

**RAJALAKSHMI ENGINEERING COLLEGE**  
**RAJALAKSHMI NAGAR, THANDALAM – 602 105**



**RAJALAKSHMI**  
**ENGINEERING COLLEGE**

**CS23333**  
**BUILD AND DEPLOYMENT OF MACHINE**  
**LEARNING APPLICATIONS**

**Laboratory Observation Note Book**

**Name :** Aniruth S.V. ....

**Year / Branch / Section :** 3<sup>rd</sup> Year/ AIML. ....

**Register No. :** 2116-231501015. ....

**Semester :** 5<sup>rd</sup> Semester. ....

**Academic Year :** 2025-2026. ....

<b>Exp No: 1</b>	<b>Setting Up the Environment And Preprocessing the Data</b>
------------------	--

**Aim:**

To set up a fully functional machine learning development environment and to perform data preprocessing operations like handling missing values, encoding categorical variables, feature scaling, and splitting datasets.

**Algorithm:**

1. Install Required Libraries:
  - Install numpy, pandas, matplotlib, seaborn, and scikit-learn using pip.
2. Import Libraries.
3. Load Dataset:
  - Load any dataset (e.g., Titanic or Iris) using pandas.
4. Data Exploration:
  - Use `df.info()`, `df.describe()`, `df.isnull().sum()` to understand the data.
5. Handle Missing Values:
  - Use `.fillna()` or `.dropna()` depending on the strategy.
6. Encode Categorical Data:
  - Use `pd.get_dummies()` or `LabelEncoder`.
7. Feature Scaling:
  - Normalize or standardize the numerical features using `StandardScaler` or `MinMaxScaler`.
8. Split Dataset:
  - Use `train_test_split()` from `sklearn` to create training and testing sets.
9. Display the Preprocessed Data.

**Code:**

```
# 1. Install necessary libraries (if not already installed)
# !pip install numpy pandas matplotlib seaborn scikit-learn

# 2. Import libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
import seaborn as sns
import matplotlib.pyplot as plt

# 3. Load dataset
df = sns.load_dataset('titanic') # Titanic dataset
df.head()

# 4. Explore the dataset
print(df.info())
print(df.describe())
print(df.isnull().sum())

# 5. Handle missing values
# Fill age with median, embark_town with mode
df['age'].fillna(df['age'].median(), inplace=True)
df['embark_town'].fillna(df['embark_town'].mode()[0], inplace=True)
df.drop(columns=['deck'], inplace=True) # too many missing values

# 6. Encode categorical variables
# Convert 'sex' and 'embark_town' using LabelEncoder
le = LabelEncoder()
```

```

df['sex'] = le.fit_transform(df['sex'])
df['embark_town'] = le.fit_transform(df['embark_town'])

# Drop non-informative or redundant columns
df.drop(columns=['embarked', 'class', 'who', 'alive', 'adult_male', 'alone'], inplace=True)

# 7. Feature Scaling
scaler = StandardScaler()
numerical_cols = ['age', 'fare']
df[numerical_cols] = scaler.fit_transform(df[numerical_cols])

# 8. Split dataset
# Define features (X) and label (y)
X = df.drop('survived', axis=1)
y = df['survived']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 9. Show final preprocessed data
print("Training Data Shape:", X_train.shape)
print("Test Data Shape:", X_test.shape)
X_train.head()

```

## Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
#   Column          Non-Null Count  Dtype
---  -
0   survived        891 non-null   int64
1   pclass          891 non-null   int64
2   sex             891 non-null   object
3   age             714 non-null   float64
4   sibsp           891 non-null   int64
5   parch           891 non-null   int64
6   fare            891 non-null   float64
7   embarked        889 non-null   object
8   class           891 non-null   category
9   who             891 non-null   object
10  adult_male      891 non-null   bool
11  deck            203 non-null   category
12  embark_town     889 non-null   object
13  alive           891 non-null   object
14  alone           891 non-null   bool
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 80.7+ KB
None
```

	survived	pclass	age	sibsp	parch	fare
count	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

```
survived      0
pclass        0
sex           0
age           177
sibsp         0
parch         0
fare          0
embarked      2
class         0
who           0
adult_male    0
deck          688
embark_town   2
alive         0
alone         0
dtype: int64
```

Training Data Shape: (712, 7)

Test Data Shape: (179, 7)

/tmp/ipython-input-4068659829.py:3: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained assignment using an inplace method.  
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, to perform the operation inplace on the original object.

```
df['age'].fillna(df['age'].median(), inplace=True)
```

/tmp/ipython-input-4068659829.py:4: FutureWarning: A value is trying to be set on a copy of a DataFrame or Series through chained assignment using an inplace method.  
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value) instead, to perform the operation inplace on the original object.

```
df['embark_town'].fillna(df['embark_town'].mode()[0], inplace=True)
```

	pclass	sex	age	sibsp	parch	fare	embark_town
331	1	1	1.240235	0	0	-0.074583	2
733	2	1	-0.488887	0	0	-0.386671	2
382	3	1	0.202762	0	0	-0.488854	2
704	3	1	-0.258337	1	0	-0.490280	2
813	3	0	-1.795334	4	2	-0.018709	2

<b>Exp No: 2</b>	<b>Support Vector Machine (SVM) and Random Forest for Binary &amp; Multiclass Classification</b>
------------------	--

### **Aim**

To build classification models using **Support Vector Machines (SVM)** and **Random Forest**, apply them to a dataset, and evaluate the models using performance metrics like accuracy and confusion matrix.

### **Algorithm**

#### Part A: SVM Model

1. Import necessary libraries
2. Load and explore the dataset
3. Handle missing values if any
4. Encode categorical variables
5. Split dataset into training and testing sets
6. Build SVM classifier using SVC()
7. Train and predict
8. Evaluate the model using accuracy and confusion matrix

#### Part B: Random Forest Model

1. Initialize Random Forest using RandomForestClassifier()
2. Train and predict
3. Evaluate and compare with SVM

### **Code:**

```
# 1. Import libraries
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
```

```

from sklearn.metrics import accuracy_score, confusion_matrix

import seaborn as sns

import matplotlib.pyplot as plt


# 2. Load dataset
iris = load_iris()
X = iris.data
y = iris.target


# 3. Feature scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)


# 4. Train-test split
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.3, random_state=42)


# -----
# Part A: SUPPORT VECTOR MACHINE
# -----


# 5. Initialize and train SVM
svm_model = SVC(kernel='linear') # You can also try 'rbf', 'poly'
svm_model.fit(X_train, y_train)


# 6. Predict and evaluate SVM
y_pred_svm = svm_model.predict(X_test)
print("SVM Accuracy:", accuracy_score(y_test, y_pred_svm))
print("SVM Confusion Matrix:\n", confusion_matrix(y_test, y_pred_svm))

```



```

# -----

# Part B: RANDOM FOREST

# -----

# 7. Initialize and train Random Forest
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# 8. Predict and evaluate Random Forest
y_pred_rf = rf_model.predict(X_test)
print("Random Forest Accuracy:", accuracy_score(y_test, y_pred_rf))
print("Random Forest Confusion Matrix:\n", confusion_matrix(y_test, y_pred_rf))

# -----

# 9. Visual comparison using seaborn heatmap

# -----

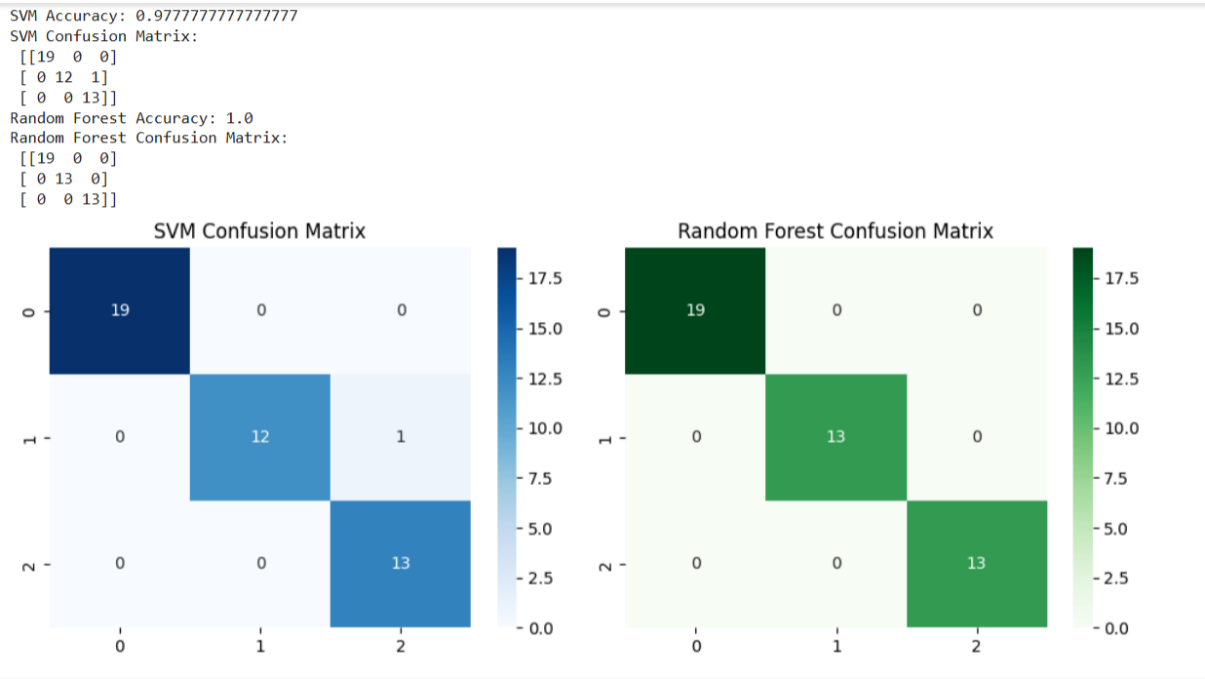
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
sns.heatmap(confusion_matrix(y_test, y_pred_svm), annot=True, cmap='Blues', fmt='d')
plt.title("SVM Confusion Matrix")

plt.subplot(1, 2, 2)
sns.heatmap(confusion_matrix(y_test, y_pred_rf), annot=True, cmap='Greens', fmt='d')
plt.title("Random Forest Confusion Matrix")
plt.tight_layout()
plt.show()

```

**OUTPUT:**



<b>ExpNo:3</b>	<b>Classification with Decision Trees</b>
----------------	---

### **Aim**

To implement a Decision Tree classifier and evaluate its performance using **accuracy score** and **confusion matrix** on a real-world dataset.

### **Algorithm**

1. Import necessary libraries
2. Load a classification dataset (e.g., Iris or Titanic)
3. Split the dataset into training and test sets
4. Preprocess data if needed
5. Train a DecisionTreeClassifier from sklearn.tree
6. Predict on test data
7. Evaluate using:
  - Confusion Matrix
  - Accuracy Score
8. Visualize the Decision Tree (optional)

### **Code:**

# Step 1: Import Libraries

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# Step 2: Load Dataset
```

```
iris = load_iris()
```

```
X = iris.data
```

```
y = iris.target
```

```
# Step 3: Split the dataset
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```
# Step 4: Train the Decision Tree Classifier
```

```
dt_model = DecisionTreeClassifier(criterion='gini', random_state=0)
```

```
dt_model.fit(X_train, y_train)
```

```
# Step 5: Predict
```

```
y_pred = dt_model.predict(X_test)
```

```
# Step 6: Evaluate the Model
```

```
cm = confusion_matrix(y_test, y_pred)
```

```
acc = accuracy_score(y_test, y_pred)
```

```
print("Confusion Matrix:\n", cm)
```

```
print("Accuracy Score:", acc)
```

```
# Step 7: Visualize Confusion Matrix
```

```
sns.heatmap(cm, annot=True, cmap="Blues", xticklabels=iris.target_names,  
yticklabels=iris.target_names)
```

```
plt.xlabel("Predicted")
```

```
plt.ylabel("Actual")
```

```
plt.title("Confusion Matrix")
```

```
plt.show()
```

```
# Step 8: Visualize the Decision Tree
```

```
plt.figure(figsize=(12,8))
```

```
plot_tree(dt_model, filled=True, feature_names=iris.feature_names, class_names=iris.target_names)
```

```
plt.title("Decision Tree Visualization")
```

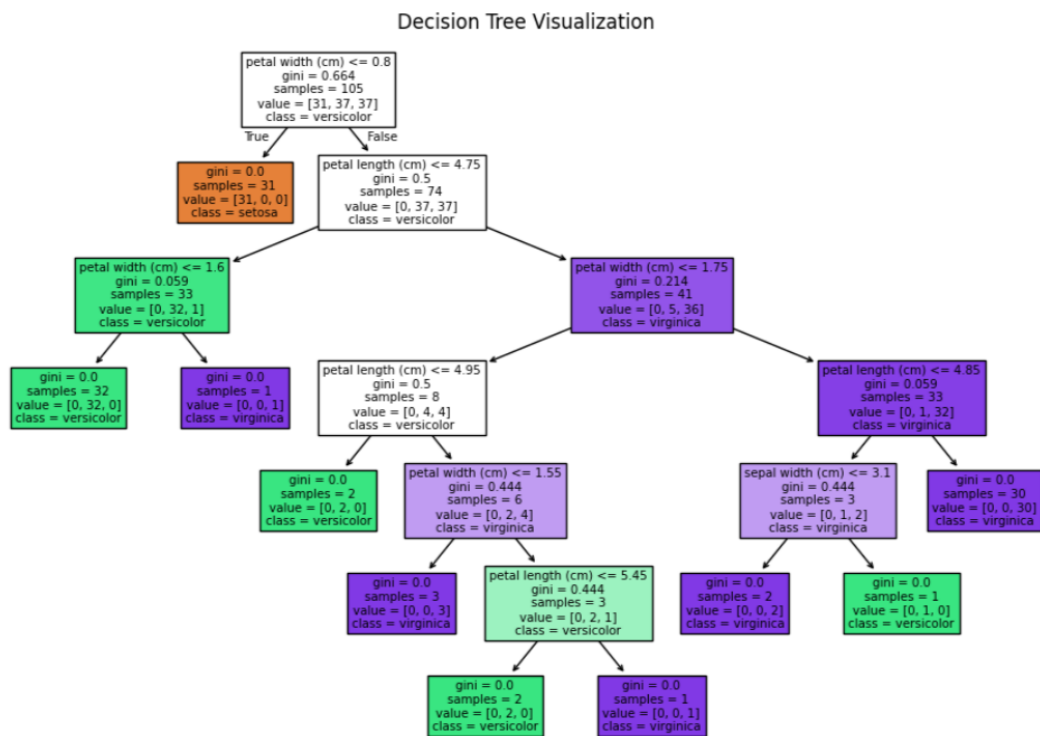
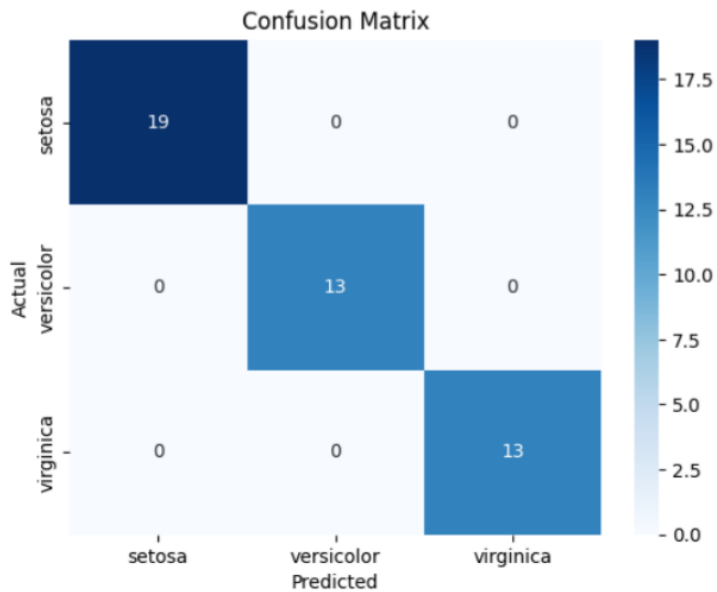
```
plt.show()
```

## OUTPUT:

Confusion Matrix:

```
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
```

Accuracy Score: 1.0



<b>Exp No: 4A</b>	<b>Support Vector Machines (SVM)</b>
-------------------	--------------------------------------

**Aim:**

To build an SVM model for a binary classification task, tune its hyperparameters, and evaluate it using accuracy, precision, recall, F1-score, confusion matrix, and ROC-AUC.

