

**M.Sc. (Five Year Integrated) in Computer Science
(Artificial Intelligence & Data Science)**

Sixth Semester

Laboratory Record

**21-805-0606: MACHINE LEARNING AND
PARALLEL COMPUTING LAB**

*Submitted in partial fulfillment
of the requirements for the award of degree in
Master of Science (Five Year Integrated)
in Computer Science (Artificial Intelligence & Data Science) of
Cochin University of Science and Technology (CUSAT)
Kochi*



Submitted by

**AKSHAY K S
(80521003)**

**DEPARTMENT OF COMPUTER SCIENCE
COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY (CUSAT)
KOCHI-682022
JULY 2024**

DEPARTMENT OF COMPUTER SCIENCE
COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY (CUSAT)
KOCHI, KERALA-682022



*This is to certify that the software laboratory record for **21-805-0606: Machine Learning and Parallel computing Lab** is a record of work carried out by **AKSHAY (80521005)**, in partial fulfillment of the requirements for the award of degree in **Master of Science (Five Year Integrated) in Computer Science (Artificial Intelligence & Data Science)** of Cochin University of Science and Technology (CUSAT), Kochi. The lab record has been approved as it satisfies the academic requirements in respect of the sixth semester laboratory prescribed for the Master of Science (Five Year Integrated) in Computer Science degree.*

Faculty Member in-charge

Dr. Shailesh
Assistant Professor
Department of Computer Science
CUSAT

Dr. Madhu S. Nair
Professor and Head
Department of Computer Science
CUSAT

Table of Contents

Sl.No.	Program	Pg.No.
1	Data Preprocessing	pg
2	Classification	pg
3	Clustering	pg
4	Feature Selection	pg
5	Association Rule Mining	pg
6	Collaborative Filtering and Recommender Systems	pg
7	Solving Maze Problem using OpenAI Gym Library	pg
8	Parallel Image Processing with OpenMP	pg
9	Feature extraction	pg
10	Thread Creation	pg
11	Performance evaluation of sequential and thread execution programs	pg
12	Implementing clustering techniques on IRIS dataset	pg
13	Evaluating classifiers and applying ensemble methods	pg

DATA PREPROCESSING

AIM

Load the MNIST handwritten digit dataset and perform the following pre-processing steps:

1. Normalize the pixel values of the images.
2. Apply one-hot encoding to the target labels.
3. Split the data into training, validation, and test sets.

Dataset: <https://github.com/iamavieira/handwritten-digits-mnist>

PROGRAM

```
import tensorflow as tf
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split

mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

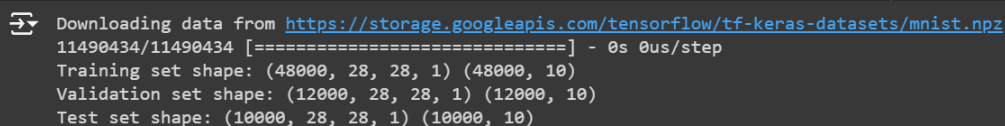
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2, random_

print("Training set shape:", x_train.shape, y_train.shape)
print("Validation set shape:", x_val.shape, y_val.shape)
print("Test set shape:", x_test.shape, y_test.shape)
```

SAMPLE INPUT-OUTPUT



```
📄 Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [=====] - 0s 0us/step
Training set shape: (48000, 28, 28, 1) (48000, 10)
Validation set shape: (12000, 28, 28, 1) (12000, 10)
Test set shape: (10000, 28, 28, 1) (10000, 10)
```

CLASSIFICATION

AIM

1. Build a logistic regression model to classify handwritten digits from the MNIST dataset.
2. Evaluate the model performance using accuracy, precision, recall, and F1 score.
3. Fine-tune the model hyperparameters using grid search CV to improve performance.
4. Visualize the decision boundary of the model.

PROGRAM

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confu
from sklearn.model_selection import GridSearchCV, train_test_split

mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize the pixel values of the images
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Flatten the images
x_train = x_train.reshape(-1, 28*28)
x_test = x_test.reshape(-1, 28*28)

# Made the subset snaller for faster iteration
x_train_small, _, y_train_small, _ = train_test_split(x_train, y_train, train_size=0.1, ra
x_val, _, y_val, _ = train_test_split(x_test, y_test, train_size=0.1, random_state=42)

# Build a logistic regression model
log_reg = LogisticRegression(max_iter=1000, solver='saga', multi_class='multinomial', n_jo

# Train the model
log_reg.fit(x_train_small, y_train_small)
```

```
# Predict on the validation set
y_val_pred = log_reg.predict(x_val)

# Evaluate the model
accuracy = accuracy_score(y_val, y_val_pred)
precision = precision_score(y_val, y_val_pred, average='macro')
recall = recall_score(y_val, y_val_pred, average='macro')
f1 = f1_score(y_val, y_val_pred, average='macro')

print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")

# Display the confusion matrix
conf_matrix = confusion_matrix(y_val, y_val_pred)
ConfusionMatrixDisplay(conf_matrix).plot()
plt.show()

param_grid = {
    'C': [0.1, 1, 10],
    'solver': ['saga']}

grid_search = GridSearchCV(LogisticRegression(max_iter=1000, multi_class='multinomial', n_
grid_search.fit(x_train_small, y_train_small)

print(f"Best parameters: {grid_search.best_params_}")
print(f"Best cross-validation accuracy: {grid_search.best_score_:.4f}")

# Predict on the validation set with the best estimator
best_log_reg = grid_search.best_estimator_
y_val_pred_best = best_log_reg.predict(x_val)
accuracy_best = accuracy_score(y_val, y_val_pred_best)
precision_best = precision_score(y_val, y_val_pred_best, average='macro')
recall_best = recall_score(y_val, y_val_pred_best, average='macro')
f1_best = f1_score(y_val, y_val_pred_best, average='macro')

print(f"Best Model Accuracy: {accuracy_best:.4f}")
print(f"Best Model Precision: {precision_best:.4f}")
print(f"Best Model Recall: {recall_best:.4f}")
```

```
print(f"Best Model F1 Score: {f1_best:.4f}")
conf_matrix_best = confusion_matrix(y_val, y_val_pred_best)
ConfusionMatrixDisplay(conf_matrix_best).plot()
plt.show()
```

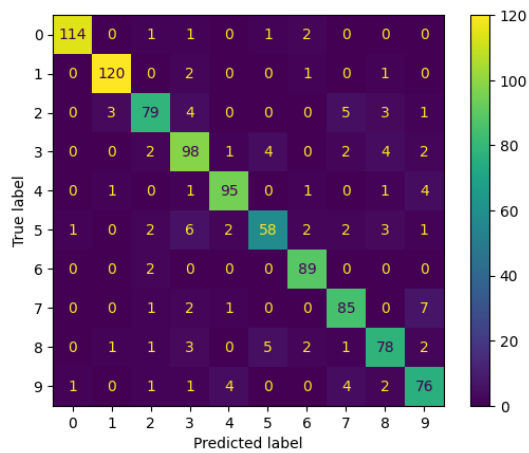
SAMPLE INPUT-OUTPUT

Accuracy: 0.8920

Precision: 0.8877

Recall: 0.8856

F1 Score: 0.8861



Best parameters: 'C': 0.1, 'solver': 'saga'

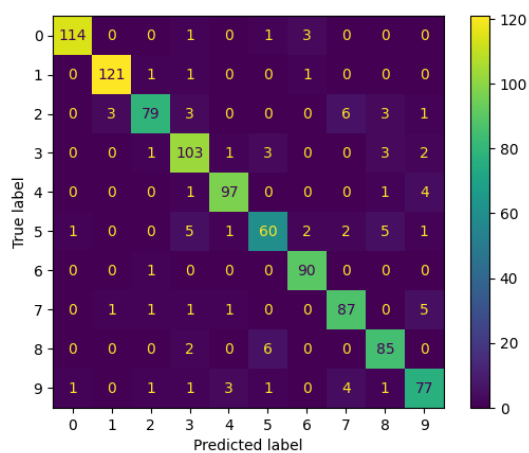
Best cross-validation accuracy: 0.9030

Best Model Accuracy: 0.9130

Best Model Precision: 0.9090

Best Model Recall: 0.9072

Best Model F1 Score: 0.9074



CLUSTERING

AIM

1. Apply K-means clustering to group customers based on their purchase history and demographic information.
2. Determine the optimal number of clusters using the elbow method.
3. Analyze the characteristics of each cluster to identify customer segments.
4. Visualize the clusters using scatter plots and dimensionality reduction techniques.

Dataset: <https://www.kaggle.com/datasets/aungpyaeap/supermarket-sales>

PROGRAM

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score

# Load the dataset
url = 'https://www.kaggle.com/datasets/aungpyaeap/supermarket-sales/download'
dataset_path = '/content/supermarket_sales - Sheet1.csv'
data = pd.read_csv(dataset_path)

# Display first few rows of the dataset
print(data.head())

# Preprocess the data
# Encode categorical variables
label_encoders = {}
for column in data.select_dtypes(include=['object']).columns:
    le = LabelEncoder()
    data[column] = le.fit_transform(data[column])
    label_encoders[column] = le

# Scale the data
scaler = StandardScaler()
```



```
scaled_data = scaler.fit_transform(data)

# Apply K-means clustering
kmeans = KMeans(n_clusters=3, random_state=42)
clusters = kmeans.fit_predict(scaled_data)
data['Cluster'] = clusters

# Determine the optimal number of clusters using the elbow method
inertia = []
for n in range(1, 11):
    kmeans = KMeans(n_clusters=n, random_state=42)
    kmeans.fit(scaled_data)
    inertia.append(kmeans.inertia_)

plt.figure(figsize=(8, 4))
plt.plot(range(1, 11), inertia, marker='o')
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia')
plt.title('Elbow Method for Optimal Number of Clusters')
plt.show()

# Analyze the characteristics of each cluster
cluster_analysis = data.groupby('Cluster').mean()
print(cluster_analysis)

# Visualize the clusters using PCA for dimensionality reduction
pca = PCA(n_components=2)
pca_components = pca.fit_transform(scaled_data)

plt.figure(figsize=(10, 6))
sns.scatterplot(x=pca_components[:, 0], y=pca_components[:, 1], hue=clusters, palette='vir')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.title('Cluster Visualization using PCA')
plt.legend(title='Cluster')
plt.show()

# Calculate silhouette score for cluster validation
silhouette_avg = silhouette_score(scaled_data, clusters)
print(f'Silhouette Score: {silhouette_avg:.2f}')
```

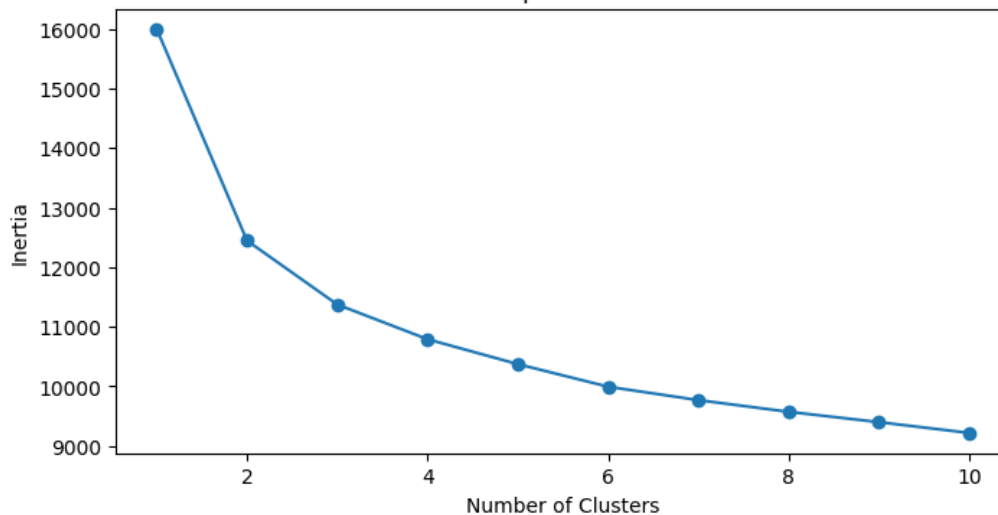
SAMPLE INPUT-OUTPUT

	Invoice ID	Branch	City	Customer type	Gender	\
0	750-67-8428	A	Yangon	Member	Female	
1	226-31-3081	C	Naypyitaw	Normal	Female	
2	631-41-3108	A	Yangon	Normal	Male	
3	123-19-1176	A	Yangon	Member	Male	
4	373-73-7910	A	Yangon	Normal	Male	

