Workflow of a Typical Machine Learning Problem

October 15, 2025

Machine Learning Workflow Overview

1. Import Libraries
2. Import Dataset
3. Exploratory Data Analysis (EDA)
4. Data Scrubbing / Preprocessing
5. Pre-Model Algorithms (Feature Engineering)
6. Split & Cross Validation
7. Set Algorithm (Model Selection)
8. Predict
9. Evaluate
10. Optimize

Step 1 – Import Libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, cross_val_scor
from sklearn.metrics import accuracy_score, classification_report
```

Step 2 – Import Dataset (Built-ins)

Dataset Name	Code	Suggested Use Case
Boston House Prices	`load_boston`	Regression (deprecated in recent sklearn)
Iris	`load_iris`	Classification
Diabetes	`load_diabetes`	Regression
Digits	`load_digits`	Classification
Linnerud	`load_linnerud`	Multivariate regression
Wine	`load_wine`	Classification
Breast Cancer	`load_breast_cancer`	Classification

```
from sklearn.datasets import load_breast_cancer
data = load_breast_cancer()
X, y = data.data, data.target
```

Step 3 – Exploratory Data Analysis (EDA)

- Inspect shape, dtypes, missing values, outliers, correlations.
- Visualize distributions and pairwise relationships.

```
df.info(); df.describe()
sns.pairplot(df, diag_kind="hist")
sns.heatmap(df.corr(numeric_only=True), annot=True, fmt=".2f")
```

Step 4 – Data Scrubbing / Preprocessing

- Handle missing values, duplicates, and wrong dtypes.
- Encode categoricals; normalize/standardize features.

```
df = df.drop_duplicates()
df = df.fillna(df.mean(numeric_only=True))
df = pd.get_dummies(df, drop_first=True)
```

Step 5 – Pre-Model Algorithms (Feature Engineering)

- Feature selection (filter, wrapper, embedded); PCA for dimensionality reduction.
- Create meaningful derived variables to reduce variance/overfitting.

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2).fit(X_train)
X_train_pca = pca.transform(X_train)
X_test_pca  = pca.transform(X_test)
```

Step 6A – Train/Test Split: Split validation

Definition:

- Divides data into two subsets — a training set to learn from and a testing set to evaluate model performance.
- Common ratios: 70:30 or 80:20.

Python Example:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, shuffle=True
)
```

Purpose:

- Provides a quick estimate of model performance.
- May vary depending on how data is split.

Step 6B – Cross Validation

Definition:

- Data is divided into k folds.
- Model is trained on $k-1$ folds and validated on the remaining one.
- Repeated k times for more stable results.

Equation:

$$\text{CV Score} = \frac{1}{k} \sum_{i=1}^{k} \text{Metric}_i$$

Python Example:

```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(random_state=42)
scores = cross_val_score(model, X, y, cv=5, scoring="accuracy")
print(scores.mean(), scores.std())
```

Summary:

- Split validation: single performance estimate.
- Cross-validation: averaged estimate → reduces bias & variance, good for small datasets

Step 7 – Set Algorithm (Model Selection)

Algorithm	Target	Data Type	Method	Resources	Transparency
Linear Regression	Continuous	Linear, clean	Supervised	Low	High
Logistic Regression	Discrete	Reliable patterns	Supervised	Low	Medium
k-Means	Discrete	Complex	Unsupervised	Medium	High
Decision Trees	Both	Few outliers	Supervised	Medium	High
Gradient Boosting	Both	Limited outliers	Ensemble	High	Low
Random Forests	Both	Messy, complex	Ensemble	Medium	Low
SVM	Both	Complex, high volume	Supervised	High	Low
MLP	Both	Complex, high volume	Supervised	High	Low

Table: Overview of popular ML algorithms.

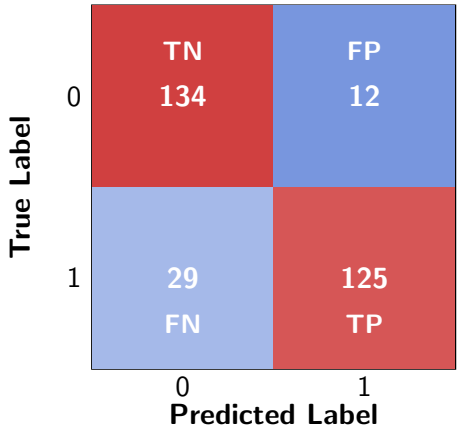
Learning Categories:

- **Supervised Learning:** Learns from labeled input–output data to predict future outcomes.
- **Unsupervised Learning:** Finds structure or clusters in unlabeled data.
- **Ensemble Learning:** Combines multiple weak models to improve accuracy and stability.

Step 8 – Predict

