import pandas as pd

import numpy as np

import statistics

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.linear\_model import LogisticRegression

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

from sklearn.preprocessing import LabelEncoder, MinMaxScaler

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

from sklearn.metrics.pairwise import cosine\_similarity

#question1

def solve\_A1(file):

# Load purchase data

df = pd.read\_excel(file, sheet\_name="Purchase data")

# Drop non-numeric columns before creating matrix A

numeric\_df = df.select\_dtypes(include=np.number)

# Drop columns with all NaN values

numeric\_df = numeric\_df.dropna(axis=1, how='all')

# Impute remaining missing values with the mean

numeric\_df = numeric\_df.fillna(numeric\_df.mean())

# Create matrix A (features) and C (payments)

A = numeric\_df.drop(columns=["Payment (Rs)"]).values

C = numeric\_df[["Payment (Rs)"]].values

# Dimensionality = number of features

dimensionality = A.shape[1]

# Number of vectors = number of data rows

num\_vectors = A.shape[0]

# Rank of matrix A

rank\_A = np.linalg.matrix\_rank(A)

# Estimate cost vector using pseudo-inverse

X = np.dot(np.linalg.pinv(A), C)

return {

"dimensionality": dimensionality,

"num\_vectors": num\_vectors,

"rank": rank\_A,

"product\_costs": X.flatten()

}

#question2

def solve\_A2(file):

# Load the data

df = pd.read\_excel(file, sheet\_name="Purchase data")

# Create binary labels based on payment amount

df["Label"] = df["Payment (Rs)"].apply(lambda x: "RICH" if x > 200 else "POOR")

# Drop 'Unnamed' columns as they likely contain non-numeric or empty data

df = df.loc[:, ~df.columns.str.contains('^Unnamed')]

# Impute missing values (column-wise)

for col in df.columns:

if df[col].isnull().sum() > 0:

if df[col].dtype in [np.float64, np.int64]:

df[col] = df[col].fillna(df[col].mean()) # Mean for numeric

else:

df[col] = df[col].fillna(df[col].mode()[0]) # Mode for categorical

# Select only numeric columns except target column

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

if "Payment (Rs)" in numeric\_cols:

numeric\_cols.remove("Payment (Rs)") # Remove the target

X = df[numeric\_cols]

y = LabelEncoder().fit\_transform(df["Label"])

# Check for any remaining NaNs in X

if X.isnull().sum().sum() > 0:

print("ERROR: X still contains NaN values after imputation.")

print("NaN counts per column:\n", X.isnull().sum())

raise ValueError("X still contains NaN values after cleaning.")

# Train-test split

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

# Train classifier

model = LogisticRegression()

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

# Predict & Evaluate

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

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

report = classification\_report(y\_test, y\_pred, output\_dict=True)

return {

"accuracy": accuracy,

"report": report,

"labeled\_data": df # Return the dataframe with the label column

}

#question3

def solve\_A3(file):

# Load stock price data

df = pd.read\_excel(file, sheet\_name="IRCTC Stock Price")

df["Date"] = pd.to\_datetime(df["Date"])

df["Day"] = df["Date"].dt.day\_name()

# Compute population mean and variance of Price

price\_col = df["Price"]

mean = statistics.mean(price\_col)

var = statistics.variance(price\_col)

# Sample mean for Wednesdays

wed\_df = df[df["Day"] == "Wednesday"]

mean\_wed = wed\_df["Price"].mean()

# Sample mean for April

april\_df = df[df["Date"].dt.month == 4]

mean\_april = april\_df["Price"].mean()

# Probability of making a loss (Chg% < 0)

chg = df["Chg%"]

prob\_loss = (chg < 0).mean()

# Probability of making a profit on Wednesday

prob\_profit\_wed = (wed\_df["Chg%"] > 0).mean()

# Conditional probability: P(profit | Wednesday)

cond\_prob\_profit\_given\_wed = (wed\_df["Chg%"] > 0).sum() / (chg > 0).sum()

# Scatter plot of Chg% by Day

plt.figure(figsize=(8, 5))

sns.scatterplot(x="Day", y="Chg%", data=df)

plt.title("Chg% vs Day")

plt.xticks(rotation=45)

plt.tight\_layout()

plt.show()

return {

"mean": mean,

"variance": var,

"mean\_wednesday": mean\_wed,

"mean\_april": mean\_april,

"prob\_loss": prob\_loss,

"prob\_profit\_wed": prob\_profit\_wed,

"cond\_prob\_profit\_given\_wed": cond\_prob\_profit\_given\_wed

}

#question4

def solve\_A4(file):

