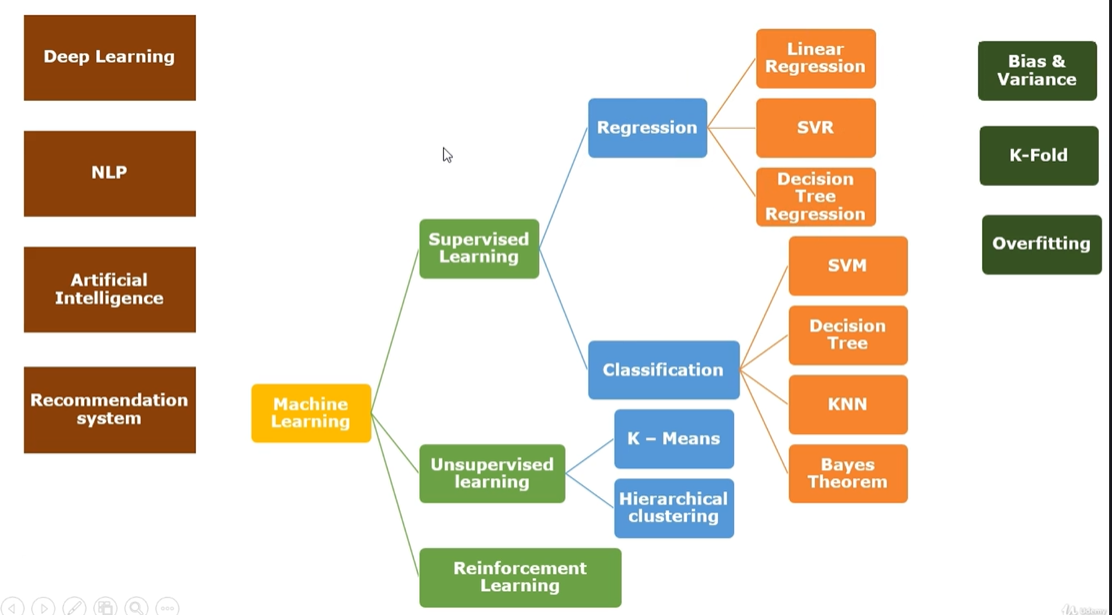
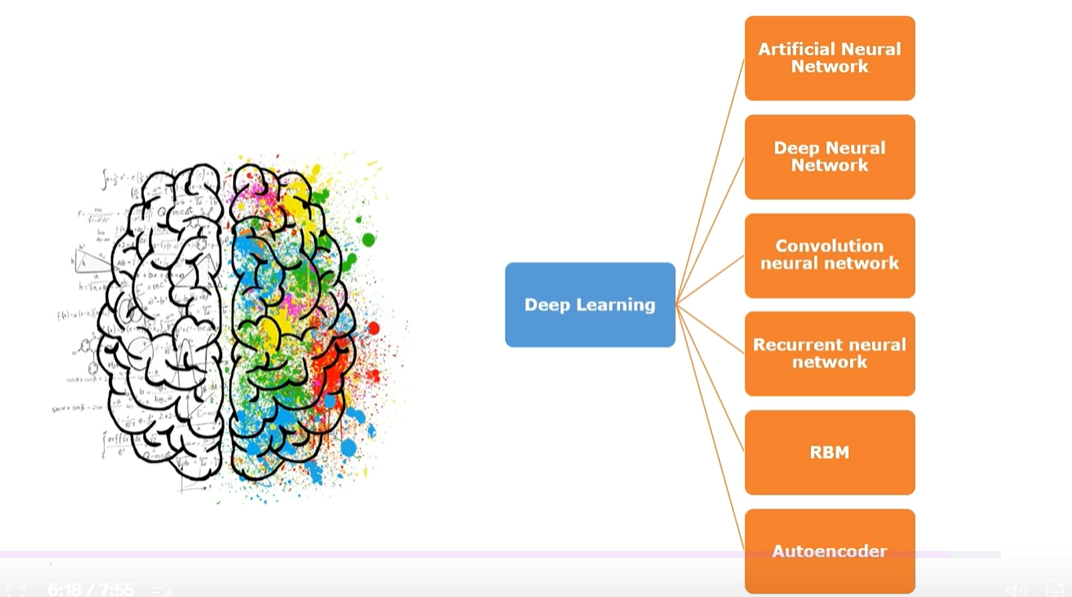
