Department of Computer Science
Session 2025-26

Data Mining Practical File

Supervisor

Gagandeep Sir

Name: shanik
Course: **B.Sc (H) Computer Science**
College Roll No: **CSC/24/42**
University Roll No: **2405957004**8

INDEX

S.NO.	PARTICULARS	PAGE NO.
1.	Data Cleaning on Dataset (e.g. Wine): Handle Missing Values, Outliers, Inconsistencies; Prepare & Apply Validation Rules.	3
2.	Data Pre-processing on Dataset: Standardization/Normalization, Transformation, Aggregation, Discretization/Binarization, Sampling, etc.	9
3.	Apriori Algorithm on 2 Real Datasets: Frequent Itemsets & Association Rules; Evaluate Pattern Correctness. a) Min. Support 50%, Confidence 75% b) Min. Support 60%, Confidence 60%	13
4.	Naive Bays, KNN, Decision Tree Classifiers on 2 Datasets, Compare Accuracies, Scale Data	21
5.	Perform K-means Clustering on Dataset: Vary Parameters, Compare Cluster Performance; Plot MSE per Iteration	28
6.	Perform Density Based Clustering on a Dataset and Evaluate Cluster Quality by changing the algorithm's pattern	35

GOOGLE COLAB LINK AS REFERENCE FOR ALL PRACTICALS :

https://colab.research.google.com/drive/13SbhU42nGGmxDw0pPjrMID_NN4N0TS8c?usp=sharing

PRACTICAL - 1

Apply data cleaning techniques on any dataset (e,g, wine dataset). Techniques may include handling missing values, outliers, inconsistent values. A set of validation rules can be prepared based on the dataset and validations can be performed.

INPUT :

```
# QUES - 1 STARTS
import pandas as pd
import numpy as np

df = pd.read_csv('winequality-red.csv', sep=';') # Use
of Semicolon Explain

print("--- After Loading ---")
print(df.info())
print(df.head())
print(df.columns)
```

OUTPUT :

```
--- After Corrected Loading ---
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   fixed acidity         1599 non-null   float64
 1   volatile acidity      1599 non-null   float64
 2   citric acid           1599 non-null   float64
 3   residual sugar        1599 non-null   float64
 4   chlorides             1599 non-null   float64
 5   free sulfur dioxide   1599 non-null   float64
 6   total sulfur dioxide  1599 non-null   float64
 7   density               1599 non-null   float64
 8   pH                    1599 non-null   float64
 9   sulphates             1599 non-null   float64
 10  alcohol               1599 non-null   float64
 11  quality               1599 non-null   int64
dtypes: float64(11), int64(1)
memory usage: 150.0 KB
None
```

```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.4              0.70         0.00             1.9      0.076
1            7.8              0.88         0.00             2.6      0.098
2            7.8              0.76         0.04             2.3      0.092
3           11.2              0.28         0.56             1.9      0.075
4            7.4              0.70         0.00             1.9      0.076

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 11.0                  34.0   0.9978  3.51       0.56
1                 25.0                  67.0   0.9968  3.20       0.68
2                 15.0                  54.0   0.9970  3.26       0.65
3                 17.0                  60.0   0.9980  3.16       0.58
4                 11.0                  34.0   0.9978  3.51       0.56

   alcohol  quality
0      9.4        5
1      9.8        5
2      9.8        5
3      9.8        6
4      9.4        5
Index(['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
       'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
       'pH', 'sulphates', 'alcohol', 'quality'],
      dtype='object')
```

INPUT :

```
# Check for null values
print("\n--- Missing Values Check ---")
print(df.isnull().sum())
```

OUTPUT :

```
--- Missing Values Check ---
fixed acidity           0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
quality                 0
dtype: int64
```

INPUT :

```
# --- Outlier Handling ---
col_name = 'sulphates'

Q1 = df[col_name].quantile(0.25)
Q3 = df[col_name].quantile(0.75)
IQR = Q3 - Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

print(f"\n--- Outlier Bounds for {col_name} ---")
print(f"Lower Bound: {lower_bound}, Upper Bound:
{upper_bound}")

# Handle the Outliers (Capping)
df[col_name] = np.where(df[col_name] < lower_bound,
lower_bound, df[col_name])
df[col_name] = np.where(df[col_name] > upper_bound,
upper_bound, df[col_name])

print("\nOutliers in 'sulphates' have been capped.")
print(df[col_name].describe())
```

OUTPUT :

```
--- Outlier Bounds for sulphates ---
Lower Bound: 0.28000000000000014, Upper Bound: 0.9999999999999999

Outliers in 'sulphates' have been capped.
count    1599.000000
mean        0.649831
std         0.137086
min         0.330000
25%         0.550000
50%         0.620000
75%         0.730000
max         1.000000
Name: sulphates, dtype: float64
```

INPUT :

```
# Check for values below the minimum expected range
(e.g., alcohol < 5)
invalid_alcohol = df[df['alcohol'] < 5]
print("\n--- Inconsistent/Invalid Alcohol Values (< 5)
---")
print(f"Number of invalid entries:
{len(invalid_alcohol)}")
```

OUTPUT :

```
--- Inconsistent/Invalid Alcohol Values (< 5) ---
Number of invalid entries: 0
```