**Algorithm:**

1. Import libraries: numpy, pandas, matplotlib, sklearn.
2. Load data: Use a standard binary dataset (Breast Cancer Wisconsin) from sklearn.datasets.
3. Train/Test split: 80/20 split with a fixed random\_state.
4. Preprocess: Standardize features (StandardScaler).
5. SVMs are sensitive to feature scale.
6. Model selection: Use SVC (RBF kernel).
7. Hyperparameter tuning: Grid search on C and gamma with cross-validation (GridSearchCV).
8. Train final model: Fit on training data using best parameters.
9. Evaluate: Predict on test set; compute metrics and plot ROC curve.
10. Report: Best params, metrics, and brief observations.

**CODE:**

```
# =====
# EXPERIMENT 4A — SVM (RBF)
# =====

# 1) Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import load_breast_cancer
```

```

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, classification_report, roc_auc_score, roc_curve
)

# 2) Load dataset (binary classification)
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = pd.Series(data.target, name="target") # 0 = malignant, 1 = benign

# 3) Train/test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, random_state=42, stratify=y
)

# 4) Standardize features (important for SVMs)
scaler = StandardScaler()
X_train_sc = scaler.fit_transform(X_train)
X_test_sc = scaler.transform(X_test)

# 5) Define model
svm = SVC(kernel='rbf', probability=True, random_state=42)

# 6) Hyperparameter grid & tuning
param_grid = {
    "C": [0.1, 1, 10, 100],
    "gamma": ["scale", 0.01, 0.001, 0.0001]
}

```

```

grid = GridSearchCV(
    estimator=svm,
    param_grid=param_grid,
    scoring='f1', # You can change to 'accuracy' or 'roc_auc'
    cv=5,
    n_jobs=-1,
    verbose=0
)

grid.fit(X_train_sc, y_train)

print("Best Parameters from Grid Search:", grid.best_params_)
best_svm = grid.best_estimator_

# 7) Train final model & predict
best_svm.fit(X_train_sc, y_train)
y_pred = best_svm.predict(X_test_sc)
y_prob = best_svm.predict_proba(X_test_sc)[:, 1]

# 8) Evaluation
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred, zero_division=0)
rec = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
auc = roc_auc_score(y_test, y_prob)
cm = confusion_matrix(y_test, y_pred)

print("\n=== SVM (RBF) — Test Metrics ===")
print(f"Accuracy : {acc:.4f}")
print(f"Precision: {prec:.4f}")

```



```
print(f"Recall : {rec:.4f}")
print(f"F1-Score : {f1:.4f}")
print(f"ROC-AUC : {auc:.4f}")

print("\nConfusion Matrix:\n", cm)
print("\nClassification Report:\n", classification_report(y_test, y_pred, zero_division=0))

# 9) Plot ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
plt.figure()
plt.plot(fpr, tpr, label=f"SVM (AUC = {auc:.3f})")
plt.plot([0, 1], [0, 1], linestyle="--", color='gray')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve — SVM (RBF)")
plt.legend()
plt.grid(True)
plt.show()
```

## OUTPUT:

Best Parameters from Grid Search: {'C': 10, 'gamma': 0.01}

=== SVM (RBF) - Test Metrics ===

Accuracy : 0.9825

Precision: 0.9861

Recall : 0.9861

F1-Score : 0.9861

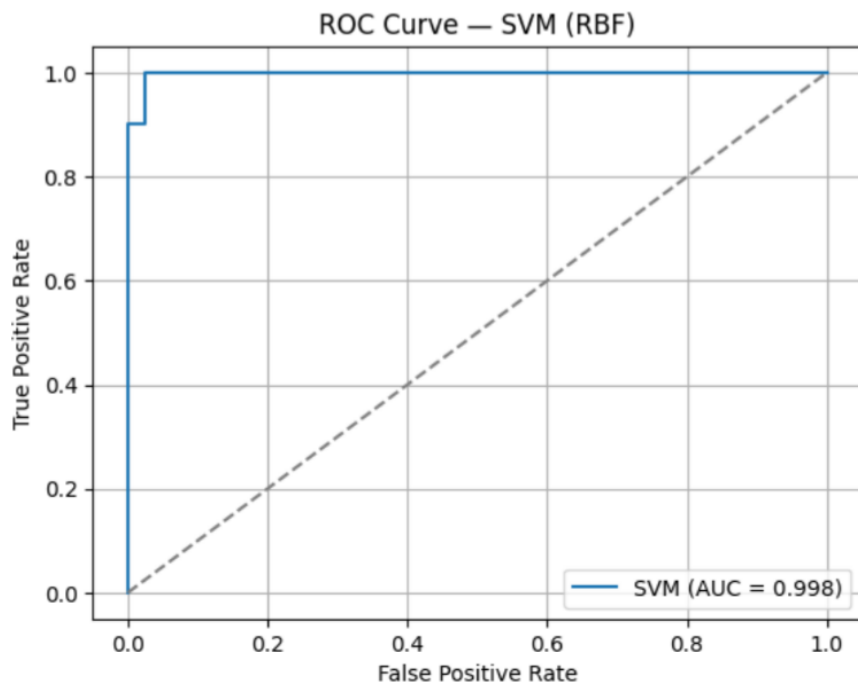
ROC-AUC : 0.9977

Confusion Matrix:

```
[[41  1]
 [ 1 71]]
```

Classification Report:

	precision	recall	f1-score	support
0	0.98	0.98	0.98	42
1	0.99	0.99	0.99	72
accuracy			0.98	114
macro avg	0.98	0.98	0.98	114
weighted avg	0.98	0.98	0.98	114



<b>Exp No: 4B</b>	<b>Ensemble Methods: Random Forest</b>
-------------------	--

### **Aim:**

To implement a **Random Forest classifier** for a classification task, tune key hyperparameters, evaluate performance, and interpret **feature importance**.

### **Algorithm:**

1. Import libraries.
2. Load data (use same dataset to compare with SVM).
3. Train/Test split with stratification.
4. (Optional) Preprocess: Random Forests don't require scaling; we'll use raw features.
5. Model: RandomForestClassifier.
6. Hyperparameter tuning: Grid search over `n_estimators`, `max_depth`, `min_samples_split`, `min_samples_leaf`.
7. Train the best model on training data.
8. Evaluate with accuracy, precision, recall, F1, confusion matrix, ROC-AUC.
9. Interpretation: Plot top feature importances.

### **CODE:**

```
# =====
# EXPERIMENT 4B — Random Forest Classifier
# =====

# 1) Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestClassifier
```

```

from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, classification_report, roc_auc_score, roc_curve
)

# 2) Load dataset (same as 4A for comparison)
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = pd.Series(data.target, name="target")

# 3) Train/test split (no scaling needed for RF)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, random_state=42, stratify=y
)

# 4) Define model
rf = RandomForestClassifier(random_state=42, n_jobs=-1)

# 5) Hyperparameter grid & tuning
param_grid = {
    "n_estimators": [100],
    "max_depth": [None, 10],
    "min_samples_split": [2],
    "min_samples_leaf": [1]
}
grid = GridSearchCV(
    estimator=rf,
    param_grid=param_grid,
    scoring="f1",
    cv=3,
    n_jobs=-1,
    verbose=0)

```

```

grid.fit(X_train, y_train)
print("Best Parameters (CV):", grid.best_params_)
best_rf = grid.best_estimator_

# 6) Train final model & predict
best_rf.fit(X_train, y_train)
y_pred = best_rf.predict(X_test)
y_prob = best_rf.predict_proba(X_test)[:, 1]

# 7) Evaluate
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred, zero_division=0)
rec = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
auc = roc_auc_score(y_test, y_prob)
cm = confusion_matrix(y_test, y_pred)

print("\n=== Random Forest — Test Metrics ===")
print(f"Accuracy : {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall   : {rec:.4f}")
print(f"F1-Score : {f1:.4f}")
print(f"ROC-AUC  : {auc:.4f}")

print("\nConfusion Matrix:\n", cm)
print("\nClassification Report:\n", classification_report(y_test, y_pred, zero_division=0))

# 8) Feature Importance (Top 10)
importances = pd.Series(best_rf.feature_importances_, index=X.columns)
top10 = importances.sort_values(ascending=False).head(10)

```

```

plt.figure()
top10[:::-1].plot(kind="barh")
plt.xlabel("Importance")
plt.title("Top 10 Feature Importances — Random Forest")
plt.grid(axis="x", alpha=0.3)
plt.show()

# 9) ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
plt.figure()
plt.plot(fpr, tpr, label=f"Random Forest (AUC = {auc:.3f})")
plt.plot([0, 1], [0, 1], linestyle="--", color='gray')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve — Random Forest")
plt.legend()
plt.grid(True)
plt.show()

```

## OUTPUT:

```
Best Parameters (CV): {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
```

```
=== Random Forest — Test Metrics ===
```

```

Accuracy : 0.9561
Precision: 0.9589
Recall   : 0.9722
F1-Score : 0.9655
ROC-AUC  : 0.9937

```

```
Confusion Matrix:
```

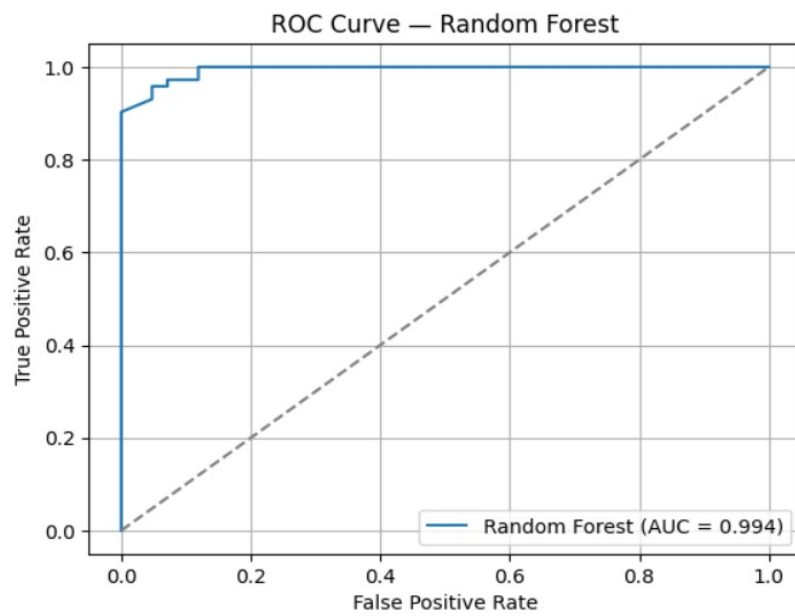
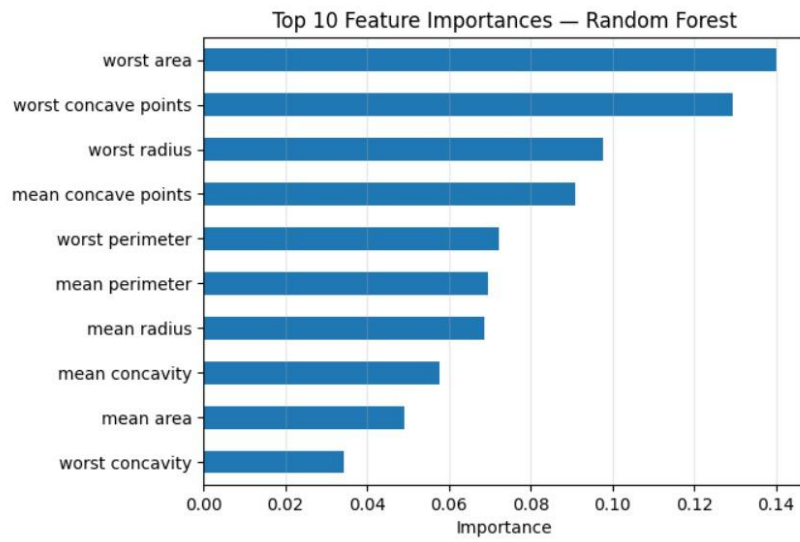
```

[[39  3]
 [ 2 70]]

```

```
Classification Report:
```

	precision	recall	f1-score	support
0	0.95	0.93	0.94	42
1	0.96	0.97	0.97	72
accuracy			0.96	114
macro avg	0.96	0.95	0.95	114
weighted avg	0.96	0.96	0.96	114



Exp No: 5	Clustering with K-Means and Dimensionality Reduction with PCA
-----------	---

**Aim:**

To demonstrate the application of Unsupervised Learning models, specifically K-Means clustering for grouping data points and Principal Component Analysis (PCA) for dimensionality reduction and visualization, using a suitable dataset.

**Algorithm:**

**1. K-Means Clustering**

K-Means is an iterative clustering algorithm that aims to partition  $n$  observations into  $k$  clusters, where each observation belongs to the cluster with the nearest mean (centroid).

**Steps:**

1. **Initialization:** Choose  $k$  initial centroids randomly from the dataset.
2. **Assignment:** Assign each data point to the cluster whose centroid is closest (e.g., using Euclidean distance).
3. **Update:** Recalculate the centroids as the mean of all data points assigned to that cluster.
4. **Iteration:** Repeat steps 2 and 3 until the centroids no longer move significantly or a maximum number of iterations is reached.

**2. Principal Component Analysis (PCA)**

PCA is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components.

**Steps:**

1. **Standardization:** Standardize the dataset (mean = 0, variance = 1).
2. **Covariance Matrix Calculation:** Compute the covariance matrix of the standardized data.
3. **Eigenvalue Decomposition:** Calculate the eigenvalues and eigenvectors of the covariance matrix.
4. **Feature Vector Creation:** Sort the eigenvectors by decreasing eigenvalues and select the top  $k$  eigenvectors to form a feature vector (projection matrix).
5. **Projection:** Project the original data onto the new feature space using the feature vector.



