**Decision Trees (ID3, CART)**

**Decision Tree Introduction:** A Decision Tree is a supervised machine learning algorithm used for both classification and regression tasks. It models decisions in a tree-like structure of nodes and branches:

* **Root Node:** The starting point of the tree (entire dataset).
* **Decision Nodes:** Nodes where data is split based on a feature.
* **Leaf Nodes:** Terminal nodes that provide the final prediction (class label or value).

**Working:** It splits the dataset into subsets based on the feature that results in the **best split** using metrics like **Entropy, Gini, or MSE**.

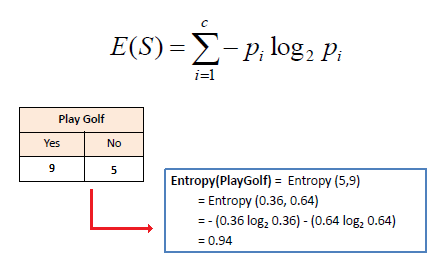
**Types of Decision Trees:**

* ID3
* CART

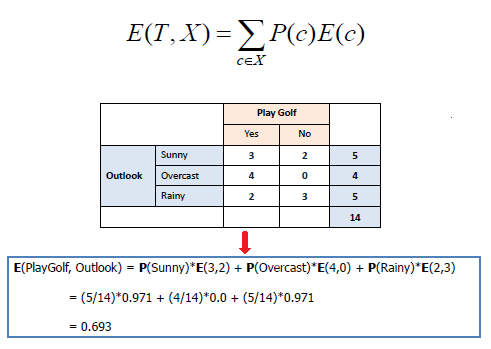
**ID3:** Works as follows:

* Calculate entropy for the dataset.
* For each feature, calculate information gain.
* Select the feature with the highest information gain to split.
* Repeat recursively on subsets until stopping criteria (all same class or no more features).

**Entropy Formula:**

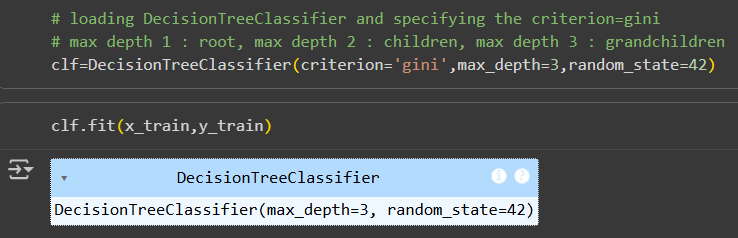


**Information Gain Formula:**

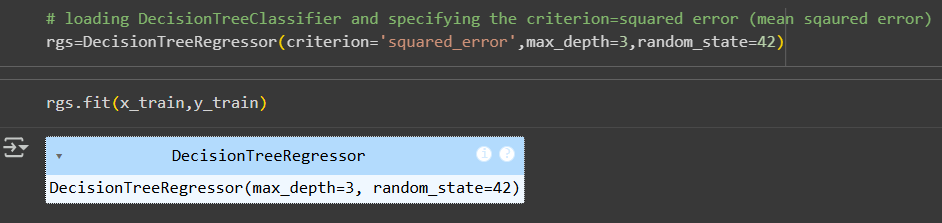


**CART:** Two types of CART:

* **Classification (CART):**
* Input: Features (numeric or categorical).
* Output: Class label (e.g., “spam” or “ham”, “disease” or “no disease”).
* Split criterion: Gini Index or Entropy.



* **Regression (CART):**
* Input: Features (numeric or categorical).
* Output: Continuous value (e.g., house price = 250,000).
* Split criterion: MSE (Mean Squared Error) or MAE (Mean Absolute Error).



**Ensemble Intro & Bagging**

**Ensemble Learning:** Combining multiple models (weak learners) to build a stronger, more accurate model. There are multiple ensemble learning methods/models:

* **Bagging**
* **Boosting**
* **Stacking**

**Bagging (Bootstrap Aggregating):** Bagging is one of the most common ensemble techniques. It works in the following way:

* **Bootstrapping**

From your dataset, create many **random samples with replacement**.

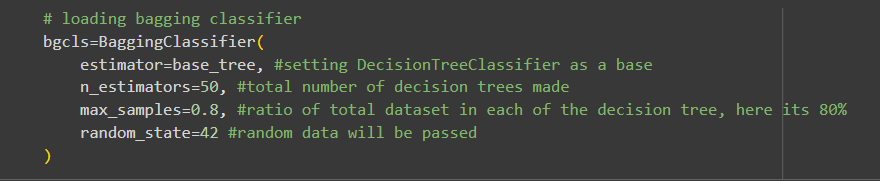
* Each sample is slightly different.
* **Train models**

Train one model (e.g., a decision tree) on each bootstrap sample.

* **Aggregate results**

For classification → take a **majority vote**.

For regression → take the **average prediction**.



**Random Forest**

**Random Forest:** Ensemble of Decision Trees

* It uses **bagging** (bootstrap aggregating)
* Plus, **random feature selection** at each split.

So instead of growing **one tree** (which might overfit), we grow **many trees** and combine them.

**Working:** Ensemble of Decision Trees

* Draw random samples of data (with replacement).
* Train a Decision Tree on each sample.

At each split, it only considers a random subset of features (not all).

* Aggregate results:

For classification → **majority vote**.

For regression → **average prediction**.

**Comparison with a Single Decision Tree:**

* A single tree has **high variance** (changes a lot with small changes in data).
* Random Forest reduces variance by averaging many trees.
* More stable, more accurate, less overfitting.