	Product line	Unit price	Quantity	Tax 5%	Total	Date	\
0	Health and beauty	74.69	7	26.1415	548.9715	1/5/2019	
1	Electronic accessories	15.28	5	3.8200	80.2200	3/8/2019	
2	Home and lifestyle	46.33	7	16.2155	340.5255	3/3/2019	
3	Health and beauty	58.22	8	23.2880	489.0480	1/27/2019	
4	Sports and travel	86.31	7	30.2085	634.3785	2/8/2019	

	Time	Payment	cogs	gross margin percentage	gross income	Rating
0	13:08	Ewallet	522.83	4.761905	26.1415	9.1
1	10:29	Cash	76.40	4.761905	3.8200	9.6
2	13:23	Credit card	324.31	4.761905	16.2155	7.4
3	20:33	Ewallet	465.76	4.761905	23.2880	8.4
4	10:37	Ewallet	604.17	4.761905	30.2085	5.3

Elbow Method for Optimal Number of Clusters

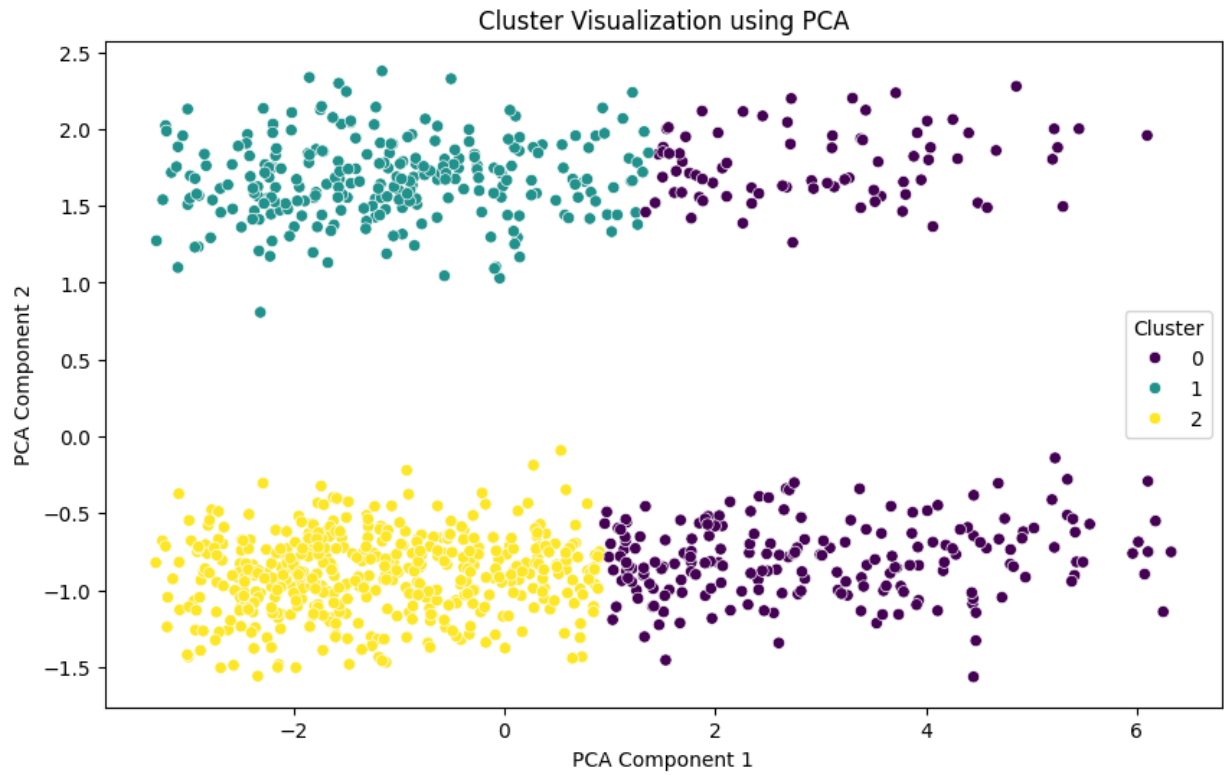


	Invoice ID	Branch	City	Customer type	Gender	\
Cluster						
0	508.820423	1.098592	0.943662	0.478873	0.461268	
1	506.284615	0.000000	2.000000	0.530769	0.538462	
2	489.826754	1.482456	0.482456	0.493421	0.500000	

	Product line	Unit price	Quantity	Tax 5%	Total	Date	\
Cluster							
0	2.524648	77.983310	8.109155	31.159685	654.353382	42.753521	
1	2.511538	47.104769	4.669231	9.749296	204.735219	46.334615	
2	2.372807	46.661469	4.370614	8.761407	183.989543	44.721491	

	Time	Payment	cogs	gross margin percentage	\
Cluster					
0	247.742958	1.007042	623.193697	4.761905	
1	239.830769	1.030769	194.985923	4.761905	
2	259.782895	0.980263	175.228136	4.761905	

	gross income	Rating
Cluster		
0	31.159685	6.833451
1	9.749296	7.087692
2	8.761407	6.993860



Silhouette Score: 0.14

FEATURE SELECTION

AIM

The Iris dataset consists of 150 samples with 4 features each: sepal length, sepal width, petal length, and petal width. Each sample belongs to one of three classes: setosa, versicolor, or virginica.

1. Load the Iris dataset and split it into features (X) and target labels (y).
2. Perform exploratory data analysis (EDA) to gain insights into the dataset.
3. Implement feature selection techniques:
 - (a) Univariate Feature Selection
 - (b) Feature Importance using Random Forest
 - (c) Recursive Feature Elimination (RFE) using Support Vector Machine (SVM)
4. Evaluate the performance of the selected features using a classification model (e.g., SVM or Logistic Regression).
5. Compare the model performance before and after feature selection.

PROGRAM

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import RFE
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler, MinMaxScaler

iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names
df = pd.DataFrame(X, columns=feature_names)
df['target'] = y
```

```
sns.pairplot(df, hue='target')
plt.show()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

min_max_scaler = MinMaxScaler()
X_train_minmax = min_max_scaler.fit_transform(X_train)
X_test_minmax = min_max_scaler.transform(X_test)

selector = SelectKBest(chi2, k=2)
X_train_kbest = selector.fit_transform(X_train_minmax, y_train)
X_test_kbest = selector.transform(X_test_minmax)
print("Selected features (Univariate Feature Selection):", np.array(feature_names)[selector.get_support()])
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train_scaled, y_train)
importances = rf.feature_importances_
indices = np.argsort(importances)[::-1]
selected_features_rf = np.array(feature_names)[indices[:2]]
print("Selected features (Random Forest Importance):", selected_features_rf)

svm = SVC(kernel='linear')
rfe = RFE(svm, n_features_to_select=2)
rfe.fit(X_train_scaled, y_train)
selected_features_rfe = np.array(feature_names)[rfe.support_]
print("Selected features (RFE using SVM):", selected_features_rfe)

def evaluate_model(X_train, X_test, y_train, y_test, model):
    model.fit(X_train, y_train)
    accuracy = model.score(X_test, y_test)
    return accuracy

lr = LogisticRegression(random_state=42)
accuracy_original = evaluate_model(X_train_scaled, X_test_scaled, y_train, y_test, lr)
print("Accuracy with original features:", accuracy_original)

accuracy_kbest = evaluate_model(X_train_kbest, X_test_kbest, y_train, y_test, lr)
print("Accuracy with selected features (Univariate):", accuracy_kbest)
```

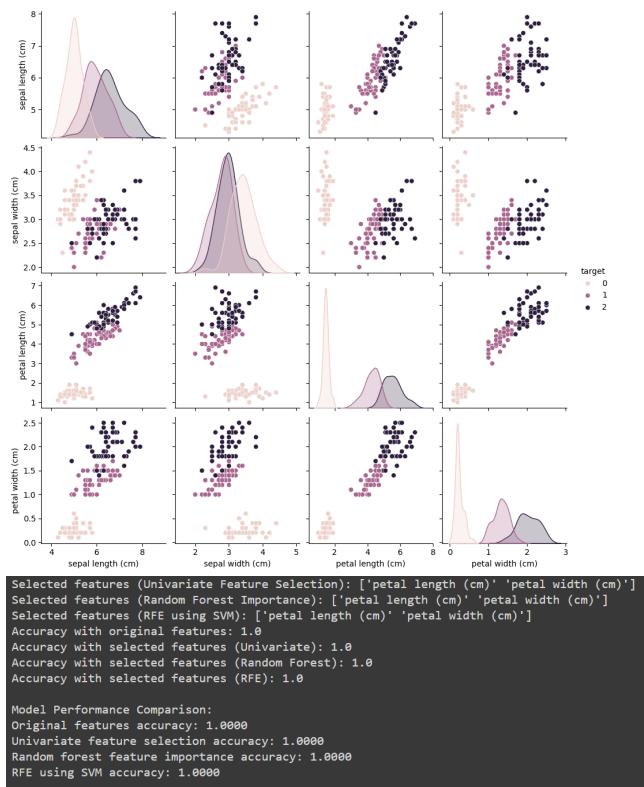
```

X_train_rf = X_train_scaled[:, indices[:2]]
X_test_rf = X_test_scaled[:, indices[:2]]
accuracy_rf = evaluate_model(X_train_rf, X_test_rf, y_train, y_test, lr)
print("Accuracy with selected features (Random Forest):", accuracy_rf)

X_train_rfe = X_train_scaled[:, rfe.support_]
X_test_rfe = X_test_scaled[:, rfe.support_]
accuracy_rfe = evaluate_model(X_train_rfe, X_test_rfe, y_train, y_test, lr)
print("Accuracy with selected features (RFE):", accuracy_rfe)
print("\nModel Performance Comparison:")
print(f"Original features accuracy: {accuracy_original:.4f}")
print(f"Univariate feature selection accuracy: {accuracy_kbest:.4f}")
print(f"Random forest feature importance accuracy: {accuracy_rf:.4f}")
print(f"RFE using SVM accuracy: {accuracy_rfe:.4f}")

```

SAMPLE INPUT-OUTPUT



ASSOCIATION RULE MINING

AIM

To understand and implement association rule mining techniques to uncover patterns and associations within a given dataset of transactional data.

1. Dataset:

- (a) Take any dataset containing transactions from a grocery store. Each transaction consists of a list of items purchased by a customer. Utilizing association rule mining techniques, analyze the dataset to uncover patterns and associations between items purchased together.
- (b) Load the provided dataset containing transactional data from a grocery store.
- (c) Examine the structure of the dataset to understand the transactional format.

2. Generating Itemsets:

- (a) Implement a function to generate individual itemsets from the dataset.
- (b) Calculate the support for each itemset, which represents the frequency of occurrence of each item.

3. Identifying Frequent Itemsets:

- (a) Implement the Apriori algorithm to identify frequent itemsets within the dataset.
- (b) Determine the minimum support threshold for identifying frequent itemsets.

4. Deriving Association Rules:

- (a) Utilize the frequent itemsets obtained from the Apriori algorithm to derive association rules.
- (b) Calculate the confidence for each association rule, representing the likelihood of one item being purchased given the purchase of another item.
- (c) Set a minimum confidence threshold for selecting meaningful association rules.

5. Evaluation of Association Rules:

- (a) Evaluate the generated association rules using appropriate metrics such as support, confidence, and lift.
- (b) Identify high-confidence rules with significant lift values, indicating strong associations between items.

PROGRAM

