**1. Decision Tree Learning**

A **Decision Tree** is a method used in machine learning to make decisions based on a series of questions. It is like a flowchart where each question (called a **node**) splits the data into branches until a final decision (called a **leaf**) is made.

**How it works?**

* Start with a dataset containing features (inputs) and labels (outputs).
* The algorithm chooses the best feature (question) to split the data.
* Each split creates subgroups of data, and the process continues until all data is classified properly or a stopping condition is met.

**Example:**

Imagine you want to classify whether a person will buy a laptop based on **income** and **age**:

1. If the person earns more than $50,000 → likely to buy.
2. If the person earns less but is younger than 25 → unlikely to buy.
3. If the person earns less but is older than 25 → may buy.

The tree structure helps in making such decisions efficiently.

**2. Maximizing Information Gain – Getting the Most Bang for the Buck**

When building a decision tree, we want each split to be as useful as possible. **Information Gain** helps measure how well a particular feature splits the data.

**What is Information Gain?**

It tells us how much a particular question (feature split) improves the classification. The more **uncertainty (entropy)** a split removes, the higher the information gain.

**Example:**

Suppose we want to classify if a person likes tea or coffee based on temperature:

* Without splitting, we have a mixed group (uncertain).
* If we split the data into **Hot** and **Cold** categories, and we see all tea lovers prefer hot and all coffee lovers prefer cold, we have maximum information gain.

Thus, the best decision tree selects questions that **maximize information gain** at each step.

**3. Building a Decision Tree**

A decision tree is built step by step using **a greedy algorithm**, meaning it picks the best possible decision at each step.

**Steps to Build a Decision Tree:**

1. **Choose the Best Feature:**
   * Use **information gain** or other metrics like **Gini impurity**.
2. **Split the Data:**
   * Divide data into branches based on the chosen feature.
3. **Repeat the Process:**
   * Keep splitting until a stopping condition is met (e.g., no more useful splits, a max depth limit, or a small dataset left).
4. **Pruning (Optional):**
   * Remove unnecessary branches to avoid overfitting.

**4. Combining Weak to Strong Learners via Random Forests**

A **Random Forest** is a collection of multiple decision trees that work together to improve accuracy.

**Why Use Random Forest?**

* A single decision tree can **overfit** (memorize training data).
* By combining multiple decision trees (each trained on random parts of the data), the model generalizes better.

**How Does It Work?**

1. **Create Multiple Decision Trees:**
   * Each tree gets a random subset of data and features.
2. **Make Predictions with Each Tree:**
   * Each tree gives its own decision.
3. **Majority Voting:**
   * The final prediction is based on the majority vote of all trees.

Random Forests are **more accurate and robust** than a single decision tree.

**5. K-Nearest Neighbors (K-NN) – A Lazy Learning Algorithm**

K-NN is a **simple and powerful** classification algorithm that does not require any training phase.

**How K-NN Works?**

1. **Store all training data.**
2. **For a new input, find the K nearest (most similar) data points** in the training dataset.
3. **Majority voting:**
   * If most of the nearest neighbors belong to category A, classify the new input as A.
   * If most belong to category B, classify as B.

**Why is it called Lazy Learning?**

* K-NN does **not** create a model in advance.
* It only makes decisions when a new input is given.
* This makes it **slow** for large datasets but **very simple** to use.

**Example:**

You want to classify a fruit as either an **apple** or **orange** based on its size and color.

1. You compare the new fruit with existing apples and oranges.
2. If most of the 5 nearest fruits are apples → classify as **apple**.
3. If most are oranges → classify as **orange**.

**Final Summary**

* **Decision Tree Learning:** A flowchart-like method that makes decisions step by step.
* **Maximizing Information Gain:** Choosing the best splits to make the tree more effective.
* **Building a Decision Tree:** Selecting the best features and stopping when needed.
* **Random Forest:** A collection of decision trees that improve accuracy.
* **K-Nearest Neighbors (K-NN):** A simple, instance-based learning method that classifies based on the closest examples.

**1. Building Good Training Sets – Data Preprocessing**

Before using any machine learning model, we need to **clean and prepare** the data. This process is called **Data Preprocessing** and helps improve model accuracy.

**Why is Data Preprocessing Important?**

* Raw data can have **errors, missing values, duplicates, or irrelevant features**.
* Machine learning models work best when data is **clean, standardized, and formatted correctly**.

