. **Machine Learning**

S.N.	Content
1	Introduction, Supervised learning algorithm: types of learning, application, Supervised learning: Linear Regression Model, Naive Bayes classifier Decision Tree, K nearest neighbor, Logistic Regression, Support Vector Machine, Random forest algorithm.
2	Unsupervised learning algorithm: Grouping unlabelled items using k-means clustering, Hierarchical Clustering, Probabilistic clustering, Association rule mining, Gaussian mixture model.
3	Feature extraction - Principal component analysis, Singular value decomposition. Feature selection – feature ranking and subset selection, introduction to deep learning, filter, wrapper and embedded methods, evaluating Machine Learning algorithms.
4	Semi supervised learning, Reinforcement learning: Markov decision process (MDP), Bellman equations, policy evaluation using Monte Carlo, Policy iteration and Value iteration, Q-Learning, State-Action-Reward-State-Action (SARSA), Model-based Reinforcement Learning.
5	Recommended system, Collaborative filtering, Content-based filtering Artificial neural network, Perceptron, Multilayer network, Backpropagation, Introduction to Deep learning

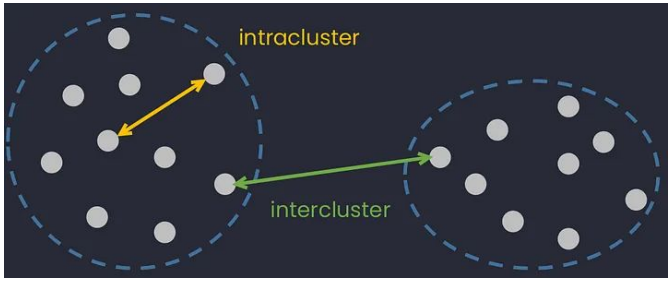
Unsupervised learning algorithm

- Unsupervised learning analyzes unlabeled data without predefined outputs.
- The algorithm discovers hidden patterns, structures, or relationships in data.
- Common tasks include clustering, association, and density estimation.
- Widely used in customer segmentation, anomaly detection, and recommendation systems.

- Unsupervised learning helps discover hidden patterns in unlabeled data.
- K-Means is simple and efficient for large datasets.
- Hierarchical clustering provides visual cluster relationships.
- GMM and probabilistic models allow soft clustering.
- Association rules reveal relationships in transactional data.

Supervised Learning	Unsupervised Learning
Uses **labeled data** for training.	Uses **unlabeled data** for training.
Model receives **direct feedback** during training.	Model **does not receive feedback**.
Predicts **output values or categories**.	Finds **hidden patterns or structures** in data.
Input and output data are both provided.	Only **input data** is provided.
Goal is to **predict outputs for new data**.	Goal is to **discover hidden insights** from data.
Requires **supervision (training with known answers)**.	**No supervision required**.
Problems include **Classification and Regression**.	Problems include **Clustering and Association**.
Used when **input and corresponding outputs are known**.	Used when **only input data is available**.
Generally provides **more accurate results**.	May produce **less accurate results** compared to supervised learning.
Not considered close to **true AI learning** because it relies on labeled data.	Closer to **human-like learning** by discovering patterns from experience.
Examples: **Linear Regression, Logistic Regression, SVM, Decision Tree**.	Examples: **K-Means, Apriori, Hierarchical Clustering**.

Clustering



- Clustering is a technique to group similar data points together
- Data points in the same cluster are more similar to each other
- Data points in different clusters are less similar
- K-Means is one of the most popular clustering algorithms

Example:

Customer segmentation, Document classification, Image compression

K-Means Clustering

Goal: Minimize intra-cluster distance

Clustering is a process of categorizing set of objects into groups called clusters.

- Unsupervised Machine Learning Algorithm
- Used for grouping similar data points
- Divides data into K clusters
- Each cluster has a centroid (center point)
- Widely used in data mining, pattern recognition, and image segmentation
- K represents the number of clusters
- The algorithm partitions N data points into K clusters
- Each cluster is represented by its centroid (mean)
- The algorithm minimizes the distance between data points and cluster centers

K-Means Clustering

Divides dataset into K clusters based on similarity.

Steps:

1. Randomly initialize K centroids
2. Assign data points to nearest centroid
3. Recalculate centroids
4. Repeat until convergence

Advantages: Simple, fast, scalable.

Limitation: Must predefine K and assumes spherical clusters.

- **Step 1: Choose K = number of clusters.**
- **Step 2: Place K initial centroids (randomly from data).**
- **Step 3: Compute distance of each point to every centroid.**
- **Step 4: Assign each point to the nearest centroid.**
- **Step 5: Recalculate centroids (mean of cluster points)**
- **Step 6: Repeat Steps 3–5 until centroids converge.**

Hierarchical Clustering

Builds a tree-like structure (dendrogram) of clusters.

Two approaches:

1. Agglomerative (bottom-up) – merge clusters.
2. Divisive (top-down) – split clusters.

- Uses linkage methods: single, complete, average.

- Advantage: No need to specify number of clusters initially.
- Limitation: Computationally expensive for large datasets.

Hierarchical clustering is a method of cluster analysis which is used to build hierarchy of clusters.

Gaussian Mixture Model (GMM)

- Assumes data comes from a mixture of Gaussian distributions.
- Uses Expectation-Maximization (EM) algorithm to estimate parameters.
- Performs soft clustering (probabilistic assignment).
- Works well when clusters overlap.
- More flexible than K-Means but computationally heavier.

Probabilistic Clustering

- Assigns probability of belonging to each cluster.
- Handles uncertainty and overlapping data.
- Common models: GMM, Bayesian clustering.
- Produces soft cluster memberships instead of hard assignments.

Association Rule Mining

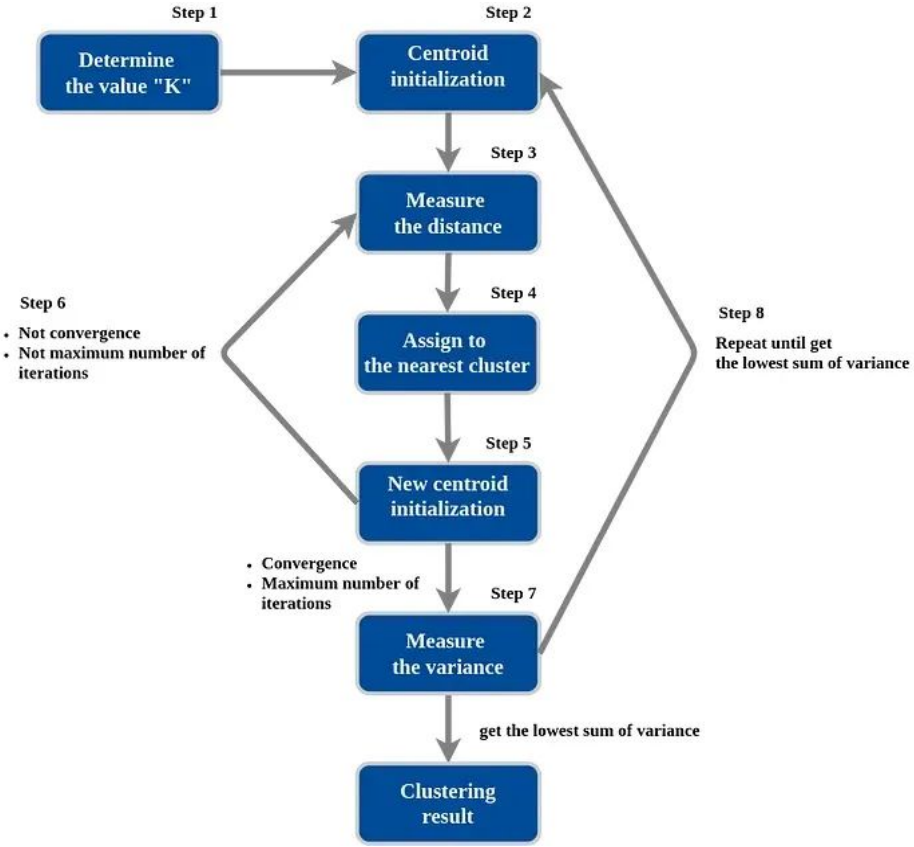
- Discovers relationships between variables in large datasets.
- Generates IF–THEN rules from transaction data.

Key metrics: Support – frequency of rule, Confidence – reliability, Lift – strength of association

- Algorithms: Apriori, FP-Growth.
- Used in market basket analysis.

Aspect	K-Means	Hierarchical	GMM	Association Rules
Assignment	Hard	Hard	Soft	Rule-based
Scalability	High (O(n))	Low (O(n²))	Medium	High
Predefine K?	Yes	No	Yes	No
Output	Centroids	Dendrogram	Probabilities	Rules

K-mean Algorithm



Algorithm 1: Standard K-means clustering algorithm pseudocode

Input: Array $X\{x_1, x_2, \cdots, x_n\}$ // Dataset to be clustered
Output: k// Number of required clusters
$C\{c_1, c_2, \cdots, c_k\}$// Cluster centroids

1. A set of k clusters
2. // Initialize Parameters
3. $X = \{x_1, x_2, \cdots, x_n\}$
4. $C = \{c_1, c_2, \cdots, c_k\}$
5. Repeat
6. //Distance calculations
7. *fori* = 1 to *n do*
8. *forj* = 1 to *k* do
9. Compute the Euclidean distance from a data object to all cluster
10. end *j*
11. //Data object assignment
12. Add data objects to the closest cluster
13. end *i*
14. //Update cluster centroid
15 Compute the new cluster centroid
16: Until the difference between the cluster centroids of two consecutive iterations remains the same
End

K-Means Clustering Algorithm

1. Cluster Formation

   * The K-Means algorithm divides the dataset into **k clusters** based on similarity between data points.

2. Similarity Measure

   * Similarity between data points and cluster centers is measured using **Euclidean distance**.

3. Initialization

   * Select **k initial centroids randomly** from the dataset.

4. Assignment Step

   * Each data point is assigned to the **nearest centroid**, forming clusters.

5. Update Step

   * The centroid of each cluster is **recalculated** by taking the **average (mean) of all data points** in that cluster.

6. Iteration

   * The **assignment and update steps are repeated** iteratively.

7. Stopping Condition

   * The process stops when:
   * Centroids **no longer change**, or
   * A **maximum number of iterations** is reached.

8. Final Result

   * The algorithm produces **k stable clusters**, where each data point belongs to the cluster with the **nearest centroid**.

1 Dataset:

$$X = \{x_1, x_2, x_3, ..., x_N\}$$

Where:

- N = number of data points
- x_n = data vector

Cluster centers:

$$\mu_1, \mu_2, ..., \mu_K$$

Objective: Assign each point to the **nearest centroid**.

3

Each data point is assigned to the **closest centroid**.

$$r_{nk} = \begin{cases} 1 & \text{if } k = \arg\min ||x_n - \mu_k||^2 \\ 0 & \text{otherwise} \end{cases}$$

This step forms **clusters**.

2 K-Means minimizes the **sum of squared distances**.

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||x_n - \mu_k||^2$$

Where:

- r_{nk} = cluster assignment variable
- x_n = data point
- μ_k = cluster centroid

Goal: **Minimize J**

4 Recalculate the centroid:

$$\mu_k = \frac{\sum r_{nk} x_n}{\sum r_{nk}}$$

Meaning:

Centroid = **Average of all points in that cluster**

The K-means clustering algorithm is categorized as a partitional clustering algorithm. Partitioning given datasets into clusters involves finding the minimum squared error between the various data points in the data set and the mean of a cluster, then assigning each data point to the cluster centre nearest to it. Mathematically, given a dataset $X = \{x_i\} where i = 1, 2, \ldots n$ of d-dimension data points of size n,

X Is partitioned into 'k' clusters $C = \{c_j\} where j = 1, 2 \ldots k$

such that

$$J(c_k) = \sum_{x_i \epsilon c_k} ||x_i - \mu_k||^2$$

The K-means algorithm aims to minimize the sum of the square error for each k cluster. That is,

Minimize

$$J(C) = \sum_{k=1}^{K} \sum_{x_i \epsilon c_k} ||x_i - \mu_k||^2$$

Initially, the K-means algorithm randomly chooses a specified k number of centroids from the dataset. Each data point's distance from all the selected centroids is evaluated, with each assigned to the closest centroid as a member of that centroid's cluster. On the assignment of a new member to a cluster, the center of a cluster is re-evaluated. This K-means algorithmic process is iteratively performed until the cluster membership is stable.

The basic steps in K-means algorithm are as follows:

- Initial partition selection with 'k' clusters
- Generate a new partition by assigning each pattern to the nearest cluster center
- New cluster center computation
- Repetition of (2) and (3) until there is stability in the cluster membership

K-Means Clustering

Person	Height	Weight
Person 1	167	55
Person 2	120	32
Person 3	113	33
Person 4	175	76
Person 5	108	25

Goal: Group the data into K = 2 clusters using K-Means.

Step 1: Initialize Centroids

- Choose **two initial centroids randomly** from the data points.
- Suppose we pick:
  - $c_1 = \text{Person } 2 \rightarrow (120, 32)$
  - $c_2 = \text{Person } 3 \rightarrow (113, 33)$
- These become the starting centers of **Cluster 1** and **Cluster 2**.

Step 2: Calculate Euclidean Distance

- Use the Euclidean distance formula:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

- Distances of each person to the centroids:

Person	Distance from c_1	Distance from c_2
Person 1	52.3	58.3
Person 2	0	7.1
Person 3	7.1	0
Person 4	70.4	75.4
Person 5	13.9	9.4

Step 3: Assign Points to Nearest Cluster

- Assign each person to the cluster whose centroid is closest (minimum distance).
- Assignments:
  - Person 1 → Cluster 1
  - Person 2 → Cluster 1
  - Person 3 → Cluster 2
  - Person 4 → Cluster 1
  - Person 5 → Cluster 2
- Current clusters:
  - Cluster 1: Person 1, Person 2, Person 4
  - Cluster 2: Person 3, Person 5

Step 4: Recalculate Centroids

- Compute the **mean** of all points in each cluster.
- New centroid for **Cluster 1**:

$$c_1' = \left(\frac{167 + 120 + 175}{3}, \frac{55 + 32 + 76}{3}\right)$$

$$c_1' = (154,\ 54.3)$$

- New centroid for **Cluster 2**:

$$c_2' = \left(\frac{113 + 108}{2}, \frac{33 + 25}{2}\right)$$

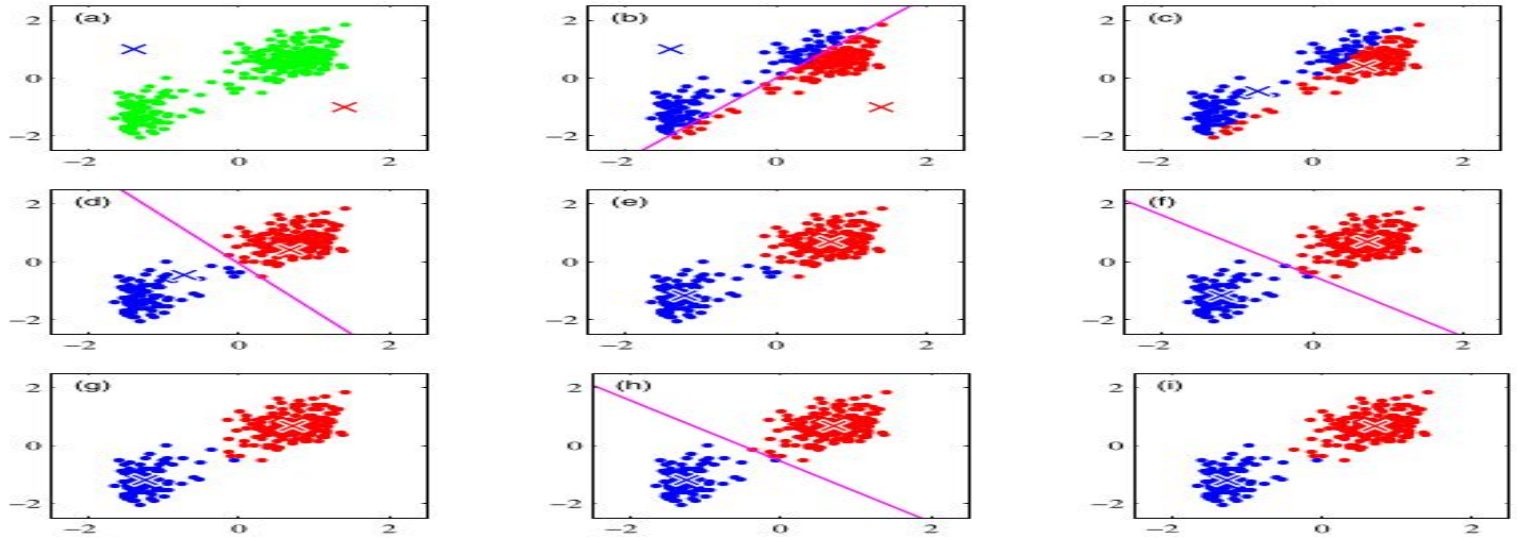
$$c_2' = (110.5,\ 29)$$

Step 5: Repeat Until Convergence

- Recalculate distances of all points to the new centroids.
- Reassign points to the nearest cluster.
- Recalculate centroids again.
- Continue this loop until the centroids stop changing (or change within a small tolerance).

Step 6: Final Result

- Points are divided into two final clusters:
  - Cluster 1: Person 1, Person 2, Person 4
  - Cluster 2: Person 3, Person 5
- Each point in a cluster is closer to its own centroid than to the other cluster's centroid.
- Centroids are often shown as cross (X) markers in cluster-plot diagrams.

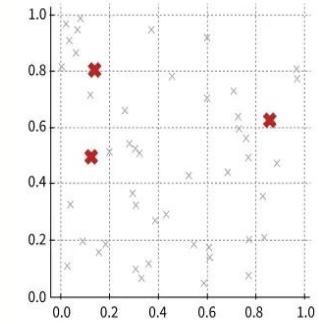


The figure illustrates the **working process of the K-Means clustering algorithm** using the Old Faithful dataset in a two-dimensional space. In **Figure (a)**, the dataset is shown with all data points represented in green, and two initial cluster centers (centroids) are randomly selected, represented by a red cross and a blue cross. In **Figure (b)**, the first **assignment step** is performed, where each data point is assigned to the nearest centroid based on the Euclidean distance. As a result, the data points are divided into two groups (red and blue). The magenta line represents the **decision boundary**, which is the perpendicular bisector between the two centroids. In **Figure (c)**, the **update step** occurs, where new centroids are calculated by taking the mean of the data points belonging to each cluster, causing the centroid positions to move. In **Figure (d)**, the assignment step is repeated using the updated centroids, and some data points may change their cluster membership depending on their distance from the new centers. In **Figure (e)**, the centroids are again recalculated as the mean of their assigned cluster points, which moves them closer to the center of their respective clusters. In **Figures (f) and (g)**, the algorithm continues repeating the assignment and update steps, gradually refining the cluster boundaries and centroid locations. In **Figure (h)**, the clusters become more stable, and very few points change their cluster assignments. Finally, in **Figure (i)**, the algorithm reaches **convergence**, where the centroids no longer change significantly and the cluster assignments remain stable. At this stage, the K-Means algorithm successfully partitions the dataset into two clusters by minimizing the **sum of squared distances between data points and their respective cluster centers**.

Choose Initial Centroids

Centroids are randomly chosen from the data points. These represent the initial cluster centers.

Centroids Data Points

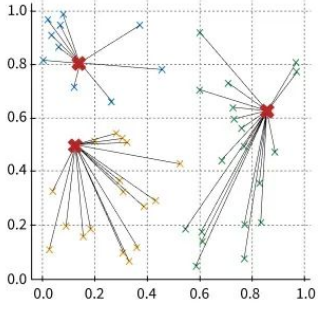


1

Assign Points to Nearest Centroid

Each point is assigned to the nearest centroid, forming clusters

Centroids Cluster 1
Cluster 2 Cluster 3

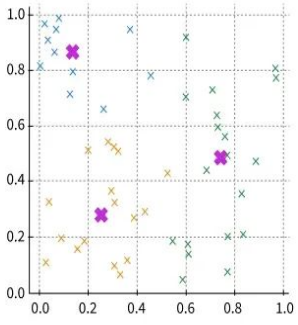


2

Update Centroids

Centroids are recalculated as the mean of the points in each cluster

New Centroids
Cluster 1
Cluster 2
Cluster 3

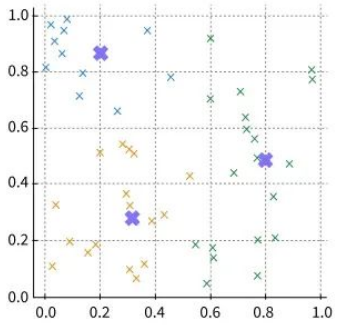


3

Repeat Until Convergence

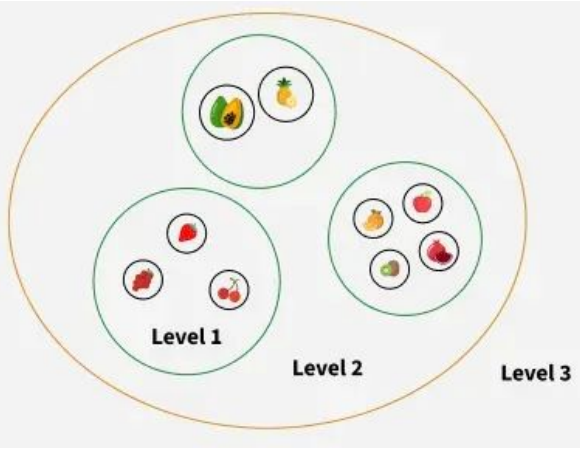
This process repeats until the centroids stabilize and do not move further.

Final Centroids
Cluster 1
Cluster 2
Cluster 3



4

Hierarchical Clustering



- Unsupervised Machine Learning technique
- Groups data into hierarchy of clusters based on similarity
- Produces a tree-like structure called a dendrogram
- Helps visualize relationships among data points
- Hierarchical clustering groups similar data points step by step
- Builds clusters in a hierarchical structure
- Does not require pre-selecting the number of clusters
- Used for data exploration and pattern discovery

There are two main types:

- Agglomerative Clustering (Bottom-Up Approach)
- Divisive Clustering (Top-Down Approach)

Example with fruits based on weight: Apple = 100 g, Banana = 120 g, Cherry = 50 g, Grape = 30 g
Clustering process:

A. Start with each fruit as its own cluster
B. Grape (30g) and Cherry (50g) merge first
C. Apple (100g) and Banana (120g) merge next
D. Finally, both clusters merge into one large cluster

- **Hierarchical clustering** is a method that groups data into nested clusters.
- The clusters are represented in a tree structure called a dendrogram.
- It converts a proximity (distance) matrix into a sequence of nested clusters.

2. Dataset Representation

- Let **X** be a set containing **n objects**:

$$X = \{x_1, x_2, x_3, \ldots, x_n\}$$

- Where:
  - x_n represents the **nth object** in the dataset.

---

3. Partition of Dataset

- The dataset **X** is divided into subsets:

$$A = \{a_1, a_2, a_3, \ldots, a_m\}$$

- Each subset represents a **cluster**.

---

4. Conditions for Partition

- Clusters **do not overlap**:

$$a_i \cap a_j = \varnothing \quad \text{for} \quad i \neq j$$

- The **union of all clusters forms the original dataset**:

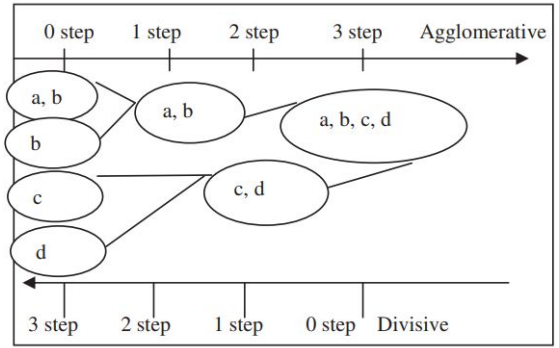
$$a_1 \cup a_2 \cup \cdots \cup a_m = X$$

5. Symbol Meaning

- ∩ → **Intersection** (common elements)
- ∪ → **Union** (combining sets)
- ∅ → **Empty set**

6. Cluster Formation

- After partitioning the dataset into subsets,
- Each **subset represents a cluster** of similar data points.



Application of agglomerative and divisive to a data set

Key Points on Dissimilarity in Hierarchical Clustering

- Dissimilarity measures the difference between sets of objects or data points in hierarchical clustering.
- In agglomerative clustering, clusters are merged based on these measures; in divisive clustering, clusters are split using them.

Role of Distance Measures

- Decisions on merging or splitting rely on hierarchical dissimilarity between clusters of observations.
- Common examples includ Euclidean distance, calculated as $\|a - b\| = \sqrt{(a-b)^2}$.
-

Distance vs. Dissimilarity

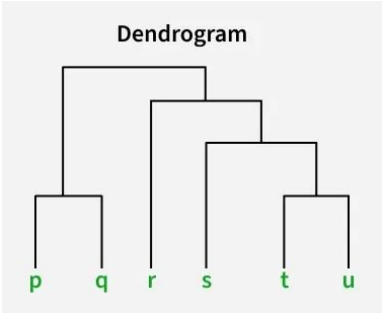
- Until a meaningful distance or proximity between pairs of objects is established, it remains uninformative.
- Greater distance indicates higher dissimilarity between objects.

Threshold-Based Clustering

- Dissimilarity is measured based on a provided distance threshold.
- More focus is placed on distance itself, as it directly quantifies object dissimilarity.

Dendrogram

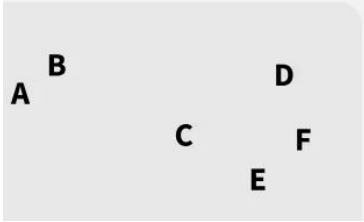
- A dendrogram is a tree diagram used to represent clusters
- Shows how data points merge step by step
- Bottom of tree → individual data points
- Upper levels → merged clusters

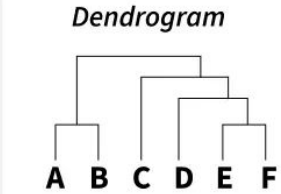


Important points:

- Lower merge height → more similarity
- Higher merge height → less similarity

We **cut** the dendrogram at a certain height to form final clusters.

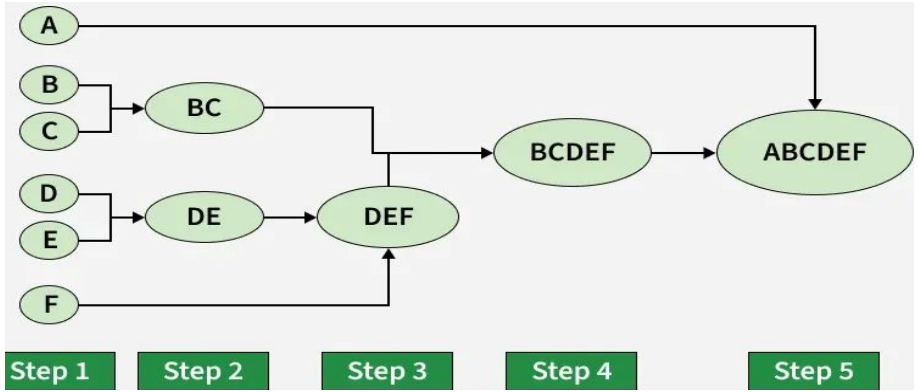




Agglomerative Clustering: Hierarchical Agglomerative Clustering (HAC).