INPUT :

```
# Check for quality values outside the 0-10 range
invalid_quality = df[(df['quality'] < 0) |
(df['quality'] > 10)]
print("\n--- Inconsistent/Invalid Quality Values (not
0-10) ---")
print(f"Number of invalid entries:
{len(invalid_quality)}")

# QUES- 1 - ENDS
```

OUTPUT :

```
--- Inconsistent/Invalid Quality Values (not 0-10) ---
Number of invalid entries: 0
```

PRACTICAL - 2

Apply data pre-processing techniques such as standardization/normalization, transformation, aggregation, discretization/binarization, sampling etc. on any dataset

INPUT :

```
# QUES - 2 - STARTS

from sklearn.model_selection import train_test_split

# X: Features (all columns except 'quality')
X = df.drop('quality', axis=1)

# y: Target variable ('quality')
y = df['quality']

print("Features (X) shape:", X.shape)
print("Target (y) shape:", y.shape)
```

OUTPUT :

```
Features (X) shape: (1599, 11)
Target (y) shape: (1599,)
```

INPUT :

```
from sklearn.preprocessing import StandardScaler

# Initialize the StandardScaler
scaler = StandardScaler()

# Fit the scaler to the features and transform them
# We create a new DataFrame for the scaled data
X_scaled_array = scaler.fit_transform(X)
X_scaled = pd.DataFrame(X_scaled_array,
columns=X.columns)

print("\n--- Standardized Data Sample (Mean ~0, Std
Dev ~1) ---")
print(X_scaled.head())
print(X_scaled.describe().loc[['mean', 'std']])
```

OUTPUT :

```
--- Standardized Data Sample (Mean ~0, Std Dev ~1) ---
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0      -0.528360          0.961877    -1.391472       -0.453218  -0.243707
1      -0.298547          1.967442    -1.391472        0.043416   0.223875
2      -0.298547          1.297065    -1.186070       -0.169427   0.096353
3       1.654856         -1.384443     1.484154       -0.453218  -0.264960
4      -0.528360          0.961877    -1.391472       -0.453218  -0.243707

   free sulfur dioxide  total sulfur dioxide   density        pH  sulphates  \
0            -0.466193             -0.379133  0.558274  1.288643  -0.579207
1             0.872638              0.624363  0.028261 -0.719933   0.128950
2            -0.083669              0.229047  0.134264 -0.331177  -0.048089
3             0.107592              0.411500  0.664277 -0.979104  -0.461180
4            -0.466193             -0.379133  0.558274  1.288643  -0.579207
```

```
    alcohol
0 -0.960246
1 -0.584777
2 -0.584777
3 -0.584777
4 -0.960246
      fixed acidity  volatile acidity   citric acid  residual sugar  \
mean   3.554936e-16      1.733031e-16 -8.887339e-17   -1.244227e-16
std    1.000313e+00      1.000313e+00  1.000313e+00    1.000313e+00

         chlorides  free sulfur dioxide  total sulfur dioxide       density  \
mean  3.732682e-16        -6.221137e-17          4.443669e-17 -3.473172e-14
std   1.000313e+00         1.000313e+00          1.000313e+00  1.000313e+00

                pH     sulphates       alcohol
mean  2.861723e-15  6.754377e-16  1.066481e-16
std   1.000313e+00  1.000313e+00  1.000313e+00
```

INPUT :

```
Create a new binary target column 'is_good'
# 1 if quality is 7 or higher, 0 otherwise.
y_binary = (y >= 7).astype(int)
print("\n--- Binarized Target Variable (is_good) ---")
print("Original Quality Counts:\n",
y.value_counts().sort_index())
print("\nBinarized Quality Counts (0=Poor,
1=Good):\n", y_binary.value_counts())
```

OUTPUT :

```
--- Binarized Target Variable (is_good) ---
Original Quality Counts:
 quality
3     10
4     53
5    681
6    638
7    199
8     18
Name: count, dtype: int64

Binarized Quality Counts (0=Poor, 1=Good):
 quality
0    1382
1     217
Name: count, dtype: int64
```

PRACTICAL - 3

Run Apriori algorithm to find frequent item sets and association rules on 2 real datasets and use appropriate evaluation measures to compute correctness of obtained patterns :

a) Use minimum support as 50% and minimum confidence as 75%

b) Use minimum support as 60% and minimum confidence as 60 %

INPUT :

```
# Q - 3 STARTS
# PART - A
!pip install apyori
import numpy as np
from apyori import apriori

# Use the original features (X) before scaling, as
binarizing scaled data is less intuitive
X_apriori = X.copy()

# Binarize all numerical features in X based on their
median
for col in X_apriori.columns:
    median_val = X_apriori[col].median()
    X_apriori[col] = np.where(X_apriori[col] >
median_val, col + '_High', col + '_Low')

print("--- Sample of Binarized Features for Apriori
---")
print(X_apriori.head())
```

