EXPERIMENT - 4

AIM: Program to demonstrate Decision Tree – ID3 Algorithm

THEORY:

A Decision Tree is a predictive model used in supervised learning for both classification and regression problems. It represents decisions in the form of a tree-like structure where each internal node corresponds to a test on an attribute, each branch represents the outcome of the test, and each leaf node represents a class label (decision outcome).

It is primarily designed for classification tasks and employs a top-down, greedy approach to build the tree.

Working Principle of ID3

The ID3 algorithm builds the decision tree by recursively partitioning the dataset into subsets based on the attribute that provides the maximum Information Gain at each step. The process continues until all records are perfectly classified or no further attributes are available.

Entropy

Entropy is a measure of uncertainty or impurity in the dataset.
For a dataset S:

$$Entropy = \sum_{i=1}^{c} -P_i * log_2(P_{i)}$$

- where : pi is the proportion of class in dataset S

  Entropy = 0 → dataset is pure (all samples belong to one class).
  Entropy=1 → dataset is maximally impure (evenly split among classes).

Information Gain (IG)

Information Gain measures the reduction in entropy after splitting the dataset based on an attribute.

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_{v)}$$

where:

- S = original dataset,
- A = attribute,
- Sv = subset of S or which attribute A has value v

The attribute with the highest IG is selected for the split.

Steps of the ID3 Algorithm

1. **Compute the entropy** of the dataset for the target attribute.
2. **For each attribute**, compute the Information Gain with respect to the target attribute.
3. Select the attribute with the highest Information Gain as the decision node.
4. **Split the dataset** into subsets based on the selected attribute.
5. **Repeat recursively** for each subset until one of the stopping conditions is met:
    - All samples in a subset belong to the same class.
    - There are no remaining attributes.
    - The dataset is empty.

Advantages of ID3

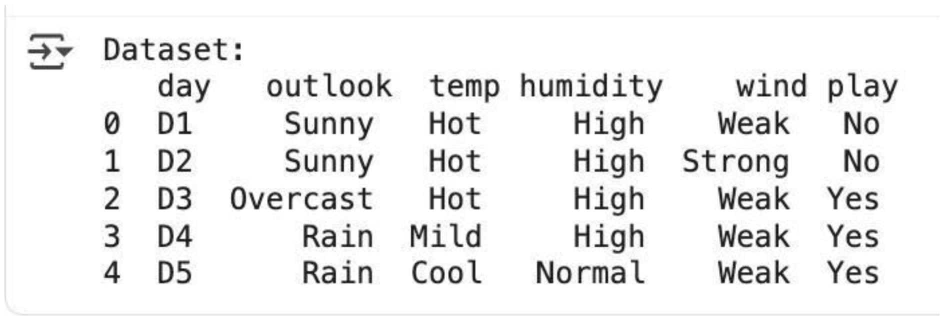
- Simple and easy to understand.
- Produces human-readable rules.
- Works well with categorical data.
- Fast to train on small datasets.

Limitations of ID3

- **Overfitting:** Can produce overly complex trees that do not generalize well.
- **Bias toward attributes with many values:** Attributes with more distinct values tend to have higher IG, which may not always be meaningful.
- Works best with **categorical data**; continuous data must be discretized.
- Sensitive to noise in the data.

CODE:

```
# import required libraries
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt

#load dataset
df = pd.read_csv("play_tennis.csv")
print("Dataset:\n", df.head())
```

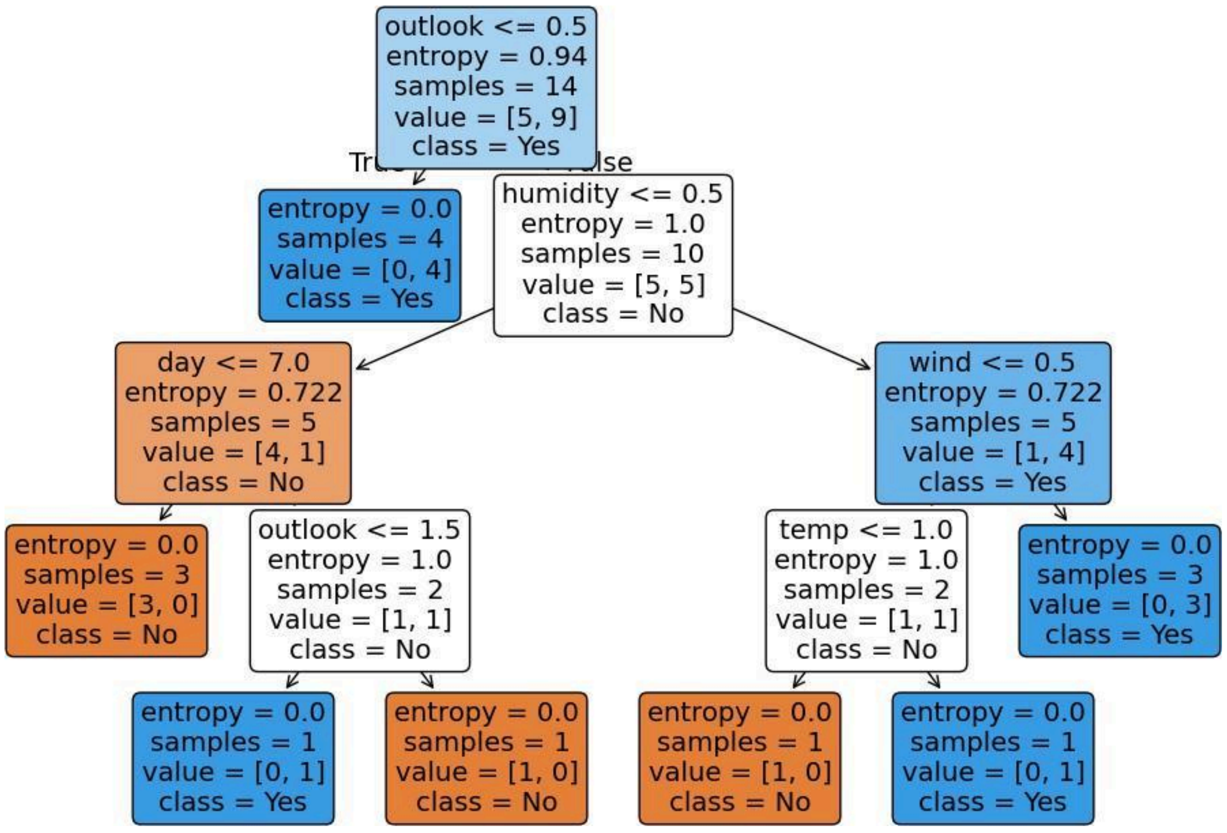


```
Dataset:
   day   outlook  temp humidity    wind play
0  D1     Sunny   Hot     High    Weak   No
1  D2     Sunny   Hot     High  Strong   No
2  D3  Overcast   Hot     High    Weak  Yes
3  D4      Rain  Mild     High    Weak  Yes
4  D5      Rain  Cool   Normal    Weak  Yes
```

```
# Encode categorical data
le = LabelEncoder()
for col in df.columns:
    df[col] = le.fit_transform(df[col])

# Features and Target
X = df.drop("play", axis=1)
```

```
y = df["play"]

# Build Decision Tree using ID3 (Entropy)
clf = DecisionTreeClassifier(criterion="entropy", random_state=0)
clf.fit(X, y)

#Plot the Decision Tree
plt.figure(figsize=(12,8)
)
plot_tree(clf, feature_names=X.columns, class_names=["No","Yes"],
          filled=True, rounded=True)
plt.show()
```

