**Data mining lab programs**

1. **Perform data cleaning for a given data set.**

# ==============================

# Data Cleaning Program

# ==============================

import pandas as pd

import numpy as np

# -------------------------------

# Load Dataset (replace filename.csv with your dataset)

# -------------------------------

# If using Google Colab, upload file first:

# from google.colab import files

# uploaded = files.upload()

# Example: load CSV file

df = pd.read\_csv("sample\_dataset.csv")

print("=== Original Dataset (first 5 rows) ===")

print(df.head())

print("\nDataset Shape:", df.shape)

# -------------------------------

# 1. Check Missing Values

# -------------------------------

print("\n=== Missing Values Before Cleaning ===")

print(df.isnull().sum())

# Fill missing values (numerical -> mean, categorical -> mode)

for col in df.columns:

if df[col].dtype == "object": # categorical

df[col].fillna(df[col].mode()[0], inplace=True)

else: # numerical

df[col].fillna(df[col].mean(), inplace=True)

# -------------------------------

# 2. Handle Duplicates

# -------------------------------

print("\nNumber of Duplicates Before Cleaning:", df.duplicated().sum())

df.drop\_duplicates(inplace=True)

# -------------------------------

# 3. Standardize Column Names

# -------------------------------

df.columns = df.columns.str.strip().str.lower().str.replace(" ", "\_")

# -------------------------------

# 4. Remove Outliers (Z-score method)

# -------------------------------

from scipy import stats

numeric\_cols = df.select\_dtypes(include=[np.number]).columns

z\_scores = np.abs(stats.zscore(df[numeric\_cols]))

df = df[(z\_scores < 3).all(axis=1)]

# -------------------------------

# 5. Convert Categorical Data to Numeric (One-Hot Encoding)

# -------------------------------

df = pd.get\_dummies(df, drop\_first=True)

# -------------------------------

# Final Cleaned Dataset

# -------------------------------

print("\n=== Cleaned Dataset (first 5 rows) ===")

print(df.head())

print("\nCleaned Dataset Shape:", df.shape)

# Save cleaned dataset

df.to\_csv("cleaned\_dataset.csv", index=False)

print("\n✅ Cleaned dataset saved as cleaned\_dataset.csv")

**Problem Statements for Data Cleaning**

**1. Handling Missing Values**

**Problem:**  
A dataset of students' marks has missing values in the "Math\_Score" column. Write a Python program to:

* Identify missing values.
* Replace them with the **mean score** of the column.

*Hint:* Use isnull(), fillna() in pandas.

**2. Removing Duplicates**

**Problem:**  
A dataset of employee records contains duplicate rows (same employee entered twice). Write a Python program to:

* Detect duplicate records.
* Remove them and display the cleaned dataset.

*Hint:* Use duplicated() and drop\_duplicates().

**3. Handling Inconsistent Data**

**Problem:**  
A dataset contains a "Gender" column with inconsistent entries like "Male", "male", "M", "FEMALE", "female", "F".  
Write a Python program to:

* Standardize the values to "Male" and "Female".

*Hint:* Use replace() or apply a custom mapping function.

**4. Handling Outliers**

**Problem:**  
A dataset of product prices has some extremely high values due to data entry errors.  
Write a Python program to:

* Detect outliers using the **IQR (Interquartile Range)** method.
* Remove the rows containing outliers.

*Hint:* Use quantile() to calculate Q1, Q3 and filter.

**5. Fixing Data Entry Errors**

**Problem:**  
A dataset of city names contains spelling errors like "Newyork, new york, NEWYORK, nyc".  
Write a Python program to:

* Correct them all to "New York".

*Hint:* Use string methods (str.lower(), replace()) or mapping dictionary.