```
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

Step 9 – Evaluate (Classification Metrics)



Confusion Matrix (Illustrative Example)

- **True Positive (TP):** Correctly predicted positive cases (bottom-right).
- **True Negative (TN):** Correctly predicted negative cases (top-left).
- **False Positive (FP):** Incorrectly labeled as positive (top-right).
- **False Negative (FN):** Missed positive cases (bottom-left).

Step 9 – Evaluate (Classification Metrics)

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

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

$$\text{Recall (Sensitivity)} = \frac{TP}{TP + FN}$$

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

Support: number of true instances for each class.

Sklearn Snippet

```
from sklearn.metrics import confusion_matrix, classification_report
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

Step 9 – Evaluate (Regression Metrics)

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}$$

Notes: MAE is more robust to outliers; RMSE penalizes larger errors more heavily.

Step 10 – Optimize: Hyperparameters

- **Hyperparameters** are external configuration values set *before* the learning process begins.
- They control the model's learning behavior, capacity, and generalization (unlike internal *parameters*, which are learned from data).

Algorithm	Common Hyperparameters	Purpose
PCA	n_components	Number of principal components to retain
K-Means	n_clusters, init, max_iter	Define cluster count, initialization, and iterations
SVM	C, kernel, gamma	Control regularization and decision boundary complexity
Random Forest	n_estimators, max_depth, min_samples_split	Trees count, depth, and split control
Gradient Boosting	learning_rate, n_estimators, subsample	Step size, number of boosting rounds, sampling ratio
Neural Network	hidden_layers, learning_rate, batch_size	Network size, training speed, stability
KNN	n_neighbors, metric, weights	Number of neighbors and distance calculation rule
Logistic Regression	penalty, C, solver	Regularization strength and optimization method

Goal: Find the best hyperparameter combination (e.g., via *Grid Search*, *Random Search*, or *Bayesian Optimization*) to improve model performance.

Step 10 – Optimize (Hyperparameters)

```
from sklearn.model_selection import GridSearchCV

param_grid = {"n_estimators": [100, 200],
              "max_depth": [4, 6, 8]}

grid = GridSearchCV(RandomForestClassifier(),
                    param_grid, cv=5,
                    scoring="accuracy", n_jobs=-1)