## CODE:

```
# =====  
# EXPERIMENT — K-Means & PCA  
# =====  
  
# Import necessary libraries  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns  
  
from sklearn.datasets import make_blobs  
from sklearn.preprocessing import StandardScaler  
from sklearn.cluster import KMeans  
from sklearn.decomposition import PCA  
from sklearn.metrics import silhouette_score  
  
# --- Part 1: K-Means Clustering ---  
  
print("--- Part 1: K-Means Clustering ---")  
  
# 1. Generate dataset  
X, y = make_blobs(n_samples=300, centers=3, cluster_std=0.60, random_state=42)  
df_kmeans = pd.DataFrame(X, columns=['Feature_1', 'Feature_2'])  
print("\nOriginal K-Means Dataset Head:")  
print(df_kmeans.head())  
  
# 2. Elbow Method  
wcss = []  
for i in range(1, 11):  
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10,  
random_state=42)  
    kmeans.fit(X)  
    wcss.append(kmeans.inertia_)  
  
plt.figure(figsize=(10, 6))  
plt.plot(range(1, 11), wcss, marker='o', linestyle='--')  
plt.title('Elbow Method for Optimal K (K-Means)')  
plt.xlabel('Number of Clusters (K)')  
plt.ylabel('WCSS')  
plt.grid(True)  
plt.show()  
  
# 3. Apply K-Means with chosen K  
optimal_k = 3
```

```

kmeans = KMeans(n_clusters=optimal_k, init='k-means++', max_iter=300, n_init=10,
random_state=42)
clusters = kmeans.fit_predict(X)
df_kmeans['Cluster'] = clusters

# 4. Visualize K-Means clusters
plt.figure(figsize=(10, 8))
sns.scatterplot(x='Feature_1', y='Feature_2', hue='Cluster', data=df_kmeans, palette='viridis',
s=100, alpha=0.8)
plt.scatter(kmeans.cluster_centers_[0], kmeans.cluster_centers_[1], s=300, c='red',
marker='X', label='Centroids')
plt.title(f'K-Means Clustering with K={optimal_k}')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.grid(True)
plt.show()

# 5. Silhouette Score
silhouette_avg = silhouette_score(X, clusters)
print(f"\nSilhouette Score for K-Means (K={optimal_k}): {silhouette_avg:.3f}")

# --- Part 2: Dimensionality Reduction with PCA ---

print("\n--- Part 2: Dimensionality Reduction with PCA ---")

# 1. Generate 4D dataset
X_pca, y_pca = make_blobs(n_samples=500, n_features=4, centers=4, cluster_std=1.0,
random_state=25)
df_pca_original = pd.DataFrame(X_pca, columns=[f'Feature_{i+1}' for i in
range(X_pca.shape[1])])
df_pca_original['True_Cluster'] = y_pca
print("\nOriginal PCA Dataset Head:")
print(df_pca_original.head())
print(f"Original PCA Dataset Shape: {df_pca_original.shape}")

# 2. Standardize
scaler = StandardScaler()
X_pca_scaled = scaler.fit_transform(X_pca)

# 3. PCA (4D → 2D)
pca = PCA(n_components=2)
principal_components = pca.fit_transform(X_pca_scaled)
df_principal_components = pd.DataFrame(principal_components,
columns=['Principal_Component_1', 'Principal_Component_2'])
df_principal_components['True_Cluster'] = y_pca

```

```

explained_variance = pca.explained_variance_ratio_
print("\nPrincipal Components Head:")
print(df_principal_components.head())
print(f"\nExplained Variance Ratio: {explained_variance}")
print(f"Total Explained Variance by 2 PCs: {explained_variance.sum():.3f}")

# 4. Visualize PCA result
plt.figure(figsize=(10, 8))
sns.scatterplot(x='Principal_Component_1', y='Principal_Component_2', hue='True_Cluster',
                data=df_principal_components, palette='Paired', s=100, alpha=0.8)
plt.title('PCA - Dimensionality Reduction to 2 Components')
plt.xlabel(f'PC1 ({explained_variance[0]*100:.2f}%)')
plt.ylabel(f'PC2 ({explained_variance[1]*100:.2f}%)')
plt.grid(True)
plt.show()

# 5. K-Means on PCA-reduced data
kmeans_pca = KMeans(n_clusters=4, init='k-means++', max_iter=300, n_init=10,
                    random_state=42)
clusters_pca = kmeans_pca.fit_predict(principal_components)
df_principal_components['KMeans_Cluster_on_PCA'] = clusters_pca

plt.figure(figsize=(10, 8))
sns.scatterplot(x='Principal_Component_1', y='Principal_Component_2',
                hue='KMeans_Cluster_on_PCA',
                data=df_principal_components, palette='viridis', s=100, alpha=0.8)
plt.scatter(kmeans_pca.cluster_centers_[0, 0], kmeans_pca.cluster_centers_[0, 1], s=300,
            c='red', marker='X', label='Centroids')
plt.title('K-Means Clustering on PCA-Reduced Data')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.grid(True)
plt.show()

# 6. Silhouette Score for PCA-reduced KMeans
silhouette_avg_pca = silhouette_score(principal_components, clusters_pca)
print(f"\nSilhouette Score for K-Means on PCA-Reduced Data (K=4):
{silhouette_avg_pca:.3f}")

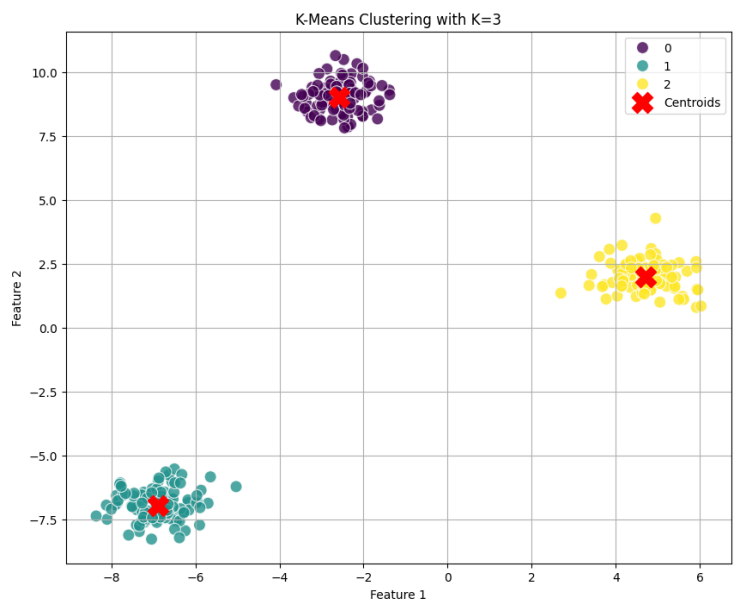
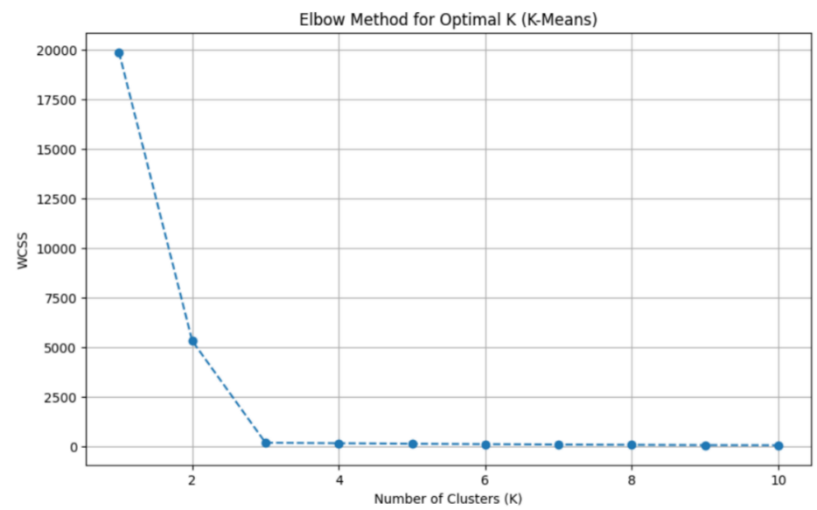
```

**OUTPUT:**

--- Part 1: K-Means Clustering ---

Original K-Means Dataset Head:

	Feature_1	Feature_2
0	-7.155244	-7.390016
1	-7.395875	-7.110843
2	-2.015671	8.281780
3	4.509270	2.632436
4	-8.102502	-7.484961



Silhouette Score for K-Means (K=3): 0.908

--- Part 2: Dimensionality Reduction with PCA ---

Original PCA Dataset Head:

	Feature_1	Feature_2	Feature_3	Feature_4	True_Cluster
0	-0.638667	1.110057	-6.400722	-0.204990	3
1	-2.951556	-7.657445	3.844794	0.903589	1
2	-0.253177	2.125103	-7.869801	0.559678	3
3	-2.151209	3.401400	-5.734930	0.965230	3
4	-2.347519	-7.230467	3.478891	-0.443440	1

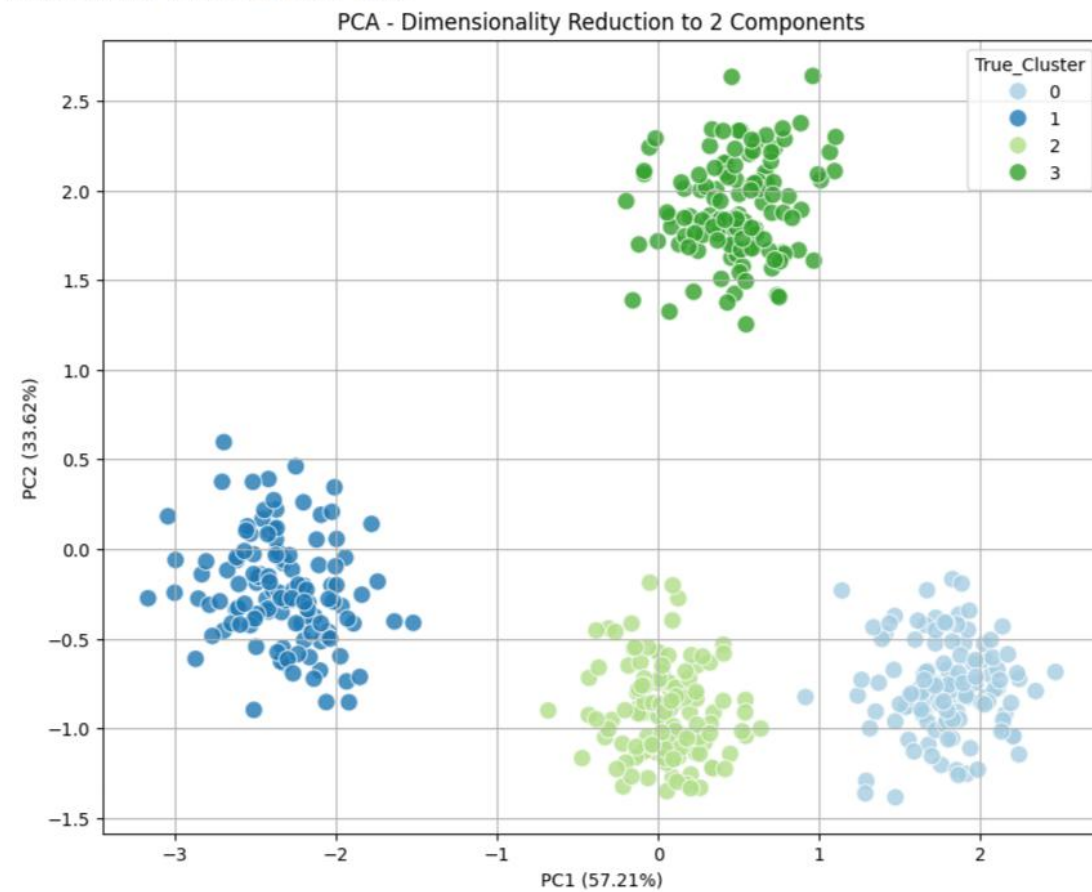
Original PCA Dataset Shape: (500, 5)

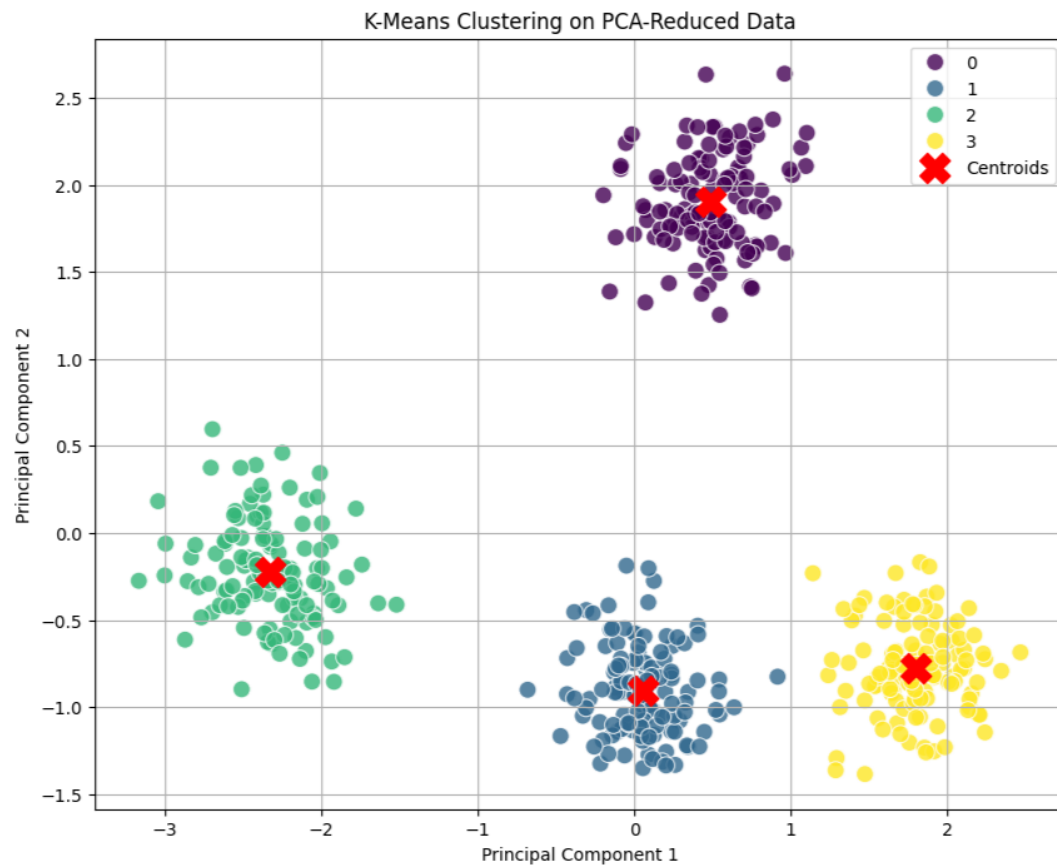
Principal Components Head:

	Principal_Component_1	Principal_Component_2	True_Cluster
0	0.455305	1.623917	3
1	-2.705622	0.375012	1
2	0.810234	1.966926	3
3	0.427139	2.149626	3
4	-2.407508	0.099250	1

Explained Variance Ratio: [0.57208431 0.33622342]

Total Explained Variance by 2 PCs: 0.908





Silhouette Score for K-Means on PCA-Reduced Data (K=4): 0.776

<b>Exp No: 6</b>	<b>Feedforward and Convolutional Neural Networks</b>
------------------	--

**Aim:**

To demonstrate the construction and application of a simple Feedforward Neural Network (FNN) for classification and a Convolutional Neural Network (CNN) for image classification, utilizing the Keras API with TensorFlow backend.

**Algorithm:**

**1. Feedforward Neural Network (FNN)**

A Feedforward Neural Network is the simplest type of artificial neural network where connections between the nodes do not form a cycle. It consists of an input layer, one or more hidden layers, and an output layer. Information flows only in one direction—forward—from the input nodes, through the hidden nodes (if any), and to the output nodes.

**Steps:**

1. Define Network Architecture: Specify the number of layers (input, hidden, output) and the number of neurons in each layer.
2. Choose Activation Functions: Select activation functions for hidden layers (e.g., ReLU) and the output layer (e.g., Sigmoid for binary classification, Softmax for multi-class classification).
3. Define Loss Function: Choose a loss function appropriate for the task (e.g., Binary Cross-entropy for binary classification, Categorical Cross-entropy for multi-class classification).
4. Choose Optimizer: Select an optimization algorithm (e.g., Adam, SGD) to update network weights during training.
5. Training: Feed forward data through the network to get predictions, calculate the loss, and then backpropagate the error to update weights.
6. Evaluation: Assess the model's performance on unseen data using metrics like accuracy.

**2. Convolutional Neural Network (CNN)**

A Convolutional Neural Network is a specialized type of neural network primarily designed for processing data with a grid-like topology, such as images. Key components include convolutional layers, pooling layers, and fully connected layers.

**Steps:**

1. Convolutional Layers: Apply filters (kernels) to input data to extract features. Each filter detects a specific pattern (e.g., edges, textures).
2. Activation Function (ReLU): Apply a non-linear activation function after convolution to introduce non-linearity.
3. Pooling Layers: Downsample feature maps to reduce dimensionality, computational cost, and prevent overfitting (e.g., Max Pooling).
4. Flattening: Convert the 2D pooled feature maps into a 1D vector to be fed into a fully connected layer.
5. Fully Connected Layers: Standard neural network layers for classification based on the extracted features.
6. Output Layer: Final layer with an activation function (e.g., Softmax) to output class probabilities.
7. Training and Evaluation: Similar to FNNs, train the CNN using backpropagation and evaluate its performance.