# Load thyroid data

df = pd.read\_excel(file, sheet\_name="thyroid0387\_UCI")

# Attribute data types

types = df.dtypes

# Missing values per column

missing = df.isnull().sum()

# Numeric column range and statistics

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

ranges = numeric.describe().loc[["min", "max"]]

mean\_std = numeric.agg(["mean", "std"])

# Detect outliers using IQR method

Q1 = numeric.quantile(0.25)

Q3 = numeric.quantile(0.75)

IQR = Q3 - Q1

outliers = ((numeric < (Q1 - 1.5 \* IQR)) | (numeric > (Q3 + 1.5 \* IQR))).sum()

return {

"types": types,

"missing": missing,

"range": ranges,

"outliers": outliers,

"mean\_std": mean\_std

}

#question5

def solve\_A5(file):

# Load thyroid data

df = pd.read\_excel(file, sheet\_name="thyroid0387\_UCI")

# Select the first two rows

v1\_df = df.iloc[[0]]

v2\_df = df.iloc[[1]]

# Identify binary columns (containing only 0s and 1s, ignoring NaNs)

binary\_cols = []

for col in df.columns:

# Try converting to numeric, coercing errors

numeric\_col = pd.to\_numeric(df[col], errors='coerce')

# Drop NaNs for checking

cleaned\_col = numeric\_col.dropna()

# Check if the set of unique values (excluding NaNs) is a subset of {0.0, 1.0}

# Ensure we handle empty cleaned\_col case where unique() would be empty

unique\_values = cleaned\_col.unique()

if len(unique\_values) > 0 and set(unique\_values).issubset({0.0, 1.0}):

binary\_cols.append(col)

if not binary\_cols:

return {"error": "No binary columns found in the first two rows."}

# Select only binary columns for the first two vectors

v1 = v1\_df[binary\_cols].iloc[0]

v2 = v2\_df[binary\_cols].iloc[0]

# Impute any remaining NaNs in the selected binary vectors with 0 or mode (if applicable, though ideally binary should be clean)

# Given the nature of binary attributes, filling with 0 is a reasonable approach if some binary values were missing.

v1 = v1.fillna(0)

v2 = v2.fillna(0)

# Compute binary similarity values

# Ensure vectors are treated as binary (0 or 1)

v1\_binary = (v1 > 0).astype(int)

v2\_binary = (v2 > 0).astype(int)

f11 = np.sum((v1\_binary == 1) & (v2\_binary == 1))

f00 = np.sum((v1\_binary == 0) & (v2\_binary == 0))

f10 = np.sum((v1\_binary == 1) & (v2\_binary == 0))

f01 = np.sum((v1\_binary == 0) & (v2\_binary == 1))

# Calculate JC and SMC

# Add a small epsilon to denominator to avoid division by zero if all counts are zero

jc\_denominator = (f01 + f10 + f11)

jc = f11 / (jc\_denominator + 1e-10) if jc\_denominator > 0 else 0.0

smc\_denominator = (f11 + f00 + f01 + f10)

smc = (f11 + f00) / (smc\_denominator + 1e-10) if smc\_denominator > 0 else 0.0

# Comparison and judgment

comparison = "Jaccard Coefficient ignores mutual absences (0-0 matches), focusing only on mutual presences (1-1 matches) relative to the total number of attributes that are present in at least one of the vectors (1-1, 1-0, 0-1 matches). It is suitable when the absence of an attribute is not informative. Simple Matching Coefficient considers both mutual presences (1-1) and mutual absences (0-0) in the calculation relative to the total number of attributes. It is appropriate when 0-0 matches are as informative as 1-1 matches."

judgment = ""

if jc > smc:

judgment = "Jaccard Coefficient is higher than Simple Matching Coefficient. This suggests that there are more 1-1 matches relative to the number of attributes present in at least one vector, compared to the overall agreement (including 0-0 matches)."

elif smc > jc:

judgment = "Simple Matching Coefficient is higher than Jaccard Coefficient. This indicates that there are a significant number of 0-0 matches contributing to the overall similarity, which are ignored by the Jaccard Coefficient."

