greentecAI

Theoretical Study on Machine Learning

Written By: Ahnaf Mahmud

Completed On: July 19, 2025

Table of Contents

What is Machine Learning? 4

What are the branches of Machine Learning? 4

Foundations and Data Pipeline 4

Problem Types & Terminology 4

Online vs Batch Learning 5

- ❖ **Learning-Schedule Dimension** 5
- ❖ **Representation Dimension** 5

Machine Learning Project Pipeline 6

Data Preparation Workflow 8

Splits & Cross-Validation 8

- ❖ **Train/Validation/Test Splits** 8
- ❖ **Cross-Validation Schemes** 9
- ❖ **Practical Guidelines & Pitfalls** 9
- ❖ **Feature Scaling, Encoding, Imputation & Outlier Handling** 10

Bias–Variance Trade-off & Model Capacity 11

Error Decomposition 11

Model Capacity & Regularisation 11

Diagnostic Curves 11

Cheat-Sheet: Symptoms → Likely Cause → Remedy 12

Supervised Learning 13

Linear Regression 13

Logistic Regression 15

k-Nearest Neighbours (k-NN) 18

Decision Trees (CART) 20

Random Forests 24

Gradient-Boosted Trees 27

Unsupervised Learning 30

Clustering & Dimensionality-Reduction Metrics 30

k-Means Clustering 31

Hierarchical Clustering 35

DBSCAN – Density-Based Spatial Clustering of Applications with Noise 37

Dimensionality Reduction – PCA and t-SNE 39

Reinforcement Learning 42

Tabular Q-Learning 43

SARSA (State–Action–Reward–State–Action) 45

Deep Q-Network (DQN) 47

Model-Evaluation Metrics and Decision Guidelines 49

Classification Metrics 49

Regression Metrics 50

Decision Matrix – Choosing the Right Metric 52

Choosing the Right Algorithm 53

Conclusion 55

What is Machine Learning?

Machine learning is a branch of Artificial Intelligence that centers on creating models and algorithms enabling computers to learn from data without needing explicit instructions for each specific task. Simply put, it allows systems to learn and make decisions based on data, much like how humans learn through experience.

What are the branches of Machine Learning?

Machine Learning is divided into three main branches – Supervised Learning, Unsupervised Learning, and Reinforcement Learning.

- **Supervised Learning**: Supervised learning is a type of machine learning that uses labeled datasets to train algorithms to predict outcomes and recognize patterns. For example, a model can learn to predict a house's sale price from features like square footage, location, and number of bedrooms.

- **Unsupervised learning**: Unsupervised learning is a type of machine learning where algorithms work with unlabeled datasets, discovering underlying patterns, structures, or groupings without any human-provided labels. For example, k-Means can automatically group e-commerce shoppers into distinct customer segments based on their browsing and purchase patterns.

- **Reinforcement Learning:** Reinforcement learning is a type of machine learning where an agent learns to make decisions by interacting with an environment, receiving rewards or penalties based on its actions, and aiming to maximize cumulative reward over time. For example, an agent can learn to play the Atari game Breakout by receiving positive reward points each time it successfully bounces the ball and breaks a brick.

What are labeled and unlabeled datasets?

- **Labeled datasets** contain input data paired with the correct output (label), used for supervised learning.
- **Unlabeled datasets** consist of input data without any corresponding output labels, used in unsupervised learning.

Foundations and Data Pipeline

Problem Types & Terminology

Regression

Predicts a continuous numerical value for each new input by learning the quantitative relationship between features and a target variable. Typical tasks include forecasting prices, temperatures, or loads, where success is measured by how small the prediction error is (e.g., low Mean Square Error (MSE) or Mean Absolute Error (MAE)).

Classification

Assigns each input to one of several discrete categories. The model learns class-separating boundaries so

that new examples are labeled correctly—such as “spam vs. not-spam” emails or “benign vs. malignant” tumors. Performance is judged with metrics like accuracy, precision-recall, F1, or area under the Receiver Operating Characteristic curve (ROC-AUC).

Clustering
Groups unlabeled data into coherent clusters based on similarity, revealing hidden structure without predefined labels. It answers “which points naturally belong together?” for tasks like customer segmentation or gene-expression profiling. Quality is assessed with internal scores (e.g., silhouette, Davies-Bouldin) or external benchmarks if ground truth exists.

Control / Reinforcement Learning
Focuses on learning a policy—an action strategy—that maximizes cumulative reward while interacting with an environment over time. The agent balances exploration and exploitation to solve sequential decision problems such as robot navigation or playing video games. Progress is tracked with average return per episode, convergence speed, or sample efficiency.

Optimiser (Training Algorithm)
The **optimiser** adjusts a model’s parameters to minimise its loss function—e.g., Gradient Descent, Stochastic Gradient Descent (SGD), Adam, or L-BFGS—driving convergence for any of the tasks above.

Online vs Batch Learning

❖ Learning-Schedule Dimension

In the context of how models are trained over time, we distinguish between **Batch (Offline) Learning** and **Online (Incremental) Learning**:

- **Batch (Offline) Learning** involves training the model once on the entire dataset. The model’s parameters remain fixed until it is retrained with a new dataset. This method is common in use-cases like nightly Extract, Transform, Load (ETL) pipelines or image classification competitions.
  Advantages include the ability to leverage thorough preprocessing and optimization, and easier debugging and reproducibility.
  Drawbacks include high computational cost as data grows and poor adaptability to changes or concept drift.

- **Online (Incremental) Learning**, on the other hand, updates the model continuously as new data arrives, either one sample at a time or in small batches. It is well-suited for real-time systems such as click-through-rate prediction, IoT sensor data analysis, or fraud detection.
  Advantages are quick adaptation to changes and efficient memory usage.
  Drawbacks include sensitivity to noisy data and challenges in reproducibility due to the constant state of flux.

❖ Representation Dimension

This dimension focuses on how the model represents knowledge and performs generalization.

- **Instance-based (Lazy) Learning** stores the training data and postpones generalization until a query is made. It depends on a similarity measure to make predictions. Algorithms like k-Nearest

Neighbors (k-NN), Self-Organizing Maps (SOM), and Case-Based Reasoning fall into this category.
Pros include zero training time and the ability to handle complex decision boundaries naturally.
Cons involve high memory usage and slow prediction speed since all computations are deferred until query time.

- **Model-based (Eager) Learning** creates a parametric model during training and only stores the learned parameters (e.g., weights). Predictions are fast and do not require access to the entire dataset. Examples include Linear/Logistic Regression, Support Vector Machines (SVMs), Random Forests, and Neural Networks.
Pros are fast inference, storage, and compatibility with hardware accelerators.
Cons include the need for upfront training, which can be computationally expensive, and a risk of underfitting or overfitting if the model's capacity doesn't match the problem.

Machine Learning Project Pipeline

A successful machine learning project follows a structured pipeline that ensures clarity, reproducibility, and performance. Each step is essential and builds upon the previous one.

1. Problem Definition

Clearly define the business or research objective. Understand the type of ML problem: Is it classification, regression, clustering, or reinforcement learning? This sets the direction for everything that follows.

2. Data Collection

Gather data from reliable sources—databases, APIs, sensors, or web scraping. Ensure that sufficient quantity and diversity of data are present to represent the real-world scenario accurately.

3. Data Exploration and Analysis (EDA)

Use statistical summaries and visualizations to:

- Understand feature distributions
- Detect outliers and anomalies
- Identify missing values
- Reveal relationships and trends in the data

4. Data Cleaning and Preprocessing

Apply preprocessing techniques to make the data suitable for modeling:

- Handle missing values (imputation or removal)
- Encode categorical variables
- Scale or normalize features

- Address outliers
- Handle class imbalance (e.g., oversample minority classes, undersample majority classes, apply class-weighting, or use SMOTE)
- Split data into train, validation, and test sets

5. Feature Engineering

Create or transform variables that enhance model performance:

- Combine features
- Create interaction terms
- Extract date/time parts (e.g., hour from timestamp)
- Reduce dimensionality (e.g., Principal Component Analysis (PCA))

6. Model Selection

Choose suitable algorithms based on problem type, dataset size, interpretability needs, and computational constraints. Common choices include:

- Regression: Linear, Ridge, Lasso
- Classification: Logistic Regression, SVM, Random Forest, Extreme Gradient Boosting (XGBoost)
- Clustering: K-Means, DBSCAN
- Reinforcement: Q-Learning, Deep Q-Networks

7. Model Training

Train the model using the training set. Apply techniques like cross-validation to evaluate performance and avoid overfitting.

8. Hyperparameter Tuning

Optimize model parameters using:

- Grid Search
- Random Search
- Bayesian Optimization
  Evaluation is done on the validation set to prevent test leakage.

9. Model Evaluation

Assess final performance on the test set using appropriate metrics:

- Classification: Accuracy, Precision, Recall, F1, ROC-AUC

- Regression: MSE, Root MSE (RMSE), MAE, R^2

10. Model Interpretation

Use tools like SHAP, LIME, or feature importance to explain predictions and validate trust in the model.

11. Deployment

Export the trained model to a production environment using formats like:

- .pkl (pickle)
- .joblib
- ONNX, TensorFlow SavedModel
  Integrate via APIs or batch pipelines.

12. Monitoring and Maintenance

Track live performance using monitoring tools. Retrain the model when performance drops due to data drift, model decay, or changing requirements.

Data Preparation Workflow

Splits & Cross-Validation

Proper evaluation is the only way to approximate how the model will behave on genuinely unseen data—guarding against the seductive but deadly illusion of overfitting to a single snapshot of the dataset. If the split or cross-validation scheme is flawed, every downstream decision (from hyper-parameter tuning to model choice) can be biased, leading to models that shine in development yet stumble in the real world.

❖ **Train/Validation/Test Splits**

Before training a machine learning model, the dataset is typically divided into three parts:

Training Set: The portion of data used to fit the model's parameters. This is the data the model "learns" from.

Validation Set: A separate subset used to fine-tune hyperparameters and make decisions about model architecture or early stopping.

Test Set: A completely untouched subset used only once to evaluate the model's final performance and generalization ability.

Split	Typical share
Training set	60-80 %
Validation set	10-20 %
Test set	10-20 %

❖ Cross-Validation Schemes

Scheme	How it works	When to prefer	Pros	Cons
k-Fold CV	Partition data into *k* equally sized folds; iterate *k* times so each fold is the "mini-test-set" once	Most tabular data; balanced classes	Uses every sample for train **and** validation; variance ↓	Cost × k; can still leak if preprocessing done incorrectly
Stratified k-Fold	Same as above but preserves label proportions in every fold	Imbalanced classification (e.g., fraud 1 %)	Fair class representation	Slightly complex sampling
Leave-One-Out (LOOCV)	k = *n*; validate on a single point each round	Tiny datasets (< 200 rows)	Nearly unbiased	Extremely slow; high variance
Nested CV	Outer loop = model evaluation; inner loop = hyper-parameter search, both with k-folds	When grid / random search tuning is heavy	Mitigates over-fitting & leakage	Computation × k^2
Time-Series (Rolling / Expanding Window)	Train on all data **earlier** than validation fold, slide window forward in time	Temporal data (stock prices, IoT)	Respects causality; drift detection	Smaller effective train size; no shuffle
Repeated k-Fold	Run k-fold CV *m* times with different random splits; average over m×k runs	Very small or noisy data where variance dominates	Lower variance estimate	Cost × m

❖ Practical Guidelines & Pitfalls

1. **Fit-transform only on training folds.** Scaling, encoding or imputation done before splitting leaks statistics.
2. **Shuffle vs. temporal order:** shuffle for iid data, keep chronological order for forecasts.
3. **Class imbalance:** combine *stratified* folds with appropriate metrics (F1, PR-AUC) instead of accuracy.
4. **Early-stopping:** use the validation loss inside each training fold—not the test set—to decide when to stop training deep models.

5. **Data shortage:** prefer k-fold (k ≥ 5) to a single hold-out; LOOCV if *n* is very small but expect long runtimes.
6. **Model comparison fairness:** evaluate *every* candidate model with the same CV object and random seed.

❖ Feature Scaling, Encoding, Imputation & Outlier Handling

Before any model can learn reliably, the raw data must first be put into a form *the algorithm can* actually understand.
If these preprocessing steps are skipped—or done in the wrong order—the model will optimise noise rather than signal, leading to impressive validation scores that collapse the moment the code reaches production.

Task	Core techniques	Algorithms most affected	Tips & leakage traps
Scaling / Normalisation	• *Standardisation* (z-score) • *Min–max* to [0,1] • *Robust scaler* (IQR)	Distance-based models (k-NN, SVM, K-Means), gradient methods, neural nets	Fit the scaler **only on the training folds**; reuse its mean/σ on validation & test.
Encoding categorical features	• *One-hot (dummy) encoding* • *Ordinal / label encoding* • *Target encoding* for high cardinality	Tree-free models (linear/logistic reg., SVM, NN)	Apply *after* the train/val/test split so the test set can surface unseen categories.
Imputation of missing values	• *Mean / median / mode* • *k-NN* or *Iterative (MICE)* • Add *"was missing"* indicator flag	Any dataset with NaNs; small-n problems where row deletion hurts	Compute imputation stats on **train data only**; never use the target when filling predictors.
Outlier handling	• *Winsorise / clip* • *Log* or *Box–Cox* transform • *Isolation Forest / DBSCAN* to drop or down-weight extremes	Linear models, k-means are sensitive; tree ensembles less so	Always visualise first (box plot, z-score). Keep an untouched copy for audit purposes.