1. **Perform data reduction for a given data set**

import pandas as pd

import numpy as np

from sklearn.decomposition import PCA

from sklearn.preprocessing import LabelEncoder

import matplotlib.pyplot as plt

# -----------------------------

# Step 1: Create a sample dataset

# -----------------------------

data = {

'Age': [23, 25, 47, 52, 46, 56, 55, 60, 62, 61],

'Income': [25000, 29000, 47000, 52000, 46000, 56000, 55000, 60000, 62000, 61000],

'SpendingScore': [30, 40, 60, 65, 50, 70, 68, 80, 85, 78],

'Gender': ['Male', 'Female', 'Female', 'Male', 'Female', 'Male', 'Male', 'Female', 'Female', 'Male']

}

df = pd.DataFrame(data)

print("Original Dataset:\n", df)

# ====================================================

# 1. DIMENSIONALITY REDUCTION (PCA)

# ====================================================

pca = PCA(n\_components=2)

features = df[['Age', 'Income', 'SpendingScore']]

pca\_result = pca.fit\_transform(features)

pca\_df = pd.DataFrame(pca\_result, columns=['PC1','PC2'])

print("\nDimensionality Reduction using PCA:\n", pca\_df)

# Visualization: PCA scatter plot

plt.figure(figsize=(6,4))

plt.scatter(pca\_df['PC1'], pca\_df['PC2'], c='blue', s=50)

plt.title("PCA Result (2D)")

plt.xlabel("PC1")

plt.ylabel("PC2")

plt.show()

# ====================================================

# 2. NUMEROSITY REDUCTION (Histogram, Regression, Sampling)

# ====================================================

# Histogram (for Age)

plt.figure(figsize=(6,4))

plt.hist(df['Age'], bins=3, edgecolor='black')

plt.title("Histogram of Age")

plt.xlabel("Age")

plt.ylabel("Frequency")

plt.show()

# Simple linear regression model (Income ~ Age)

coeffs = np.polyfit(df['Age'], df['Income'], 1)

regression\_line = np.polyval(coeffs, df['Age'])

plt.figure(figsize=(6,4))

plt.scatter(df['Age'], df['Income'], label="Data")

plt.plot(df['Age'], regression\_line, color='red', label="Regression Line")

plt.title("Regression (Income vs Age)")

plt.xlabel("Age")

plt.ylabel("Income")

plt.legend()

plt.show()

# Sampling (take 5 random rows)

sample\_df = df.sample(n=5, random\_state=42)

print("\nSampled Data:\n", sample\_df)

# ====================================================

# 3. DATA COMPRESSION (Dummy variable encoding)

# ====================================================

le = LabelEncoder()

df['Gender\_Compressed'] = le.fit\_transform(df['Gender'])

print("\nData Compression (Label Encoding for Gender):\n", df[['Gender','Gender\_Compressed']])

# ====================================================

# 4. DISCRETIZATION (Binning Age)

# ====================================================

labels = ['Young','Middle-aged','Senior']

df['AgeGroup'] = pd.cut(df['Age'], bins=[20,35,50,70], labels=labels)

print("\nDiscretization (Age → AgeGroup):\n", df[['Age','AgeGroup']])

# ====================================================

# 5. AGGREGATION (Average Income by AgeGroup)

# ====================================================

agg\_df = df.groupby('AgeGroup')['Income'].mean()

print("\nAggregation (Avg Income by AgeGroup):\n", agg\_df)

# Visualization: Aggregation bar chart

agg\_df.plot(kind='bar', color='orange', figsize=(6,4))

plt.title("Average Income by AgeGroup")

plt.xlabel("Age Group")

plt.ylabel("Average Income")

plt.show()

# ====================================================

# 6. SAMPLING (already shown above)

# ====================================================

print("\nFinal Sample (5 rows from dataset):\n", sample\_df)

**Problem Statements on Data Reduction**

**1. Dimensionality Reduction with Correlation Filtering**

**Problem:**  
You are given a dataset of car features (Horsepower, EngineSize, Cylinders, Weight, MPG).

* Remove highly correlated attributes (correlation coefficient > 0.85).
* Display the reduced dataset with only independent features.