else:

judgment = "Jaccard Coefficient and Simple Matching Coefficient are equal. This happens when either there are no 0-0 matches or no 1-1 matches, or when the proportions align in a specific way."

return {

"JC": jc,

"SMC": smc,

"comparison": comparison,

"judgment": judgment,

"binary\_cols\_used": binary\_cols

}

#question6

def solve\_A6(file):

# Load thyroid data and fill NaNs

df = pd.read\_excel(file, sheet\_name="thyroid0387\_UCI").fillna(0)

# Identify and convert non-numeric columns that should be numeric

for col in df.columns:

if df[col].dtype == 'object':

try:

df[col] = pd.to\_numeric(df[col])

except ValueError:

# If conversion to numeric fails, drop the column

df = df.drop(columns=[col])

# Cosine similarity between first 2 rows

v1 = df.iloc[0].values.reshape(1, -1)

v2 = df.iloc[1].values.reshape(1, -1)

cos = cosine\_similarity(v1, v2)[0][0]

return {"Cosine Similarity": cos}

#question7

def solve\_A7(file):

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.metrics.pairwise import cosine\_similarity

# Load and preprocess data

df = pd.read\_excel(file, sheet\_name="thyroid0387\_UCI")

df.replace('?', np.nan, inplace=True)

df = df.fillna(0)

df.columns = df.columns.str.strip()

# Select first 20 observations

data = df.iloc[:20].copy() # Create a copy to avoid SettingWithCopyWarning

# Identify truly numeric columns after handling '?' and filling NaNs

numeric\_cols = data.select\_dtypes(include=np.number).columns.tolist()

# Further filter numeric columns to exclude those that might still contain non-numeric strings

# This is a more robust check than just select\_dtypes after fillna(0)

final\_numeric\_cols = []

for col in numeric\_cols:

# Attempt to convert the entire column to numeric, coercing errors

if pd.to\_numeric(data[col], errors='coerce').notna().all():

final\_numeric\_cols.append(col)

# Select only the final determined numeric columns

data\_numeric = data[final\_numeric\_cols]

n = data\_numeric.shape[0]

num\_features = data\_numeric.shape[1]

# Check if there are enough numeric features to proceed

if num\_features == 0:

print("No purely numeric columns found after cleaning for similarity calculations.")

return {"JC": np.zeros((n, n)), "SMC": np.zeros((n, n)), "COS": np.zeros((n, n))}

# Initialize similarity matrices

jc\_matrix = np.zeros((n, n))

smc\_matrix = np.zeros((n, n))

cos\_matrix = np.zeros((n, n))

# Calculate similarity for each pair (i, j)

for i in range(n):

for j in range(n):

v1 = data\_numeric.iloc[i]

v2 = data\_numeric.iloc[j]

# Convert to binary representation for JC and SMC (assuming 0/non-zero binary)

v1\_binary = (v1 > 0).astype(int)

v2\_binary = (v2 > 0).astype(int)

# Binary similarity: JC and SMC

f11 = np.sum((v1\_binary == 1) & (v2\_binary == 1))

f00 = np.sum((v1\_binary == 0) & (v2\_binary == 0))

f10 = np.sum((v1\_binary == 1) & (v2\_binary == 0))

f01 = np.sum((v1\_binary == 0) & (v2\_binary == 1))

# JC and SMC with epsilon to avoid divide-by-zero

jc\_denominator = (f01 + f10 + f11)

jc\_matrix[i][j] = f11 / (jc\_denominator + 1e-10) if jc\_denominator > 0 else 0.0

smc\_denominator = (f11 + f00 + f01 + f10)

smc\_matrix[i][j] = (f11 + f00) / (smc\_denominator + 1e-10) if smc\_denominator > 0 else 0.0

# Cosine similarity requires numeric values

cos\_matrix[i][j] = cosine\_similarity([v1.values], [v2.values])[0][0]

# Plotting

plt.figure(figsize=(10, 6))

sns.heatmap(jc\_matrix, annot=False, cmap="Blues")

plt.title("Jaccard Coefficient Heatmap (Numeric Columns)")

plt.show()

plt.figure(figsize=(10, 6))

sns.heatmap(smc\_matrix, annot=False, cmap="Greens")

plt.title("Simple Matching Coefficient Heatmap (Numeric Columns)")

plt.show()

plt.figure(figsize=(10, 6))

sns.heatmap(cos\_matrix, annot=False, cmap="Reds")

plt.title("Cosine Similarity Heatmap (Numeric Columns)")

plt.show()