Leakage watch-list:

1. Scaling or encoding the *entire* dataset **before** splitting.
2. Filling NaNs with the *global* mean—including rows that should be held out.
3. Performing target encoding with labels visible **during** cross-validation.

4. Deleting "outliers" after inspecting validation performance.
5. Shuffling a time-series before scaling, which mixes future information into the past.

Bias–Variance Trade-off & Model Capacity

Even a perfectly split and pre-processed dataset can yield a useless model if the learning algorithm has the *wrong capacity*: too simple and it under-fits (high bias); too flexible and it over-fits (high variance). Understanding this trade-off is essential for selecting appropriate algorithms, tuning hyper-parameters, and deciding how much data or regularisation is needed.

Error Decomposition

For a fixed point x, the expected squared error of any supervised model trained on random dataset $\mathbb{D}$ is

$$E_D\left[\left(\widehat{f_{\mathcal{D}}}(x) - f(x)\right)^2\right] = \underbrace{\left(Bias\left[\widehat{f_{\mathcal{D}}}(x)\right]\right)^2}_{\text{under-fit}} + \underbrace{Var\left[\widehat{f_{\mathcal{D}}}(x)\right]}_{\text{over-fit}} + \sigma_\varepsilon^2$$

The irreducible noise term sets a floor below which no model can go.

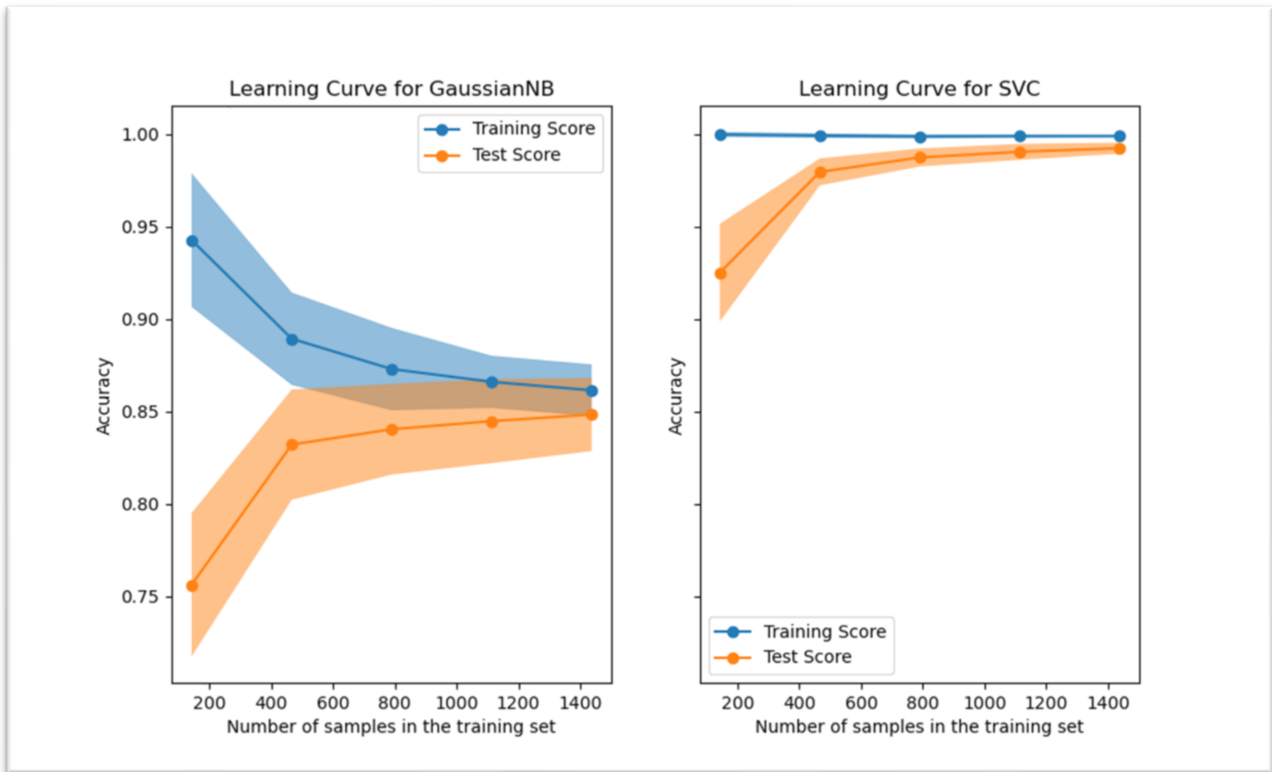
Model Capacity & Regularisation

Capacity dial	Effect on bias	Effect on variance	Typical hyper-parameter
Increase params / depth (more neurons, deeper trees)	↓ bias	↑ variance	n_estimators, network layers
Add regularisation (L1/L2, dropout, pruning)	↑ bias	↓ variance	alpha, C, max_depth
Add data / augmentation	↘ bias	↘ variance	sample size, synthetic images
Bagging / ensembling	—	↓ variance	Random-Forest, Gradient Boosting

Rule-of-thumb: If training and validation errors are both high then boost capacity; if train error is low but validation error is high then add regularisation or gather more data.

Diagnostic Curves

1. **Learning curve** – plot training vs validation error as sample size grows.
   - High & parallel curves → high bias.
   - Diverging gap with low train error → high variance.
2. **Validation curve** – plot error against a capacity hyper-parameter (e.g., SVM's **C** or tree depth).
   - U-shaped curve pinpoints the sweet spot where bias and variance are balanced



1Learning Curves for GaussianNB and SVC (scikit-learn example)

Cheat-Sheet: Symptoms → Likely Cause → Remedy

Observation (on CV)	Interpretation	First fixes to try
High train error & high val error	High bias (under-fit)	Add features, raise model capacity, lower regularisation
Low train error & high val error	High variance (over-fit)	Increase regularisation, use dropout/bagging, collect more data
Both errors low but production error spikes	Dataset shift / concept drift	Adopt online learning, retrain with recent data, feature normalisation check
Validation curve flat across capacity values	Metric insensitive or wrong model family	Switch metric, choose algorithm better matched to problem type

Key Takeaways:

- Bias and variance move *in opposite directions* when capacity changes; perfection lies in the compromise.
- Regularisation techniques (L1, L2, dropout, tree-pruning) are practical levers for shrinking effective capacity without redesigning the model.
- Always inspect learning and validation curves before making itinerary-changing decisions like collecting more data or switching algorithms.

- The bias–variance framework applies equally to regression and classification; in reinforcement learning the same tension appears as value-function approximation error vs exploration noise.

Supervised Learning

Supervised learning deals with datasets that come pre-labeled: for every input x we know a target y.
The learner's objective is to approximate an unknown mapping $f\colon\ x \mapsto y$ such that, for new unseen examples, the predicted $\hat{y}$ is as close as possible to the true value.
Supervised tasks fall into two broad flavours: **regression** (continuous targets) and **classification** (discrete class labels).

Linear Regression

❖ Problem Type

Supervised **regression**: predict a continuous target such as house price, temperature, or demand.

❖ Core Idea

Fit a straight hyper-plane that minimises the mean-squared error between predicted values $\hat{y} = X$ and true targets y.
Because the hypothesis is linear in parameters β, we obtain a closed-form estimator via ordinary least squares (OLS).

❖ Hypothesis

$$\hat{y} = X\beta + \beta_0,$$

Where:

- $X \in R^{n \times p}$ is the design matrix (one column per feature, plus an optional intercept column),
- $\beta \in R^p$ are the feature weights,
- β_0 is the bias term.

The hypothesis class is **all linear functions** in the input features.

❖ Cost / Loss Function

The model parameters are learned by minimising the **mean-squared error (MSE)**:

$$\mathcal{L}(\beta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- **Closed-form OLS solution**
  $\hat{\boldsymbol{\beta}}_{OLS} = (X^\top X)^{-1} X^\top y,$

(solved via QR/SVD in practice).

- **Iterative optimiser (large p or streaming)**
  Gradient-descent update
  $$\beta \leftarrow \beta - \eta \frac{2}{n} X^{\top}(X\beta - y),$$
  with learning-rate η.

Regularised variants add an ℓ_2 (Ridge) or ℓ_1 (Lasso) penalty to the loss.

❖ **Optimiser**

- **Closed-form OLS** Solve the normal equations once via QR or SVD
  $\hat{\beta}_{OLS} = (X^{\top}X)^{-1}X^{\top}y.$
  Pros: exact, one-shot; *Cons:* cubic in p, memory-heavy for big data.

- **Batch / Stochastic Gradient Descent** Iteratively update
  $$\beta \leftarrow \beta - \eta \frac{2}{n} X^{\top}(X\beta - y).$$
  Use small mini-batches for very large n; choose a decaying learning-rate schedule.

- **Regularised variants** Replace OLS with Ridge (closed-form using λI) or apply SGDRegressor with penalty='l2' / 'l1'.

❖ **Hyper-parameters**

Name	Typical range	Effect on model	Notes
fit_intercept	True / False	Adds bias term	Needed if mean of $y \neq 0$
normalize/standardize	True / False	Centres & scales features	Improves numerical stability
Regularisers			
alpha (Ridge/Lasso)	$10^{-4} – 10^{2}$	Shrinks coefficients	Higher ⇒ more bias, less variance

❖ **Pros / Cons**

Pros

- **Fast to train:** the closed-form OLS solution is $O(p^3)$ for a one-off matrix decomposition, and stochastic gradient descent scales to millions of rows.

- **Interpretable coefficients:** each β weight shows the marginal effect of its feature when others are held constant, which is valuable for explanatory studies.

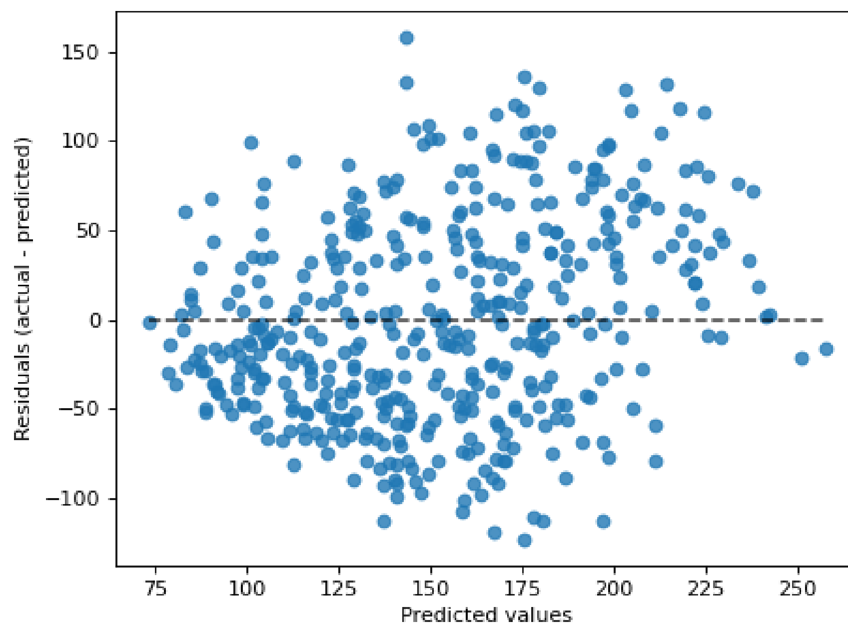
- **Straightforward regularisation:** Ridge (ℓ_2), Lasso (ℓ_1) and Elastic-Net extend the model with just one additional hyper-parameter, giving easy control over variance.

Cons

- **Limited to linear relationships:** real-world phenomena often contain non-linear interactions that plain OLS cannot capture without engineered features.
- **Sensitive to multicollinearity:** highly correlated predictors inflate variance and yield unstable coefficient estimates (check VIF > 5 as a rule of thumb).
- **Outlier influence:** observations with large residuals can pull the fitted line—use robust regressors or transform the target to mitigate.

❖ **Typical Applications**

- Real-estate price estimation
- Forecasting energy consumption
- Assessing the effect size of medical treatments (when causal assumptions hold)



2Residuals vs predicted values for a linear-regression fit, showing random scatter around 0.

Logistic Regression

❖ **Problem Type**

Supervised **binary (and multinomial) classification**: assign each input vector x to one of two (or k) discrete classes and estimate the probability of membership.

❖ **Core Idea**

Model the log-odds of the positive class as a linear function of the features, then squash that linear score through the sigmoid to obtain a probability.

$$p(\,y = 1 \mid x\,) = \sigma(z) = \frac{1}{1+e^{-z}},\ z\ =\ \beta_0 + \beta^{\top}x$$

A decision boundary at p=0.5 (or any chosen threshold) turns these probabilities into class labels.