#### **CODE:**

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.datasets import mnist, fashion_mnist
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns

# Suppress TensorFlow warnings for cleaner output
tf.keras.utils.disable_interactive_logging()

# --- Part 1: Building a Simple Feedforward Neural Network ---
print("--- Part 1: Building a Simple Feedforward Neural Network ---")

# 1. Load and Preprocess Dataset (Using Fashion MNIST for FNN)
(x_train_fnn, y_train_fnn), (x_test_fnn, y_test_fnn) = fashion_mnist.load_data()

print(f"\nOriginal FNN training data shape: {x_train_fnn.shape}")
print(f"Original FNN test data shape: {x_test_fnn.shape}")

# Flatten images to 1D array
x_train_fnn_flat = x_train_fnn.reshape(-1, 28 * 28)
```



```

x_test_fnn_flat = x_test_fnn.reshape(-1, 28 * 28)

# Normalize pixel values
x_train_fnn_norm = x_train_fnn_flat / 255.0
x_test_fnn_norm = x_test_fnn_flat / 255.0

print(f"Flattened & Normalized FNN training data shape: {x_train_fnn_norm.shape}")
print(f"Flattened & Normalized FNN test data shape: {x_test_fnn_norm.shape}")

# 2. Build FNN Model
model_fnn = keras.Sequential([
    layers.Dense(128, activation='relu', input_shape=(784,)),
    layers.Dropout(0.2),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# 3. Compile Model
model_fnn.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

print("\n--- FNN Model Summary ---")
model_fnn.summary()

# 4. Train Model
print("\n--- Training FNN Model ---")
history_fnn = model_fnn.fit(x_train_fnn_norm, y_train_fnn, epochs=10,
                           validation_split=0.1, verbose=1)

# 5. Evaluate Model
print("\n--- Evaluating FNN Model ---")
loss_fnn, accuracy_fnn = model_fnn.evaluate(x_test_fnn_norm, y_test_fnn, verbose=0)
print(f"FNN Test Loss: {loss_fnn:.4f}")
print(f"FNN Test Accuracy: {accuracy_fnn:.4f}")

# Classification report & confusion matrix
y_pred_fnn = np.argmax(model_fnn.predict(x_test_fnn_norm), axis=-1)
print("\n--- FNN Classification Report ---")
print(classification_report(y_test_fnn, y_pred_fnn))

```

```

print("\n--- FNN Confusion Matrix ---")
cm_fnn = confusion_matrix(y_test_fnn, y_pred_fnn)
plt.figure(figsize=(10, 8))
sns.heatmap(cm_fnn, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.title("FNN Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()

# Plot Accuracy & Loss
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history_fnn.history['accuracy'], label='Training Accuracy')
plt.plot(history_fnn.history['val_accuracy'], label='Validation Accuracy')
plt.title('FNN Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(history_fnn.history['loss'], label='Training Loss')
plt.plot(history_fnn.history['val_loss'], label='Validation Loss')
plt.title('FNN Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# --- Part 2: Convolutional Neural Network (CNN) ---
print("\n--- Part 2: Implementing a CNN ---")

# 1. Load MNIST for CNN
(x_train_cnn, y_train_cnn), (x_test_cnn, y_test_cnn) = mnist.load_data()
print(f"\nOriginal CNN training data shape: {x_train_cnn.shape}")
print(f"Original CNN test data shape: {x_test_cnn.shape}")

```

```

# Reshape for channel dimension
x_train_cnn = x_train_cnn.reshape(x_train_cnn.shape[0], 28, 28, 1)
x_test_cnn = x_test_cnn.reshape(x_test_cnn.shape[0], 28, 28, 1)

# Normalize
x_train_cnn = x_train_cnn.astype('float32') / 255.0
x_test_cnn = x_test_cnn.astype('float32') / 255.0

print(f"Reshaped & Normalized CNN training data shape: {x_train_cnn.shape}")
print(f"Reshaped & Normalized CNN test data shape: {x_test_cnn.shape}")

num_classes_cnn = 10

# 2. Build CNN Model
model_cnn = keras.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(num_classes_cnn, activation='softmax')
])

# 3. Compile Model
model_cnn.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

print("\n--- CNN Model Summary ---")
model_cnn.summary()

# 4. Train Model
print("\n--- Training CNN Model ---")
history_cnn = model_cnn.fit(x_train_cnn, y_train_cnn, epochs=10,
                           validation_split=0.1, verbose=1)

# 5. Evaluate Model
print("\n--- Evaluating CNN Model ---")

```

```

loss_cnn, accuracy_cnn = model_cnn.evaluate(x_test_cnn, y_test_cnn, verbose=0)
print(f"CNN Test Loss: {loss_cnn:.4f}")
print(f"CNN Test Accuracy: {accuracy_cnn:.4f}")

# Classification report & confusion matrix
y_pred_cnn = np.argmax(model_cnn.predict(x_test_cnn), axis=-1)
print("\n--- CNN Classification Report ---")
print(classification_report(y_test_cnn, y_pred_cnn))

print("\n--- CNN Confusion Matrix ---")
cm_cnn = confusion_matrix(y_test_cnn, y_pred_cnn)
plt.figure(figsize=(10, 8))
sns.heatmap(cm_cnn, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.title("CNN Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()

# Plot Accuracy & Loss
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history_cnn.history['accuracy'], label='Training Accuracy')
plt.plot(history_cnn.history['val_accuracy'], label='Validation Accuracy')
plt.title('CNN Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(history_cnn.history['loss'], label='Training Loss')
plt.plot(history_cnn.history['val_loss'], label='Validation Loss')
plt.title('CNN Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

```

```

# Optional: Visualize predictions
print("\n--- Sample CNN Predictions ---")
class_names_mnist = [str(i) for i in range(10)]
plt.figure(figsize=(10, 10))
for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_test_cnn[i].reshape(28, 28), cmap=plt.cm.binary)
    true_label = y_test_cnn[i]
    predicted_label = y_pred_cnn[i]
    color = 'green' if true_label == predicted_label else 'red'
    plt.xlabel(f"True: {class_names_mnist[true_label]}\nPred: {class_names_mnist[predicted_label]}", color=color)
plt.suptitle("Sample CNN Predictions (Green: Correct, Red: Incorrect)", y=1.02, fontsize=16)
plt.tight_layout(rect=[0, 0, 1, 0.98])
plt.show()

```

## OUTPUT:

```

FNN Test Loss: 0.3404
FNN Test Accuracy: 0.8824

```

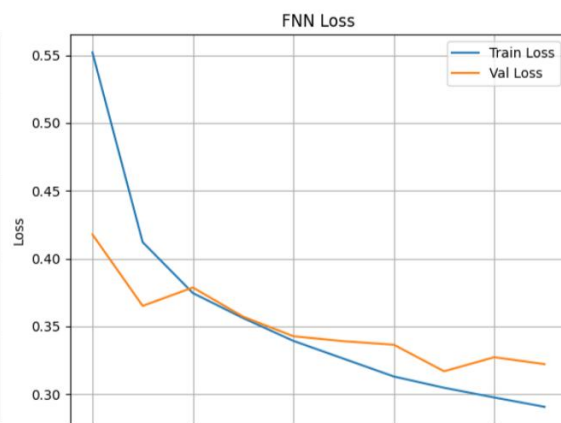
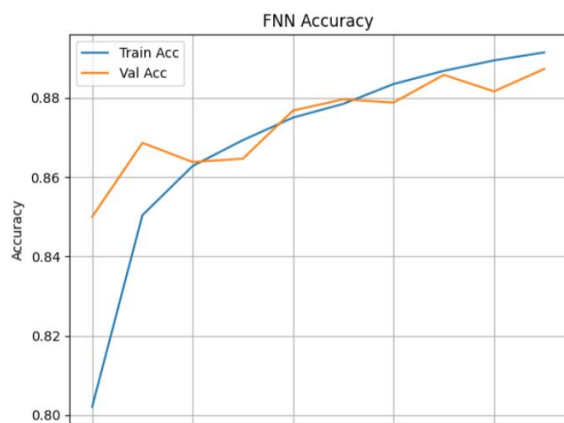
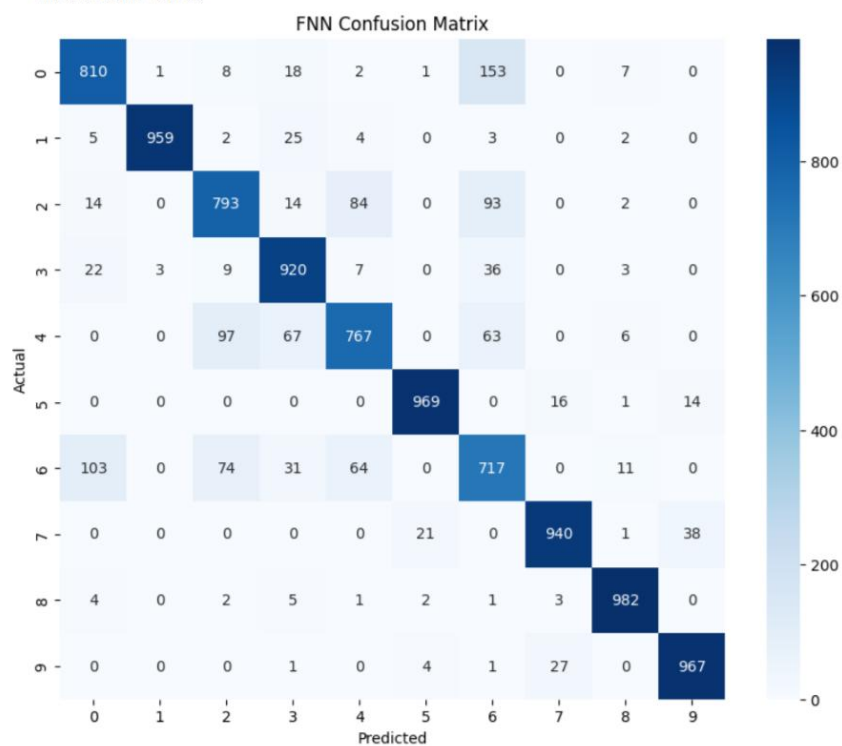
```

--- FNN Classification Report ---

```

	precision	recall	f1-score	support
0	0.85	0.81	0.83	1000
1	1.00	0.96	0.98	1000
2	0.81	0.79	0.80	1000
3	0.85	0.92	0.88	1000
4	0.83	0.77	0.80	1000
5	0.97	0.97	0.97	1000
6	0.67	0.72	0.69	1000
7	0.95	0.94	0.95	1000
8	0.97	0.98	0.97	1000
9	0.95	0.97	0.96	1000
accuracy			0.88	10000
macro avg	0.88	0.88	0.88	10000
weighted avg	0.88	0.88	0.88	10000

--- FNN Confusion Matrix ---

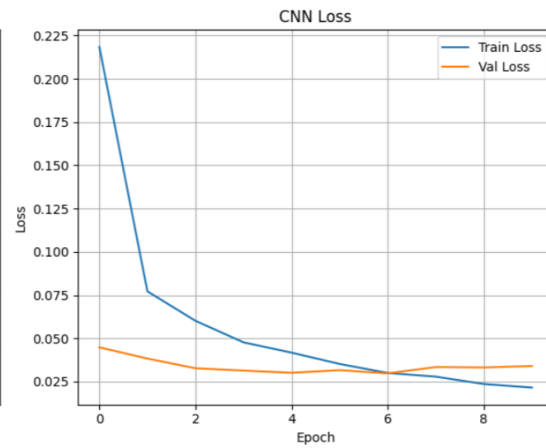
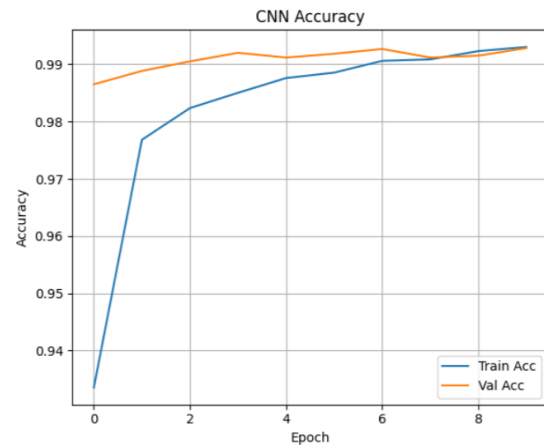
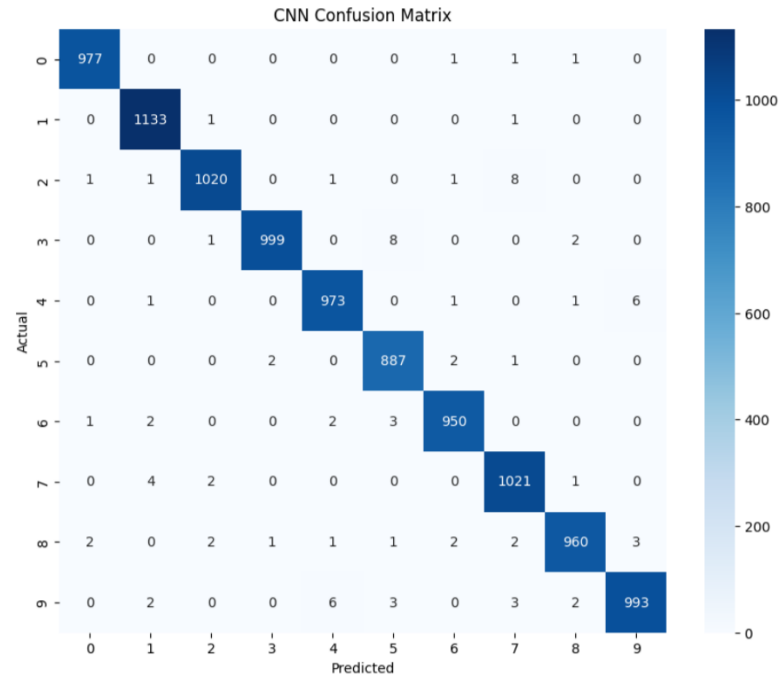


CNN Test Loss: 0.0285  
CNN Test Accuracy: 0.9913

--- CNN Classification Report ---



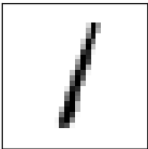
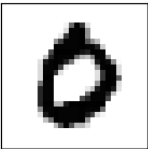
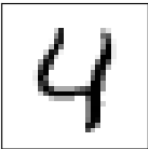
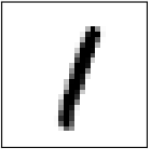


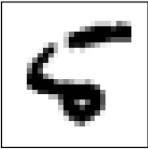

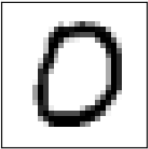


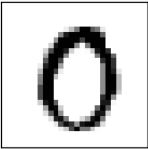
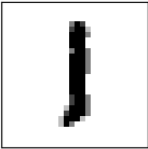
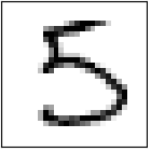

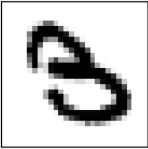




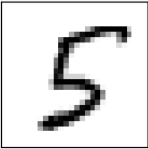

	precision	recall	f1-score	support
0	1.00	1.00	1.00	980
1	0.99	1.00	0.99	1135
2	0.99	0.99	0.99	1032
3	1.00	0.99	0.99	1010
4	0.99	0.99	0.99	982
5	0.98	0.99	0.99	892
6	0.99	0.99	0.99	958
7	0.98	0.99	0.99	1028
8	0.99	0.99	0.99	974
9	0.99	0.98	0.99	1009
accuracy			0.99	10000
macro avg	0.99	0.99	0.99	10000
weighted avg	0.99	0.99	0.99	10000

--- CNN Confusion Matrix ---



--- Sample CNN Predictions ---

CNN Predictions (Green = Correct, Red = Incorrect)

 True: 7 Pred: 7	 True: 2 Pred: 2	 True: 1 Pred: 1	 True: 0 Pred: 0	 True: 4 Pred: 4
 True: 1 Pred: 1	 True: 4 Pred: 4	 True: 9 Pred: 9	 True: 5 Pred: 5	 True: 9 Pred: 9
 True: 0 Pred: 0	 True: 6 Pred: 6	 True: 9 Pred: 9	 True: 0 Pred: 0	 True: 1 Pred: 1
 True: 5 Pred: 5	 True: 9 Pred: 9	 True: 7 Pred: 7	 True: 3 Pred: 5	 True: 4 Pred: 4
 True: 9 Pred: 9	 True: 6 Pred: 6	 True: 6 Pred: 6	 True: 5 Pred: 5	 True: 4 Pred: 4



<b>Exp No: 7</b>	<b>Generative Models with GANs: Creating and Training a Generative Adversarial Network</b>
------------------	--

### **Aim:**

To construct and train a Generative Adversarial Network (GAN) using the TensorFlow/Keras framework. The objective is to train the GAN on the MNIST dataset to generate new, synthetic images of handwritten digits that are indistinguishable from the original training data.

### **Algorithm:**

#### **Generative Adversarial Networks (GANs)**

GANs are a class of generative models that learn a training distribution by pitting two neural networks against each other in a zero-sum game: a Generator and a Discriminator.

**1. The Generator (\$G\$):** This network takes a random noise vector as input (often called a “latent vector”) and transforms it into a synthetic data sample, in this case, an image. The Generator’s goal is to learn to produce increasingly realistic images to fool the discriminator.

**2. The Discriminator (\$D\$):** This is a binary classifier network. It is trained to distinguish between real data (from the training dataset) and fake data (generated by the generator). Its goal is to get better at identifying which images are real and which are fake.

#### **3. The Adversarial Process:**

Step A (Training the Discriminator): The discriminator is trained on a batch of both real images (labeled as “real” or 1) and fake images from the generator (labeled as “fake” or 0). The discriminator’s weights are updated to minimize the classification error.

Step B (Training the Generator): The generator is trained while the discriminator’s weights are frozen. The generator creates fake images and feeds them to the discriminator. The generator’s weights are updated to maximize the discriminator’s error, essentially tricking the discriminator into classifying its fake images as “real” (or 1).

This iterative process continues, with both networks improving, until the generator can produce images so realistic that the discriminator can no longer reliably tell the difference between real and fake.

### **CODE:**