grid.fit(X_train, y_train)
print(grid.best_params_, grid.best_score_)
```

Summary

1. Prepare data: EDA & scrubbing.
2. Engineer features; select an algorithm.
3. Train with proper validation (K-fold).
4. Evaluate with the right metrics (classification/regression).
5. Tune hyperparameters and iterate.

Linear regression

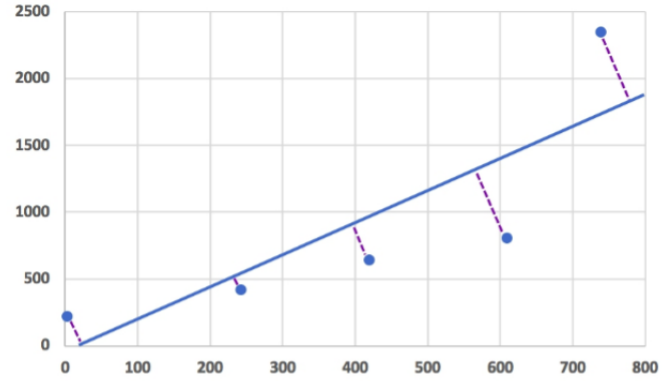
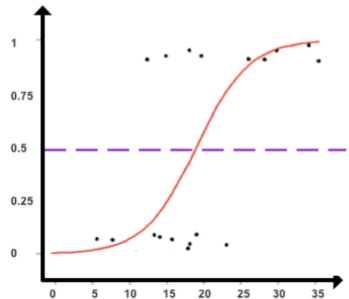


Figure 24: The distance of the data points to the hyperplane

$$y = \beta_0 + \sum_{i=1}^{p} \beta_j x_i.$$

Logistic regression: Probability (sigmoid) form



$$p \equiv P(y = 1 \mid \mathbf{x}_i) = \sigma\left(\beta_0 + \sum_{i=1}^{p} \beta_j x_i\right) = \frac{1}{1 + \exp\left(-\beta_0 - \sum_{i=1}^{p} \beta_j x_i\right)}.$$

Logistic regression: Hyperplane

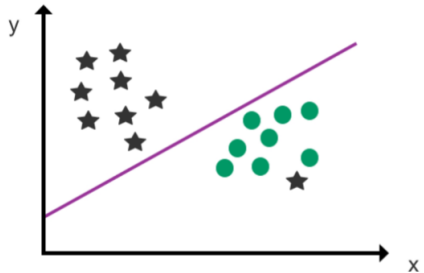


Figure 26: Logistic regression hyperplane is used to split the two classes

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \sum_{i=1}^{p} \beta_j x_i,$$

Learning linear functions of features

Given data $(\mathbf{x}_i, y_i)$ with $\mathbf{x}_i \in \mathbb{R}^p$:

- Postulate a **linear score** $z(\mathbf{x}) = w_0 + \mathbf{w}^\top \mathbf{x}$.
- **Linear regression**: predict a *continuous* target with $\hat{y} = z(\mathbf{x})$.
- **Logistic regression**: predict a *binary* target with $\hat{p} = \sigma\big(z(\mathbf{x})\big)$, $\sigma(t) = \frac{1}{1+e^{-t}}$.
- **Hyperplane view**: $\{\mathbf{x} \,|\, \mathbf{w}^\top \mathbf{x} + w_0 = 0\}$ is a $(p-1)$-dimensional hyperplane.

When to use which?

Linear Regression

- Continuous targets; approximate linear relation.
- Goal: predictive mean; interpretable effect sizes.
- Metrics: MAE/RMSE/R^2; inspect residuals.

Logistic Regression

- Binary (or multinomial) targets; probabilistic outputs.
- Goal: calibrated probabilities; linear decision boundary $\mathbf{w}^\top \mathbf{x} + w_0 = 0$.
- Metrics: Precision/Recall/F1, ROC–AUC, calibration.

Takeaway: both learn a *linear function in feature space*; linear predicts a value; logistic maps the same linear score through a sigmoid to yield class probabilities and a hyperplane decision boundary.

Linear Classifiers

- Both **Logistic Regression (LR)** and **Support Vector Machine (SVM)** are supervised learning algorithms used for classification.
- They find a separating **hyperplane** between two classes.
- However, their objective functions differ fundamentally.