❖ Hypothesis (Model Form)

$$\hat{y} = \sigma(\beta_0 + \beta^{\top}x),$$

where $\beta \in \mathbb{R}^p$ are feature weights and β_0 is the bias.
The hypothesis class is all linear decision boundaries in feature space, mapped to probabilities via the sigmoid (or soft-max for multiclass).

❖ Cost / Loss Function

Binary **cross-entropy (negative log-likelihood)**

$$\mathcal{L}(\beta) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log p_i\ + (1 - y_i)\log(1 - p_i)],\qquad p_i = \sigma(\beta_0 + \boldsymbol{\beta}^{\top}x_i).$$

Gradient

$$\nabla_\beta \mathcal{L} = \sum_{i=1}^{n}(p_i - y_i)x_i$$

Regularised variants add an ℓ_2 (Ridge) or ℓ_1 (Lasso) penalty to $\mathcal{L}$.

❖ Optimiser

- **Newton–Raphson / L-BFGS** — second-order; fast for small to medium p.
- **liblinear / saga** — coordinate-descent solvers in scikit-learn; handle ℓ_1 and elastic-net.
- **Stochastic Gradient Descent (SGD)** — streaming or very large, sparse data.

❖ Hyper-parameters

Parameter	Typical values	Effect
penalty	l2, l1, elasticnet, none	Kind of regularisation
C	0.001 – 100 (inverse strength)	Smaller $C \Rightarrow$ heavier regularisation
solver	lbfgs, liblinear, saga, newton-cg	Optimiser; impacts speed / penalty support
class_weight	None, 'balanced', dict	Re-weights loss to fight class imbalance
max_iter	100 – 1 000	Extra iterations when convergence is slow
multi_class	auto, ovr, multinomial	Strategy for $k > 2$ classes

❖ Practical Considerations — Data Imbalance & Thresholds

When positive cases are rare (fraud, disease), plain logistic regression skews toward the majority class. Use class_weight='balanced' or supply custom weights, oversample the minority class (SMOTE), or tune the decision threshold on the validation ROC/PR curve to maximise F_1 or a cost-weighted metric.

❖ Pros and Cons

Pros

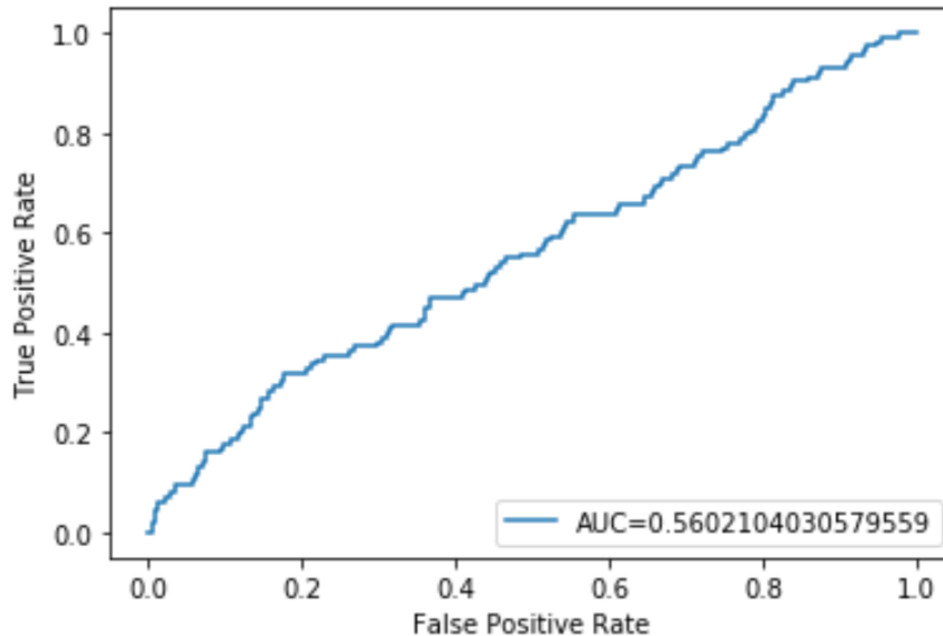
- Probabilistic outputs make threshold-tuning and ranking straightforward.
- Fast to train; scales to large, sparse text features.
- Coefficients are interpretable as log-odds ratios, useful for domain explanations.
- Supports ℓ_1, ℓ_2, and elastic-net regularisation for feature selection.

Cons

- Can only learn linear decision boundaries in the original feature space.
- Performance suffers when classes are heavily overlapping or non-linearly separable.
- Sensitive to multicollinearity; correlated features inflate variance of coefficients.
- Requires careful scaling of numeric predictors; otherwise optimisation slows or diverges.

❖ Typical Applications

- Spam vs. ham e-mail filtering
- Medical diagnosis (benign / malignant)
- Credit-default risk scoring
- Customer churn prediction



3ROC curve for a logistic-regression classifier on a binary dataset (AUC ≈ 0.56)

k-Nearest Neighbours (k-NN)

❖ Problem Type

Instance-based **classification *or* regression**.
Given a new example x_0, the algorithm consults the labelled training set, finds the k closest instances, and predicts either the majority class (classification) or the mean/median target value (regression).

❖ Core Idea

k-NN makes **no explicit model** during "training"—it stores the data verbatim.
At query time it measures the distance from x_0 to every stored point, selects the k smallest distances, and lets those neighbours "vote".

$$\hat{y}(x_0) = \begin{cases} mode\big(y_i : x_i \in \mathcal{N}_k(x_0)\big), \text{classification} \\ \dfrac{1}{k} \sum_{x_i \in \mathcal{N}_k(x_0)} y_i \text{, regression} \end{cases}$$

where $\mathcal{N}_k(x_0)$ is the set of the k nearest neighbours of x_0.

❖ Distance Metrics & Algorithmic Variants

Metric	Formula	Notes
Euclidean	$d(x,x') = \sqrt{\sum_j (x_j - x'_j)^2}$	Default for continuous, scaled features
Manhattan	$d = \sum_j \lvert x_j - x'_j \rvert$	x_j - x'_j
Minkowski (p-norm)	$d = \left(\sum_j \lvert x_j - x'_j \rvert^p\right)^{1/p}$	x_j - x'_j
Cosine similarity	$1 - \frac{x \cdot x'}{\lVert x \rVert \lVert x' \rVert}$	Text & high-dim sparse data
Hamming	$\sum_j 1[x_j \neq x'_j]$	Categorical / binary strings

Acceleration structures
– **KD-Tree** (low-dimensional, < ~20 D) – **Ball-Tree** (medium D) – **Approximate NN** libraries (ANNoy, FAISS) for very large n or high D.

❖ Hyper-parameters

Name	Typical range	Effect
n_neighbors (k)	1 – 50	Small *k* lowers bias, raises variance; large *k* does the reverse
weights	'uniform', 'distance'	Whether closer neighbours are weighted more
metric	see table above	Choice must match feature type & scaling
leaf_size	10 – 50	Trade-off between build time & query time in tree indexes
p (Minkowski)	1 or 2	Controls L^p norm (1 = Manhattan, 2 = Euclidean)

❖ Pros and Cons

Pros

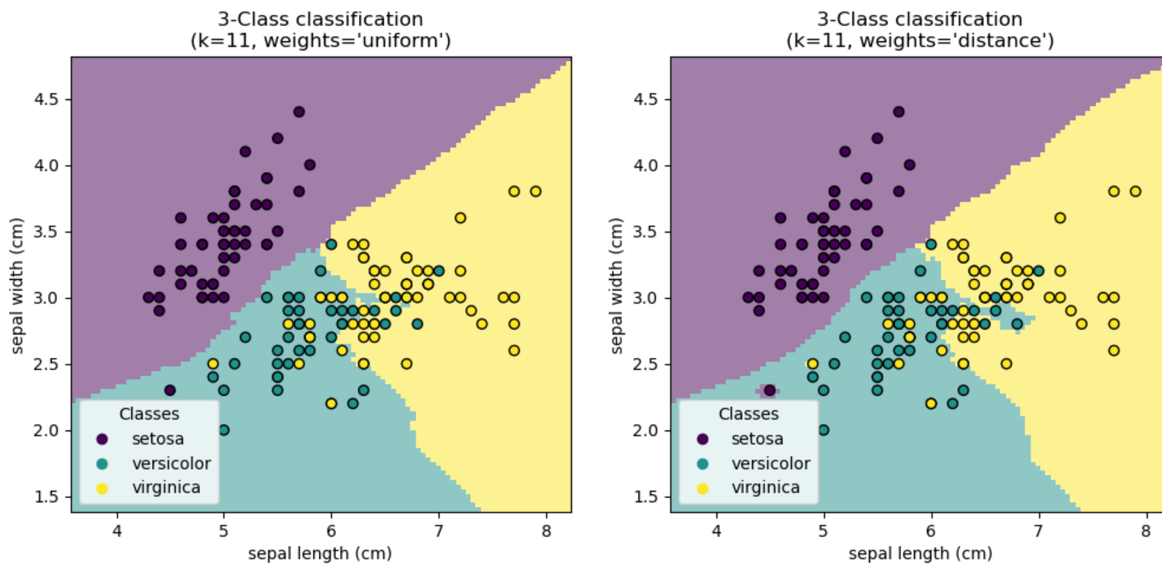
- **Zero training time:** model consists only of memorised examples.
- **Naturally multi-class** and works for both regression and classification.
- **Flexible decision boundaries:** with small *k*, can approximate any shape.

Cons

- **Query-time cost:** naïve search is $O(n \cdot d)$; scales poorly without an index.
- **Curse of dimensionality:** distances become less meaningful as dimensionality grows; feature selection or PCA often required.
- **Sensitive to feature scales and irrelevant attributes:** always standardise numeric variables and remove noise features.
- **Imbalanced classes:** majority class can dominate votes—use distance-weighting or stratified resampling.

❖ **Typical Applications**

- Handwritten-digit recognition (with small-n datasets)
- Recommender systems (item-to-item similarity)
- Anomaly detection via distance to nearest neighbour
- Simple baseline model for tabular competitions



4Decision boundaries learned by k-NN with k = 11 on the two-moons dataset.

<u>Decision Trees (CART)</u>

❖ **Problem Type**

Supervised **classification or regression**.
The model recursively partitions the feature space into axis-aligned rectangles (or hyper-rectangles) and assigns a constant prediction to every leaf.

❖ **Core Idea**

At every internal node pick the feature–threshold pair that maximises the impurity-reduction (classification) or minimises the mean-squared error (regression).
Splitting stops when a designated stopping rule (max-depth, min-samples, cost-complexity pruning) is met.

❖ Hypothesis (Model Form)

A tree T is a set of decision nodes and terminal leaves:

- **Classification:** each leaf stores a class-probability vector $\hat{p}(c \mid \text{leaf})$; prediction = $\arg\max_c \hat{p}(c)$.
- **Regression:** each leaf stores the mean target of its samples $\hat{y}_{\text{leaf}}$; prediction = that constant value.

Each internal node tests
$x_j \leq \theta$ **or** $x_j > \theta$; paths define axis-aligned regions.

❖ Cost / Loss Function

Task	Node impurity to minimise	Formula
Classification	Gini	$G(t) = 1 - \sum_c p_c^2$
	Entropy	$H(t) = -\sum_c p_c \log p_c$
Regression	Mean-squared error	$\text{MSE}(t) = \frac{1}{\lvert t \rvert} \sum_{i \in t} (y_i - \bar{y}_t)^2$

Split-gain (classification)

$$\text{Gain} = I(\text{parent}) - \frac{n_L}{n} I(L) - \frac{n_R}{n} I(R).$$

Cost-complexity pruning

$$R_\alpha(T) = \text{Error}(T) + \alpha \lvert T \rvert,$$

remove sub-trees whose removal lowers R_α.

❖ Optimiser

- Start with the full training set at the root.
- **For each candidate feature–threshold:** compute impurity (or MSE) gain.
- Pick the best split, create left/right children.

- Recurse on each child until any stopping rule triggers.
- (Optional) **Prune** bottom-up via cost-complexity $R_\alpha(T)$.

Greedy top-down induction runs in $O(n\,p\log n)$ (with presorted features).

❖ Hyper-parameters

Parameter	Typical range	Effect
criterion	'gini', 'entropy', 'log_loss' / 'squared_error'	Split quality metric
max_depth	3 – None	Limits model capacity; controls over-fit
min_samples_split	2 – 50	Fewer splits ⇒ shallower tree
min_samples_leaf	1 – 50	Forces each leaf to carry enough data
max_features	None, 'sqrt', 'log2', int	Random-subspace effect (RF similarity)
ccp_alpha	0 – 0.02	Post-pruning strength (cost-complexity)
class_weight	None, 'balanced', dict	Re-weights impurity for imbalanced classes

❖ Pros and Cons

Pros

- **Interpretability** – path from root to leaf reads as an IF/THEN rule.
- **Handles mixed data types** without scaling or dummy variables.
- **Works with missing values** (can send NaNs left/right or by surrogate splits).
- **Captures non-linear interactions** automatically.

