Pradnya Niketan Education Society's
NAGESH KARAJAGI *ORCHID* COLLEGE OF ENGINEERING & TECHNOLOGY, SOLAPUR
NAAC Accredited, Approved by AICTE, New Delhi & Affiliated to DBATU, Lonere
E-mail : office@orchidengg.ac.in, Website : www.orchidengg.ac.in, Phone No. 9423084363
Post Box No. 154, Gut No. 16, Solapur-Tuljapur Road, Tale Hipparaga, Solapur- 413 002.

Department of Artificial Intelligence & Data Science Engineering

LABORATORY MANUAL

Advance Machine Learning Lab

Semester: VI

Subject Code: BTAIL606

Prepared by,

Prof. N. B. Aherwadi

Pradnya Niketan Education Society, Pune.

NAGESH KARAJAGI *ORCHID* COLLEGE OF

ENGINEERING & TECHNOLOGY

SOLAPUR.

INDEX

Lab 1: Implementing K-means Clustering.

Aim: To Study of Implementation of K-means Clustering Algorithm.

Theory:

k-means clustering is a method of vector quantization, originally from signal processing, that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells.

k-means clustering minimizes within-cluster variances (squared Euclidean distances), but not regular Euclidean distances, which would be the more difficult Weber problem: the mean optimizes squared errors, whereas only the geometric median minimizes Euclidean distances. For instance, better Euclidean solutions can be found using k-medians and k-medoids.

Cluster analysis is a technique used in data mining and machine learning to group similar objects into clusters. K-means clustering is a widely used method for cluster analysis where the aim is to partition a set of objects into K clusters in such a way that the sum of the squared distances between the objects and their assigned cluster mean is minimized.

k -means is quite straightforward.

1. Decide how many clusters you want, i.e. choose k
2. Randomly assign a centroid to each of the k clusters
3. Calculate the distance of all observation to each of the k centroids
4. Assign observations to the closest centroid
5. Find the new location of the centroid by taking the mean of all the observations in each cluster
6. Repeat steps 3-5 until the centroids do not change position

Program and Output:

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Load the Iris dataset
iris = load_iris()
X = iris.data

# Instantiate the K-means clustering model with 3 clusters (because we know there are 3 species in the Iris dataset)
kmeans = KMeans(n_clusters=3, random_state=42)

# Fit the model to the data
kmeans.fit(X)
```

```
# Get the cluster assignments for each data point
labels = kmeans.labels_
# Visualize the clusters (assuming we can plot two features at a time for simplicity)
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.title('K-means Clustering on Iris Dataset')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

- The Iris dataset is loaded, and the features are stored in X.
- A K-means clustering model is instantiated with 3 clusters since we know there are 3 species in the Iris dataset.
- The model is fit to the data, and cluster assignments are obtained.
- The clusters are visualized based on the first two features.

Lab 2: Implementing Hierarchical Clustering.

Aim: Study of Implementation of Hierarchical Clustering Algorithm.

Theory:

Hierarchical clustering is a method used for grouping similar data points into clusters, forming a hierarchy or tree-like structure known as a dendrogram. This technique does not require the number of clusters to be specified beforehand. Hierarchical clustering can be broadly classified into two types: Agglomerative (bottom-up) and Divisive (top-down).

Agglomerative Hierarchical Clustering (Bottom-Up):
Agglomerative hierarchical clustering starts with each data point as a single cluster and merge the most similar clusters until only one cluster remains. The process can be summarized as follows:

Individual Data Points as Clusters: Treat each data point as a single cluster.
Merge Similar Clusters: Iteratively merge the two most similar clusters into a new cluster.
Continue Merging: Continue merging clusters until a single cluster containing all data points is formed.
The choice of the linkage method (how to measure the distance between clusters) influences the merging process. Common linkage methods include Ward, Single, Complete, and Average linkage.

Divisive Hierarchical Clustering (Top-Down):
Divisive hierarchical clustering starts with all data points in a single cluster and recursively divides the dataset into smaller clusters until each data point is in its own cluster.

Dendrogram:
A dendrogram is a tree-like diagram that represents the hierarchical structure of clusters. It illustrates the order and distances at which clusters are merged. The vertical lines in a dendrogram represent clusters, and the height of the line indicates the dissimilarity between the merged clusters.