# Return matrices in case you want to export or validate

return {

"JC": jc\_matrix,

"SMC": smc\_matrix,

"COS": cos\_matrix

}

#question8

def solve\_A8(file, sheet\_name="thyroid0387\_UCI"):

# Load data from the specified sheet

df = pd.read\_excel(file, sheet\_name=sheet\_name)

# Replace '?' with NaN to be recognized as missing values

df.replace('?', np.nan, inplace=True)

# Identify numeric columns after replacing '?'

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

# Detect outliers in numeric columns (using IQR)

outlier\_counts = {}

for col in numeric\_cols:

Q1 = df[col].quantile(0.25)

Q3 = df[col].quantile(0.75)

IQR = Q3 - Q1

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

# Ensure we only consider non-NaN values for outlier detection

col\_series = df[col].dropna()

outliers = col\_series[(col\_series < lower\_bound) | (col\_series > upper\_bound)]

outlier\_counts[col] = outliers.shape[0]

# Impute missing values based on variable type and outliers

for col in df.columns:

if df[col].isnull().sum() > 0:

if col in numeric\_cols: # Check if the column is in the identified numeric columns

# Check if the numeric column has outliers

if outlier\_counts.get(col, 0) > 0: # Use .get to handle cases where a numeric col might not be in outlier\_counts

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

print(f"Imputed missing values in numeric column '{col}' with median.")

else:

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

print(f"Imputed missing values in numeric column '{col}' with mean.")

elif df[col].dtype == 'object':

# Use mode for object type columns, handling potential empty mode

mode\_val = df[col].mode()

if not mode\_val.empty:

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

print(f"Imputed missing values in categorical column '{col}' with mode.")

else:

# If mode is empty (e.g., all NaNs), fill with a placeholder string

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

print(f"Imputed missing values in categorical column '{col}' with 'Unknown' (mode was empty).")

return df

#question9

def solve\_A9(file):

# Load thyroid data from the specified sheet

df = pd.read\_excel(file, sheet\_name="thyroid0387\_UCI")

# Replace '?' with NaN to be recognized as missing values

df.replace('?', np.nan, inplace=True)

# Identify and convert columns that should be numeric

# This step attempts to convert object type columns to numeric, coercing errors to NaN

for col in df.columns:

if df[col].dtype == 'object':

try:

df[col] = pd.to\_numeric(df[col])

except ValueError:

# If conversion to numeric fails, it's not a numeric column, so skip it

pass

# Select only numeric columns for normalization after conversion

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

# Exclude 'Record ID' and 'age' from the list of numeric columns to normalize

if 'Record ID' in numeric\_cols:

numeric\_cols.remove('Record ID')

if 'age' in numeric\_cols:

numeric\_cols.remove('age')

# Check if there are any numeric columns to normalize after exclusion

if numeric\_cols:

# Initialize the MinMaxScaler

scaler = MinMaxScaler()

# Apply MinMax scaling to the selected numeric columns

# Before scaling, impute any remaining NaNs in numeric columns with the mean

# This is necessary because MinMaxScaler does not handle NaN values

df[numeric\_cols] = df[numeric\_cols].fillna(df[numeric\_cols].mean())

df[numeric\_cols] = scaler.fit\_transform(df[numeric\_cols])

print(f"Normalized {len(numeric\_cols)} numeric columns.")

else:

# Print a warning if no numeric columns are found for normalization

print("Warning: No numeric columns found for normalization in solve\_A9 after excluding 'Record ID' and 'age'.")