**Input Sample (CSV columns):**

Car, Horsepower, EngineSize, Cylinders, Weight, MPG

A, 130, 2.0, 4, 2600, 30

B, 220, 3.0, 6, 3400, 24

C, 300, 4.0, 8, 4000, 18

**2. Numerosity Reduction using Clustering**

**Problem:**  
You have a dataset of shopping mall customers (Annual Income, Spending Score).

* Apply **k-Means clustering** with k=3.
* Replace original records with **cluster centroids** to reduce data size.

**Input Sample:**

CustomerID, AnnualIncome, SpendingScore

1, 15000, 39

2, 18000, 81

3, 35000, 6

4, 40000, 77

**3. Data Compression using Principal Component Analysis (PCA)**

**Problem:**  
A dataset contains 6 numerical features about wines (Alcohol, Ash, Magnesium, Flavanoids, Phenols, ColorIntensity).

* Apply PCA to reduce them to **2 components**.
* Plot the transformed data in 2D.

**Input Sample:**

WineID, Alcohol, Ash, Magnesium, Flavanoids, Phenols, ColorIntensity

1, 14.2, 2.1, 127, 3.1, 2.0, 5.6

2, 13.8, 2.4, 100, 2.8, 1.9, 4.3

**4. Discretization using Decision Tree (Entropy-based Binning)**

**Problem:**  
You are given a dataset of student marks in Math, Science, English.

* Convert continuous marks into grade categories (Low, Medium, High) using **Decision Tree-based discretization**.

**Input Sample:**

Student, Math, Science, English

A, 56, 68, 72

B, 78, 81, 65

C, 34, 40, 50

**5. Sampling with Stratification**

**Problem:**  
A dataset of bank customers contains CustomerID, AgeGroup, Balance.

* Perform **stratified sampling** so that the distribution of AgeGroup (Young, Middle, Senior) is preserved in the sample.

**Input Sample:**

CustomerID, AgeGroup, Balance

1, Young, 2500

2, Middle, 4600

3, Senior, 8900

4, Young, 3000

1. **Perform data transformation for a given data set.**

import pandas as pd

import numpy as np

from sklearn.preprocessing import MinMaxScaler, StandardScaler

import matplotlib.pyplot as plt

# -----------------------------

# Step 1: Create a sample dataset

# -----------------------------

data = {

'Age': [23, 25, 47, 52, 46, 56, 55, 60, 62, 61],

'Income': [25000, 29000, 47000, 52000, 46000, 56000, 55000, 60000, 62000, 61000],

'SpendingScore': [30, 40, 60, 65, 50, 70, 68, 80, 85, 78]

}

df = pd.DataFrame(data)

print("Original Dataset:\n", df)

# ====================================================

# 1. NORMALIZATION & STANDARDIZATION

# ====================================================

scaler\_minmax = MinMaxScaler()

scaler\_std = StandardScaler()

# Min-Max Normalization (0-1 scale)

df['Income\_MinMax'] = scaler\_minmax.fit\_transform(df[['Income']])

# Z-score Standardization

df['Income\_Standardized'] = scaler\_std.fit\_transform(df[['Income']])

print("\nAfter Normalization & Standardization:\n", df[['Income','Income\_MinMax','Income\_Standardized']])

# Visualization

plt.figure(figsize=(6,4))

plt.plot(df['Income'], label='Original Income', marker='o')

plt.plot(df['Income\_MinMax'], label='MinMax Normalized', marker='x')

plt.plot(df['Income\_Standardized'], label='Standardized', marker='s')

plt.title("Normalization & Standardization")

plt.xlabel("Record Index")

plt.ylabel("Value")

plt.legend()

plt.show()

# ====================================================

# 2. AGGREGATION

# ====================================================

# Aggregate to compute average spending per age group

labels = ['Young','Middle-aged','Senior']

df['AgeGroup'] = pd.cut(df['Age'], bins=[20,35,50,70], labels=labels)

agg\_result = df.groupby('AgeGroup')['SpendingScore'].mean()

print("\nAggregation (Avg SpendingScore by AgeGroup):\n", agg\_result)