OUTPUT :

```
Successfully built apyori
Installing collected packages: apyori
Successfully installed apyori-1.1.2
--- Sample of Binarized Features for Apriori ---
        fixed acidity       volatile acidity       citric acid  \
0   fixed acidity_Low  volatile acidity_High   citric acid_Low
1   fixed acidity_Low  volatile acidity_High   citric acid_Low
2   fixed acidity_Low  volatile acidity_High   citric acid_Low
3  fixed acidity_High   volatile acidity_Low  citric acid_High
4   fixed acidity_Low  volatile acidity_High   citric acid_Low

         residual sugar       chlorides       free sulfur dioxide  \
0   residual sugar_Low   chlorides_Low   free sulfur dioxide_Low
1  residual sugar_High  chlorides_High  free sulfur dioxide_High
2  residual sugar_High  chlorides_High  free sulfur dioxide_High
3   residual sugar_Low   chlorides_Low  free sulfur dioxide_High
4   residual sugar_Low   chlorides_Low   free sulfur dioxide_Low

         total sulfur dioxide       density       pH       sulphates  \
0   total sulfur dioxide_Low  density_High  pH_High   sulphates_Low
1  total sulfur dioxide_High  density_High   pH_Low  sulphates_High
2  total sulfur dioxide_High  density_High   pH_Low  sulphates_High
3  total sulfur dioxide_High  density_High   pH_Low   sulphates_Low
4   total sulfur dioxide_Low  density_High  pH_High   sulphates_Low

       alcohol
0  alcohol_Low
1  alcohol_Low
2  alcohol_Low
3  alcohol_Low
4  alcohol_Low
```

INPUT :

```
# Convert the DataFrame rows into a list of lists

transactions = X_apriori.values.tolist()
```

```
print(f"\nTotal transactions: {len(transactions)}")
print(f"Sample transaction: {transactions[0]} ", "\n")
```

OUTPUT :

```
Total transactions: 1599
Sample transaction: ['fixed acidity_Low', 'volatile acidity_High', 'citric acid_Low', 'residual sugar_Low',

'chlorides_Low', 'free sulfur dioxide_Low', 'total sulfur dioxide_Low', 'density_High',

'pH_High', 'sulphates_Low', 'alcohol_Low']
```

INPUT :

```
# Set parameters for 3a
min_support_3a = 0.50
min_confidence_3a = 0.75

# Run the Apriori algorithm
rules_3a = apriori(
    transactions,
    min_support=min_support_3a,
    min_confidence=min_confidence_3a,
    min_lift=1.0, # Lift should be > 1.0 for
meaningful associations
    min_length=2 # Find rules with at least 2 items
)

# Convert results into a readable Pandas DataFrame
def inspect_rules(results):
    rhs = [tuple(result[2][0][0]) for result in
results]
```

```
    lhs = [tuple(result[2][0][1]) for result in
results]
    supports = [result[1] for result in results]
    confidences = [result[2][0][2] for result in
results]
    lifts = [result[2][0][3] for result in results]

    return pd.DataFrame({
        'Rule_Antecedent': lhs,
        'Rule_Consequent': rhs,
        'Support': supports,
        'Confidence': confidences,
        'Lift': lifts
    })

results_3a = list(rules_3a)
rules_df_3a = inspect_rules(results_3a)

print(f"\n--- Apriori Results (Dataset 1: Wine
Features) ---")
print(f"Min Support: {min_support_3a}, Min Confidence:
{min_confidence_3a}")
if rules_df_3a.empty:
    print("No rules found with these strict
parameters.")
else:
    print(rules_df_3a.sort_values(by='Confidence',
ascending=False).head())
```

OUTPUT :

```
--- Apriori Results (Dataset 1: Wine Features) ---
Min Support: 0.5, Min Confidence: 0.75
No rules found with these strict parameters.
```

INPUT :

```
# PART - B

# Download and load a transactional dataset
df_mba = pd.read_csv('Groceries_dataset.csv',
header=None)

# Using a standard MBA example:
transactions_mba = df_mba.values.tolist()
# A simplified way using a fixed list for
demonstration:
transactions_mba = [
    ['Milk', 'Bread', 'Butter'],
    ['Milk', 'Diapers', 'Beer'],
    ['Bread', 'Butter', 'Cheese'],
    ['Milk', 'Bread', 'Cheese'],
    ['Bread', 'Butter', 'Diapers']
]
print(f"\n--- Dataset 2: Standard MBA (Sample
Transactions) ---")
print(transactions_mba)
```

OUTPUT :

```
--- Dataset 2: Standard MBA (Sample Transactions) ---
[['Milk', 'Bread', 'Butter'], ['Milk', 'Diapers', 'Beer'], ['Bread', 'Butter', 'Cheese'],
```

```
['Milk', 'Bread', 'Cheese'], ['Bread', 'Butter', 'Diapers']]
```

INPUT :

```
# Set parameters for 3b
min_support_3b = 0.60
min_confidence_3b = 0.60

# Run the Apriori algorithm on the MBA transactions
rules_3b = apriori(
    transactions_mba,
    min_support=min_support_3b,
    min_confidence=min_confidence_3b,
    min_lift=1.0,
    min_length=2
)

results_3b = list(rules_3b)
rules_df_3b = inspect_rules(results_3b)

print(f"\n--- Apriori Results (Dataset 2: Standard
MBA) ---")
print(f"Min Support: {min_support_3b}, Min Confidence:
{min_confidence_3b}")

if rules_df_3b.empty:
    print("No rules found with these parameters.")
else:
```

```
    print(rules_df_3b.sort_values(by=['Lift',
'Confidence'], ascending=False))

    # Q - 3 ends
```