Cons

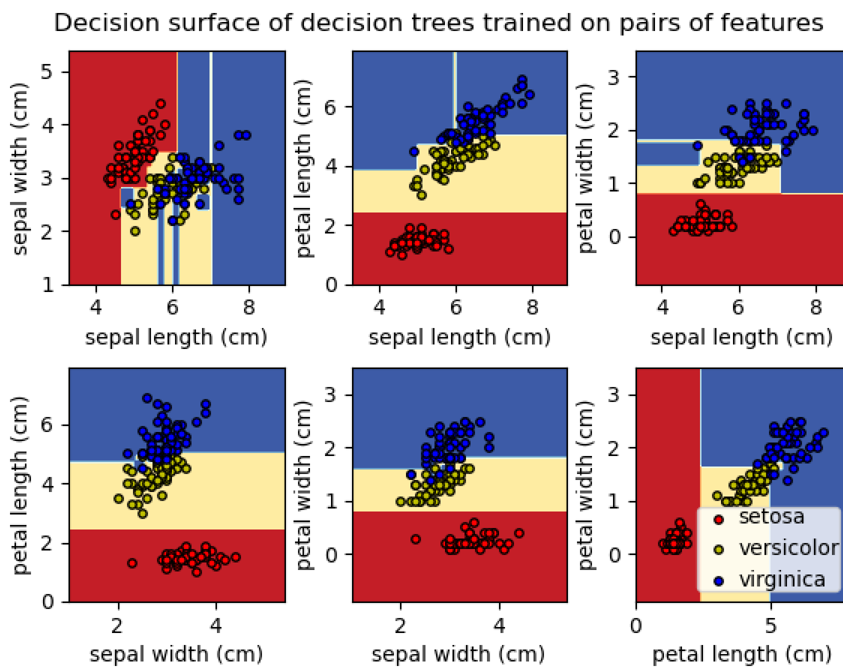
- **Prone to over-fitting**; deep trees memorise noise.
- **Unstable** – small data perturbations can change structure markedly.
- **Greedy splits** may miss globally optimal partitions.
- **Biased toward features with many levels** unless max_features or one-hot control is used.

❖ Handling Class Imbalance

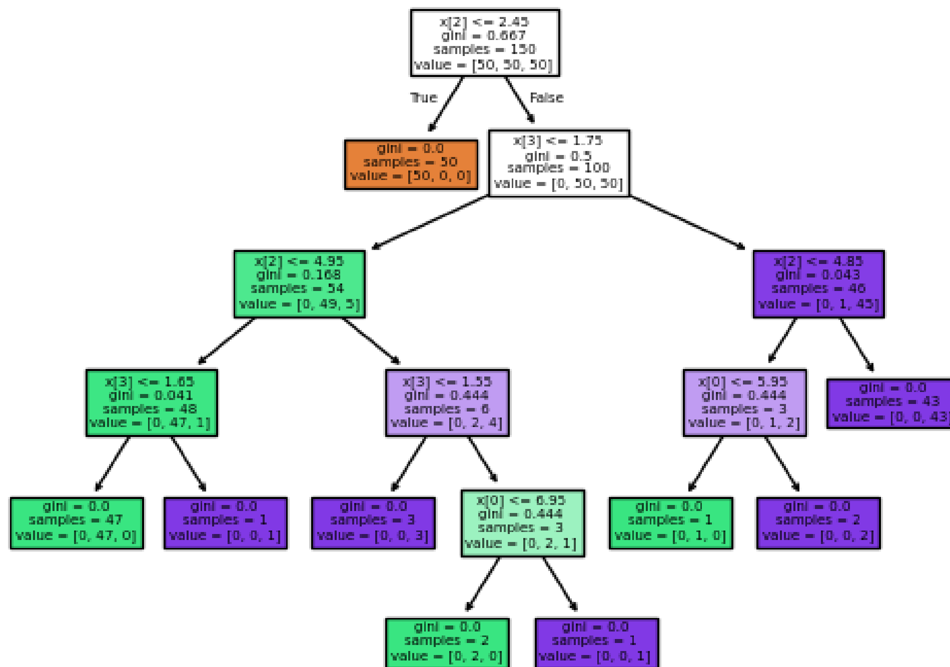
During each split, impurity is weighted by class counts; highly skewed datasets still bias toward the majority.
Mitigations: set class_weight = 'balanced', supply custom weights, or down/over-sample before fitting.

❖ Typical Applications

• Credit-risk scorecards (simple tree)
• Medical triage rules
• Feature-importance base learner for Random Forests & Gradient Boosting
• Initial baseline on tabular tasks



5Decision surfaces learned by shallow decision trees for every pair of Iris flower features; colored regions show the species each region is classified as.



6Depth-3 CART model fitted to all four Iris features, illustrating axis-aligned splits and class counts at each leaf.

Random Forests

❖ **Problem Type**

Supervised **classification and regression**.
A Random Forest (RF) is an **ensemble** of decision-tree base-learners whose individual predictions are averaged (regression) or majority-voted (classification).

❖ **Core Idea & Variance-Reduction Math**

1. **Bootstrap aggregation (bagging)** Draw T bootstrap samples; fit one CART on each.
2. **Feature sub-sampling** At every split, each tree considers only a random subset of features ($\sqrt{D}$ for classification, $D/3$ for regression by default).
   These two sources of randomness decorrelate trees; averaging then slashes variance.

❖ **Hypothesis (Model Form)**

$$\hat{f}_{RF}(x) = \frac{1}{T}\sum_{t=1}^{T} f_t(x)\,,$$

where each f_t is an axis-aligned decision tree trained on its own bootstrap sample and random-feature subsets.

Classification: final label = $mode\{f_t(x)\}$.

Regression: final prediction = the arithmetic mean above.

❖ **Cost / Loss Function**

Level	Loss minimised
Individual tree	Same split criterion as CART: Gini/Entropy (classification) or MSE (regression).
Ensemble	No global differentiable loss; performance is assessed by OOB error or validation set. Variance of the averaged predictor is $Var\left[\widehat{f_{\mathrm{RF}}}(x)\right] = \frac{1}{T}\,\bar{\sigma}^2(1 + (T-1)\rho)$ where $\bar{\sigma}^2$ is mean tree variance and ρ their pair-wise correlation.

❖ **Optimiser**

- **Training loop**
    1. Sample data with replacement (bootstrap).
    2. Grow a full, unpruned CART using greedy impurity gain; at each node evaluate only max_features random predictors.
    3. Repeat for T trees in parallel (bagging).
- **Prediction** Aggregate tree outputs by mean (regression) or vote (classification).
- **OOB estimate** For each sample, average predictions from trees that did not see it—gives a built-in validation score without extra cross-validation.

❖ **Hyper-parameters**

Parameter	Typical range	Effect
n_estimators	100 – 1 000	More trees → lower variance, higher cost
max_depth	None, 5 – 50	Controls over-fit; shallow trees faster
min_samples_leaf	1 – 50	Larger leaves smooth noisy splits
max_features	'sqrt', 'log2', int/float	Smaller ⇒ trees less correlated
bootstrap	True/False	Enable bagging; False = full sample

oob_score	True	Adds out-of-bag validation (no extra CV)
class_weight	None, 'balanced', dict	Re-weights impurity for imbalanced classes

❖ Pros and Cons

Pros

- Handles **non-linear interactions** and mixed feature types with almost no preprocessing.
- **Robust** to outliers and to over-fitting compared with a single tree.
- Provides **feature-importance** scores and OOB (out-of-bag) error estimate "for free."
- Parallelisable both at training (trees) and inference (votes).

Cons

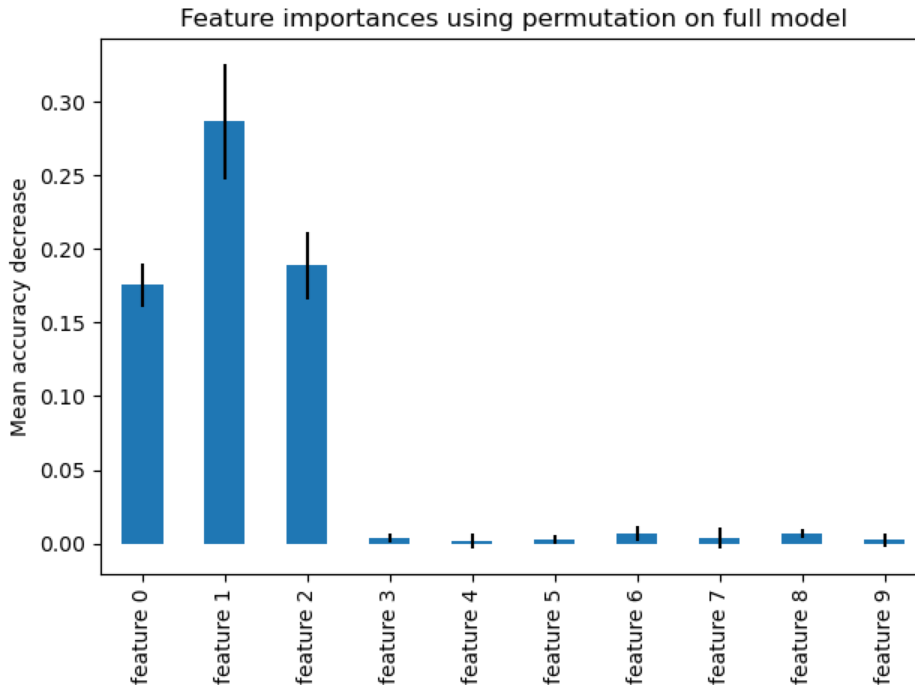
1. Large memory footprint: hundreds of deep trees stored.
2. **Slower inference** than boosted trees with few trees or linear models.
3. Interpretability limited to global/importances or SHAP; individual paths are many.
4. Still **biased on severe class imbalance** unless class weights or sampling are applied.

❖ Handling Class Imbalance

Set class_weight='balanced' or pass a custom weight dict so minority-class errors count more in the split criterion.
Alternatively oversample (SMOTE) or undersample before fitting—bagging amplifies resampling benefits.

❖ Typical Applications

• Tabular Kaggle leaderboards • Credit-risk and fraud detection • Remote-sensing land-cover mapping
• Genomic feature ranking



7Feature-importance rankings from a 300-tree Random Forest (mean decrease-in-impurity).

Gradient-Boosted Trees

❖ Problem Type

Supervised **regression and classification**.
A gradient-boosting model builds an additive ensemble of shallow decision trees, each one trained to correct the residual errors of the trees that came before it.

❖ Core Idea

Start with a constant prediction $F_0(x)$.
For each boosting round $m = 1, \dots, M$:

1. Compute pseudo-residuals

$$r_i^{(m)} = -\left[\frac{\partial \mathcal{L}(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{m-1}},$$

   the **negative gradient** of the loss w.r.t. current model output.

2. **Fit a new tree** $h_m(x)$ to these residuals.
3. Update the ensemble

   $F_m(x) = F_{m-1}(x) + \eta h_m(x),$
   where $0 < \eta \leq 1$ is the learning rate (shrinkage).

With a small η and many trees, the ensemble approximates complex, non-linear functions while keeping individual trees simple (depth ≤ 8).

❖ **Hypothesis (Model Form)**

$$\hat{y} = F_M(x) = F_0(x) + \sum_{m=1}^{M} \eta \, h_m(x),$$

where each h_m is a decision tree with limited depth/leaf count.
Thus the hypothesis class is **additive, stage-wise sums of weak CART learners**.

❖ **Cost / Loss Function**

The ensemble minimises a differentiable loss over the training set:

$$\mathcal{L}(\{h_m\}) = \sum_{i=1}^{n} \mathcal{L}\big(y_i, F_M(x_i)\big),$$

- **Regression** $\mathcal{L} = (y - \hat{y})^2/2$ (squared error)
- **Binary classification** $\mathcal{L} = \log\left(1 + e^{-y\hat{y}}\right)$ (log-loss)

At each boosting iteration the negative gradient
$r_i^{(m)}$ serves as the **target** for the next tree.

❖ **Optimiser**

- **Stage-wise gradient boosting algorithm**
  1. Compute pseudo-residuals $r_i^{(m)}$.
  2. Fit a CART $h_m(x)$ to $\{\left(x_i, r_i^{(m)}\right)\}$.
  3. Update $F_m(x) = F_{m-1}(x) + \eta h_m(x)$.
- **Stochastic variants** Set subsample < 1 and/or colsample_bytree to inject randomness and reduce over-fitting.
- **Early stopping** Monitor validation loss; stop when it fails to improve for early_stopping_rounds.

❖ **Hyper-parameters**

Parameter	Typical range	Effect
n_estimators	100 – 2 000	More rounds = higher capacity
learning_rate	0.01 – 0.3	Lower η → needs more trees but generalises better
max_depth / max_leaf_nodes	3 – 10	Controls individual tree complexity
subsample	0.5 – 1.0	Row sampling per round (stochastic boosting)
colsample_bytree / max_features	0.5 – 1.0	Column sampling — decorrelates trees
min_child_weight, min_samples_leaf	1 – 20	Minimum Hessian / sample count in a leaf
l2_regularization, lambda	0 – 10	Penalise large leaf scores
early_stopping_rounds	10 – 100	Halts when validation loss stops improving

❖ Pros and Cons

Pros

- Among the **highest-accuracy** algorithms for structured/tabular data.
- Handles missing values and mixed feature types out-of-the-box.
- Built-in *feature importance* and SHAP values aid interpretability.
- Hyper-parameters allow for a trade-off between speed, accuracy, and model size.