# Bar chart for aggregation

agg\_result.plot(kind='bar', color='skyblue', figsize=(6,4))

plt.title("Average Spending Score by AgeGroup")

plt.xlabel("Age Group")

plt.ylabel("Avg Spending Score")

plt.show()

# ====================================================

# 3. GENERALIZATION

# ====================================================

# Example: Replace detailed income with categories

income\_bins = [20000,40000,60000,70000]

income\_labels = ['Low','Medium','High']

df['IncomeCategory'] = pd.cut(df['Income'], bins=income\_bins, labels=income\_labels)

print("\nGeneralization (Income → IncomeCategory):\n", df[['Income','IncomeCategory']])

# ====================================================

# 4. ATTRIBUTE CONSTRUCTION (Feature Engineering)

# ====================================================

# Example: Create a new attribute - Income to Age Ratio

df['Income\_to\_Age\_Ratio'] = df['Income'] / df['Age']

print("\nAttribute Construction (Income\_to\_Age\_Ratio):\n", df[['Age','Income','Income\_to\_Age\_Ratio']])

# Visualization of new feature

plt.figure(figsize=(6,4))

plt.scatter(df['Age'], df['Income\_to\_Age\_Ratio'], c='green', s=60)

plt.title("Constructed Feature: Income-to-Age Ratio")

plt.xlabel("Age")

plt.ylabel("Income/Age")

plt.show()

# ====================================================

# Final Transformed Dataset

# ====================================================

print("\nFinal Transformed Dataset:\n", df)

**Problem Statements on Data Transformation**

**1. Log Transformation for Skewed Data**

**Problem:**  
You are given a dataset of company revenues (Revenue). The values are highly skewed with some very large outliers.

* Apply **log transformation** to normalize the distribution.
* Plot before and after transformation.

**Input Sample:**

Company, Revenue

A, 1200

B, 15000

C, 300000

D, 800

**2. One-Hot Encoding of Categorical Variables**

**Problem:**  
A dataset of fruits has a categorical column FruitType (Apple, Banana, Orange).

* Transform this column into **dummy/one-hot variables** so that ML models can use it.

**Input Sample:**

FruitID, FruitType, Price

1, Apple, 120

2, Banana, 60

3, Orange, 80

4, Apple, 100

**3. Polynomial Feature Transformation**

**Problem:**  
You are given a dataset with features Area and Rooms for houses.

* Create new features by applying a **polynomial transformation** (e.g., Area², Rooms², Area\*Rooms).

**Input Sample:**

HouseID, Area, Rooms, Price

1, 1200, 3, 250000

2, 1500, 4, 320000

3, 900, 2, 180000

**4. Scaling with Decimal Scaling Method**

**Problem:**  
A dataset contains customer account balances (Balance).

* Transform the values into a **scaled format using decimal scaling**.

**Input Sample:**

CustomerID, Balance

1, 25000

2, 180000

3, 520

4, 3500

**5. Feature Construction from Date Attributes**

**Problem:**  
A dataset contains a TransactionDate column.

* Extract new attributes like **Year, Month, Day, DayOfWeek** from it.

**Input Sample:**

TransactionID, TransactionDate, Amount

1, 2024-01-15, 2000

2, 2024-03-22, 3500

3, 2024-07-05, 1800

1. **Program for extracting the frequent itemset using Apriori Algorithm.**

# ============================================

# Apriori Algorithm with Association Rules

# ============================================

# Step 1: Sample Dataset (Transactions)

transactions = [

['Milk', 'Bread', 'Eggs'],

['Milk', 'Bread'],

['Milk', 'Eggs'],

['Bread', 'Eggs'],

['Milk', 'Bread', 'Eggs'],

['Bread', 'Butter'],

['Milk', 'Butter']

]