```
# Import necessary libraries
```

```

import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.datasets import mnist
import os

# Suppress TensorFlow warnings for cleaner output
tf.keras.utils.disable_interactive_logging()

# --- Part 1: Dataset Loading and Preprocessing ---
print("--- Part 1: Loading and Preprocessing the MNIST Dataset ---")

(x_train, _), (_, _) = mnist.load_data()

x_train = x_train.reshape(x_train.shape[0], 28, 28, 1).astype('float32')
x_train = (x_train - 127.5) / 127.5 # Normalize to [-1, 1]

print(f"Normalized training data shape: {x_train.shape}")
print("Example of a normalized pixel value:", x_train[0, 0, 0, 0])

# --- Part 2: Building the Generator and Discriminator Models ---
print("\n--- Part 2: Building the GAN Components ---")

latent_dim = 100

# Generator
def build_generator():
    model = keras.Sequential(name="generator")
    model.add(layers.Dense(7 * 7 * 256, use_bias=False, input_shape=(latent_dim,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Reshape((7, 7, 256)))
    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same',
    use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',
    use_bias=False))

```

```

    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same',
                                     use_bias=False, activation='tanh'))
    return model

generator = build_generator()
print("\n--- Generator Model Summary ---")
generator.summary()

# Discriminator
def build_discriminator():
    model = keras.Sequential(name="discriminator")
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))
    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))
    model.add(layers.Flatten())
    model.add(layers.Dense(1, activation='sigmoid'))
    return model

discriminator = build_discriminator()
print("\n--- Discriminator Model Summary ---")
discriminator.summary()

# --- Part 3: Training Setup ---
cross_entropy = keras.losses.BinaryCrossentropy(from_logits=False)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    return real_loss + fake_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

generator_optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4)

```

```

discriminator_optimizer = tf.keras.optimizers.Adam(learning_rate=1e-4)

@tf.function
def train_step(images, latent_dim=latent_dim):
    noise = tf.random.normal([batch_size, latent_dim])
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)
        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)
        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)
        gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
        gradients_of_discriminator = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)
        generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))
        discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))
    return gen_loss, disc_loss

def generate_and_save_images(model, epoch, test_input):
    predictions = model(test_input, training=False)
    predictions_rescaled = (predictions * 0.5) + 0.5 # Scale back to [0, 1]
    fig = plt.figure(figsize=(4, 4))
    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i + 1)
        plt.imshow(predictions_rescaled[i, :, :, 0], cmap='gray')
        plt.axis('off')
    plt.suptitle(f"Epoch {epoch}", fontsize=16)
    if not os.path.exists('images'):
        os.makedirs('images')
    plt.savefig(f'images/image_at_epoch_{epoch:04d}.png')
    plt.show()

# Training parameters
EPOCHS = 200
batch_size = 256
num_examples_to_generate = 16
seed = tf.random.normal([num_examples_to_generate, latent_dim])

```

```

train_dataset
tf.data.Dataset.from_tensor_slices(x_train).shuffle(x_train.shape[0]).batch(batch_size)

# Training loop
def train(dataset, epochs):
    print("\n--- Beginning GAN Training ---")
    for epoch in range(epochs):
        gen_loss_list = []
        disc_loss_list = []
        for image_batch in dataset:
            gen_loss, disc_loss = train_step(image_batch)
            gen_loss_list.append(gen_loss.numpy())
            disc_loss_list.append(disc_loss.numpy())
        avg_gen_loss = np.mean(gen_loss_list)
        avg_disc_loss = np.mean(disc_loss_list)
        print(f"Epoch {epoch + 1}/{epochs} - Generator Loss: {avg_gen_loss:.4f}, Discriminator
Loss: {avg_disc_loss:.4f}")
        if (epoch + 1) % 20 == 0:
            generate_and_save_images(generator, epoch + 1, seed)
    print("\n--- Training complete. Generating final images. ---")
    generate_and_save_images(generator, epochs, seed)