OUTPUT:



LEARNING OUTCOME:

EXPERIMENT - 5

AIM: Program to demonstrate PCA and LDA on Pima Diabetes dataset.

THEORY:

Principal Component Analysis

Principal Component Analysis (PCA) is one of the most widely used dimensionality reduction techniques in statistics and machine learning.
PCA is an unsupervised linear transformation technique that projects high-dimensional data into a lower-dimensional subspace while retaining as much variance (information) as possible.

Objectives of PCA

1. Dimensionality Reduction → Reduce the number of features while preserving maximum information.
2. Data Visualization → Represent high-dimensional data in 2D or 3D for easier interpretation.
3. Noise Reduction → Remove less informative features (low variance components).
4. Feature Extraction → Create new uncorrelated features (principal components).

Steps of PCA Algorithm

1. Standardize the dataset (mean = 0, variance = 1).
2. Compute the covariance matrix of the features.
3. Find eigenvalues and eigenvectors of the covariance matrix.
4. Sort eigenvectors by eigenvalues (descending order).
5. Select top k eigenvectors → these are the principal components.
6. Project the data onto the new subspace formed by these components.

Advantages of PCA

- Reduces dimensionality, improving efficiency.
- Removes multicollinearity by creating uncorrelated components.
- Helps visualization of high-dimensional data.
- Can act as a noise filter.

Limitations of PCA

- Assumes linear relationships (not suitable for nonlinear data).
- Difficult to interpret principal components (they are linear combinations of original features).
- Sensitive to scaling (standardization is necessary).
- May discard features that are important for prediction but have low variance.

Applications of PCA

- Image compression (reducing pixels while keeping structure).
- Face recognition (Eigenfaces).
- Gene expression analysis in bioinformatics.
- Data preprocessing before applying machine learning algorithms.
- Finance (reducing correlated stock market indicators).

Linear Discriminant Analysis (LDA) is a **supervised dimensionality reduction technique** used in statistics, pattern recognition, and machine learning.
Unlike PCA, which focuses on maximizing variance, **LDA aims to maximize class separability** by finding a linear combination of features that best separates two or more classes.

Objectives of LDA

- **Dimensionality Reduction** → Reduce the number of input features while preserving class-discriminatory information.
- **Class Separability** → Maximize the distance between class means and minimize the spread within each class.
- **Feature Extraction** → Create new axes (discriminant functions) that provide the best separation between known categories.

Steps of LDA Algorithm

1. Compute the mean vectors for each class in the dataset.
2. Compute the within-class scatter matrix (Sw) — measures how much the data points of each class vary around their mean.
3. Compute the between-class scatter matrix (Sb) — measures how far the class means are from the overall mean.
4. Compute the eigenvalues and eigenvectors of the matrix ((Sw)^-1).Sb
5. Select the top k eigenvectors corresponding to the largest eigenvalues → these form the new axes (Linear Discriminants).
6. Project the data onto the new subspace to obtain reduced-dimensional representation with maximum class separability.

Advantages of LDA

- Works well for classification problems with labeled data.
- Reduces overfitting by eliminating redundant features.
- Improves model performance when classes are linearly separable.
- Easier to interpret compared to PCA since it uses class information.