# Step 2: Function to calculate support

def calculate\_support(transactions, itemset):

count = 0

for transaction in transactions:

if itemset.issubset(set(transaction)):

count += 1

return count / len(transactions)

# Step 3: Candidate Generation

def generate\_candidates(frequent\_itemsets, k):

candidates = []

len\_f = len(frequent\_itemsets)

for i in range(len\_f):

for j in range(i+1, len\_f):

l1 = list(frequent\_itemsets[i])[:k-2]

l2 = list(frequent\_itemsets[j])[:k-2]

l1.sort(); l2.sort()

if l1 == l2:

candidates.append(frequent\_itemsets[i] | frequent\_itemsets[j])

return candidates

# Step 4: Apriori Algorithm

def apriori(transactions, min\_support):

item\_counts = {}

for transaction in transactions:

for item in transaction:

itemset = frozenset([item])

item\_counts[itemset] = item\_counts.get(itemset, 0) + 1

num\_transactions = len(transactions)

frequent\_itemsets = {}

# First pass (1-itemsets)

L1 = []

for itemset, count in item\_counts.items():

support = count / num\_transactions

if support >= min\_support:

L1.append(itemset)

frequent\_itemsets[itemset] = support

k = 2

Lk = L1

while Lk:

candidates = generate\_candidates(Lk, k)

item\_counts = {}

for candidate in candidates:

support = calculate\_support(transactions, candidate)

if support >= min\_support:

item\_counts[candidate] = support

Lk = list(item\_counts.keys())

frequent\_itemsets.update(item\_counts)

k += 1

return frequent\_itemsets

# Step 5: Generate Association Rules

def generate\_rules(frequent\_itemsets, min\_confidence):

rules = []

for itemset in frequent\_itemsets:

if len(itemset) > 1: # Only for itemsets of size >= 2

subsets = [frozenset([item]) for item in itemset]

for antecedent in subsets:

consequent = itemset - antecedent

if consequent:

support\_itemset = frequent\_itemsets[itemset]

support\_antecedent = frequent\_itemsets.get(antecedent, 0)

if support\_antecedent > 0:

confidence = support\_itemset / support\_antecedent

lift = confidence / frequent\_itemsets.get(consequent, 1e-6)

if confidence >= min\_confidence:

rules.append((set(antecedent), set(consequent), support\_itemset, confidence, lift))

return rules

# Step 6: Run Apriori + Rules

min\_support = 0.3

min\_confidence = 0.6

frequent\_itemsets = apriori(transactions, min\_support)

print("Frequent Itemsets:")

for itemset, support in frequent\_itemsets.items():

print(set(itemset), "=> Support:", round(support, 2))

rules = generate\_rules(frequent\_itemsets, min\_confidence)

print("\nAssociation Rules (min\_confidence =", min\_confidence, "):")

for antecedent, consequent, support, confidence, lift in rules:

print(f"{antecedent} -> {consequent} | support={round(support,2)} | confidence={round(confidence,2)} | lift={round(lift,2)}")

1. **Program for generation of strong association rule(FP- Growth).**

from collections import defaultdict, namedtuple

import itertools

# ==============================

# FP-Tree Node Definition

# ==============================

class FPNode:

def \_\_init\_\_(self, item, count, parent):

self.item = item

self.count = count

self.parent = parent

self.children = {}

self.link = None # next node with same item

def increment(self, count=1):

self.count += count

# ==============================

# FP-Tree Construction

# ==============================

class FPTree:

def \_\_init\_\_(self, transactions, min\_support):

self.min\_support = min\_support

self.header\_table = defaultdict(int)

# Count frequency of each item

for transaction in transactions:

for item in transaction:

self.header\_table[item] += 1

# Remove infrequent items

self.header\_table = {item: count for item, count in self.header\_table.items() if count >= min\_support}

# Sort header table by frequency

self.header\_table = dict(sorted(self.header\_table.items(), key=lambda x: (-x[1], x[0])))

self.root = FPNode(None, 1, None)

self.node\_links = {}

# Build FP-tree

for transaction in transactions:

# Keep only frequent items, ordered by frequency

ordered\_items = [item for item in transaction if item in self.header\_table]

ordered\_items.sort(key=lambda x: (-self.header\_table[x], x))

self.\_insert\_tree(ordered\_items, self.root)

def \_insert\_tree(self, items, node):

if not items:

return

first\_item = items[0]

if first\_item in node.children:

node.children[first\_item].increment()

else:

child = FPNode(first\_item, 1, node)

node.children[first\_item] = child

# Update node link

if first\_item in self.node\_links:

current = self.node\_links[first\_item]

while current.link is not None:

current = current.link

current.link = child

else:

self.node\_links[first\_item] = child

# Recursive insert

remaining\_items = items[1:]

self.\_insert\_tree(remaining\_items, node.children[first\_item])

# ==============================

# Mine Frequent Patterns

# ==============================

def ascend\_fpnode(node):

"""Ascend from leaf node to root"""

path = []

while node and node.parent and node.parent.item is not None:

node = node.parent

path.append(node.item)

return path

def find\_prefix\_paths(base\_pat, node):

"""Find prefix paths for an item"""

cond\_pats = []

while node is not None:

prefix\_path = ascend\_fpnode(node)

if prefix\_path:

cond\_pats.append(prefix\_path \* node.count)

node = node.link

return cond\_pats

def mine\_tree(tree, prefix, frequent\_itemsets):

items = sorted(tree.header\_table.items(), key=lambda p: (p[1], p[0]))

for base\_item, \_ in items:

new\_freq\_set = prefix.copy()

new\_freq\_set.add(base\_item)

frequent\_itemsets.append(new\_freq\_set)

# Build conditional FP-tree

cond\_pats = []

node = tree.node\_links[base\_item]

while node is not None:

path = ascend\_fpnode(node)

for \_ in range(node.count):

cond\_pats.append(path)

node = node.link

# Build conditional tree

cond\_tree = FPTree(cond\_pats, tree.min\_support)

if cond\_tree.header\_table:

mine\_tree(cond\_tree, new\_freq\_set, frequent\_itemsets)

# ==============================

# Association Rules Generation

# ==============================

def generate\_association\_rules(frequent\_itemsets, transactions, min\_conf=0.6):

rules = []

transaction\_list = list(map(set, transactions))

num\_trans = len(transactions)

# Support counts

support\_count = {}

for itemset in frequent\_itemsets:

support\_count[frozenset(itemset)] = sum(1 for t in transaction\_list if itemset.issubset(t))

# Generate rules

for itemset in frequent\_itemsets:

if len(itemset) < 2:

continue

for i in range(1, len(itemset)):

for antecedent in itertools.combinations(itemset, i):

antecedent = frozenset(antecedent)

consequent = frozenset(itemset) - antecedent

if consequent:

sup\_itemset = support\_count[frozenset(itemset)] / num\_trans

sup\_antecedent = support\_count[antecedent] / num\_trans

conf = sup\_itemset / sup\_antecedent

lift = conf / (support\_count[consequent] / num\_trans)

if conf >= min\_conf:

rules.append((set(antecedent), set(consequent), sup\_itemset, conf, lift))

return rules

# ==============================

# Example Run

# ==============================

transactions = [

['Milk', 'Bread', 'Eggs'],

['Milk', 'Bread'],

['Milk', 'Eggs'],

['Bread', 'Eggs'],

['Milk', 'Bread', 'Eggs'],

['Bread', 'Butter'],

['Milk', 'Butter']

]

min\_support = 2

tree = FPTree(transactions, min\_support)

frequent\_itemsets = []

mine\_tree(tree, set(), frequent\_itemsets)

print("Frequent Itemsets:")

for itemset in frequent\_itemsets:

print(itemset)

print("\nStrong Association Rules (min\_conf = 0.6):")

rules = generate\_association\_rules(frequent\_itemsets, transactions, min\_conf=0.6)

for antecedent, consequent, support, confidence, lift in rules:

print(f"{antecedent} => {consequent} (support={support:.2f}, confidence={confidence:.2f}, lift={lift:.2f})")