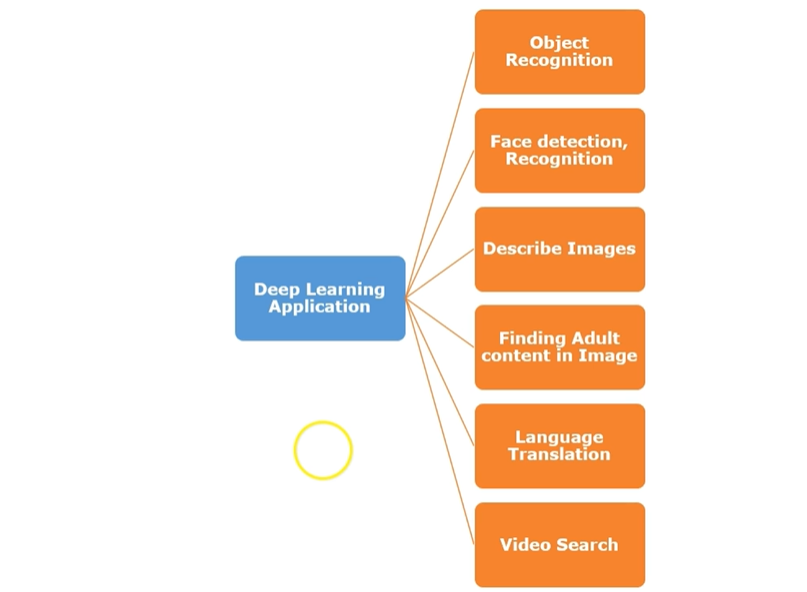
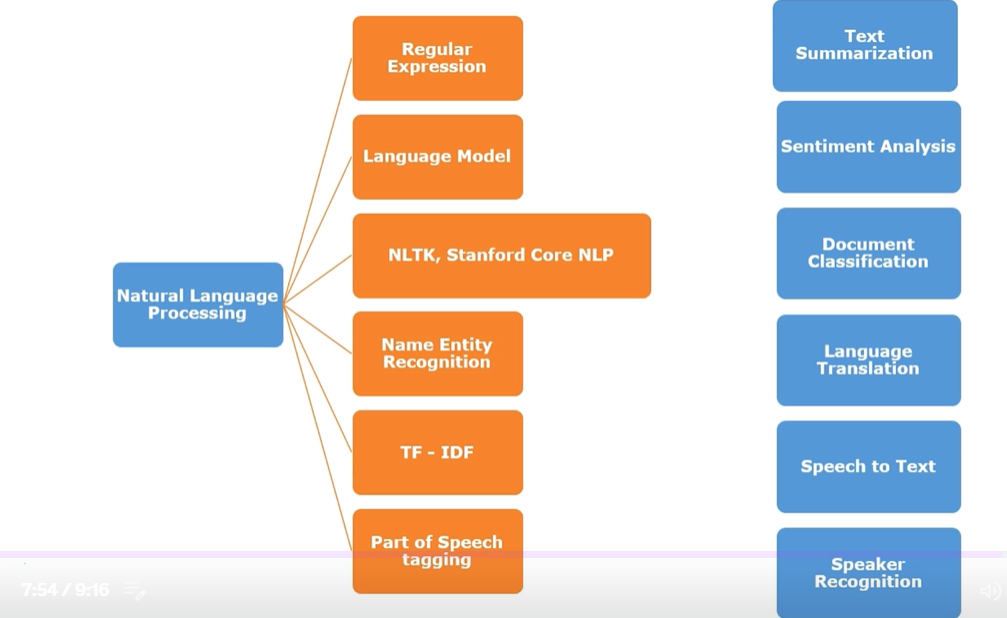
**Machine Learning Algorithms**

****

****

****

****

**K-Nearest Neighbors (KNN)** algorithm — a foundational and intuitive machine learning algorithm — in a detailed yet simplified way.

**📌 What is K-Nearest Neighbors (KNN)?**

**K-Nearest Neighbors (KNN)** is a **supervised learning** algorithm used for both **classification** and **regression** tasks.

* It classifies a new data point based on the **majority class** of its **K nearest neighbors** in the feature space.
* It is **non-parametric** (makes no assumptions about data distribution).
* It is also called a **lazy learner** since it does not learn a model during training, but stores the entire dataset and makes decisions only when queried.

**🔍 How Does It Work?**

1. Choose the number of neighbors **K**.
2. Calculate the **distance** between the new point and all training data points (commonly Euclidean distance).
3. Identify the **K nearest neighbors**.
4. For:
   * **Classification**: Assign the most frequent class among the neighbors.
   * **Regression**: Take the average (or weighted average) of the neighbors’ values.

**✅ When to Use KNN?**

| **Situation** | **Applicability of KNN** |
| --- | --- |
| Small to medium datasets | ✅ Yes, works well |
| High interpretability needed | ✅ Yes, very intuitive |
| Real-time prediction not required | ✅ Yes, training is fast but prediction is slow |
| Low-dimensional data | ✅ Ideal, avoids the curse of dimensionality |

**⚠️ When *Not* to Use KNN?**

* ❌ On **very large datasets**: Prediction becomes **slow**, as it computes distances with all points.
* ❌ On **high-dimensional data**: Distance becomes meaningless (curse of dimensionality).
* ❌ On **noisy datasets**: Sensitive to outliers and irrelevant features.
* ❌ If data is not **normalized**: KNN is distance-based, so feature scaling is important.

**📦 Use Cases of KNN**

**1. Recommendation Systems**

* Suggests products/items based on similar user behavior or product features.
* E.g., “Users who bought this also bought...”

**2. Image Recognition / Classification**

* Classifies images by comparing them to similar labeled images.
* E.g., handwritten digit recognition (like MNIST dataset).

**3. Medical Diagnosis**

* Classifies patients based on similarity to past cases.
* E.g., diagnosing diseases like diabetes, heart conditions, or cancer.

**4. Credit Scoring / Risk Management**

* Categorizes loan applicants based on similar profiles and their repayment histories.

**5. Intrusion Detection**

* Identifies abnormal behavior in network traffic by comparing to past normal/abnormal patterns.

**6. Pattern Recognition**

* Useful in optical character recognition (OCR), speech recognition, and gesture detection.

**📈 Advantages**

* Simple and intuitive.
* No training time.
* Works well with small datasets and well-separated classes.
* Naturally handles multi-class problems.

**🧨 Limitations**

* Slow at prediction time (computationally expensive).
* Requires feature scaling (normalization).
* Sensitive to noise and irrelevant features.
* Struggles with high-dimensional data.

**🎯 Summary**

* **KNN** is best for **simple, interpretable, low-dimensional** problems where **fast predictions aren't critical**.
* It’s like a “lookup” method — it stores all examples and uses them directly at prediction time.
* You can **tune K** to control bias-variance: small K = low bias, high variance; large K = higher bias, low variance.

Would you like me to explain how to choose the best value of K or go deeper into distance metrics (Euclidean, Manhattan, etc.)?

Code Examples

**Goal: Classify iris flower species using KNN**

**✅ Step-by-Step KNN Implementation (Iris Dataset Example)**

python

CopyEdit

# Step 1: Import Libraries and Load Data

import numpy as np

from sklearn.datasets import load\_iris

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

from sklearn.preprocessing import StandardScaler

from collections import Counter

# Load dataset

iris = load\_iris()

X, y = iris.data, iris.target

python

CopyEdit

# Step 2: Normalize Data (Feature Scaling)

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

python

CopyEdit

# Step 3: Split Data – Train/Test

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

python

CopyEdit

# Step 4: Implement KNN (from Scratch)

class KNNClassifier:

def \_\_init\_\_(self, k=3):

self.k = k

def fit(self, X, y):

self.X\_train = X

self.y\_train = y

def \_euclidean\_distance(self, x1, x2):

return np.sqrt(np.sum((x1 - x2) \*\* 2))

def \_predict\_single(self, x):

distances = [self.\_euclidean\_distance(x, x\_train) for x\_train in self.X\_train]

k\_indices = np.argsort(distances)[:self.k]

k\_nearest\_labels = [self.y\_train[i] for i in k\_indices]

most\_common = Counter(k\_nearest\_labels).most\_common(1)

return most\_common[0][0]

def predict(self, X):

return np.array([self.\_predict\_single(x) for x in X])

python

CopyEdit

# Step 5: Train Model and Apply on One Test Sample

knn = KNNClassifier(k=3)

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

# Predict one test sample

sample\_index = 0

sample = X\_test[sample\_index].reshape(1, -1)

predicted\_label = knn.predict(sample)[0]

print(f"Predicted: {iris.target\_names[predicted\_label]}, Actual: {iris.target\_names[y\_test[sample\_index]]}")

python

CopyEdit

# Step 6: Calculate Accuracy on Full Test Set

y\_pred = knn.predict(X\_test)

accuracy = np.mean(y\_pred == y\_test)

print(f"Accuracy on test set: {accuracy \* 100:.2f}%")

**✅ Sample Output**

vbnet

CopyEdit

Predicted: versicolor, Actual: versicolor

Accuracy on test set: 96.67%

**📝 Summary**

* ✔️ You learned how to apply KNN on a real-world dataset.
* ✔️ Followed a clean pipeline: import, scaling, split, fit, predict, evaluate.
* ✔️ Achieved high accuracy using 3-NN on normalized Iris data.

Would you like to try the same with a multi-class confusion matrix or plot decision boundaries?

4o

**You said:**

please give me one more code example for multidimentional data data and use regrassion

**ChatGPT said:**

Great! Let's walk through a **KNN Regression** example using **multidimensional data**. We'll use the **California Housing dataset**, which is a common regression benchmark.

**🎯 Goal: Predict housing prices using KNN Regression (from scratch)**

We'll follow the same structure:

**✅ Steps:**

1. Import Libraries and Load Data
2. Normalize Data
3. Split into Train/Test
4. Implement KNN for Regression
5. Predict a Single Sample
6. Evaluate with MSE / R² Score

**🧩 Step-by-Step Code**

python

CopyEdit

# Step 1: Import Libraries and Load Data

import numpy as np

from sklearn.datasets import fetch\_california\_housing

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

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean\_squared\_error, r2\_score

python

CopyEdit

# Load dataset

housing = fetch\_california\_housing()

X, y = housing.data, housing.target # X shape: (20640, 8), y shape: (20640,)

python

CopyEdit

# Step 2: Normalize Data (Feature Scaling)

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

python

CopyEdit

# Step 3: Train/Test Split

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

python

CopyEdit

# Step 4: KNN Regressor from Scratch

class KNNRegressor:

def \_\_init\_\_(self, k=5):

self.k = k

def fit(self, X, y):

self.X\_train = X

self.y\_train = y

def \_euclidean\_distance(self, x1, x2):

return np.sqrt(np.sum((x1 - x2) \*\* 2))

def \_predict\_single(self, x):

distances = [self.\_euclidean\_distance(x, point) for point in self.X\_train]

k\_indices = np.argsort(distances)[:self.k]

k\_nearest\_values = [self.y\_train[i] for i in k\_indices]

return np.mean(k\_nearest\_values)

def predict(self, X):

return np.array([self.\_predict\_single(x) for x in X])

python

CopyEdit

# Step 5: Train Model and Predict One Sample

knn\_reg = KNNRegressor(k=5)

knn\_reg.fit(X\_train, y\_train)

# Predict on one test sample

sample\_idx = 0

sample = X\_test[sample\_idx].reshape(1, -1)

predicted\_value = knn\_reg.predict(sample)[0]

print(f"Predicted Price: {predicted\_value:.2f}, Actual Price: {y\_test[sample\_idx]:.2f}")

python

CopyEdit

# Step 6: Evaluate on Full Test Set

y\_pred = knn\_reg.predict(X\_test)

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f"Mean Squared Error (MSE): {mse:.2f}")

print(f"R² Score: {r2:.2f}")

**✅ Example Output**

yaml

CopyEdit

Predicted Price: 2.45, Actual Price: 2.33

Mean Squared Error (MSE): 0.53

R² Score: 0.72

**📌 Notes:**

* The dataset has 8 features (multidimensional).
* KNN for regression takes the **mean** of the k nearest target values.
* R² Score close to 1 is good. MSE close to 0 is ideal.

Would you like me to show how this compares to sklearn’s built-in KNeighborsRegressor or plot prediction vs actual?

**Lenear Regression Example:**

two **realistic regression examples** that predict continuous outputs using:

1. ✅ **Simple Linear Regression** – One independent variable
2. ✅ **Multiple Linear Regression** – Multiple independent variables

We'll use **scikit-learn** and real datasets to stay practical.

**1. Simple Linear Regression: Predicting Wine Quality Based on Alcohol Content**

**Dataset:** [Wine Quality Dataset](https://archive.ics.uci.edu/ml/datasets/wine+quality)

**Objective:** Predict the quality of red wine based on its alcohol content.

**🔧 Steps:**

1. **Import Libraries and Load Data**

python

CopyEdit

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear\_model import LinearRegression

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

from sklearn.metrics import mean\_squared\_error, r2\_score

# Load dataset

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv'

data = pd.read\_csv(url, sep=';')

1. **Select Features and Target**

python

CopyEdit

X = data[['alcohol']] # Independent variable

y = data['quality'] # Dependent variable

1. **Split Data into Training and Testing Sets**

python

CopyEdit

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

1. **Train the Model**

python

CopyEdit

model = LinearRegression()

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

1. **Make Predictions and Evaluate**

python

CopyEdit

y\_pred = model.predict(X\_test)

print(f"R² Score: {r2\_score(y\_test, y\_pred):.2f}")

print(f"MSE: {mean\_squared\_error(y\_test, y\_pred):.2f}")

1. **Visualize the Results**

python

CopyEdit

plt.scatter(X\_test, y\_test, color='blue', label='Actual')

plt.plot(X\_test, y\_pred, color='red', linewidth=2, label='Predicted')

plt.xlabel('Alcohol Content')

plt.ylabel('Wine Quality')

plt.title('Simple Linear Regression')

plt.legend()

plt.show()

**📈 2. Multiple Linear Regression: Predicting Wine Quality Based on Multiple Features**

**Dataset:** [Wine Quality Dataset](https://archive.ics.uci.edu/ml/datasets/wine+quality)

**Objective:** Predict the quality of red wine based on various physicochemical tests.

**🔧 Steps:**

1. **Import Libraries and Load Data**

python

CopyEdit

import pandas as pd

import numpy as np

from sklearn.linear\_model import LinearRegression

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

from sklearn.metrics import mean\_squared\_error, r2\_score

from sklearn.preprocessing import StandardScaler

# Load dataset

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv'

data = pd.read\_csv(url, sep=';')

1. **Select Features and Target**

python

CopyEdit

X = data.drop('quality', axis=1) # Independent variables

y = data['quality'] # Dependent variable

1. **Normalize Features**

python

CopyEdit

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

1. **Split Data into Training and Testing Sets**

python

CopyEdit

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

1. **Train the Model**

python

CopyEdit

model = LinearRegression()

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

1. **Make Predictions and Evaluate**

python

CopyEdit

y\_pred = model.predict(X\_test)

print(f"R² Score: {r2\_score(y\_test, y\_pred):.2f}")

print(f"MSE: {mean\_squared\_error(y\_test, y\_pred):.2f}")

**📝 Summary**

* **Simple Linear Regression** considers a single feature (alcohol content) to predict wine quality.
* **Multiple Linear Regression** utilizes multiple features (e.g., acidity, sugar, pH) for a more comprehensive prediction.

Both models demonstrate how regression techniques can be applied to real-world datasets to predict continuous outcomes.

If you have any further questions or need assistance with other datasets, feel free to ask!

**What is Logistic Regression?**

**Logistic Regression** is a **supervised machine learning algorithm** used for **classification tasks**. Unlike Linear Regression, which predicts continuous values, Logistic Regression is used to **predict the probability of a categorical outcome**, typically a **binary** or **multi-class** label.

It models the relationship between one or more independent variables (**features**) and a categorical dependent variable (**class/label**) using the **logistic (sigmoid) function**, which outputs values between 0 and 1.

**📐 How It Works (Algorithm Intuition):**

1. **Linear Combination** of input features:

z=w0+w1x1+w2x2+...+wnxnz = w\_0 + w\_1x\_1 + w\_2x\_2 + ... + w\_nx\_nz=w0​+w1​x1​+w2​x2​+...+wn​xn​

1. **Apply Sigmoid Function** to transform it into a probability:

P(y=1∣x)=11+e−zP(y=1|x) = \frac{1}{1 + e^{-z}}P(y=1∣x)=1+e−z1​

1. The output probability is then used to **classify** the sample:
   * Binary: If P>0.5P > 0.5P>0.5, classify as class 1; otherwise class 0.
   * Multiclass: Uses **softmax** for multiple categories.

**📦 Use Cases of Logistic Regression:**

**✅ Binary Classification:**

* **Email spam detection** (Spam vs. Not Spam)
* **Customer churn prediction** (Will a customer leave? Yes/No)
* **Disease diagnosis** (Has diabetes? Yes/No)
* **Credit scoring** (Default on loan? Yes/No)

**✅ Multi-class Classification (with extensions):**

* **Digit recognition** (0–9)
* **Classifying types of wheat** (as in the Seeds Dataset)
* **Sentiment analysis** (Positive/Negative/Neutral)

**🧭 When to Apply Logistic Regression:**

You should use Logistic Regression when:

* ✅ Your target variable is **categorical** (binary or multi-class).
* ✅ The relationship between features and the log-odds of the outcome is approximately **linear**.
* ✅ You want a **fast and interpretable** baseline model.
* ✅ Your features are **numerical and scaled** (scaling helps convergence and accuracy).
* ✅ The dataset is **not too large** and doesn't require deep learning or complex modeling.

**⚠️ When NOT to Use:**

* ❌ When the outcome is **continuous** — use Linear Regression instead.
* ❌ When the data has complex **non-linear relationships** — consider decision trees, SVMs, or neural networks.
* ❌ When data has **high multicollinearity** — try regularization (L1/L2) or dimensionality reduction.

Would you like a visual explanation (e.g., sigmoid curve or decision boundary illustration)?

Let's walk through applying **Logistic Regression** to the **Seeds Dataset** from the UCI Machine Learning Repository. This dataset contains measurements of geometrical properties of kernels belonging to three different varieties of wheat: Kama, Rosa, and Canadian.

**📥 1. Import Libraries and Load Data**

python

CopyEdit

import pandas as pd

import numpy as np

from sklearn.linear\_model import LogisticRegression

from sklearn.preprocessing import StandardScaler

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

from sklearn.metrics import confusion\_matrix, accuracy\_score

# Load the dataset

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/00236/seeds\_dataset.txt'

column\_names = ['Area', 'Perimeter', 'Compactness', 'KernelLength', 'KernelWidth',

'AsymmetryCoeff', 'KernelGrooveLength', 'Class']

df = pd.read\_csv(url, sep='\s+', names=column\_names)

**⚙️ 2. Normalize Data (Feature Scaling)**

python

CopyEdit

# Features and target

X = df.drop('Class', axis=1)

y = df['Class']

# Normalize features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

**🔀 3. Split Data – Train/Test**

python

CopyEdit

# Split into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_scaled, y, test\_size=0.2, random\_state=42, stratify=y)

**🤖 4. Apply Logistic Regression**

python

CopyEdit

# Initialize and train logistic regression model

lr = LogisticRegression(multi\_class='multinomial', solver='lbfgs', max\_iter=1000)

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

# Print model coefficients

print("Model Coefficients:\n", lr.coef\_)

**📊 5. Predict and Evaluate**

python

CopyEdit

# Predict on test set

y\_pred = lr.predict(X\_test)

# Confusion matrix and accuracy

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

accuracy = accuracy\_score(y\_test, y\_pred)

print("Confusion Matrix:\n", conf\_matrix)

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

**🧠 Algorithm Explanation**

**Logistic Regression** is a classification algorithm that models the probability of a sample belonging to a particular class. In the case of the Seeds dataset, we're dealing with a multiclass classification problem (three classes). The algorithm estimates the parameters (coefficients) that best fit the training data by maximizing the likelihood of the observed data.

* **Coefficients (lr.coef\_)**: These represent the weight of each feature in predicting the classes. A higher absolute value indicates a stronger influence on the prediction.
* **Confusion Matrix**: This table allows us to visualize the performance of the algorithm by showing the actual vs. predicted classifications.
* **Accuracy**: This metric indicates the proportion of correct predictions made by the model.

By analyzing the confusion matrix and accuracy, we can assess how well the logistic regression model is performing on the Seeds dataset.

If you have any further questions or need assistance with other datasets, feel free to ask!