Limitations of LDA

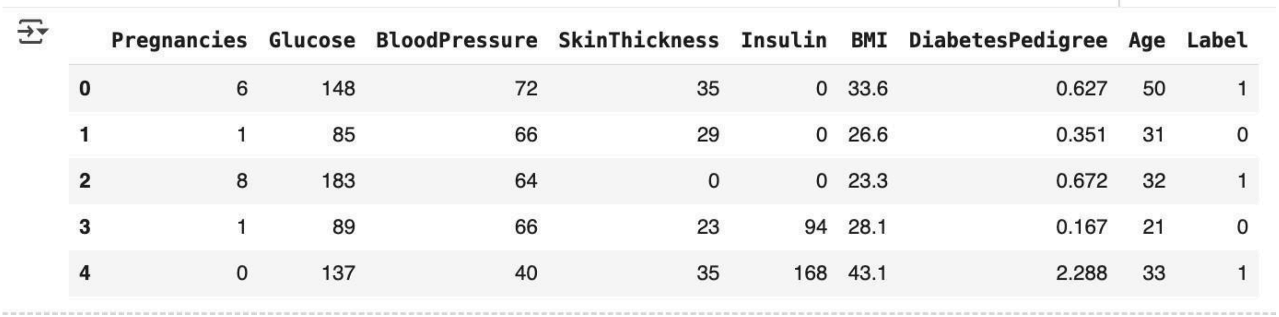
- Assumes normal distribution of features and equal covariance matrices across classes.
- Not suitable for nonlinear class boundaries.
- Can perform poorly if classes overlap significantly.
- Requires labeled data (supervised method).

Applications of LDA

- Face recognition (Fisherfaces method).
- Medical diagnosis (classifying healthy vs diseased cases).
- Financial risk analysis (predicting creditworthiness).
- Text classification and speech recognition tasks.
- Dimensionality reduction before applying supervised learning models.

CODE:

```
# Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import
train_test_split from sklearn.linear_model import
LogisticRegression from sklearn.metrics import
accuracy_score

# Set column names for Pima Indians Diabetes dataset and Load the dataset
col_names = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness',
'Insulin', 'BMI', 'DiabetesPedigree', 'Age', 'Label']
pima = pd.read_csv("pima-indians-diabetes.csv", header=None,
names=col_names)
pima.head()
```

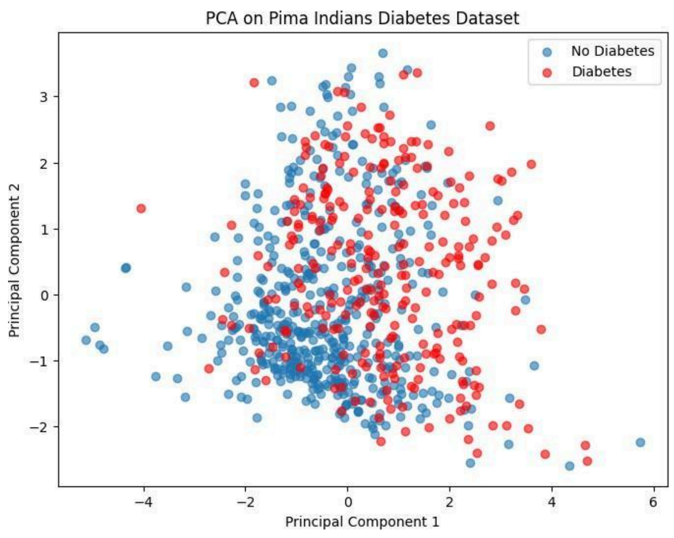


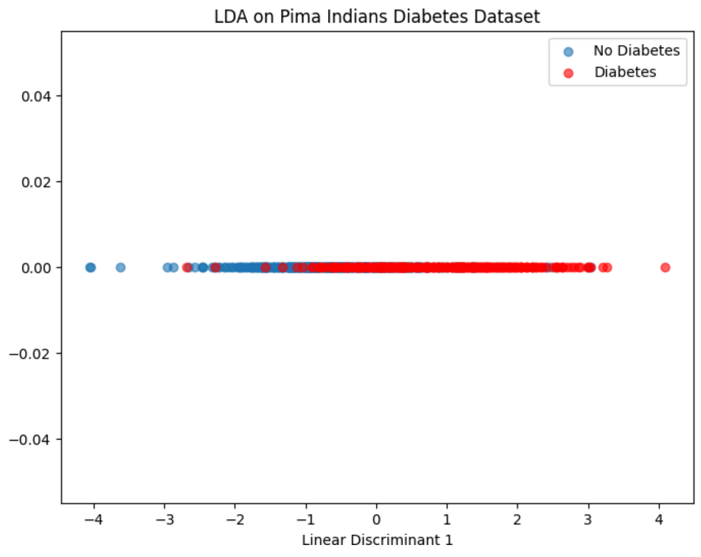
	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigree	Age	Label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

```
# Features and target
X = pima.drop("Label",
axis=1) y = pima["Label"]

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply PCA (reduce to 2 components for
visualization) pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Create DataFrame with PCA results
pca_df = pd.DataFrame(data=X_pca, columns=['PC1', 'PC2'])
pca_df['label'] = y.values

# Plot PCA result
plt.figure(figsize=(8,6))
plt.scatter(pca_df[pca_df['label']==0]['PC1'],
pca_df[pca_df['label']==0]['PC2'],
            label="No Diabetes", alpha=0.6)
plt.scatter(pca_df[pca_df['label']==1]['PC1'],
pca_df[pca_df['label']==1]['PC2'],
            label="Diabetes", alpha=0.6, color="red")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("PCA on Pima Indians Diabetes
Dataset") plt.legend()
```

```
plt.show()

# Apply LDA
lda = LDA(n_components=1)
X_lda = lda.fit_transform(X_scaled, y)

# Create DataFrame with LDA results
lda_df = pd.DataFrame(data=X_lda, columns=['LD1'])
lda_df['label'] = y.values

# Plot LDA result
plt.figure(figsize=(8,6))
plt.scatter(lda_df[lda_df['label']==0]['LD1'],
[0]*len(lda_df[lda_df['label']==0]),
            label="No Diabetes", alpha=0.6)
plt.scatter(lda_df[lda_df['label']==1]['LD1'],
[0]*len(lda_df[lda_df['label']==1]),
            label="Diabetes", alpha=0.6, color="red")
plt.xlabel("Linear Discriminant 1")
plt.title("LDA on Pima Indians Diabetes Dataset")
plt.legend()
plt.show()
```