```
import pandas as pd
from itertools import combinations
# Load the dataset
dataset = pd.read_csv('/content/Groceries_dataset.csv')
# Inspect the dataset structure
print("Dataset head:")
print(dataset.head())
print("\nDataset info:")
print(dataset.info())

# Group by Member_number and Date to create transactions
grouped = dataset.groupby(['Member_number', 'Date'])['itemDescription'].
apply(list).reset_index()

# Extract transactions
transactions = grouped['itemDescription'].tolist()

# Print the generated transactions to verify correctness
print("Generated transactions:")
for i, transaction in enumerate(transactions[:10]):
# Print only the first 10 for brevity
    print(f"Transaction {i+1}: {transaction}")
# Function to generate individual itemsets
def get_itemsets(transactions):
    itemsets = {}
    for transaction in transactions:
        for item in transaction:
            if item not in itemsets:
                itemsets[item] = 0
            itemsets[item] += 1
    return itemsets
itemsets = get_itemsets(transactions)
# Print itemsets to verify correctness
print("Generated itemsets:")
print(itemsets)
# Calculate Support
def calculate_support(itemsets, transactions):
    support = {}
    total_transactions = len(transactions)
    for itemset, count in itemsets.items():
        support[itemset] = count / total_transactions
```



```
    return support
support = calculate_support(itemsets, transactions)
# Print support to verify correctness
print("Support for itemsets:")
print(support)
#Implement the Apriori algorithm to identify frequent itemsets within the dataset.
def apriori(transactions, min_support):
    itemsets = get_itemsets(transactions)
    support = calculate_support(itemsets, transactions)
    frequent_itemsets = {itemset: sup for itemset, sup in support.items()
        if sup >= min_support}
    k = 2
    while True:
        candidate_itemsets = {}
        for combination in combinations(frequent_itemsets, k):
            candidate = tuple(set().union(*combination))
            count = sum(1 for transaction in transactions if
                set(candidate).issubset(transaction))
            if count / len(transactions) >= min_support:
                candidate_itemsets[candidate] = count
        if not candidate_itemsets:
            break frequent_itemsets.update(candidate_itemsets)
        k += 1
    return frequent_itemsets
min_support = 0.01
frequent_itemsets = apriori(transactions, min_support)
print(frequent_itemsets)
def calculate_metrics(frequent_itemsets, transactions):
    rules = []
    for itemset in frequent_itemsets:
        if len(itemset) > 1:
            for consequent in combinations(itemset, 1):
                antecedent = tuple(set(itemset) - set(consequent))
                antecedent_support = frequent_itemsets.get(antecedent, 0)
                consequent_support = frequent_itemsets.get(consequent, 0)
                itemset_support = frequent_itemsets[itemset]
                confidence = itemset_support / antecedent_support
                if antecedent_support else 0
                lift = confidence / consequent_support
                if consequent_support else 0
                rules.append({
```

```

        'antecedent': antecedent,
        'consequent': consequent,
        'support': itemset_support,
        'confidence': confidence,
        'lift': lift
    })

    return rules

rules = calculate_metrics(frequent_itemsets, transactions)
# Print the generated rules with metrics
for rule in rules:
    print(f"Rule: {rule['antecedent']} -> {rule['consequent']}")
    print(f"   Support: {rule['support']}") # Remove the formatting :.4f
    print(f"   Confidence: {rule['confidence']}") # Remove the formatting :.4f
    print(f"   Lift: {rule['lift']}") # Remove the formatting :.4f
    print()

# Identify high-confidence rules with significant lift
high_confidence_rules = [rule for rule in rules
    if rule['confidence'] >= 0.5 and rule['lift'] > 1]
print("High-confidence rules with significant lift:")
for rule in high_confidence_rules:
    print(f"Rule: {rule['antecedent']} -> {rule['consequent']}")
    print(f"   Support: {rule['support']}") # Remove the formatting :.4f
    print(f"   Confidence: {rule['confidence']}") # Remove the formatting :.4f
    print(f"   Lift: {rule['lift']}") # Remove the formatting :.4f
    print()

```

SAMPLE INPUT-OUTPUT

```

{'sausage': 0.061752322395241595, 'whole milk': 0.16721245739490745, 'yogurt': 0.08915324467018645, 'pastry': 0.05246274142885785, 'salty snack': 0.018913319521486335,
Dataset head:
  Member_number  Date  itemDescription
0      1808      21-07-2015  tropical fruit
1      2552      05-01-2015    whole milk
2      2380      19-09-2015    pip fruit
3      1187      12-12-2015  other vegetables
4      3037      01-02-2015    whole milk

Dataset info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 38765 entries, 0 to 38764
Data columns (total 3 columns):
 #   Column        Non-Null Count  Dtype
---  -
 0   Member_number  38765 non-null  int64
 1   Date           38765 non-null  object
 2   itemDescription 38765 non-null  object
dtypes: int64(1), object(2)
memory usage: 908.7+ KB
None
Generated transactions:
Transaction 1: ['sausage', 'whole milk', 'semi-finished bread', 'yogurt']
Transaction 2: ['whole milk', 'pastry', 'salty snack']
Transaction 3: ['canned beer', 'misc. beverages']
Transaction 4: ['sausage', 'hygiene articles']
...
Generated itemsets:
{'sausage': 924, 'whole milk': 2502, 'semi-finished bread': 142, 'yogurt': 1334, 'pastry': 8
Support for itemsets:
{'sausage': 0.061752322395241595, 'whole milk': 0.16721245739490745, 'semi-finished bread'
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings

Rule: ('o', 'u', 'p', 't', 'r') -> ('s',)
Support: 0      0.024068
1      0.020947
2      0.048962
3      0.020250
4      0.044267
5      0.027628
6      0.023836
7      0.039056
8      0.026622
9      0.064543
10     0.034412
Name: support, dtype: float64
Confidence: 0
Lift: 0

Rule: ('s', 'o', 'p', 't', 'r') -> ('u',)
Support: 0      0.024068
1      0.020947
2      0.048962
3      0.020250
4      0.044267
5      0.027628
6      0.023836
7      0.039056
8      0.026622
9      0.064543
10     0.034412
Name: support, dtype: float64

```

COLLABORATIVE FILTERING AND RECOMMENDER SYSTEM

AIM

To understand and implement collaborative filtering techniques for building recommender systems.

1. A. Understanding Collaborative Filtering:
 - (a) Provide an overview of collaborative filtering and its importance in building recommender systems.
 - (b) Explain the concepts of user-item interactions, user-based collaborative filtering, and item-based collaborative filtering.
2. Dataset Exploration:
 - (a) Select a suitable dataset for building a recommender system (e.g., movie ratings, book reviews, product ratings).
 - (b) Load the dataset and explore its structure and attributes.
3. User-Based Collaborative Filtering:
 - (a) Implement a user-based collaborative filtering algorithm to recommend items to users based on similarities between users.
 - (b) Discuss different similarity metrics such as cosine similarity, Pearson correlation, and Euclidean distance.
 - (c) Evaluate the performance of the user-based collaborative filtering approach using appropriate evaluation metrics (e.g., precision, recall, F1-score).
4. Item-Based Collaborative Filtering:
 - (a) Implement an item-based collaborative filtering algorithm to recommend items to users based on similarities between items.
 - (b) Discuss the advantages and disadvantages of item-based collaborative filtering compared to user-based collaborative filtering.
 - (c) Evaluate the performance of the item-based collaborative filtering approach using similar evaluation metrics as in the user-based approach.
5. Hybrid Approaches:
 - (a) Discuss hybrid recommender systems that combine collaborative filtering with other techniques such as content-based filtering or matrix factorization.

- (b) Implement a simple hybrid recommender system by combining user-based and item-based collaborative filtering approaches.
- (c) Evaluate the performance of the hybrid recommender system and compare it with the individual approaches.

6. Evaluation and Interpretation:

- (a) Analyze the results of the collaborative filtering and hybrid approaches.
- (b) Interpret the recommended items and their relevance to users.
- (c) Discuss potential improvements and future directions for enhancing the recommender system's performance.

PROGRAM

```
!pip install surprise
import pandas as pd
from surprise import Dataset, Reader
from surprise.model_selection import train_test_split

# Load the MovieLens 100K dataset
# The 'ml-100k' is a built-in dataset name, not a file path.
data = Dataset.load_builtin('ml-100k')

# Explore the dataset structure
trainset, testset = train_test_split(data, test_size=.25)
# Print some ratings from the training set
print("Some ratings from the training set:")
# Iterate and unpack the ratings, handling potential variations in tuple length
for rating in list(trainset.all_ratings())[:5]:
    if len(rating) == 4:
        user_id, item_id, rating_value, _ = rating
        print(f"User {user_id} rated item {item_id} with {rating_value}")
    elif len(rating) == 3:
        user_id, item_id, rating_value = rating
        print(f"User {user_id} rated item {item_id} with {rating_value}")
    else:
        print(f"Unexpected rating format: {rating}")

# Get the raw ratings data
raw_ratings = data.raw_ratings
```

```
# Create a DataFrame from the raw ratings
df = pd.DataFrame(raw_ratings, columns=['user_id', 'item_id', 'rating', 'timestamp'])

# Explore the DataFrame
print("\nDataFrame head:")
print(df.head())

print("\nDataFrame info:")
print(df.info())

print("\nSummary statistics for ratings:")
print(df['rating'].describe())

import numpy as np
from surprise import KNNBasic, accuracy
from surprise.model_selection import cross_validate

sim_options = {'name': ['cosine', 'pearson', 'msd', 'pearson_baseline']}

for sim_name in sim_options['name']:
    # Initialize the KNNBasic algorithm with the specified similarity metric
    algo = KNNBasic(sim_options={'name': sim_name, 'user_based': True})

    # Perform cross-validation and evaluate performance
    results = cross_validate(algo, data, measures=['RMSE', 'MAE'], cv=5, verbose=True)

    # Print average RMSE and MAE for the current similarity metric
    print(f"Average RMSE for {sim_name} similarity: {np.mean(results['test_rmse']):.4f}")
    print(f"Average MAE for {sim_name} similarity: {np.mean(results['test_mae']):.4f}")
    print("-" * 30)

import numpy as np
from surprise import accuracy
from surprise.model_selection import KFold
best_sim_name = 'pearson_baseline' # Replace with the actual best metric
algo = KNNBasic(sim_options={'name': best_sim_name, 'user_based': True})
# Use K-fold cross-validation for evaluation
kf = KFold(n_splits=5)
precisions = []
recalls = []
f1_scores = []
```