Cons

- Training can be **slow** and memory-intensive without tuning.
- Sensitive to *learning-rate / tree-depth* combo—needs cross-validation.
- Less interpretable than linear models; single prediction path is opaque.
- Can still over-fit noisy labels if early stopping or regularisation is ignored.

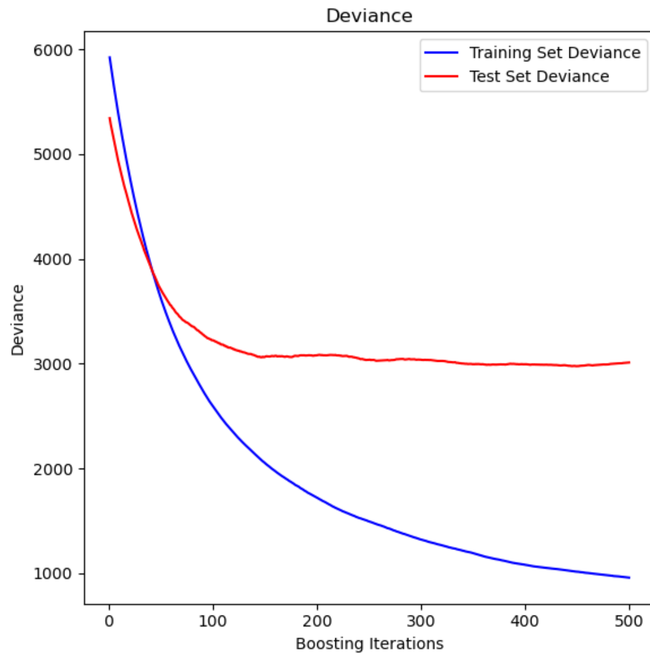
❖ Handling Class Imbalance

Use a **scale-positive-weight** (XGBoost) or class_weight (HistGB) to raise the loss contribution of minority samples.
Alternative: oversample minority class before training or optimise a class-balanced loss (e.g., focal loss).

❖ Typical Applications

greentecAI

• Kaggle/tabular competitions • Click-through-rate prediction • Credit-risk and fraud detection • Genomic risk scores • Industrial sensor fault diagnostics



8Training & Test deviance plotted against the number of boosted trees.

Unsupervised Learning

Unsupervised methods discover structure hidden inside unlabeled data—clustering, dimensionality reduction, anomaly detection, association rules, and more.
We begin with the work-horse clustering algorithm **k-Means**.

Clustering & Dimensionality-Reduction Metrics

Metric	Formula	Range	When to prefer / interpret
Silhouette Score	$\frac{b(i)-a(i)}{\max\{a(i),\, b(i)\}}$	–1 … 1	Internal measure; higher = denser, well-separated clusters
Davies–Bouldin Index	$\frac{1}{k}\sum_{i=1}^{k}\max_{j\neq i}\frac{\sigma_i+\sigma_j}{d(c_i,c_j)}$	0 … ∞ (lower better)	Penalises clusters with high intra/low inter distance; **scale-free**
Calinski–Harabasz Index	$\frac{\mathrm{tr}(B_k)}{\mathrm{tr}(W_k)}\cdot\frac{n-k}{k-1}$	0 … ∞ (higher better)	Good for k-selection via elbow; fast to compute
ARI – Adjusted Rand Index	$\frac{\mathrm{RI}-\mathrm{E}[\mathrm{RI}]}{\max(\mathrm{RI})-\mathrm{E}[\mathrm{RI}]}$	–1 … 1	Compare clustering to **ground truth** labels; baseline≈0

AMI – Adjusted Mutual Info.	$\frac{\text{MI} - \text{E[MI]}}{\max(H_U, H_V) - \text{E[MI]}}$	0 … 1	Handles many clusters; label permutations invariant
Explained-Variance Ratio *(PCA)*	$\frac{\sum_{j=1}^{q} \lambda_j}{\sum_{j=1}^{d} \lambda_j}$	0 … 1	Choose #PCs; 0.90–0.99 common for denoising

Notes:

- $a(i)$ = mean distance of point *i* to its own cluster; $b(i)$ = mean distance to nearest other cluster.
- σ_i = avg. intra-cluster distance of cluster i; c_i = its centroid.
- B_k, W_k = between- and within-cluster scatter matrices.
- λ_j = eigenvalues (principal-component variances).

k-Means Clustering

❖ **Problem Type**

Partition *n* unlabeled samples into **k disjoint clusters** so that points in the same cluster are more similar to each other than to points in different clusters.

❖ **Core Idea**

Iteratively assign each sample to the nearest cluster centroid, then recompute centroids as the mean of current members, until assignments stop changing or a maximum iteration count is reached.

❖ **Hypothesis**

- **Centroids** $\boldsymbol{\mu} = \{\mu_1, \dots, \mu_k\}, \mu_j \in R^d$
- **Cluster assignment function** $c(i) \in \{1, \dots, k\}$ maps each sample x_i to its nearest centroid by Euclidean distance.

❖ **Cost / Loss Function**

$$\mathcal{L}(\mu) = \sum_{i=1}^{n} \left\| x_i - \mu_{c(i)} \right\|_2^2,$$

i.e. the **within-cluster sum-of-squared errors (inertia)**, which k-Means seeks to minimise.

❖ **Objective Function**

$$\min_{\{\mu_j\}} \sum_{i=1}^{n} \left\| x_i - \mu_{c(i)} \right\|_2^2$$

where $c(i)$ is the cluster index of point x_i and μ_j is the centroid of cluster j.

❖ **Optimiser**

Greedy **Expectation–Maximisation** loop:

1. **Initialise centroids** (random or *k-means*++).
2. **E-step** Assign each x_i to the nearest centroid $c(i)$.
3. **M-step** Update each centroid
$$\mu_j \leftarrow \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$$
4. Repeat steps 2 – 3 until assignments stop changing or inertia drops below tol.
5. Run the whole procedure n_init times with different seeds; keep the lowest-inertia solution.

Time complexity per iteration: $O(n\,k\,d)$; MiniBatchKMeans subsamples data to scale to millions of points.

❖ **Hyper-parameters**

Parameter	**Typical values**	**Effect**
n_clusters (k)	2 – 20	Number of clusters expected
init	'k-means++', 'random', ndarray	Startup centroids; k-means++ reduces bad local minima
n_init	10 – 100	Repeats with different seeds; best inertia kept
max_iter	100 – 1 000	Safety cap on EM iterations
tol	10^{-4} – 10^{-2}	Convergence threshold on inertia
algorithm	'lloyd', 'elkan', 'auto'	Lloyd = classic, Elkan = triangle-inequality speed-up

❖ **Pros and Cons**

Pros

- **Fast and scalable** (O(n·k·d) per iteration; MiniBatch scales to millions).
- Simple, widely implemented, easy to interpret centroids.
- Works well when clusters are spherical and equally sized.

Cons

- Requires pre-specifying k; wrong k ⇒ misleading clusters.
- Sensitive to initialisation and outliers; can converge to local minima.

- Assumes isotropic variance—fails on elongated or density-varying clusters.
- Feature scaling crucial: Euclidean distance dominates unscaled numeric values.

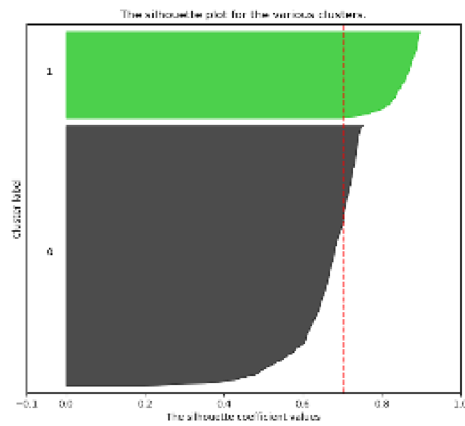
❖ Evaluating Cluster Quality

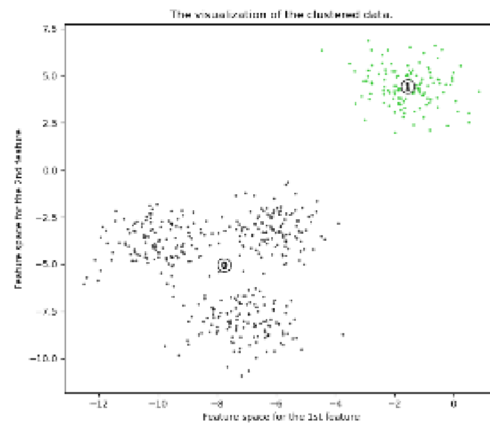
Common internal indices: inertia (within-cluster SSE), silhouette score (−1 to 1), Calinski–Harabasz, Davies–Bouldin.
When ground truth exists, use adjusted Rand index (ARI) or mutual information.

❖ Typical Applications

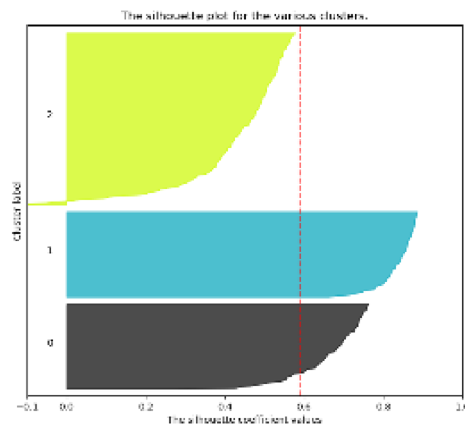
• Customer segmentation in marketing analytics
• Image compression via vector quantisation
• Initial centroids for Gaussian Mixture EM
• Anchor points for bag-of-visual-words in computer vision

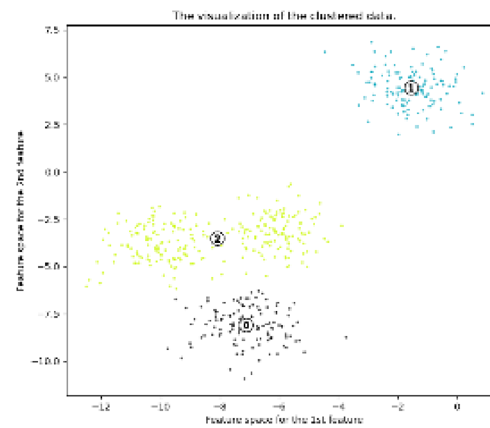
Silhouette analysis for KMeans clustering on sample data with n_clusters = 2



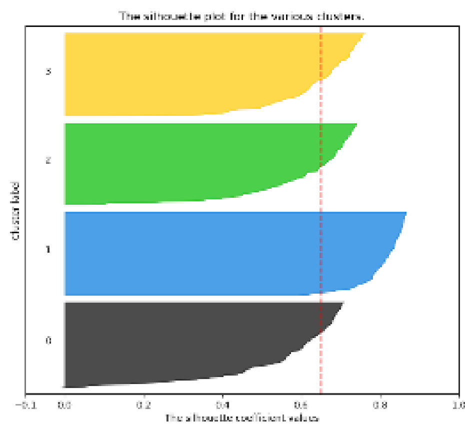


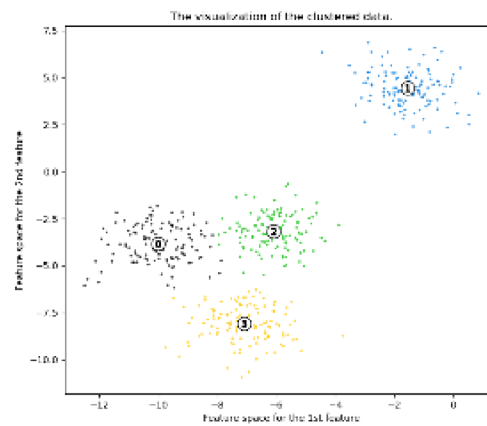
Silhouette analysis for KMeans clustering on sample data with n_clusters = 3

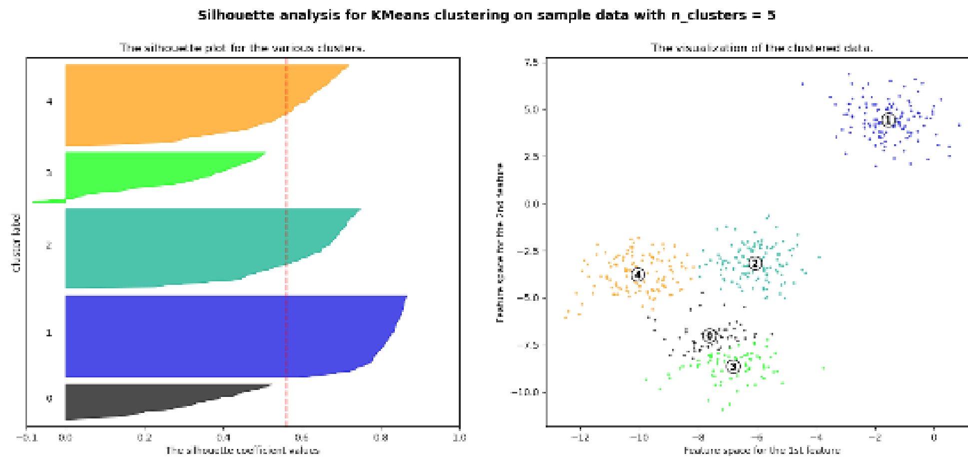


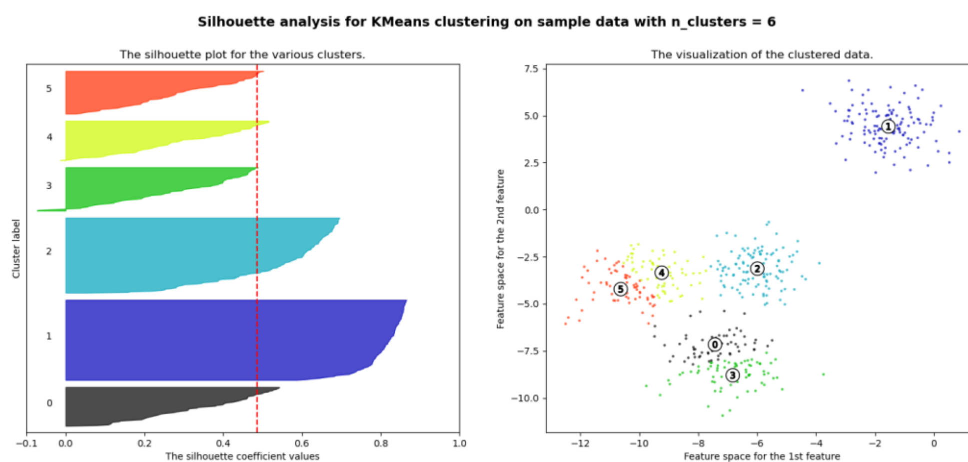


Silhouette analysis for KMeans clustering on sample data with n_clusters = 4









9Elbow (inertia) and silhouette curves suggesting an optimal cluster count of k ≈ 3.