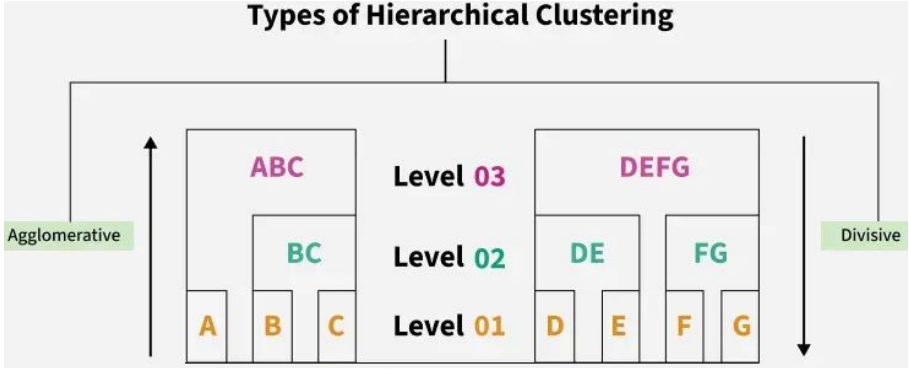
Process:

- Start with each data point as its own cluster
- Merge the closest clusters step by step
- Continue merging until all points form one cluster



Steps of Agglomerative Clustering

- Start with individual data points as clusters
- Calculate distance between clusters
- Merge the closest clusters
- Update distance matrix
- Repeat merging until one cluster remains
- Represent results using dendrogram

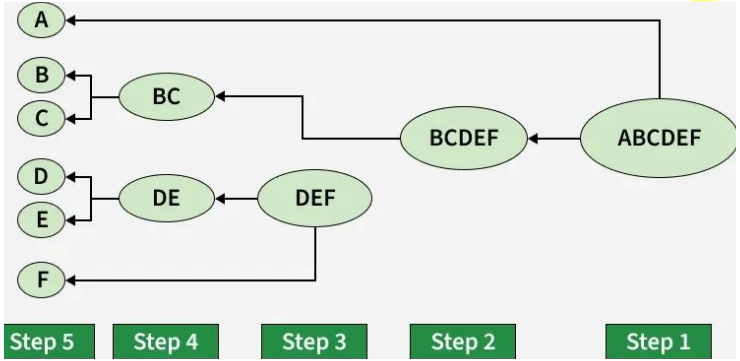


Divisive Clustering: Top-Down Clustering.

- Start with all data points in one cluster
- Divide the cluster into smaller clusters
- Continue splitting clusters recursively
- Stop when each point becomes a separate cluster or desired clusters are obtained

Steps of Divisive Clustering

- Start with one cluster containing all data
- Find the most dissimilar points
- Split the cluster into two smaller clusters
- Repeat splitting for new clusters
- Continue until desired clusters are formed



- ***Agglomerative clustering*** is a **bottom-up hierarchical clustering method**.
- Each data point or document initially forms its **own individual cluster**.
- Clusters are gradually **merged together based on similarity**.
- Start with **n clusters for n data points**.
- At each step, the **two most similar clusters are merged**.
- The process continues until **only one cluster remains** containing all data.

This method is called **Hierarchical Agglomerative Clustering (HAC)** because:

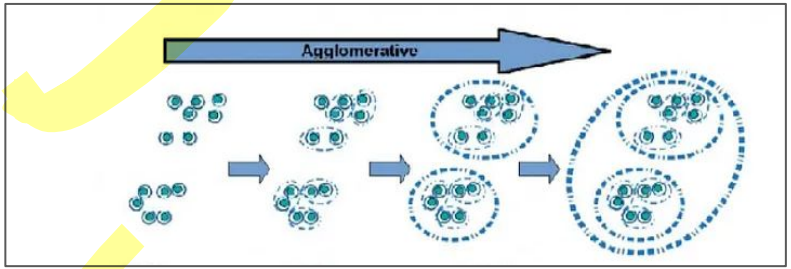
- It builds a **hierarchical structure** of clusters.
- Clusters are **agglomerated (merged) step by step**.
- The merging process is represented using a **dendrogram (tree diagram)**.
- It shows **how clusters are combined step by step**.

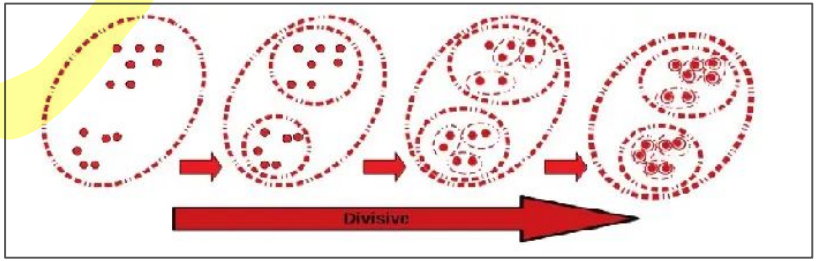
Step 1: Treat **each document/data point as a separate cluster**.
Step 2: Compute the **distance (similarity) between all clusters**.
Step 3: **Merge the two closest clusters**.
Step 4: **Update the distance matrix**.
Step 5: Repeat until **all points form one cluster**.

- The general time complexity of agglomerative hierarchical clustering : **O(n^3)**

Where:

- **n = number of data points**
- Due to this high complexity, it becomes **slow for large datasets**.



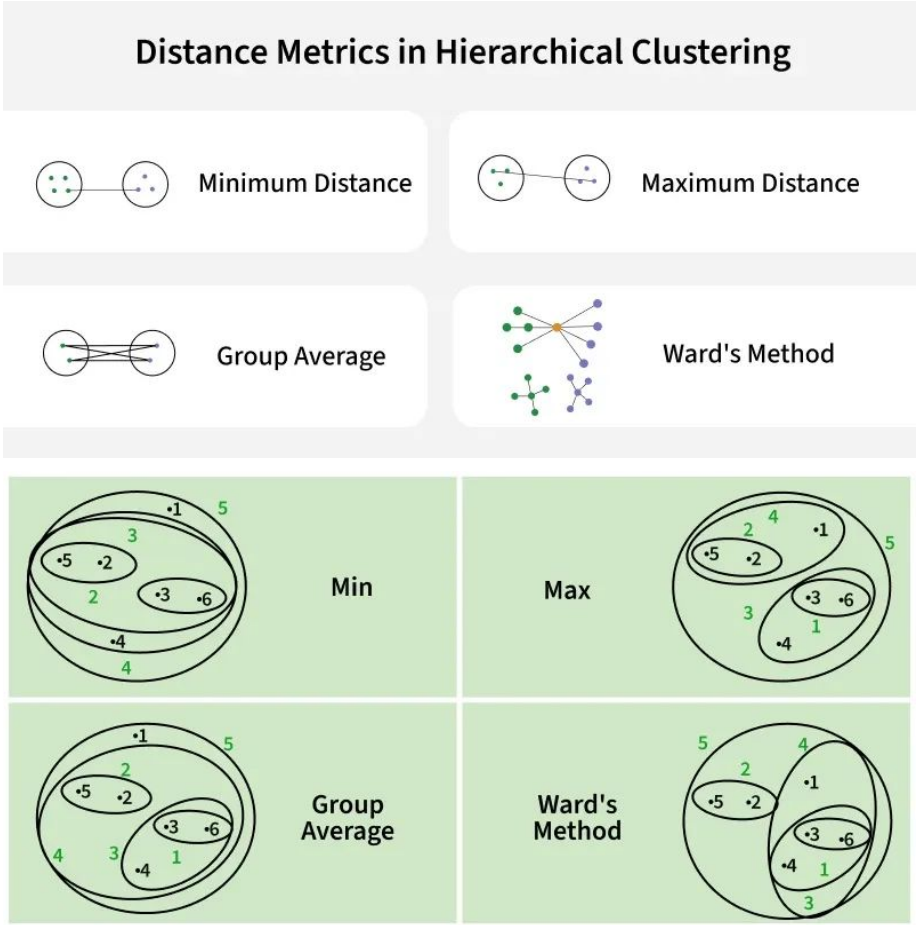


Distance Matrix

- Used to measure similarity between clusters
- Stores distance between every pair of clusters

Common distance methods:

- Min Distance (Single Linkage)
- Max Distance (Complete Linkage)
- Group Average Linkage
- Ward’s Method



Probabilistic clustering uses statistical methods to group data by modeling clusters as probability distributions. Algorithms estimate model parameters and assign points to clusters based on probabilities.

Probabilistic clustering assigns probabilities to data points for membership in multiple clusters, enabling soft assignments unlike hard partitioning in traditional methods.

Uses models like Gaussian Mixture Models (GMM) to capture uncertainty and handle overlapping data effectively.

Ideal for applications such as customer segmentation, where data points span multiple behavioral groups.

- Accommodates real-world data overlap by quantifying degrees of membership.
- Provides nuanced insights into data uncertainty and ambiguity.
- Employ parallel processing to distribute workload across systems.
- Use dimensionality reduction techniques (e.g., PCA) to simplify high-dimensional data.

Key Algorithms

EM Algorithm

- Iteratively improves parameters via E-step (estimates hidden data probabilities) and M-step (updates parameters to maximize likelihood).
- Repeats until convergence for incomplete or hidden variable data.

Bayesian Hierarchical Clustering (BHC)

- Agglomerative method using Bayesian probability to merge clusters based on posterior probabilities.
- Outputs dendrogram with uncertainty measures for cluster existence.

Variational Bayesian Inference (VI)

- Approximates posterior distributions by optimizing a simpler family, minimizing KL divergence via ELBO.
- Uses mean-field assumption and iterative updates for tractable inference.

Gibbs Sampling

- MCMC method for sampling from complex distributions in clustering.
- Iteratively updates cluster assignments and parameters; converges after burn-in.

Advantages

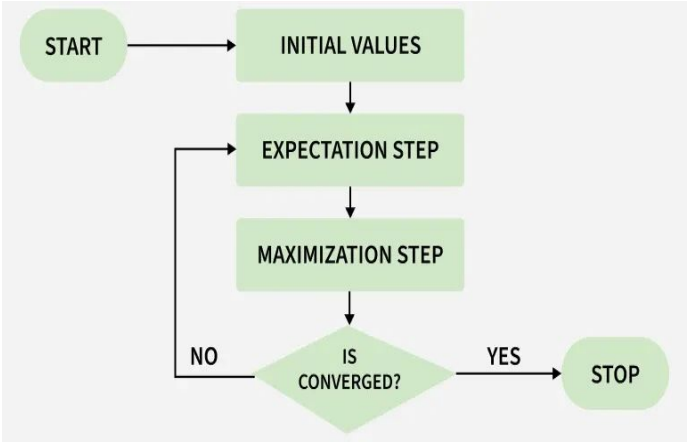
- Provides probabilistic memberships for overlapping clusters.
- Enables rigorous inference with stats like BIC/AIC.
- Handles varied cluster shapes and detects outliers.

Limitations

- Assumes specific distributions (e.g., Gaussian).
- Computationally intensive for large data.
- Sensitive to initialization; hard to select cluster count.

Expectation–Maximization (EM) algorithm

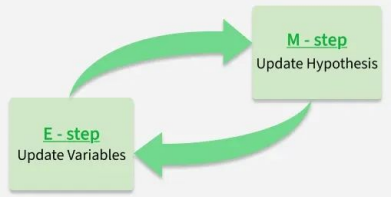
- Iterative optimization technique.
- It is used to estimate unknown parameters in probabilistic models.
- EM is particularly useful when: Data is incomplete, Data is noisy, Data contains hidden (latent) variables.
- The algorithm aims to find maximum likelihood estimates of model parameters.
- Guarantees non-decreasing likelihood at each iteration.
- Widely used in Gaussian Mixture Models (GMM), clustering, and statistical learning.
- Handles missing or hidden variables.



The EM algorithm works in two main iterative steps:

- **Expectation Step (E-Step)**
- **Maximization Step (M-Step)**

These steps are repeated until convergence.



A latent variable model contains: Observable variables, Unobservable (latent/hidden) variables

Observed variables:

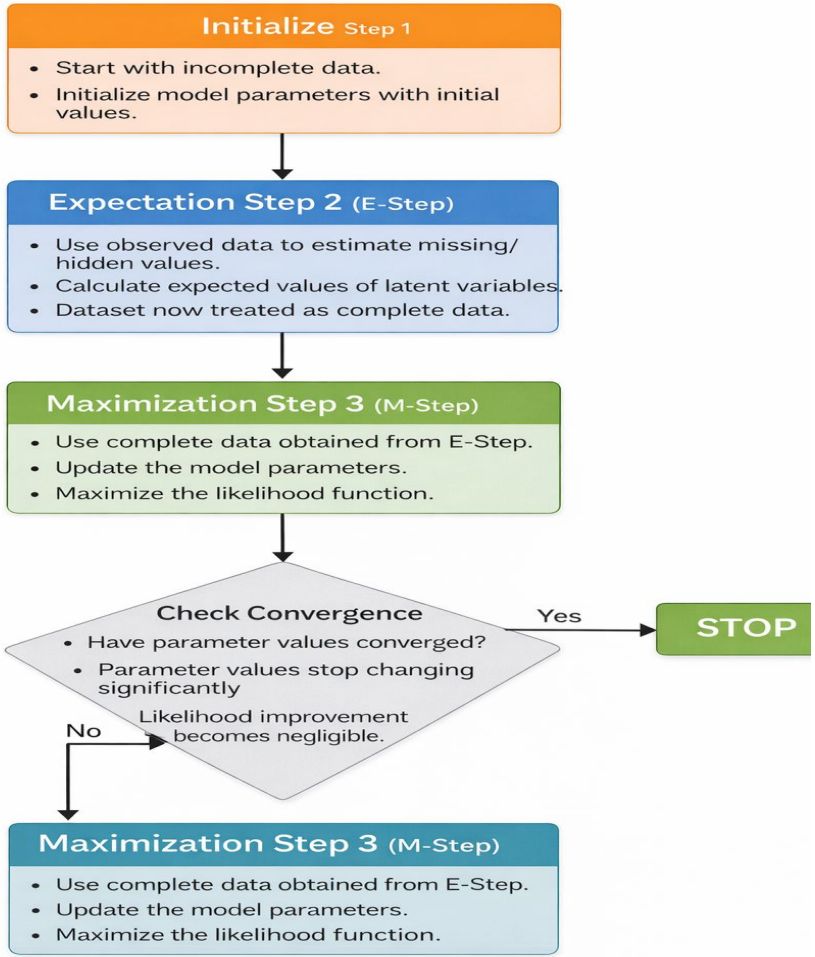
- Variables that can be directly measured from the dataset.

Latent variables:

- Variables that cannot be observed directly.
- Their values are inferred from observed variables.

The EM algorithm is used to estimate:

- Maximum Likelihood Estimation (MLE) parameters
- Maximum A Posteriori (MAP) parameters
- These parameters are estimated for models containing latent variables.
- Use observable samples to estimate unobservable sample
- Estimate hidden variable values from available data.
- Update model parameters based on these estimates.
- Repeat the process until convergence.



2. Variables and Notation

- **Observed data:** X
- **Latent (hidden) variables:** Z
- **Model parameters:** θ

Each row of the dataset is represented as:

$$X = \{x_1, x_2, ..., x_n\}$$

Latent variables correspond to:

$$Z = \{z_1, z_2, ..., z_n\}$$

3. Log-Likelihood Function

The likelihood of observed data is:

$$\ln p(X|\theta) = \ln \left(\sum_Z p(X, Z|\theta) \right)$$

Key Points

- The likelihood depends on **both observed variables (X) and latent variables (Z)**.
- The **summation over latent variables is inside the logarithm**, which makes direct maximization difficult.
- For **continuous latent variables**, the summation becomes an **integral**.

5. E-Step (Expectation Step)

Purpose

- Estimate the **expected values of hidden variables**.

Process

- Use the **current parameter estimates** θ^{old}.
- Compute the **posterior probability**:

$$p(Z|X, \theta^{old})$$

- Calculate the **expected complete log-likelihood**:

$$Q(\theta, \theta^{old}) = \sum_Z p(Z|X, \theta^{old}) \ln p(X, Z|\theta)$$

Interpretation

- Assigns **probabilities (responsibilities)** to hidden outcomes.
- Determines how likely each latent variable explains the data.

6. M-Step (Maximization Step)

Purpose

- Update model parameters to **maximize expected log-likelihood**.

Parameter Update

$$\theta^{new} = \arg\max_\theta Q(\theta, \theta^{old})$$

Key Idea

- Find new parameters that **maximize Q function**.
- Improves how well the **model explains observed data**.

7. General EM Algorithm Steps

Step 1: Initialization

- Choose initial parameters:

$$\theta^{old}$$

- Assume the observed data follows a **specific probabilistic model**.

Step 2: E-Step

- Estimate **missing or hidden data**.
- Compute:

$$p(Z|X, \theta^{old})$$

- Evaluate **expected log-likelihood**.

Step 3: M-Step

- Update model parameters:

$$\theta^{new} = \arg\max_{\theta} Q(\theta, \theta^{old})$$

where

$$Q(\theta, \theta^{old}) = \sum_{Z} p(Z|X, \theta^{old}) \ln p(X, Z|\theta)$$

Step 4: Convergence Check

- Check whether:
  - **Parameter values stop changing significantly**, or
  - **Log-likelihood improvement becomes negligible**.

If not converged:

$$\theta^{old} \leftarrow \theta^{new}$$

Repeat **E-step and M-step**.

Example

In this scenario, we have two coins (A and B) with unknown biases (θ_A and θ_B). We flip a coin 10 times, repeat this for 5 sets, but **we don't know which coin was used for which set.**

---

1. The Core Problem: Hidden Variables

If we knew which coin was used for each set (the "labels"), we could just use simple math:

$$\hat{\theta}_A = \frac{\text{Total Heads from Coin A}}{\text{Total Flips from Coin A}}$$

However, because the labels are **hidden (latent variables)**, we can't do this directly. The EM algorithm solves this by iterating between two steps.

---

2. Initialization

We start by making a "best guess" for the biases. In the example provided:

- **Initial Guess:** $\theta_A^{(0)} = 0.60$ and $\theta_B^{(0)} = 0.50$.

3. The E-Step (Expectation)

Since we don't know which coin produced a specific result (e.g., 5 heads and 5 tails), we use **Bayes' Theorem** to calculate the probability that the result came from Coin A vs. Coin B based on our current guess.

Calculating Membership Weights

For the first set (5 Heads, 5 Tails):

- **Likelihood for A:** Based on 0.6 bias, how likely is 5H/5T?
- **Likelihood for B:** Based on 0.5 bias, how likely is 5H/5T?
- **Result:** The math shows there is a **45% chance** it was Coin A and a **55% chance** it was Coin B.

We then "assign" the 10 flips to each coin proportionally:

- **Coin A gets:** $10 \text{ flips} \times 0.45 = 4.5 \text{ flips}$ (specifically 2.25 heads and 2.25 tails).
- **Coin B gets:** $10 \text{ flips} \times 0.55 = 5.5 \text{ flips}$ (specifically 2.75 heads and 2.75 tails).

Repeating this for all 5 sets gives us the totals seen in the table:

- **Coin A Total:** ≈ 21.3 Heads and 8.6 Tails.
- **Coin B Total:** ≈ 11.7 Heads and 8.4 Tails.

4. The M-Step (Maximization)

Now that we have "weighted" counts of heads and tails, we treat them as real data to update our estimates for the biases. This is the **Maximum Likelihood Estimation** part.

- **New Estimate for A:** $\theta_A^{(1)} = \frac{21.3}{21.3+8.6} \approx \mathbf{0.71}$
- **New Estimate for B:** $\theta_B^{(1)} = \frac{11.7}{11.7+8.4} \approx \mathbf{0.58}$

---

5. Iteration and Convergence

The algorithm doesn't stop there. We take these new estimates (0.71 and 0.58) and plug them back into the **E-Step** to get even more accurate weights.

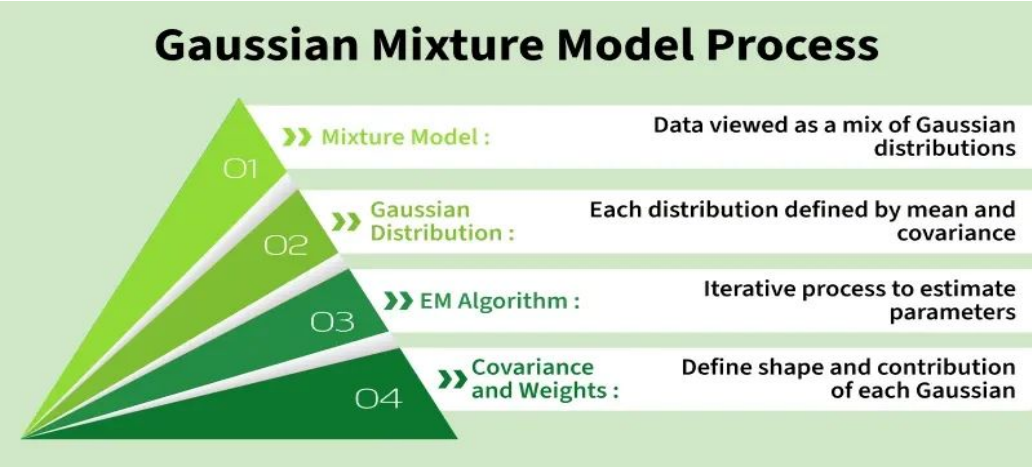
As the biases become more distinct, it becomes clearer which sets belonged to which coin. By the **10th iteration**, the values settle (converge) at:

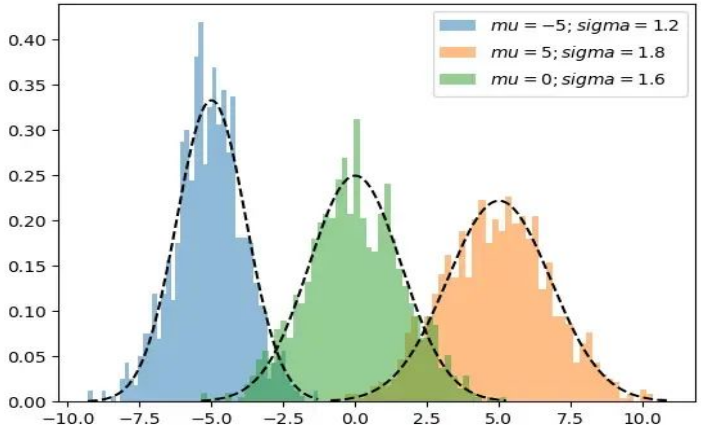
$$\theta_A^{(10)} \approx 0.80, \quad \theta_B^{(10)} \approx 0.52$$

Gaussian Mixture Model (GMM)

- Gaussian Mixture Model (GMM) is a probabilistic clustering method.
- It assumes that data is generated from a mixture of several Gaussian distributions.
- Each cluster is represented by a Gaussian distribution.
- GMM uses the Expectation–Maximization (EM) algorithm to estimate parameters.

GMM uses soft clustering, unlike K-Means.
Hard Clustering (K-Means): Each data point belongs to exactly one cluster.
Example: “You are in Cluster 1.”
Soft Clustering (GMM): Each data point has a probability of belonging to each cluster.
Example: 75% probability in Cluster A, 25% probability in Cluster B. This probability is called "responsibility."





three one-dimensional Gaussian distributions with distinct means and variances

- Univariate Gaussian Distribution

$$\mathcal{N}(x \mid \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

mean variance

- Multi-Variate Gaussian Distribution

$$\mathcal{N}(x \mid \mu, \Sigma) = \frac{1}{(2\pi|\Sigma|)^{1/2}} \exp\left\{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right\}$$

mean covariance

- Consider log of Gaussian Distribution

$$\ln p(x \mid \mu, \Sigma) = -\frac{1}{2}\ln(2\pi) - \frac{1}{2}\ln|\Sigma| - \frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)$$

- Take the derivative and equate it to zero

$$\frac{\partial \ln p(x \mid \mu, \Sigma)}{\partial \mu} = 0 \qquad \frac{\partial \ln p(x \mid \mu, \Sigma)}{\partial \Sigma} = 0$$

$$\mu_{ML} = \frac{1}{N}\sum_{n=1}^{N} x_n \qquad \Sigma_{ML} = \frac{1}{N}\sum_{n=1}^{N}(x_n - \mu_{ML})(x_n - \mu_{ML})^T$$

Where, N is the number of samples or data points

- Linear super-position of Gaussians

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x \mid \mu_k, \Sigma_k)$$

Number of Gaussians

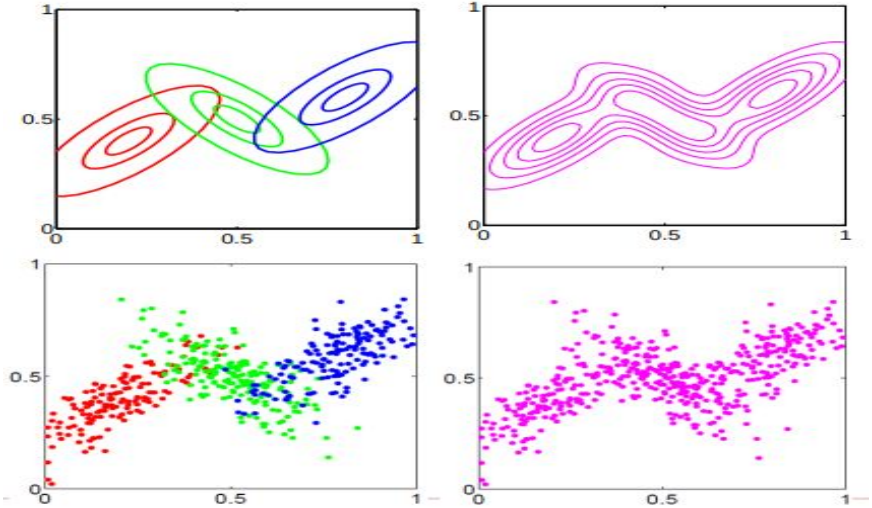
Mixing coefficient: weightage for each Gaussian dist.

- Normalization and positivity require $0 \le \pi_k \le 1, \sum_{k=1}^{K} \pi_k = 1$
- Consider log likelihood

$$\ln p(X \mid \mu, \Sigma, \pi) = \sum_{n=1}^{N} \ln p(x_n) = \sum_{n=1}^{N} \ln\left\{\sum_{k=1}^{K} \pi_k N(x_n \mid \mu_k, \Sigma_k)\right\}$$

ML does not work here as there is no closed form solution

Parameters can be calculated using Expectation Maximization (EM) technique



Mixture of 3 Gaussian

Latent variable: posterior prob.