```
for trainset, testset in kf.split(data):
    # Train the algorithm on the training set
    algo.fit(trainset)
    # Make predictions on the test set
    predictions = algo.test(testset)
    # Calculate precision, recall, and F1-score at a specific threshold
    (e.g., rating >= 3.5)
    threshold = 3.5
    true_positives = 0
    false_positives = 0
    false_negatives = 0
    for prediction in predictions:
        actual_rating = prediction.r_ui
        predicted_rating = prediction.est
        if predicted_rating >= threshold and actual_rating >= threshold:
            true_positives += 1
        elif predicted_rating >= threshold and actual_rating < threshold:
            false_positives += 1
        elif predicted_rating < threshold and actual_rating >= threshold:
            false_negatives += 1
    if true_positives + false_positives > 0:
        precision = true_positives / (true_positives + false_positives)
    else:
        precision = 0
    if true_positives + false_negatives > 0:
        recall = true_positives / (true_positives + false_negatives)
    else:
        recall = 0
    if precision + recall > 0:
        f1 = 2 * precision * recall / (precision + recall)
    else:
        f1 = 0
    precisions.append(precision)
    recalls.append(recall)
    f1_scores.append(f1)
# Print average precision, recall, and F1-score
print(f"Average Precision: {np.mean(precisions):.4f}")
print(f"Average Recall: {np.mean(recalls):.4f}")
print(f"Average F1-score: {np.mean(f1_scores):.4f}")
import numpy as np
# Item-based collaborative filtering
```

```
algo = KNNBasic(sim_options={'name': 'cosine', 'user_based': False}) # Set user_based to False
# Perform cross-validation and evaluate performance
results = cross_validate(algo, data, measures=['RMSE', 'MAE'], cv=5, verbose=True)
print(f"Average RMSE for item-based collaborative filtering: {np.mean(results['test_rmse']):.4f}")
print(f"Average MAE for item-based collaborative filtering: {np.mean(results['test_mae']):.4f}")

import numpy as np
from surprise import KNNBasic, accuracy
from surprise.model_selection import KFold

# Item-based collaborative filtering
algo = KNNBasic(sim_options={'name': 'cosine', 'user_based': False}) # Set user_based to False
# Use K-fold cross-validation for evaluation
kf = KFold(n_splits=5)
precisions = []
recalls = []
f1_scores = []
for trainset, testset in kf.split(data):
    # Train the algorithm on the training set
    algo.fit(trainset)
    # Make predictions on the test set
    predictions = algo.test(testset)
    # Calculate precision, recall,
    # and F1-score at a specific threshold (e.g., rating >= 3.5)
    threshold = 3.5
    true_positives = 0
    false_positives = 0
    false_negatives = 0
    for prediction in predictions:
        actual_rating = prediction.r_ui
        predicted_rating = prediction.est
        if predicted_rating >= threshold and actual_rating >= threshold:
            true_positives += 1
        elif predicted_rating >= threshold and actual_rating < threshold:
            false_positives += 1
        elif predicted_rating < threshold and actual_rating >= threshold:
            false_negatives += 1
    if true_positives + false_positives > 0:
        precision = true_positives / (true_positives + false_positives)
    else:
        precision = 0
    if true_positives + false_negatives > 0:
```

```
    recall = true_positives / (true_positives + false_negatives)
else:
    recall = 0
if precision + recall > 0:
    f1 = 2 * precision * recall / (precision + recall)
else:
    f1 = 0
precisions.append(precision)
recalls.append(recall)
f1_scores.append(f1)
# Print average precision, recall, and F1-score for item-based collaborative filtering
print(f"Average Precision for Item-based CF: {np.mean(precisions):.4f}")
print(f"Average Recall for Item-based CF: {np.mean(recalls):.4f}")
print(f"Average F1-score for Item-based CF: {np.mean(f1_scores):.4f}")
# Initialize and train user-based and item-based collaborative filtering models
algo_user = KNNBasic(sim_options={'name': 'cosine', 'user_based': True})
algo_item = KNNBasic(sim_options={'name': 'cosine', 'user_based': False})
# Fit the models to your data (replace 'data' with your actual dataset)
algo_user.fit(data.build_full_trainset())
algo_item.fit(data.build_full_trainset())
# Get predictions from both models for a given user and item
user_id = '196' # Example user ID
item_id = '302' # Example item ID
prediction_user = algo_user.predict(user_id, item_id).est
prediction_item = algo_item.predict(user_id, item_id).est
# Combine predictions using a weighted average
weight_user = 0.6 # Adjust weights as needed
weight_item = 0.4
hybrid_prediction = weight_user * prediction_user + weight_item * prediction_item
print(f"Hybrid prediction for user {user_id} and item {item_id}: {hybrid_prediction:.4f}")
import numpy as np
# Evaluate the hybrid recommender using cross-validation
kf = KFold(n_splits=5)
rmse_hybrid = []
mae_hybrid = []
for trainset, testset in kf.split(data):
    # Train the user-based and item-based models
    algo_user.fit(trainset)
    algo_item.fit(trainset)
    predictions_hybrid = []
    for uid, iid, true_r in testset:
```



```

pred_user = algo_user.predict(uid, iid).est
pred_item = algo_item.predict(uid, iid).est
pred_hybrid = weight_user * pred_user + weight_item * pred_item
predictions_hybrid.append((uid, iid, true_r, pred_hybrid, None))
rmse_hybrid.append(accuracy.rmse(predictions_hybrid, verbose=False))
mae_hybrid.append(accuracy.mae(predictions_hybrid, verbose=False))

# Print average RMSE and MAE for the hybrid recommender
print(f"Average RMSE for Hybrid Recommender: {np.mean(rmse_hybrid):.4f}")
print(f"Average MAE for Hybrid Recommender: {np.mean(mae_hybrid):.4f}")
print("Average RMSE:")
print(f"  User-based CF: {np.mean(results['test_rmse']):.4f}")
print(f"  Item-based CF: {np.mean(results['test_rmse']):.4f}")
print(f"  Hybrid CF: {np.mean(rmse_hybrid):.4f}")
print("\nAverage MAE:")
print(f"  User-based CF: {np.mean(results['test_mae']):.4f}")
print(f"  Item-based CF: {np.mean(results['test_mae']):.4f}")
print(f"  Hybrid CF: {np.mean(mae_hybrid):.4f}")

```

SAMPLE INPUT-OUTPUT

```

/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: 'should_run_async' will not call 'transform_cell' automatically
and should_run_async(code)
Requirement already satisfied: surprise in /usr/local/lib/python3.10/dist-packages (0.1)
Requirement already satisfied: scikit-surprise in /usr/local/lib/python3.10/dist-packages (from surprise) (1.1.4)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from scikit-surprise->surprise) (1.4.2)
Requirement already satisfied: numpy>=1.19.5 in /usr/local/lib/python3.10/dist-packages (from scikit-surprise->surprise) (1.25.2)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.10/dist-packages (from scikit-surprise->surprise) (1.11.4)
Some ratings from the training set:
User 0 rated item 0 with 3.0
User 0 rated item 43 with 5.0
User 0 rated item 138 with 1.0
User 0 rated item 408 with 4.0
User 0 rated item 823 with 4.0

DataFrame head:
  user_id item_id rating timestamp
0    196     242      3.0  881258949
1    186     302      3.0  891717742
2     22     377      1.0  878887116
3    244     51      2.0  880606923
4    166     346      1.0  886397596

DataFrame info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 4 columns):
#   Column      Non-Null Count  Dtype
---  -
0   user_id    100000 non-null    object
1   item_id    100000 non-null    object
2   rating     100000 non-null    float64
3   timestamp  100000 non-null    object
dtypes: float64(1), object(3)
memory usage: 3.1+ MB
None

Summary statistics for ratings:
count    100000.000000
mean         3.529860
std          1.125674
min           1.000000
25%           3.000000
50%           4.000000
75%           4.000000
max           5.000000
Name: rating, dtype: float64

/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: 'should_run_async' will not call 'transform_cell' automatically
and should_run_async(code)
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
Evaluating RMSE, MAE of algorithm KNNBasic on 5 split(s).

   Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean  Std
RMSE (testset)  1.0142  1.0125  1.0185  1.0152  1.0273  1.0175  0.0053
MAE (testset)   0.8017  0.8019  0.8035  0.8021  0.8128  0.8044  0.0043
Fit time        0.85    0.70    0.61    0.59    0.76    0.70    0.10
Test time       4.49    3.23    3.46    4.55    3.25    3.80    0.59
Average RMSE for cosine similarity: 1.0175
Average MAE for cosine similarity: 0.8044
Computing the pearson similarity matrix...
Done computing similarity matrix.
Computing the pearson similarity matrix...
Done computing similarity matrix.

```

```

Done computing similarity matrix...
Computing the pearson similarity matrix...
Done computing similarity matrix...
Computing the pearson similarity matrix...
Done computing similarity matrix...
Computing the pearson similarity matrix...
Done computing similarity matrix...
Evaluating RMSE, MAE of algorithm KNNBasic on 5 split(s).

RMSE (testset)    Fold 1    Fold 2    Fold 3    Fold 4    Fold 5    Mean    Std
MAE (testset)     0.8018    0.8083    0.7990    0.8001    0.8038    0.8026    0.0033
Fit time          0.75      0.73      0.72      0.84      1.17      0.84      0.17
Test time         3.56      4.61      3.34      4.23      3.27      3.80      0.53
Average RMSE for pearson similarity: 1.0112
Average MAE for pearson similarity: 0.8026
-----
Computing the msd similarity matrix...
Done computing similarity matrix...
Computing the msd similarity matrix...
Done computing similarity matrix...
Computing the msd similarity matrix...
Done computing similarity matrix...
Computing the msd similarity matrix...
Done computing similarity matrix...
Computing the pearson similarity matrix...
Done computing similarity matrix...
Computing the pearson similarity matrix...
Done computing similarity matrix...
Computing the pearson similarity matrix...
Done computing similarity matrix...
Evaluating RMSE, MAE of algorithm KNNBasic on 5 split(s).

RMSE (testset)    Fold 1    Fold 2    Fold 3    Fold 4    Fold 5    Mean    Std
MAE (testset)     0.8018    0.8083    0.7990    0.8001    0.8038    0.8026    0.0033
Fit time          0.75      0.73      0.72      0.84      1.17      0.84      0.17
Test time         3.56      4.61      3.34      4.23      3.27      3.80      0.53
Average RMSE for pearson similarity: 1.0112
Average MAE for pearson similarity: 0.8026
-----
Computing the msd similarity matrix...
Done computing similarity matrix...
Computing the msd similarity matrix...
Done computing similarity matrix...
Computing the msd similarity matrix...
Done computing similarity matrix...
Computing the msd similarity matrix...
Done computing similarity matrix...
Computing the msd similarity matrix...
Done computing similarity matrix...
Computing the msd similarity matrix...
Done computing similarity matrix...
Computing the msd similarity matrix...
Done computing similarity matrix...
Evaluating RMSE, MAE of algorithm KNNBasic on 5 split(s).

RMSE (testset)    Fold 1    Fold 2    Fold 3    Fold 4    Fold 5    Mean    Std
MAE (testset)     0.9773    0.9816    0.9754    0.9802    0.9819    0.9793    0.0025
Fit time          0.43      0.41      0.48      0.39      0.38      0.42      0.03
Test time         3.52      4.69      3.85      3.24      4.66      3.99      0.59
Average RMSE for msd similarity: 0.9793
Average MAE for msd similarity: 0.7734
-----
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix...
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix...
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix...
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix...
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix...
Evaluating RMSE, MAE of algorithm KNNBasic on 5 split(s).

Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix...
Evaluating RMSE, MAE of algorithm KNNBasic on 5 split(s).