OUTPUT:





LEARNING OUTCOME:

EXPERIMENT - 6

AIM: Program to demonstrate k-Nearest Neighbour - Breast Cancer Wisconsin (Diagnostic) Dataset

THEORY:

The k-Nearest Neighbors (k-NN) algorithm is a supervised learning technique used for both classification and regression tasks. It is one of the simplest and most intuitive machine learning algorithms based on the concept of similarity — objects that are close to each other (in feature space) are likely to belong to the same category.
In k-NN classification, a data point is assigned the class most common among its k nearest neighbors, where distance is usually measured using Euclidean distance. The value of k is a user-defined parameter that determines how many neighbors influence the classification.

Working of k-NN Algorithm

1. Load the Dataset: Collect training data with known class labels.
2. Choose the Value of k: Select the number of neighbors (usually an odd number to avoid ties).
3. Calculate Distance: For a new data point, compute the distance (commonly Euclidean) from all training samples.
4. Find Nearest Neighbors: Identify the k samples with the smallest distance values.
5. Voting Mechanism: Assign the class that occurs most frequently among these neighbors.
6. Prediction: Return this majority class as the predicted label for the test point.

Characteristics of k-NN

- Lazy Learner: No explicit model is trained; computation occurs at prediction time.
- Non-parametric: Makes no assumption about data distribution.
- Distance-based: Works effectively with continuous and normalized data.

Advantages

- Simple and easy to implement.
- Works well for small datasets.
- No prior training or model assumption required.
- Can adapt to complex decision boundaries.

Limitations

- Computationally expensive for large datasets (requires distance calculation for each test point).
- Sensitive to irrelevant or unscaled features.

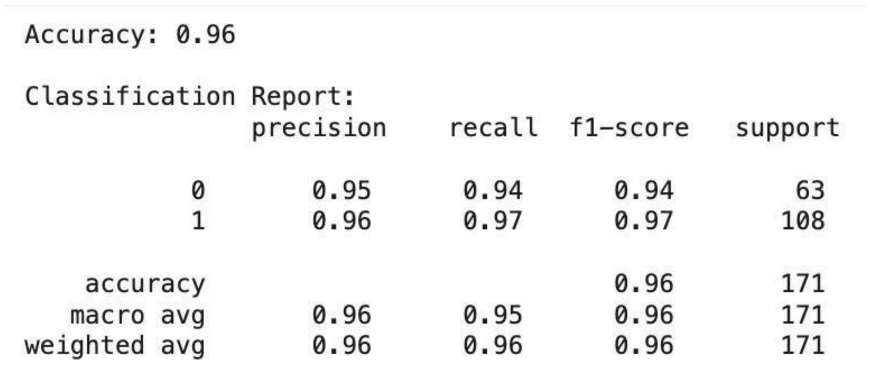
Application in This Experiment

The Breast Cancer Wisconsin dataset is used to classify tumors as:

- Benign (non-cancerous) or
- Malignant (cancerous)

CODE:

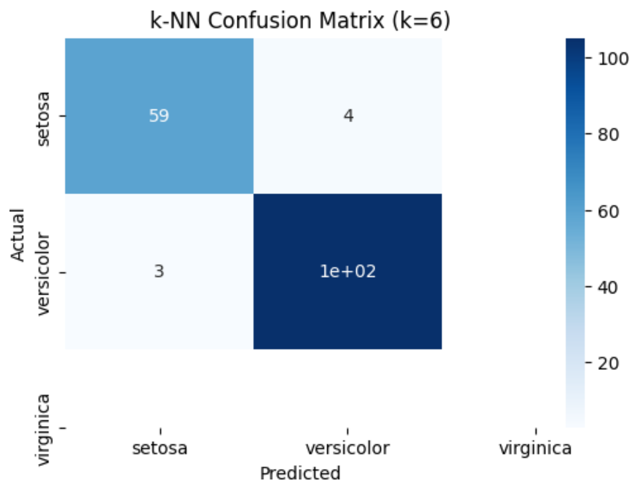
```
# Import libraries
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import
train_test_split from sklearn.preprocessing import
StandardScaler from sklearn.neighbors import
KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Load dataset
data = load_breast_cancer()
X, y = data.data, data.target

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Feature scaling
scaler = StandardScaler()
X_train =
scaler.fit_transform(X_train) X_test =
scaler.transform(X_test)

# Initialize and train KNN
model k = 6 # number of
neighbors
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)

# Make predictions
y_pred = knn.predict(X_test)

# Evaluate model
print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```



```
Accuracy: 0.96

Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.94      0.94        63
           1       0.96      0.97      0.97       108

    accuracy                           0.96       171
   macro avg       0.96      0.95      0.96       171
weighted avg       0.96      0.96      0.96       171
```

```
# Confusion matrix visualization
plt.figure(figsize=(6, 4))
sns.heatmap(confusion_matrix(y_test,
    y_pred),
            annot=True, cmap="Blues",
            xticklabels=iris.target_names,
            yticklabels=iris.target_names)
plt.title(f"k-NN Confusion Matrix
(k={k})") plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

OUTPUT:



LEARNING OUTCOME:

EXPERIMENT - 7

AIM: Program to demonstrate Naïve- Bayes Classifier

THEORY:

The Naïve Bayes Classifier is a supervised learning algorithm based on Bayes' Theorem. It is called "naïve" because it assumes that all features are independent of each other, which simplifies computation.