Program and Output:

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram, linkage
import matplotlib.pyplot as plt

# Load the Iris dataset
iris = load_iris()
X = iris.data

# Instantiate the Agglomerative Hierarchical Clustering model with 3 clusters
agg_cluster = AgglomerativeClustering(n_clusters=3)
```

```
# Fit the model to the data
agg_labels = agg_cluster.fit_predict(X)

# Visualize the dendrogram
linkage_matrix = linkage(X, method='ward')  # Ward linkage is used to minimize variance within clusters
dendrogram(linkage_matrix)
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Sample Index')
plt.ylabel('Distance')
plt.show()

# Visualize the clusters (assuming we can plot two features at a time for simplicity)
plt.scatter(X[:, 0], X[:, 1], c=agg_labels, cmap='viridis')
plt.title('Hierarchical Clustering on Iris Dataset')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

- The Iris dataset is loaded, and the features are stored in X.
- Agglomerative Hierarchical Clustering is instantiated with 3 clusters.
- The model is fit to the data, and cluster assignments are obtained.
- A dendrogram is visualized to show the hierarchy of clustering.

Lab 3: Implementation of Apriori Algorithm.

Aim: Study of Implementation of Apriori Algorithm

Theory:

Apriori algorithm was the first algorithm that was proposed for frequent itemset mining. It was later improved by R Agarwal and R Srikant and came to be known as Apriori. This algorithm uses two steps "join" and "prune" to reduce the search space. It is an iterative approach to discover the most frequent itemsets.

Apriori says:

The probability that item I is not frequent is if:

P(I) < minimum support threshold, then I is not frequent.
P (I+A) < minimum support threshold, then I+A is not frequent, where A also belongs to itemset.
If an itemset set has value less than minimum support then all of its supersets will also fall below min support, and thus can be ignored. This property is called the Antimonotone property.
The steps followed in the Apriori Algorithm of data mining are:

Join Step: This step generates (K+1) itemset from K-itemsets by joining each item with itself.
Prune Step: This step scans the count of each item in the database. If the candidate item does not meet minimum support, then it is regarded as infrequent and thus it is removed. This step is performed to reduce the size of the candidate itemsets.

Steps In Apriori
Apriori algorithm is a sequence of steps to be followed to find the most frequent itemset in the given database. This data mining technique follows the join and the prune steps iteratively until the most frequent itemset is achieved. A minimum support threshold is given in the problem or it is assumed by the user.

#1) In the first iteration of the algorithm, each item is taken as a 1-itemsets candidate. The algorithm will count the occurrences of each item.

#2) Let there be some minimum support, min_sup (eg 2). The set of 1 – itemsets whose occurrence is satisfying the min sup are determined. Only those candidates which count more than or equal to min_sup, are taken ahead for the next iteration and the others are pruned.

#3) Next, 2-itemset frequent items with min_sup are discovered. For this in the join step, the 2-itemset is generated by forming a group of 2 by combining items with itself.

#4) The 2-itemset candidates are pruned using min-sup threshold value. Now the table will have 2 – itemsets with min-sup only.

#5) The next iteration will form 3 –itemsets using join and prune step. This iteration will follow antimonotone property where the subsets of 3-itemsets, that is the 2 –itemset subsets of each group fall in min_sup. If all 2-itemset subsets are frequent then the superset will be frequent otherwise it is pruned.

#6) Next step will follow making 4-itemset by joining 3-itemset with itself and pruning if its subset does not meet the min_sup criteria. The algorithm is stopped when the most frequent itemset is achieved.

Program and Output:

```
# Install mlxtend library if not already installed
# !pip install mlxtend

import pandas as pd
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori, association_rules

# Sample dataset (list of transactions)
dataset = [['Milk', 'Bread', 'Eggs'],
           ['Milk', 'Apple', 'Eggs', 'Cheese'],
           ['Bread', 'Apple', 'Cheese'],
           ['Bread', 'Eggs'],
           ['Milk', 'Bread', 'Apple', 'Cheese']]

# Convert the dataset to a one-hot encoded format
te = TransactionEncoder()
te_ary = te.fit(dataset).transform(dataset)
df = pd.DataFrame(te_ary, columns=te.columns_)

# Apply Apriori algorithm to find frequent itemsets
frequent_itemsets = apriori(df, min_support=0.2, use_colnames=True)

# Generate association rules
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.7)

# Display the results
print("Frequent Itemsets:")
print(frequent_itemsets)

print("\nAssociation Rules:")
print(rules)
```

Lab 4: Implementation of Market Basket Analysis.