RMSE (testset)    Fold 1    Fold 2    Fold 3    Fold 4    Fold 5    Mean    Std
MAE (testset)     0.7880    0.7941    0.7855    0.7891    0.7958    0.7905    0.0038
Fit time          0.97      0.99      1.20      1.06      1.00      1.04      0.08
Test time         3.19      3.25      4.65      3.23      4.42      3.75      0.65
Average RMSE for pearson_baseline similarity: 0.9992
Average MAE for pearson_baseline similarity: 0.7905
-----
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: 'should_run_async' will not call 'transform_cell' automatically
and should_run_async(code)

Computing the pearson_baseline similarity matrix...
Done computing similarity matrix...
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix...
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix...
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix...
Estimating biases using als...
Computing the pearson_baseline similarity matrix...
Done computing similarity matrix...
Average Precision: 0.6968
Average Recall: 0.7481
Average f1-score: 0.7178

```

[illegible]

SOLVING MAZE PROBLEM USING OPENAI GYM LIBRARY

AIM

To understand and implement a reinforcement learning solution to solve a maze problem using the OpenAI Gym library.

1. . Introduction to OpenAI Gym:
 - (a) Provide an overview of the OpenAI Gym library and its functionalities for reinforcement learning tasks.
 - (b) Explain the concept of environments, agents, actions, and observations in the context of Gym.
2. Setting up the Maze Environment:
 - (a)
 - (b) Define a custom maze environment using Gym that represents a maze with walls, a start point, and a goal point.
 - (c) Implement functions to initialize the environment, reset it to the starting state, and render the maze for visualization.
3. Defining Actions and Observations:
 - (a) Define the possible actions that an agent can take in the maze environment (e.g., move up, down, left, right).
 - (b) Determine the observations available to the agent at each state (e.g., current position, proximity to walls or goal).
4. Implementing Q-Learning Algorithm:
 - (a) Implement the Q-learning algorithm to train an agent to navigate through the maze environment.
 - (b) Define the Q-table to store Q-values for state-action pairs.
 - (c) Implement the exploration-exploitation trade-off strategy (e.g., epsilon-greedy) to balance exploration and exploitation during training.
5. Training the Agent:
 - (a) Train the agent using the Q-learning algorithm to learn an optimal policy for navigating the maze.
 - (b) Monitor the agent's learning progress by tracking rewards obtained during training episodes.

(c) Visualize the learned policy and the agent's trajectory through the maze.

6. Evaluation and Testing:

- (a) Evaluate the trained agent's performance by running episodes in the maze environment and measuring its success rate in reaching the goal.
- (b) Analyze the agent's behavior and performance under different conditions (e.g., varying maze sizes, obstacle configurations).

7. Extensions and Improvements:

- (a) Experiment with different hyperparameters (e.g., learning rate, discount factor) and observe their impact on the agent's learning and performance.
- (b) Explore advanced reinforcement learning techniques (e.g., deep Q-learning) for solving more complex maze environments.

PROGRAM

```
import gym
from gym import spaces
import numpy as np

class MazeEnv(gym.Env):
    def __init__(self, maze):
        super(MazeEnv, self).__init__()
        self.maze = maze
        self.start = (0, 0)
        self.goal = (len(maze) - 1, len(maze[0]) - 1)
        self.current_position = self.start

        # Define action space: up, down, left, right
        self.action_space = spaces.Discrete(4)

        # Define observation space: current position in the maze
        self.observation_space = spaces.Box(low=0, high=max(len(maze), len(maze[0])), shape=(2,))

    def reset(self):
        self.current_position = self.start
        return np.array(self.current_position, dtype=np.int32)

    def step(self, action):
        x, y = self.current_position
        if action == 0:    # Up
            x = max(x - 1, 0)
```

```
        elif action == 1: # Down
            x = min(x + 1, len(self.maze) - 1)
        elif action == 2: # Left
            y = max(y - 1, 0)
        elif action == 3: # Right
            y = min(y + 1, len(self.maze[0]) - 1)

        if self.maze[x][y] == 1: # Hit a wall
            x, y = self.current_position

        self.current_position = (x, y)
        done = self.current_position == self.goal
        reward = 1 if done else -0.1
        return np.array(self.current_position, dtype=np.int32), reward, done, {}

    def render(self):
        maze_copy = self.maze.copy()
        x, y = self.current_position
        maze_copy[x][y] = 2
        for row in maze_copy:
            print(' '.join(str(cell) for cell in row))

maze = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 0, 1, 0]]
env = MazeEnv(maze)
import random
from collections import defaultdict

class QLearningAgent:
    def __init__(self, env, learning_rate=0.1, discount_factor=0.99, epsilon=0.1):
        self.env = env
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.epsilon = epsilon
        self.q_table = defaultdict(lambda: np.zeros(env.action_space.n))

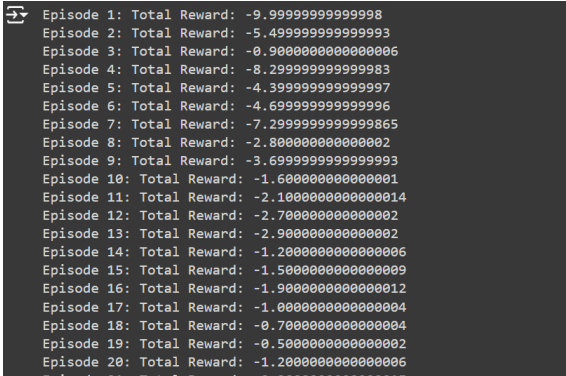
    def choose_action(self, state):
        if random.uniform(0, 1) < self.epsilon:
```

```
        return self.env.action_space.sample()
    else:
        return np.argmax(self.q_table[state])

    def learn(self, state, action, reward, next_state):
        best_next_action = np.argmax(self.q_table[next_state])
        td_target = reward + self.discount_factor * self.q_table[next_state][best_next_act
        td_error = td_target - self.q_table[state][action]
        self.q_table[state][action] += self.learning_rate * td_error

num_episodes = 1000
max_steps_per_episode = 100
agent = QLearningAgent(env)
for episode in range(num_episodes):
    state = tuple(env.reset())
    total_rewards = 0
    for step in range(max_steps_per_episode):
        action = agent.choose_action(state)
        next_state, reward, done, _ = env.step(action)
        next_state = tuple(next_state)
        agent.learn(state, action, reward, next_state)
        state = next_state
        total_rewards += reward
    if done:
        break
    print(f"Episode {episode + 1}: Total Reward: {total_rewards}")
```

SAMPLE INPUT-OUTPUT



```
Episode 1: Total Reward: -9.999999999999998
Episode 2: Total Reward: -5.4999999999999993
Episode 3: Total Reward: -0.9000000000000006
Episode 4: Total Reward: -8.2999999999999983
Episode 5: Total Reward: -4.3999999999999997
Episode 6: Total Reward: -4.6999999999999996
Episode 7: Total Reward: -7.2999999999999985
Episode 8: Total Reward: -2.8000000000000002
Episode 9: Total Reward: -3.6999999999999993
Episode 10: Total Reward: -1.6000000000000001
Episode 11: Total Reward: -2.10000000000000014
Episode 12: Total Reward: -2.7000000000000002
Episode 13: Total Reward: -2.9000000000000002
Episode 14: Total Reward: -1.2000000000000006
Episode 15: Total Reward: -1.5000000000000009
Episode 16: Total Reward: -1.90000000000000012
Episode 17: Total Reward: -1.0000000000000004
Episode 18: Total Reward: -0.7000000000000004
Episode 19: Total Reward: -0.5000000000000002
Episode 20: Total Reward: -1.2000000000000006
```

```
import random
from collections import defaultdict
class QLearningAgent:
    def __init__(self, env, learning_rate=0.1, discount_factor=0.99, epsilon=0.1):
```

```
        self.env = env
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.epsilon = epsilon
        self.q_table = defaultdict(lambda: np.zeros(env.action_space.n))

    def choose_action(self, state):
        if random.uniform(0, 1) < self.epsilon:
            return self.env.action_space.sample()
        else:
            return np.argmax(self.q_table[state])

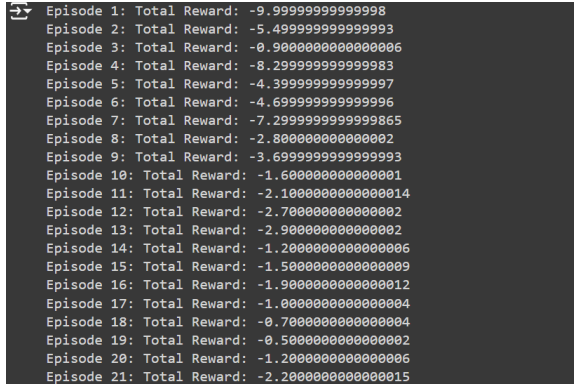
    def learn(self, state, action, reward, next_state):
        best_next_action = np.argmax(self.q_table[next_state])
        td_target = reward + self.discount_factor * self.q_table[next_state][best_next_act
        td_error = td_target - self.q_table[state][action]
        self.q_table[state][action] += self.learning_rate * td_error

num_episodes = 1000
max_steps_per_episode = 100
agent = QLearningAgent(env)
for episode in range(num_episodes):
    state = tuple(env.reset())
    total_rewards = 0
    for step in range(max_steps_per_episode):
        action = agent.choose_action(state)
        next_state, reward, done, _ = env.step(action)
        next_state = tuple(next_state)

        agent.learn(state, action, reward, next_state)

    state = next_state
    total_rewards += reward
    if done:
        break
    print(f"Episode {episode + 1}: Total Reward: {total_rewards}")
```

SAMPLE INPUT-OUTPUT

A terminal window with a dark background and light-colored text. It displays the results of 21 training episodes. Each line shows the episode number followed by 'Total Reward:' and a numerical value. The values range from approximately -9.99 to -2.20.

```
Episode 1: Total Reward: -9.999999999999998
Episode 2: Total Reward: -5.4999999999999993
Episode 3: Total Reward: -0.9000000000000006
Episode 4: Total Reward: -8.2999999999999983
Episode 5: Total Reward: -4.3999999999999997
Episode 6: Total Reward: -4.6999999999999996
Episode 7: Total Reward: -7.2999999999999865
Episode 8: Total Reward: -2.8000000000000002
Episode 9: Total Reward: -3.6999999999999993
Episode 10: Total Reward: -1.6000000000000001
Episode 11: Total Reward: -2.1000000000000014
Episode 12: Total Reward: -2.7000000000000002
Episode 13: Total Reward: -2.9000000000000002
Episode 14: Total Reward: -1.2000000000000006
Episode 15: Total Reward: -1.5000000000000009
Episode 16: Total Reward: -1.9000000000000012
Episode 17: Total Reward: -1.0000000000000004
Episode 18: Total Reward: -0.7000000000000004
Episode 19: Total Reward: -0.5000000000000002
Episode 20: Total Reward: -1.2000000000000006
Episode 21: Total Reward: -2.2000000000000015
```

```
state = tuple(env.reset())
done = False
env.render()
while not done:
    action = np.argmax(agent.q_table[state])
    state, _, done, _ = env.step(action)
    state = tuple(state)
    env.render()
success_count = 0
num_test_episodes = 100

for episode in range(num_test_episodes):
    state = tuple(env.reset())
    done = False
    while not done:
        action = np.argmax(agent.q_table[state])
        state, _, done, _ = env.step(action)
        state = tuple(state)

    if state == env.goal:
        success_count += 1
success_rate = success_count / num_test_episodes
print(f"Success Rate: {success_rate * 100}%")
# Define different maze configurations
maze_small = [
    [0, 0, 0],
    [0, 1, 0],
    [0, 0, 0]
]

maze_large = [
    [0, 1, 0, 0, 0],
```

```
[0, 1, 0, 1, 0],
[0, 0, 0, 1, 0],
[0, 1, 0, 0, 0],
[0, 0, 0, 1, 0]
]