1. **Program to find the classification rule and classification accuracy using decision tree, Bayesian classification and back propagation algorithm**

# ===========================================

# Classification: Decision Tree, Naive Bayes, Backpropagation

# ===========================================

import pandas as pd

from sklearn.datasets import load\_iris

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

from sklearn.tree import DecisionTreeClassifier, export\_text

from sklearn.naive\_bayes import GaussianNB

from sklearn.neural\_network import MLPClassifier

from sklearn.metrics import accuracy\_score, classification\_report

# -------------------------------

# Load Dataset (Iris)

# -------------------------------

iris = load\_iris()

X = pd.DataFrame(iris.data, columns=iris.feature\_names)

y = pd.Series(iris.target)

# Train-Test Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42, stratify=y)

# ==============================

# 1. Decision Tree Classifier

# ==============================

dt = DecisionTreeClassifier(criterion="entropy", random\_state=42)

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

# Rules (text format)

tree\_rules = export\_text(dt, feature\_names=list(X.columns))

print("==== Decision Tree Rules ====")

print(tree\_rules)

y\_pred\_dt = dt.predict(X\_test)

print("\nDecision Tree Accuracy:", accuracy\_score(y\_test, y\_pred\_dt))

print(classification\_report(y\_test, y\_pred\_dt, target\_names=iris.target\_names))

# ==============================

# 2. Naive Bayes Classifier

# ==============================

nb = GaussianNB()

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

y\_pred\_nb = nb.predict(X\_test)

print("\n==== Naive Bayes Classification ====")

print("Naive Bayes Accuracy:", accuracy\_score(y\_test, y\_pred\_nb))

print(classification\_report(y\_test, y\_pred\_nb, target\_names=iris.target\_names))

# ==============================

# 3. Backpropagation (Neural Network - MLP)

# ==============================

mlp = MLPClassifier(hidden\_layer\_sizes=(10,), max\_iter=1000, random\_state=42)

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

y\_pred\_mlp = mlp.predict(X\_test)

print("\n==== Backpropagation Neural Network Classification ====")

print("Neural Network Accuracy:", accuracy\_score(y\_test, y\_pred\_mlp))

print(classification\_report(y\_test, y\_pred\_mlp, target\_names=iris.target\_names))

1. **Implement the partition-based clustering algorithm**

# ==============================

# Partition-Based Clustering (K-Means)

# ==============================

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

from sklearn.cluster import KMeans

from sklearn.decomposition import PCA

# -------------------------------

# Load Iris Dataset

# -------------------------------

iris = load\_iris()

X = pd.DataFrame(iris.data, columns=iris.feature\_names)

# -------------------------------

# Apply K-Means Clustering

# -------------------------------

k = 3 # since we know Iris has 3 classes

kmeans = KMeans(n\_clusters=k, random\_state=42, n\_init=10)

kmeans.fit(X)

# Cluster labels

labels = kmeans.labels\_

centroids = kmeans.cluster\_centers\_

print("Cluster Labels (first 20):", labels[:20])

print("\nCluster Centroids:\n", centroids)

# -------------------------------

# Visualize Clusters (using PCA for 2D projection)

# -------------------------------

pca = PCA(n\_components=2)

X\_pca = pca.fit\_transform(X)

plt.figure(figsize=(8,6))

plt.scatter(X\_pca[:,0], X\_pca[:,1], c=labels, cmap='viridis', marker='o', edgecolor='k')

plt.scatter(pca.transform(centroids)[:,0], pca.transform(centroids)[:,1],

c='red', marker='X', s=200, label='Centroids')

plt.title("Partition-Based Clustering (K-Means) on Iris Data")

plt.xlabel("PCA Component 1")

plt.ylabel("PCA Component 2")

plt.legend()

plt.show()