# Run training
train(train_dataset, EPOCHS)

```

**OUTPUT:**

--- Part 1: Loading and Preprocessing the MNIST Dataset ---

Normalized training data shape: (60000, 28, 28, 1)

Example normalized pixel value: -1.0

--- Beginning GAN Training ---

Epoch 1/20 - Generator Loss: 0.7877, Discriminator Loss: 1.0228

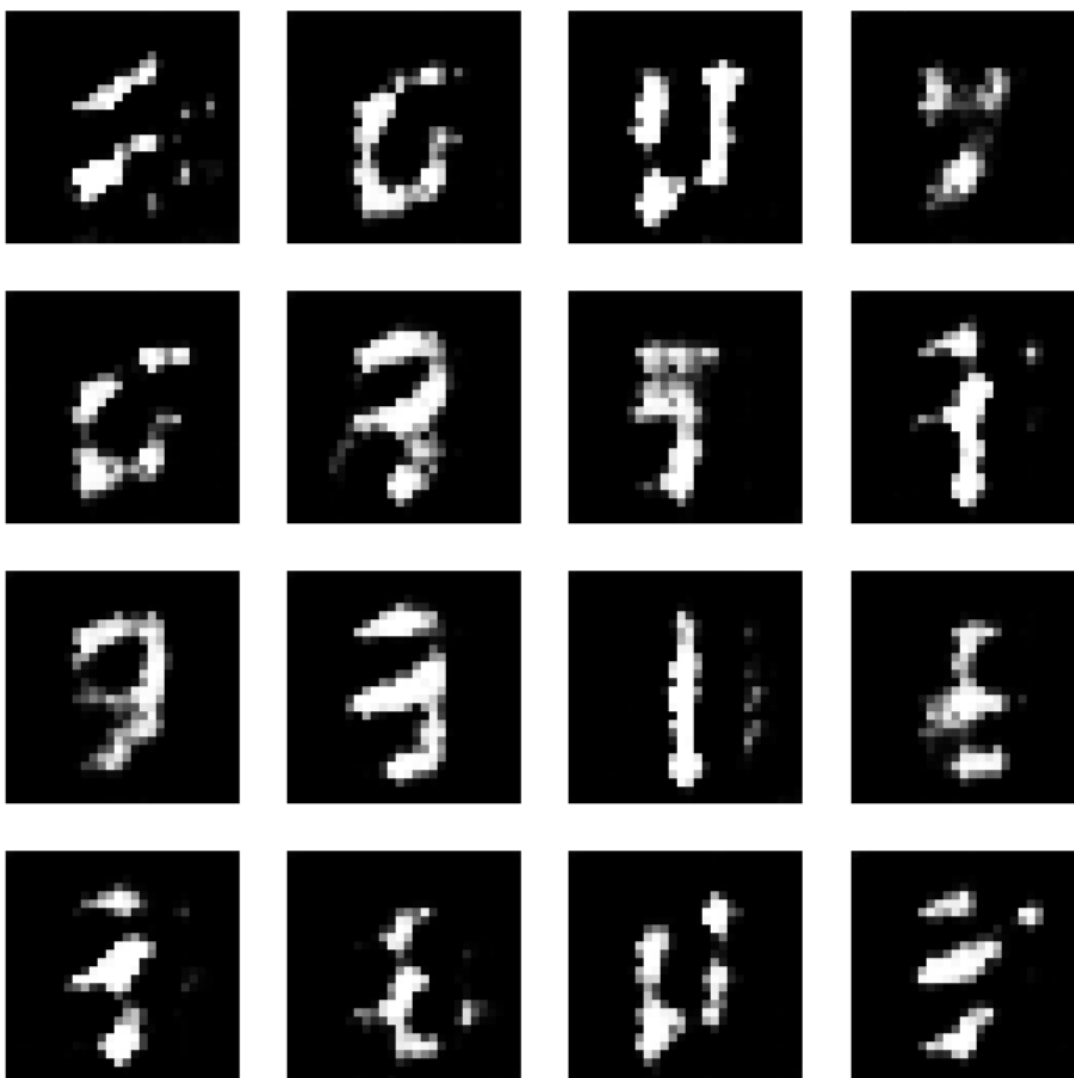
Epoch 2/20 - Generator Loss: 0.8148, Discriminator Loss: 1.2225

Epoch 3/20 - Generator Loss: 0.8448, Discriminator Loss: 1.3034

Epoch 4/20 - Generator Loss: 0.8534, Discriminator Loss: 1.2366

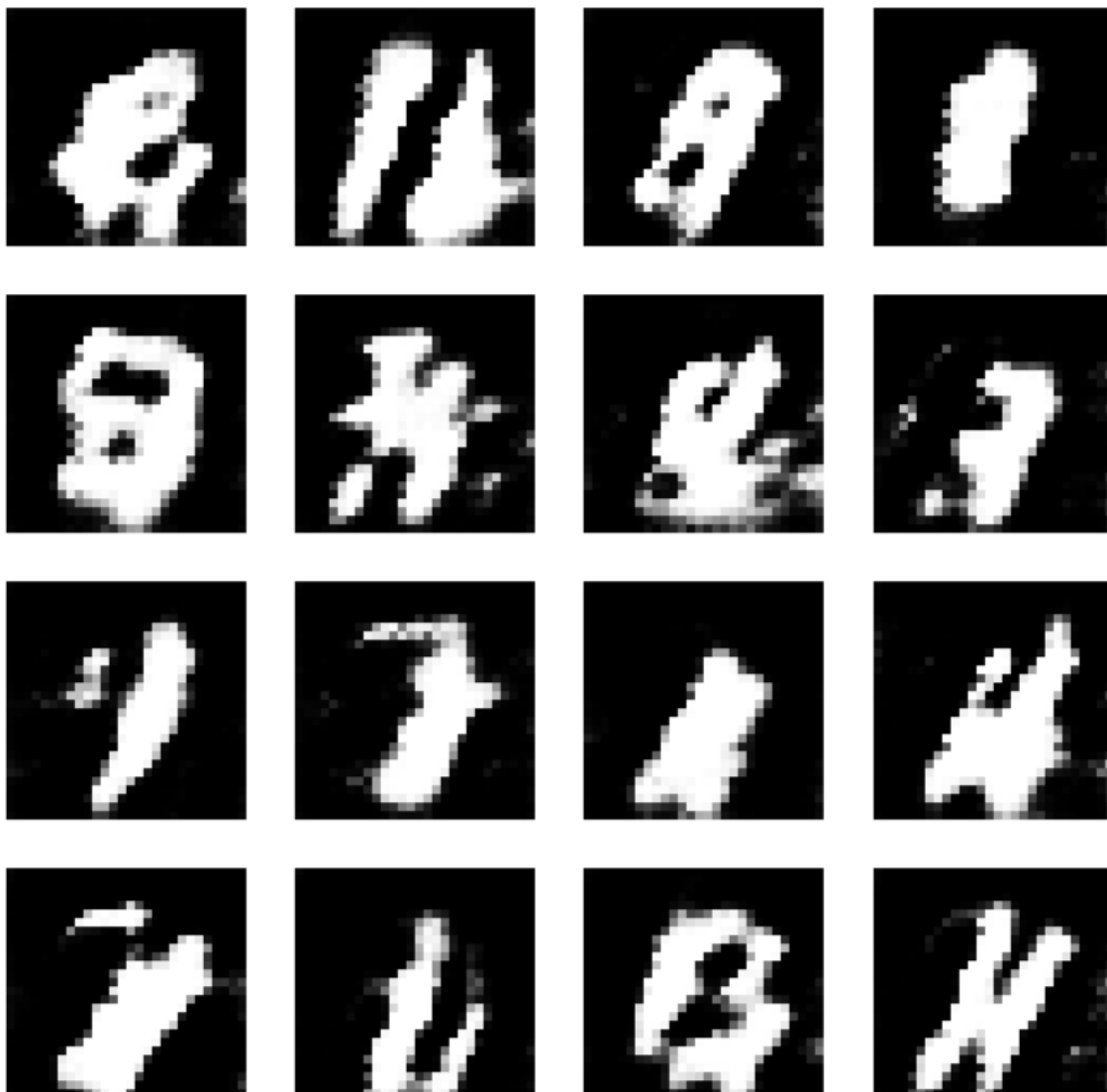
Epoch 5/20 - Generator Loss: 0.8372, Discriminator Loss: 1.2497

Epoch 5



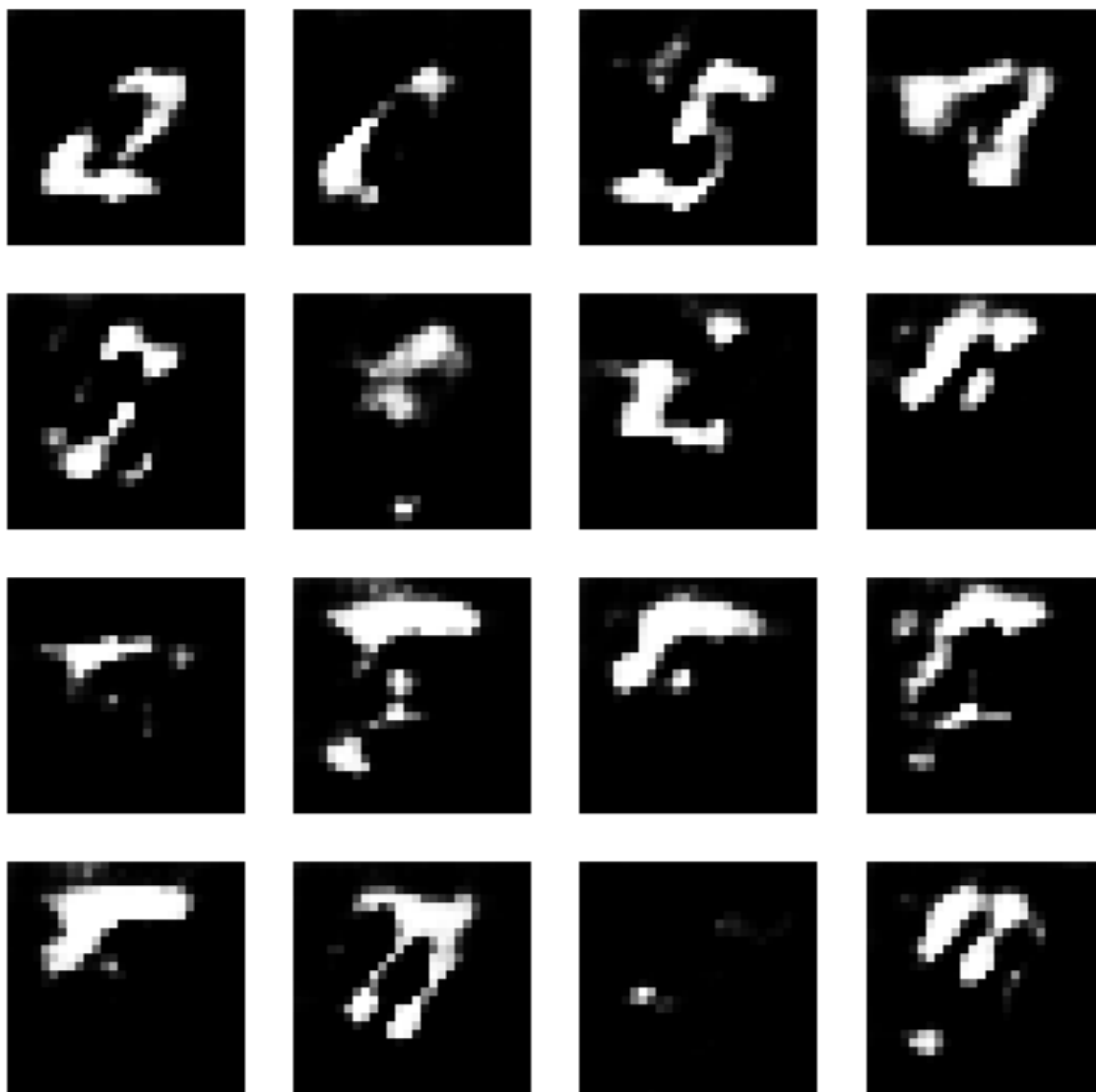
Epoch 6/20 - Generator Loss: 0.8516, Discriminator Loss: 1.2705  
Epoch 7/20 - Generator Loss: 0.8888, Discriminator Loss: 1.3028  
Epoch 8/20 - Generator Loss: 0.8739, Discriminator Loss: 1.2512  
Epoch 9/20 - Generator Loss: 0.8691, Discriminator Loss: 1.3130  
Epoch 10/20 - Generator Loss: 0.8862, Discriminator Loss: 1.2320

### Epoch 10



Epoch 11/20 - Generator Loss: 0.9361, Discriminator Loss: 1.2244  
Epoch 12/20 - Generator Loss: 0.9946, Discriminator Loss: 1.1719  
Epoch 13/20 - Generator Loss: 0.9948, Discriminator Loss: 1.1944  
Epoch 14/20 - Generator Loss: 0.9786, Discriminator Loss: 1.1809  
Epoch 15/20 - Generator Loss: 1.0420, Discriminator Loss: 1.1079

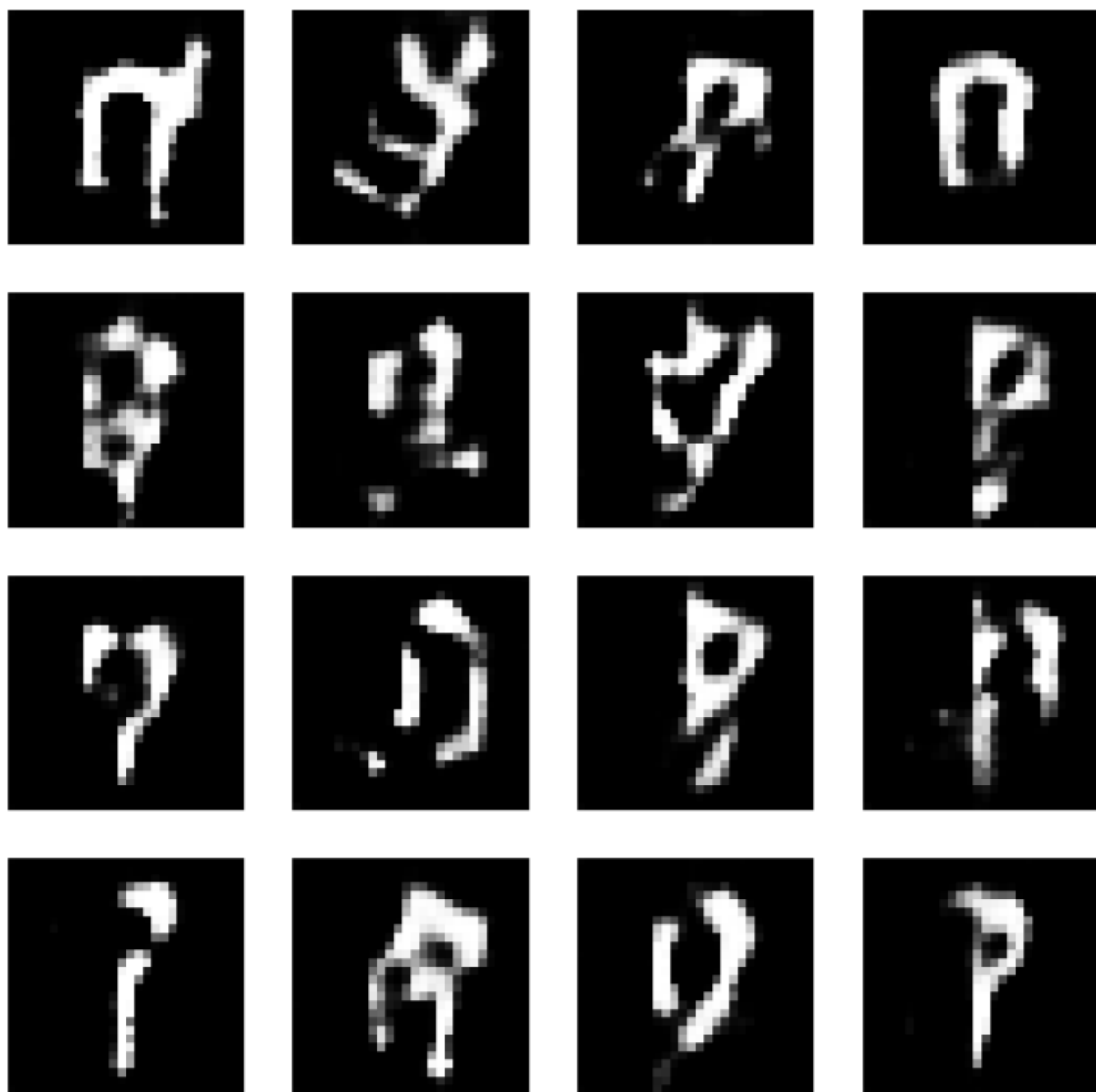
### Epoch 15





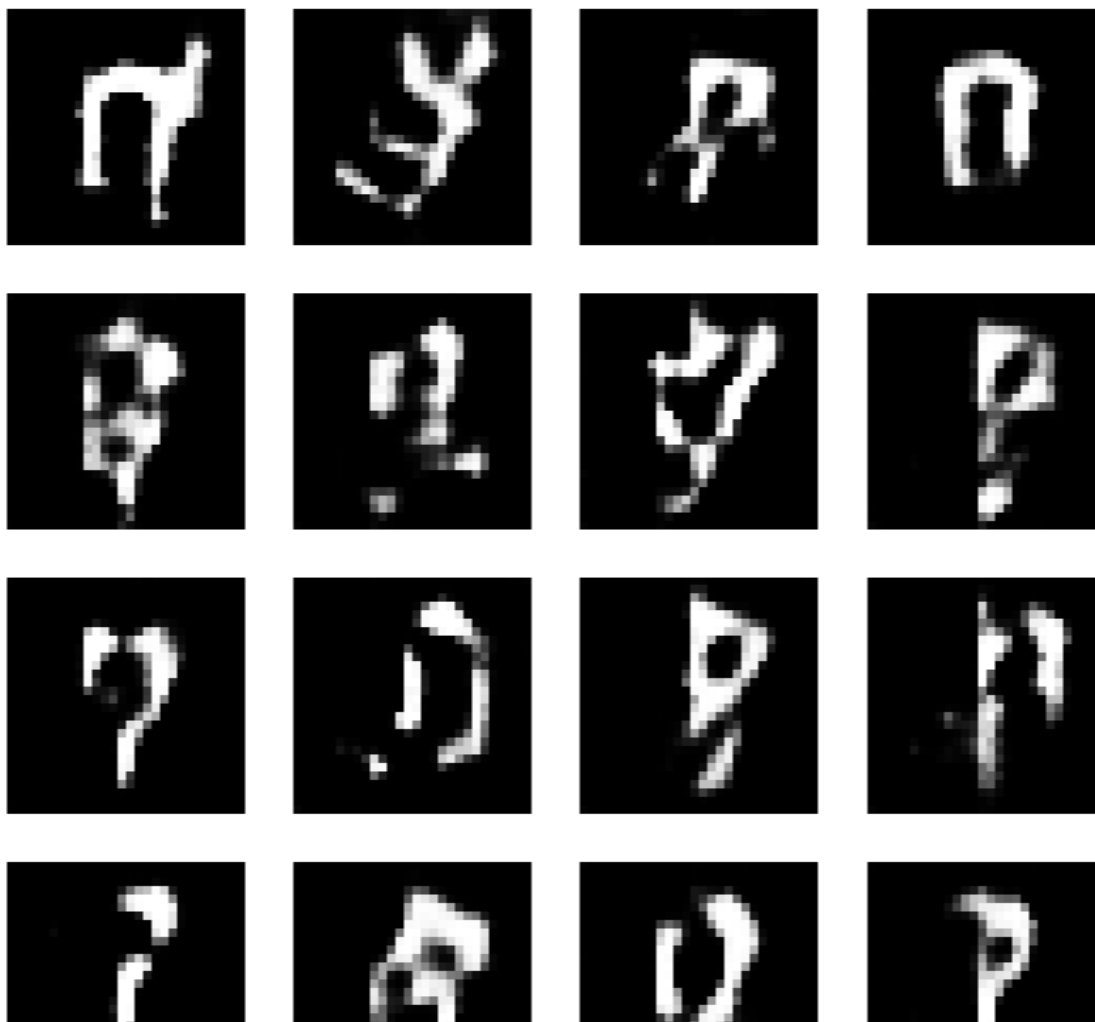
Epoch 16/20 - Generator Loss: 1.2020, Discriminator Loss: 1.0483  
Epoch 17/20 - Generator Loss: 1.2648, Discriminator Loss: 1.0605  
Epoch 18/20 - Generator Loss: 1.1657, Discriminator Loss: 1.0404  
Epoch 19/20 - Generator Loss: 1.1644, Discriminator Loss: 1.0897  
Epoch 20/20 - Generator Loss: 1.1770, Discriminator Loss: 1.0938

## Epoch 20



--- Training complete. Generating final images. ---

Epoch 20



<b>Exp No: 8</b>	<b>Model Evaluation and Improvement: Hyperparameter Tuning with Grid Search and Cross-Validation</b>
------------------	--

### **Aim:**

To demonstrate key techniques for model evaluation and improvement:

- 1. Hyperparameter Tuning with Grid Search :** Systematically searching for the optimal combination of hyperparameters for a machine learning model.
- 2. Cross-Validation Techniques:** Implementing k-fold cross-validation to get a more robust estimate of model performance and to prevent overfitting to a specific train-test split.

### **Algorithm:**

#### **1. Hyperparameter Tuning with Grid Search**

Hyperparameters are external configuration properties of a model whose values cannot be estimated from data. Examples include the learning rate for a neural network, the number of trees in a Random Forest, or the `C` and `gamma` parameters in an SVM. Tuning these parameters is crucial for optimal model performance.

**Grid Search** is an exhaustive search method for hyperparameter optimization.

#### **Steps:**

1. Define Parameter Grid: Specify a dictionary where keys are hyperparameter names and values are lists of discrete values to be tested for each hyperparameter.
2. Instantiate Model: Choose a machine learning model.
3. Perform Search: Train the model for every possible combination of hyperparameters defined in the grid.
4. Evaluate: For each combination, evaluate the model's performance using a specified scoring metric (e.g., accuracy, F1-score) and often in conjunction with cross-validation.
5. Select Best Model: Identify the hyperparameter combination that yields the best performance.

#### **2. Cross-Validation Techniques**

Cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample. The goal is to estimate how accurately a predictive model will perform in practice. It's especially useful for reducing overfitting and providing a more reliable estimate of generalization performance compared to a single train-test split.

## **k-Fold Cross-Validation:**

### **Steps:**

1. Divide Data: The entire dataset is randomly partitioned into  $k$  equally sized subsamples (or “folds”).

2. Iterate  $k$  Times:

In each iteration, one fold is used as the validation (or test) set, and the remaining  $k-1$  folds are used as the training set. The model is trained on the training set and evaluated on the validation set.

3. Aggregate Results: The performance metric (e.g., accuracy) from each of the  $k$  iterations is collected.

4. Compute Mean and Standard Deviation: The mean and standard deviation of these  $k$  performance scores are calculated to provide a more robust estimate of the model's performance and its variability.

### **CODE:**

```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.datasets import load_iris # A classic dataset for classification
from sklearn.model_selection import train_test_split, KFold, cross_val_score, GridSearchCV
from sklearn.svm import SVC # Support Vector Classifier, a common model for tuning
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.preprocessing import StandardScaler

# --- Part 1: Hyperparameter Tuning with Grid Search ---

print("--- Part 1: Hyperparameter Tuning with Grid Search ---")

# 1. Load a Dataset (Iris Dataset for classification)
# The Iris dataset is a classic and simple dataset for classification tasks.
# It contains measurements of iris flowers (sepal length, sepal width, petal length, petal width)
# and their corresponding species (Setosa, Versicolor, Virginica).
iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names
```

```

print(f"\nDataset Features (X) shape: {X.shape}")
print(f"Dataset Labels (y) shape: {y.shape}")
print(f"Feature Names: {feature_names}")
print(f"Target Names: {target_names}")

# 2. Split Data into Training and Testing Sets
# It's crucial to split the data before scaling to prevent data leakage.
# The test set will be used for final model evaluation, after tuning.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42,
stratify=y)

print(f"\nTraining set size: {X_train.shape[0]} samples")
print(f"Test set size: {X_test.shape[0]} samples")

# 3. Standardize Features
# Scaling features is important for SVMs as they are sensitive to feature scales.
# Fit scaler only on training data to prevent data leakage.
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("\nFeatures standardized.")

# 4. Define the Model and Hyperparameter Grid
# We'll use a Support Vector Classifier (SVC) as our model.
# Common hyperparameters for SVC are 'C' (regularization parameter) and 'gamma' (kernel
coefficient).
# 'kernel' also can be tuned (e.g., 'linear', 'rbf').

# Define the parameter grid for Grid Search
param_grid = {
    'C': [0.1, 1, 10, 100],      # Regularization parameter
    'gamma': [1, 0.1, 0.01, 0.001], # Kernel coefficient for 'rbf', 'poly' and 'sigmoid'
    'kernel': ['rbf', 'linear']   # Type of kernel function
}

print("\nHyperparameter grid defined:")
for param, values in param_grid.items():
    print(f" {param}: {values}")

```

```

# 5. Perform Grid Search with Cross-Validation
# GridSearchCV automatically performs k-fold cross-validation for each combination.
# cv=5 means 5-fold cross-validation.
# scoring='accuracy' means we want to optimize for accuracy.
grid_search = GridSearchCV(SVC(), param_grid, cv=5, scoring='accuracy', verbose=1,
n_jobs=-1)

print("\nStarting Grid Search with 5-fold Cross-Validation...")
# Fit GridSearchCV on the scaled training data
grid_search.fit(X_train_scaled, y_train)

print("\nGrid Search completed.")