Aim: Study of Implementation of Market Basket Analysis**.**

Theory:

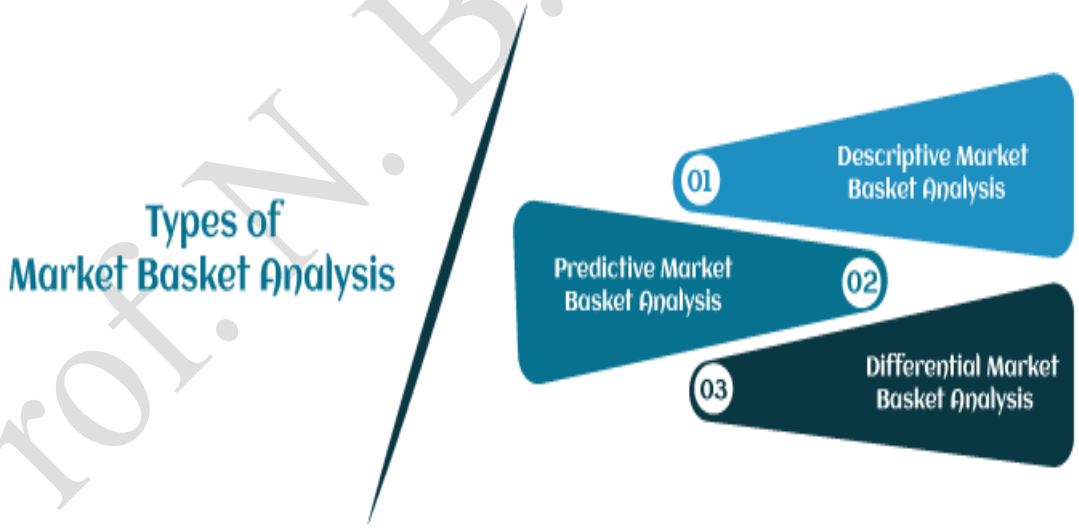
Market basket analysis is a data mining technique used by retailers to increase sales by better understanding customer purchasing patterns. It involves analyzing large data sets, such as purchase history, to reveal product groupings and products that are likely to be purchased together.

The adoption of market basket analysis was aided by the advent of electronic point-of-sale (POS) systems. Compared to handwritten records kept by store owners, the digital records generated by POS systems made it easier for applications to process and analyze large volumes of purchase data.

Implementation of market basket analysis requires a background in statistics and data science and some algorithmic computer programming skills. For those without the needed technical skills, commercial, off-the-shelf tools exist.

One example is the Shopping Basket Analysis tool in Microsoft Excel, which analyzes transaction data contained in a spreadsheet and performs market basket analysis. A transaction ID must relate to the items to be analyzed. The Shopping Basket Analysis tool then creates two worksheets:

The Shopping Basket Item Groups worksheet, which lists items that are frequently purchased together,
And the Shopping Basket Rules worksheet shows how items are related (For example, purchasers of Product A are likely to buy Product B).



Program and Output:

```
# Install mlxtend library if not already installed
# !pip install mlxtend

import pandas as pd
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori, association_rules
```

```
# Example dataset (list of transactions with different items)
dataset = [['Coffee', 'Bread', 'Butter'],
        ['Tea', 'Coffee', 'Sugar', 'Bread'],
        ['Milk', 'Sugar', 'Butter'],
        ['Tea', 'Coffee', 'Milk'],
        ['Tea', 'Bread', 'Butter']]

# Convert the dataset to a one-hot encoded format
te = TransactionEncoder()
te_ary = te.fit(dataset).transform(dataset)
df = pd.DataFrame(te_ary, columns=te.columns_)

# Apply Apriori algorithm to find frequent itemsets
frequent_itemsets = apriori(df, min_support=0.2, use_colnames=True)

# Generate association rules
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.7)

# Display the results
print("Frequent Itemsets:")
print(frequent_itemsets)

print("\nAssociation Rules:")
print(rules)
```

Lab 5: Reinforcement Learning
a. Calculating Reward
b. Discounted Reward
c. Calculating Optimal quantities
d. Implementing Q Learning
e. Setting up an Optimal Action.