Logistic Regression: Probabilistic Model

- Logistic regression models the **probability** of class membership using the sigmoid function:

$$P(y = 1|x) = \frac{1}{1 + e^{-w^T x - b}}$$

- It minimizes the **logistic loss (cross-entropy)**:

$$L = \sum_i \log(1 + e^{-y_i(w^T x_i + b)})$$

- Decision boundary: where $P = 0.5$, i.e. $w^T x + b = 0$.

Support Vector Machine: Geometric Model

- SVM seeks the **maximum-margin** hyperplane separating the two classes.
- Objective:

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{s.t. } y_i(w^T x_i + b) \geq 1$$

- Only the **support vectors** (closest points) influence the boundary.
- The margin width is $\frac{2}{\|w\|}$.

Comparison of Decision Boundaries

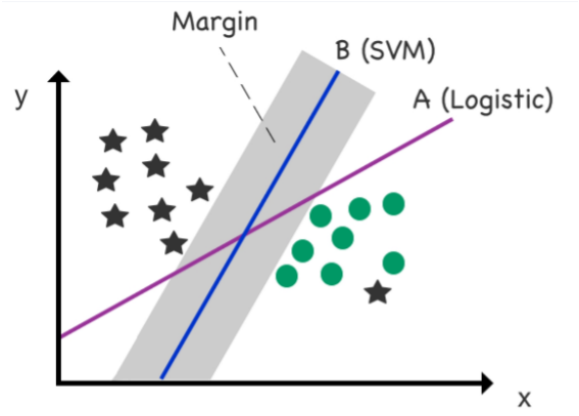
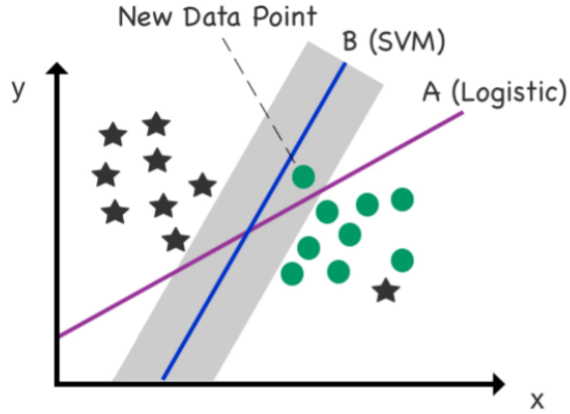


Figure 27: Logistic regression versus SVM

Figure: Logistic regression (A) vs SVM (B) – SVM maximizes the margin, logistic regression fits probability boundary.

Effect of New Data Point



igure 28: A new data point is added to the scatterplot

Observation: Logistic regression boundary (A) shifts more due to new data; SVM (B) is less sensitive as it depends mainly on support vectors.

Key Differences

Aspect	**Logistic Regression**	**SVM**
Nature	Probabilistic (predicts $P(y\|x)$)	Geometric (margin-based)
Loss Function	Logistic / Cross-entropy	Hinge loss: $\max(0, 1 - y_i(w^T x_i + b))$
Output	Probabilities	Class labels only
Robustness	Sensitive to outliers	More robust (due to margin)
Computation	Faster, simpler	Slower for large datasets

Summary

- Logistic Regression: interpretable, probabilistic, sensitive to outliers.
- SVM: focuses on maximizing margin, effective for complex/nonlinear boundaries.
- Both can use kernel transformations for nonlinear data.

Hyperparameters & Grid Search (SVM)

- **C (regularization):** trade-off between *wide margin* (small C) and *fewer violations* (large C). Too big $C \Rightarrow$ overfit; too small $\Rightarrow$ underfit.
- γ **(RBF width):** influence range of a single point. Small $\gamma \Rightarrow$ smoother boundary; large $\gamma \Rightarrow$ wiggly boundary.
- **Kernel:** linear (fast, interpretable), RBF (default, flexible), poly/sigmoid (situational). Always **scale features** before non-linear kernels.
- **Cross-Validation (CV):** reliable model selection on the *training* set; stabilizes choices but may not change test accuracy.

GridSearchCV (best practice)