OUTPUT :

```
--- Apriori Results (Dataset 2: Standard MBA) ---
Min Support: 0.6, Min Confidence: 0.6
   Rule_Antecedent Rule_Consequent  Support  Confidence  Lift
0         (Bread,)              ()      0.8         0.8   1.0
1        (Butter,)              ()      0.6         0.6   1.0
2          (Milk,)              ()      0.6         0.6   1.0
3  (Butter, Bread)              ()      0.6         0.6   1.0
```

PRACTICAL - 4

Use Naive bayes, K-nearest, and Decision tree classification algorithms and build classifiers on any two datasets. Divide the data set into training and test set. Compare the accuracy of the different classifiers under the following situations:

I.

a) Training set = 75% Test set = 25%

b) Training set = 66.6% (2/3rd of total), Test set = 33.3%

II.

Training set is chosen by :

i) hold out method

ii) Random subsampling

iii) Cross-Validation. Compare the accuracy of the classifiers obtained. Data needs to be scaled to standard format.

INPUT :

```
# QUES - 4 STARTS

from sklearn.model_selection import train_test_split,
KFold, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
import pandas as pd
import numpy as np

# Assuming X_scaled (Standardized Features) and
y_binary (Binarized Target)
# are available from previous steps. If you reset the
notebook, run Step 1 & 2 of Q2.
# X_scaled = ...
# y_binary = …

def evaluate_classifiers(X_train, X_test, y_train,
y_test):
    # 1. Naive Bayes (Gaussian)
    nb_model = GaussianNB()
    nb_model.fit(X_train, y_train)
    nb_pred = nb_model.predict(X_test)
    nb_accuracy = accuracy_score(y_test, nb_pred)
```

```
    # 2. K-Nearest Neighbors (KNN, using k=5 as
default)
    knn_model = KNeighborsClassifier(n_neighbors=5)
    knn_model.fit(X_train, y_train)
    knn_pred = knn_model.predict(X_test)
    knn_accuracy = accuracy_score(y_test, knn_pred)

    # 3. Decision Tree
    dt_model = DecisionTreeClassifier(random_state=42)
    dt_model.fit(X_train, y_train)
    dt_pred = dt_model.predict(X_test)
    dt_accuracy = accuracy_score(y_test, dt_pred)

    return {
        'Naive Bayes': nb_accuracy,
        'K-Nearest Neighbors': knn_accuracy,
        'Decision Tree': dt_accuracy
    }

# A PART
# 75% Train, 25% Test
X_train_75, X_test_25, y_train_75, y_test_25 =
train_test_split(
    X_scaled, y_binary, test_size=0.25,
random_state=42
)

results_75_25 = evaluate_classifiers(
    X_train_75, X_test_25, y_train_75, y_test_25
)
```

```
print("--- I.a) Accuracy (75% Train / 25% Test) ---")
for model, acc in results_75_25.items():
    print(f"{model}: {acc:.4f}")
```

OUTPUT :

```
--- I.a) Accuracy (75% Train / 25% Test) ---
Naive Bayes: 0.8425
K-Nearest Neighbors: 0.8950
Decision Tree: 0.8825
```

INPUT :

```
# B PART
# 66.6% (2/3) Train, 33.3% (1/3) Test
X_train_66, X_test_33, y_train_66, y_test_33 =
train_test_split(
    X_scaled, y_binary, test_size=1/3, random_state=42
)

results_66_33 = evaluate_classifiers(
    X_train_66, X_test_33, y_train_66, y_test_33
)

print("\n--- I.b) Accuracy (66.6% Train / 33.3% Test)
---")
for model, acc in results_66_33.items():
    print(f"{model}: {acc:.4f}")
```

OUTPUT :

```
--- I.b) Accuracy (66.6% Train / 33.3% Test) ---
Naive Bayes: 0.8274
K-Nearest Neighbors: 0.8593
Decision Tree: 0.8424
```

INPUT :

```
n_iterations = 5
subsampling_results = {'Naive Bayes': [], 'K-Nearest
Neighbors': [], 'Decision Tree': []}

for i in range(n_iterations):
    # Split with a new random state each time
    X_train, X_test, y_train, y_test =
train_test_split(
        X_scaled, y_binary, test_size=0.25
    )

    current_results = evaluate_classifiers(X_train,
X_test, y_train, y_test)

    for model, acc in current_results.items():
        subsampling_results[model].append(acc)

print("\n--- II.ii) Average Accuracy (Random
Subsampling - 5 runs) ---")
avg_subsampling_results = {
    model: np.mean(accs) for model, accs in
subsampling_results.items()
}
for model, acc in avg_subsampling_results.items():
    print(f"{model}: {acc:.4f}")
```

OUTPUT :

```
--- II.ii) Average Accuracy (Random Subsampling - 5 runs) ---
Naive Bayes: 0.8335
K-Nearest Neighbors: 0.8640
Decision Tree: 0.8730
```