**Key Hyperparameters:**

* **n\_estimators:** number of trees (default 100, more = better but slower).
* **max\_features:** how many features to consider at each split.
* **max\_depth:** maximum depth of each tree.
* **min\_samples\_split, min\_samples\_leaf:** control overfitting.

**K-Nearest Neighbour (K-NN)**

**What is KNN:**

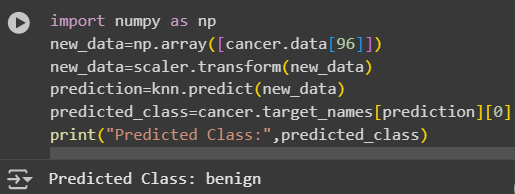
* A **supervised learning** algorithm.
* Can be used for both:
  + **Classification** → majority vote of neighbours.
  + **Regression** → average of neighbours.
* **Lazy learner**: no training phase, just stores data.
* **Non-parametric**: makes no assumptions about data distribution.

**Working:** Works in the following way:

1. Choose a value of k (number of neighbors).
2. For a new data point:

* Compute the distance to all training points.
* Pick the k nearest neighbors.

1. **Classification:** assign the label based on majority vote.  
   **Regression:** output = average of neighbors’ values.



**Distance Metrics:** Following distance metrics can be used to compute the distance to the neighbours:

* **Euclidean Distance:**



* It’s the straight-line distance ("as the crow flies").
* When movement is free in all directions and distances matter
* **Manhattan Distance:**



* It’s the distance if you can only move horizontally and vertically (like a taxi driving on a city grid).
* When movement is restricted to axes or when features are independent and not related by geometry

**Choosing the value of k:**

* **Small k** (e.g., 1, 3):
  + High variance, low bias (risk of overfitting, during training the accuracy will always be 100% as with k=1, the nearest neighbour to the datapoint is itself, but during testing, as the model just looks for 1 neighbour, to it might be an outlier or noise)
* **Large k** (e.g., 15, 20):
  + Low variance, high bias (risk of underfitting, as the model is looking for more neighbours to decide the class, to the it makes the model too simple and smooth).
* Usually, odd values are chosen for classification (to avoid ties).
* Best k is found using **cross-validation** (e.g., elbow method).

**Strengths of k-NN**

* Simple and intuitive.
* Works for classification & regression.
* Adapts to irregular decision boundaries.
* No training time (lazy learner).

**Limitations of k-NN**

* Slow at prediction (must compute distance to all points).
* Needs feature scaling (because distance is sensitive to scale).
* Sensitive to irrelevant/noisy features.
* Not great for very large datasets.

**Cross-validation to find best k**

* We don’t know the “best” k (number of neighbors).
* So, we try different k values (e.g., 1 → 30).
* For each k, we check performance using cross-validation.
* Then we choose the k that gives the best accuracy (or lowest error).



**Elbow method**

* We plot k (x-axis) vs accuracy/error (y-axis).
* Usually, accuracy rises quickly as k increases from 1, then levels off.
* The “elbow point” (where improvement slows down) is often the best k.

**Model Training & Tuning**

**What is Model Training?**

* Training = teaching your model to learn patterns from data.
* Process:
  + Split data → **train set** (to learn) and **test set** (to check performance).
  + Choose a model (e.g., Decision Tree, k-NN, Random Forest).
  + Feed training data → model adjusts itself to minimize errors.
  + Test on unseen data → see how well it generalizes.

**What is Model Tuning?**

* Tuning = adjusting **hyperparameters** (the settings you choose before training) to improve performance.
* Examples of hyperparameters:
  + Decision Tree → max\_depth, criterion
  + k-NN → n\_neighbors
  + Random Forest → n\_estimators
  + SVM → C, kernel
* Tuning is NOT about training weights (that’s the model’s job) — it’s about finding the best **settings**.

**Methods of Tuning**

* **Manual tuning** → try different values manually.
* **Grid Search** → automatically test all combinations of hyperparameters.
* **Random Search** → test random combinations (faster than grid search).
* **Cross-validation** → split training data into folds (like 5 parts), train on 4, validate on 1, rotate → more reliable accuracy.

**Coding Example:**

|  |
| --- |
| from sklearn.datasets import load\_breast\_cancer  from sklearn.model\_selection import train\_test\_split, GridSearchCV  from sklearn.preprocessing import StandardScaler  from sklearn.neighbors import KNeighborsClassifier  # Load data  X, y = load\_breast\_cancer(return\_X\_y=True)  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  # Scale features  scaler = StandardScaler()  X\_train = scaler.fit\_transform(X\_train)  X\_test = scaler.transform(X\_test)  # Define model  knn = KNeighborsClassifier()  # Define hyperparameter grid  param\_grid = {  'n\_neighbors': [3, 5, 7, 9, 11],  'metric': ['euclidean', 'manhattan']  }  # Grid search with cross-validation (how splits we make, here, 4 will be used for training and 1 will be used for validation, will continue for 5 rounds each time with a different validation split)  grid = GridSearchCV(knn, param\_grid, cv=5)  grid.fit(X\_train, y\_train)  print("Best Parameters:", grid.best\_params\_) # best combinition of params  print("Best Cross-Validation Score:", grid.best\_score\_) # best params score  print("Test Accuracy:", grid.score(X\_test, y\_test)) # Accuracy |

**Key Idea**

* Training = **fit the model**.
* Tuning = **choose the best hyperparameters** so the model is not underfitting or overfitting.