Hierarchical Clustering

❖ Problem Type

Unsupervised clustering that produces a nested sequence of partitions, visualised as a dendrogram. Cuts taken at different heights yield clusterings at different granularity.

❖ Core Idea

Start with every sample in its own singleton cluster (agglomerative) or all samples in one cluster (divisive).
Iteratively merge (or split) the two clusters whose inter-cluster dissimilarity is minimal under a chosen linkage criterion until a single root (or desired number of clusters) is reached.

❖ Linkage Criteria

- **Single linkage** — distance between nearest points (can produce "chaining").
- **Complete linkage** — distance between farthest points (compact, equal-sized clusters).
- **Average linkage** — mean pair-wise distance (compromise between single/complete).

- **Ward linkage** — minimises increase in total within-cluster variance; preferred for Euclidean data.

❖ **Algorithm Sketch**

1. Compute initial **distance matrix** $D^{(0)}$ for all n points.
2. **Repeat** until one cluster remains:
    - Find the pair (A, B) with minimal $D(A, B)$.
    - Merge A and B into C.
    - Update the distance matrix using the chosen linkage formula.
3. Store merge order and distances to build a dendrogram.

Complexity: $O(n^2 logn)$ with efficient heap + nearest-neighbor chain, otherwise $O(n^3)$.

❖ **Hyper-parameters**

- n_clusters — cut height automatically when provided.
- distance_threshold — alternative to n_clusters; cut where dendrogram height ≤ threshold.
- linkage — 'ward', 'complete', 'average', 'single'.
- metric — 'euclidean' (Ward requires), 'manhattan', 'cosine', custom callable.
- memory — caches distance matrix for large datasets.

❖ **Pros and Cons**

Pros

- Produces a full **hierarchy**; user can inspect clusters at any level.
- No need to pre-specify *k* if a dendrogram cut is chosen visually.
- Works with arbitrary distance metrics and non-spherical cluster shapes.

Cons

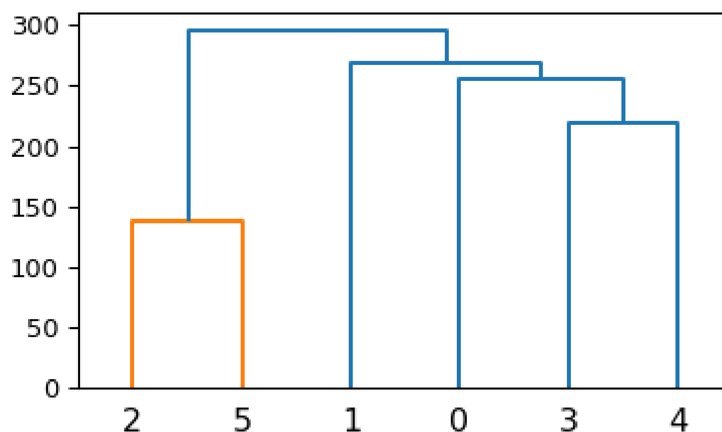
- Quadratic (or worse) memory/time for large *n*—unsuitable for > 50 000 points without approximations.
- Sensitive to noisy points and small variations in distance metric.
- Once a merge is made it cannot be undone (greedy).

❖ **Evaluating Cluster Quality**

Use cophenetic correlation coefficient to measure how faithfully the dendrogram preserves pair-wise distances, or compute silhouette / Davies–Bouldin after selecting a cut to compare with other clustering algorithms.

❖ **Typical Applications**

- Gene-expression analysis (heat-map + dendrogram)
- Customer taxonomy building in marketing analytics
- Taxonomy of documents or images when tree-like grouping is meaningful
- Anomaly detection: small branches at large heights often flag outliers



10Ward-linkage dendrogram

DBSCAN – Density-Based Spatial Clustering of Applications with Noise

❖ **Problem Type**

Unsupervised density-based clustering that discovers arbitrarily shaped clusters and labels points in sparse regions as noise / outliers.

❖ **Core Idea**

A point is a **core point** if at least *minPts* neighbors fall within an ε-radius (ε-neighborhood).
Clusters grow by recursively linking density-reachable points:

- **Directly density-reachable:** $p \to q$ if q is inside the ε-neighborhood of core point p.
- **Density-reachable:** a chain of direct links connects p to q.
- **Noise:** points that are neither core nor density-reachable from any core.

❖ **Algorithm Sketch**

1. Label all points "unvisited."
2. For each unvisited point p
   a. Mark-as-visited; find its ε-neighbors.
   b. If neighbors < *minPts* → label noise and continue.
   c. Else create new cluster; add p and all density-reachable neighbors via a flood-fill queue.
3. Repeat until all points are visited.

Complexity:
– Naïve search $O(n^2)$
– With KD/Ball-Tree or spatial index ≈ $O(nlogn)$ in low dimension.

❖ **Hyper-parameters**

Parameter	**Typical range**	**Effect**
eps	0.1 – 5 (depends on scale)	Radius of ε-neighborhood
min_samples (*minPts*)	4 – 10	Minimum points to form a core
metric	'euclidean', 'manhattan', 'cosine', callable	Distance function
leaf_size (for tree)	20 – 40	Speed vs memory in KD/Ball-Tree
algorithm	'auto', 'ball_tree', 'kd_tree', 'brute'	Search backend

❖ **Pros and Cons**

Pros

- Detects clusters of arbitrary shape; no need to pre-specify *k*.
- Explicitly identifies noise/outliers.
- Invariant to cluster ordering; deterministic given ε & *minPts*.
- Works with any distance metric.

Cons

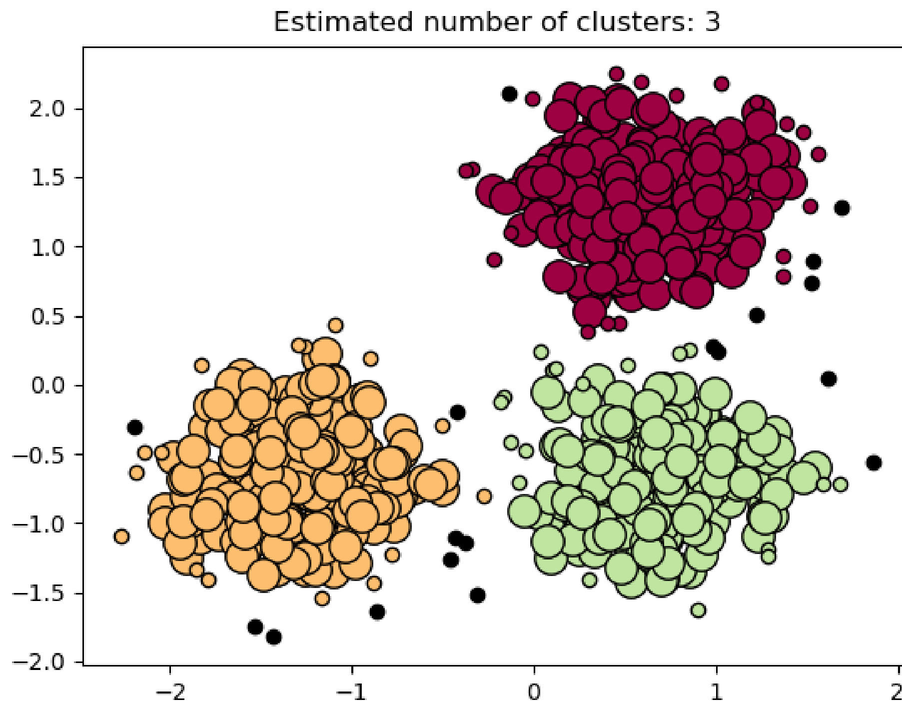
- Sensitive to the single-scale ε choice; varying densities can break it.
- Memory/time quadratic in high dimensions (curse of dimensionality).
- Struggles when clusters strongly overlap or data are extremely sparse.
- Difficult to combine with categorical features (distance definition).

❖ **Evaluating Cluster Quality**

Use silhouette score on the non-noise subset, cluster vs noise ratio, or compare to ground truth with adjusted Rand index after removing noise labels.

❖ **Typical Applications**

• Geo-spatial hotspot detection (earthquake epicenters, crime)
• Image feature clustering in SIFT/SURF pipelines
• Fraud or network intrusion anomaly detection
• Filtering noise before running k-Means or GMMs



11. DBSCAN result on a synthetic three-blob dataset: core samples are shown as coloured circles (maroon, green, and orange), while black points represent noise

Dimensionality Reduction – PCA and t-SNE

❖ **Problem Type**

Unsupervised feature extraction / visualisation.
Reduce a high-dimensional dataset $X \in \mathbb{R}^{n \times d}$ to a low-dimensional representation $Z \in \mathbb{R}^{n \times q}$ (with $q \ll d$) that preserves as much "interesting structure" as possible—variance for PCA, local neighbourhood geometry for t-SNE.

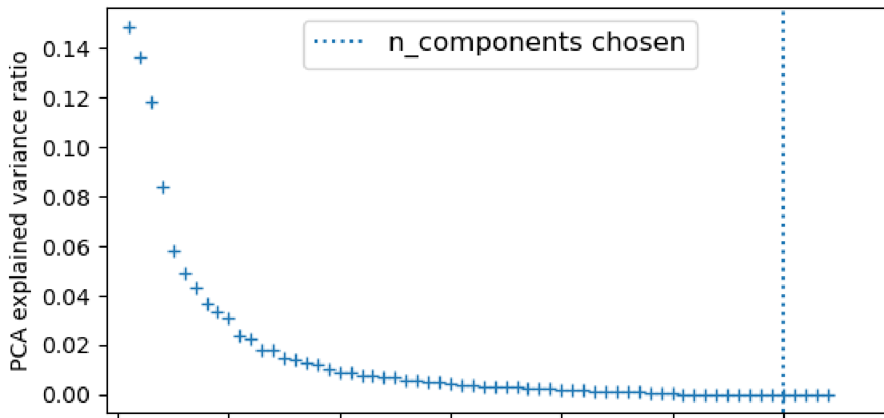
❖ **Core Ideas**

Principal Component Analysis (PCA)
Project the data onto the orthogonal directions of maximum variance.

1. Centre X.

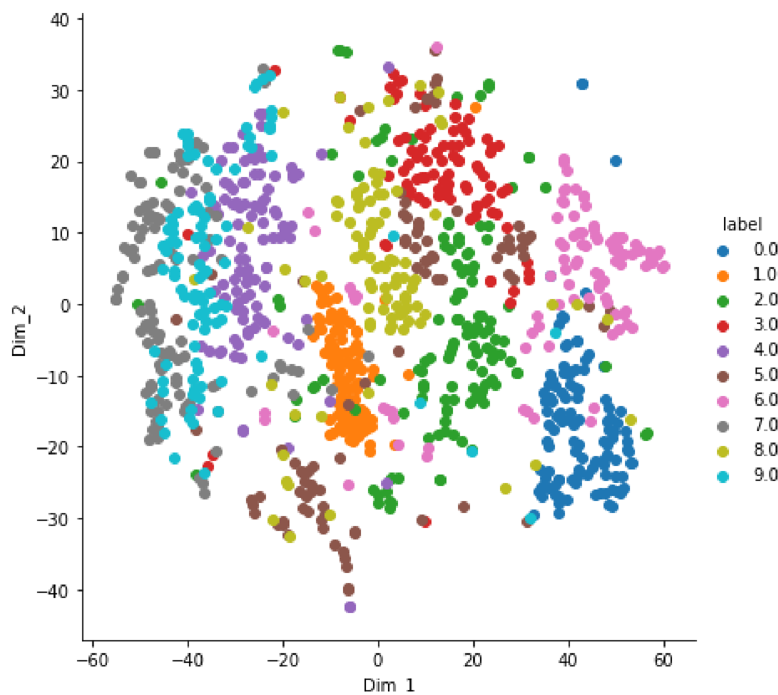
2. Solve the eigen-decomposition of the covariance matrix $\frac{1}{n}X^{\top}X = V\Lambda V^{\top}$.