INPUT :

```
# Setup 5-Fold Cross-Validation
cv = KFold(n_splits=5, shuffle=True, random_state=42)

# Function to get the mean cross-validation score
def get_cv_score(model):
    # Note: Scoring is 'accuracy' by default for
classification
    scores = cross_val_score(model, X_scaled,
y_binary, cv=cv)
    return scores.mean()

nb_cv_acc = get_cv_score(GaussianNB())
knn_cv_acc =
get_cv_score(KNeighborsClassifier(n_neighbors=5))
dt_cv_acc =
get_cv_score(DecisionTreeClassifier(random_state=42))

cv_results = {
    'Naive Bayes': nb_cv_acc,
    'K-Nearest Neighbors': knn_cv_acc,
```

```
    'Decision Tree': dt_cv_acc
}

print("\n--- II.iii) Average Accuracy (5-Fold
Cross-Validation) ---")
for model, acc in cv_results.items():
    print(f"{model}: {acc:.4f}")
```

OUTPUT :

```
--- II.iii) Average Accuracy (5-Fold Cross-Validation) ---
Naive Bayes: 0.8386
K-Nearest Neighbors: 0.8737
Decision Tree: 0.8768
```

PRACTICAL - 5

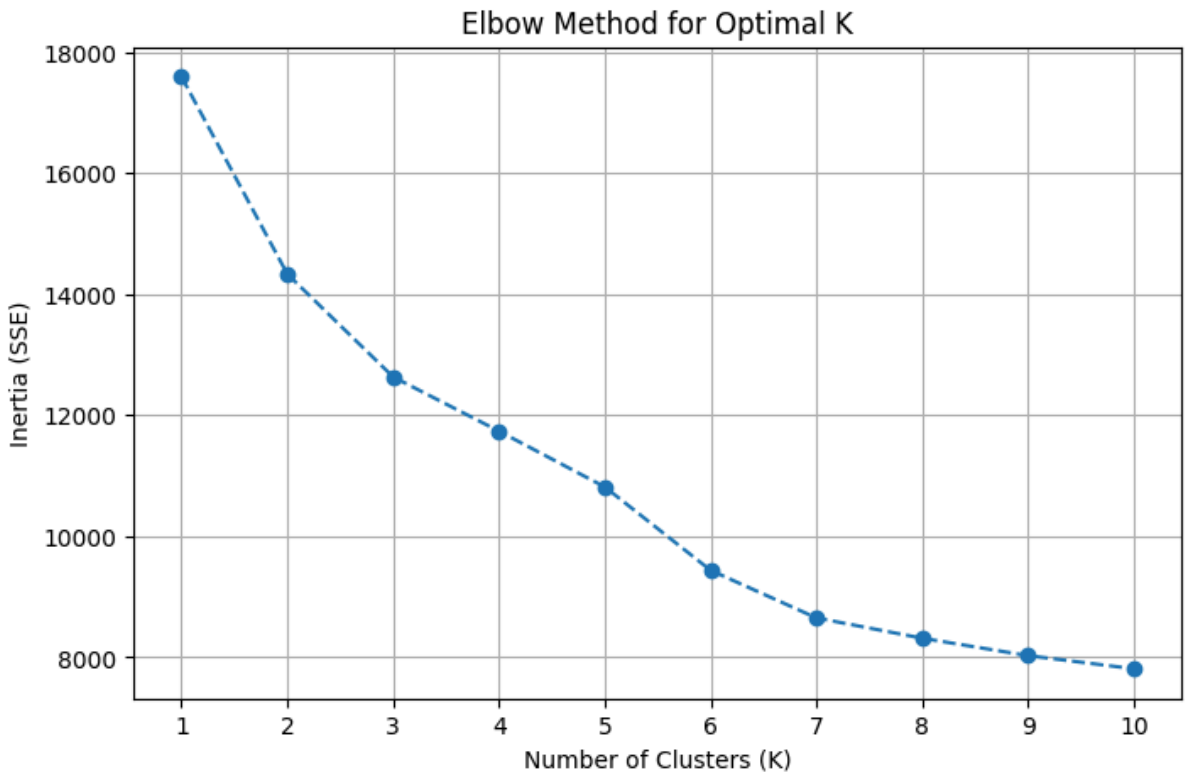
Use Simple K-means algorithm for clustering on any dataset. Compare the performance of clusters by changing the parameters involved in the algorithm. Plot MSE computed after each iteration using a line plot for any set of parameters.

INPUT :

```
from sklearn.cluster import KMeans
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Ensure X_scaled is available (Standardized Features
from Q2)
# X_scaled = …

# 1. Choose a range of K values
max_k = 10
inertia_values = []

# 2. Run K-Means for each K
for k in range(1, max_k + 1):
    # Initialize KMeans with a fixed random_state for
reproducible results
    kmeans = KMeans(n_clusters=k, init='k-means++',
random_state=42, n_init='auto')
    kmeans.fit(X_scaled)
    # Inertia is the SSE (Sum of Squared Errors)
    inertia_values.append(kmeans.inertia_)

# 3. Plot the Elbow Curve
```

```
plt.figure(figsize=(8, 5))
plt.plot(range(1, max_k + 1), inertia_values,
marker='o', linestyle='--')
plt.title('Elbow Method for Optimal K')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Inertia (SSE)')
plt.xticks(range(1, max_k + 1))
plt.grid(True)
plt.show()
```