- We can think of the mixing coefficients as prior probabilities for the components
- For a given value of 'x', we can evaluate the corresponding posterior probabilities, called responsibilities
- From Bayes rule

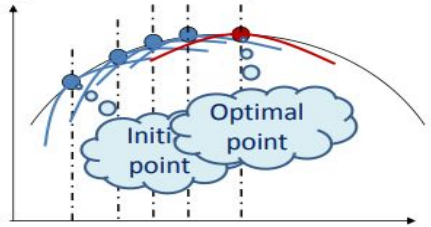
$$\gamma_k(\mathbf{x}) = p(k \mid \mathbf{x}) = \frac{p(k)p(\mathbf{x} \mid k)}{p(\mathbf{x})}$$

Latent Variable

$$= \frac{\pi_k \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_j, \Sigma_j)} \quad \text{where,} \quad \pi_k = \frac{N_k}{N}$$

Interpret N_k as the effective no. of points assigned to cluster k.

- EM algorithm is an iterative optimization technique which is operated locally



- Estimation step: for given parameter values we can compute the expected values of the latent variable.
- Maximization step: updates the parameters of our model based on the latent variable calculated using ML method.

EM Algorithm for GMM

Given a Gaussian mixture model, the goal is to maximize the likelihood function with respect to the parameters comprising the means and covariances of the components and the mixing coefficients).

1. Initialize the means $\boldsymbol{\mu}_j$, covariances Σ_j and mixing coefficients π_j, and evaluate the initial value of the log likelihood.
2. E step. Evaluate the responsibilities using the current parameter values

$$\gamma_j(\mathbf{x}) = \frac{\pi_k \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_j, \Sigma_j)}$$

3. M step. Re-estimate the parameters using the current responsibilities

$$\boldsymbol{\mu}_j = \frac{\sum_{n=1}^{N} \gamma_j(\mathbf{x}_n)\mathbf{x}_n}{\sum_{n=1}^{N} \gamma_j(\mathbf{x}_n)} \qquad \Sigma_j = \frac{\sum_{n=1}^{N} \gamma_j(\mathbf{x}_n)(\mathbf{x}_n - \boldsymbol{\mu}_j)(\mathbf{x}_n - \boldsymbol{\mu}_j)^T}{\sum_{n=1}^{N} \gamma_j(\mathbf{x}_n)} \qquad \pi_j = \frac{1}{N}\sum_{n=1}^{N} \gamma_j(\mathbf{x}_n)$$

4. Evaluate log likelihood

$$\ln p(\mathbf{X} \mid \boldsymbol{\mu}, \Sigma, \boldsymbol{\pi}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n \mid \boldsymbol{\mu}_k, \Sigma_k) \right\}$$

If there is no convergence, return to step 2.

A Gaussian Mixture Model is defined as a weighted sum of M component Gaussian densities. It is represented by the following probability density function:

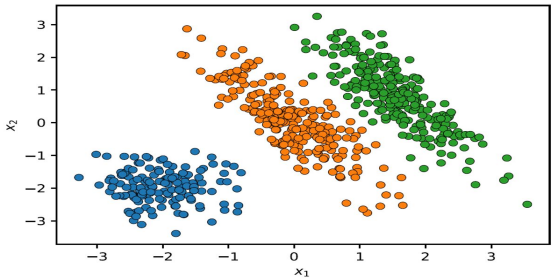
$$p(\mathbf{x}|\lambda) = \sum_{i=1}^{M} w_i g(\mathbf{x}|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$$

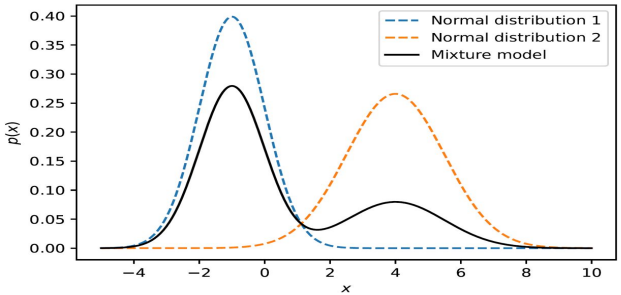
- $\mathbf{x}$: Represents a D-dimensional continuous-valued data vector, such as measurements or features.
- $g(\mathbf{x}|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$: Represents the component Gaussian densities, each defined by a mean vector $\boldsymbol{\mu}_i$ and a covariance matrix $\boldsymbol{\Sigma}_i$.

The complete GMM is collectively represented by the notation λ, which encompasses all component parameters:

- **Notation**: $\lambda = \{w_i, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i\}$ for $i = 1, \ldots, M$.
- **Weight Constraint**: The mixture weights must satisfy the constraint that their sum equals 1: $\sum_{i=1}^{M} w_i = 1$.

GMM clustering





- **Definition**: A continuous probability distribution used to model feature vectors in a D-dimensional space.
- **Mathematical Form**: The distribution is defined by the formula:

$$\mathcal{N}(x|\mu,\Sigma) = \frac{1}{(2\pi|\Sigma|)^{1/2}} \exp\left\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\}$$

.

- **Core Parameters**:
  - **Mean (μ)**: The center of the distribution.
  - **Covariance (Σ)**: Represents the spread and orientation of the data.
- **Estimation**: Standard parameters can be estimated using **Maximum Likelihood (ML)** by taking derivatives of the log distribution and equating to zero.
- **Definition**: A GMM is a parametric probability density function represented as a **weighted sum** of M component Gaussian densities.
- **Mixture Equation**:

$$p(\mathbf{x}|\lambda) = \sum_{i=1}^{M} w_i g(\mathbf{x}|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$$

.

- **Mixture Weights (w_i)**: These represent the influence of each component and must satisfy the constraint $\sum_{i=1}^{M} w_i = 1$.
- **Application**: Commonly used in biometric systems to model continuous measurements like vocal-tract spectral features.

The defining feature of a GMM is its ability to handle uncertainty through **soft clustering**.

- **Contrast with K-Means:** While K-Means "forcefully" assigns a point to one cluster (**hard clustering**), GMMs assign a probability distribution.
- **Likelihood Scores:** Every data point receives a set of probabilities indicating how likely it belongs to *each* Gaussian component in the model.

How does the model "learn" the best parameters from training data? There are two primary paths:

- **EM Algorithm (Expectation-Maximization):** The standard iterative approach that refines weights, means, and covariances from scratch.
- **MAP Estimation (Maximum A Posteriori):** Used when you already have a "well-trained prior model" (a general idea of what the data should look like) and want to adapt it to a specific set of new data.

Mathematically, a GMM isn't just a clustering tool; it is a **Parametric Probability Density Function (PDF)**.

- **Weighted Sum:** It is represented as the sum of several Gaussian densities, each scaled by a weight (π_k or w_k).
- **Formulaic View:**

$$p(x) = \sum_{k=1}^{K} w_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

where the weights w_k must sum to 1.

Parameters of a Gaussian Cluster: Each cluster k is defined by **three parameters**:

- **Mean (μ)** → location of cluster, Center of the cluster
- **Covariance (Σ)** → shape and orientation, Shape, spread, and orientation of the cluster
- **Mixing weight (π)** → probability of choosing that cluster, relative importance, or size of the cluster

4. Posterior Probability (Responsibility)

This is calculated in the **Expectation Step (E-Step)**.

$$P(z_n = k|x_n) = \frac{\pi_k \cdot N(x_n|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \cdot N(x_n|\mu_j, \Sigma_j)}$$

Meaning

- Probability that **data point** x_n belongs to **cluster** k.
- Also called **cluster responsibility**.

---

5. Total Likelihood of Data

The probability of observing data point x_n:

$$P(x_n) = \sum_{k=1}^{K} \pi_k N(x_n|\mu_k, \Sigma_k)$$

Meaning

- Measures how well the **mixture model explains the data**.

6. EM Algorithm in GMM

The **Expectation–Maximization (EM) algorithm** is used to estimate the parameters.

Step 1: E-Step (Expectation Step)

- Keep cluster parameters fixed.
- Calculate **responsibility values** for each data point.

Output:

- Probability that a point belongs to each cluster.

Step 2: M-Step (Maximization Step)

- Update the parameters using the responsibilities.

Update:

- Mean μ_k
- Covariance Σ_k
- Mixing weight π_k

Goal:

- **Maximize the likelihood of the dataset.**

Step 3: Iteration

- Repeat **E-step and M-step**.
- Continue until **log-likelihood stops improving**.

7. Convergence Condition

Algorithm stops when:

- Parameter values **stop changing significantly**, or
- **Log-likelihood improvement becomes very small**.

4. Log-Likelihood of the Mixture Model

The objective optimized by EM is:

$$L(\mu_k, \Sigma_k, \pi_k) = \prod_{n=1}^{N} \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n \mid \mu_k, \Sigma_k)$$

EM increases this likelihood in every iteration.

Cluster Shapes in GMM

In GMM, each cluster is a Gaussian defined by:

- **Mean (μ):** Center of the cluster.
- **Covariance (∑):** Controls the shape, orientation and spread of the cluster.

Advantages of GMM over K-Means

GMM Advantages: Handles **elliptical clusters,**
Handles **overlapping clusters ,** Provides **probabilistic cluster membership**

Example Cluster Shapes

GMM can model: ,Elongated clusters, Tilted clusters, Overlapping clusters

EM Algorithm for Gaussian Mixture Models

- probabilistic clustering algorithms.
- It assumes that data points are generated from a mixture of Gaussian distributions.
- The Expectation–Maximization (EM) algorithm is used to estimate the model parameters.
- EM iteratively updates cluster parameters until convergence.
- Parameters include means, covariance matrices, and mixing weights.

Input

- Data $O = \{o_1, o_2, \ldots, o_n\}$
- Number of clusters K

Output

- Parameters $\theta = \{\mu_j, \Sigma_j, \pi_j\}$ for $j = 1$ to K

EM Algorithm for Gaussian Mixture Model (GMM)

1. Initialization (INIT)

Randomly initialize:

- μ_j — mean of cluster j
- Σ_j — covariance matrix (often initialized as identity matrix I)
- π_j — mixing weight $= 1/K$

Ensure:

$$\sum \pi_j = 1$$

2. Repeat Until Convergence

E-Step (Expectation)

For each data point o_i and cluster j:

$$\gamma_{ij} = \frac{\pi_j \cdot N(o_i \mid \mu_j, \Sigma_j)}{\sum_l [\pi_l \cdot N(o_i \mid \mu_l, \Sigma_l)]}$$

Where:

- γ_{ij} = probability that point o_i belongs to cluster j
- $N(o_i \mid \mu_j, \Sigma_j)$ = Gaussian probability density function

M-Step (Maximization)

Update Mean

$$\mu_j = \frac{\sum(\gamma_{ij} \cdot o_i)}{\sum \gamma_{ij}}$$

Update Covariance

$$\Sigma_j = \frac{\sum [\gamma_{ij}(o_i - \mu_j)(o_i - \mu_j)^T]}{\sum \gamma_{ij}}$$

Update Mixing Weight

$$\pi_j = \frac{\sum \gamma_{ij}}{n}$$

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}).$$

Evaluate the log likelihood

$$\ln p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}$$

3. Return

- $\theta = (\mu_j, \Sigma_j, \pi_j)$
- γ matrix (**responsibility matrix**)

Key

- $\gamma_{ij} = P(\text{cluster } j \mid \text{point } i)$ → **soft cluster assignment**
- γ_{ij} represents the probability that data point i belongs to cluster j.
- Unlike **K-Means (hard clustering)**, **GMM performs soft clustering**, where each point can belong to multiple clusters with different probabilities.

Association Rule Mining

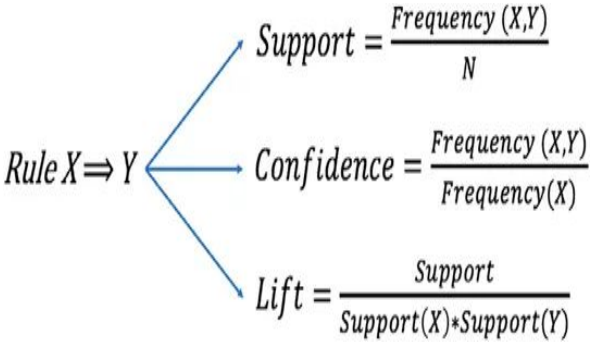
- A data mining technique used to identify frequent patterns, correlations, or associations within large datasets.
- Primary Goal: To uncover "if-then" relationships between variables that are not immediately apparent.
- Format: Expressed as **Antecedent (If) → Consequent (Then).**
- Example: If a customer buys bread, they are likely to buy butter.
- Value: Enables data-driven decision-making in retail, marketing, and healthcare.

Antecedent: * The "If" part of the statement.

- Represents the condition or item(s) already present in the transaction.

Consequent:

- The "Then" part of the statement.
- Represents the item(s) found in combination with the antecedent.



Logic: The rule does not necessarily imply causality, but rather a strong mathematical co-occurrence.

To ensure rules are meaningful and not just random noise, we use specific metrics:

- **1. Support:** Measures how frequently an itemset appears in the total dataset.
  - **Formula:** $Support(A \rightarrow B) = \frac{\text{Transactions containing both A and B}}{\text{Total Transactions}}$
  - *Insight:* High support means the pattern is common across the entire database.
- **2. Confidence:** Measures the reliability of the rule (how often B appears when A is present).
  - **Formula:** $Confidence(A \rightarrow B) = \frac{\text{Transactions containing both A and B}}{\text{Transactions containing A}}$
  - *Insight:* Higher confidence suggests a stronger predictive relationship.
- **3. Lift:** Measures the strength of the association compared to random chance.
  - **Formula:** $Lift(A \rightarrow B) = \frac{Confidence(A \rightarrow B)}{Support(B)}$
- **Interpreting Lift Values:**
  - **Lift > 1:** Positive association (A and B appear together more than expected).
  - **Lift = 1:** No association (A and B are independent).
  - **Lift < 1:** Negative association (The presence of A makes B less likely).

Association rule learning typically follows a two-step iterative process:

Identify Frequent Itemsets: Find all combinations of items that meet a user-defined minimum support threshold.

Generate Association Rules: From the frequent itemsets, create rules that meet a minimum confidence threshold.

Iterative Refinement: Thresholds are adjusted until the most actionable and significant rules are selected.

Apriori Algorithms:

- Breadth-first search (BFS) using a "bottom-up" strategy.
- Key Principle: All subsets of a frequent itemset must also be frequent (pruning).

Process: Starts with individual items → combines them into pairs → triples, etc.

* Pros: Easy to implement; works well on small/low-dimensional data.
  Cons: Requires multiple passes over the database; slow on large/dense datasets.

FP-Growth (Frequent Pattern Growth):

- Uses a compressed tree structure called an FP-Tree.
- Advantage: Does not require candidate generation; significantly faster and more memory-efficient than Apriori for large data.

ECLAT (Equivalence Class Clustering):

- Uses a depth-first search (DFS) and a vertical data format.
- Advantage: Directly computes intersections of transaction sets; efficient for sparse datasets.

Example Sample Transaction Dataset

- **Context**: A small retail dataset tracking five customer transactions.
- **Data Summary**:
  - **T1**: Bread, Butter , **T2**: Bread, Milk , **T3**: Bread, Butter, Milk , **T4**: Milk , **T5**: Bread, Butter
- **Goal**: Identify the relationship between purchasing **Bread (Antecedent)** and **Butter (Consequent)**.

Step 1 – Support Calculation

- **Definition**: Measures how common the specific combination is in the entire dataset.
- **Formula**:
- **Calculation**:

$$\text{Support} = \frac{\text{Transactions with both Bread and Butter}}{\text{Total Transactions}}$$

  - Occurrences of {Bread, Butter}: 3 (T1, T3, T5)
  - Total Transactions: 5
  - **Support = 3 / 5 = 0.6 (60%)**

Step 2 – Confidence Calculation

- **Definition**: Measures how often Butter is purchased given that Bread has already been purchased.
- **Calculation**:

$$\text{Confidence}(A \rightarrow B) = \frac{\text{Support}(A \cap B)}{\text{Support}(A)}$$

{Confidence} =
{ Support of (Bread & Butter)} / {Support of Bread} }

  - Support of Bread: 4 / 5 = 0.8 (80%)
  - **Confidence = 0.6 / 0.8 = 0.75 (75%)**
- **Interpretation**: There is a 75% chance that a customer buying Bread will also buy Butter.

Step 3 – Lift Calculation

- **Definition**: Measures the strength of the association while accounting for how popular Butter is on its own.
- **Formula**: **{Lift} = {Confidence} / {Support of Butter}**
- **Calculation**:
  - Support of Butter: 3 / 5 = 0.6 (60%)
  - **Lift = 0.75 / 0.6 = 1.25**

Interpretation & Actionable Insights

- **Lift Value (1.25)**: Since the lift is **greater than 1**, it indicates a positive association.
- **Meaning**: Customers are **1.25 times more likely** to buy Butter if they buy Bread, compared to buying Butter randomly.
- **Business Decisions**:
  - Place Bread and Butter in the same aisle.
  - Create a "bundle" discount for both items.
  - Target Bread buyers with Butter coupons.

Apriori Algorithms

- The Apriori Algorithm is an influential unsupervised machine learning algorithm for mining frequent itemsets and generating boolean association rules from transactional databases. Identify relationships between items by identifying frequent itemsets.
  - Finds all frequent itemsets exceeding a minimum support threshold
  - Generates association rules with minimum confidence
  - Uses bottom-up breadth-first search strategy
  - Exploits the Apriori property to prune the candidate space
  - Handles large transactional databases efficiently

Support

The probability (frequency) that a given itemset appears in the dataset. An itemset is frequent if its support ≥ min_support threshold.

Confidence

The likelihood that item B occurs given item A. Defined as: frequency(A ∪ B) / frequency(A). A rule A→B is valid if confidence ≥ min_conf.

Frequent Itemset

A set of items whose support meets or exceeds the minimum support threshold. Denoted L_k for k-itemsets. Basis for all association rules.

Apriori Property

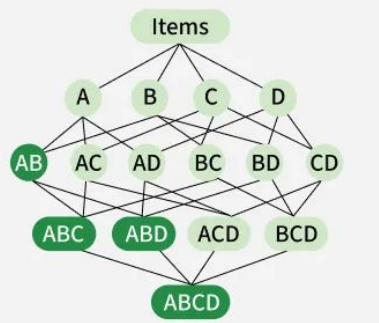
All subsets of a frequent itemset must also be frequent. Conversely, any superset of an infrequent itemset must be infrequent (anti-monotonicity).

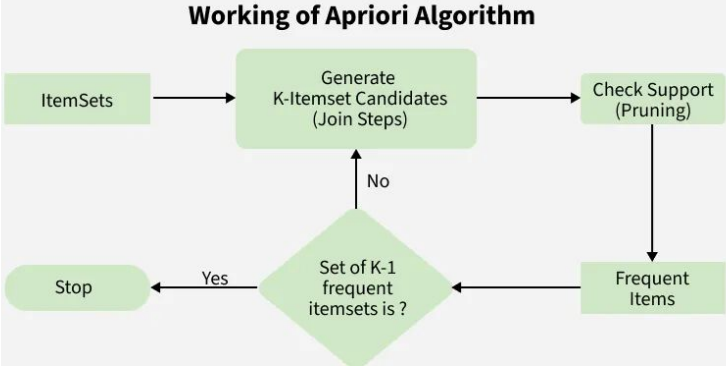
Association Rule

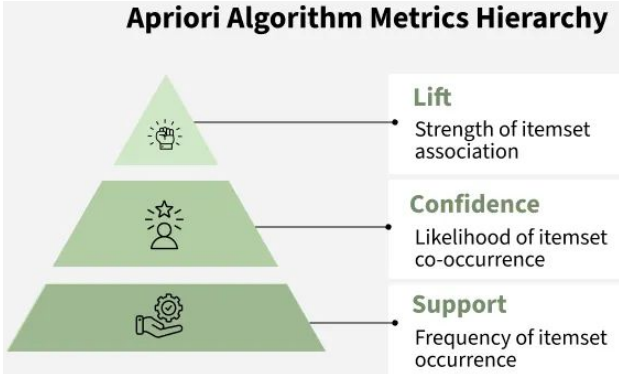
An implication of the form $E \rightarrow (F \setminus E)$, where $E \subset F$ and E is non-empty. A valid rule satisfies both minimum support and minimum confidence.

Candidate Itemset

Itemsets generated during each iteration by self-joining frequent (k-1)-itemsets. Candidates are tested against the database before promotion to frequent itemsets.







Step 1 – Identifying Frequent Itemsets

- The algorithm scans the dataset to count how many times each item appears.
- These are called 1-Itemsets.
- A minimum support threshold is defined.
- If an item appears equal to or above the minimum support, it is considered a frequent itemset.

Step 2 – Creating Possible Item Groups

- Frequent 1-itemsets are combined to create 2-itemsets.
- The algorithm counts how often these pairs appear in the dataset.
- If they meet the minimum support, they become frequent 2-itemsets.
- The same process continues for 3-itemsets, 4-itemsets, etc.
- The process stops when no larger frequent itemsets are found.

Step 3 – Removing Infrequent Item Groups

- Apriori uses an important rule called the Apriori Property.
- If an itemset is not frequent, any larger itemset containing it will also be infrequent.
- Such itemsets are pruned (removed) from further consideration.
- This reduces computation time and makes the algorithm more efficient.

Step 4 – Generating Association Rules

- After finding frequent itemsets, the algorithm generates association rules.

- These rules show relationships between items.

- Each rule is evaluated using:
  - Support – frequency of itemset in the dataset
  - Confidence – likelihood that item B is purchased with item A
  - Lift – strength of the relationship between items

- The rules with high support, confidence, and lift are considered strong.

Apriori Algorithms Step-by-Step

Frequency Table:

Create a frequency table for all individual data items in the transactional database. Each item is counted across all transactions.

Support Threshold:

Define the support threshold (e.g., 50%). Items contributing to more than this percentage of transactions are considered significant.

Generate Combinations:

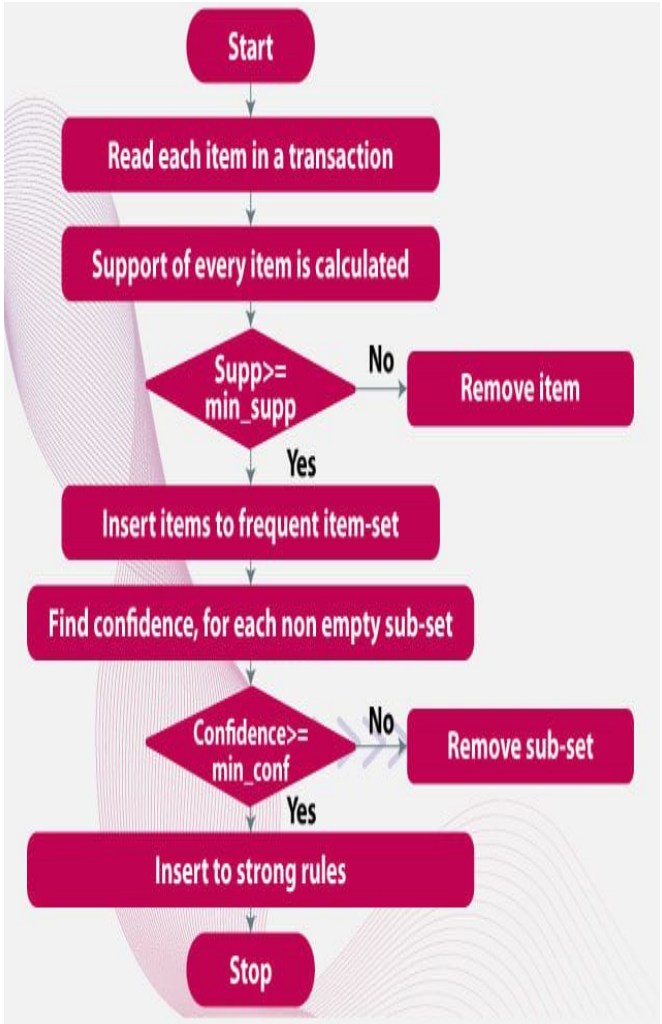
Make all possible combinations (candidate itemsets) of the significant items obtained in previous steps. Start with pairs, then triples, etc.

Filter by Support:

Determine the frequency of all possible combinations. Keep only those whose support value exceeds the predetermined threshold.

Self-Join & Repeat:

Self-join the frequent itemsets to obtain combinations with one additional item. Prune using the Apriori property. Identify significant parameter combinations.



Phase 1: Find Frequent Itemsets

- Start with singletons (1-itemsets) as candidates
- Scan database to count frequencies of all candidates
- Retain sets with support ≥ min_support (frequent)
- Generate k+1 candidates from frequent k-itemsets by self-join
- Prune any candidate with an infrequent subset (Apriori property)
- Repeat until no more candidates can be generated

Phase 2: Generate Association Rules

- For each frequent itemset F found in Phase 1
- Consider every non-empty proper subset $E \subset F$
- Form the rule: $E \rightarrow (F \setminus E)$
- Compute confidence = frequency(F) / frequency(E)
- Output the rule if confidence ≥ min_conf threshold
- Result: all frequent + confident association rules

- The Apriori Algorithm mines frequent itemsets and association rules from transactional data using breadth-first search.
- Iteratively generates and prunes candidates using the Apriori Property — all subsets of a frequent itemset must be frequent.

Input:

- Item set I, transactional database D, minimum frequency threshold min_fr, minimum confidence threshold min_conf.

Output:

- All frequent itemsets and all valid association rules $E \rightarrow (F \setminus E)$ in D satisfying both thresholds.

Join Step

C_k is generated by joining L_{k-1} with itself

- Two itemsets p and q from L_{k-1} are joined if they share the first (k-2) items in lexicographic order
- The joined candidate has k items total
- Example: {A, B, C} and {A, B, D} → candidate {A, B, C, D}
- This generates all possible candidates of size k from k-1 frequent itemsets

Prune Step

Any (k-1)-subset that is not frequent cannot be a subset of a frequent k-itemset

- After joining, check all (k-1)-subsets of each candidate
- Remove the candidate if any subset is NOT in L_{k-1}
- This exploits the Apriori Property (anti-monotonicity): a subset of an infrequent set cannot be frequent
- Significantly reduces number of database scans needed

Ex: Consider the following dataset and we will find frequent Item-Sets and generate association rules for them:

Transaction ID	Items Bought
T1	Bread, Butter, Milk
T2	Bread, Butter
T3	Bread, Milk
T4	Butter, Milk
T5	Bread, Milk

Step 1: Setting the Parameters

Minimum Support Threshold: 50%

Item must appear in at least 3 of 5 transactions

Support(A) =

Transactions containing A ÷ Total Transactions

Example: Bread appears in 4/5 transactions
→ Support = 4/5 = 80% ✓ (≥ 50%)

Minimum Confidence Threshold: 70%

If customer buys X, likelihood they also buy Y

Confidence(X→Y) =

Support(X ∪ Y) ÷ Support(X)

Example: Bread→Milk
= Support({Bread,Milk}) / Support({Bread})
= 3/4 = 75% ✓ (≥ 70%)

Step 2: Find Frequent 1-Itemsets

Count how many transactions include each item. Items with Support ≥ 50% qualify.

