and effectively managing the **broadcast** of centroid coordinates, Hadoop can turn a previously *intractable Clustering problem* into a routine batch job that takes advantage of the aggregate bandwidth of the entire cluster.

10.1.13 What are the specific challenges of Parallel FP-Growth on Hadoop?

The **Parallel FP-Growth (PFP)** algorithm is a distributed adaptation of the frequent pattern mining method designed to operate within the *MapReduce framework*. While the original **FP-Growth** algorithm is highly efficient on a single machine because it avoids candidate generation, it faces severe *scalability bottlenecks* when applied to Big Data. The primary challenge is that the algorithm is inherently **recursive** and relies on a global **FP-Tree** structure that must reside in *physical memory*. When a dataset is massive, this tree grows too large for a single commodity node, and the sequential nature of the *prefix-path exploration* makes simple distribution across a cluster difficult.

The algorithm addresses these issues by using a **divide-and-conquer** strategy that partitions the work through *data sharding*. The primary parameters involved are the **minimum support threshold** (min_sup), which filters infrequent items, and the **number of groups** (G), which determines how the frequent items are partitioned across the cluster to manage memory and parallelize the mining task.

The functionality and operational steps of the Parallel FP-Growth algorithm on Hadoop are as follows:

1. Scan the database in parallel to count the frequency of each item and identify the set of **frequent 1-itemsets**;
2. Sort the frequent items in descending order of frequency and divide them into G **independent groups** to ensure that each group can be processed separately;
3. Execute a **Map phase** where each transaction is analyzed to determine which item groups it belongs to based on the items it contains;
4. Output *group-dependent transactions* from the mappers, effectively creating a **projected database** for each group;
5. Run a **Reduce phase** where each reducer receives all transactions related to a specific group and builds a local, independent **FP-Tree** for that group;
6. Perform local *FP-Growth mining* within each reducer to discover frequent patterns and then aggregate these results into a final global set.

The strengths of the Parallel FP-Growth algorithm include:

- It enables **massive scalability** by allowing the FP-Tree construction to be distributed across many nodes;
- It maintains the **efficiency** of the original FP-Growth by avoiding the generation and testing of millions of candidate sets;
- The use of *projected databases* reduces the amount of data each node must process in its local memory;
- It effectively leverages the **parallel bandwidth** of the Hadoop cluster to handle datasets that are petabytes in size.

However, the algorithm also presents significant weaknesses:

- It can suffer from **severe load imbalance** because some items are much more frequent than others, causing some reducers to take much longer to finish;
- The **memory overhead** remains a risk if a single group's projected database and resulting tree still exceed the RAM of the assigned reducer;
- There is a high *communication cost* associated with the shuffling of transactions from mappers to reducers;
- The complexity of managing **redundant information** across groups can lead to increased processing time compared to non-parallel methods for smaller datasets.

PFP is a fundamental advancement because it transforms a memory-intensive recursive problem into a series of **parallel tasks**. By sharding the data and focusing on *local tree construction*, Hadoop can discover deep regularities in **unstructured data** that would be impossible to uncover using a single-machine approach.

10.1.14 What is Mahout?

Apache Mahout is an open-source framework from the *Apache Software Foundation* designed to create scalable Machine Learning (ML) algorithms.

While many ML libraries are built to run on a single machine, Mahout was specifically built to handle "Big Data" by running on top of distributed computing systems like Apache Spark, and Apache Hadoop.

Traditionally, Mahout has focused on three primary areas of machine learning:

- **Collaborative Filtering (Recommendation Engines):** This is Mahout's most famous use case. It analyzes user behavior (like what you bought or watched) to suggest new items. Think of the "Users who bought this also bought..." feature on Amazon;
- **Clustering:** This involves taking a large set of data and grouping similar items together without being told what the groups are. For example, grouping news articles by topic or segmenting customers by purchasing habits;
- **Classification:** This is supervised learning where the system learns from existing "labeled" data to categorize new data. A common example is a spam filter that learns to distinguish "spam" from "ham".

GOOD LUCK ON THE EXAM. YOU'LL NEED IT.