# 6. Get the Best Parameters and Best Score
print(f"\nBest hyperparameters found: {grid_search.best_params_}")
print(f"Best cross-validation accuracy: {grid_search.best_score_:.4f}")

# 7. Evaluate the Best Model on the Test Set
# The best_estimator_ attribute provides the model trained with the best parameters.
best_model = grid_search.best_estimator_
y_pred_tuned = best_model.predict(X_test_scaled)

test_accuracy_tuned = accuracy_score(y_test, y_pred_tuned)
print(f"\nTest set accuracy with tuned model: {test_accuracy_tuned:.4f}")

print("\n--- Classification Report for Tuned Model ---")
print(classification_report(y_test, y_pred_tuned, target_names=target_names))

print("\n--- Confusion Matrix for Tuned Model ---")
cm_tuned = confusion_matrix(y_test, y_pred_tuned)
plt.figure(figsize=(8, 6))
sns.heatmap(cm_tuned, annot=True, fmt='d', cmap='Blues', xticklabels=target_names,
yticklabels=target_names)
plt.title('Confusion Matrix (Tuned SVM)')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()

# Visualize Grid Search results (optional, but good for understanding)

```

```

# Convert results to a DataFrame for easier analysis
results_df = pd.DataFrame(grid_search.cv_results_)
print("\n--- Top 5 Grid Search Results ---")
print(results_df[['param_C',      'param_gamma',      'param_kernel',      'mean_test_score',
'rank_test_score']].sort_values(by='rank_test_score').head())

# --- Part 2: Cross-Validation Techniques (k-fold) ---

print("\n--- Part 2: Cross-Validation Techniques (k-fold) ---")

# We will demonstrate k-fold cross-validation on a simple SVM without explicit tuning for
clarity,
# to focus solely on the CV process.

# 1. Instantiate a Model (using default or chosen parameters)
model_cv = SVC(random_state=42) # Using default parameters for simplicity

# 2. Define k-fold Cross-Validation Strategy
# We'll use 5-fold cross-validation.
# KFold ensures that each fold is distinct.
# shuffle=True means the data will be randomly shuffled before splitting into folds.
# random_state for reproducibility.
k_folds = 5
kf = KFold(n_splits=k_folds, shuffle=True, random_state=42)

print(f"\nPerforming {k_folds}-fold cross-validation...")

# 3. Perform Cross-Validation and Get Scores
# cross_val_score performs the KFold splitting, training, and evaluation automatically.
# It returns an array of scores, one for each fold.
cv_scores = cross_val_score(model_cv, X_train_scaled, y_train, cv=kf, scoring='accuracy')

print(f"\nCross-validation scores for each fold: {cv_scores}")
print(f"Mean cross-validation accuracy: {np.mean(cv_scores):.4f}")
print(f"Standard deviation of cross-validation accuracy: {np.std(cv_scores):.4f}")

# 4. Visualize Cross-Validation Scores
plt.figure(figsize=(8, 5))
plt.bar(range(1, k_folds + 1), cv_scores, color='skyblue')
plt.axhline(y=np.mean(cv_scores), color='r', linestyle='--', label=f'Mean Accuracy')

```

```
({np.mean(cv_scores):.4f}))')
plt.title(f'{k_folds}-Fold Cross-Validation Accuracy Scores')
plt.xlabel('Fold Number')
plt.ylabel('Accuracy')
plt.ylim(0.8, 1.0) # Set y-axis limits for better visualization
plt.legend()
plt.grid(axis='y', linestyle='--')
plt.show()
```

# 5. Discuss why CV is useful

```
print("\n--- Why is Cross-Validation Important? ---")
print("1. More Reliable Performance Estimate: Reduces bias from a single train-test split.")
print("2. Better Generalization: Helps ensure the model performs well on unseen data.")
print("3. Efficient Data Usage: All data points are used for both training and validation across different folds.")
print("4. Detects Overfitting/Underfitting: Variability in scores can indicate instability.")
```

## OUTPUT:

```
--- Part 1: Hyperparameter Tuning with Grid Search ---

Dataset Features (X) shape: (150, 4)
Dataset Labels (y) shape: (150,)
Feature Names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Target Names: ['setosa' 'versicolor' 'virginica']

Training set size: 105 samples
Test set size: 45 samples

Features standardized.

Hyperparameter grid defined:
C: [0.1, 1, 10, 100]
gamma: [1, 0.1, 0.01, 0.001]
kernel: ['rbf', 'linear']

Starting Grid Search with 5-fold Cross-Validation...
Fitting 5 folds for each of 32 candidates, totalling 160 fits

Grid Search completed.

Best hyperparameters found: {'C': 1, 'gamma': 0.1, 'kernel': 'rbf'}
Best cross-validation accuracy: 0.9810

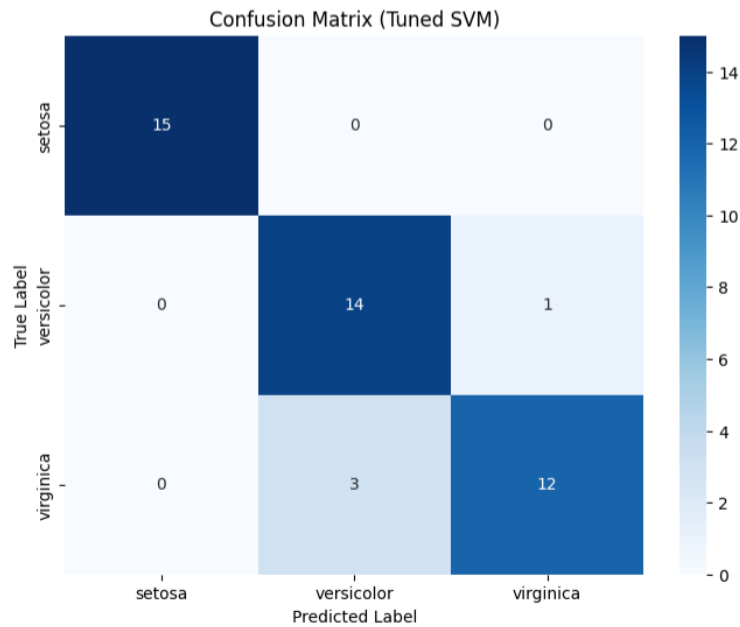
Test set accuracy with tuned model: 0.9111

--- Classification Report for Tuned Model ---
      precision    recall  f1-score   support

   setosa         1.00      1.00      1.00        15
  versicolor      0.82      0.93      0.88        15
   virginica      0.92      0.80      0.86        15

   accuracy                   0.91        45
  macro avg       0.92      0.91      0.91        45
 weighted avg       0.92      0.91      0.91        45
```





```

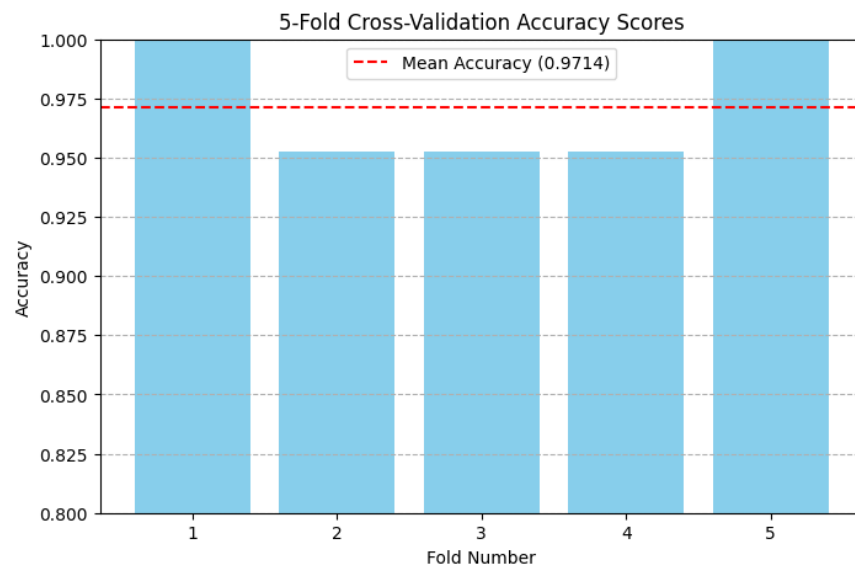
--- Top 5 Grid Search Results ---
param_C param_gamma param_kernel mean_test_score rank_test_score
10      1.0         0.100      rbf          0.980952           1
27     100.0         0.100      linear        0.980952           1
31     100.0         0.001      linear        0.980952           1
29     100.0         0.010      linear        0.980952           1
25     100.0         1.000      linear        0.980952           1

```

--- Part 2: Cross-Validation Techniques (k-fold) ---

Performing 5-fold cross-validation...

Cross-validation scores for each fold: [1. 0.95238095 0.95238095 0.95238095 1.]  
Mean cross-validation accuracy: 0.9714  
Standard deviation of cross-validation accuracy: 0.0233



--- Why is Cross-Validation Important? ---

1. More Reliable Performance Estimate: Reduces bias from a single train-test split.
2. Better Generalization: Helps ensure the model performs well on unseen data.
3. Efficient Data Usage: All data points are used for both training and validation across different folds.
4. Detects Overfitting/Underfitting: Variability in scores can indicate instability.

**ABSTRACT:**

Customer segmentation is a vital strategy in modern retail analytics, helping businesses understand and target their customers effectively. This project, Customer Segmentation for Retail Using K-Means and PCA, aims to group customers based on purchasing patterns and demographic attributes to support data-driven marketing decisions. By identifying segments such as high-income high-spenders or budget-conscious customers, retailers can design personalized offers and improve overall customer satisfaction.

The project employs the K-Means clustering algorithm to discover natural customer groupings and uses Principal Component Analysis (PCA) for dimensionality reduction and visual interpretation of clusters. The dataset is preprocessed by handling missing values, encoding categorical data, and standardizing numerical features. The Elbow Method and Silhouette Score are applied to determine the optimal number of clusters, ensuring meaningful segmentation and strong cluster separation.

An interactive Streamlit application is developed to make the model accessible and user-friendly. The app allows users to upload data, adjust clustering parameters, visualize customer clusters in real time, and predict the segment of new customers. This project demonstrates how unsupervised learning can transform raw retail data into actionable business insights, enabling smarter marketing strategies and more efficient customer relationship management.

**CODE:**

```
import streamlit as st
import pandas as pd
import numpy as np
import plotly.express as px
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
import io

# --- App Configuration ---
st.set_page_config(
    page_title="Retail Customer Segmentation",
    page_icon="🛒",
```

```
    layout="wide"  
)
```

```
# --- Abstract ---
```

```
APP_ABSTRACT = """
```

Customer segmentation is a vital strategy in modern retail analytics, helping businesses understand and target their customers effectively. This project, Customer Segmentation for Retail Using K-Means and PCA, aims to group customers based on purchasing patterns and demographic attributes to support data-driven marketing decisions. By identifying segments such as high-income high-spenders or budget-conscious customers, retailers can design personalized offers and improve overall customer satisfaction.

The project employs the K-Means clustering algorithm to discover natural customer groupings and uses Principal Component Analysis (PCA) for dimensionality reduction and visual interpretation of clusters. The dataset is preprocessed by handling missing values, encoding categorical data, and standardizing numerical features. The Elbow Method and Silhouette Score are applied to determine the optimal number of clusters, ensuring meaningful segmentation and strong cluster separation.

An interactive Streamlit application is developed to make the model accessible and user-friendly. The app allows users to upload data, adjust clustering parameters, visualize customer clusters in real time, and predict the segment of new customers. This project demonstrates how unsupervised learning can transform raw retail data into actionable business insights, enabling smarter marketing strategies and more efficient customer relationship management.

```
"""
```

```
# --- Helper Functions ---
```

```
@st.cache_data
```

```
def load_default_data():
```

```
    """Loads the default Mall Customers dataset."""
```

```
    csv_data = """CustomerID,Gender,Age,Annual Income (k$),Spending Score (1-100)
```

```
1,Male,19,15,39
```

```
2,Male,21,15,81
```

```
3,Female,20,16,6
```

```
4,Female,23,16,77
```

```
5,Female,31,17,40
```

```
6,Female,22,17,76
```

```
7,Female,35,18,6
```

```
8,Female,23,18,94
```

```
9,Male,64,19,3
```

```
10,Female,30,19,72
```

```
11,Male,67,19,14
```

```
12,Female,35,19,99
```

```
13,Female,58,20,15
```

```
14,Female,24,20,77
```

```

15,Male,37,20,13
16,Male,22,20,79
17,Female,35,21,35
18,Male,20,21,66
19,Male,52,23,29
20,Female,35,23,98
21,Male,35,24,35
22,Male,25,24,73
23,Female,46,25,5
24,Male,31,25,73
25,Female,54,28,14
26,Male,29,28,82
27,Female,45,28,32
28,Male,35,28,61
29,Female,40,29,31
30,Female,23,29,87
31,Male,60,30,4
32,Female,21,30,73
33,Male,53,33,4
34,Female,18,33,92
35,Female,49,33,14
36,Female,21,33,81
37,Female,42,34,17
38,Female,30,34,73
39,Female,36,37,26
40,Female,65,38,35
41,Female,20,38,92
42,Male,48,39,36
43,Female,24,39,65
44,Female,31,39,8
45,Female,49,39,28
46,Female,24,39,68
47,Female,50,40,55
48,Female,27,40,47
49,Female,29,40,42
50,Female,31,40,42
"""

```

```

    return pd.read_csv(io.StringIO(csv_data))

```

```

@st.cache_resource
def create_preprocessing_pipeline(df):
    """Creates a preprocessing pipeline based on dataframe columns."""

    # Identify numeric and categorical features (exclude CustomerID)

```

```

numeric_features = df.select_dtypes(include=np.number).columns.tolist()
if 'CustomerID' in numeric_features:
    numeric_features.remove('CustomerID')

categorical_features = df.select_dtypes(include=['object', 'category']).columns.tolist()

# Create transformers
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Create the full preprocessor
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ],
    remainder='passthrough'
)

return preprocessor, numeric_features, categorical_features

@st.cache_data
def calculate_cluster_metrics(_processed_data):
    """Calculates inertia and silhouette scores for a range of k."""
    inertia = []
    silhouette_scores = []
    k_range = range(2, 11)

    for k in k_range:
        kmeans = KMeans(n_clusters=k, init='k-means++', n_init=10, random_state=42)
        kmeans.fit(_processed_data)
        inertia.append(kmeans.inertia_)
        score = silhouette_score(_processed_data, kmeans.labels_)
        silhouette_scores.append(score)

    metrics_df = pd.DataFrame({
        'K': k_range,

```

```

        'Inertia (Elbow)': inertia,
        'Silhouette Score': silhouette_scores
    })
    return metrics_df

# --- Sidebar Controls ---
st.sidebar.title("🛠️ Segmentation Controls")
st.sidebar.markdown("Configure the analysis parameters.")

uploaded_file = st.sidebar.file_uploader("Upload your own CSV", type=["csv"])

if uploaded_file is not None:
    df = pd.read_csv(uploaded_file)
else:
    st.sidebar.info("Using default 'Mall Customers' dataset. Upload a CSV to use your own data.")
    df = load_default_data()

# Drop CustomerID if it exists, as it's just an identifier
if 'CustomerID' in df.columns:
    df_analysis = df.drop(columns=['CustomerID'])
else:
    df_analysis = df.copy()

# Create pipeline and process data
try:
    preprocessor, numeric_cols, cat_cols = create_preprocessing_pipeline(df_analysis)
    processed_data = preprocessor.fit_transform(df_analysis)

    st.sidebar.markdown("---")
    st.sidebar.header("Model Parameters")
    k_slider = st.sidebar.slider(
        "Select Number of Clusters (K)",
        min_value=2,
        max_value=10,
        value=5,
        step=1,
        help="Use the 'Optimal K Analysis' tab to find the best value for K."
    )

# --- Main Application ---
st.title("Retail Customer Segmentation Dashboard")
st.info(APP_ABSTRACT)

tab1, tab2, tab3, tab4 = st.tabs([

```

```

    "Data Overview",
    "Optimal K Analysis",
    "Segmentation Results",
    "Predict New Customer"
)