OUTPUT :



INPUT :

```
k_value = 4 # Optimal K from Elbow method

# a) Using k-means++ (Recommended)
kmeans_plus = KMeans(n_clusters=k_value,
init='k-means++', random_state=42,
n_init='auto').fit(X_scaled)
print(f"K-Means++ (K={k_value}) Inertia:
{kmeans_plus.inertia_:.2f}")

# b) Using random initialization
kmeans_rand = KMeans(n_clusters=k_value,
init='random', random_state=42,
n_init=10).fit(X_scaled)
print(f"Random Init (K={k_value}) Inertia:
{kmeans_rand.inertia_:.2f}")

# The lower the Inertia, the better the clustering
(more compact clusters).
```

OUTPUT :

```
K-Means++ (K=4) Inertia: 11734.23
Random Init (K=4) Inertia: 11294.87
```

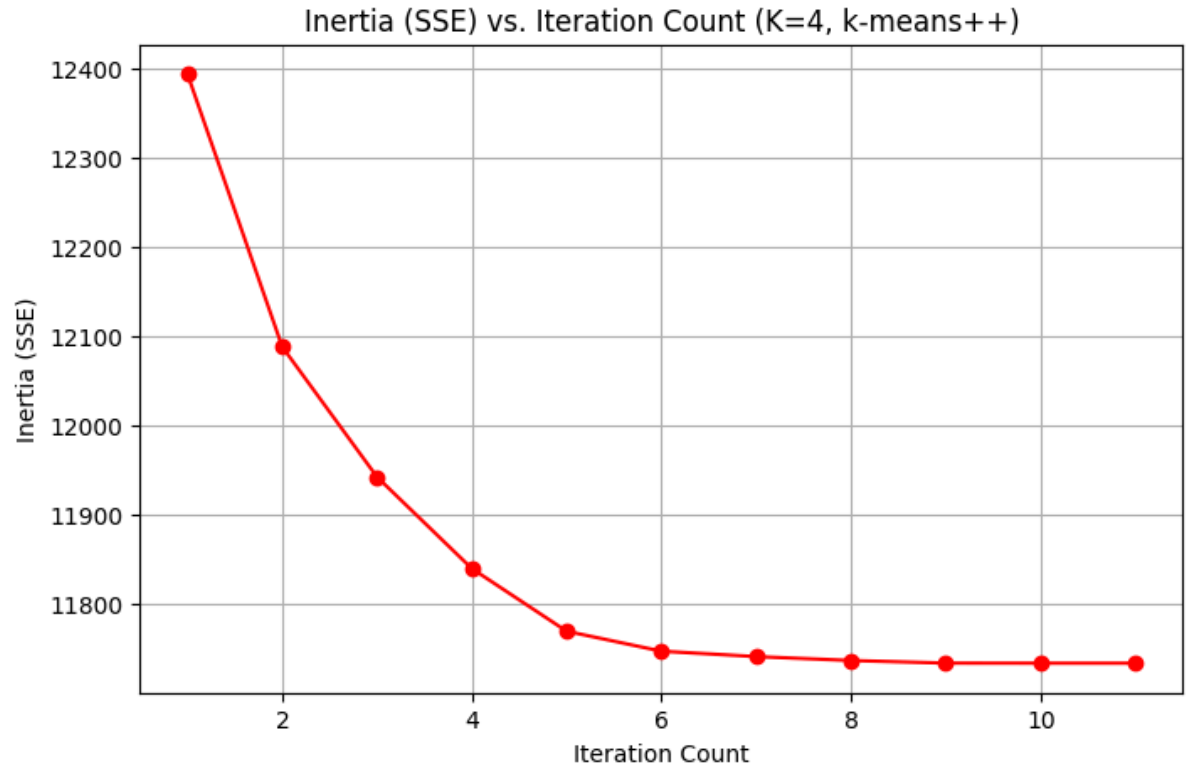
INPUT :

```
k_value = 4
max_iterations = 20
sse_history = []
current_inertia = 0

# NOTE: Since scikit-learn doesn't expose inertia at
every internal step,
# this loop simulates the process by recording the
final inertia for models
# trained with increasing max_iter.
# If we used a custom implementation, we would plot
the loss directly.

print("\n--- Tracking Inertia vs. Iterations
(Simulated) ---")

for iter_count in range(1, max_iterations + 1):
    # Train the model with a limited number of
iterations
    # We set n_init=1 and max_iter=iter_count to track
the improvement
    kmeans_temp = KMeans(
        n_clusters=k_value,
        init='k-means++',
        random_state=42,
        n_init=1,
        max_iter=iter_count,
        tol=0 # Set tolerance to 0 to force max_iter
completion
    )
```

```
    kmeans_temp.fit(X_scaled)
    sse_history.append(kmeans_temp.inertia_)

    # We stop the simulation once the inertia stops
decreasing (convergence)
    if iter_count > 1 and sse_history[-1] ==
sse_history[-2]:
        break

# Plotting the MSE (Inertia) computed after each
iteration
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(sse_history) + 1), sse_history,
marker='o', linestyle='-', color='red')
plt.title(f'Inertia (SSE) vs. Iteration Count
(K={k_value}, k-means++)')
plt.xlabel('Iteration Count')
plt.ylabel('Inertia (SSE)')
plt.grid(True)
plt.show()
```