# Return the DataFrame with normalized numeric columns

return df

def main():

# Path to your Excel file

file = "Lab Session Data.xlsx"

# A1 - Matrix and product cost analysis

print("\n--- A1: Matrix Dimensionality & Product Costs ---")

a1\_result = solve\_A1(file)

print(f"Dimensionality: {a1\_result['dimensionality']}")

print(f"Number of Vectors: {a1\_result['num\_vectors']}")

print(f"Rank of Matrix A: {a1\_result['rank']}")

print(f"Estimated Product Costs: {a1\_result['product\_costs']}")

# A2 - Customer classification

print("\n--- A2: Classification (RICH / POOR) ---")

a2\_result = solve\_A2(file)

print(f"Accuracy: {a2\_result['accuracy']:.4f}")

print("Classification Report:")

for label, metrics in a2\_result['report'].items():

# Explicitly check for keys that should have dictionary values

if label in ['0', '1', 'macro avg', 'weighted avg']:

print(f" {label}:")

for metric, value in metrics.items():

print(f" {metric}: {value:.4f}")

elif label == 'accuracy': # Handle the overall accuracy key (float value)

print(f" {label}: {metrics:.4f}")

print("--- Displaying Labeled Data for A2 ---")

display(a2\_result['labeled\_data'])

print("--- A2 Labeled Data Displayed ---")

# A3 - Stock price statistics

print("\n--- A3: Stock Data Analysis ---")

a3\_result = solve\_A3(file)

print(f"Population Mean of Price: {a3\_result['mean']:.4f}")

print(f"Population Variance of Price: {a3\_result['variance']:.4f}")

print(f"Sample Mean for Wednesdays: {a3\_result['mean\_wednesday']:.4f}")

print(f"Sample Mean for April: {a3\_result['mean\_april']:.4f}")

print(f"Probability of Making a Loss (Chg% < 0): {a3\_result['prob\_loss']:.4f}")

print(f"Probability of Making a Profit on Wednesday: {a3\_result['prob\_profit\_wed']:.4f}")

print(f"Conditional Probability P(profit | Wednesday): {a3\_result['cond\_prob\_profit\_given\_wed']:.4f}")

# A4 - Thyroid data exploration

print("\n--- A4: Thyroid Data Summary ---")

a4\_result = solve\_A4(file)

print("Data Types:\n", a4\_result["types"])

print("\nMissing Values:\n", a4\_result["missing"])

print("\nRanges:\n", a4\_result["range"])

print("\nOutliers (IQR Method):\n", a4\_result["outliers"])

print("\nMean & Std Dev:\n", a4\_result["mean\_std"])

# A5 - Jaccard and SMC

print("\n--- A5: Binary Similarity Measures ---")

a5\_result = solve\_A5(file)

if "error" in a5\_result:

print(a5\_result["error"])

else:

print(f"Jaccard Coefficient (first 2 rows): {a5\_result['JC']:.4f}")

print(f"Simple Matching Coefficient (first 2 rows): {a5\_result['SMC']:.4f}")

print("\nComparison of JC and SMC:")

print(a5\_result["comparison"])

print("\nJudgment on Appropriateness:")

print(a5\_result["judgment"])

print("\nBinary columns used for calculation:", a5\_result["binary\_cols\_used"])

# A6 - Cosine similarity

print("\n--- A6: Cosine Similarity ---")

a6\_result = solve\_A6(file)

print(f"Cosine Similarity (first 2 rows): {a6\_result['Cosine Similarity']:.4f}")

# A7 - Heatmap Similarity Matrix

print("\n--- A7: Similarity Heatmaps ---")

a7\_result = solve\_A7(file)

print("Jaccard Coefficient Heatmap plotted.")

print("Simple Matching Coefficient Heatmap plotted.")

print("Cosine Similarity Heatmap plotted.")

# A8 - Imputed Data

print("\n--- A8: Data Imputation Completed ---")

imputed\_df = solve\_A8(file)

print("Missing values filled using mean (numeric) or mode (categorical).")

# Save the imputed data to a CSV file

imputed\_df.to\_csv("imputed\_thyroid\_data.csv", index=False)

print("Imputed data saved to 'imputed\_thyroid\_data.csv'")

# Call the updated solve\_A9 function to get the normalized DataFrame

normalized\_df = solve\_A9(file)

# Save the normalized DataFrame to a CSV file

normalized\_df.to\_csv("normalized\_thyroid\_data.csv", index=False)

print("Normalized data saved to 'normalized\_thyroid\_data.csv'")

# to confirm they were not normalized

print("\n--- Checking 'Record ID' and 'age' after normalization ---")

normalized\_df\_check = pd.read\_csv("normalized\_thyroid\_data.csv")

display(normalized\_df\_check.head())

print("\nMin and Max values for 'Record ID' and 'age' columns:")

print(normalized\_df\_check[['Record ID', 'age']].agg(['min', 'max']))

main()