# Optimized test agent performance function
def test_agent_performance(env, num_episodes=50, max_steps_per_episode=50):
    agent = QLearningAgent(env)
    success_count = 0

    for episode in range(num_episodes):
        state = tuple(env.reset())
        done = False

        for step in range(max_steps_per_episode):
            action = agent.choose_action(state)
            next_state, reward, done, _ = env.step(action)
            next_state = tuple(next_state)

            agent.learn(state, action, reward, next_state)
            state = next_state

            if done:
                break

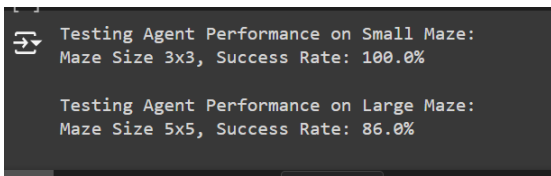
        if state == env.goal:
            success_count += 1

    success_rate = success_count / num_episodes * 100
    print(f"Maze Size {len(env.maze)}x{len(env.maze[0])}, Success Rate: {success_rate}%")

# Test on different maze configurations
env_small = MazeEnv(maze_small)
env_large = MazeEnv(maze_large)
print("Testing Agent Performance on Small Maze:")
test_agent_performance(env_small)

print("\nTesting Agent Performance on Large Maze:")
test_agent_performance(env_large)
```

SAMPLE INPUT-OUTPUT



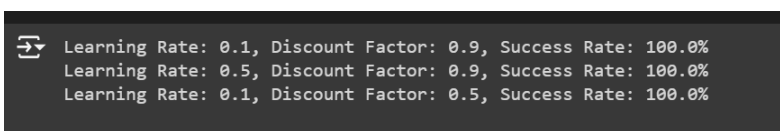
```
Testing Agent Performance on Small Maze:
Maze Size 3x3, Success Rate: 100.0%

Testing Agent Performance on Large Maze:
Maze Size 5x5, Success Rate: 86.0%
```

```
def test_hyperparameters(learning_rate, discount_factor, num_episodes=100):
    env = MazeEnv(maze_large)
    agent = QLearningAgent(env, learning_rate=learning_rate, discount_factor=discount_factor)
    success_count = 0
    for episode in range(num_episodes):
        state = tuple(env.reset())
        done = False
        while not done:
            action = agent.choose_action(state)
            next_state, reward, done, _ = env.step(action)
            next_state = tuple(next_state)
            agent.learn(state, action, reward, next_state)
            state = next_state
        if state == env.goal:
            success_count += 1
    success_rate = success_count / num_episodes * 100
    print(f"Learning Rate: {learning_rate}, Discount Factor: {discount_factor}, Success Rate: {success_rate}%")

# Test different hyperparameters
test_hyperparameters(learning_rate=0.1, discount_factor=0.9)
test_hyperparameters(learning_rate=0.5, discount_factor=0.9)
test_hyperparameters(learning_rate=0.1, discount_factor=0.5)
```

SAMPLE INPUT-OUTPUT



```
Learning Rate: 0.1, Discount Factor: 0.9, Success Rate: 100.0%
Learning Rate: 0.5, Discount Factor: 0.9, Success Rate: 100.0%
Learning Rate: 0.1, Discount Factor: 0.5, Success Rate: 100.0%
```

Deep Q-Learning Implementation

```
import torch
import torch.nn as nn
import torch.optim as optim
class QNetwork(nn.Module):
    def __init__(self, state_size, action_size):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(state_size, 64)
        self.fc2 = nn.Linear(64, 64)
```

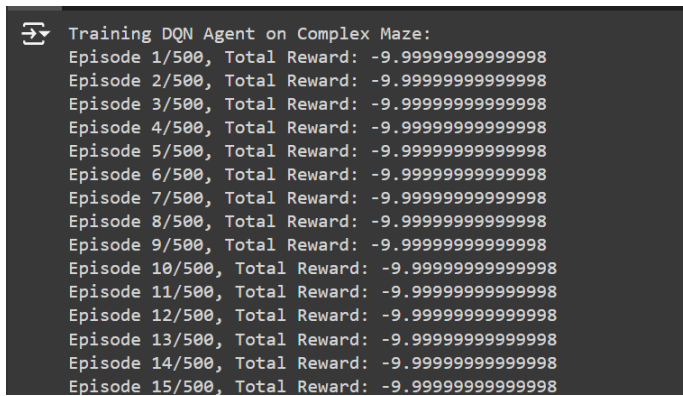
```
        self.fc3 = nn.Linear(64, action_size)
def forward(self, state):
    x = torch.relu(self.fc1(state))
    x = torch.relu(self.fc2(x))
    return self.fc3(x)
class DQNAgent:
def __init__(self, state_size, action_size, learning_rate=0.001, discount_factor=0.99,
    self.state_size = state_size
    self.action_size = action_size
    self.learning_rate = learning_rate
    self.discount_factor = discount_factor
    self.epsilon = epsilon
    self.epsilon_decay = epsilon_decay
    self.epsilon_min = epsilon_min

    self.q_network = QNetwork(state_size, action_size)
    self.optimizer = optim.Adam(self.q_network.parameters(), lr=learning_rate)
    self.criterion = nn.MSELoss()
    self.memory = []
    self.batch_size = 64
def remember(self, state, action, reward, next_state, done):
    self.memory.append((state, action, reward, next_state, done))
    if len(self.memory) > 10000:
        self.memory.pop(0)
def choose_action(self, state):
    if np.random.rand() <= self.epsilon:
        return np.random.randint(self.action_size)
    state = torch.FloatTensor(state).unsqueeze(0)
    q_values = self.q_network(state)
    return torch.argmax(q_values).item()
def replay(self):
    if len(self.memory) < self.batch_size:
        return
    batch = random.sample(self.memory, self.batch_size)
    for state, action, reward, next_state, done in batch:
        state = torch.FloatTensor(state).unsqueeze(0)
        next_state = torch.FloatTensor(next_state).unsqueeze(0)
        q_update = reward
        if not done:
            q_update += self.discount_factor * torch.max(self.q_network(next_state)).item()
        q_values = self.q_network(state)
```

```
        q_values = q_values.squeeze(0)
        q_values[action] = q_update
        loss = self.criterion(self.q_network(state), q_values.unsqueeze(0))
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay
def train_dqn_agent(env, agent, num_episodes=500, max_steps_per_episode=100):
    for episode in range(num_episodes):
        state = env.reset()
        state = np.array(state)
        total_reward = 0
        for step in range(max_steps_per_episode):
            action = agent.choose_action(state)
            next_state, reward, done, _ = env.step(action)
            next_state = np.array(next_state)
            agent.remember(state, action, reward, next_state, done)
            agent.replay()
            state = next_state
            total_reward += reward
            if done:
                break
        print(f"Episode {episode + 1}/{num_episodes}, Total Reward: {total_reward}")
def test_dqn_agent(env, agent, num_test_episodes=50):
    success_count = 0
    for episode in range(num_test_episodes):
        state = env.reset()
        state = np.array(state)
        done = False
        while not done:
            action = agent.choose_action(state)
            next_state, reward, done, _ = env.step(action)
            state = np.array(next_state)
        if state.tolist() == list(env.goal):
            success_count += 1
    success_rate = success_count / num_test_episodes * 100
    print(f"Success Rate: {success_rate}%")
# Define the environment and agent
maze_complex = [
    [0, 1, 0, 0, 0],
```

```
[0, 1, 0, 1, 0],  
[0, 0, 0, 1, 0],  
[0, 1, 0, 0, 0],  
[0, 0, 0, 1, 0]]  
env_complex = MazeEnv(maze_complex)  
state_size = env_complex.observation_space.shape[0]  
action_size = env_complex.action_space.n  
dqn_agent = DQNAgent(state_size, action_size)  
print("Training DQN Agent on Complex Maze:")  
train_dqn_agent(env_complex, dqn_agent)  
print("\nTesting DQN Agent on Complex Maze:")  
test_dqn_agent(env_complex, dqn_agent)
```

SAMPLE INPUT-OUTPUT



```
Training DQN Agent on Complex Maze:  
Episode 1/500, Total Reward: -9.99999999999998  
Episode 2/500, Total Reward: -9.99999999999998  
Episode 3/500, Total Reward: -9.99999999999998  
Episode 4/500, Total Reward: -9.99999999999998  
Episode 5/500, Total Reward: -9.99999999999998  
Episode 6/500, Total Reward: -9.99999999999998  
Episode 7/500, Total Reward: -9.99999999999998  
Episode 8/500, Total Reward: -9.99999999999998  
Episode 9/500, Total Reward: -9.99999999999998  
Episode 10/500, Total Reward: -9.99999999999998  
Episode 11/500, Total Reward: -9.99999999999998  
Episode 12/500, Total Reward: -9.99999999999998  
Episode 13/500, Total Reward: -9.99999999999998  
Episode 14/500, Total Reward: -9.99999999999998  
Episode 15/500, Total Reward: -9.99999999999998  
  
Testing DQN Agent on Complex Maze:  
Success Rate: 100.0 percent
```

PARALLEL IMAGE PROCESSING WITH OPENMP

AIM

To understand and implement parallel image processing operations using OpenMP, focusing on loading an image and performing matrix operations on it.

1. Introduction to OpenMP:

- (a) Provide an overview of OpenMP and its usage for parallel programming
- (b) Explain the concept of parallelism, threads, and parallel regions in OpenMP.

2. Loading Image:

- (a) Write a function to load an image from a file into a matrix representation.
- (b) Choose a common image format (e.g., JPEG, PNG) and use appropriate libraries (e.g., OpenCV) for image loading.
- (c) Display the loaded image for visualization.

3. Image Processing Operations:

- (a) Implement the following image processing operations as matrix operations:
 - i. Image Blurring: Apply a blur filter to the image using a convolution matrix/kernel.
 - ii. Image Sharpening: Apply a sharpening filter to the image using a convolution matrix/kernel.
 - iii. Image Edge Detection: Apply an edge detection filter (e.g., Sobel operator) to detect edges in the image.
 - iv. Parallelize each of the matrix operations using OpenMP directives (e.g., `#pragma omp parallel for`) to exploit parallelism.

4. Performance Analysis:

- (a) Measure the execution time of each image processing operation with and without parallelization.
- (b) Compare the performance improvement achieved by parallelizing the operations using OpenMP.
- (c) Analyze the speedup and efficiency achieved by parallelization.

5. Visualization and Output:

- (a) Display the processed images after each image processing operation for visualization.

- (b) Save the processed images to files for further analysis and comparison.

PROGRAM

CODE

SAMPLE INPUT-OUTPUT

```
Here are the calculated grades :  
ROLL NO :11  
NAME    :Dev  
SCORE   :98.6667  
GRADE   :A
```

```
ROLL NO :12  
NAME    :Raju  
SCORE   :72.5  
GRADE   :B
```

```
ROLL NO :13  
NAME    :Ayan  
SCORE   :63.0267  
GRADE   :B
```


FEATURE EXTRACTION

AIM

Perform feature extraction on iris dataset using LDA, PCA, TSNE and SVD to reduce it to,

1. Two features
2. Three features and apply cross validation in each case and report your inference

PROGRAM

```
import numpy as np
import pandas as pd
from sklearn import datasets
from sklearn.decomposition import PCA, TruncatedSVD
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.manifold import TSNE
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
import warnings

warnings.filterwarnings("ignore")

iris = datasets.load_iris()
X = iris.data
y = iris.target
def cross_validation(model, X, y, cv=5):
    clf = RandomForestClassifier()
    scores = cross_val_score(clf, X, y, cv=cv)
    return scores.mean()