```
Pipeline([('scaler', StandardScaler()), ('svm', SVC())])
param_grid = { 'svm__kernel':  ['linear','rbf'], 'svm__C': [0.1,1,10,100], 'svm__gamma': ['scale',1e-3,1e-4] }
GridSearchCV(pipe, param_grid, cv=5, scoring='accuracy', n_jobs=-1)
```

Reading results: inspect `best_params_`, `best_score_`, and `cv_results_` (mean $\pm$ std across folds).

Overview

- **K-Nearest Neighbors (KNN):** A non-parametric, instance-based supervised learning algorithm.
- **Tree-Based Methods:** Include Decision Trees, Random Forests, Bagging, and Boosting.
- All are **supervised learning algorithms**, capable of handling both classification and regression tasks.
- Focus: interpretability, flexibility, and adaptability to nonlinear data.

K-Nearest Neighbors (KNN): Concept

- Classifies a new point based on the **majority label of its k-nearest neighbors**.
- Distance metrics: Euclidean, Manhattan, or Minkowski.
- **Small k:** Overfitting, sensitive to noise.
  Large k: Oversmoothing, underfitting.
- Lazy learner — no model training, computes during prediction.
- Works only with **numerical or distance-computable features**.

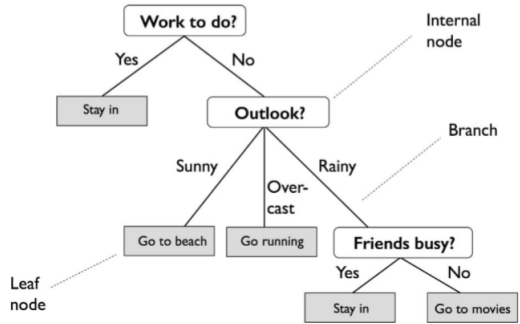
Figure 29: An example of k-NN clustering used to predict the class of a new data poi

Tree-Based Methods: Overview

- **Decision Trees, Random Forests, Bagging, Gradient Boosting**
- Used for both **classification** and **regression**.
- Handle **categorical and numerical data**.
- Advantages:
  - Highly interpretable (tree-graph structure).
  - Nonlinear decision boundaries.
  - Can work without feature scaling.
- Disadvantages:
  - High variance (overfitting) if tree depth is not controlled.
  - Sensitive to small perturbations in data.

Decision Tree: Concept

- Builds a model in a tree-like structure of decisions and outcomes.
- Splits data recursively using variables that best reduce **Gini impurity** or **entropy**.
- Internal nodes: decision based on a feature.
  Branches: outcome of test.
  Leaves: predicted label/value.
- Works well for small-to-medium datasets.



Decision Tree for what to do today. https://towardsdatascience.com

Tree Ensembles: Bagging, Random Forest, Boosting

- **Bagging:** Builds multiple trees on bootstrap samples and averages predictions (reduces variance).
- **Random Forest:** Bagging + random feature selection at each split (decorrelates trees).
- **Boosting:** Sequentially builds trees, each correcting errors of the previous (reduces bias).
- **Gradient Boosting:** Uses gradient descent to minimize loss function across weak learners.

Supervised Learning Context:

- All tree-based models are supervised.
- Input: labeled data (X, y).
  Output: predictive model.
- For unlabeled data, clustering or dimensionality reduction (unsupervised) methods are used instead.

Comparison: KNN vs Tree-Based Methods

Aspect	**KNN**	**Tree-Based Methods**
Learning type	Supervised (instance-based)	Supervised (model-based)
Training time	None (lazy learner)	Requires training (tree construction)
Prediction time	Expensive (distance calc.)	Fast after training
Interpretability	Low	High (visual trees)
Feature scaling	Required	Not required
Data type	Numeric only	Numeric + Categorical
Overfitting control	via k	via pruning / ensembles

Summary

- KNN: intuitive, simple, but computationally heavy on large datasets.
- Decision Trees: interpretable but unstable.
- Random Forest / Bagging: reduce variance.
- Boosting / Gradient Boosting: reduce bias.
- Ensemble learning combines multiple weak learners to achieve robust models.

- **Decision Tree:** one interpretable single-tree model, high variance.
- **Random Forest:** multiple trees trained in parallel; reduces variance.
- **Gradient Boosting:** sequential small trees ((sequential correction); reduces bias by correcting errors.

Decision Tree/ Random Forest Classifier Parameters

Parameter	Description	Typical Impact
`max_depth=10`	Limits how deep the tree can grow (how many levels of decisions).	Prevents **overfitting**. Larger depth → more complex model; smaller depth → simpler model.
`criterion='gini'`	Metric to measure the quality of a split. Options: `'gini'` or `'entropy'`.	`'gini'` uses **Gini impurity** (faster), while `'entropy'` uses **information gain** (Shannon entropy).
`random_state=42`	Sets the seed for random processes (like choosing feature splits).	Ensures **reproducibility** – same random seed ⇒ same model each time you run.
`n_jobs= -1` (only for random forest)	Number of CPU cores to use for parallel processing.	-1 uses all cores ⇒ faster training; use smaller values to limit CPU usage