Aim: To Study Reinforcement Learning - Calculating Reward, Discounted Reward, Calculating Optimal quantities, Implementing Q Learning, Setting up an Optimal Action.

Theory:

Reinforcement learning is a type of machine learning where an agent learns to make decisions by interacting with an environment.

a. Calculating Reward:

In reinforcement learning, the reward is a numerical value that the agent receives from the environment as feedback for its actions. It indicates how well the agent is performing. The goal of the agent is to maximize its cumulative reward over time. Rewards can be positive, negative, or zero.

b. Discounted Reward:

The discounted reward is used to give less importance to future rewards compared to immediate rewards. This is achieved by applying a discount factor (gamma) to future rewards.

c. Calculating Optimal Quantities:

In reinforcement learning, the agent aims to learn the optimal policy, which is a strategy or set of rules that maximizes the expected cumulative reward. The optimal policy provides the agent with the best action to take in each state.

d. Implementing Q Learning:

Q-Learning is a model-free reinforcement learning algorithm that learns the quality of actions in each state. The Q-value represents the expected cumulative reward of taking an action in a particular state and following the optimal policy thereafter.

e. Setting up an Optimal Action:

Once the Q-values are learned, the optimal action in a given state is the one with the highest Q-value. The agent selects the action that maximizes its expected cumulative reward.

Program and Output:

```
import numpy as np

# Q-learning parameters
alpha = 0.1  # learning rate
gamma = 0.9  # discount factor

# Q-table initialization
num_states = 5
num_actions = 3
Q = np.zeros((num_states, num_actions))

# Example transition
```

```
current_state = 0
action = 1
next_state = 2
reward = 1

# Q-learning update
Q[current_state, action] += alpha * (reward + gamma * np.max(Q[next_state, :]) - Q[current_state, action])

# Optimal action in a given state
optimal_action = np.argmax(Q[current_state, :])
```

Second Program:

```
import numpy as np
# Environment definition
num_states = 6
num_actions = 2

# Define the Q-table
Q = np.zeros((num_states, num_actions))

# Define the reward matrix (rows: current state, columns: actions)
R = np.array([
   [-1, -1],  # state 0
   [-1, -1],  # state 1
   [-1, -1],  # state 2
   [-1, -1],  # state 3
   [-1, -1],  # state 4
   [10, -1]   # state 5 (goal state)
])

# Parameters
alpha = 0.1  # learning rate
gamma = 0.9  # discount factor
epsilon = 0.1  # exploration-exploitation trade-off

# Q-learning
num_episodes = 1000

for episode in range(num_episodes):
   current_state = np.random.randint(0, num_states - 1)  # starting state

   while current_state != 5:  # goal state
      # Exploration-exploitation trade-off
      if np.random.uniform(0, 1) < epsilon:
         action = np.random.randint(0, num_actions)  # explore
      else:
```

```
            action = np.argmax(Q[current_state, :])  # exploit

        next_state = action if action == 1 else current_state + 1  # simple transition

        reward = R[current_state, action]

        # Q-learning update
        Q[current_state, action] += alpha * (reward + gamma * np.max(Q[next_state, :]) -
Q[current_state, action])

        current_state = next_state

# Optimal policy
optimal_policy = np.argmax(Q, axis=1)

print("Optimal Policy:")
print(optimal_policy)
```

Lab 6: Time Series Analysis
a.Checking Stationary
b. Converting a non-stationary data to stationary
c. Implementing Dickey Fuller Test
d. Plot ACF and PACF
e. Generating the ARIMA plot
f. TSA Forecasting.

Aim: Study of Time Series Analysis - Checking Stationary, Converting a non-stationary data to stationary, Implementing Dickey Fuller Test, Plot ACF and PACF, Generating the ARIMA plot, TSA Forecasting.

Theory:

a. Checking Stationarity:
Stationarity is a key assumption in time series analysis. A time series is considered stationary if its statistical properties do not change over time. You can check stationarity visually by plotting the time series data and also using statistical tests like the Augmented Dickey-Fuller (ADF) test.

b. Converting Non-Stationary Data to Stationary:
If the data is non-stationary, you can make it stationary by removing trends and seasonality. Common techniques include differencing (subtracting the previous value from the current value) and log transformations.

c. Implementing Dickey-Fuller Test:
The Dickey-Fuller test is a statistical test used to test the null hypothesis that a unit root is present in an autoregressive time series. A low p-value indicates that the time series is stationary.

d. Plotting ACF and PACF:
Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) plots help identify the order of autoregressive and moving average components in an ARIMA model.

e. Generating the ARIMA Plot:
ARIMA (AutoRegressive Integrated Moving Average) is a popular time series forecasting model. The ARIMA model is specified with three parameters: p (order of autoregression), d (degree of differencing), and q (order of moving average).

f. Time Series Analysis Forecasting:
Forecasting involves predicting future values of a time series based on historical data. ARIMA models, along with other advanced forecasting techniques, can be used for time series forecasting.