3. The first q eigenvectors V_q give the projection $Z = XV_q$. Variance retained = $\sum_{j=1}^{q}\lambda_j \,/\, \sum_{j=1}^{d}\lambda_j$.



12. Explained-variance ratio for the first 64 principal components of the Digits dataset

t-Distributed Stochastic Neighbour Embedding (t-SNE)
Convert high-dimensional pair-wise distances into probability similarities and minimise the Kullback–Leibler divergence between those and a Student-t (ν = 1) distribution in the low-dim space, thereby preserving local neighbourhoods for visualisation.



13. 2-D t-SNE embedding of MNIST digits; colour encodes digit class

❖ Hypothesis (Model Form)

PCA
Find an orthonormal projection matrix $W_q \in \mathbb{R}^{d\times q}$ so that $Z = XW_q$.

t-SNE
Learn low-dim coordinates $Y \in \mathbb{R}^{n\times q}$ (usually q=2) whose Student-t similarities Q_{ij} approximate high-dim Gaussian similarities P_{ij}.

❖ Cost / Loss Function

PCA
Reconstruction (variance) loss: $\mathcal{L}_{\text{PCA}} = \frac{1}{n} | X - XW_q W_q^\top |_F^2$.

t-SNE
Kullback–Leibler divergence: $\mathcal{L}_{\text{t-SNE}} = \sum_{i\neq j} P_{ij} \log \frac{P_{ij}}{Q_{ij}}$.

❖ Optimiser

- **PCA** – closed-form via SVD/eigen-decomposition; take the first q eigenvectors.
- **t-SNE** – gradient descent with momentum; Barnes–Hut or FFT acceleration; perplexity sets bandwidth of P_{ij}.

❖ Hyper-parameters

Model	Key params	Typical values	Effect
PCA (sklearn.PCA)	n_components	0.90 – 0.99 (variance) or 2–100 (int)	Number of PCs or variance to retain
	whiten	False, True	Scales PCs to unit variance (useful before ICA)
t-SNE (sklearn.manifold.TSNE)	perplexity	5 – 50	Balances local vs global structure
	learning_rate	50 – 1 000	Too low: crowding; too high: loss spikes
	n_iter	250 – 2 000	More iterations → smoother embedding
	Init	'pca', 'random'	PCA init speeds convergence

❖ Pros and Cons

PCA – Pros: fast SVD; linear mapping for new points; denoises; reveals latent factors.
Cons: only captures linear variance; components sometimes hard to interpret.

t-SNE – Pros: excellent 2-D visual "cluster pop-out"; handles non-linear manifolds.
Cons: purely visual (no transform for new data); sensitive to perplexity; slow on > 50 k points.

❖ **Typical Applications**

• Pre-whitening before ICA or clustering
• Noise reduction in image pipelines
• 2-D embedding of word-vectors and MNIST digits
• Anomaly detection using top-k PCs' residuals

Reinforcement Learning

Reinforcement Learning (RL) studies how an agent can learn to make a sequence of decisions by interacting with an environment and maximising long-term reward. Unlike supervised learning, there are no labelled examples; instead, the agent must discover which actions yield the most cumulative reward through trial and error.

Agent–Environment Interaction Loop

1. **Observe state** s_t (e.g., grid-world cell, robot pose, game frame).
2. **Choose action** a_t from the available action set A.
3. **Environment transitions** to new state s_{t+1} and emits **reward** r_{t+1}.
4. **Agent updates** its policy or value estimates based on $(s_t, a_t, r_{t+1}, s_{t+1})$.
5. Repeat until a terminal state or horizon is reached; the agent's goal is to maximise the **return**

$$G_t = \sum_{k=0}^{\infty} \gamma^k \, r_{t+1+k},$$

where $\gamma \in [0,1]$ is the **discount factor** weighting immediate vs future rewards.

Markov Decision Process (MDP) Formalism

An MDP is defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$:

- $\mathcal{S}$: set of states.
- $\mathcal{A}$: set of actions.
- $P(s' \mid s, a)$: transition probability to next state s′s's′.
- $R(s, a, s')$: expected reward for taking aaa in sss and arriving at s′s's′.
- γ: discount factor.

The **optimal action-value function** satisfies the Bellman optimality equation

$$Q^*(s,a) \;=\; R(s,a) + \gamma \sum_{s'} P(s' \mid s, a) \max_{a'} Q^*(s', a').$$

Policy, Value Functions, and Exploration

- **Policy** $\pi(a \mid s)$: mapping from states to action probabilities (deterministic or stochastic).
- **State-value** $V^{\pi}(s) = E_{\pi}[G_t \mid s_t = s]$.
- **Action-value** $Q^{\pi}(s, a)$: expected return after taking action a in state s and thereafter following π.
- **Exploration vs exploitation**: the agent must balance trying new actions (explore) and choosing the best-known action (exploit). Common strategies: ε-greedy, softmax/Boltzmann, Upper-Confidence Bound (UCB).

RL Algorithm Taxonomy

Axis	Categories	Example algorithms
Model use	Model-free vs Model-based	Q-Learning vs Dyna-Q
Update target	Value-based vs Policy-based vs Actor–Critic	Q-Learning / SARSA vs REINFORCE vs A2C
On-policy vs Off-policy	Learns about the policy being executed (SARSA) vs learns a separate target policy (Q-Learning, DQN)	
Tabular vs Function Approx.	State table vs neural network	Q-table vs Deep Q-Network

Evaluation Metrics in RL

- Average return per episode (episodic tasks).
- Cumulative reward over horizon (continuing tasks).
- Sample efficiency: reward as a function of environment interactions.
- Stability & variance across random seeds.
- Wall-clock training time when comparing deep RL algorithms.

Tabular Q-Learning

❖ Problem Type

Episodic or continuing control problems with small, discrete state–action spaces (e.g., 48 states × 4 actions grid-world). The agent seeks a policy π^* that maximises the expected discounted return.

❖ Core Idea

Keep an explicit table $Q(s, a)$ of action-values.
After every transition the agent **bootstraps**—moving the current cell-estimate toward a one-step look-ahead target that uses the *best* action in the *next* state. This makes Q-Learning **off-policy**: it can learn the greedy policy while following an exploratory one (ε-greedy, softmax, etc.).

❖ Hypothesis (Model Form)

$$Q: S \times A \longrightarrow \mathbb{R},$$

a lookup table whose entry $Q(s, a)$ estimates the expected return starting from state s, taking action a, and thereafter following a greedy policy.

❖ **Cost / Loss Function**

The update minimises the **temporal-difference (TD) error**

$$\delta = r_{t+1} + \gamma \max_{a'} Q\,(s_{t+1}, a') - Q(s_t, a_t),$$

often viewed as reducing the squared loss $\frac{1}{2}\delta^2$ for the visited state–action pair.

❖ **Optimiser**

One-step **Q-Learning update**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta,$$

applied online after every transition.

- **Exploration** ε-greedy or soft-max; ε decays over time.
- **Convergence** guaranteed for finite MDPs under diminishing α and sufficient visitations.

❖ **Hyper-parameters**

Name	Typical range	Effect
α – learning rate	0.1 – 0.7	High α ⟶ faster learning, higher variance
γ – discount factor	0.90 – 0.99	Lower γ favours short-term reward
ε – exploration (ε-greedy)	Start 0.9 ⟶ decay to 0.01	Controls explore / exploit balance
Episode length	env-specific	Too short truncates learning; too long delays updates
Initial Q(s,a)	0 or optimistic (1)	Optimistic starts encourage exploration

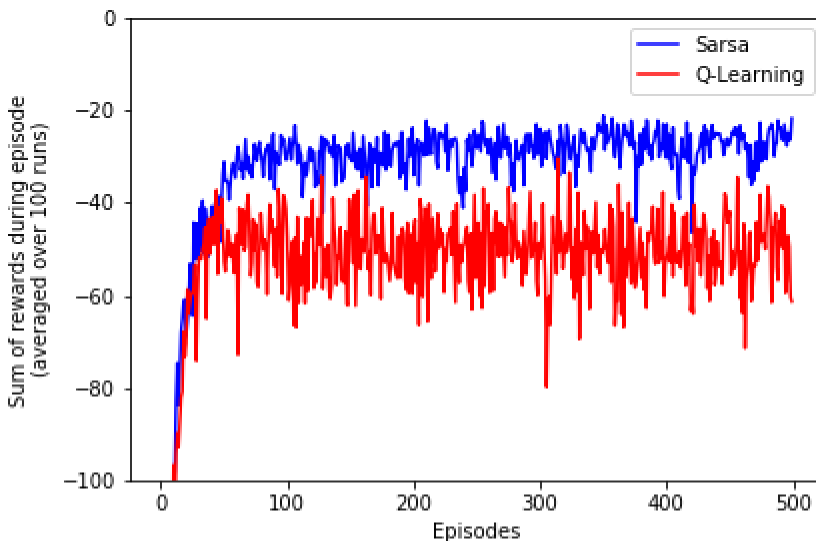
❖ **Pros and Cons**

Pros – Simple, provably convergent for finite MDPs with visiting-all-state pairs and diminishing α; off-policy learning enables the agent to learn an optimal greedy policy while following any given behaviour policy.
Cons – Requires enumerating $|S| \times |A|$; impossible for large or continuous spaces; learning can be unstable if α or ε schedules are not tuned; still needs extensive exploration to converge.

❖ **Typical Applications**

- Cliff-Walking and Frozen-Lake toy benchmarks

- Simple robot-grid navigation prototypes
- Teaching RL basics before introducing function approximation



14. Episode returns averaged over 100 runs for CliffWalking-v0: SARSA (blue) converges faster and to higher returns than off-policy Q-Learning (red).

SARSA (State–Action–Reward–State–Action)

❖ **Problem Type**

On-policy control for finite, discrete Markov-decision processes.
Learns the value of the policy it is currently executing (typically ε-greedy), rather than learning about a separate greedy target policy as Q-Learning does.

❖ **Core Idea**

Update the action-value table $Q(s, a)$ toward the return of the exact action actually taken in the next state. Because the target contains a_{t+1} generated by the behaviour policy, SARSA is inherently on-policy and tends to learn safer policies in environments where exploratory actions are risky (e.g., Cliff-Walking).

❖ **Hypothesis (Model Form)**

$$Q\colon\ S \times A \ \longrightarrow\ \mathbb{R},$$

a lookup table whose entry $Q(s, a)$ estimates the expected return when following the current (ε-greedy) policy.

❖ **Cost / Loss Function**

Temporal-difference error for on-policy target

$$\delta = r_{t+1} + \gamma\, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

often viewed as minimising the per-visit squared loss $\frac{1}{2}\delta^2$.

❖ **Optimiser**

One-step **SARSA update**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta,$$

applied online after every transition.

Sequence remembered: **S**tate, **A**ction, **R**eward, **S**tate, **A**ction → "SARSA".

❖ Hyper-parameters (ε-greedy agent)

Name	Typical range	Effect
α (learning rate)	0.1 – 0.7	Higher α → faster but noisier updates
γ (discount)	0.90 – 0.99	Short- vs long-term reward emphasis
ε (exploration)	Start ≈ 0.9 → decay to 0.05	Drives on-policy exploration
Episode length / max steps	env-specific	Truncation can bias learning
Initial Q	0 or optimistic	Optimistic starts speed discovery

❖ Pros and Cons

Pros

- On-policy: learns behaviour that naturally incorporates the exploration strategy; safer in domains where exploratory actions can be catastrophic.
- Same computational cost as Q-Learning; converges under the same conditions (decaying α, infinite visits).

Cons

- Often converges more slowly than Q-Learning because it backs up *expected* exploratory returns, not the greedy max.
- Still infeasible for large or continuous state spaces without function approximation.
- Performance strongly coupled to the ε schedule; too much exploration hurts asymptotic return.

❖ Typical Applications

• Cliff-Walking variant where stepping off the cliff yields heavy negative reward (SARSA learns a "safer" path)
• Educational demos contrasting on- vs off-policy methods
• Small autonomous-robot mazes where exploration risk must be internalised

Deep Q-Network (DQN)

❖ Problem Type

Model-free, off-policy control for large or continuous image-like state spaces where a tabular Q-table is impossible (e.g., Atari, 3-D simulators, robot camera feeds).

❖ Core Idea

Replace the table $Q(s, a)$ with a **deep neural network** $Q_\theta(s, a)$ that approximates action values.
Two stabilising tricks make learning workable:

1. **Experience replay** – store transitions (s, a, r, s') in a buffer; sample mini-batches uniformly to break temporal correlation.
2. **Target network** – keep a delayed copy θ^-; compute TD targets with it and update it every C steps.