Gradient Boosting Parameters

Parameter	Description	Typical Impact / Guidance
`n_estimators=250`	Number of boosting stages (trees).	More estimators → lower bias but higher training time and possible overfitting. Common range: 100–500.
`learning_rate=0.1`	Shrinkage rate applied to each tree's contribution.	Lower rate requires more estimators; balances model complexity vs. accuracy. Typical: 0.01–0.2.
`max_depth=10`	Maximum depth of individual regression trees.	Deeper trees learn more complex relations but risk overfitting; shallower trees increase bias.
`min_samples_split=4`	Minimum number of samples required to split an internal node.	Higher values → simpler, more generalized trees.
`min_samples_leaf=6`	Minimum number of samples at a leaf node.	Prevents very small leaves; helps regularization and reduces overfitting.
`max_features=0.6`	Fraction of features considered for each split.	Adds randomness, improves generalization, speeds up training. Typical: 0.3–1.0.
`loss='log_loss'`	Loss function optimized during boosting.	`'log_loss'` = logistic regression loss for classification; lower loss = better fit. Also, `'exponential'`

Loss functions in GradientBoostingRegressor:

`'squared_error'` `'absolute_error'` `'huber'` **— a mix between squared and absolute loss (less sensitive to outliers),** `'quantile'`

BaggingClassifier Parameters

Parameter	Default Value	Description
estimator	None	Base estimator to fit on random subsets. Defaults to a decision tree (DecisionTreeClassifier()) if None.
n_estimators	10	Number of base estimators (bagged models) in the ensemble.
max_samples	1.0	Fraction or number of samples to draw from the training set to train each base estimator. (1.0 = 100% of the data, with replacement).
max_features	1.0	Fraction or number of features to draw for training each base estimator. (1.0 = all features).
bootstrap	True	Whether samples are drawn with replacement (bootstrap sampling).
bootstrap_features	False	Whether features are drawn with replacement.
oob_score	False	If True, compute the out-of-bag score using data not seen by each base estimator.
warm_start	False	If True, reuse the solution of previous calls to .fit() and add more estimators incrementally.
n_jobs	None	Number of CPU cores to use for parallel training. None = 1 core; -1 = all cores.
random_state	None	Seed for reproducibility.
verbose	0	Controls the verbosity level during fitting.

Comparison of Tree-based Ensemble Methods

Model	Core Idea	Training Strategy	Bias–Variance Tradeoff	Typical Features / Remarks
Decision Tree	Single tree partitions data using impurity measures (Gini / entropy).	Trained once on all data; greedy top-down splitting.	Low bias, high variance.	Easy to interpret; prone to overfitting.
Bagging	Train multiple models on bootstrapped samples and average results.	Parallel, independent models (each on random sample).	Reduces variance; bias unchanged.	Stabilizes high-variance models like trees. Example: Random Forest.
Random Forest	Bagging + random feature selection at each split.	Parallel ensemble of many decision trees.	Lower variance than single tree.	Adds feature randomness; robust, less interpretable.
Boosting	Sequentially add weak learners focusing on previous errors.	Each model trained on weighted data (misclassified points emphasized).	Reduces bias; may increase variance.	AdaBoost is classic version; uses exponential loss.
Gradient Boosting	Boosting via gradient descent on a loss function.	Sequential models fit to residuals (negative gradients).	Reduces both bias and error progressively.	More flexible; supports various losses (`log_loss`, `squared_error`); forms basis of XGBoost, LightGBM.

Evolution of Methods (Simple Flow)

Decision Tree $\Rightarrow$ **Bagging** $\Rightarrow$ **Random Forest** $\Rightarrow$ **Boosting**
$\Rightarrow$ **Gradient Boosting**

- **Decision Tree:** single model; high variance.
- **Bagging:** variance reduction via bootstrap + averaging.
- **Random Forest:** bagging + random features (decorrelates trees).
- **Boosting:** sequential weak learners correct errors (bias $\downarrow$).
- **Gradient Boosting:** boosting via gradient descent on a loss.

Base Estimator → Ensemble Classifier (Simple Flow)

Base Estimator
(e.g., `DecisionTreeClassifier(max_depth=1)`)

$\Downarrow \quad \Downarrow \quad \Downarrow \quad \cdots$ (*many copies / rounds*)

Combine Predictions
(vote / average / weighted sum)

$\Downarrow$

Ensemble Classifier
(e.g., `AdaBoostClassifier`, `BaggingClassifier`, `RandomForestClassifier`)

Idea: The ensemble trains many simple models and *combines* them into one strong predictor.