Program and Output:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA

# Generate a sample time series
```

```
np.random.seed(42)
time = np.arange(100)
data = np.random.randn(100) + 0.1 * time  # Non-stationary data with a trend
# Plot the time series
plt.plot(time, data)
plt.title('Non-Stationary Time Series')
plt.xlabel('Time')
plt.ylabel('Value')
plt.show()

# a. Checking Stationarity
def check_stationarity(ts):
    result = adfuller(ts)
    print('ADF Statistic:', result[0])
    print('p-value:', result[1])
    print('Critical Values:', result[4])

# b. Converting to Stationary (e.g., Differencing)
stationary_data = np.diff(data, n=1)

# c. Implementing Dickey-Fuller Test
print("\nDickey-Fuller Test for Stationarity:")
check_stationarity(stationary_data)

# d. Plotting ACF and PACF
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
plot_acf(stationary_data, ax=ax1, lags=20)
plot_pacf(stationary_data, ax=ax2, lags=20)
plt.show()

# e. Generating the ARIMA Plot
order = (1, 1, 1)  # Example ARIMA order
model = ARIMA(data, order=order)
results = model.fit()

# f. Time Series Analysis Forecasting
forecast_steps = 10
forecast_values = results.get_forecast(steps=forecast_steps).predicted_mean

# Plotting original data and forecast
plt.plot(time, data, label='Original Data')
plt.plot(np.arange(100, 100 + forecast_steps), forecast_values, label='Forecast')
plt.title('Time Series Forecasting with ARIMA')
plt.xlabel('Time')
plt.ylabel('Value')
plt.legend()
plt.show()
```

Lab 7: Boosting
a. Cross Validation
b. AdaBoost

Aim: To Study Boosting - Cross Validation, AdaBoost Algorithm implementation.

Theory, Program and Output:

a. Cross Validation:

Cross-validation is a technique used to assess the performance of a machine learning model. It involves splitting the dataset into multiple subsets, training the model on some subsets, and evaluating it on the remaining subset. Common types of cross-validation include k-fold cross-validation, stratified k-fold cross-validation, and leave-one-out cross-validation.

```
from sklearn.model_selection import cross_val_score, KFold
from sklearn.ensemble import RandomForestClassifier

# Example with Random Forest Classifier and k-fold cross-validation
X, y = ...  # Your feature matrix X and target vector y

# Create a Random Forest Classifier
rf_classifier = RandomForestClassifier()

# Perform k-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
cross_val_scores = cross_val_score(rf_classifier, X, y, cv=kf)

# Display cross-validation scores
print("Cross-Validation Scores:", cross_val_scores)
print("Mean Cross-Validation Score:", np.mean(cross_val_scores))
```

b. AdaBoost:

AdaBoost (Adaptive Boosting) is an ensemble learning technique that combines the predictions of multiple weak learners to create a strong learner. Weak learners are models that perform slightly better than random chance. AdaBoost assigns different weights to the training samples based on their classification errors and adjusts the weights at each iteration to focus on the misclassified samples.

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Example with AdaBoost Classifier
X, y = ...  # Your feature matrix X and target vector y

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a base weak learner (e.g., Decision Tree)
```

```
base_classifier = DecisionTreeClassifier(max_depth=1)

# Create an AdaBoost Classifier
adaboost_classifier = AdaBoostClassifier(base_classifier, n_estimators=50, random_state=42)

# Train the AdaBoost Classifier
adaboost_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = adaboost_classifier.predict(X_test)

# Evaluate the accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Conclusion:

Cross-validation is used to assess the performance of a machine learning model by dividing the dataset into multiple subsets.

AdaBoost is implemented using the AdaBoostClassifier from scikit-learn, and a base weak learner (e.g., Decision Tree) is specified.