Bayes' Theorem

$$P(A/B) = \frac{P(A \; and \; B)}{P(B)}$$

Where:

- P(A|B) - Posterior probability of class C given features X
- P(B∣ A) - Likelihood of features given class
- P(B) - Probability of the features
- P(A) - Prior probability of the class

Working of Naïve Bayes

1. Calculate the prior probability of each class in the training data.
2. For each feature, calculate the likelihood of its value given each class.
3. Compute the posterior probability for each class using Bayes' theorem.
4. Assign the data point to the class with the highest posterior probability

Advantages

- Simple and fast
- Works well with high-dimensional data
- Performs well even with small datasets

Limitations

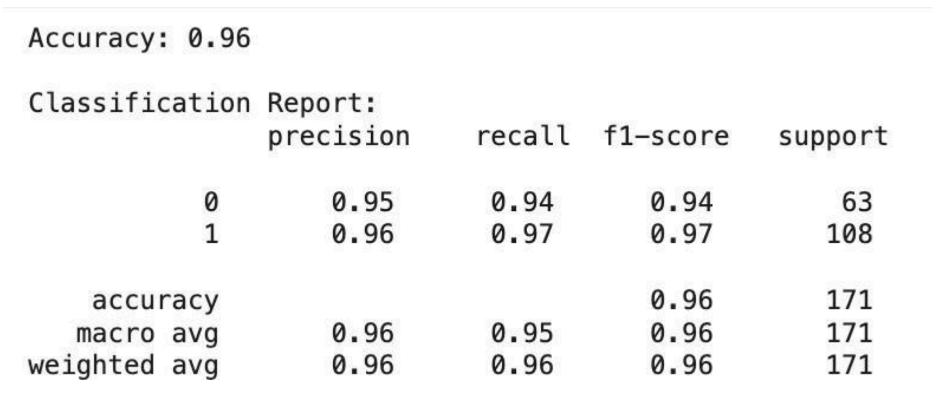
- Assumes feature independence
- Zero probability problem for unseen feature values (solved using Laplace smoothing)

Applications

- Email spam detection
- Medical diagnosis
- Document classification

CODE:

```
# Import Libraries
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import
train_test_split from sklearn.preprocessing import
StandardScaler from sklearn.naive_bayes import
GaussianNB
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Load Dataset
data = load_breast_cancer()
X, y = data.data, data.target
feature_names = data.feature_names
target_names = data.target_names

# Split Data into Training and Testing Sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Feature Scaling
scaler = StandardScaler()
X_train =
scaler.fit_transform(X_train) X_test =
scaler.transform(X_test)

# Initialize and Train Gaussian Naive Bayes Model
nb_model = GaussianNB()
nb_model.fit(X_train, y_train)

# Make Predictions
y_pred = nb_model.predict(X_test)

# Evaluate Model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}\n")
print("Classification Report:\n", classification_report(y_test, y_pred))
```