❖ Hypothesis (Model Form)

$$Q_\theta\colon S \times A \longrightarrow \mathbb{R}, \quad (s, a) \mapsto Q_\theta(s, a),$$

where Q_θ is a convolutional (or MLP) network with shared state backbone and one linear head per discrete action.

❖ Cost / Loss Function

Huber (or MSE) TD-error loss on a replay mini-batch of size B:

$$\mathcal{L}(\theta) = \frac{1}{B}\sum_{i=1}^{B}\left(r_i + \gamma \max_{a'} Q_{\theta^-}(s_i', a') - Q_\theta(s_i, a_i)\right)^2.$$

❖ Optimiser

1. **Sample** a mini-batch from the replay buffer.
2. **Compute** TD targets with the frozen target net Q_{θ^-}.
3. **Back-propagate** $\nabla_\theta \mathcal{L}$ and update θ via Adam (learning-rate ~ 10^{-4} – 10^{-3}).
4. **Every *C* steps** copy weights: $\theta^- \leftarrow \theta$.
5. **Exploration** via ε-greedy schedule (ε decays from 1.0 to 0.05).

❖ Hyper-parameters

Parameter	Typical values	Effect
learning_rate	1e-4 – 1e-3	Adam optimiser step size
batch_size	32 – 256	Larger ⇒ smoother gradient

replay_buffer_size	10^5 – 10^6	Memory vs experience diversity
gamma	0.99	Discount factor
epsilon_start / end / decay	1.0 → 0.05 over 1e6 steps	ε-greedy exploration schedule
target_update_freq C	1 000 – 10 000 steps	Stabilises learning
network_arch	Conv layers + 2 FC	Depth affects capacity

❖ Pros / Cons

Pros

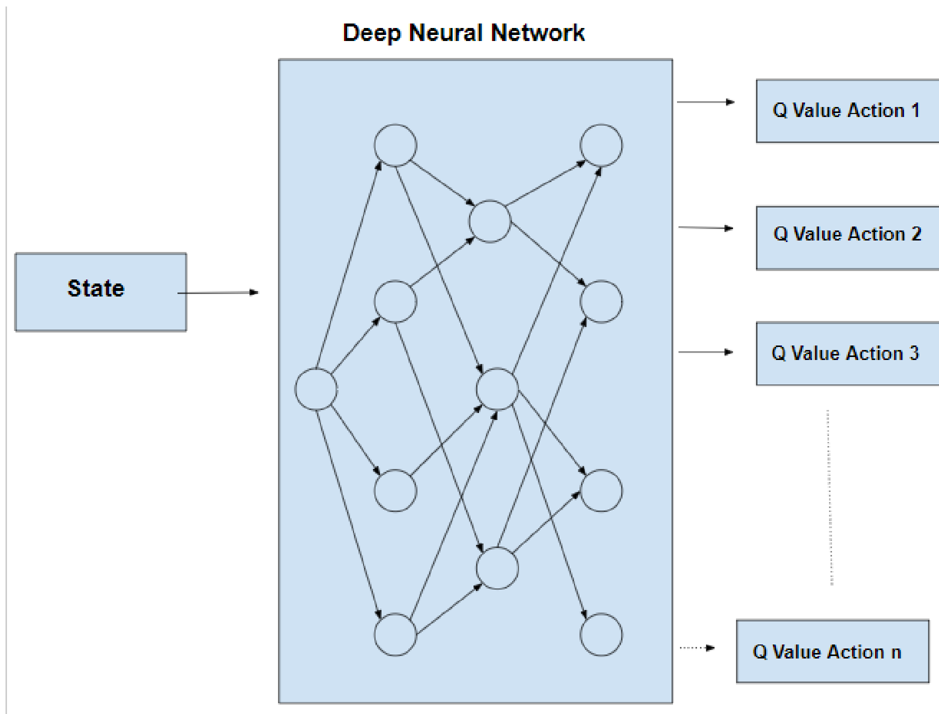
- Handles raw pixel input—no manual feature engineering.
- Replay buffer improves sample efficiency; target net stabilises bootstrapping.
- Extensible: Double DQN, Dueling DQN, Rainbow reach near-human Atari scores.

Cons

- Millions of environment steps + GPU time → costly.
- Hyper-parameter sensitive; training can diverge without careful tuning.
- Still limited to discrete actions; continuous control needs DDPG/SAC.

❖ Typical Applications

• Atari 2600 benchmark (Breakout, Pong)
• Grid-worlds with high-dim observations
• Early prototypes for robotic manipulation from images



15. Schematic of a Deep Q-Network (DQN)

Model-Evaluation Metrics and Decision Guidelines

Classification Metrics

Metric	Formula	Range	When to prefer

Accuracy	$$\frac{TP + TN}{TP + FP + FN + TN}$$	0 – 1	Classes are **balanced** and all errors cost the same
Precision	$$\frac{TP}{TP + FP}$$	0 – 1	**False-positives are expensive** (e-mail spam, medical screening)
Recall	$$\frac{TP}{TP + FN}$$	0 – 1	**False-negatives are expensive** (disease detection, fraud)
F-1 score	$$\frac{2\,(\text{Precision})(\text{Recall})}{\text{Precision} + \text{Recall}}$$	0 – 1	Imbalanced classes; need single number balancing Prec & Rec
ROC–AUC	$$\int_0^1 \text{TPR}(FPR)\ d(\text{FPR})$$	0.5 – 1	Ranking quality over all thresholds; works even if threshold is unknown
PR–AUC	$$\int_0^1 \text{Precision}(Recall)\ d(\text{Recall})$$	Baseline = pos-rate → 1	**Highly imbalanced** data where positives ≪ negatives
Matthews Corr. Coeff. (MCC)	$$\frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$	–1 … 1	Single, balanced metric even when class sizes differ greatly

TP = True Positives, FP = False Positives, FN = False Negatives, TN = True Negatives

Regression Metrics

Metric	**Formula**	**Range**	**When to prefer**
Mean-Squared Error (MSE)	$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$	$[0, \infty)$	Penalise large errors *quadratically* (smooth targets, no major outliers)

Root-Mean-Squared Error (RMSE)	$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}$	[0, ∞)	Same units as target y; easy to interpret vs MSE
Mean-Absolute Error (MAE)	$\frac{1}{n}\sum_{i=1}^{n}\lvert y_i-\hat{y}_i\rvert$	[0, ∞)	Robust to outliers; L^1 loss (median-like)
Coefficient of Determination (R^2)	$1-\frac{\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}{\sum_{i=1}^{n}(y_i-\bar{y})^2}$	(−∞, 1]	Compare to *null model*; good for linear models, balanced variance
Mean-Absolute Percentage Error (MAPE)	$\frac{100}{n}\sum_{i=1}^{n}\left\lvert\frac{y_i-\hat{y}_i}{y_i}\right\rvert$	[0, ∞)%	Business KPIs where *percentage* error matters; targets non-zero & positive
Huber Loss (δ-clip)	$\frac{1}{n}\sum_{i=1}^{n}\begin{cases}\frac{1}{2}(y_i-\hat{y}_i)^2, & \lvert y_i-\hat{y}_i\rvert\le\delta\\ \delta\left(\lvert y_i-\hat{y}_i\rvert-\frac{\delta}{2}\right), & \lvert y_i-\hat{y}_i\rvert>\delta\end{cases}$	[0, ∞)	Mix of MSE (small residuals) & MAE (large); choose δ≈1 for moderate outliers

Notes:

- n = number of observations; $\bar{y}$ = mean of true targets.
- RMSE and MAE share the same units as y; MAPE is unit-free (%).
- Huber with $\delta \approx 1.0$ often outperforms plain MSE when moderate outliers exist.

Decision Matrix – Choosing the Right Metric

Problem setting	Accuracy	Precision	Recall	F-1	ROC-AUC	PR-AUC	MCC	MSE	MAE	RMSE	Silhouette	D-B	Expl-Var
Balanced binary clf	✓	✓	✓	✓	✓	⚠	✓	—	—	—	—	—	—
Imbalanced binary clf	✕	✓	✓	✓	⚠	✓	✓	—	—	—	—	—	—
Multi-class balanced	✓	⚠	⚠	✓	✓	✕	✓	—	—	—	—	—	—
Regression (smooth, few outliers)	—	—	—	—	—	—	—	✓	✓	✓	—	—	—
Regression (heavy outliers)	—	—	—	—	—	—	—	⚠	✓	⚠	—	—	—
Clustering, no labels	—	—	—	—	—	—	—	—	—	—	✓	✓	—
Clustering, ground truth	—	—	—	—	—	—	—	—	—	—	⚠	⚠	—
Dim-red (PCA, t-SNE)	—	—	—	—	—	—	—	—	—	—	—	—	✓

✓ = **preferred** ⚠ = usable but often sub-optimal ✕ = misleading / not recommended — = not applicable

Choosing the Right Algorithm

Decision axis	Questions to ask	Algorithms that usually fit best
Problem type	• Is the target **continuous** or **categorical**? • Are labels absent (clustering) or rewards sequential (RL)?	Regression → Linear/GBT/NN Categorical → Logistic/CART/RF/GBT Unlabelled → k-Means/DBSCAN/PCA Sequential → Q-Learning, DQN
Data size & dimensionality	• How many rows nn? features dd? • Can the full data fit in memory?	$n \ll 10^5$: any algo $n \approx 10^6$: k-NN slow, prefer RF/HistGB/SGD High d (>500): use PCA, L1, or tree-ensembles; avoid k-Means with Euclidean distance
Linear vs nonlinear patterns	• Do exploratory plots show linear trends? • Does adding polynomial terms help?	Linear trend → Linear/Logistic/Ridge Non-linear → Trees, RF, GBT, DNN
Interpretability needs	• Must domain experts explain coefficients? • Is a white-box rule set required?	High interpretability → Linear, CART, Generalised Additive Models Medium → RF (feature importance, SHAP) Low → GBT, DNN
Training time / compute	• Real-time prototype or overnight batch? • GPU available?	Fast prototyping → Linear, CART, k-Means Large tabular → RF/HistGB (CPU) Images → CNN-based DNN (GPU)
Class imbalance / cost asymmetry	• Minority class <5 %? • Different costs for FP vs FN?	Algorithms with class_weight or intrinsic sampling (RF, GBT) Use metrics: PR-AUC, F1; avoid raw Accuracy
Outliers & noise	• Heavy-tailed errors? sensor glitches?	Robust → MAE, Huber, RF, GBT, k-NN (distance-weighted)
Deployment constraints	• Latency and RAM limits? • On-device or server?	Tiny RAM / low-latency → Logistic, shallow Trees Batch scoring OK → RF, GBT, DNN

❖ Decision workflow

1. **Frame the objective**
   Identify target type, success metric, constraints, and stakeholder requirements (explainability, latency).

2. **Baseline & sanity check**
   Always fit a trivial baseline (mean regressor, majority classifier).
   Follow with a **fast transparent model** (Linear or small CART) to gauge signal strength.

3. **Complexity ladder**
   Climb model families incrementally:
   Linear → regularised → trees → ensembles → DNN.
   Stop when added complexity yields diminishing metric gains relative to deployment cost.

4. **Cross-validate & stress-test**
   Use k-fold CV or OOB error; compare not only headline metric but stability, variance, and calibration.
   For imbalanced data, evaluate PR-AUC / F1 and inspect confusion matrix at relevant thresholds.

5. **Diagnose & iterate**
   • High bias? → add features, move up the complexity ladder.
   • High variance? → more data, stronger regularisation, ensembles.
   • Slow inference? → prune trees, distil DNN, quantise weights.

If you need…	…and data are	Start with
Fast, interpretable baseline	Any	Linear/Logistic + regularisation
Strong tabular accuracy	≤ 1 M rows, ≤ 1 k features	Gradient-Boosted Trees (XGBoost/LightGBM/HistGB)
Robust model with minimal tuning	Heterogeneous features, some outliers	Random Forest
Sparse high-dim text	Tens of k features	Linear SVM / Logistic with SGD
Image or raw sensor pixels	> 10 k samples, GPU available	Convolutional Neural Net
Non-convex clusters	2-D / 3-D data	DBSCAN
Sequential decision making	Simulated environment	Q-Learning (small) → DQN (large)

Conclusion

This study has systematically examined the theoretical underpinnings of machine learning, emphasizing not only its major branches—supervised, unsupervised, and reinforcement learning—but also the critical concepts and workflows that make up a robust ML pipeline. From data preparation and model selection to cross-validation and performance metrics, each section has contributed to a holistic understanding of how intelligent systems are built and evaluated.

The exploration of various algorithm families, their strengths and limitations, and their appropriate use cases demonstrates that machine learning is not a one-size-fits-all discipline. Instead, it is a field grounded in trade-offs—between bias and variance, complexity and interpretability, accuracy and generalizability.

By consolidating foundational knowledge with diagnostic tools, evaluation techniques, and best practices, this document offers a comprehensive reference for making informed, context-aware decisions in the development of machine learning solutions. It underscores the importance of rigor, clarity, and adaptability in approaching real-world data challenges through the lens of algorithmic learning.