**Key Steps in Data Preprocessing:**

1. **Handling Missing Data**
2. **Scaling and Normalization** (adjusting values for consistency)
3. **Encoding Categorical Data** (converting text into numbers)
4. **Feature Selection** (choosing the most important variables)

**2. Dealing with Missing Data**

Missing data is a common problem in datasets. If we don’t handle it properly, it can lead to **wrong predictions** or **errors** in machine learning models.

**Ways to Handle Missing Data:**

* **Remove missing values** (eliminate samples or features with missing values).
* **Impute missing values** (fill in missing values using techniques).

**3. Eliminating Samples or Features with Missing Values**

If missing data is **too much** (e.g., more than 50% of the dataset), it’s better to **remove** that row or column.

**How to do it?**

In Python using pandas:

python

CopyEdit

import pandas as pd

df = pd.read\_csv("data.csv") # Load data

# Remove rows with missing values

df\_cleaned = df.dropna()

# Remove columns with missing values

df\_cleaned = df.dropna(axis=1)

This method is **only recommended** if the missing data is **not important** or there’s a lot of missing values.

**4. Imputing Missing Values**

If we can’t afford to lose data, we should **fill in the missing values** using different strategies.

**Common Techniques for Imputing Missing Data:**

1. **Mean Imputation:** Replace missing values with the **average** of the column.

python

CopyEdit

from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy='mean') # Use 'median' for median

df\_filled = imputer.fit\_transform(df)

1. **Most Frequent Value:** Replace missing values with the most **common** value in the column.

python

CopyEdit

imputer = SimpleImputer(strategy='most\_frequent')

df\_filled = imputer.fit\_transform(df)

1. **Forward Fill (Use previous value):**

python

CopyEdit

df.fillna(method='ffill', inplace=True)

1. **Backward Fill (Use next value):**

python

CopyEdit

df.fillna(method='bfill', inplace=True)

Choosing the right method depends on the **type of data** and how important missing values are.

**5. Understanding the Scikit-Learn Estimator API**

**Scikit-learn** is a popular machine learning library in Python. It provides a standard way to build models using the **Estimator API**.

**How the Estimator API Works?**

Every model in Scikit-learn follows three main steps:

1. **Fit the Model (Training)**
2. **Predict (Make Predictions)**
3. **Evaluate the Model**

**Example: Training a Simple Machine Learning Model**

python

CopyEdit

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.datasets import load\_iris

# Load dataset

iris = load\_iris()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(iris.data, iris.target, test\_size=0.2, random\_state=42)

# Step 1: Create model

model = LogisticRegression()

# Step 2: Train model

model.fit(X\_train, y\_train)

# Step 3: Make predictions

predictions = model.predict(X\_test)

# Step 4: Check accuracy

accuracy = model.score(X\_test, y\_test)

print(f"Model Accuracy: {accuracy:.2f}")

**Main Functions in Scikit-Learn API:**

* **fit(X, y)** → Train the model on data X (features) and y (labels).
* **predict(X)** → Predict outcomes for new data.
* **score(X, y)** → Evaluate model performance.

**Final Summary**

* **Data Preprocessing** helps clean and prepare data for better model performance.
* **Missing Data Handling** includes **removing** missing values or **imputing** them (mean, median, etc.).
* **Scikit-learn Estimator API** provides a structured way to train, predict, and evaluate machine learning models.

**1. Handling Categorical Data**

In machine learning, data can be of two types:

* **Numerical (numbers)**
* **Categorical (labels or categories like "Red", "Blue", "Male", "Female")**

Most machine learning models **cannot** work with categorical data directly. So, we need to convert it into numbers. There are different ways to do this, depending on the type of categorical data.

**Types of Categorical Data:**

1. **Ordinal Data** → Has a meaningful order (e.g., Small < Medium < Large).
2. **Nominal Data** → Has no specific order (e.g., Red, Blue, Green).

**2. Mapping Ordinal Features**

For **ordinal features** (where categories have a meaningful order), we can manually assign numbers.