# Perform feature extraction and cross-validation for each method and each case
results = {}

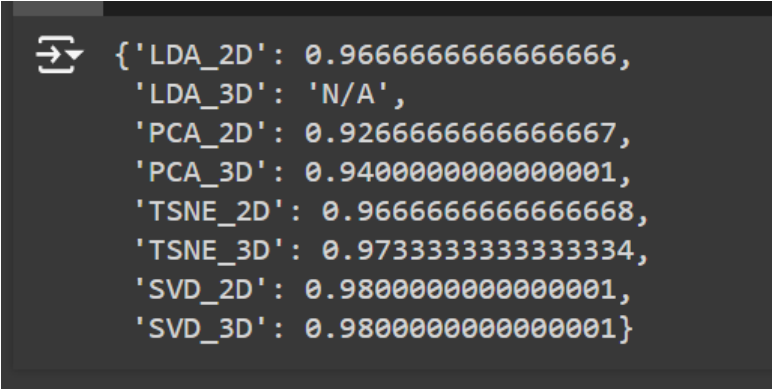
# LDA - Linear Discriminant Analysis (can only be applied for 2 features because it depend
lda_2d = LinearDiscriminantAnalysis(n_components=2).fit_transform(X, y)
lda_3d = LinearDiscriminantAnalysis(n_components=2).fit_transform(X, y) # LDA can only re

# PCA - Principal Component Analysis
pca_2d = PCA(n_components=2).fit_transform(X)
pca_3d = PCA(n_components=3).fit_transform(X)

# TSNE - t-Distributed Stochastic Neighbor Embedding
tsne_2d = TSNE(n_components=2).fit_transform(X)
```

```
tsne_3d = TSNE(n_components=3).fit_transform(X)
# SVD - Singular Value Decomposition
svd_2d = TruncatedSVD(n_components=2).fit_transform(X)
svd_3d = TruncatedSVD(n_components=3).fit_transform(X)
# Perform cross-validation
results['LDA_2D'] = cross_validation(RandomForestClassifier(), lda_2d, y)
results['LDA_3D'] = "N/A" # LDA cannot produce 3 features for this dataset
results['PCA_2D'] = cross_validation(RandomForestClassifier(), pca_2d, y)
results['PCA_3D'] = cross_validation(RandomForestClassifier(), pca_3d, y)
results['TSNE_2D'] = cross_validation(RandomForestClassifier(), tsne_2d, y)
results['TSNE_3D'] = cross_validation(RandomForestClassifier(), tsne_3d, y)
results['SVD_2D'] = cross_validation(RandomForestClassifier(), svd_2d, y)
results['SVD_3D'] = cross_validation(RandomForestClassifier(), svd_3d, y)
# Print results
results
```

SAMPLE INPUT-OUTPUT



```
{'LDA_2D': 0.9666666666666666,
 'LDA_3D': 'N/A',
 'PCA_2D': 0.9266666666666667,
 'PCA_3D': 0.9400000000000001,
 'TSNE_2D': 0.9666666666666668,
 'TSNE_3D': 0.9733333333333334,
 'SVD_2D': 0.9800000000000001,
 'SVD_3D': 0.9800000000000001}
```

THREAD CREATION

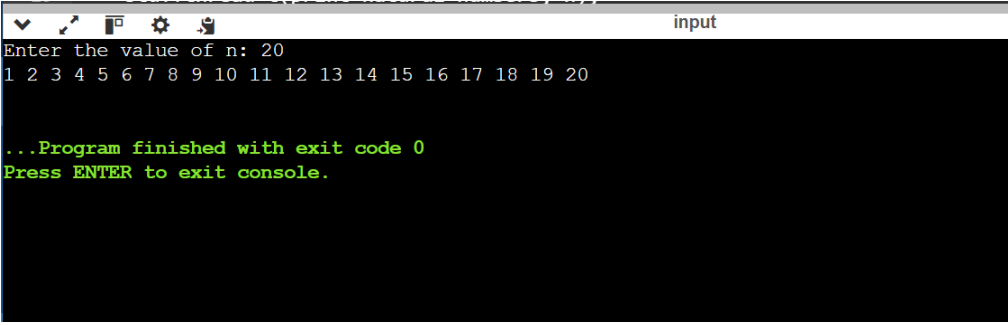
AIM

Write a C/C++ program to create a thread to print first n natural numbers

PROGRAM

```
#include <iostream>
#include <thread>
void print_natural_numbers(int n) {
    for (int i = 1; i <= n; ++i) {
        std::cout << i << " ";
    }
    std::cout << std::endl;
}
int main() {
    int n;
    std::cout << "Enter the value of n: ";
    std::cin >> n;
    // Thread Creation
    std::thread t(print_natural_numbers, n);
    // Wait for the thread to complete
    t.join();
    return 0;
}
```

SAMPLE INPUT-OUTPUT

A screenshot of a terminal window titled 'input'. The prompt 'Enter the value of n: 20' is shown. Below it, the output '1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20' is displayed. At the bottom, green text indicates '...Program finished with exit code 0' and 'Press ENTER to exit console.'

```
input
Enter the value of n: 20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
...Program finished with exit code 0
Press ENTER to exit console.
```

PERFORMANCE EVALUATION OF SEQUENTIAL AND THREAD EXECUTION PROGRAMS

AIM

Write a program in C/C++ to create a random integer array of size n and evaluate the performance in terms of execution time.

1. Sequential program with functions
 - (a) Find the sum of elements in an array
 - (b) Search a key element in an array
2. A thread based program to partition the array and perform computation in each thread.
 - (a) Find the sum of elements in an array
 - (b) Search a key element in an array

PROGRAM

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <chrono>

std::vector<int> generateRandomArray(int n) {
    std::vector<int> array(n);
    for(int i = 0; i < n; ++i) {
        array[i] = rand() % 100; // Random numbers between 0 and 99
    }
    return array;
}

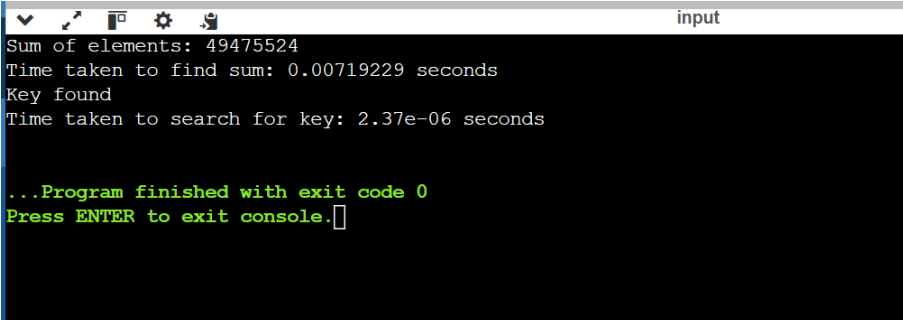
int findSum(const std::vector<int>& array) {
    int sum = 0;
    for(int num : array) {
        sum += num;
    }
    return sum;
}

bool searchKey(const std::vector<int>& array, int key) {
    for(int num : array) {
```

```
        if(num == key) {
            return true;
        }
    }
    return false;
}

int main() {
    int n = 1000000; // Size of the array
    int key = 50; // Key to search in the array
    srand(time(0));
    std::vector<int> array = generateRandomArray(n);
    auto start = std::chrono::high_resolution_clock::now();
    int sum = findSum(array);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    std::cout << "Sum of elements: " << sum << std::endl;
    std::cout << "Time taken to find sum: " << duration.count() << " seconds" << std::endl;
    start = std::chrono::high_resolution_clock::now();
    bool found = searchKey(array, key);
    end = std::chrono::high_resolution_clock::now();
    duration = end - start;
    std::cout << "Key " << (found ? "found" : "not found") << std::endl;
    std::cout << "Time taken to search for key: " << duration.count() << " seconds" << std::endl;
    return 0;
}
```

SAMPLE INPUT-OUTPUT



```
Sum of elements: 49475524
Time taken to find sum: 0.00719229 seconds
Key found
Time taken to search for key: 2.37e-06 seconds

...Program finished with exit code 0
Press ENTER to exit console.
```

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <chrono>
#include <thread>
```

```
#include <mutex>

std::mutex mtx; // Mutex for thread synchronization
std::vector<int> generateRandomArray(int n) {
    std::vector<int> array(n);
    for(int i = 0; i < n; ++i) {
        array[i] = rand() % 100; // Random numbers between 0 and 99}
    return array;}

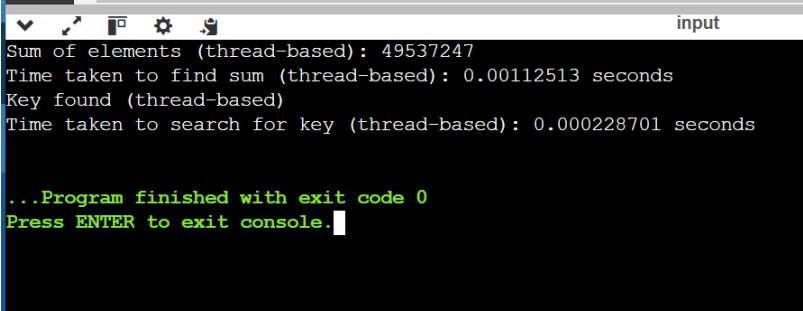
void findSumPartition(const std::vector<int>& array, int start, int end, int& result) {
    int sum = 0;
    for(int i = start; i < end; ++i) {
        sum += array[i];}
    mtx.lock();
    result += sum;
    mtx.unlock();}

void searchKeyPartition(const std::vector<int>& array, int start, int end, int key, bool& found) {
    for(int i = start; i < end; ++i) {
        if(array[i] == key) {
            mtx.lock();
            found = true;
            mtx.unlock();
            return;}}}

int main() {
    int n = 1000000; // Size of the array
    int key = 50; // Key to search in the array
    int numThreads = std::thread::hardware_concurrency(); // Number of threads
    srand(time(0));
    std::vector<int> array = generateRandomArray(n);
    int sum = 0;
    std::vector<std::thread> threads;
    int partitionSize = n / numThreads;
    auto start = std::chrono::high_resolution_clock::now();
    for(int i = 0; i < numThreads; ++i) {
        int startIdx = i * partitionSize;
        int endIdx = (i == numThreads - 1) ? n : startIdx + partitionSize;
        threads.push_back(std::thread(findSumPartition, std::cref(array), startIdx, endIdx, sum));
    }
    for(auto& th : threads) {
        th.join();
    }
    auto end = std::chrono::high_resolution_clock::now();
```

```
std::chrono::duration<double> duration = end - start;
std::cout << "Sum of elements (thread-based): " << sum << std::endl;
std::cout << "Time taken to find sum (thread-based): " << duration.count() << " second
// Thread-based key search
bool found = false;
threads.clear();
start = std::chrono::high_resolution_clock::now();
for(int i = 0; i < numThreads; ++i) {
    int startIdx = i * partitionSize;
    int endIdx = (i == numThreads - 1) ? n : startIdx + partitionSize;
    threads.push_back(std::thread(searchKeyPartition, std::cref(array), startIdx, endI
}
for(auto& th : threads) {
    th.join();
}
end = std::chrono::high_resolution_clock::now();
duration = end - start;
std::cout << "Key " << (found ? "found" : "not found") << " (thread-based)" << std::en
std::cout << "Time taken to search for key (thread-based): " << duration.count() << "
return 0;
}
```

SAMPLE INPUT-OUTPUT

A screenshot of a terminal window with a dark background. The window title is "input". The output text is as follows:

```
Sum of elements (thread-based): 49537247
Time taken to find sum (thread-based): 0.00112513 seconds
Key found (thread-based)
Time taken to search for key (thread-based): 0.000228701 seconds

...Program finished with exit code 0
Press ENTER to exit console.
```

IMPLEMENTING CLUSTERING TECHNIQUES ON IRIS DATASET

AIM

Implement K-means, K-medoid and Fuzzy C-means using IRIS dataset.

PROGRAM

K-Means