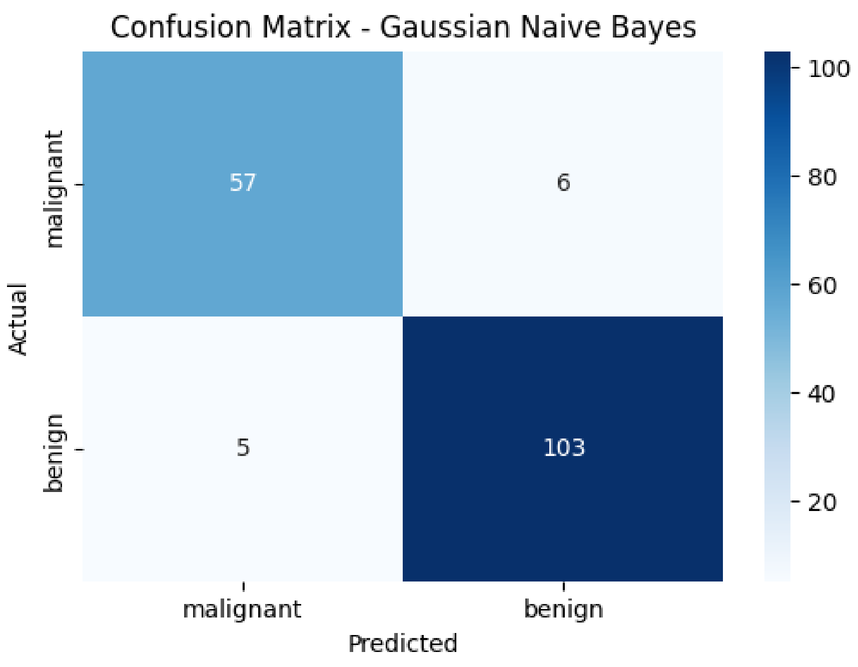


```
Accuracy: 0.96

Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.94      0.94        63
           1       0.96      0.97      0.97       108

    accuracy                           0.96       171
   macro avg       0.96      0.95      0.96       171
weighted avg       0.96      0.96      0.96       171
```

```
# Confusion Matrix Visualization
cm = confusion_matrix(y_test, y_pred)
```

```
plt.figure(figsize=(6,4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=target_names,
            yticklabels=target_names)
plt.title("Confusion Matrix - Gaussian Naive Bayes")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

OUTPUT:



LEARNING OUTCOME:

____
____
____
____
____

EXPERIMENT - 8

AIM: Program to demonstrate DBSCAN clustering algorithm

THEORY:

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an unsupervised machine learning algorithm used for clustering tasks. Unlike k-means, it does not require specifying the number of clusters beforehand and can detect clusters of arbitrary shape. It also identifies outliers/noise points that do not belong to any cluster.

Key Concepts

1. Epsilon (ε): Maximum distance between two points to be considered neighbors.
2. MinPts: Minimum number of points required to form a dense region (cluster).
3. Core Point: A point with at least MinPts neighbors within ε.
4. Border Point: A point within ε distance of a core point but with fewer than MinPts neighbors.
5. Noise Point: A point that is neither a core point nor a border point.

Working of DBSCAN

1. Randomly select an unvisited point.
2. Check if it is a core point (≥ MinPts neighbors within ε) .
   - Yes → form a new cluster and expand it by adding all reachable points.
   - No → mark it as noise (may later become a border point).
3. Repeat until all points are visited.

Advantages:

- Can find clusters of any shape
- Automatically detects outliers
- No need to specify number of clusters

Limitations:

- Sensitive to ε and MinPts parameters
- Not ideal for datasets with varying density clusters

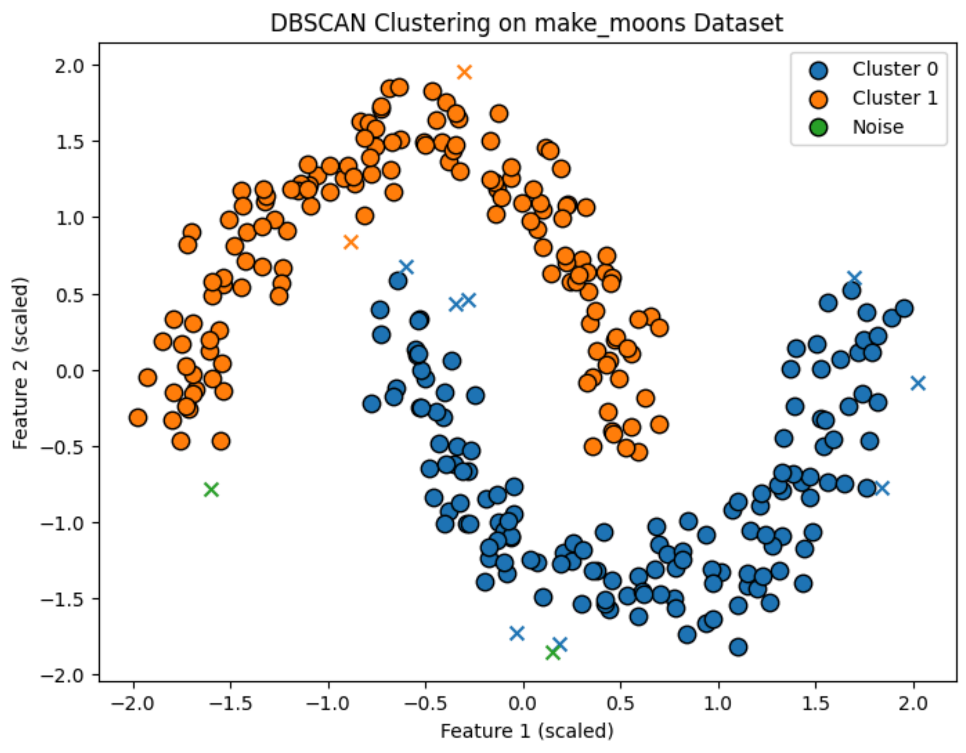
Applications

- Geospatial clustering (e.g., crime hotspots, GPS data)
- Anomaly detection
- Image segmentation

CODE:

```
# Import libraries
from sklearn.datasets import
make_moons from sklearn.cluster import
DBSCAN
from sklearn.preprocessing import
StandardScaler import matplotlib.pyplot as plt
import numpy as np

# Generate synthetic dataset
X, y_true = make_moons(n_samples=300, noise=0.1, random_state=42)

# Feature scaling (important for distance-based algorithms)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply DBSCAN
dbscan = DBSCAN(eps=0.3, min_samples=5) # eps: max distance, min_samples:
MinPts
clusters = dbscan.fit_predict(X_scaled)

# Identify core points
core_samples_mask = np.zeros_like(clusters, dtype=bool)
core_samples_mask[dbscan.core_sample_indices_] = True

# Plot the clusters
plt.figure(figsize=(8,6))
unique_labels = set(clusters)
colors = [plt.cm.tab10(each) for each in range(len(unique_labels))]

for k, col in zip(unique_labels, colors):
    class_member_mask = (clusters == k)
    xy = X_scaled[class_member_mask & core_samples_mask]
    plt.scatter(xy[:,0], xy[:,1], c=[col], marker='o', edgecolor='k',
s=80, label=f'Cluster {k}' if k != -1 else 'Noise')

    xy = X_scaled[class_member_mask & ~core_samples_mask]
    plt.scatter(xy[:,0], xy[:,1], c=[col], marker='x', s=50)