# --- Tab 1: Data Overview ---
with tab1:
    st.header("Data Preview")
    st.dataframe(df.head(), width='stretch')

    st.header("Data Statistics")
    st.dataframe(df.describe(), width='stretch')

    st.header("Features Identified")
    st.markdown(f"*Numeric Features:* {' '.join(numeric_cols)}")
    st.markdown(f"*Categorical Features:* {' '.join(cat_cols)}")

# --- Tab 2: Optimal K Analysis ---
with tab2:
    st.header("Finding the Optimal Number of Clusters (K)")
    st.markdown("""
    We use two methods to find the best 'K':
    1. *Elbow Method:* Look for the "elbow" point where the inertia (sum of squared distances) stops
    decreasing rapidly.
    2. *Silhouette Score:* Measures how similar an object is to its own cluster compared to other
    clusters. A higher score (closer to 1) is better.
    """)

    with st.spinner("Calculating cluster metrics for K=2 to 10..."):
        metrics_df = calculate_cluster_metrics(processed_data)

    col1, col2 = st.columns(2)

    with col1:
        st.subheader("Elbow Method (Inertia)")
        fig_elbow = px.line(metrics_df, x='K', y='Inertia (Elbow)', title='Elbow Method',
        markers=True)
        st.plotly_chart(fig_elbow, use_container_width=True) # Note: Keeping use_container_width as
        'width' is not a direct replacement for plotly

    with col2:
        st.subheader("Silhouette Score")
        fig_sil = px.line(metrics_df, x='K', y='Silhouette Score', title='Silhouette Score', markers=True)

```

```
st.plotly_chart(fig_sil, use_container_width=True) # Note: Keeping use_container_width as
'width' is not a direct replacement for plotly
```

```
# --- Tab 3: Segmentation Results ---
```

```
with tab3:
```

```
st.header(f"Customer Segments (K={k_slider})")
```

```
# Run K-Means with the selected K
```

```
kmeans_model = KMeans(n_clusters=k_slider, init='k-means++', n_init=10, random_state=42)
```

```
kmeans_model.fit(processed_data)
```

```
# Add cluster labels to the original dataframe
```

```
df_results = df_analysis.copy()
```

```
df_results['Cluster'] = kmeans_model.labels_.astype(str) # Convert to string for Plotly
```

```
# Run PCA for visualization
```

```
pca = PCA(n_components=2, random_state=42)
```

```
pca_features = pca.fit_transform(processed_data)
```

```
df_results['PC1'] = pca_features[:, 0]
```

```
df_results['PC2'] = pca_features[:, 1]
```

```
# Create PCA Scatter Plot
```

```
st.subheader("Customer Segments (PCA Visualization)")
```

```
hover_cols = numeric_cols + cat_cols + ['Cluster']
```

```
fig_pca = px.scatter(
```

```
    df_results,
```

```
    x='PC1',
```

```
    y='PC2',
```

```
    color='Cluster',
```

```
    title=f"2D PCA Visualization of {k_slider} Clusters",
```

```
    hover_data=[col for col in hover_cols if col in df_results.columns],
```

```
    color_discrete_sequence=px.colors.qualitative.Plotly
```

```
)
```

```
st.plotly_chart(fig_pca, use_container_width=True) # Note: Keeping use_container_width as
'width' is not a direct replacement for plotly
```

```
# Show Cluster Profiles
```

```
st.subheader("Cluster Profiles (Mean Values)")
```

```
st.markdown("This table shows the average values for each numeric feature by cluster.")
```

```
cluster_profile = df_results.groupby('Cluster')[numeric_cols].mean().reset_index()
```

```
st.dataframe(cluster_profile, width='stretch')
```

```
# --- Tab 4: Predict New Customer ---
```



```

with tab4:
    st.header("Predict New Customer Segment")
    st.markdown("Enter the details of a new customer to predict their segment.")

    # Use a form for input
    with st.form(key="prediction_form"):
        input_data = { }

        # Create input fields dynamically
        for col in numeric_cols:
            input_data[col] = st.number_input(
                f"Enter {col}",
                value=float(df_analysis[col].median())
            )

        for col in cat_cols:
            unique_vals = df_analysis[col].unique().tolist()
            input_data[col] = st.selectbox(
                f"Select {col}",
                options=unique_vals,
                index=0
            )

        submit_button = st.form_submit_button(label="Predict Segment")

    if submit_button:
        # Create a DataFrame from the input
        new_customer_df = pd.DataFrame([input_data])

        # Preprocess the new data
        try:
            new_data_processed = preprocessor.transform(new_customer_df)

            # Predict the cluster
            prediction = kmeans_model.predict(new_data_processed)

            st.success(f"This customer belongs to *Cluster {prediction[0]}*")

            # Show profile of the predicted cluster
            st.subheader(f"Profile of Predicted Cluster {prediction[0]}")
            st.dataframe(cluster_profile[cluster_profile['Cluster'] == str(prediction[0])], width='stretch')

        except Exception as e:
            st.error(f"An error occurred during prediction: {e}")

```

except Exception as e:

```
st.error(f"An error occurred during data processing: {e}")
```

```
st.error("Please check your uploaded CSV file. It may have an incorrect format, missing columns, or  
data types that don't match the expected schema (numeric and categorical).")
```

## Output:

The screenshot shows a Streamlit application titled "Retail Customer Segmentation Dashboard". On the left, there's a sidebar with "Segmentation Controls" and "Model Parameters". The main area has a title, a description of the project, and a section for "Predict New Customer Segment".

**Segmentation Controls**

Configure the analysis parameters.

Upload your own CSV

Drag and drop file here  
Limit 200MB per file • CSV

Browse files

Using default 'Mall Customers' dataset.  
Upload a CSV to use your own data.

**Model Parameters**

Select Number of Clusters (K)

5

**Retail Customer Segmentation Dashboard**

Customer segmentation is a vital strategy in modern retail analytics, helping businesses understand and target their customers effectively. This project, Customer Segmentation for Retail Using K-Means and PCA, aims to group customers based on purchasing patterns and demographic attributes to support data-driven marketing decisions. By identifying segments such as high-income high-spenders or budget-conscious customers, retailers can design personalized offers and improve overall customer satisfaction.

The project employs the K-Means clustering algorithm to discover natural customer groupings and uses Principal Component Analysis (PCA) for dimensionality reduction and visual interpretation of clusters. The dataset is preprocessed by handling missing values, encoding categorical data, and standardizing numerical features. The Elbow Method and Silhouette Score are applied to determine the optimal number of clusters, ensuring meaningful segmentation and strong cluster separation.

An interactive Streamlit application is developed to make the model accessible and user-friendly. The app allows users to upload data, adjust clustering parameters, visualize customer clusters in real time, and predict the segment of new customers. This project demonstrates how unsupervised learning can transform raw retail data into actionable business insights, enabling smarter marketing strategies and more efficient customer relationship management.

Data Overview Optimal K Analysis Segmentation Results **Predict New Customer**

**Predict New Customer Segment**

Enter the details of a new customer to predict their segment.

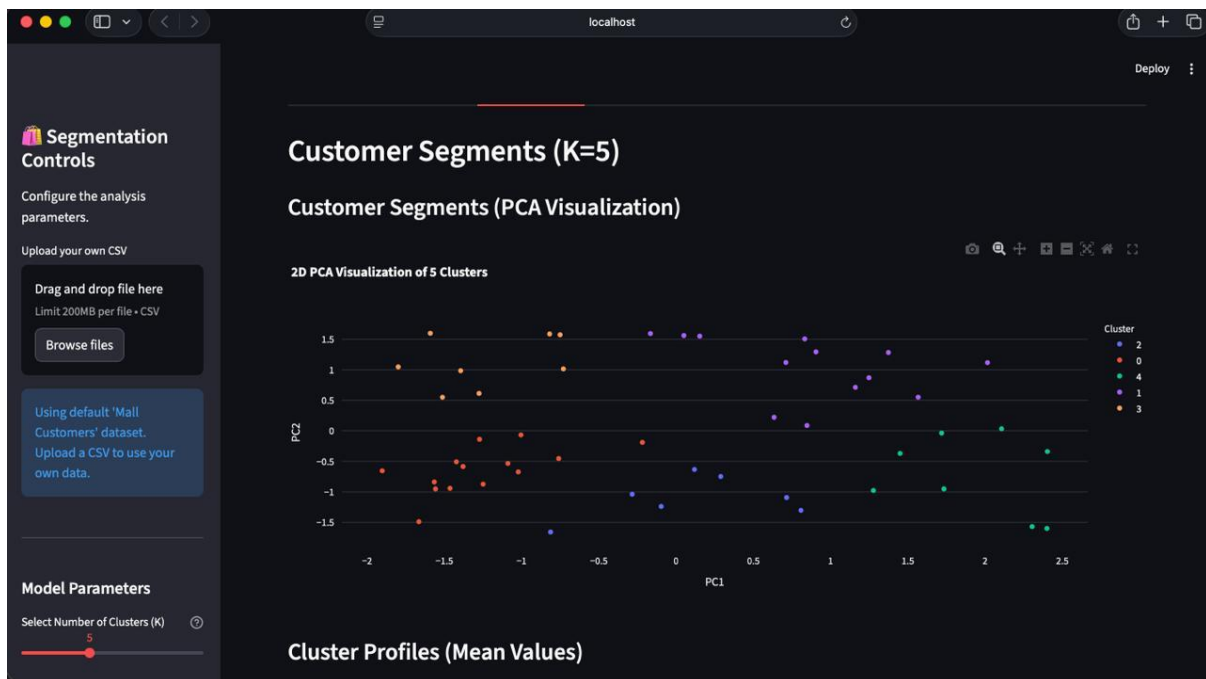
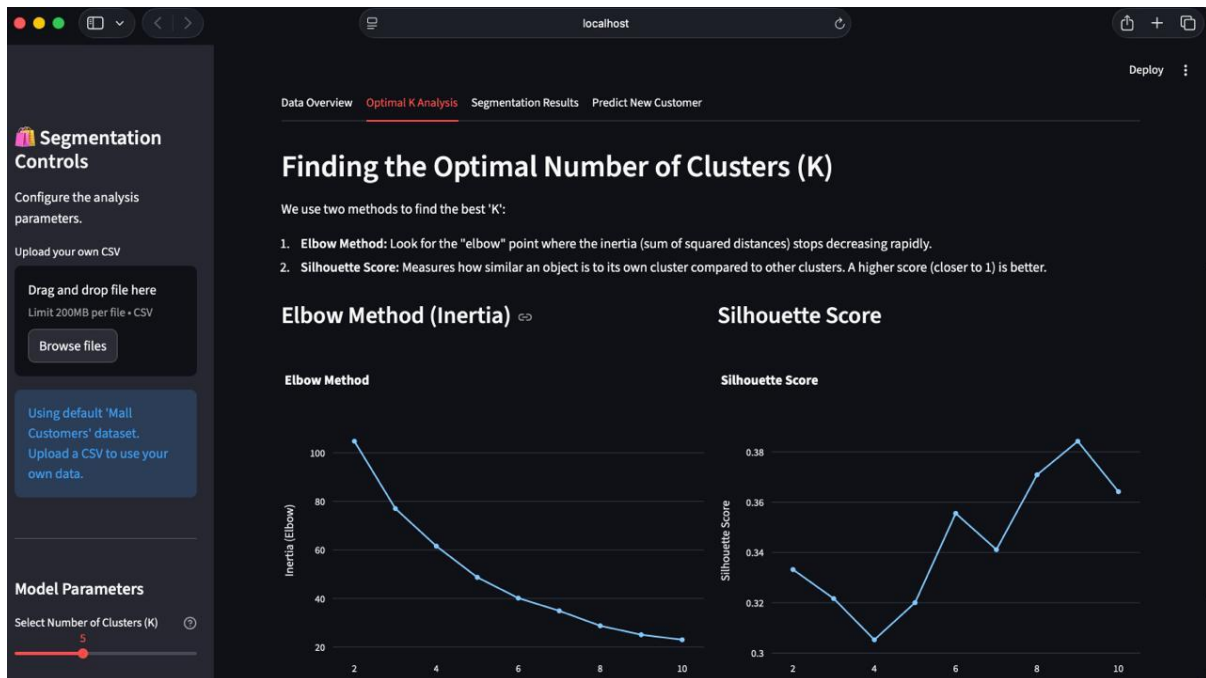
The screenshot shows the "Data Preview" and "Data Statistics" sections of the application. The "Data Preview" section displays a table of customer data, and the "Data Statistics" section displays a table of summary statistics.

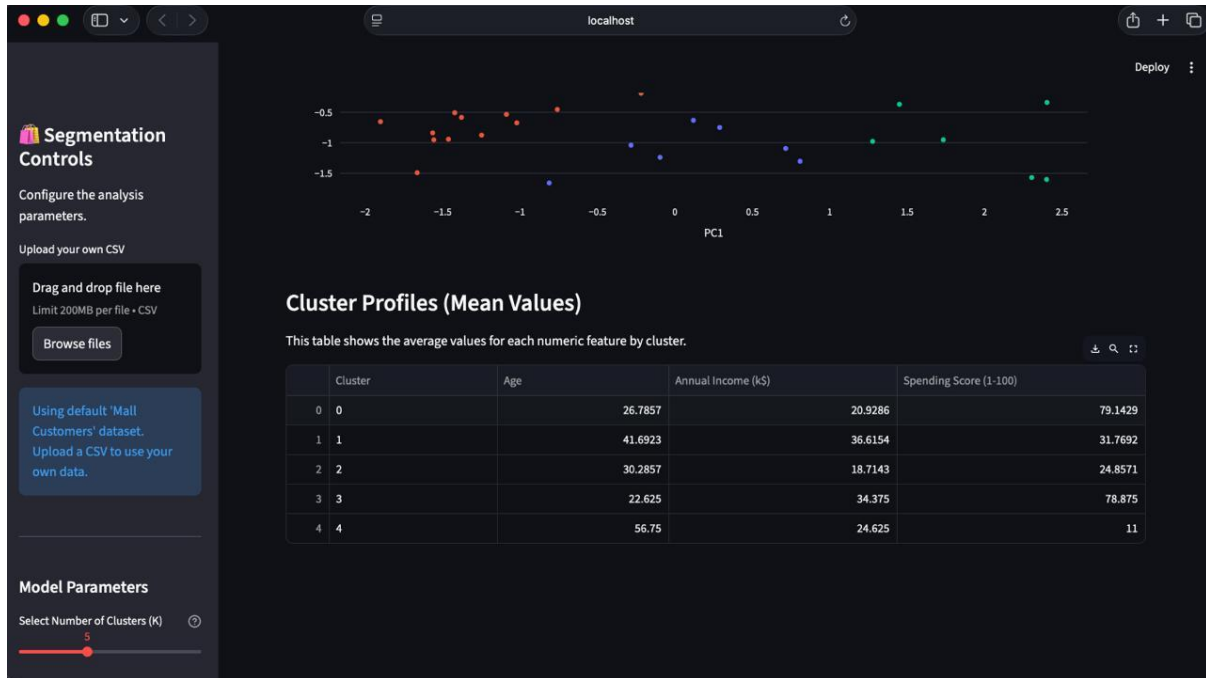
**Data Preview**

	CustomerID	Gender	Age	Annual Income (k\$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

**Data Statistics**

	CustomerID	Age	Annual Income (k\$)	Spending Score (1-100)
count	50	50	50	50
mean	25.5	35.28	27.44	48.28
std	14.5774	13.7515	8.4205	30.6228
min	1	18	15	3
25%	13.25	23.25	20	19.25
50%	25.5	31	28	42





The screenshot shows the same web application interface, but now focused on predicting a new customer segment. The sidebar remains the same. The main area is titled "Predict New Customer Segment" and includes a form to enter customer details: Age (31.00), Annual Income (k\$) (28.00), Spending Score (1-100) (42.00), and Gender (Male). A "Predict Segment" button is present. Below the form, a green banner indicates "This customer belongs to Cluster 2". At the bottom, a section titled "Profile of Predicted Cluster 2" shows a table with the mean values for Cluster 2.

Cluster	Age	Annual Income (k\$)	Spending Score (1-100)
2	30.2857	18.7143	24.8571