Item	T1	T2	T3	T4	T5	Count	Support %	Status
Bread	✓	✓	✓		✓	4	80%	✓ **PASS**
Butter	✓	✓		✓		3	60%	✓ **PASS**
Milk		✓	✓	✓	✓	4	80%	✓ **PASS**

Apriori Principle

All 3 items (Bread, Butter, Milk) meet the 50% threshold → They all qualify as Frequent 1-Itemsets.
Any item with Support < 50% would be pruned at this stage and excluded from further consideration.

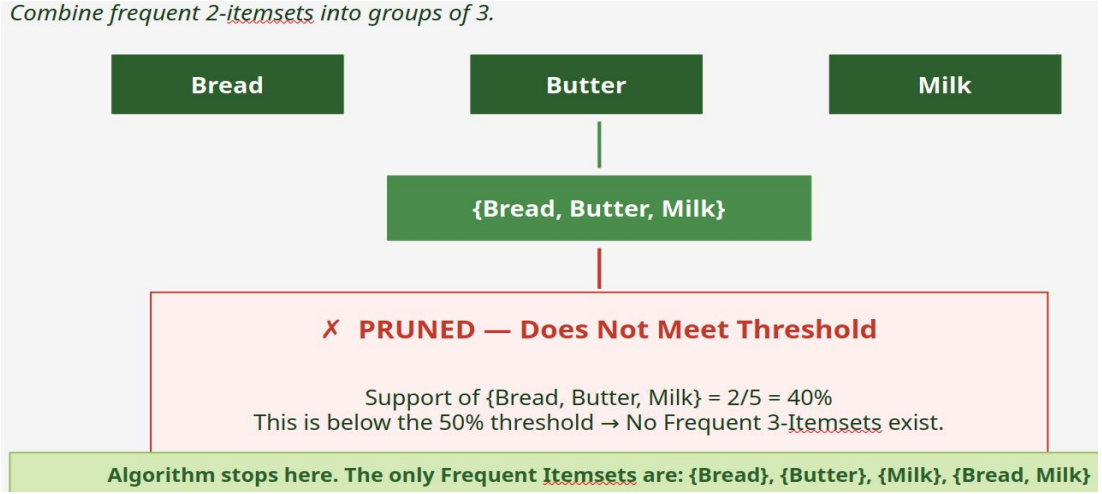
Step 3: Generate Candidate 2-Itemsets

Combine frequent 1-itemsets into pairs and calculate their support.

✗ PRUNED	✓ FREQUENT	✗ PRUNED
{Bread, Butter}	{Bread, Milk}	{Butter, Milk}
40%	60%	40%
2/5 transactions	3/5 transactions	2/5 transactions
Appears in T1, T2	*Appears in T2, T3, T5*	*Appears in T2, T4*
Below 50% threshold	Passes 50% threshold	Below 50% threshold

→ **Only {Bread, Milk} qualifies as a Frequent 2-Itemset**

Step 4: Generate Candidate 3-Itemsets



Step 5: Generate Association Rules

Generate rules from frequent itemsets and calculate confidence. Rules with Confidence ≥ 70% are accepted.

Rule 1	Rule 2	Rule 3
Bread → Butter	**Butter → Bread**	**Bread → Milk**
Support({Bread, Butter}) = 2 Support({Bread}) = 4 **Confidence = 2/4 = 50%**	Support({Bread, Butter}) = 2 Support({Butter}) = 3 **Confidence = 2/3 = 66.67%**	Support({Bread, Milk}) = 3 Support({Bread}) = 4 **Confidence = 3/4 = 75%**
✗ Failed — Below 70% threshold	✗ Failed — Below 70% threshold	✓ Passed — Meets 70% threshold!

✓ **Final Strong Rule: Bread → Milk (Confidence: 75%)**

"If a customer buys Bread, they are 75% likely to also buy Milk."

FP-Growth Algorithm (Frequent Pattern Growth)

- It is used to mine frequent itemsets from large transactional datasets.
- Widely used in Data Mining and Market Basket Analysis.
- Example: Finding that “Milk and Bread are often purchased together.”
- It is more efficient than the Apriori algorithm.

Traditional algorithms like Apriori:

- Generate a large number of candidate itemsets.
- Require multiple database scans.

- This causes: High computational cost, Slow performance with large datasets
- FP-Growth solves these problems using a compact structure called FP-Tree.

- A tree-based data structure used to store the dataset in a compressed form.
- Represents frequent items and their relationships.
- Helps in efficient frequent pattern mining.
- Avoids candidate generation.
- Components of FP-Tree: Root Node, Item Nodes, Header Table, Node Links, Child Nodes

Feature	FP-Growth	Apriori
Candidate Generation	Not required	Required
Database Scans	2 scans	Multiple scans
Speed	Faster for large datasets	Slower
Memory Usage	Higher (tree structure)	Lower
Implementation	Complex	Simpler

The algorithm works in the following steps:

- Scan the Database: Count frequency of each item.
- Sort Items: Arrange items in descending order of frequency, Remove items below minimum support threshold.
- Construct FP-Tree: Insert transactions into the tree, Shared prefixes are merged.
- Mine the FP-Tree: Generate frequent itemsets recursively.
- Generate Association Rules: Identify relationships between items.

Data Compression in FP-Growth

- The dataset is compressed into FP-Tree structure.
- Transactions are stored as paths in the tree.
- Common prefixes share nodes, reducing memory usage.
- This makes the algorithm efficient for large datasets.

Mining the FP-Tree

- The algorithm breaks the FP-tree into conditional trees.
- Each conditional tree represents patterns related to a specific item.
- Recursive mining is used to discover frequent itemsets.
- Frequent patterns are extracted without candidate generation.

Unlike Apriori, FP-Growth avoids repeated database scans by compressing data into a tree structure — making it much faster on large datasets.

Transaction ID	Sorted Items by Frequency
T1	Bread, Milk, Butter
T2	Bread , Butter
T3	Bread, Milk
T4	Milk , Butter
T5	Bread

Grocery Store Transactions

We have 5 customer transactions.
Our goal is to find frequent itemsets — groups of items bought together often.

Minimum Support Count: ≥ 2 transactions

Step 1: Count Item Frequency

Step 2: Sort Items by Frequency

Re-order items within each transaction by descending support count. This maximises prefix sharing in the FP-Tree.

ORIGINAL ORDER			SORTED BY FREQUENCY ↓	
TXN	Items (Original)	→	TXN	Items (Sorted)
T1	Milk, Bread, Butter		T1	Bread, Milk, Butter
T2	Bread, Butter		T2	Bread, Butter
T3	Milk, Bread		T3	Bread, Milk
T4	Milk, Butter		T4	Milk, Butter
T5	Bread		T5	Bread

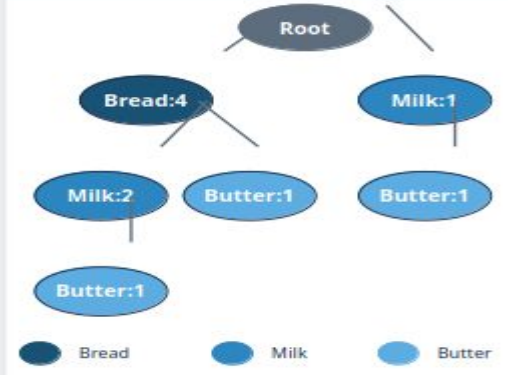
Why sort? Sorting by frequency ensures that high-frequency items appear at the top of the FP-Tree, maximising shared prefixes and minimising tree size.

Step 3: Build the FP-Tree

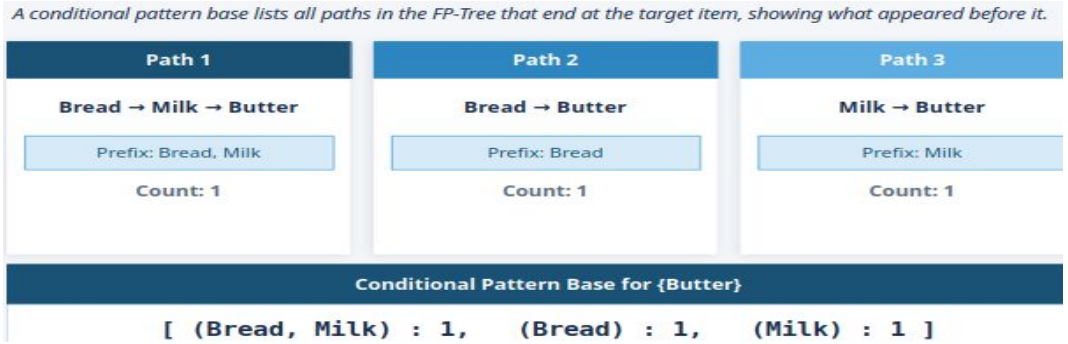
Transaction Insertions

T1	**Bread → Milk → Butter**	*New path from root*
T2	**Bread → Butter**	*Shares 'Bread' prefix*
T3	**Bread → Milk**	*Shares 'Bread→Milk' prefix*
T4	**Milk → Butter**	*New branch (no Bread)*
T5	**Bread**	*Increments Bread count*

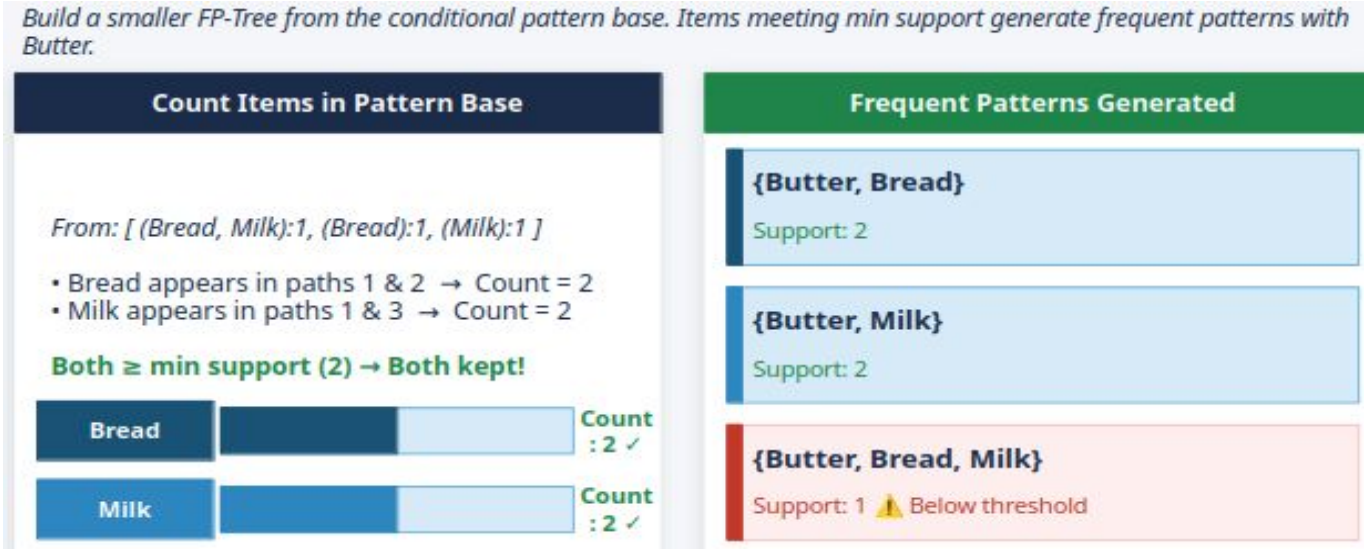
FP-Tree Structure



Step 4: Conditional Pattern Base (for Butter)



Step 5: Build Conditional FP-Tree for {Butter}



Step 6: All Frequent Itemsets

Repeating the conditional FP-tree process for every item yields the complete set of frequent itemsets.

Frequent Itemset	Support Count	Type	Status
{Bread}	4	1-Itemset	✓ **Frequent**
{Milk}	3	1-Itemset	✓ **Frequent**
{Butter}	3	1-Itemset	✓ **Frequent**
{Bread, Milk}	2	2-Itemset	✓ **Frequent**
{Bread, Butter}	2	2-Itemset	✓ **Frequent**
{Milk, Butter}	2	2-Itemset	✓ **Frequent**
{Bread, Milk, Butter}	1	3-Itemset	✗ **Not Frequent**

6 frequent itemsets found | {Bread, Milk, Butter} excluded — support count 1 < minimum support 2

FP-Growth is generally preferred for large-scale real-world datasets due to its efficiency.

Advantages of FP-Growth

- No candidate generation required.
- Fewer database scans.
- Efficient for large datasets.
- Uses compact data structure (FP-Tree).
- Can be parallelized for faster processing.

- FP-Growth is a powerful frequent pattern mining algorithm.

- Uses FP-tree to compress and analyze data efficiently.

- Performs better than Apriori for large datasets.

- Widely used in data mining and business analytics.

Dimensionality Reduction

Dimensionality reduction can be formalized as follows.

Consider a sample $S = (x_1, \ldots, x_m)$, a feature mapping $\Phi : X \to \mathbb{R}^N$ and the data matrix $X \in \mathbb{R}^{N \times m}$ defined as

$$(\Phi(x_1), \ldots, \Phi(x_m))$$

The i^{th} data point is represented by

$$x_i = \Phi(x_i)$$

or the i^{th} column of X, which is an **N-dimensional vector**.

Dimensionality reduction techniques broadly aim to find, for $k \ll N$, a **k-dimensional representation of the data**,

$$Y \in \mathbb{R}^{k \times m}$$

that is in some way **faithful to the original representation** X.

Dimensionality Reduction

- Process of reducing the number of input variables (features) in a dataset.

Main Approaches:

- Feature Extraction: Create new features by combining original ones.
- Feature Selection: Choose a subset of existing features without modifying them.

Benefits:

- Reduces the Curse of Dimensionality, Improves model training speed.,
- Enhances visualization and interpretability, Reduces overfitting by removing noise and redundant data.

Feature Extraction – Principal Component Analysis (PCA)

PCA: Linear transformation technique used to reduce dimensionality.
Converts original variables into a new coordinate system called Principal Components (PCs).
Key Idea: Capture maximum variance in fewer dimensions.

Principal Components:
PC1: Direction with maximum variance.
PC2: Orthogonal to PC1 and captures second highest variance.

Steps in PCA:

- Standardize the dataset.
- Compute the Covariance Matrix.
- Calculate Eigenvalues and Eigenvectors.
- Select top k eigenvectors.
- Project data onto these components.

Feature Extraction – Singular Value Decomposition (SVD)

SVD: Matrix factorization method used in dimensionality reduction.

Decomposes matrix A (m × n) into three matrices: $A = U\Sigma V^T$

U: Left singular vectors (orthogonal matrix).
Σ: Diagonal matrix containing singular values.
V^T: Right singular vectors (orthogonal matrix).

Relation to PCA: PCA can be computed using SVD on centered data.

Applications: Latent Semantic Indexing (LSI), Image compression, Noise reduction

Feature Selection – Feature Ranking

- Feature Ranking: A Filter Method for selecting features.
- Each feature is scored independently based on its relationship with the target variable.

Process:

1. Calculate statistical score for each feature.
2. Rank features from highest to lowest.
3. Select the top N features.

Common Metrics:

- Correlation Coefficient – measures linear relationship.
- Chi-Square Test – used for categorical variables.
- Information Gain / Mutual Information – detects nonlinear relationships.

Advantages: Very fast, Independent of ML algorithm.

Disadvantages: Ignores interaction between features.

Feature Selection – Subset Selection

Subset Selection: A Wrapper Method that evaluates combinations of features using a machine learning model.

Techniques:

Forward Selection

- Start with no features.
- Add the best feature at each step.

Backward Elimination

- Start with all features.
- Remove the least significant feature step by step.

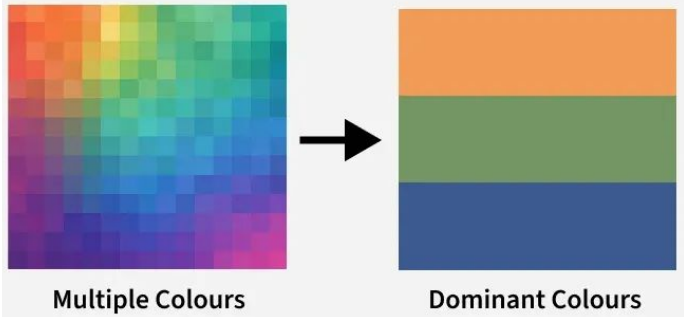
Exhaustive Search

- Tests every possible feature combination.

Aspect	Feature Extraction	Feature Selection
Transformation	Creates new features	Keeps original features
Interpretability	Low	High
Information Loss	Minimal (variance preserved)	Possible (features removed)
Usage	Image & signal processing	Business analytics, clinical data

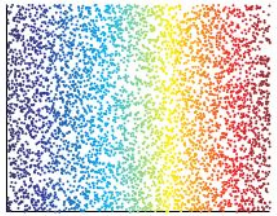
Principal Component Analysis (PCA)

- PCA is a dimensionality reduction technique.
- It reduces the number of features in a dataset while keeping the most important information.
- Converts correlated variables into a smaller set of uncorrelated variables called Principal Components.
  Helps to:
- Remove redundancy
- Improve computational efficiency
- Make data easier to visualize and analyze



How PCA Works

- PCA uses linear algebra to transform data into new features.
- These new features are called Principal Components (PCs).



high-dimensional lower-dimensional

It calculates:

- Eigenvectors → directions of maximum variance.
- Eigenvalues → importance of those directions.

- PCA selects components with the highest eigenvalues.
- Data is projected onto these components to simplify the dataset.
- Idea: Higher variance = more important information.

Steps of PCA

Step 1: Standardize the Data

- Features may have different scales or units (e.g., age vs salary).
- PCA standardizes data so all features contribute equally.
- After standardization: Mean = 0, Standard deviation = 1

Formula: $Z = \frac{X - \mu}{\sigma}$ $z = \frac{value - mean}{standard\ deviation}$

Where:
X = original value, μ = mean of the independent features,
σ = standard deviation

Step 2: Calculate Covariance Matrix

$\mu = \{ \mu 1, \mu 2, \cdots, \mu m \}$

$\sigma = \{\sigma 1, \sigma 2, \cdots, \sigma m\}$

- Covariance measures how two variables change together.
- It shows whether features are positively or negatively related.

Covariance formula: $cov(x_1, x_2) = \frac{\sum_{i=1}^{n}(x_{1i} - \bar{x_1})(x_{2i} - \bar{x_2})}{n-1}$

$$\begin{bmatrix} Cov(x,x) & Cov(x,y) & Cov(x,z) \\ Cov(y,x) & Cov(y,y) & Cov(y,z) \\ Cov(z,x) & Cov(z,y) & Cov(z,z) \end{bmatrix}$$

Where: bar(x1), bar(x2) = mean values, n = number of data points

Covariance values:
Positive → variables increase together, Negative → (one increases, other decreases), Zero → no relationship

Step 3: Find Principal Components

- PCA finds new axes where data variance is highest.
- Principal Components
- PC1 (First Principal Component)
  - Direction with maximum variance.
  - Contains most information.

- PC2 (Second Principal Component)
  - Perpendicular (orthogonal) to PC1.
  - Captures second highest variance.

Eigenvector equation:

$$AX = \lambda X$$

These directions come from eigenvectors of the covariance matrix.

- Eigenvalues help rank these directions by importance.

Where: A = covariance matrix, X = eigenvector, λ = eigenvalue

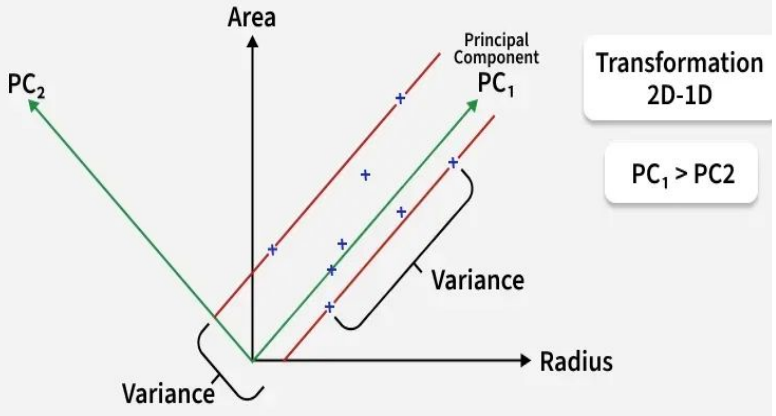
- Matrix A stretches vector X by factor λ; and Direction of X does not change.

Step 4: Select Top Components & Transform Data

- PCA ranks eigenvectors by their eigenvalues.
- Select top k components that capture most variance (e.g., 95%).

Then:

- Project original data onto these components.
- Reduce the number of dimensions while preserving patterns.



Example:

- Original data → 2D (Radius, Area)
- PCA finds PC1 and PC2.
- PC1 has larger variance than PC2.
- Projecting onto PC1 reduces data from **2D → 1D** while keeping most information.

$$FinalDataSet = FeatureVector^T * StandardizedOriginalDataSet^T$$

PCA steps:

- Start with original data matrix X
- Perform mean centering
- Compute covariance matrix
- Find eigenvectors and eigenvalues
- Select top k principal components
- Transform data to lower dimension

Step 1: Original Data in PCA

- PCA starts with the original dataset.
- The dataset contains m observations (samples) and N features (variables).
- It is represented by a data matrix X.
- Each row represents a data point.
- Each column represents a feature.

$$X \in \mathbb{R}^{N \times m}$$

Step 2: Data Matrix Representation

The original data matrix is written as:

$$X = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \vdots & \vdots & \dots & \vdots \\ x_{N1} & x_{N2} & \dots & x_{Nm} \end{bmatrix}$$

Where:

m = number of samples

N = number of features

xij = value of feature i in sample j

Step 3: Mean Centering of Data

$$\sum_{i=1}^{m} X_i = 0$$

Before applying PCA, the data must be mean-centered.
Steps:

- Calculate the mean of each feature.
- Subtract the mean from each data value.
- This ensures the dataset has zero mean.

Mean centering helps to:

- Remove bias in data.
- Ensure PCA captures true variance.
- Make the origin the center of the dataset.
- Improve accuracy of principal components.

Step 5: PCA Goal

Goal: Minimize reconstruction error.

Objective: $$\min_{P \in P_k} ||PX - X||_F^2$$

PCA projects the original data into k-dimensional space.

Where:
P = projection matrix, Pk = set of rank-k projection matrices , ||.||F = Frobenius norm

Step 6: Reconstruction Error

Reconstruction error is: $$||PX - X||_F^2$$

Meaning:

- Difference between original data and projected data.
- PCA tries to minimize this error.

Step 7: Covariance Matrix

To analyze data relationships, PCA computes: $$C = \frac{1}{m} X X^T$$

Where:

- C = covariance matrix
- Shows correlation between features.

Step 8: Principal Components

Principal components are:

- Directions of maximum variance in the dataset.
- Obtained from eigenvectors of covariance matrix.
- Sorted by largest eigenvalues.

Step 9: PCA Projection

The optimal projection matrix is:

$$P^* = U_k U_k^T$$

Where:

- **Uk** contains **top k singular vectors.**

Reduced representation:

$$Y = U_k^T X$$

Pseudocode of PCA:

input
A matrix of m examples $X \in \mathbb{R}^{m,d}$
number of components n
if $(m > d)$
$A = X^\top X$
Let $\mathbf{u}_1, \ldots, \mathbf{u}_n$ be the eigenvectors of A with largest eigenvalues
else
$B = XX^\top$
Let $\mathbf{v}_1, \ldots, \mathbf{v}_n$ be the eigenvectors of B with largest eigenvalues
for $i = 1, \ldots, n$ set $\mathbf{u}_i = \frac{1}{\|X^\top \mathbf{v}_i\|} X^\top \mathbf{v}_i$
output: $\mathbf{u}_1, \ldots, \mathbf{u}_n$

Ex: Suppose we have 4 observations with 2 features (X, Y). Goal: Reduce to 1D while preserving max variance.

Obs	X	Y
1	2	0
2	0	2
3	3	1
4	1	3

X = [2 0; 0 2; 3 1; 1 3]

Step 1 – Compute Mean

Mean per feature (n=4):
$\bar{x} = (2+0+3+1)/4 = 1.5$
$\bar{y} = (0+2+1+3)/4 = 1.5$

Mean vector: $\mu = [1.5, 1.5]$

Subtract from every data point next.

Step 2 – Mean-Center the Data

Xc = X - μ

Obs	X-1.5	Y-1.5
1	0.5	-1.5
2	-1.5	0.5
3	1.5	-0.5
4	-0.5	1.5

Xc = [0.5 -1.5; -1.5 0.5; 1.5 -0.5; -0.5 1.5]

Step 3 – Covariance Matrix

C = 1/(n-1) * Xc^T * Xc = 1/3 * Xc^T * Xc

C ≈ [1.67 -1.33;
-1.33 1.67]

Off-diagonals: Negative correlation (X↑ → Y↓)

Step 4 – Eigenvalues & Eigenvectors

$\det(C - \lambda I) = 0 \rightarrow$
$\lambda_1 \approx 3$ (largest)
$\lambda_2 \approx 0.33$

$v_1 = [0.707; -0.707]$
$v_2 = [0.707; 0.707]$

Step 5 – Select Principal Component

Pick v1 (λ_1=3 max variance):
PC1 = [0.707; -0.707]

Captures ~90% variance (λ_1 / trace(C))
Direction: 45° line (flipped)

Step 6 – Project onto PC1

$Y = Xc * PC1$

Obs	PC1 Value
1	1.414
2	-1.414
3	1.414
4	-1.414

Now 1D data!

Results & Advantages

Original: 2D → Reduced: 1D
Variance preserved: High

Advantages:
- **Lower complexity (ML/storage)**
- **Max variance retained**
- **Easier visualization**

Steps: Data → Center → Cov → Eig → PC → Project

Singular Value Decomposition (SVD)

- Rank represents the amount of independent information in a matrix.
- SVD decomposes a matrix into three matrices: $A = U\Sigma V^T$
- Only top singular values carry most information.
- Using k largest singular values, we obtain a low-rank approximation.
- This helps in data compression, dimensionality reduction, and faster computations.
- The rank of a matrix is the maximum number of linearly independent rows or columns in the matrix.
- It represents the amount of unique information contained in the matrix.
- Linear Independence $r \neq ar_1 + br_2$

  and where a and b are constants.

  Interpretation

  Higher rank → More independent information
  Lower rank → Redundant or dependent data

Applications:

- **Image compression**
- **Recommendation systems** (e.g., movie recommendations)
- **Digit recognition**
- **Destroyed / noisy image reconstruction**
- **Latent Semantic Analysis (LSA)**
- **Principal Component Analysis (PCA)**

$$A = U\Sigma V^T$$

For a matrix A of size $m \times n$:

1. Matrix U

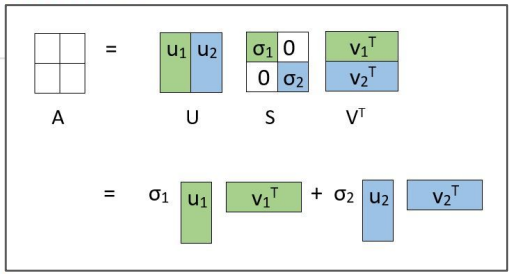
- Size: $m \times m$
- Orthogonal matrix
- Columns are **Left Singular Vectors**

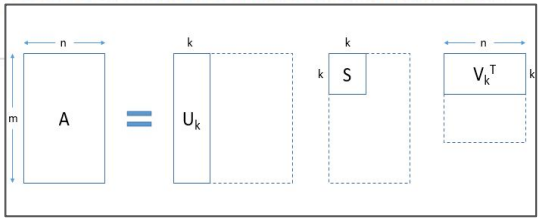
2. Matrix Σ (Sigma)

- Size: $m \times n$
- Diagonal matrix
- Contains **Singular Values**
- Values are **non-negative and arranged in decreasing order**

3. Matrix V^T

- Size: $n \times n$
- Transpose of matrix V
- Columns are **Right Singular Vectors**





Columns of U are orthonormal eigenvectors of: AA^T

Columns of V are orthonormal eigenvectors of: A^TA

SVD allows us to **express a matrix as a sum of low-rank matrices.**

$$A = U\Sigma V^T$$

Key idea:

- Only **few singular values are large**
- Remaining values are **very small (close to zero)**

Therefore:

- Small values can be **ignored**
- Data can be **compressed**

Instead of using all singular values:

Use only **top k singular values**

$$A_k = U_k\Sigma_k V_k^T$$

Benefits:

- Reduced storage
- Faster computation
- Minimal information loss

1. Matrix U: $(m \times r)$

r=rank(M)

Column-orthonormal matrix
Each column is a unit vector
Dot product between any two columns = 0
Represents left singular vectors

2. Matrix Σ: $(r \times r)$

Diagonal matrix
Contains singular values of matrix M
Singular values indicate importance of corresponding components
Larger values → more important information

Scales along each principal axis ; By the singular values σi ; Stretches/compresses the data; Larger σi = more dominant axis

3. Matrix V^T : $(r \times n)$

Row-orthonormal matrix
Each row is a unit vector
Dot product between rows = 0
Represents right singular vectors

Rotates the input vector x ; Aligns it with principal axes; Orthogonal transformation; Preserves lengths & angles

- SVD has strong connections with **eigenvalue decomposition**.
- It relates to eigenvectors of:

$$M^T M$$

and

$$M M^T$$

- Right singular vectors come from **eigenvectors of** $M^T M$.
- Left singular vectors come from **eigenvectors of** $M M^T$.

A = U · Σ · Vᵀ

U (m × m) Left Singular Vectors	**Describes the 'people' (row patterns) — e.g. user preferences**
Σ (m × n) Singular Values	**Diagonal matrix: shows the importance/weight of each factor**
Vᵀ (n × n) Right Singular Vectors	**Describes the 'items' (column patterns) — e.g. product similarity**

Example: A = [3 2 2 / 2 3 -2]

1. Compute AA^T and A^TA

Multiply A by its transpose (both ways).

Both share the same positive eigenvalues.

AA^T is $m \times m$; A^TA is $n \times n$ — both are symmetric & positive semidefinite.

---

2. Find Eigenvalues $(\det(A^TA - \lambda I) = 0)$

Solve the characteristic equation to get eigenvalues $\lambda_1, \lambda_2, \ldots$

Example: $\lambda_1 = 25, \lambda_2 = 9$

Singular values:

$\sigma_1 = 5,\ \sigma_2 = 3$

Singular values: $\sigma_i = \sqrt{\lambda_i}$ (always real and non-negative)

---

3. Find Right Singular Vectors (V)

Compute eigenvectors of A^TA for each eigenvalue.

Solve $(A^TA - \lambda I)v = 0$ for each λ.

Normalize and orthogonalize the resulting vectors → columns of V.

4. Compute Left Singular Vectors (U)

Use the relation:

$$u_i = \frac{1}{\sigma_i} A v_i$$

Each left vector is derived from the corresponding right vector.

Collect u_i columns → matrix U.

---

5. Assemble $A = U\Sigma V^T$

Stack singular values into diagonal matrix Σ.

Verify:

$$A = U\Sigma V^T$$

This can also be written as:

$$A = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \ldots$$

Example

$$A = \begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix}$$

We calculate:

$$AA^T = \begin{pmatrix} 17 & 8 \\ 8 & 17 \end{pmatrix}, \qquad A^TA = \begin{pmatrix} 13 & 12 & 2 \\ 12 & 13 & -2 \\ 2 & -2 & 8 \end{pmatrix}$$

These matrices are at least positive semidefinite (all eigenvalues are positive or zero). As shown, they share the same positive eigenvalues (25 and 9). The figure below also shows their corresponding eigenvectors.

$$AA^T = \begin{pmatrix} 17 & 8 \\ 8 & 17 \end{pmatrix}$$

eigenvalues: $\lambda_1 = 25, \lambda_2 = 9$

eigenvectors

$$u_1 = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} \quad u_2 = \begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix}$$

$$A^TA = \begin{pmatrix} 13 & 12 & 2 \\ 12 & 13 & -2 \\ 2 & -2 & 8 \end{pmatrix}$$

eigenvalues: $\lambda_1 = 25, \lambda_2 = 9, \lambda_3 = 0$

eigenvectors

$$v_1 = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \\ 0 \end{pmatrix} \quad v_2 = \begin{pmatrix} 1/\sqrt{18} \\ -1/\sqrt{18} \\ 4/\sqrt{18} \end{pmatrix} \quad v_3 = \begin{pmatrix} 2/3 \\ -2/3 \\ -1/3 \end{pmatrix}$$

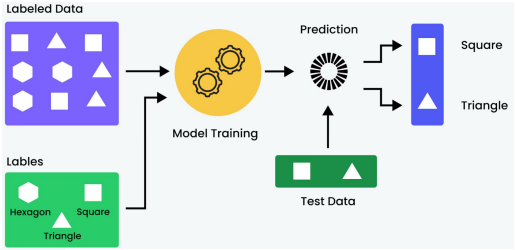
The singular values are the square root of positive eigenvalues, i.e. 5 and 3. Therefore, the SVD composition is

$$A = USV^T = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix} \begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 1/\sqrt{18} & -1/\sqrt{18} & 4/\sqrt{18} \\ 2/3 & -2/3 & -1/3 \end{pmatrix}$$

Next Topics…

Concept	Purpose
Semi-Supervised Learning	Use small labeled + large unlabeled data
MDP	Framework for decision making
Bellman Equation	Recursive value computation
Monte Carlo	Policy evaluation using episodes
Policy Iteration	Evaluate then improve policy
Value Iteration	Directly compute optimal values

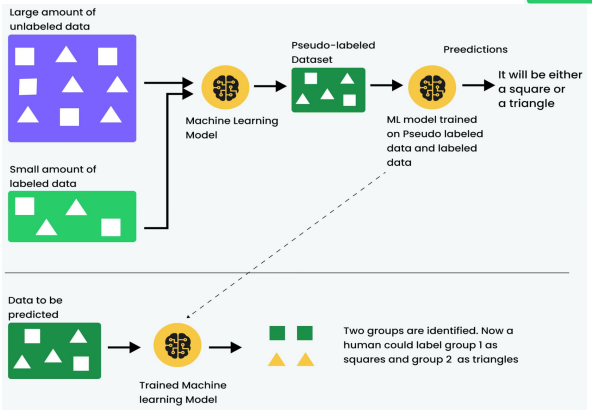
Semi-Supervised Learning

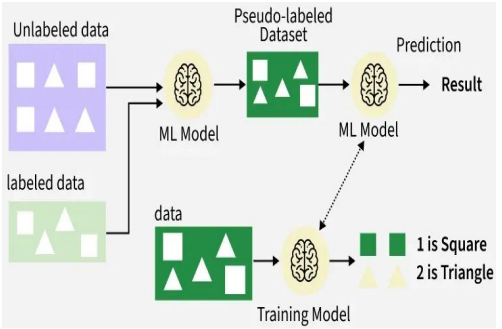


- Works well with **high-dimensional data** such as **images and text**.
- Uses both **labeled data (for guidance)** and **unlabeled data (to learn structure)**.
- These techniques **improve model performance with limited labeled data**.
- They expand labeled datasets and **use relationships and data distributions** to enhance learning.

Key Semi-Supervised Techniques

- **Self-Training**
- **Co-Training**
- **Multi-View Learning**
- **Graph-Based Methods**
- **Generative Models**





- Label Propagation: Data represented as a graph,
  Similar data points connected by edges,
  Labels spread through neighbors based on similarity.
- Co-Training: Uses two models trained on different feature views, Each model labels data for the other.
- Pseudo-Labeling: Model assigns labels to unlabeled data, Only high-confidence predictions used as true labels.

Necessary Condition : In Semi-Supervised Learning (SSL), the unlabeled data must be relevant to the learning task.

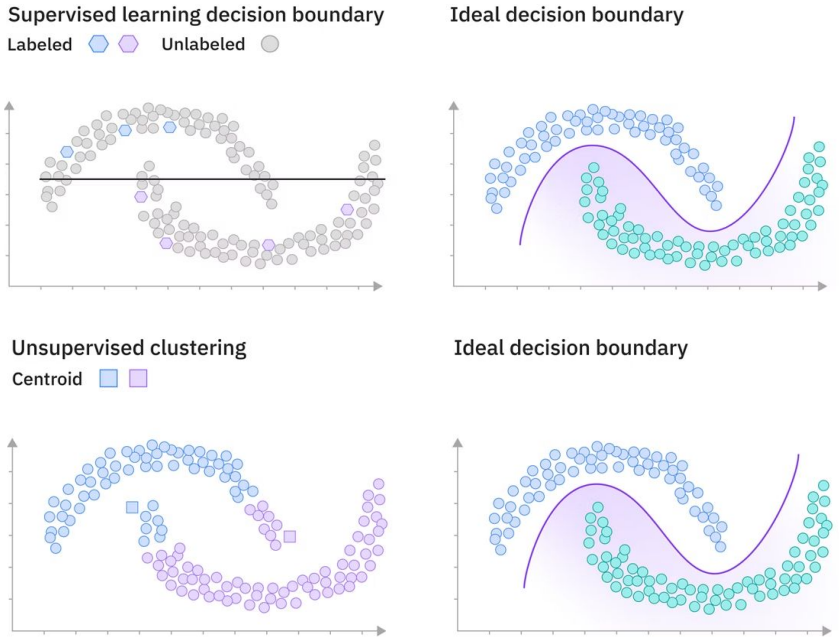
- The **input data distribution p(x)** should provide information about the **class probability p(y|x)**.
- This means the structure of the data should help determine **which class a data point belongs to**.
- SSL works best when **unlabeled data comes from the same data distribution as the labeled data**.

Ex: For an **image classifier (cats vs dogs)**, the dataset should contain **images of cats and dogs**. Unrelated images such as **horses or motorcycles** will **not help the model learn**.

Semi-Supervised Learning

- Helps train models when labeled data is limited.
- Useful when labeling data is expensive or time-consuming.

- Hybrid machine learning approach.
- Uses small labeled data + large unlabeled data.
- Lies between supervised and unsupervised learning.
- Combines supervised learning and unsupervised
- Helps reduce expensive manual labeling.
- Assumes similar inputs → similar outputs.
- Utilizes structure/patterns in unlabeled data.



- Key Techniques: Self-Training, Model trains on labeled data first,
  Then predicts labels for unlabeled data, High-confidence predictions are added to training data,
  Process repeats iteratively.

Working of Semi-Supervised Learning:

- Start with small labeled dataset.
- Add large unlabeled dataset.
- Model learns patterns from both.
- Improves prediction accuracy over time.
- Techniques in Semi-Supervised Learning:

1.Self-Training:

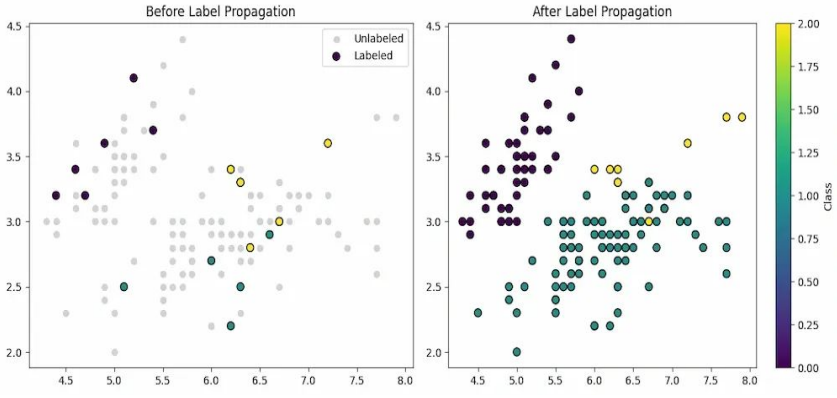
- ❖ Model first trained using labeled data.
- ❖ Predicts labels for unlabeled data.
- ❖ High-confidence predictions added to labeled dataset.
- ❖ Process repeats to improve the model.

2.Co-Training:

- ❖ Two models are trained on different feature sets.
- ❖ Each model predicts labels for unlabeled data.
- ❖ Models teach each other using their predictions.

3.Multi-View Training:

- ❖ Extension of Co-Training.
- ❖ Uses different data representations.
- ❖ Example: image + text information.
- ❖ Both models predict the same output.



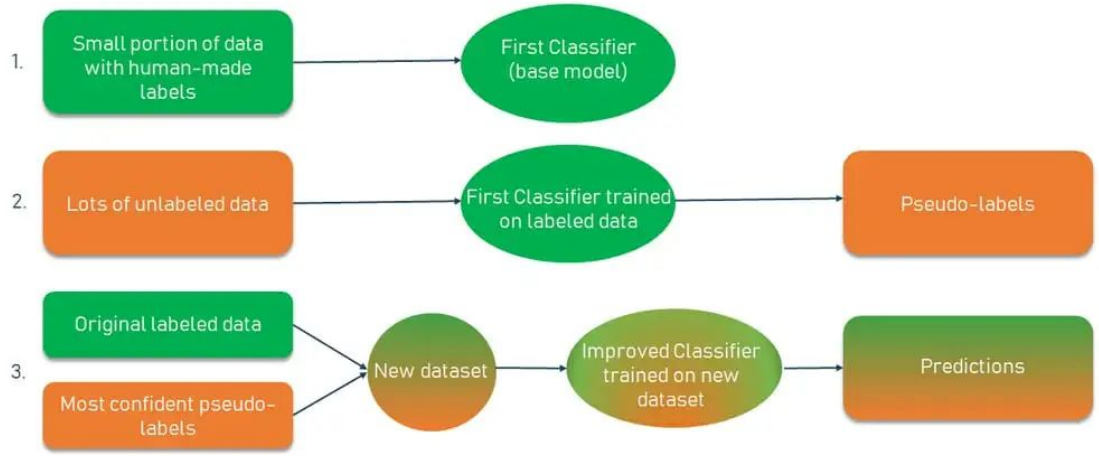
model was able to classify data into the categories or labels after successful operations of semi-supervised learning.

4. Graph-Based Models:

- ❖ Data represented as a graph structure.
- ❖ Nodes = data points.
- ❖ Edges = similarity between points.
- ❖ Labels spread from labeled nodes to unlabeled nodes.

Self-Training

- Self-training is a semi-supervised learning method.
- It modifies a supervised learning algorithm to work with labeled and unlabeled data.
- Uses a small labeled dataset and a large unlabeled dataset.
- Helps improve model performance when labeled data is limited.



semi-supervised self-training learning

Initial Training

- Select a small labeled dataset (e.g., images of cats and dogs with labels).
- Train a base model using supervised learning.
- The model learns patterns from the labeled data.

Pseudo-Labeling

- The trained model predicts labels for unlabeled data.
- These predicted labels are called pseudo-labels.
- They are not actual labels but model-generated predictions.

Selecting Confident Predictions

- Only predictions with high confidence are selected.
- Example: predictions with more than 80% confidence.
- These high-confidence pseudo-labels are added to the labeled dataset.

Model Improvement

- The combined dataset (original + pseudo-labeled) is used to train the model again.
- This improves the model's learning capability.
- The process helps use more data without manual labeling.

Iterative Process

- Self-training is repeated for multiple iterations.
- Each iteration adds more pseudo-labeled data.
- Model accuracy can improve over time if data is suitable.

Limitations

- Performance may vary depending on the dataset.
- Errors in pseudo-labels can reduce model accuracy.
- Sometimes supervised learning alone may perform better.

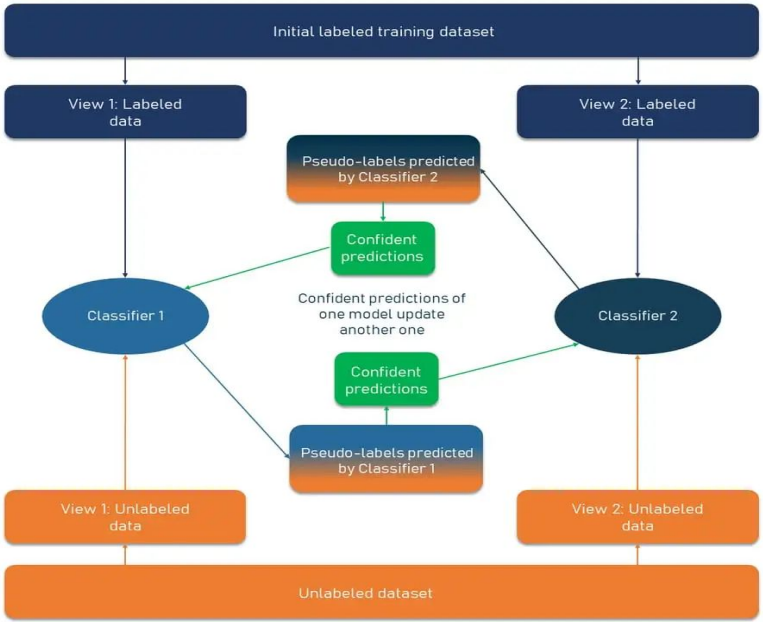
- **Co-training** is a semi-supervised learning technique.

- It is derived from self-training and considered an improved version.
- Used when only a small amount of labeled data is available.
- Uses both labeled and unlabeled data for training.

- Two separate classifiers (models) are trained.
- Each classifier uses a different view of the data.
- Both models work together to improve learning.

- Data Views mean different sets of features describing the same data.
- Each view provides additional information about the data.
- The views are independent given the class.

- Each feature set (view) is sufficient to predict the class.
- Even one view alone can classify the data correctly.
- Combining both views improves learning performance.



Ex: Used in web content classification.
A webpage can have two views:

1. Words on the webpage.
2. Anchor words in links pointing to the webpage.

Working of Co-Training:

- Train two classifiers on different feature sets.
- Each classifier predicts labels for unlabeled data.
- The most confident predictions are shared with the other classifier.
    - Both models improve by learning from each other.

Co-Training – Working

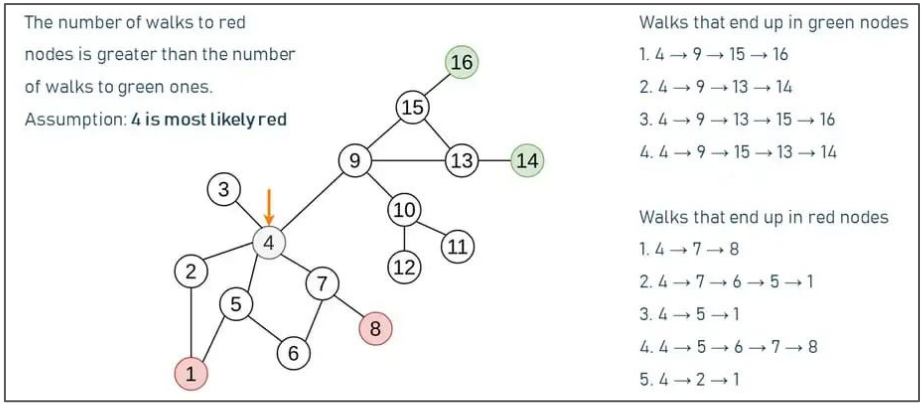
- **Step 1:** Train **two separate classifiers** using a **small labeled dataset**, each on a **different feature view** of the data.
- **Step 2:** Apply both classifiers to a **large pool of unlabeled data** to generate **pseudo-labels**.
- **Step 3:** Each classifier selects **high-confidence predictions** from the unlabeled data.
- **Step 4:** These confident pseudo-labeled samples are **shared with the other classifier** to improve its training.
- **Step 5:** Both classifiers are **updated and retrained** using the expanded labeled dataset.
- **Step 6:** The **final prediction** is obtained by **combining outputs from both classifiers**.
- **Step 7:** The process is **repeated for multiple iterations** to gradually improve model performance.

Graph-Based Models / Label Propagation

- **Graph-based SSL** represents labeled and unlabeled data as a **graph network**.
- **Nodes** represent data points and **edges** represent similarity or connection between them.
- Only a **few nodes have labels**, while most are unlabeled.

Working Process

- Labels from labeled nodes are **propagated (spread)** to nearby unlabeled nodes.
- The algorithm checks **connections or paths** between nodes.
- If a node is more connected to a certain labeled class, it **takes that class label**.



Ex: If a node has **more connections to red-labeled points than green ones**, it is classified as **red**.

Applications: Used in **personalization and recommendation systems**, Predicts **customer interests based on similar users**, Based on the **continuity assumption**: connected users often share **similar interests**.

Multi-View Learning

- **Multi-View Learning** is a technique related to **semi-supervised and unsupervised learning**.
- It uses **multiple datasets or feature sets (called views)** instead of a single dataset.
- Each **view provides a different perspective** of the same data.

Key Idea

- Multiple models can be trained, **one for each view** of the data.
- The goal is to **combine information from different views** to improve performance.

Example Approach

- A dataset is divided into **two independent views**:
  - **View 1:** First 10 features
  - **View 2:** Last 10 features

Unified Learning

- Features from both views can be **concatenated and used as a single input**.
- The model learns from **combined information of both views**.

Advantage

- Works best when **each view contains unique and useful information** without too much redundancy.

Generative Models

- **Generative models** are an important technique in **semi-supervised learning**.
- They learn the **probability distribution of the data**.
- Once the distribution is learned, the model can **generate new data and estimate class probability**.

Common Techniques

- **Generative Adversarial Networks (GANs)**
- **Variational Autoencoders (VAEs)**

Working Approach

- **Labeled data** guides the learning process.
- **Unlabeled data** helps the model learn the **structure of the input space**.
- Both types of data are used to **improve model performance**.

Example

- A **Variational Autoencoder (VAE)** learns a **latent representation** from unlabeled data.
- This representation helps **improve classification accuracy**.

Deep Semi-Supervised Learning

- **Deep Semi-Supervised Learning (Deep SSL)** combines **semi-supervised learning with deep learning architectures**.
- It uses both **labeled and large amounts of unlabeled data** to train deep models.

Key Approaches

- **GAN-based methods:** Use **generator and discriminator networks** to learn from labeled and unlabeled data.
- **Contrastive Learning:** Learns useful representations by **comparing similar and dissimilar data samples**.
- **Transformer-based Models:** Models like **BERT and GPT** process large text datasets effectively in semi-supervised settings.

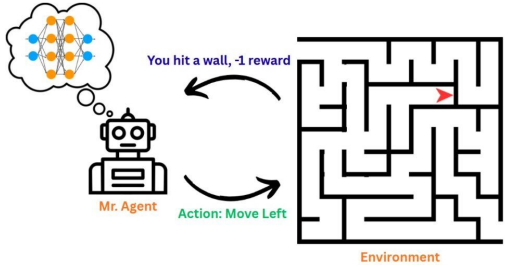
Advantages

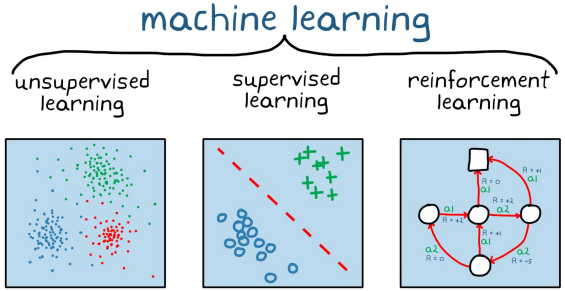
- Handles **large and complex datasets** such as text, images, and speech.
- Achieves **high performance with limited labeled data**.

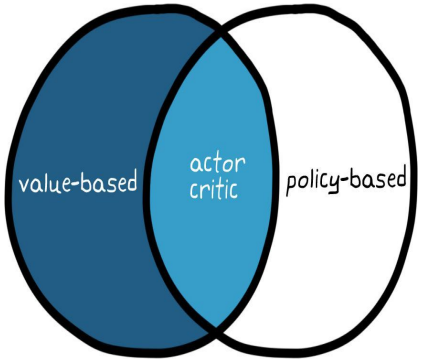
Challenges

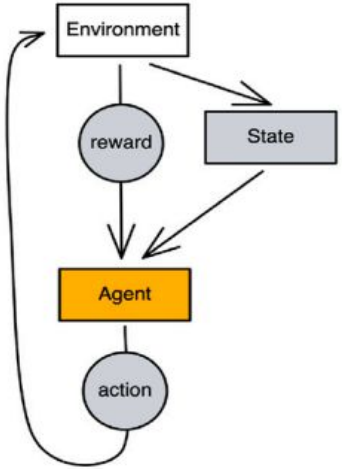
- **Training instability** in some deep models.
- Need for **accurate annotations** for labeled samples.
- **Scalability issues** with very large or imbalanced datasets.

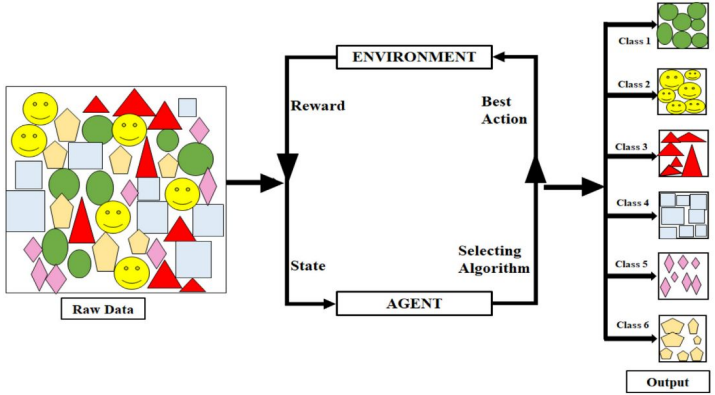
Reinforcement learning

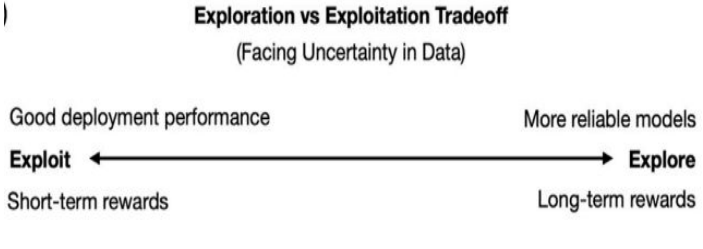


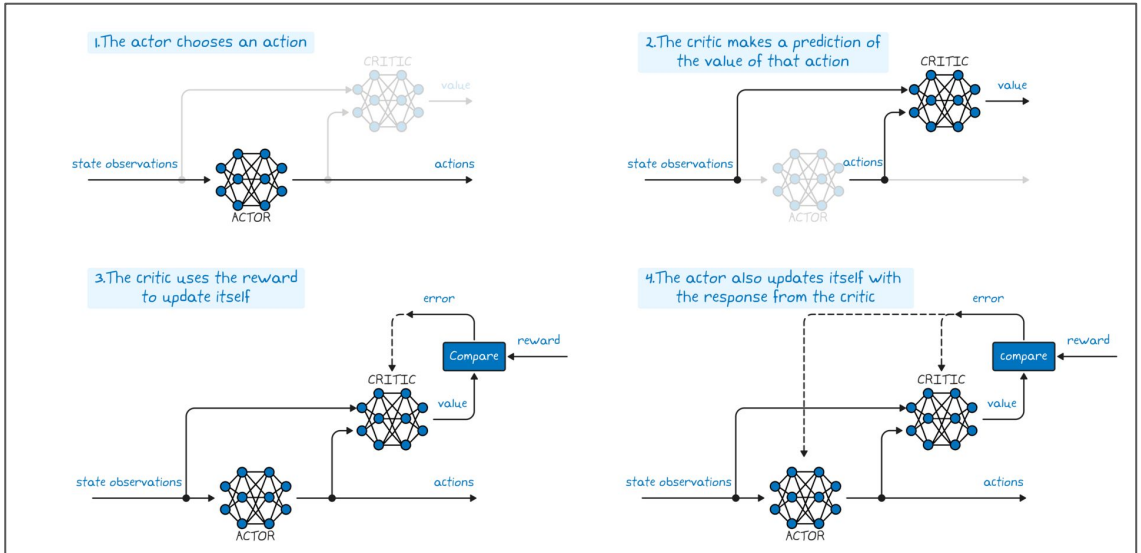






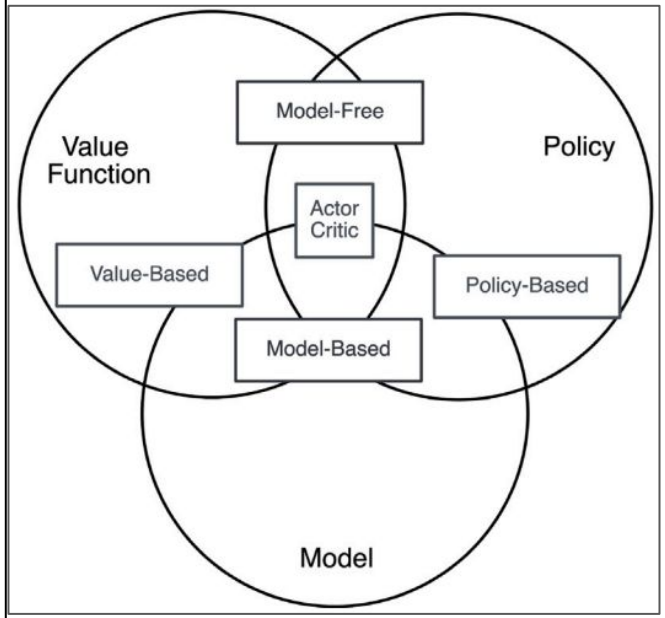




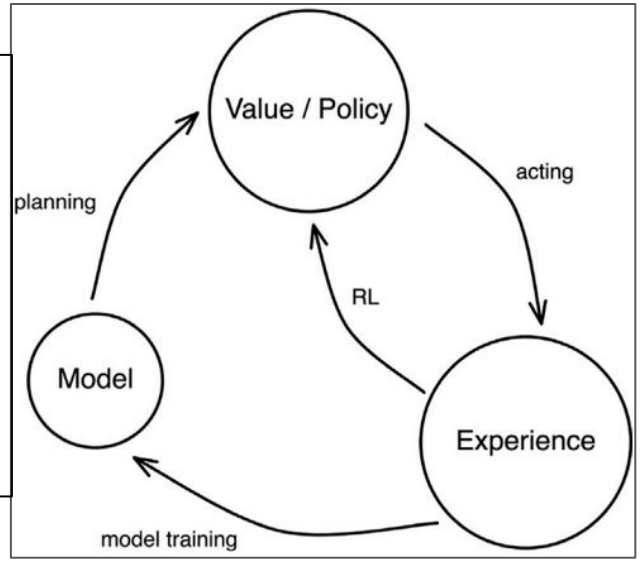


Reinforcement learning

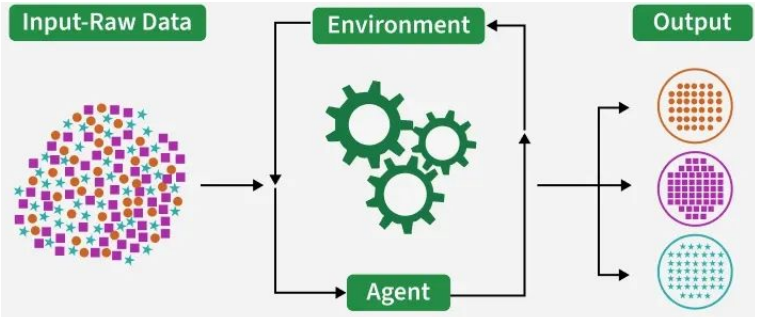
Categorization of RL approaches

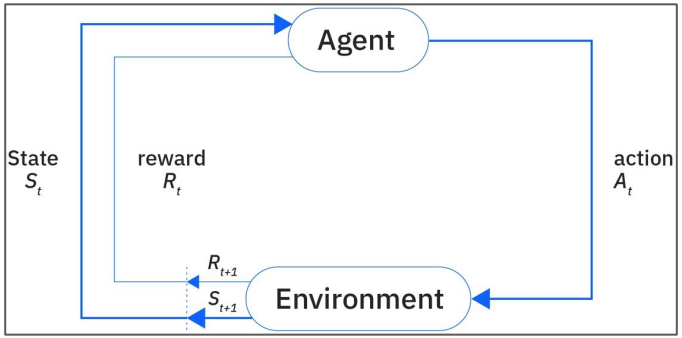


Sub-processes of RL



Reinforcement learning





- Reinforcement Learning (RL) is a branch of machine learning where an agent learns to make decisions through trial and error.
- The goal is to maximize cumulative rewards over time.
- The agent learns by interacting with an environment.
- After performing actions, the agent receives feedback in the form of rewards or penalties.
- Based on this feedback, the agent improves its decision-making policy.

Key Elements of Reinforcement Learning:

- Agent: The decision-maker in the RL system, Performs actions to achieve a goal.
- Environment: The system or world where the agent operates.
- State: The current situation or condition of the agent in the environment.
- Action: The possible decisions or moves the agent can take.
- Reward: Feedback received after performing an action, Can be positive (reward) or negative (penalty).

http://incompleteideas.net/book/RLbook2020.pdf https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf

Components of Reinforcement Learning

1. Policy

- Defines the behavior of the agent.
- Maps states to actions.
- Can be simple rules or complex algorithms.

Example: An autonomous car detects pedestrians and decides to stop or slow down.

2. Reward Signal

- Indicates the goal of the RL problem.
- Provides positive or negative feedback to guide learning.

Example: Fewer collisions, Shorter travel time, Maintaining lane discipline

3. Value Function

- Measures the long-term benefit of a state.
- Considers future rewards, not just immediate ones.

Example: A vehicle avoids risky shortcuts to maximize overall safety.

4. Model

- Represents the environment's behavior.
- Predicts future states and rewards.
- Helps the agent perform planning.

Example: Predicting movements of nearby vehicles to plan a safer path.

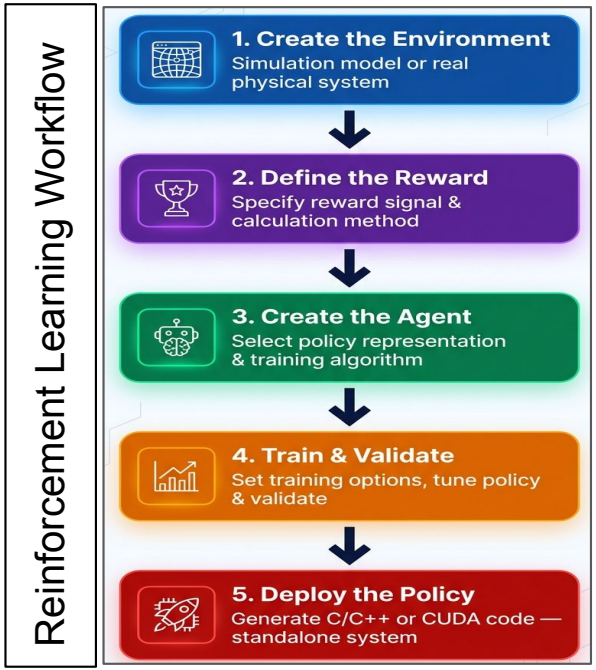
Algorithm 7 Reinforcement Learning (RL) Problem

1: **for** t = 1, 2, 3, ···, T **do**
2: Agent makes an observation $o(t)$ to the state $s(t)$
3: Agent chooses an action given the current state or observation: $a_t = \pi_t(o(t))$
4: Environment progresses to the next step s_{t+1} given the action a_t
5: Agent receives a reward feedback $r_{a_t,s_t}(t)$ for the action a_t taken on state s_t
6: Agent updates its policy π_t
7: **end for**

Working of Reinforcement Learning

The learning process follows a continuous feedback loop:

- The agent observes the current state of the environment.
- It chooses an action based on its policy.
- The environment moves to a new state.
- The agent receives a reward or penalty.
- The agent updates its knowledge (policy or value function).
- The cycle repeats to improve performance.



Exploration vs Exploitation

- Exploration: Trying new actions to discover better rewards.
- Exploitation: Using known actions that give the highest reward.
- The agent balances both to maximize cumulative rewards.

The agent balances both to maximize cumulative rewards.

- Reinforcement Learning problems are often modeled as a Markov Decision Process (MDP).

Key idea: The next state depends only on the current state and action, not on previous history.

Types of Reinforcement

1. Positive Reinforcement: Behavior increases because it produces a positive outcome.

Advantages: Improves performance, Encourages consistent behavior

Disadvantages: Overuse may reduce effectiveness

2. Negative Reinforcement: Behavior strengthens by avoiding a negative condition.

Advantages: Encourages behavior, Maintains minimum performance

Disadvantages: May only encourage minimum effort to avoid punishment

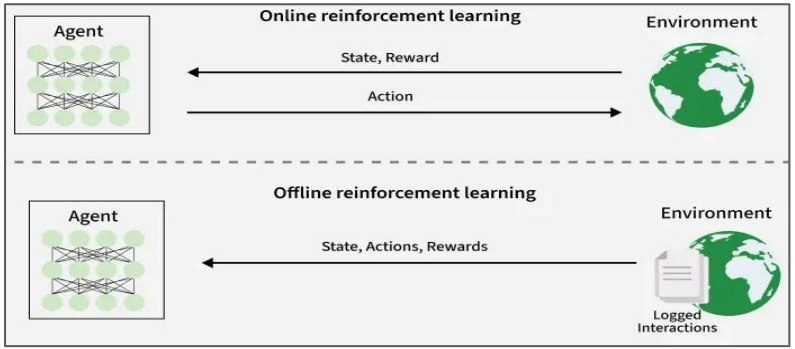
Online vs Offline Reinforcement Learning:

- **Online Reinforcement Learning:** Agent learns through direct interaction with the environment, Data is collected in real-time.

Characteristics: High adaptability, Continuous learning
Challenge: Can be resource-intensive and risky

- **Offline Reinforcement Learning:** Agent learns from pre-collected datasets, No direct interaction with the environment during training.

Characteristics: Safer for risky environments, Uses historical data
Challenge: Limited by dataset quality and coverage



Types of Reinforcement Learning

1. Model-Based Reinforcement Learning

- Agent builds an internal model of the environment.
- Predicts outcomes of actions before performing them.
- Enables planning and strategic decision-making.

Example: A robot maps a maze and plans the best path to reach the exit.

2. Model-Free Reinforcement Learning

- Agent does not build an environment model.
- Learns the best actions directly from experience and rewards.
- Focuses on trial-and-error learning.

Example: A robot learns which turns lead to the maze exit based on past rewards.

Gradient-Based vs. Evolutionary RL

Gradient-Based Algorithms:

- Mechanism: Uses backpropagation to leverage noisy estimates of the policy gradient.
- Pros: Fast learning and high sample efficiency.
- Cons: Highly sensitive to hyperparameters, prone to getting stuck in local optima, and requires a differentiable objective.

Evolutionary RL:

- Mechanism: Treats the policy as a "black box"; uses population-based mutations and selection for global searching.
- Pros: Robust against sparse or non-differentiable rewards; effective in rugged search spaces or when gradients are untrustworthy.
- Cons: Low sample efficiency and slow convergence; requires massive parallel computing power.

Hybrid Techniques:

- Combines both methods by using evolution for broad exploration and gradients for fine-tuning promising policies.

Tabular vs. Neural Network-Based RL

Tabular RL:

- Use Case: Best for environments with discrete, small state and action spaces.
- Examples: Q-learning and SARSA.

Limitation: Becomes infeasible as state/action pairs increase (the "Curse of Dimensionality").

Neural Network-Based RL:

- Use Case: Necessary for large, complex, or infinite state/action spaces.
- Function: Acts as a function approximator to represent policy parameters where tables cannot.
- Status: The standard for most modern, high-complexity RL problems.

Single-Agent vs. Multi-Agent RL (MARL)

Single-Agent RL:

- Characteristics: Only one agent interacts with the environment.
- Advantages: Learning is stable, simple to implement, and easy to analyze.

Multi-Agent RL (MARL):

Characteristics: Multiple agents interact simultaneously; agents influence one another.

Advantages: Can solve highly complex tasks and foster emergent behaviors like coordination or negotiation.

Challenges: * N**on-stationari**ty: The environment changes as agents learn, violating Markov assumptions.

Instability: Independent updates by agents can lead to training failure.

Resources: High computational demand and difficult convergence.

Reinforcement Learning Techniques

- Q-Learning: Model-free method that stores Q-values for each state–action pair to find the best action.
- SARSA: Similar to Q-learning but updates values based on the action actually taken (on-policy learning).
- Deep Q-Network (DQN): Uses a deep neural network to estimate Q-values for very large state spaces.
- Policy Gradient Methods: Learn a probability-based policy directly and adjust it to maximize rewards.

Methods of Reinforcement Learning

Dynamic Programming (DP)

- Breaks larger tasks into smaller sub-tasks.
- Models problems as workflows of sequential decisions made at discrete time steps.
- Each decision is made based on possible next states.
- The reward is not just one fixed value — it's a weighted average across all reachable next states.
- Each next-state reward r□(s,a,s') is weighted by its transition probability p□(s'|s,a).
- This makes DP model-based → it needs to know transition probabilities.
- The agent uses this formula to evaluate which action a is most rewarding from any state s.

Reward Function

$$r_t(s,a) = \sum_{s' \in S} r_t(s,a,s') \cdot p_t(s'|s,a)$$

Symbol	Meaning
$r_t(s, a)$	Expected reward at time t, given state s and action a
$\Sigma\ s' \in S$	Sum over **all possible next states** s' in state space S
$r_t(s, a, s')$	Reward received when transitioning from state s → s' via action a
$p_t(s'\|s, a)$	**Probability** of reaching next state s' from s by taking action a

Bellman Equation

$$v_t(s) = \max_{a \in A(s)} \left\{ r_t(s,a) + \sum_{s' \in S} p(s'|s,a) \cdot v_{t+1}(s') \right\}$$

- It is recursive → the value of the current state depends on the value of the next state.
- The agent picks the action that maximizes the combined immediate + future reward.
- This is the backbone of DP's optimal policy search.
- The agent maximizes its value function by consistently choosing high-reward actions across successive states.
- The Reward Function computes the expected reward for a single action.

Symbol	Meaning
$v_t(s)$	**Value function** — total expected reward from state s at time t onward
$\max_{a \in A(s)}$	Choose the action a from available actions $A(s)$ that **maximizes** reward
$r_t(s,a)$	**Immediate reward** at time t for taking action a in state s
$\sum p(s'\|s,a) \cdot v_{t+1}(s')$	**Discounted future reward** — weighted sum of expected future state values
$v_{t+1}(s')$	Value of the **next state** s' at time $t+1$ (recursive component)

- The Bellman Equation uses that reward + recursive future values to find the optimal long-term policy.
- The Bellman Equation splits total reward into two parts:
  - ✅ **Immediate reward** → $r_t(s,a)$
  - ✅ **Future reward** → $\sum p(s'|s,a) \cdot v_{t+1}(s')$

Monte Carlo (MC) Method

- Model-free → assumes a black-box environment.
- Learns exclusively from experience (trial and error).
- Samples sequences of states, actions, and rewards through direct interaction.
- Value function = average of all returns for each state–action pair.
- Must wait until end of episode before updating the policy.
- No use of transition probabilities p(s'|s,a) ← key difference from DP.

Temporal Difference (TD) Learning

- Hybrid of DP + Monte Carlo.
- From DP → updates policy after each step (no need to wait for episode end).
- From Monte Carlo → learns through raw environment interaction (model-free).
- Updates based on difference between predicted and actual rewards.

Feature	SARSA	Q-Learning
Policy Type	On-policy	Off-policy
Policies Used	Single policy	Target (exploit) + Behavior (explore)

Feature	Dynamic Programming	Monte Carlo	TD Learning
Model Type	Model-based	Model-free	Model-free
Uses p(s'|s,a)	✅ Yes	❌ No	❌ No
Learning Style	Predictive	Trial & Error	Hybrid
Update Timing	Each step	End of episode	Each step
Core Formula	Bellman Equation	Return Average	TD Error

Markov Decision Process (MDP)

- mathematical framework used for decision-making in uncertain environments.
- It describes how an agent interacts with an environment to achieve a goal.
- The agent chooses actions and receives rewards based on the results.
- MDPs help determine the best sequence of actions to maximize total reward.
  - An MDP consists of **five main components**: **States (S), Actions (A), Transition Model (T), Reward ®, Policy (π)**
    These components define how the agent interacts with the environment.

❖ A **state** represents the current situation of the agent.
- It describes where the agent is or what condition it is in.
  Example: Position in a grid world like (1,1) or (2,3).
- The agent moves from one state to another after performing actions.

❖ **Actions** are the possible moves the agent can take.
- Each state may have one or multiple possible actions.
- Actions cause the agent to change from one state to another.
  Example actions: Move UP, Move DOWN, Move LEFT, Move RIGHT

❖ **transition model** describes what happens after an action.
- It gives the probability of moving from one state to another.
- Sometimes actions are uncertain (stochastic).
  Example: 80% chance of moving in intended direction, 10% chance of slipping to the left, 10% chance of slipping to the right,
  This randomness is called stochastic transition.

❖ A **reward** is a numerical value given after performing an action.
- It tells the agent whether the action result is good or bad.
- Rewards help the agent learn better actions.
  Examples: +1 → Reaching the goal, -1 → Stepping into fire or danger, -0.1 → Small penalty for each step

❖ A **policy (π)** is the strategy or plan followed by the agent.
- It tells the agent which action to take in each state.
- Find the optimal policy that maximizes total reward over time.
  Example: If state = (1,1) → move RIGHT, If state = (1,2) → move UP

Types of Policies:
- **Deterministic Policy** (π(s) = a): Always selects the same action for a given state.
- **Stochastic Policy** (π(a|s)): Selects an action probabilistically based on a distribution.

Markov Decision Process (MDP)

MDP is defined as:

$$MDP = \{S, A, P, R, \gamma\}$$

Goal: maximize cumulative discounted reward

Components

S – States

- All possible situations the **agent can be in**.

A – Actions

- Set of actions available in each state.

P – Transition Probability

- $P(s'|s, a)$
- Probability of reaching state **s'** from **s** after action **a**.

R – Reward

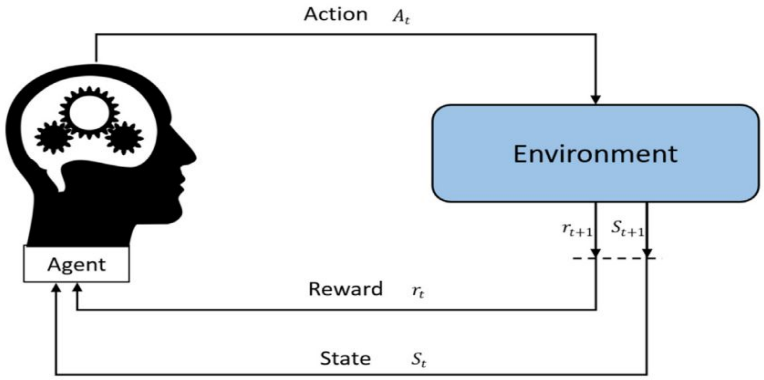
- Immediate feedback received after transition.

γ – Discount Factor

- $0 \leq \gamma \leq 1$
- Determines importance of **future rewards** vs **immediate rewards**.

Markov Property

- **Next state depends only on:**
  - Current state
  - Current action
- **Not dependent on past history.**



States:	S
Model:	T (S, a, S') ~ P(S' | S,a)
Actions:	A(S), A
Reward:	R(S), R(S, a), R(S, a, S')
Policy:	Π(S) → a Π*

- MDP is used for sequential decision-making problems.
- Helps agents make optimal decisions under uncertainty.
- Widely used in AI, robotics, games, and reinforcement learning.

Solving Markov Decision Processes

- MDPs are solved using Reinforcement Learning techniques.
- The goal is to find the optimal policy that maximizes total rewards.
- Three main approaches are used: Dynamic Programming (DP), Monte Carlo Methods, Temporal-Difference (TD) Learning
- **Dynamic Programming (DP)** solves MDPs using Bellman Equations.
- It breaks the problem into smaller subproblems and solves them iteratively.
- Requires full knowledge of the environment model (transition probabilities and rewards).

$$V(s) = \sum_{s'} P(s'|s,a)\,[R(s,a,s') + \gamma V(s')]$$

V(s) = value of state *s,*
P(s'|s,a) = probability of moving to state *s',*
R(s,a,s') = reward received,
γ (gamma) = discount factor for future rewards

Dynamic Programming Methods

1. Policy Iteration

Two main steps:

Policy Evaluation: Calculate value function $V(s)$ for current policy.
Policy Improvement: Update policy to choose better actions.

2. Value Iteration

- Directly updates the value function.
- Faster because it skips full policy evaluation.
- Gradually finds the optimal policy.

Limitation: Works well only for small state spaces, Becomes slow when the environment is very large.

Monte Carlo Methods

- Monte Carlo (MC) methods learn from actual experience (episodes).
- They do not require transition probabilities.

How it works:

- Agent runs many episodes.
- Collects rewards from each episode.
- Updates value estimates based on averages.

Strengths

- No environment model needed.
- Works well for large and complex problems.

Limitations

- Requires many episodes to learn.
- Cannot update values until the episode ends.

Monte Carlo

- Waits for **complete episode**
- Uses **actual return** G_t
- Update rule:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

Temporal Difference

- Updates **after each step**
- Uses **estimated return (TD target)**

$$R_{t+1} + \gamma V(S_{t+1})$$

Method	Update Time	Return Used
Monte Carlo	End of episode	Actual return (G_t)
TD Learning	Every step	Estimated return

Temporal-Difference (TD) Learning

- TD Learning combines ideas from Dynamic Programming and Monte Carlo.
- Updates value estimates after every step, not after the entire episode.
- Allows faster learning.
- Used widely in real-time AI systems.
- TD Learning is widely used in deep reinforcement learning, helping AI agents learn optimal behaviors in real-time applications like autonomous driving and robotic control.

- Monte Carlo learning waits until the episode finishes before updating the value of a state.
- An episode means a full sequence from start state → terminal state.

Example:

- Cat tries to catch the mouse in a grid.

State (S_t)	Action (A_t)	Reward (R_{t+1})	Next State (S_{t+1})

After the episode ends, the agent calculates the return G_t

Return Formula:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots$$

This is the total reward from time t until the episode ends.

Example: $G_0 = 1+0+0+0+0+0+1+1+0+0 = 3$ So the total reward = 3.

Value Update Rule (Monte Carlo)

New value = Old value + Learning correction.

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

- $V(S_t)$: New value of state t
- $V(S_t)$: Former estimation of value of state t (= Expected return starting at that state)
- α: Learning Rate
- G_t: Return at timestep t
- $V(S_t)$: Former estimation of value of state t (= Expected return starting at that state)

$G_t - V(S_t)$:Error in prediction

- The agent adjusts its estimate using the actual return observed at the end of the episode.

Key Characteristics of Monte Carlo:

• Learns only after episode finishes • Uses actual total reward (Gt) • Requires complete episode

• Works well when episodes naturally end (games, simulations)

Temporal Difference (TD) Learning

- Temporal Difference Learning updates the value function at every time step.
- It does not wait for the entire episode to finish.

The update uses:

- Immediate reward R_{t+1}
- Estimated value of next state $V(S_{t+1})$

- Main idea: Learn incrementally while interacting with the environment.

TD Learning Update Rule

New Value = Old Value + Adjustment based on error

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

New value of state t

Former estimation of value of state t

Learning Rate

Reward

Discounted value of next state

TD Target

TD Learning does not know the full return G_t because the episode is not finished.

Instead it estimates the return using:

$$\text{TD Target} = R_{t+1} + \gamma V(S_{t+1})$$

- This estimate is used to update $V(S_t)$

Learning Process

- *The agent continues interacting with the environment.*
- *After each step: It updates the value function.*
- *Over time: The value estimates become more accurate.*
- *The agent gradually learns better behavior.*

- TD learning uses **bootstrapping**.
- Bootstrapping means:
  - **Updating estimates using other estimates.**
- Instead of waiting for actual return G_t, TD uses:
  - **Estimated value of the next state $V(S_{t+1})$.**
- Because of this:
  - TD learning can **learn online**.
  - Updates happen **after every step**.

- The basic TD method is called **TD(0)**.
- It performs **one-step updates**.
- Update occurs after **each state transition**:

$$(S_t, A_t, R_{t+1}, S_{t+1})$$

Steps:

1. Observe reward R_{t+1}
2. Observe next state S_{t+1}
3. Compute TD target
4. Update $V(S_t)$

Updating value of **State S_0**:

Given:

- $V(S_0) = 0$
- Learning rate $\alpha = 0.1$
- Reward $R_1 = 1$
- Discount $\gamma = 1$
- $V(S_1) = 0$

Calculation:

$$V(S_0) = 0 + 0.1[1 + 1 \times 0 - 0]$$

Result:

- **New $V(S_0) = 0.1$**

So the **value function for State 0 is updated to 0.1**.

Algorithms Based on MDP:

Several reinforcement learning algorithms rely on MDPs, including:

- **Q-Learning** – learns the value of actions in each state.
- **Deep Q-Networks (DQN)** – uses deep neural networks to estimate Q-values.
- **SARSA** – updates values based on actions taken by the policy.

These algorithms help AI systems learn optimal strategies through experience.

- Markov Decision Processes (MDPs) are widely used in Artificial Intelligence.
- They help AI systems make sequential decisions in uncertain environments.
- MDPs provide a mathematical framework for learning the best actions over time.
- Many modern AI systems use MDPs as their core decision-making model.
- Reinforcement Learning (RL) is built on the concept of MDPs.
- RL allows an AI agent to learn by interacting with the environment.
- The agent performs actions, receives rewards, and improves its strategy.
- The goal is to maximize long-term cumulative rewards.

Q-Learning

- Q-Learning is a model-free reinforcement learning algorithm.

Update rule:

$$Q(s,a) = Q(s,a) + \alpha[R + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

Where:

Q(s,a) = value of taking action a in state s, α (alpha) = learning rate, R = reward received,
γ = discount factor, max Q(s',a') = best future reward

Applications: Robotics, Autonomous systems, AI games

SARSA Algorithm (State→Action→Reward→State→Action):

- Similar to Q-learning but follows the current policy.
- Updates Q-values based on the actual action taken.

Difference:

- Q-learning: Uses the maximum possible future reward.
- SARSA: Uses the reward from the action actually chosen.

Algorithm 8 Q-Learning Algorithm

1: **Initialize:** $Q_{s,a} = 0 \forall s \in S, \forall a \in A$
2: **for** each episode e **do**
3: Initialize state s
4: **Repeat** for each step t of the episode e
5: Take action $a = \arg\max_{a'} Q(s,a')$, and
6: Observe $s' \in S$, $r \in R(s)$
7: $s = s'$
8: $Q(s,a) := \hat{Q}(s,a) + \alpha_t(r + \gamma \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a))$
9: **until** s is the terminal state
10: **end for**

Bellman Equations

- The Bellman Equation is used in Reinforcement Learning to calculate the value of a state.
- It states that the value of a state = immediate reward + expected value of the next state.
- Helps an agent choose actions that maximize long-term rewards.
- Based on the Principle of Optimality.

Principle of Optimality:
The optimal value of a state depends on the immediate reward and the optimal value of the next state.

Bellman Equation for State Value Function

- The state value function represents the expected total reward from a state while following a policy.

$$V^{\pi}(s) = E[R(s,a) + \gamma V^{\pi}(s')]$$

Expanded form:

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi}(s')]$$

Where: V(s) → value of state s, P(s'|s,a) → transition probability, R(s,a,s') → reward received,
γ (gamma) → discount factor, π(a|s) → probability of taking action a

Bellman Equation for Action Value Function (Q-Function)

- The Q-function represents the expected reward for taking action a in state s.
- Helps calculate future rewards for state–action pairs.
- Used in many reinforcement learning algorithms.

$$Q^{\pi}(s,a) = E[R(s,a) + \gamma V^{\pi}(s')]$$

Expanded form:

$$Q^{\pi}(s,a) = \sum_{s'} P(s'|s,a)[R(s,a) + \gamma \sum_{a'} \pi(a'|s') Q^{\pi}(s',a')]$$

Bellman Optimality Equations

- When using the optimal policy (π*), the equations become:
- These equations help find the best possible policy.

Optimal State Value Function

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a) + \gamma V^*(s')]$$

Optimal Action Value Function

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a) + \gamma \max_{a'} Q^*(s',a')]$$

MDPs can be solved using Dynamic Programming methods based on Bellman equations.

1. Value Iteration:
   - Uses the Bellman Optimality Equation.
   - Updates state values repeatedly.
   - Stops when values converge (stabilize).
2. Policy Iteration

Two steps:

- Policy Evaluation – calculate state values using Bellman expectation equation.
- Policy Improvement – update policy based on new values.

Q-Learning uses the Bellman Optimality Equation for Q-values.

$$V(s) = \max_a [R(s,a) + \gamma V(s')]$$

Where:

V(s) = value of current state, R(s,a) = immediate reward, γ (gamma) = discount factor (0–1), V(s′) = value of next state, $\max_a$ = choose action with highest value, This helps the agent select the best action.

Bellman Equations

State Value Function $V(s)$

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)[R + \gamma V_\pi(s')]$$

Meaning:

- Expected return starting from **state s**
- Following **policy π**

Key Points:

- Recursive definition.
- Value = **immediate reward + discounted future value**.

Optimal State Value:

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a)[R + \gamma V^*(s')]$$

Action Value Function $Q(s,a)$

$$Q_\pi(s,a) = \sum_{s'} P(s'|s,a)[R + \gamma \sum_{a'} \pi(a'|s') Q_\pi(s',a')]$$

Meaning:

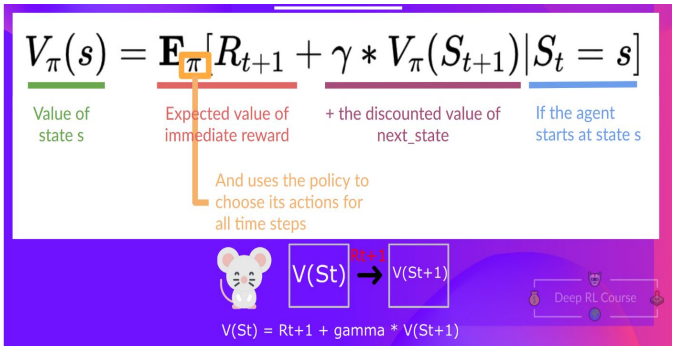
- Expected return if:
  - Start in state **s**
  - Take action **a**
  - Then follow policy **π**

Optimal Q-Function:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[R + \gamma \max_{a'} Q^*(s',a')]$$

Relationship:

$$V^*(s) = \max_a Q^*(s,a)$$



Example: **Grid-world game**

- Agent moves through grid cells.
- Each move has reward = -1.
- Goal state has positive reward.

Bellman Equation:

- Combines current reward with discounted value of next cell.

- Helps the agent find the shortest path to the goal.

A. Shows how state value is calculated from action values.

Process:

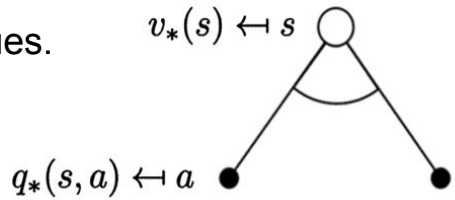
Start from state s.

Look at all possible actions.

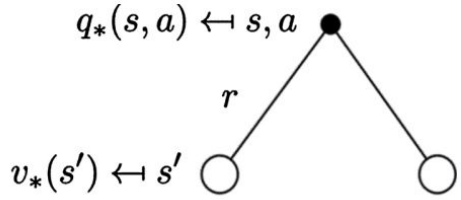
Choose the action with maximum Q-value.

Result:

$$v_*(s) = \max_a q_*(s,a)$$



a



b

B. Shows how action value depends on next state values.

Process:

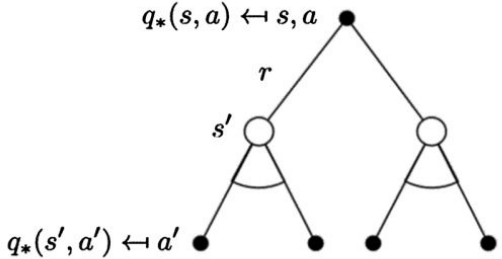
Agent takes action a in state s.

Receives reward r.

Moves to next state s′.

Uses value of next state

This calculates q*(s,a).



c

Backup diagrams illustrate the relationship between:

- State value function $v_*(s)$
- Action value function $q_*(s,a)$

C. Combines state value and action value relationships.

After reaching next state s′, the agent again chooses the best action.

Process:

Take action a.

Receive reward r.

Move to next state s′.

Choose max action value in s′.

This gives the Bellman Optimality Equation.

Policy Evaluation – Monte Carlo Method

Core Idea

- Learn V^π by **averaging returns from complete episodes**.
- **No environment model required**.

Steps

1. **Generate Episode**
   - Follow policy π until terminal state.
   - Example:
     $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$
2. **Compute Returns**
   - For time step t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots$$

3. **First-Visit MC**
   - Record return **only on first visit** to state in an episode.
4. **Update Value**

$$V(s) = average(G_t)$$

5. **Convergence**

- As number of episodes → ∞
- $V^\pi(s)$ approaches **true value**.

Pros

- Model-free.
- Unbiased estimates.

Cons

- Requires **complete episodes**.
- **High variance**.

- Learn value by averaging complete episode returns
- Model-free
- Unbiased but high variance

Policy Iteration vs Value Iteration

Policy Iteration

Steps:

1. **Policy Evaluation**
   - Compute V^{π} using Bellman equation.
2. **Policy Improvement**

$$\pi'(s) = argmax_a \sum_{s'} P(s'|s,a)[R + \gamma V^{\pi}(s')]$$

3. **Repeat**
   - Update policy until **no change**.

Properties:

- Guaranteed convergence for **finite MDPs**.
- Evaluation step can be **computationally expensive**.

Value Iteration

Combines **policy evaluation + improvement**.

Update Rule

$$V(s) = \max_a \sum_{s'} P(s'|s,a)[R + \gamma V(s')]$$

Steps:

1. Initialize V(s)
2. Update values for all states repeatedly
3. Stop when change $\Delta V < \varepsilon$
4. Extract optimal policy:

$$\pi^*(s) = argmax_a \sum_{s'} P(s'|s,a)[R + \gamma V^*(s')]$$

Advantages

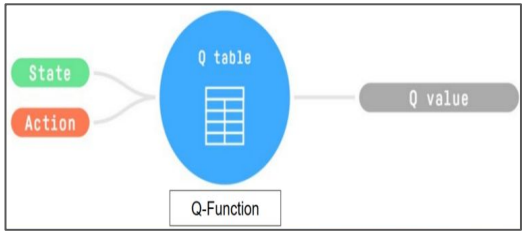
- Faster convergence.
- No need to track policy during iteration.

Q-Learning (Reinforcement Learning)

The link between Value and Policy:

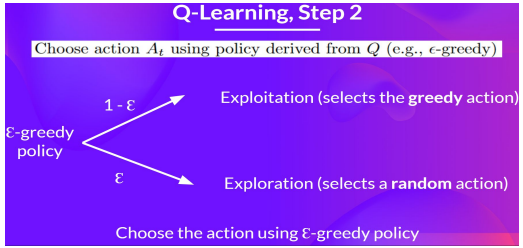
$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

Finding an **optimal value function** leads to having an **optimal policy.**



Algorithm 14: Sarsamax (Q-Learning)

Input: policy π, positive integer *num_episodes*, small positive fraction α, GLIE $\{\epsilon_i\}$
Output: value function Q ($\approx q_\pi$ if *num_episodes* is large enough)
Initialize Q arbitrarily (e.g., $Q(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(\textit{terminal-state}, \cdot) = 0$)
for $i \leftarrow 1$ *to* *num_episodes* **do** ← Step 1
  $\epsilon \leftarrow \epsilon_i$
  Observe S_0
  $t \leftarrow 0$
  repeat
    Choose action A_t using policy derived from Q (e.g., ϵ-greedy) Step 2
    Take action A_t and observe R_{t+1}, S_{t+1} Step 3
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$ Step 4
    $t \leftarrow t + 1$
  until S_t *is terminal*;
end
return Q



https://huggingface.co/learn/deep-rl-course/en/unit2/q-learning

1. Step 1: Initialize the Q-Table

- Q-learning starts by creating a **Q-table**.
- The table stores **Q-values for each state–action pair**.
- Usually all values are **initialized to 0**.
- Formula representation:

$$Q(s,a) = 0 \quad \forall s \in S, a \in A(s)$$

- **Terminal states** are also initialized with value **0**.
- Purpose: provides a starting point for learning.

2. Step 2: Choose an Action (ε-Greedy Strategy)

- The agent selects an action using **epsilon-greedy policy**.
- This balances **exploration and exploitation**.

Exploitation

- Probability = **1 – ε**
- Agent chooses the **best known action** with highest Q-value.

Exploration

- Probability = **ε**
- Agent chooses a **random action**.

Important Points

- Initially **ε is large** → more exploration.
- As training progresses **ε decreases**.
- Later the agent mostly **exploits learned knowledge**.

3. Step 3: Perform the Action

- Agent performs selected action A_t.
- Environment returns:
  - **Reward** R_{t+1}
  - **Next state** S_{t+1}

So the agent:

1. Takes action
2. Receives reward
3. Moves to next state

4. Step 4: Update the Q-Value

The Q-table is updated using the **Q-learning update rule**.

Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]

Meaning of Each Term

- $Q(S_t, A_t)$ → Current Q-value
- **α (alpha)** → Learning rate
- R_{t+1} → Immediate reward
- **γ (gamma)** → Discount factor
- $\max_a Q(S_{t+1}, a)$ → Best possible future reward

TD Target

$$R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$$

TD Error

Difference between **target value and current Q-value**.

5. Information Required to Update Q-Value

To update $Q(S_t, A_t)$, we need:

- Current state S_t
- Action taken A_t
- Reward received R_{t+1}
- Next state S_{t+1}

6. Off-Policy vs On-Policy Learning

Off-Policy (Q-Learning)

- **Different policy for acting and updating.**
- Acting policy: **ε-greedy**
- Updating policy: **greedy max Q**

Update uses:

$$\gamma \max_a Q(S_{t+1}, a)$$

On-Policy (SARSA)

- **Same policy used for acting and updating.**
- Uses the next action actually taken.

Update term:

$$\gamma Q(S_{t+1}, A_{t+1})$$

Q-Learning (Reinforcement Learning)

- model-free reinforcement learning algorithm that learns the optimal action for each state by estimating the Q-value (action-value function).

Key Properties:

- **Off-policy method**
  - The agent learns the optimal policy independent of the policy used to explore.
- **Value-based method**
  - It learns a value function (Q-function) instead of directly learning the policy.
- **Temporal Difference (TD) learning**
  - The value is updated after every step, not only at the end of an episode.

Q-Function (Action-Value Function)

The Q-function represents the expected cumulative reward when:

- The agent is in state s
- Takes action a
- Then follows the optimal policy **Q(s,a)**

It measures the **quality of taking action a in state s**.

State	Left	Right	Up	Down
S1	Q(S1,Left)	Q(S1,Right)	Q(S1,Up)	Q(S1,Down)

- Internally, the Q-function is stored in a Q-table.
- Each cell contains the expected reward of taking that action in that state.

Initially:

Q(s,a) = 0

As the agent explores, these values are updated.

Value vs Reward

- Reward: **Immediate feedback from environment**

Example: +10 for reaching goal

- Value: **Expected cumulative future reward**

Example:

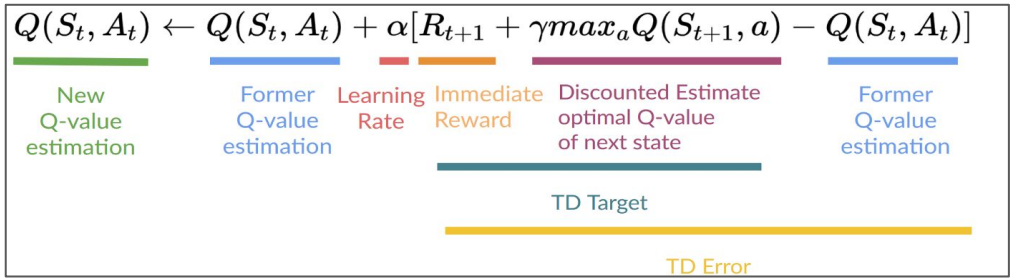
Value = immediate reward + future rewards

- **New Q value = Old Q value + Learning rate × (Target − Current estimate)**

  Where: Target = Reward + Discounted future reward

The core of Q-learning is the update equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$



The optimal policy is derived from the Q-function.

$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

Meaning:

Choose the action with the highest Q-value in that state.

Q-Learning (Model-Free RL)

Q-Learning Update Equation

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

Key Concepts

- **Off-policy algorithm** – learns optimal policy Q^* independent of behavior policy
- **Model-free** – no transition probability $P(s'|s,a)$ required
- **α (alpha)** – learning rate controlling update speed
- **γ (gamma)** – discount factor for future rewards
- Uses **ε-greedy exploration** (random action with probability ε)
- **TD Error** = $R + \gamma \max Q(s',a') - Q(s,a)$
- Converges to optimal policy with enough exploration

Q-Learning Algorithm Steps

1. **Initialize** Q(s,a) = 0 for all states and actions
2. **Observe** current state s
3. **Select action** a using ε-greedy from Q(s,:)
4. **Perform action** and observe reward R and next state s'
5. **Update Q-value** using Q-learning formula
6. **Set** $s = s'$
7. **Repeat** until convergence

Advantages of Q-Learning:
✔ Simple and easy to implement
✔ Model-free (does not require environment model)
✔ Converges to optimal policy
✔ Works well for discrete state spaces

Limitations:
❌ Q-table becomes huge for large state spaces
❌ Slow learning in complex environments

Solution:
Deep Q-Learning (DQN) using neural networks.

Algorithm Steps

1. **Initialize Q-table:**
   $Q(s,a) = 0$
2. **Observe Current State:**
   S_0
3. **Choose Action (ε-greedy):**
   - **Explore:** probability ε
   - **Exploit:** choose best action with probability $1 - \varepsilon$
4. **Take Action:**
   Environment returns **Reward** R and **Next State** S'
5. **Update Q-value:**
   Use the Q-learning update rule to improve the Q-table.
6. **Move to Next State:**
   $S \leftarrow S'$
   Repeat until **terminal state**.

Training Process

Before Training (Initial Q-table)

State	A1	A2	A3	A4
S1	0	0	0	0
S2	0	0	0	0
S3	0	0	0	0

After Training (Learned Q-table)

State	A1	A2	A3	A4
S1	0	10.8	0	0
S2	0	9.9	0	-10
S3	0	0	0	10
S4	0	-10	0	0

The **agent learns the best action in each state**.

Why Q-Learning Works

The algorithm repeatedly updates the action values:

$$Q(s,a) \rightarrow Q^*(s,a)$$

Once the **optimal Q-function** Q^* is learned, the **optimal policy** is obtained by choosing the action with the highest Q-value.

SARSA (State-Action-Reward-State-Action)

SARSA Update Equation

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma Q(s',a') - Q(s,a)]$$

SARSA Structure

S → A → R → S' → A'

SARSA Key Points

- On-policy algorithm
- Uses actual next action a' chosen by the policy
- Action a' is selected before the update
- Learns value of the current policy
- More conservative than Q-learning
- Penalizes risky actions
- Suitable for safety-critical environments

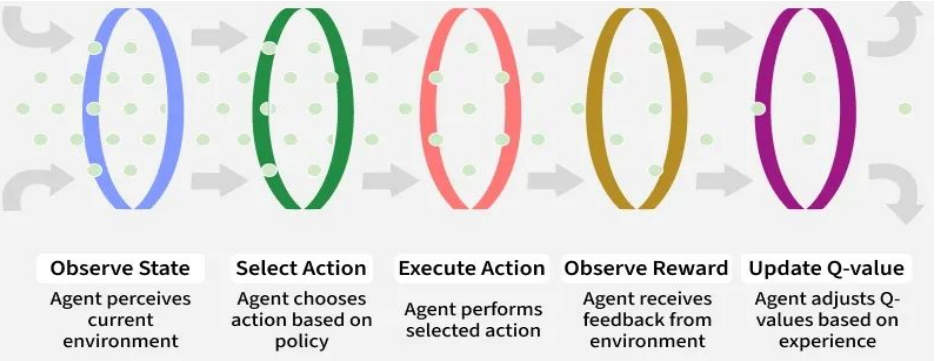
5 Components (S-A-R-S-A)

1. **State (s_t):** The current situation.
2. **Action (a_t):** The decision made in the current state.
3. **Reward (r_{t+1}):** Feedback received after the action.
4. **Next State (s_{t+1}):** The new position after the action.
5. **Next Action (a_{t+1}):** The action planned for the next state (crucial for the update).

Feature	SARSA	Q-Learning
Policy type	On-policy	Off-policy
Update uses	Q(s',a')	max Q(s',a')
Risk behavior	Conservative	Aggressive
Exploration effect	Considered	Ignored
Action selection	Required before update	Not required
Best use	Safe environments	Offline/batch learning

SARSA (State-Action-Reward-State-Action) in Reinforcement Learning

On-Policy RL: SARSA learns from the actions it actually takes, including exploratory (random) moves.
Goal: To learn an optimal policy by interacting with the environment and updating behavior based on real-time feedback.



Realism: Safer and more grounded than off-policy methods because it accounts for the risks of exploration.

- **Immediate Reward:** The agent gets reward r_{t+1} after taking action a_t in state s_t.
- **Future Reward:** It uses $Q(s_{t+1}, a_{t+1})$ to estimate future returns.
- **Correction:** The Q-value is adjusted based on the difference between expected and actual rewards.

To balance finding new paths vs. using known ones:

- **Exploration (with probability ϵ):** Choose a random action to discover the environment.
- **Exploitation (with probability $1 - \epsilon$):** Choose the action with the highest known Q-value.
- **Decay:** ϵ usually decreases over time as the agent becomes "smarter."

The Update Rule (Bellman Equation)

SARSA adjusts the Q-value using this formula:

$$\boxed{Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]}$$

- **α (Learning Rate):** How much new info overrides old info (0 to 1).
- **γ (Discount Factor):** How much future rewards matter compared to immediate ones.
- **TD Error:** The difference between the estimated reward ($r + \gamma Q'$) and the current Q value.

SARSA Algorithm Steps

1. **Initialize:** Fill the Q-table with zeros or random numbers.
2. **Start:** Observe state s and choose action a using ϵ-greedy.
3. **Act:** Take action a, observe reward r and next state s'.
4. **Look Ahead:** Choose the **next** action a' from state s' (but don't perform it yet).
5. **Update:** Calculate the new $Q(s, a)$ using $Q(s', a')$.
6. **Loop:** Set $s = s'$ and $a = a'$; repeat until the goal is reached.

Model-Based Reinforcement Learning

• Agent learns or is given a model of the environment
• Model includes:

- ❖ Transition function: $P(s' \mid s,a)$
- ❖ Reward function: $R(s,a)$
  - • Uses model to plan actions
  - • Can simulate future trajectories
  - • More sample-efficient than model-free methods

Aspect	Model-Free	Model-Based
Data requirement	Many samples	Fewer samples
Planning	No	Yes
Complexity	Low	High
Examples	Q-Learning, SARSA	Dyna-Q, AlphaGo

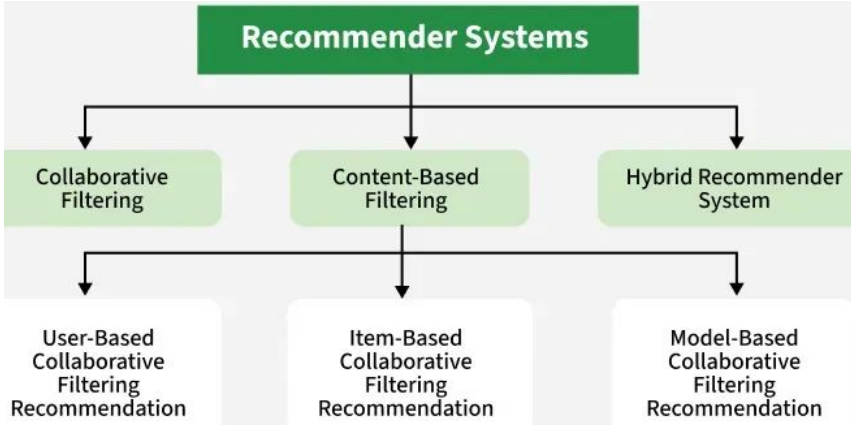
Planning Algorithms: • Value Iteration • Monte Carlo Tree Search (MCTS) • Dyna-Q

Dyna-Q Algorithm (Model-Based)

- • **Real experience** updates Q-values directly
- • Also **learns environment model** $\hat{P}, \hat{R}$
- • Generates **simulated experiences** using the model
- • Updates Q-values again using simulated data
- • Uses **n planning steps per real step**
- • More planning steps → **faster learning**

Recommended system

- Tools that suggest items automatically to users
- Based on behavior, preferences, and past interactions
- Helps users discover relevant content easily
- Widely used in YouTube, Amazon, Netflix



Content-Based Filtering

- Recommends items similar to user's past likes
- Uses item features & user profile like genre, tags, attributes
- Uses features Learns from user ratings over time

Advantages:

- Strong personalization, No need for other users' data
- Limitations: Limited diversity (low novelty)

Collaborative Filtering

- Uses behavior of similar users
- "Users like you also liked…"
- Recommends items based on similar users
- Uses user interaction patterns
- Example: "Customers also bought…"

❖ **User-Based**
❖ **Item-Based**

Advantages: Discovers new items

Limitations: Cold-start problem (new users/items)

Collaborative Filtering – Advantages

- Works even with small data
- Helps discover new interests
- No need for domain knowledge

Collaborative Filtering – Disadvantages

- Cold Start Problem (new users/items)
- Ignores side features (e.g., actor, year)

https://towardsdatascience.com/recommender-systems-a-complete-guide-to-machine-learning-models-96d3f94ea748/

User-Based Collaborative Filtering

- Based on similarity between users
- Finds users with similar preferences
- Recommends items liked by similar users

Example:
If A & C like similar items → recommend A's items to C

More items than users → Use User-Based

Item-Based Collaborative Filtering

- Based on similarity between items
- Recommends items similar to what user liked

Example:
If user likes watermelon → recommend grapes,
Example: Amazon uses Item-Based

More users than items → Use Item-Based

Working of Content-Based

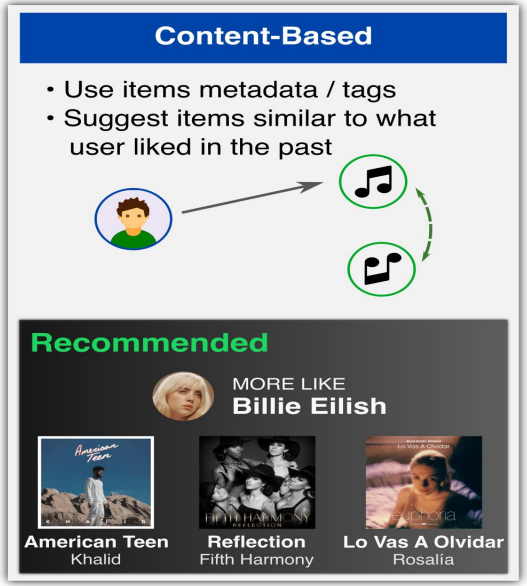
- Compare item features using similarity
- Recommend items similar to those liked earlier
- Uses algorithms like k-NN (Nearest Neighbors)

Working of Collaborative Filtering

- Uses user-item matrix r_{ui}
- Predicts missing values (unknown ratings "?")

Techniques:

- Matrix Factorization
- Clustering



Content-Based	Collaborative Filtering
Uses metadata	Uses user behavior
Works for new items	Needs past data
Less diverse	More diverse
Independent users	Uses all users

Hybrid Recommendation Systems

- Combines content-based & collaborative methods

Advantages: More accurate & balanced, Reduces limitations
Limitations: Complex & computationally expensive

Knowledge-Based Systems

- Uses domain knowledge & user requirements
- No need for past data

Advantages: Works without history
Limitations: Requires rule-building effort

https://towardsdatascience.com/recommender-systems-a-complete-guide-to-machine-learning-models-96d3f94ea748/

Context-Aware Systems

- Uses context (time, location, device, mood)

Advantages: More relevant recommendations
Limitations: Complex data handling

Feedback in Recommender Systems

- Used to train and evaluate models
- Helps predict user-item rating ($\mathbf{r_{ui}}$)
- Items with highest predicted rating are recommended

Types of Feedback

❖ **Explicit Feedback**

- Direct input from users
- Expresses satisfaction level

Examples: Star ratings (1–5 ⭐), Like/Dislike 👍👎, Reviews

❖ **Implicit Feedback**

- Based on user behavior
- No direct ratings

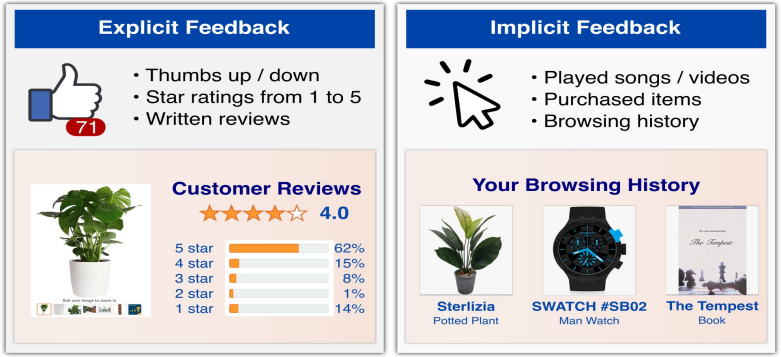
Examples: Clicks, views, purchases, Browsing history, Watch time

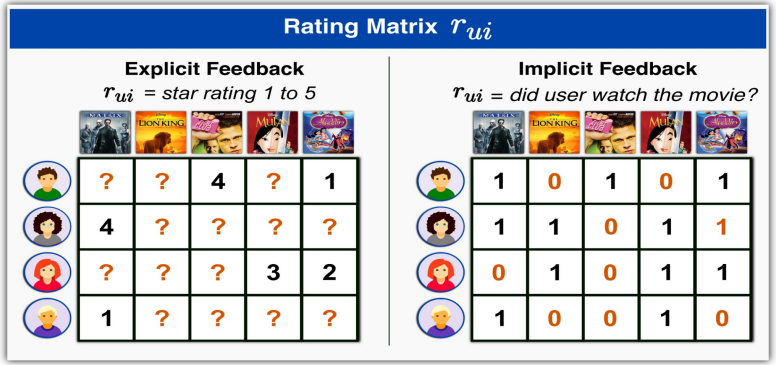
User-Item Matrix

- Matrix is sparse
- Users interact with few items only

Represented as r_{ui}

- Stores user interaction with items





Explicit Feedback:

- Numerical values (ratings)
- Missing values = not rated

Implicit Feedback:

- Boolean values (0/1 interaction)

Working of Recommender System

- Data Collection: Collect ratings, clicks, views, purchases, searches and item metadata.
- Data Preprocessing: Clean data, handle missing values and prepare user–item matrices.
- Feature Engineering: Transform user and item characteristics into meaningful vectors.
- Model Training / Similarity Computation: Use algorithms like collaborative filtering, content-based models or deep learning.
- Prediction: Generate relevance scores for unseen items based on learned patterns.
- Ranking & Recommendation: Sort items by predicted relevance and recommend the top ones.
- Feedback Loop: Update the model continuously as new user interactions arrive.

Deep Learning Models Used in Recommender Systems

Neural Collaborative Filtering

- Learns complex user-item interactions, Uses neural networks

Autoencoders

- Convert sparse data into dense embeddings, Helps in prediction & denoising

RNNs

- Capture sequential user behavior, Useful for next-item prediction

CNNs

- Extract features from images, text, audio, Useful in content-based recommendations

Transformers: Use attention mechanism, Handle long-term dependencies, State-of-the-art performance

Wide & Deep / DeepFM: Combines memorization + generalization, Used in ads & e-commerce systems

Evaluation Metrics

- Used to measure performance of recommendation systems
- Helps check if recommendations are relevant
- Evaluates quality & accuracy of predictions

1.Mean Average Precision at K (MAP at K)

- Measures relevance of top-K recommended items
- Precision at K = Relevant items in top K / K
- Higher value = better recommendations

2.Coverage

- Percentage of items the system can recommend
- Shows how much of catalog is used

Key Point: Higher coverage = more diverse recommendations

3.Personalization

- Measures uniqueness of recommendations per user
- Checks if different users get different suggestions

Key Point: Higher personalization = better user experience

4.Intra-List Similarity

- Measures similarity between items in same list
- Uses cosine similarity

Key Point:: Lower similarity = more diverse items,
Higher similarity = more similar items

Importance of Recommendation Systems

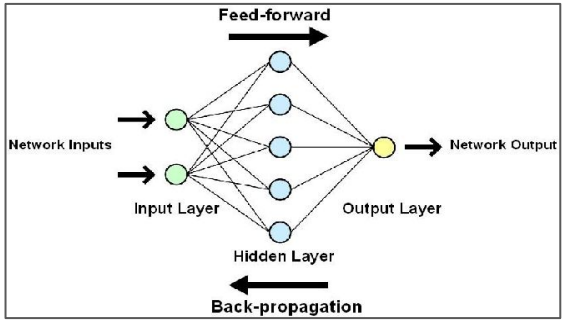
- Improves user experience
- Helps in better decision making

Benefits

- Enhanced Experience: Easy content discovery
- Business Growth: Increases sales & engagement
- Saves Time: Quick recommendations
- Builds Trust: Personalized suggestions
- Adaptive: Improves with user behavior

Artificial Neural Network (ANN)

- Inspired by the human brain
- Composed of interconnected neurons
- Used for pattern recognition & prediction
- Consists of layers: Input layer, Hidden layer(s), Output layer
- Learns from data (training)
- Handles non-linear problems
- Adaptive & improves with experience
- Used in image recognition, NLP, recommender systems



McCulloch-Pitts Neuron Model

- Introduced in 1943
- Also called Linear Threshold Gate
- One of the earliest artificial neuron models

Working of McCulloch-Pitts Model

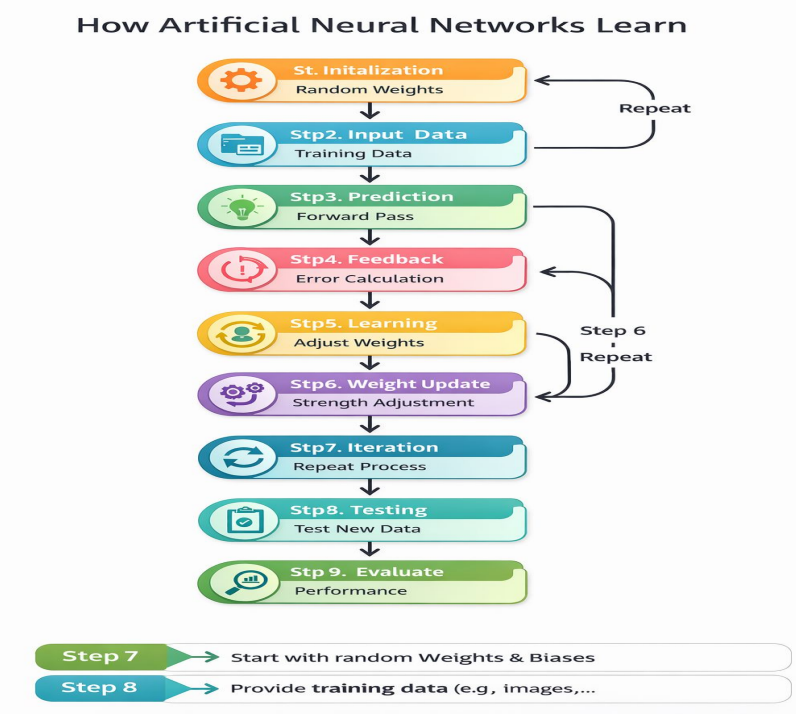
- Takes multiple inputs with weights
- Computes weighted sum
- Compares with threshold Θ

$$f(y_{in}) = \begin{cases} 1, & y_{in} \geq \Theta \\ 0, & y_{in} < \Theta \end{cases}$$

$$y_{in} = \sum x_i w_i$$

Output:

1 (fires) - if sum ≥ threshold
0 - otherwise

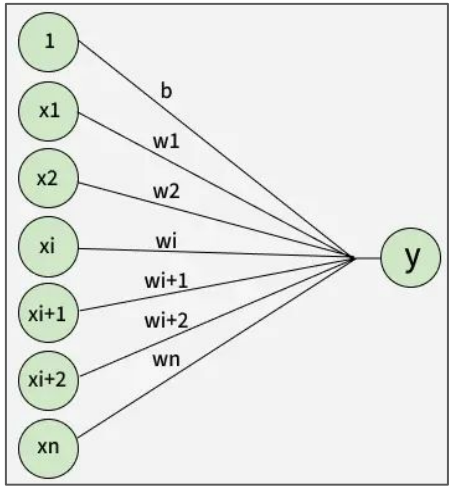


Single-Layer Neural Networks (Perceptron)

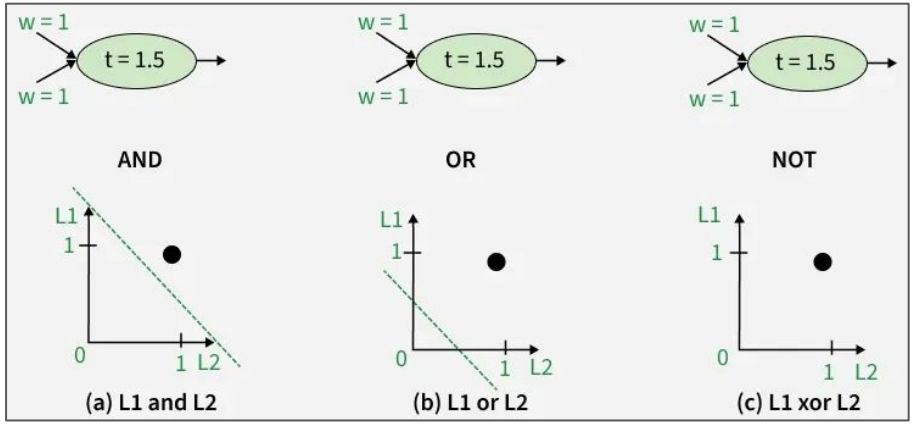
- Basic neural network model
- Input layer + one output neuron
- Used for binary classification

$$y = \begin{cases} 1, & \sum w_i x_i \geq t \\ 0, & \sum w_i x_i < t \end{cases}$$

- t: threshold



- Cannot solve non-linear problems (e.g., XOR)
- Led to development of MLP

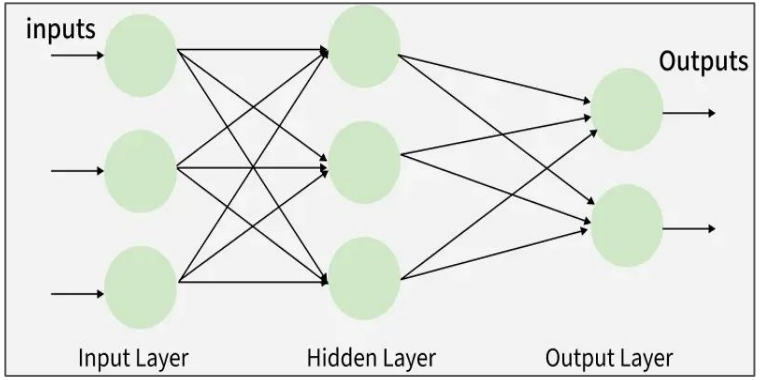


Multi-Layer Perceptron (MLP)

- Contains multiple hidden layers
- Solves complex non-linear problems

Layers: Input Layer, Hidden Layer(s), Output Layer

Input Layer: receives raw data
One or more Hidden Layers: extracts features
Output Layer: produces final predictions



Activation Functions

- Introduce non-linearity
  Common Functions: Sigmoid: Output (0,1), Tanh: Output (-1,1), ReLU: Output [0, ∞)

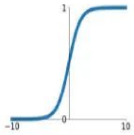
ANN Algorithm –

Steps

1. Initialize weights & bias
2. Input data
3. Forward propagation
4. Calculate error
5. Backpropagation
6. Repeat (epochs)
8. Test model

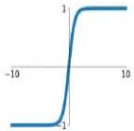
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



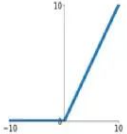
tanh

$$\tanh(x)$$



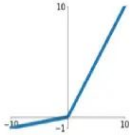
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

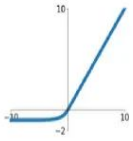


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Function	Formula	Range
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	$(0, 1)$
Tanh	$\tanh(x) = 2\sigma(2x) - 1$	$(-1, 1)$
ReLU	$f(x) = \max(0, x)$	$[0, \infty)$

Forward Propagation

- Input passes through layers
- Multiply by weights
- Apply activation function
- Generate prediction

Error Calculation

- Difference between predicted & actual output
- Loss functions: Mean Squared Error, Cross-Entropy

Backpropagation

- Updates weights to reduce error
- Uses gradient descent

Weight Update Formula

$$w = w - \alpha \frac{\partial Error}{\partial w}$$

α: learning rate

Training Process

- Repeat for multiple epochs
- Gradually reduce error
- Improve model accuracy

Testing the Model

- Use unseen data
- Check accuracy
- Fine-tune if needed

Types of Artificial Neural Networks

- Feedforward Neural Networks (FNN)
- Convolutional Neural Networks (CNN)
- Recurrent Neural Networks (RNN)
- Long Short-Term Memory (LSTM)
- Generative Adversarial Networks (GAN)

Feedforward Neural Networks (FNN)

- Simplest type of ANN
- Data flows in one direction (input → output)
- No feedback loops

Applications: Pattern recognition, Prediction tasks

Convolutional Neural Networks (CNN)

- Designed for image processing
- Detects patterns like edges, shapes

Applications: Image recognition, Object detection, Video analysis

Recurrent Neural Networks (RNN)

- Handles sequential data
- Remembers previous inputs

Applications: Language modeling, Speech recognition,Text prediction

Long Short-Term Memory (LSTM)

- Advanced type of RNN
- Handles long-term dependencies
- Solves vanishing gradient problem

Applications: Machine translation, Time-series analysis, Chatbots

Generative Adversarial Networks (GAN)

- Consists of two networks: Generator, Discriminator
- Compete to generate realistic data

Applications: Image generation, Deepfakes, AI art & content creation

Advantages of ANN

- Powerful models inspired by human brain
- Capable of solving complex problems

Noise Resilience: Handles noisy & incomplete data, Still produces accurate results, Robust to errors in training data

Versatility; Works with real-valued & discrete data

Used in: Image recognition, Speech recognition, Decision-making systems

Efficiency:

- Fast predictions after training
- Suitable for real-time applications
- Examples: Self-driving cars, Fraud detection,

Parallel Processing:

Processes large data simultaneously, Distributed computation like brain, Improves speed & performance

Limitations of ANN

- Some challenges in training & usage

Overfitting

- Model memorizes training data
- Poor performance on new data
- Happens with complex models or less data

Data Dependency

- Requires large datasets
- Small data → poor generalization

Computational Cost

- Training is time-consuming
- Requires high hardware resources (GPU/CPU)

Given:

MLP with One Hidden Layer

Inputs:

- $x_1 = 1,\ x_2 = 2$

Hidden Layer (2 neurons):

- Weights:
  - Neuron H1: $w_{11} = 0.2,\ w_{12} = 0.4$, bias $b_1 = 0.1$
  - Neuron H2: $w_{21} = 0.3,\ w_{22} = 0.5$, bias $b_2 = 0.2$

Output Layer (1 neuron):

- Weights: $v_1 = 0.6,\ v_2 = 0.7$, bias $b = 0.3$

Activation Function: Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Step 1: Hidden Layer Calculation

Neuron H1:

$$z_1 = (1 \times 0.2) + (2 \times 0.4) + 0.1 = 0.2 + 0.8 + 0.1 = 1.1$$

$$h_1 = \sigma(1.1) = \frac{1}{1 + e^{-1.1}} \approx 0.75$$

Neuron H2:

$$z_2 = (1 \times 0.3) + (2 \times 0.5) + 0.2 = 0.3 + 1.0 + 0.2 = 1.5$$

$$h_2 = \sigma(1.5) = \frac{1}{1 + e^{-1.5}} \approx 0.82$$

Step 2: Output Layer Calculation

$$z = (0.75 \times 0.6) + (0.82 \times 0.7) + 0.3$$

$$z = 0.45 + 0.574 + 0.3 = 1.324$$

Step 3: Final Output

$$y = \sigma(1.324) = \frac{1}{1 + e^{-1.324}} \approx 0.79$$

Final Answer:

- **Output ≈ 0.79** ✅

MLP Backpropagation (1 Hidden Layer)

Given:

Inputs:
$x_1 = 1,\ x_2 = 2$

Hidden Layer (2 neurons):

- H1: $w_{11} = 0.1,\ w_{12} = 0.2,\ b_1 = 0.1$
- H2: $w_{21} = 0.3,\ w_{22} = 0.4,\ b_2 = 0.1$

Output Layer:

- $v_1 = 0.5,\ v_2 = 0.6,\ b = 0.2$

Target: $t = 1$
Learning rate: $\alpha = 0.1$
Activation: Sigmoid

🔹 Step 1: Forward Pass

Hidden Layer:

$$z_1 = (1 \times 0.1) + (2 \times 0.2) + 0.1 = 0.1 + 0.4 + 0.1 = 0.6$$

$$h_1 = \sigma(0.6) \approx 0.645$$

$$z_2 = (1 \times 0.3) + (2 \times 0.4) + 0.1 = 0.3 + 0.8 + 0.1 = 1.2$$

$$h_2 = \sigma(1.2) \approx 0.768$$

Output Layer:

$$z = (0.645 \times 0.5) + (0.768 \times 0.6) + 0.2$$

$$z = 0.3225 + 0.4608 + 0.2 = 0.9833$$

$$y = \sigma(0.9833) \approx 0.728$$

Step 2: Error Calculation

$$E = \frac{1}{2}(t - y)^2 = \frac{1}{2}(1 - 0.728)^2 \approx 0.037$$

Step 3: Output Layer Gradient

$$\delta_{out} = (y - t) \cdot y(1 - y)$$

$$= (0.728 - 1) \times (0.728 \times 0.272)$$

$$= (-0.272) \times 0.198 \approx -0.0538$$

Step 4: Update Output Weights

$$v_1^{new} = 0.5 - 0.1(-0.0538 \times 0.645) \approx 0.5035$$

$$v_2^{new} = 0.6 - 0.1(-0.0538 \times 0.768) \approx 0.6041$$

Step 5: Hidden Layer Gradients

$$\delta_1 = \delta_{out} \times v_1 \times h_1(1 - h_1)$$

$$= (-0.0538 \times 0.5 \times 0.645 \times 0.355) \approx -0.0061$$

$$\delta_2 = \delta_{out} \times v_2 \times h_2(1 - h_2)$$

$$= (-0.0538 \times 0.6 \times 0.768 \times 0.232) \approx -0.0058$$

🔹 Step 6: Update Hidden Weights

For H1:

$$w_{11}^{new} = 0.1 - 0.1(-0.0061 \times 1) \approx 0.1006$$

$$w_{12}^{new} = 0.2 - 0.1(-0.0061 \times 2) \approx 0.2012$$

For H2:

$$w_{21}^{new} = 0.3 - 0.1(-0.0058 \times 1) \approx 0.3006$$

$$w_{22}^{new} = 0.4 - 0.1(-0.0058 \times 2) \approx 0.4012$$

Final Answer:

- Output ≈ **0.728**
- Error ≈ **0.037**
- Updated weights slightly increased

- Forward pass → find output
- Compute error
- Find δout
- Backpropagate to hidden layer
- Update weights

Introduction to Deep Learning (Advanced ANN)

- Subset of Machine Learning
- Based on Artificial Neural Networks (ANNs)
- Uses multiple layers (deep networks)
- Learns complex patterns automatically
- Handles large & complex data
- Learns features automatically (no manual feature engineering)
- High accuracy in real-world problem

Key Characteristics

- Multiple hidden layers (deep architecture)
- Learns hierarchical features
- Requires large datasets
- Uses high computational power (GPU)

Advantages of Deep Learning

- High accuracy
- Automatic feature extraction
- Handles unstructured data (image, audio, text)

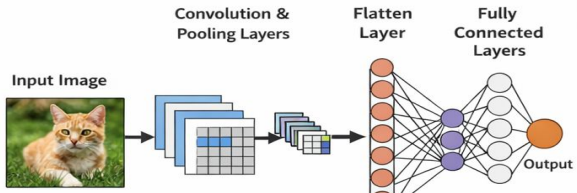
Limitations of Deep Learning

- Requires large data
- High computational cost
- Hard to interpret (black-box model)

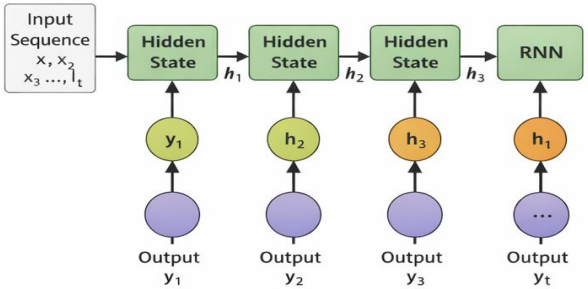
Types of Deep Learning Models

- Feedforward Neural Networks (FNN)
- Convolutional Neural Networks (CNN)
- Recurrent Neural Networks (RNN)
- LSTM Networks, GANs

Convolutional Neural Network (CNN)



Recurrent Neural Network (RNN)



CONVOLUTIONAL NEURAL NETWORK (CNN)

Used for Image Processing / Computer Vision

Input Image

1	1	0	0	1
1	0	1	1	0
0	1	1	0	1
0	0	1	1	0
1	0	0	1	1

(5 × 5 Pixels)

1. CONVOLUTION LAYER

Applies Filters/Kernels

1	0	-1
1	0	-1
1	0	-1

Feature is extracted using the kernel

2. FEATURE MAP

2	1	-1
1	2	-2
0	1	-1

Output after convolution

3. REACTIVATION (ReLU)

Adds Non-Linearity

2	1	0
1	2	0
0	1	0

All negative values become 0

4. POOLING LAYER

Reduces Dimensions (Max Pooling 2 × 2)

2	2
1	1

5. FULLY CONNECTED LAYER

Performs Classification

Cat
Dog
Car

Final Output Probabilities

RECURRENT NEURAL NETWORK (RNN)

Used for Sequential Data / Time Series / NLP

1. UNFOLDED STRUCTURE

Recurrent Connection

h_0 Initial Hidden State
h_1 h_2 h_3 ...
x_1 Input (t1)
x_2 Input (t2)
x_3 Input (t3)

The hidden state is passed from one step to the next.

2. SINGLE RNN CELL

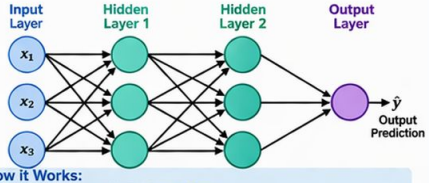
x_t Input at time t
× W_{xh}
× W_{hh}
Hidden Weight
Σ
tanh Activation
h_t New Hidden State
h_{t-1} Previous Hidden State

3. WORKING

- ✓ Takes input at time t (x_t)
- ✓ Combines with previous hidden state (h_{t-1})
- ✓ Updates hidden state (h_t)
- ✓ Captures sequence information

FEEDFORWARD NEURAL NETWORK (FNN)

Data flows in one direction (Input → Output)

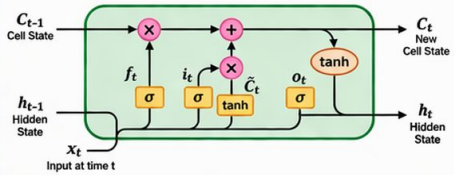


How it Works:
- Input features are multiplied by weights
- Passes through hidden layers with activation functions (ReLU, Sigmoid)
- Produces output through final layer

Applications: Classification, Regression, Pattern Recognition

LSTM NETWORKS

Specialized RNN for learning long-term dependencies

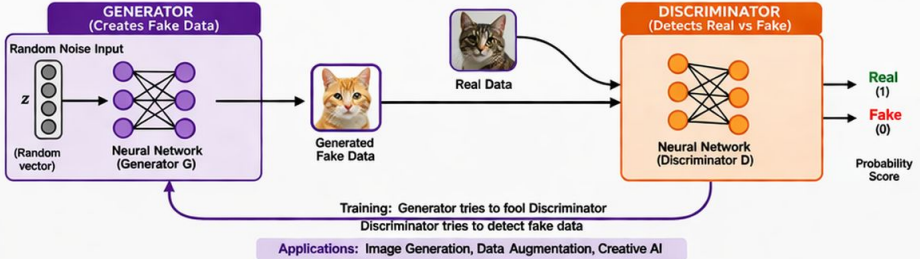


How it Works:
- Forget gate (f_t): Decides what to forget from cell state
- Input gate (i_t): Decides what new information to store
- Output gate (o_t): Decides what to output
- Maintains memory via cell state (C_t)

Applications: Text Generation, Machine Translation, Time Series

GENERATIVE ADVERSARIAL NETWORKS (GANs)

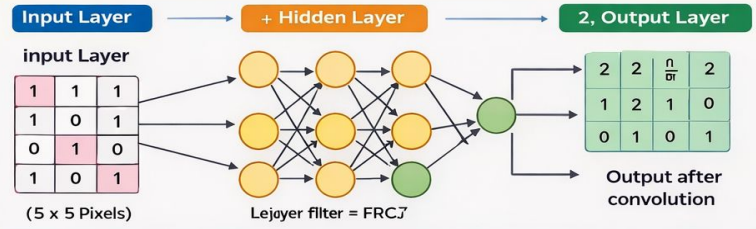
Two networks competing against each other



Applications: Image Generation, Data Augmentation, Creative AI

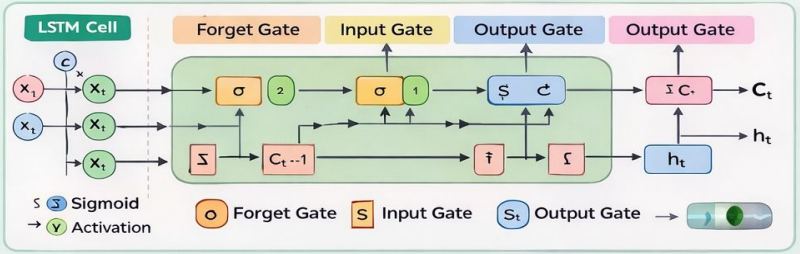
FEEDFORWARD NEURAL NETWORKS (FNN)

Used for Pattern Recognition / Classification



LONG SHORT-TERM MEMORY NETWORKS (LSTM)

Used for Sequential Data / Time Series / NLP



GENERATIVE ADVERSARIAL NETWORKS (GANs)

Used for Generative AI / Content Creation / Deepfakes