**Example: Size of T-Shirts**

| **Size** | **Numeric Value** |
| --- | --- |
| Small | 1 |
| Medium | 2 |
| Large | 3 |

**How to Implement in Python?**

python

CopyEdit

import pandas as pd

data = {'Size': ['Small', 'Medium', 'Large', 'Medium', 'Small']}

df = pd.DataFrame(data)

# Define mapping

size\_mapping = {'Small': 1, 'Medium': 2, 'Large': 3}

# Apply mapping

df['Size'] = df['Size'].map(size\_mapping)

print(df)

This method preserves the order (Small < Medium < Large).

**3. Encoding Class Labels**

Class labels (i.e., target values) also need to be converted to numbers. For example, if you have labels like **["Yes", "No"]**, you need to encode them as **0s and 1s**.

**Example: Converting Labels (Yes/No) to 0 and 1**

python

CopyEdit

from sklearn.preprocessing import LabelEncoder

labels = ['Yes', 'No', 'Yes', 'No', 'Yes']

encoder = LabelEncoder()

# Encode labels

encoded\_labels = encoder.fit\_transform(labels)

print(encoded\_labels) # Output: [1 0 1 0 1]

* Here, **"Yes" → 1** and **"No" → 0**.
* The model can now understand the data.

**4. Performing One-Hot Encoding on Nominal Features**

For **nominal features** (where categories have no meaningful order), we use **one-hot encoding**.

**Why One-Hot Encoding?**

* If we map "Red", "Blue", "Green" as **1, 2, 3**, the model may think 3 > 2 > 1, which is incorrect.
* Instead, **One-Hot Encoding** creates **separate columns** for each category and assigns 1 or 0.

**Example: Converting Colors ("Red", "Blue", "Green")**

| **Color** | **Red** | **Blue** | **Green** |
| --- | --- | --- | --- |
| Red | 1 | 0 | 0 |
| Blue | 0 | 1 | 0 |
| Green | 0 | 0 | 1 |

**How to Implement in Python?**

python

CopyEdit

from sklearn.preprocessing import OneHotEncoder

import pandas as pd

data = {'Color': ['Red', 'Blue', 'Green', 'Blue', 'Red']}

df = pd.DataFrame(data)

# Apply One-Hot Encoding

encoder = OneHotEncoder()

encoded\_array = encoder.fit\_transform(df[['Color']]).toarray()

# Convert to DataFrame

df\_encoded = pd.DataFrame(encoded\_array, columns=encoder.get\_feature\_names\_out(['Color']))

print(df\_encoded)

Now, each color is **represented as a separate column** with values **0 or 1**.

**5. Partitioning a Dataset into Training and Test Sets**

When training a machine learning model, we must **split** the dataset into:

* **Training set (80%)** → Used to train the model.
* **Test set (20%)** → Used to check how well the model performs on new data.

**Why is this important?**

* If we train and test on the same data, the model might **memorize** the data instead of learning patterns (**overfitting**).
* A separate test set ensures the model **generalizes well** to unseen data.

**How to Split the Data in Python?**

python

CopyEdit

from sklearn.model\_selection import train\_test\_split

import numpy as np

# Sample dataset

X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])

y = np.array([0, 1, 0, 1, 0]) # Labels

# Split data: 80% training, 20% testing

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

print("Training Data:", X\_train)

print("Testing Data:", X\_test)

* train\_test\_split() automatically **shuffles** the data and splits it into training and testing sets.
* The random\_state=42 ensures **consistent** results every time you run the code.

**Final Summary**

✅ **Handling Categorical Data** → Convert text data into numbers for machine learning.  
✅ **Mapping Ordinal Features** → Assign numerical values while keeping the order (e.g., Small = 1, Medium = 2, Large = 3).  
✅ **Encoding Class Labels** → Convert target labels into numbers using LabelEncoder().  
✅ **One-Hot Encoding for Nominal Features** → Convert categorical variables into separate binary columns.  
✅ **Partitioning Dataset into Training & Test Sets** → Split data into training (80%) and testing (20%) to prevent overfitting.

**1. Bringing Features onto the Same Scale (Feature Scaling)**

In a dataset, features (columns) can have different ranges of values. For example:

| **Feature** | **Value Range** |
| --- | --- |
| Age | 18 - 65 |
| Salary | 10,000 - 1,000,000 |

**Problem:**

* Some machine learning models (e.g., KNN, SVM, Gradient Descent) **perform poorly** if features are on **different scales**.
* Larger values (like salary) will **dominate** smaller values (like age).