plt.title("DBSCAN Clustering on make_moons Dataset")
plt.xlabel("Feature 1 (scaled)")
plt.ylabel("Feature 2
(scaled)") plt.legend()
plt.show()
```

OUTPUT:



LEARNING OUTCOME:

EXPERIMENT - 9

AIM: Program to demonstrate K-Medoid clustering algorithm

THEORY:

K-Medoid clustering is an unsupervised machine learning algorithm used for partitioning datasets into clusters. It is very similar to k-means, but instead of using the mean of points as the cluster center, it uses actual data points (medoids) as centers, which makes it more robust to outliers.

Key Concepts

1. Medoid:
   A representative object of a cluster whose average dissimilarity to all other points in the cluster is minimal.
   Unlike k-means, medoids are actual points from the dataset.
2. Distance/Dissimilarity:
   Can use Euclidean, Manhattan, or other distance measures.
3. Cluster Assignment:
   Each point is assigned to the cluster with the closest medoid.
4. Update Medoid:
   For each cluster, select the point that minimizes total distance to other points in the cluster.

Working of K-Medoid Algorithm

1. Initialize k medoids randomly.
2. Assign each data point to the nearest medoid.
3. For each cluster, choose a new medoid that minimizes total distance.
4. Repeat steps 2–3 until medoids do not change or a stopping criterion is met.

Advantages

- Robust to outliers (unlike k-means).
- Works with any distance metric.
- Medoids are actual data points → easier to interpret.

Limitations

- Slower than k-means for large datasets (computationally expensive).
- Requires specifying k in advance.

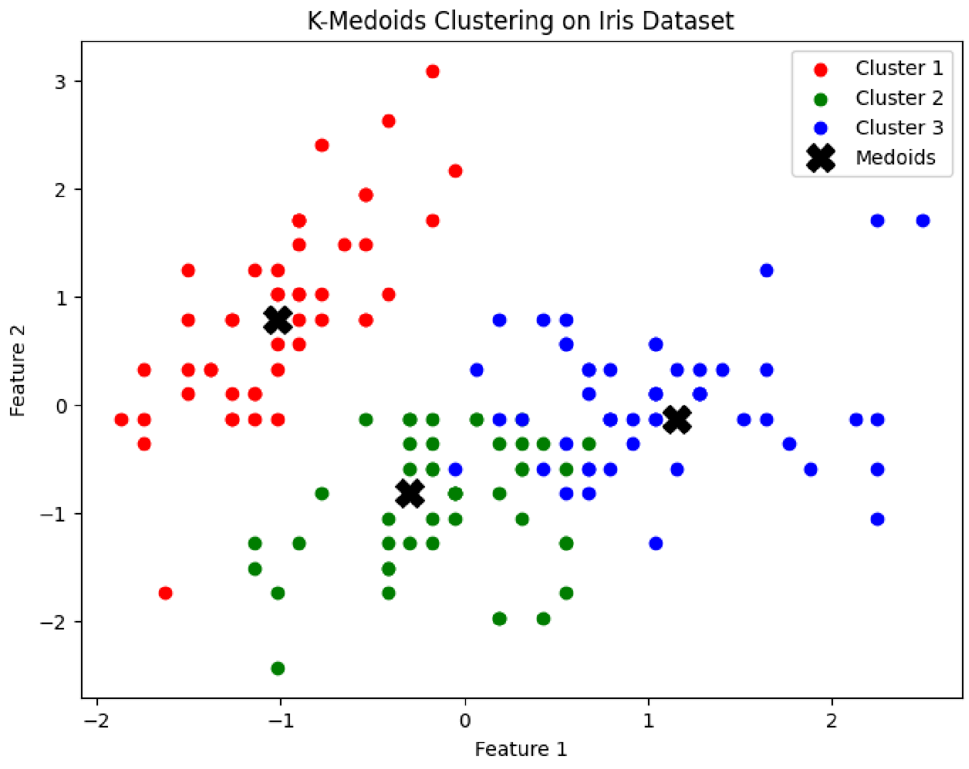
Applications

- Customer segmentation
- Image segmentation
- Network analysis
- Bioinformatics clustering

CODE:

```
!pip install pyclustering --quiet

# Import libraries
import pandas as pd
import matplotlib.pyplot as plt
from pyclustering.cluster.kmedoids import kmedoids
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris
import numpy as np

# Load Iris dataset
iris = load_iris()
X = iris.data # features
y_true = iris.target # true labels (optional)

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Initialize medoids (choose 3 random points for 3 clusters)
initial_medoids = [0, 50, 100]

# Run K-Medoids clustering
kmedoids_instance = kmedoids(X_scaled.tolist(), initial_medoids)
kmedoids_instance.process()

clusters = kmedoids_instance.get_clusters()
medoids = kmedoids_instance.get_medoids()