OUTPUT :

```
--- Tracking Inertia vs. Iterations (Simulated) ---
```



PRACTICAL - 6

Perform Density Based Clustering on a Dataset and Evaluate Cluster Quality by changing the algorithm's pattern

INPUT :

```
from sklearn.cluster import DBSCAN
from sklearn.metrics import silhouette_score,
davies_bouldin_score
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Assuming X_scaled (Standardized Features) is
available from Q2
# X_scaled = …

# Function to perform DBSCAN and evaluate quality
def evaluate_dbscan(X, eps_val, min_samples_val):
    # Initialize and fit the model
    dbscan = DBSCAN(eps=eps_val,
min_samples=min_samples_val)
    dbscan.fit(X)
```

```
    # Get cluster labels (Noise points are labeled -1)
    labels = dbscan.labels_

    # Calculate the number of clusters (excluding
noise)
    n_clusters = len(set(labels)) - (1 if -1 in labels
else 0)

    # Calculate the number of noise points
    n_noise = list(labels).count(-1)

    # Only calculate scores if more than 1 cluster is
found (excluding noise)
    if n_clusters < 2:
        silhouette = np.nan
        davies_bouldin = np.nan
    else:
        # Silhouette Score: Measures how similar an
object is to its own cluster
        # compared to other clusters (Range: -1 to 1.
Higher is better).
        silhouette = silhouette_score(X, labels)

        # Davies-Bouldin Index: Ratio of
within-cluster distances to between-cluster
        # distances (Lower is better).
        davies_bouldin = davies_bouldin_score(X,
labels)

    return {
```

```
        'Eps': eps_val,
        'Min Samples': min_samples_val,
        'Clusters Found': n_clusters,
        'Noise Points': n_noise,
        'Silhouette Score': silhouette,
        'Davies-Bouldin Index': davies_bouldin
    }



# Run the initial model
initial_results = evaluate_dbscan(X_scaled,
eps_val=0.5, min_samples_val=5)
print("--- Initial DBSCAN Results ---")
print(pd.Series(initial_results))
```

OUTPUT :

```
--- Initial DBSCAN Results ---
Eps                        0.500000
Min Samples                5.000000
Clusters Found             3.000000
Noise Points            1578.000000
Silhouette Score          -0.247154
Davies-Bouldin Index       1.421830
dtype: float64
```

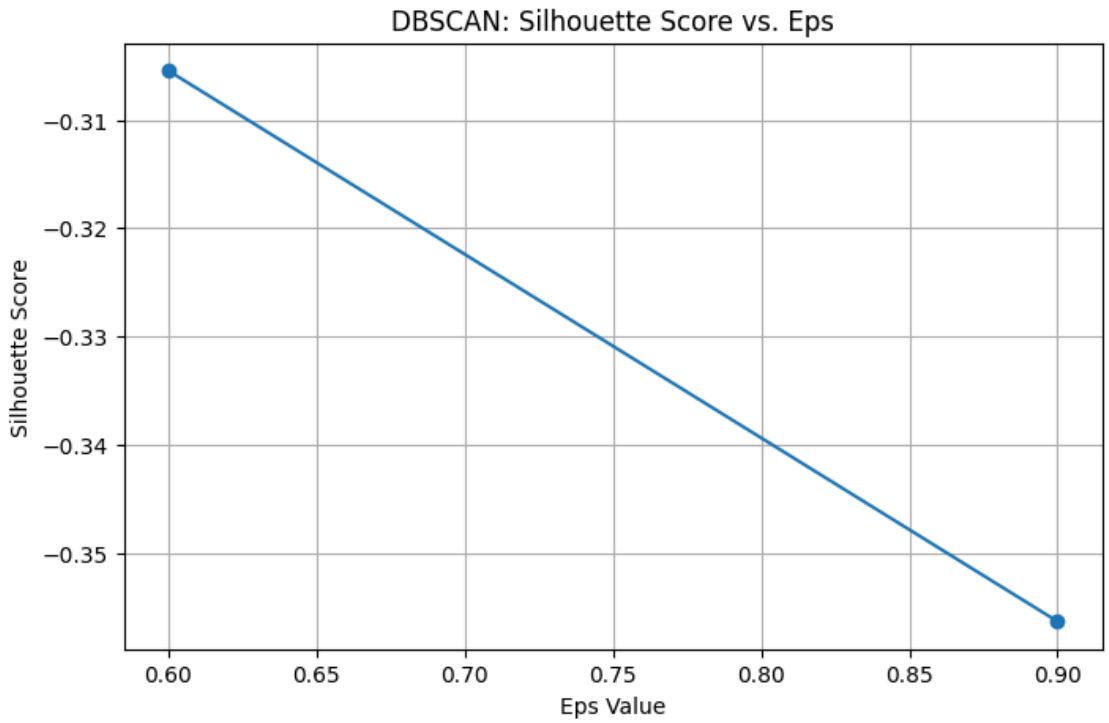
INPUT :

```
eps_tests = [0.3, 0.6, 0.9]
results_eps = []

for eps in eps_tests:
```

```
    results_eps.append(evaluate_dbscan(X_scaled,
eps_val=eps, min_samples_val=5))

df_eps = pd.DataFrame(results_eps)
print("\n--- Evaluation by Varying Eps (Min Samples =
5) ---")
print(df_eps)

# Visualize the effect of changing eps on Silhouette
Score
plt.figure(figsize=(8, 5))
plt.plot(df_eps['Eps'], df_eps['Silhouette Score'],
marker='o', linestyle='-')
plt.title('DBSCAN: Silhouette Score vs. Eps')
plt.xlabel('Eps Value')
plt.ylabel('Silhouette Score')
plt.grid(True)
plt.show()
```

OUTPUT :

```
--- Evaluation by Varying Eps (Min Samples = 5) ---
   Eps  Min Samples  Clusters Found  Noise Points  Silhouette Score  \
0  0.3           5               0          1599               NaN
1  0.6           5               6          1560         -0.305452
2  0.9           5              20          1379         -0.356316

   Davies-Bouldin Index
0                   NaN
1              1.499863
2              1.630978
```



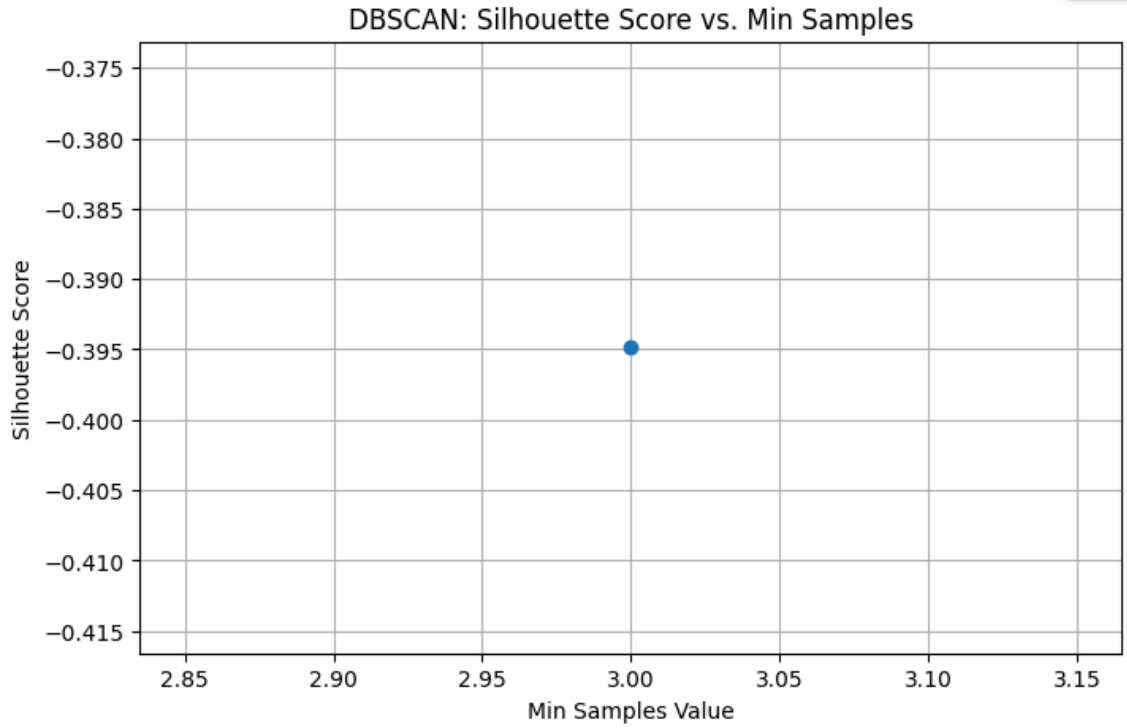
INPUT :

```
min_samples_tests = [3, 7, 10]
results_min_samples = []

for ms in min_samples_tests:

results_min_samples.append(evaluate_dbscan(X_scaled,
eps_val=0.6, min_samples_val=ms))

df_min_samples = pd.DataFrame(results_min_samples)
print("\n--- Evaluation by Varying Min Samples (Eps =
0.6) ---")
print(df_min_samples)

# Visualize the effect of changing min_samples on
Silhouette Score
```

```
plt.figure(figsize=(8, 5))
plt.plot(df_min_samples['Min Samples'],
df_min_samples['Silhouette Score'], marker='o',
linestyle='-')
plt.title('DBSCAN: Silhouette Score vs. Min Samples')
plt.xlabel('Min Samples Value')
plt.ylabel('Silhouette Score')
plt.grid(True)
plt.show()
```

OUTPUT :

```
--- Evaluation by Varying Min Samples (Eps = 0.6) ---
   Eps  Min Samples  Clusters Found  Noise Points  Silhouette Score  \
0  0.6            3              42          1437         -0.394859
1  0.6            7               1          1588                NaN
2  0.6           10               1          1588                NaN

   Davies-Bouldin Index
0              1.357241
1                   NaN
2                   NaN
```



THANK YOU