```
import numpy as np
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
iris = load_iris()
data = iris.data

def initialize_centroids(data, k):
    indices = np.random.choice(data.shape[0], k, replace=False)
    return data[indices]

def assign_clusters(data, centroids):
    clusters = []
    for point in data:
        distances = np.linalg.norm(point - centroids, axis=1)
        clusters.append(np.argmin(distances))
    return np.array(clusters)

def update_centroids(data, clusters, k):
    new_centroids = []
    for i in range(k):
        cluster_points = data[clusters == i]
        new_centroids.append(cluster_points.mean(axis=0))
    return np.array(new_centroids)

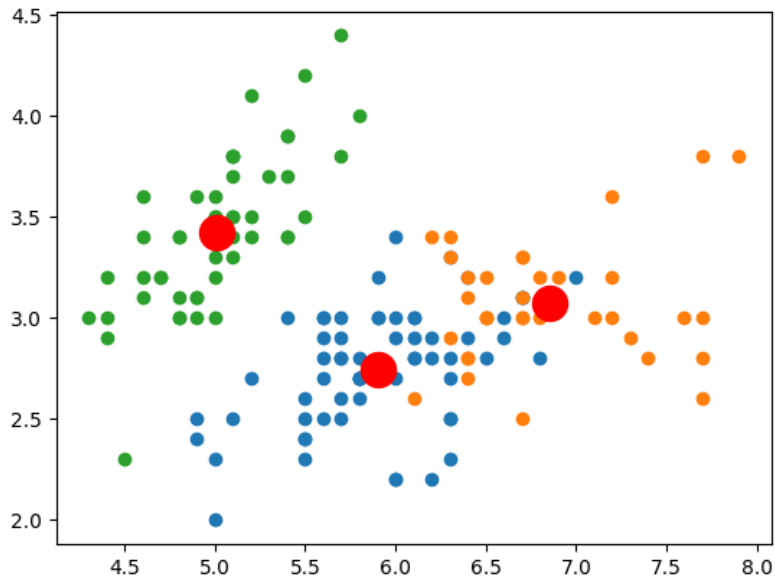
def kmeans(data, k, iterations=100):
    centroids = initialize_centroids(data, k)
    for _ in range(iterations):
        clusters = assign_clusters(data, centroids)
        centroids = update_centroids(data, clusters, k)
    return clusters, centroids

k = 3
clusters, centroids = kmeans(data, k)
# Plotting
for i in range(k):
    cluster_points = data[clusters == i]
```



```
plt.scatter(cluster_points[:, 0], cluster_points[:, 1])
plt.scatter(centroids[:, 0], centroids[:, 1], s=300, c='red')
plt.show()
```

SAMPLE INPUT-OUTPUT



K-Medoid

```
import numpy as np
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
iris = load_iris()
data = iris.data

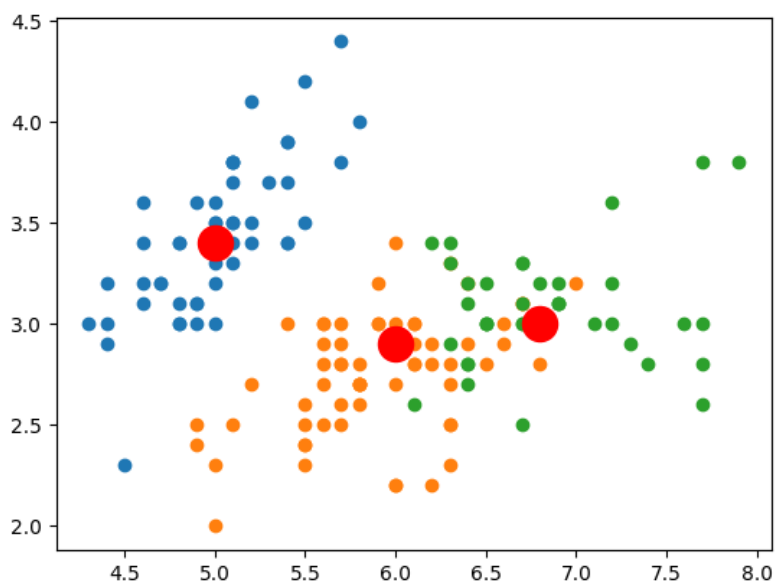
def initialize_medoids(data, k):
    indices = np.random.choice(data.shape[0], k, replace=False)
    return data[indices]

def assign_clusters(data, medoids):
    clusters = []
    for point in data:
        distances = np.linalg.norm(point - medoids, axis=1)
        clusters.append(np.argmin(distances))
    return np.array(clusters)

def update_medoids(data, clusters, k):
    new_medoids = []
    for i in range(k):
        cluster_points = data[clusters == i]
        distances = np.sum(np.linalg.norm(cluster_points[:, None] - cluster_points[None, :]
```

```
        new_medoids.append(cluster_points[np.argmin(distances)])
    return np.array(new_medoids)
def kmedoids(data, k, iterations=100):
    medoids = initialize_medoids(data, k)
    for _ in range(iterations):
        clusters = assign_clusters(data, medoids)
        medoids = update_medoids(data, clusters, k)
    return clusters, medoids
k = 3
clusters, medoids = kmedoids(data, k)
for i in range(k):
    cluster_points = data[clusters == i]
    plt.scatter(cluster_points[:, 0], cluster_points[:, 1])
plt.scatter(medoids[:, 0], medoids[:, 1], s=300, c='red')
plt.show()
```

SAMPLE INPUT-OUTPUT

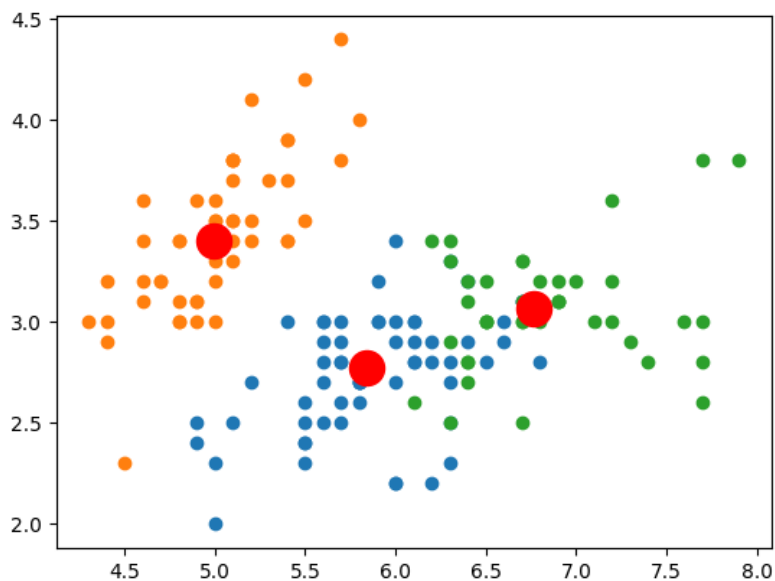


Fuzzy C-means

```
import numpy as np
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
iris = load_iris()
data = iris.data
def initialize_membership(data, k):
    membership = np.random.dirichlet(np.ones(k), size=data.shape[0])
```

```
    return membership
def update_centroids(data, membership, k, m):
    um = membership ** m
    return (um.T @ data) / np.sum(um, axis=0)[: , None]
def update_membership(data, centroids, k, m):
    distances = np.linalg.norm(data[:, None] - centroids, axis=2)
    inv_distances = 1 / distances
    inv_distances[inv_distances == np.inf] = 1e-10
    sum_inv_distances = np.sum(inv_distances, axis=1)
    return (inv_distances.T / sum_inv_distances).T ** (2 / (m - 1))
def fuzzy_cmeans(data, k, m=2, iterations=100):
    membership = initialize_membership(data, k)
    for _ in range(iterations):
        centroids = update_centroids(data, membership, k, m)
        membership = update_membership(data, centroids, k, m)
    return membership, centroids
k = 3
membership, centroids = fuzzy_cmeans(data, k)
clusters = np.argmax(membership, axis=1)
for i in range(k):
    cluster_points = data[clusters == i]
    plt.scatter(cluster_points[:, 0], cluster_points[:, 1])
plt.scatter(centroids[:, 0], centroids[:, 1], s=300, c='red')
plt.show()
```

SAMPLE INPUT-OUTPUT



EVALUATING CLASSIFIERS AND APPLYING ENSEMBLE METHODS

AIM

From sklearn, choose all classifiers and compute evaluation metrics of the iris dataset. After computation, choose the best 4 classifiers which gave the best performance. On these do ensemble methods such as bagging, boosting and stacking

PROGRAM

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split, cross_val_score, StratifiedKFold
from sklearn.metrics import accuracy_score
from sklearn.ensemble import BaggingClassifier, AdaBoostClassifier, StackingClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB

iris = load_iris()
X, y = iris.data, iris.target

classifiers = {
    'DecisionTree': DecisionTreeClassifier(),
    'KNeighbors': KNeighborsClassifier(),
    'LogisticRegression': LogisticRegression(max_iter=200),
    'SVC': SVC(probability=True),
    'RandomForest': RandomForestClassifier(),
    'GradientBoosting': GradientBoostingClassifier(),
    'GaussianNB': GaussianNB()
}

performance = {}
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)


for name, clf in classifiers.items():
    scores = cross_val_score(clf, X, y, cv=cv, scoring='accuracy')
```

```
performance[name] = np.mean(scores)
print(f"{name}: Cross-Validation Accuracy = {np.mean(scores):.4f}")

supported_classifiers = {
    name: clf for name, clf in classifiers.items() if hasattr(clf, 'fit') and hasattr
        (clf, 'predict')}
best_classifiers = [name for name in supported_classifiers if performance[name] ==
max(performance.values())][:4]
print(f"\nBest classifiers: {best_classifiers}")
selected_classifiers = [classifiers[name] for name in best_classifiers]
bagging_ensemble = BaggingClassifier(estimator=selected_classifiers[0], n_estimators=10,
random_state=42)
bagging_scores = cross_val_score(bagging_ensemble, X, y, cv=cv, scoring='accuracy')
print(f"Bagging Cross-Validation Accuracy: {np.mean(bagging_scores):.4f}")
boosting_ensemble = AdaBoostClassifier(estimator=DecisionTreeClassifier(),
n_estimators=50, random_state=42)
boosting_scores = cross_val_score(boosting_ensemble, X, y, cv=cv, scoring='accuracy')
print(f"Boosting Cross-Validation Accuracy: {np.mean(boosting_scores):.4f}")

estimators = [(name, clf) for name, clf in zip(best_classifiers, selected_classifiers)]
stacking_ensemble = StackingClassifier(estimators=estimators, final_estimator=
LogisticRegression())
stacking_scores = cross_val_score(stacking_ensemble, X, y, cv=cv, scoring='accuracy')
print(f"Stacking Cross-Validation Accuracy: {np.mean(stacking_scores):.4f}")
```

SAMPLE INPUT-OUTPUT



```
DecisionTree: Cross-Validation Accuracy = 0.9533
KNeighbors: Cross-Validation Accuracy = 0.9667
LogisticRegression: Cross-Validation Accuracy = 0.9667
SVC: Cross-Validation Accuracy = 0.9667
RandomForest: Cross-Validation Accuracy = 0.9600
GradientBoosting: Cross-Validation Accuracy = 0.9533
GaussianNB: Cross-Validation Accuracy = 0.9467

Best classifiers: ['KNeighbors', 'LogisticRegression', 'SVC']
Bagging Cross-Validation Accuracy: 0.9600
Boosting Cross-Validation Accuracy: 0.9533
Stacking Cross-Validation Accuracy: 0.9667
```