**Solution:** We **scale** all features to a **similar range**.

**Two Main Scaling Techniques**

1. **Min-Max Scaling (Normalization)**
   * Converts values between **0 and 1**.
   * Formula:
   * 
   * **Used when data does not follow a normal distribution.**
   * Example in Python:

python

CopyEdit

from sklearn.preprocessing import MinMaxScaler

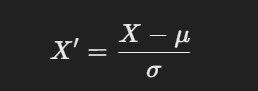
import numpy as np

data = np.array([[18], [25], [35], [45], [65]]) # Age column

scaler = MinMaxScaler()

scaled\_data = scaler.fit\_transform(data)

print(scaled\_data)

1. **Standardization (Z-Score Normalization)**
   * Converts values to have **mean = 0 and standard deviation = 1**.
   * Formula:
   * 

**Used when data follows a normal (Gaussian) distribution.**

* + Example in Python:

python

CopyEdit

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

standardized\_data = scaler.fit\_transform(data)

print(standardized\_data)

👉 **When to Use?**

* **Use Min-Max Scaling** when values have a fixed range (e.g., image pixel values 0-255).
* **Use Standardization** when data follows a **normal distribution**.

**2. Selecting Meaningful Features (Feature Selection)**

**Why is Feature Selection Important?**

* Too many features can **slow down the model** and cause **overfitting**.
* Selecting the most **important** features improves **accuracy and efficiency**.

**Types of Feature Selection Methods**

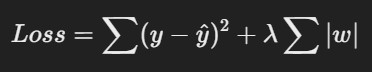
1. **L1 Regularization (Lasso Regression) – Sparse Solutions**
2. **Sequential Feature Selection Algorithms**

**3. Sparse Solutions with L1 Regularization (Lasso)**

L1 regularization, also known as **Lasso Regression**, is a technique that **automatically selects important features** and removes unnecessary ones.

**How Does L1 Regularization Work?**

* It **adds a penalty** to large coefficients in regression models.
* If a feature is **not important**, its coefficient becomes **zero**, meaning the feature is **removed**.

**Formula for Lasso Regression:**

where:

* y = actual values
* y^ = predicted values
* w = feature coefficients
* λ = regularization strength (higher value means more features will be removed)

**Example: Lasso Regression for Feature Selection**

python

CopyEdit

from sklearn.linear\_model import Lasso

import numpy as np

# Sample dataset with unnecessary features

X = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

y = np.array([1, 2, 3]) # Target

lasso = Lasso(alpha=0.1) # Regularization strength

lasso.fit(X, y)

print("Feature importance:", lasso.coef\_) # Some coefficients may be zero

* Features with **zero coefficients** are **removed**.

👉 **Use L1 Regularization when you have many features and want to automatically remove the less important ones.**

**4. Sequential Feature Selection Algorithms**

Sometimes, instead of using L1 regularization, we **manually** choose the best features **step by step** using **Sequential Feature Selection (SFS).**

**How Does SFS Work?**

* It **adds or removes** features one at a time and **evaluates performance**.
* **Forward Selection** → Start with no features, **add** the most important one at each step.
* **Backward Elimination** → Start with all features, **remove** the least important one at each step.

**Example: Using SequentialFeatureSelector in Python**

python

CopyEdit

from sklearn.feature\_selection import SequentialFeatureSelector

from sklearn.linear\_model import LinearRegression

from sklearn.datasets import make\_regression

# Create a dataset

X, y = make\_regression(n\_samples=100, n\_features=5, noise=0.1)

# Define model

model = LinearRegression()

# Apply forward feature selection

sfs = SequentialFeatureSelector(model, n\_features\_to\_select=3, direction='forward')

X\_selected = sfs.fit\_transform(X, y)

print("Selected Features Shape:", X\_selected.shape) # Should have 3 features

* The algorithm automatically **chooses the best features** based on model accuracy.

👉 **Use Sequential Feature Selection when you want to carefully pick the best features step by step.**

**Final Summary**

✅ **Feature Scaling** → Normalize features using Min-Max Scaling or Standardization.  
✅ **L1 Regularization (Lasso)** → Automatically removes less important features by setting coefficients to zero.  
✅ **Sequential Feature Selection (SFS)** → Manually selects the best features by adding/removing them step by step.