# Visualize clusters (using first 2 features for 2D plot)
colors = ['r','g','b']
plt.figure(figsize=(8,6))
for i, cluster in
    enumerate(clusters):
    cluster_points =
    X_scaled[cluster]
    plt.scatter(cluster_points[:, 0], cluster_points[:, 1], c=colors[i],
label=f'Cluster {i+1}')
plt.scatter(X_scaled[medoids,0], X_scaled[medoids,1], c='black',
marker='X', s=200, label='Medoids')
plt.title("K-Medoids Clustering on Iris Dataset")
plt.xlabel("Feature 1")
plt.ylabel("Feature
2") plt.legend()
plt.show()
```

OUTPUT:



LEARNING OUTCOME:

EXPERIMENT - 10

AIM: Program to demonstrate K-Means Clustering Algorithm on Handwritten Dataset

THEORY:

Clustering is an unsupervised machine learning technique used to group similar data points into clusters, without using labeled outputs.
K-Means Clustering Algorithm is one of the most widely used clustering methods. It partitions the dataset into k clusters by minimizing the intra-cluster variance (sum of squared distances from each point to its cluster centroid).

How it Works (The Algorithm):

1. Initialize (Choose k): First, you must decide how many clusters (k) you want to find. For handwritten digits, a natural choice would be k=10 (for digits 0-9).
2. Place Centroids: The algorithm randomly places k initial "centroids" (the center points of the clusters) in the data space.
3. Assignment Step: It goes through every data point and assigns it to the *nearest* centroid (usually based on standard Euclidean distance). This creates k groups of data points.
4. Update Step: The algorithm recalculates the position of each of the k centroids by finding the *mean* (the average) of all data points currently assigned to that cluster.
5. Iterate: Steps 3 and 4 are repeated. Data points get reassigned to the new, updated centroids. The centroids move again. This process continues until the centroids stop moving (or move very little), which means the clusters are stable and the algorithm has converged.

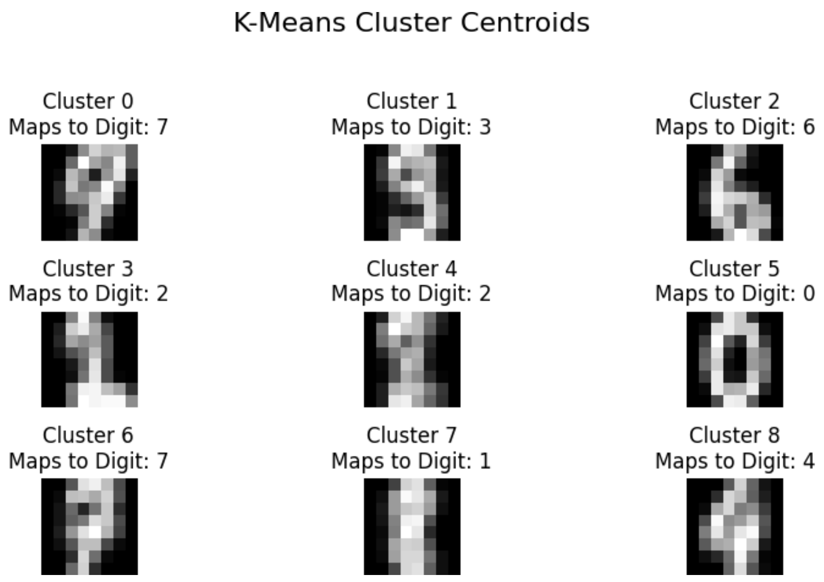
Application to Handwritten Digits:

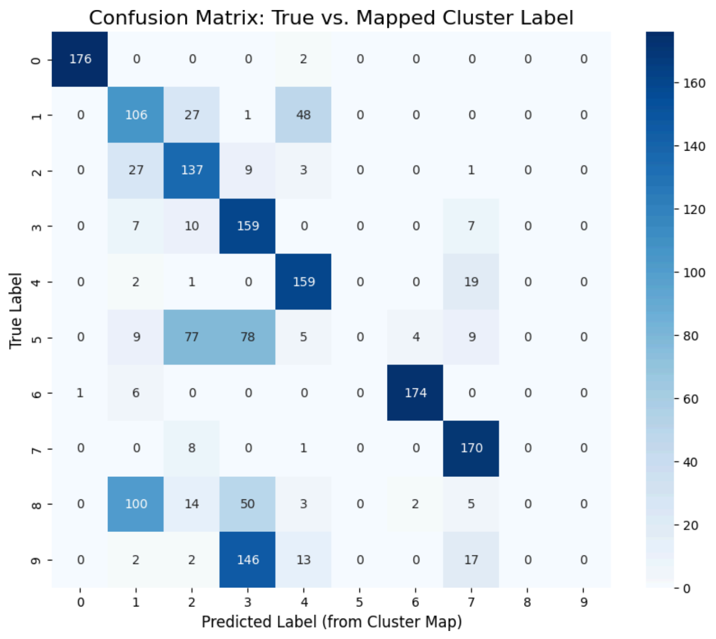
- Handwritten digits are high-dimensional images (e.g., 8×8 or 28×28 pixels).
- Each image is treated as a feature vector (flattened pixel values).
- K-Means groups images with similar pixel patterns into clusters.
- Although K-Means does not use labels, we can map clusters to actual digits using majority voting for evaluation.

CODE:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_digits
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix from
sklearn.manifold import TSNE
from scipy.stats import mode

# Load the handwritten digits dataset
digits = load_digits()
X = digits.data # 64 features (8x8 pixel images) y
= digits.target

# Normalize the data for better clustering performance
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply K-Means clustering
k = 9
kmeans = KMeans(n_clusters=k, random_state=42, n_init=20)
clusters = kmeans.fit_predict(X_scaled)

# Map each cluster to the most common true label
labels_map = np.zeros_like(clusters)
for i in range(k):
    mask = (clusters == i)
    if np.any(mask):
        labels_map[mask] = mode(y[mask], keepdims=False).mode

# Cluster Centroids
print("Displaying improved cluster centroid images...") centers_original_space
= scaler.inverse_transform(kmeans.cluster_centers_) centers_reshaped =
centers_original_space.reshape(k, 8, 8)

fig, axes = plt.subplots(3, 3, figsize=(9, 5))
fig.suptitle("K-Means Cluster Centroids", fontsize=16)

for i, ax in enumerate(axes.flat):
    mapped_digit = mode(y[clusters == i], keepdims=False).mode

    ax.imshow(centers_reshaped[i], cmap='gray') ax.set_title(f"Cluster
    {i}\nMaps to Digit: {mapped_digit}") ax.axis('off')

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()

# Confusion Matrix
cm = confusion_matrix(y, labels_map) plt.figure(figsize=(10,
8))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.title("Confusion Matrix: True vs. Mapped Cluster Label", fontsize=16)
plt.ylabel("True Label", fontsize=12)
plt.xlabel("Predicted Label (from Cluster Map)", fontsize=12)
plt.show()
```

OUTPUT:





	0	1	2	3	4	5	6	7	8	9
0	176	0	0	0	2	0	0	0	0	0
1	0	106	27	1	48	0	0	0	0	0
2	0	27	137	9	3	0	0	1	0	0
3	0	7	10	159	0	0	0	7	0	0
4	0	2	1	0	159	0	0	19	0	0
5	0	9	77	78	5	0	4	9	0	0
6	1	6	0	0	0	0	174	0	0	0
7	0	0	8	0	1	0	0	170	0	0
8	0	100	14	50	3	0	2	5	0	0
9	0	2	2	146	13	0	0	17	0	0

LEARNING OUTCOME: