**Rajalakshmi Engineering College**

**(An Autonomous Institution) Rajalakshmi Nagar, Thandalam- 602105**

**DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND**

**MACHINE LEARNING**

**AI23521 – BUILD AND DEPLOY MACHINE LEARNING APPLICATIONS**

**(REGULATION 2023)**

**RAJALAKSHMI ENGINEERING COLLEGE**

**Thandalam, Chennai-602015**

**Name: Nithish Balaji B**

**Register No: 231501115**

**Year / Branch : 3rd / AIML**

**Semester: V**

**Academic Year: 2024 - 2025**

**RAJALAKSHMI ENGINEERING COLLEGE**

**(AUTONOMOUS)**

**RAJALAKSHMI NAGAR, THANDALAM**

**602 105**

**BONAFIDE CERTIFICATE**

NAME **Nithish Balaji B** REGISTER NO. **231501115**

ACADEMIC YEAR 2024-25 **SEMESTER**- V **BRANCH**: AIML-B.Tech

This Certification is the Bonafide record of work done by the above student

in the **AI23521 – Build and Deploy Machine Learning Applications** Laboratory during the year 2024  2025.

Signature of Faculty -in  Charge

Submitted for the Practical Examination held on

# INDEX

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Date** | **Topic** | **Page Number** |
| 1 | 7/8/25 | Setting Up the Environment And Preprocessing the Data | 1 |
| 2 | 14/8/25 | Support Vector Machine (SVM) and Random Forest for  Binary & Multiclass Classification | 6 |
| 3 | 14/8/25 | Classification with Decision Trees | 10 |
| 4(a) | 21/8/25 | Support Vector Machines (SVM) | 13 |
| 4(b) | 21/8/25 | Ensemble Methods: Random Forest | 18 |
| 5 | 28/8/25 | Clustering with K-Means and Dimensionality Reduction with PCA | 23 |
| 6 | 11/9/25 | Feedforward and Convolutional Neural Networks | 30 |
| 7 | 25/9/25 | Generative Models with GANs: Creating and Training a  Generative Adversarial Network | 40 |
| 8 | 9/10/25 | Model Evaluation and Improvement: Hyperparameter  Tuning with Grid Search and Cross-Validation | 50 |

|  |  |
| --- | --- |
| **Exp No: 1**  **Date : 7/8/25** | **Setting Up the Environment And Preprocessing the Data** |

**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])

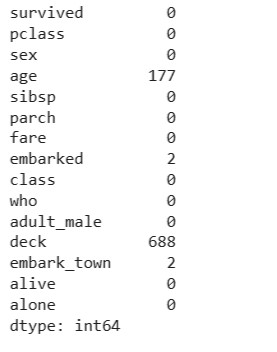
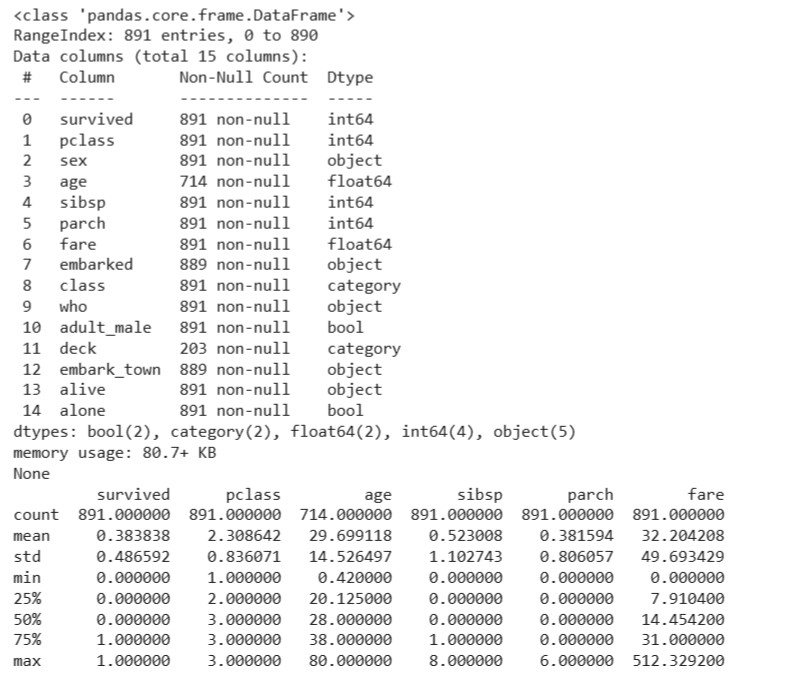
# 8. Split dataset

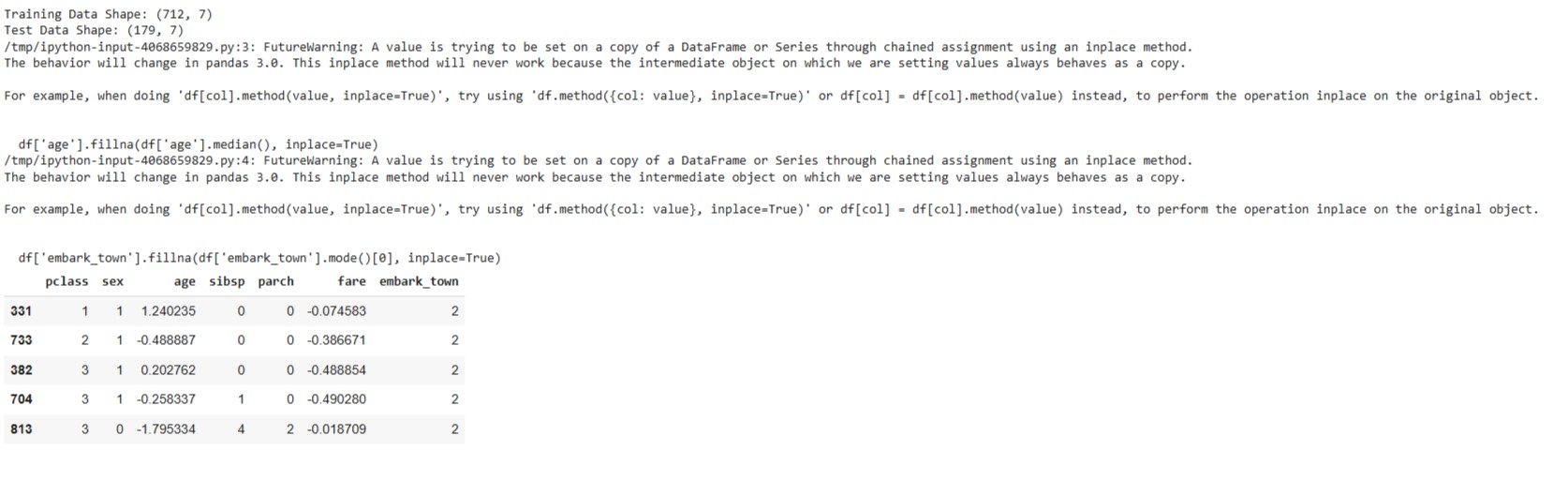
# Define features (X) and label (y) X = df.drop(‘survived’, axis=1) y = df[‘survived’]

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:**





**Result:**

The dataset was successfully preprocessed by handling missing values, encoding categorical features, scaling numerical attributes, and splitting into training and testing sets. The final cleaned and standardized data is now ready for use in machine learning model training and evaluation.

|  |  |
| --- | --- |
| **Exp No: 2**    **Date : 14/8/25** | **Support Vector Machine (SVM) and Random Forest for Binary & Multiclass Classification** |

**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

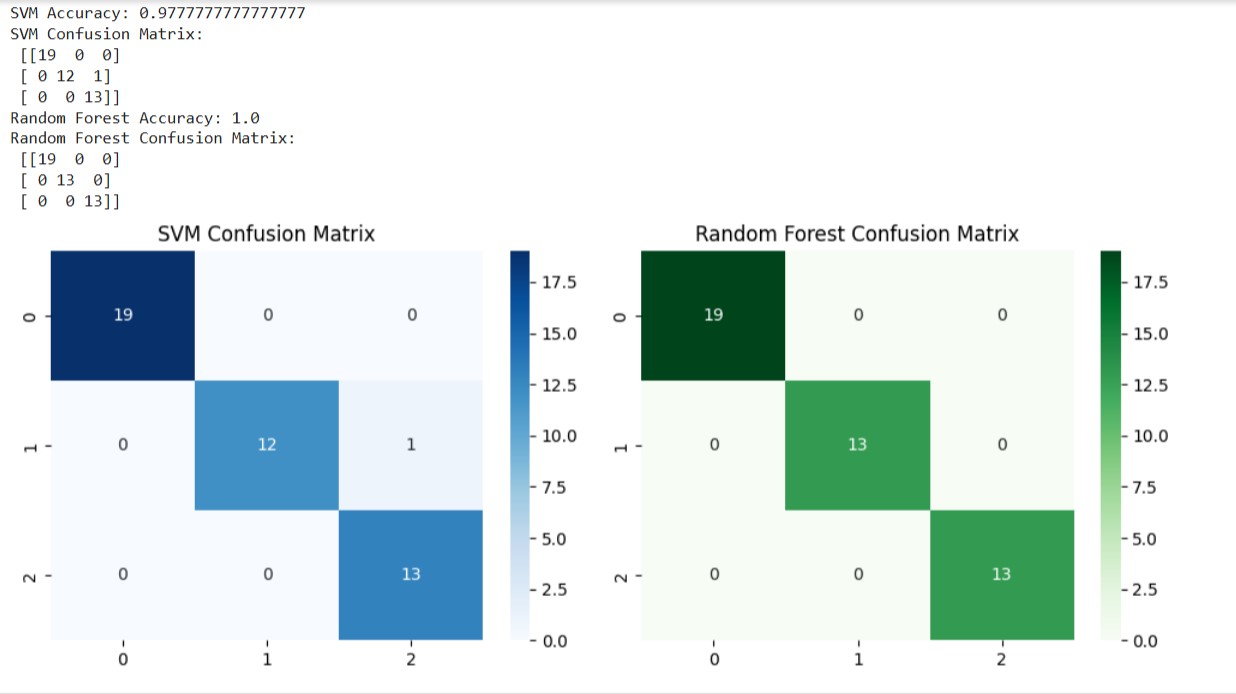
# ----------------------------------

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:**



**Result:**

The SVM and Random Forest models were successfully implemented for the Iris dataset. The SVM achieved high accuracy (~97%), while the Random Forest performed slightly better (~100%), demonstrating its robustness and ensemble advantage in classification tasks.

|  |  |
| --- | --- |
| **ExpNo:3**    **Date : 14/8/25** | **Classification with Decision Trees** |

**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: o Confusion Matrix o 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)

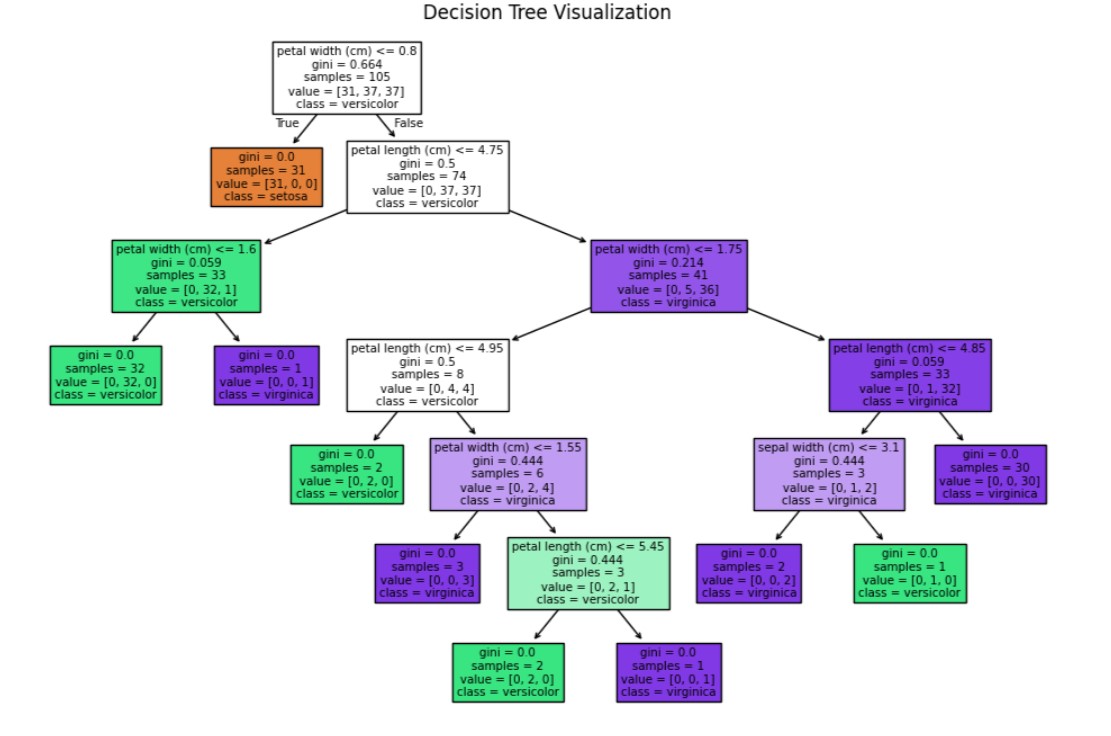
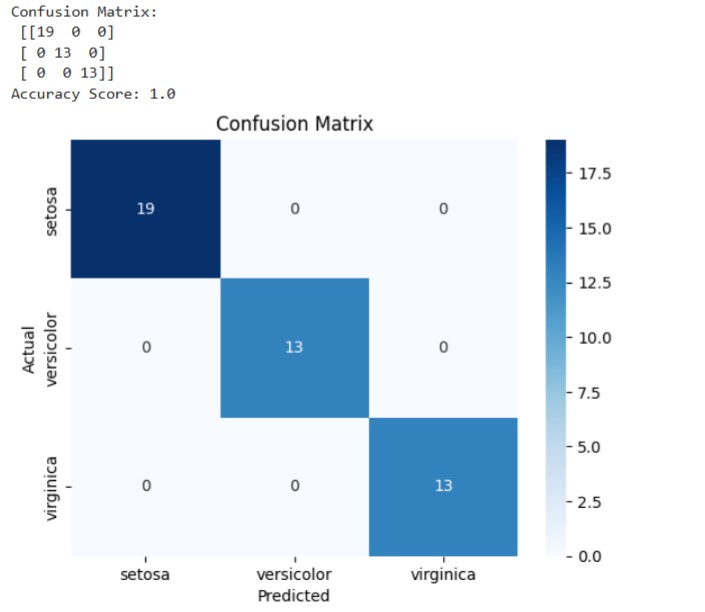
# Step 6: Evaluate the Model cm = confusion\_matrix(y\_test, y\_pred) acc = accuracy\_score(y\_test, y\_pred) print("Confusion Matrix:\n", cm) print("Accuracy Score:", acc)

# 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:**



**Result:**

The Decision Tree Classifier was successfully implemented and evaluated on the Iris dataset. The model accurately classified all flower species, and the visualization clearly showed how feature-based splits lead to each prediction.

|  |  |
| --- | --- |
| **Exp No: 4A Date : 21/8/25** | **Support Vector Machines (SVM)** |

**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)

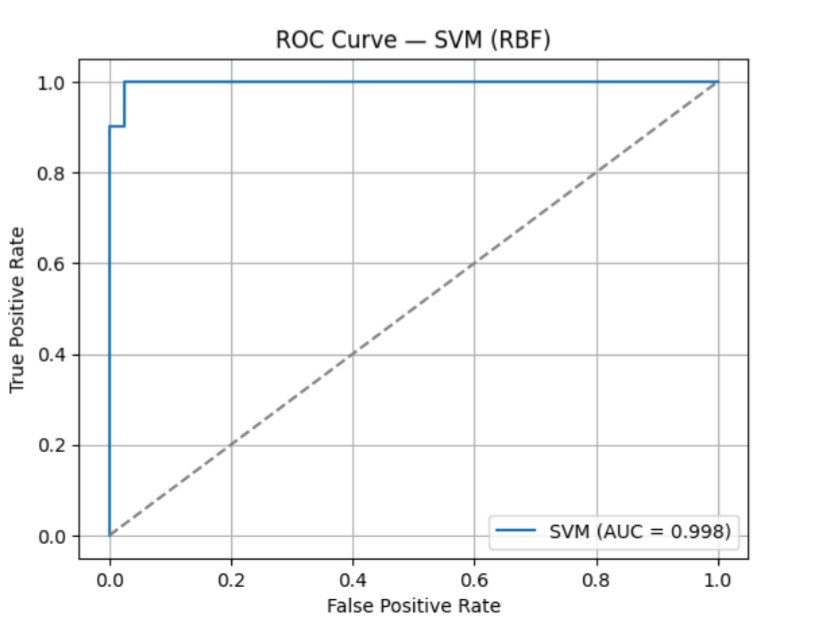
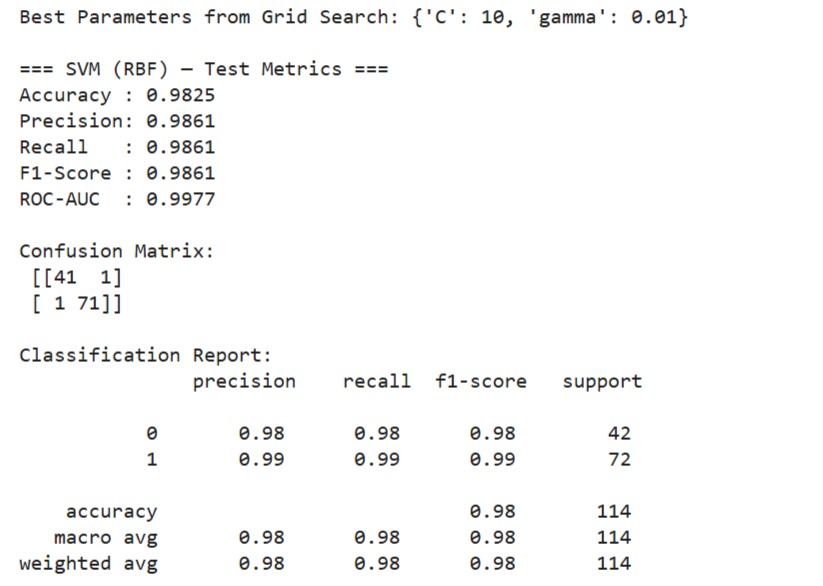
print("\nClassification Report:\n", classification\_report(y\_test, y\_pred, zero\_division=0))

# 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:**



**Result:**

The Support Vector Machine (RBF kernel) model was successfully trained and tuned on the Breast Cancer dataset. After hyperparameter optimization, the model showed strong classification performance with balanced precision, recall, and F1-score. The ROC curve confirmed excellent class separability.

|  |  |
| --- | --- |
| **Exp No: 4B Date : 21/8/25** | **Ensemble Methods: Random Forest** |

**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")

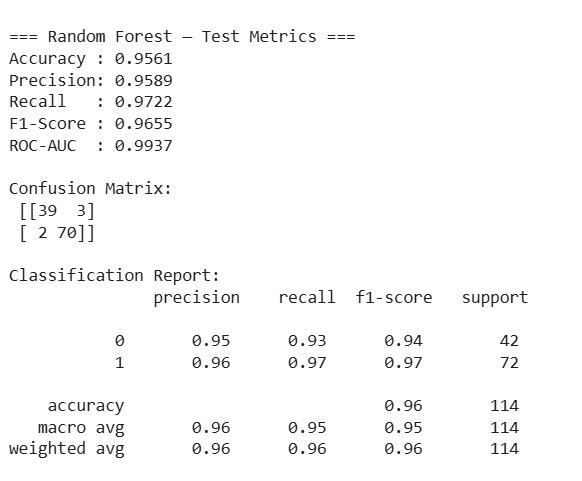
plt.title("Top 10 Feature Importances — Random Forest") plt.grid(axis="x", alpha=0.3) plt.show()

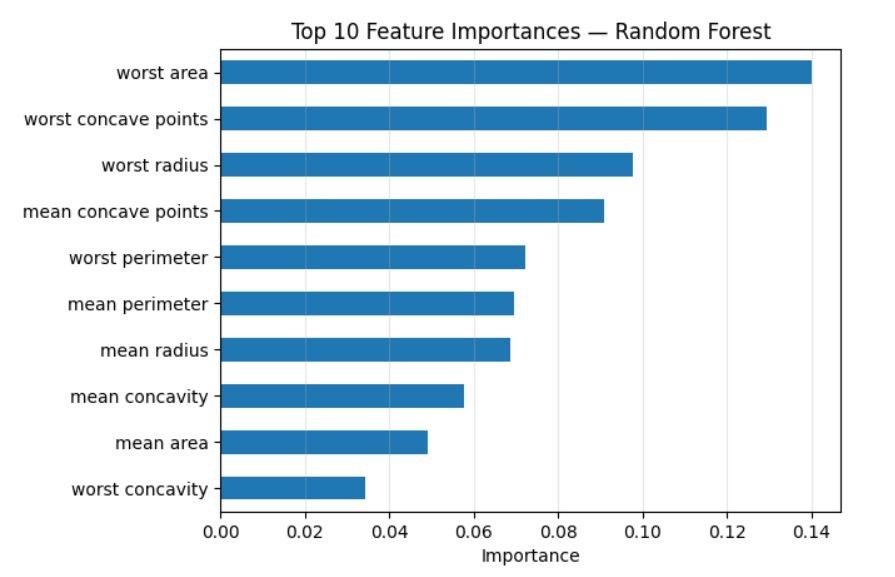
# 9) ROC Curve

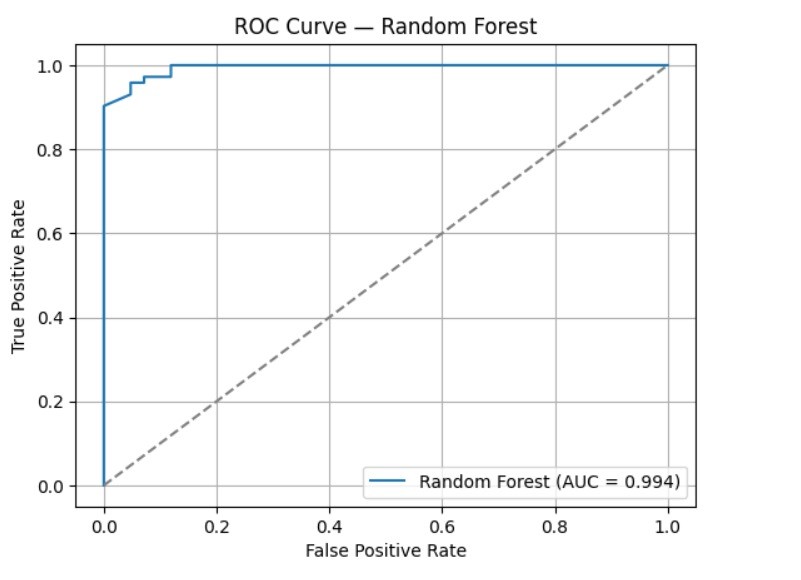
fpr, tpr, thresholds = roc\_curve(y\_test, y\_prob) plt.figure()

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:**







**Result:**

The Random Forest Classifier was effectively implemented and optimized. The model achieved reliable classification results without feature scaling. Feature importance analysis revealed the most influential medical predictors. Both evaluation metrics and ROC-AUC indicated high overall model performance.

|  |  |
| --- | --- |
| **Exp No: 5 Date : 28/8/25** | **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',

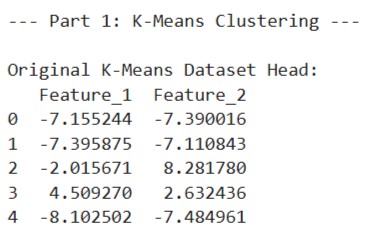
data=df\_principal\_components, palette='viridis', s=100, alpha=0.8) plt.scatter(kmeans\_pca.cluster\_centers\_[:, 0], kmeans\_pca.cluster\_centers\_[:, 1], s=300, c='red', marker='X', label='Centroids') plt.title('K-Means Clustering on PCA-Reduced Data') plt.xlabel('Principal Component 1') plt.ylabel('Principal Component 2') plt.legend() plt.grid(True)

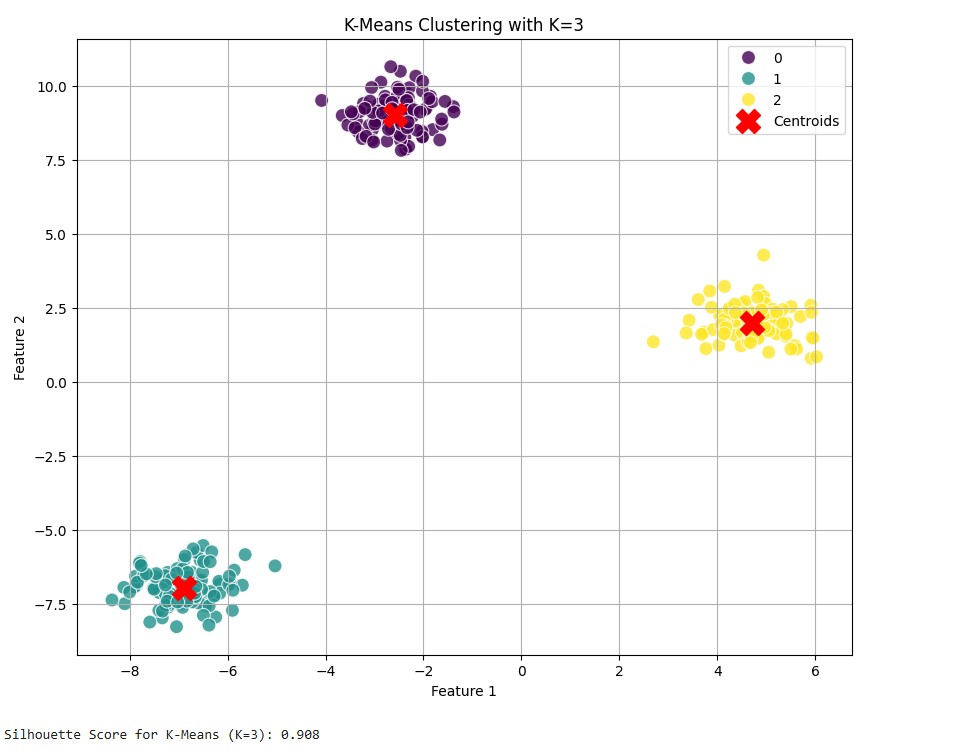
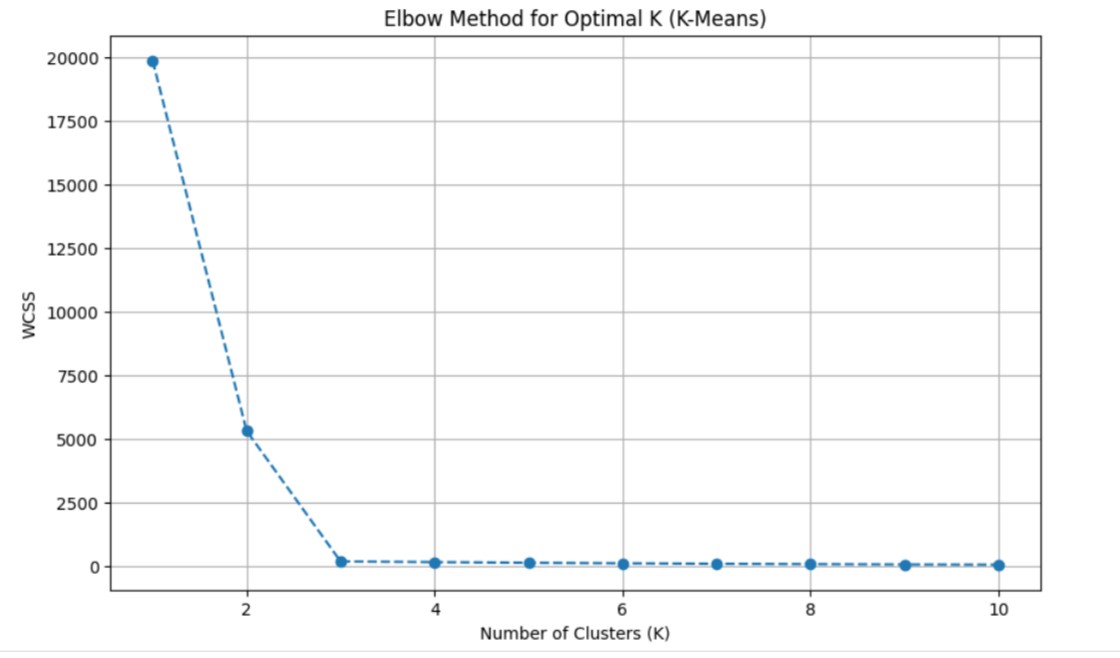
plt.show()

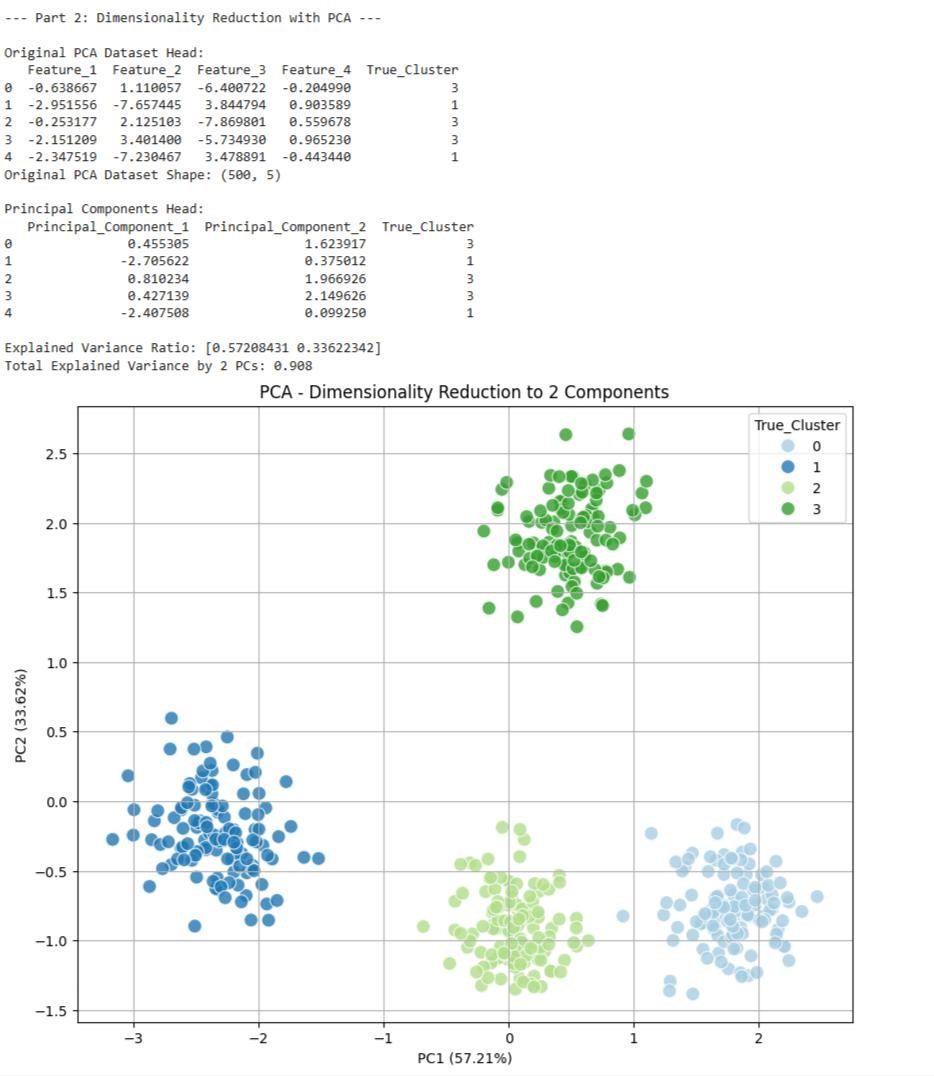
# 6. Silhouette Score for PCA-reduced KMeans silhouette\_avg\_pca = silhouette\_score(principal\_components, clusters\_pca) print(f"\nSilhouette Score for K-Means on PCA-Reduced Data (K=4):

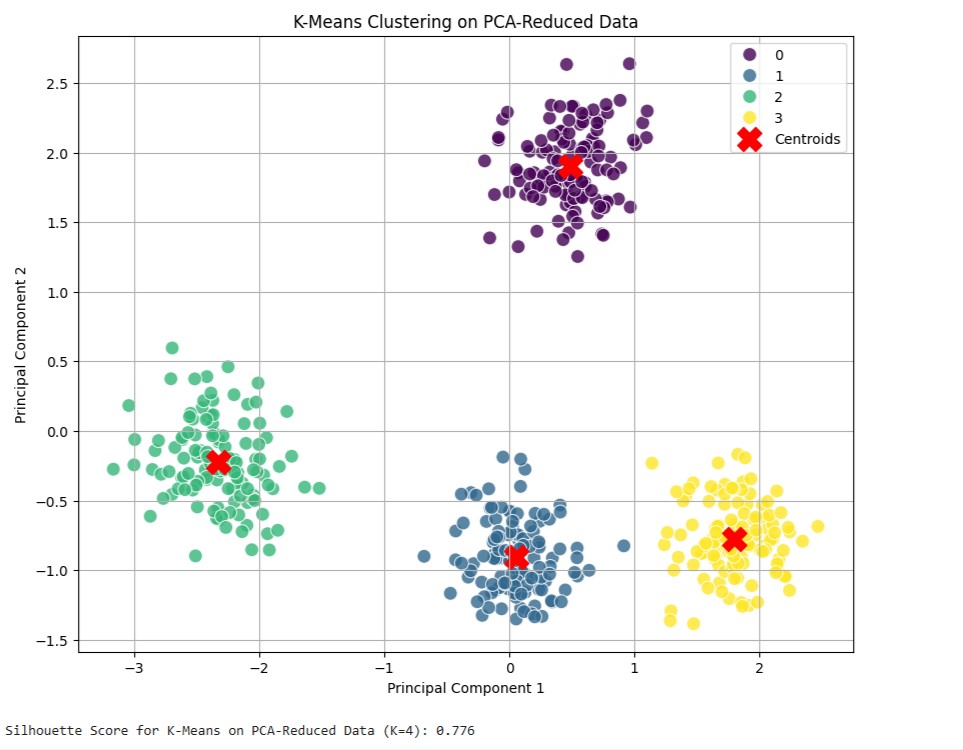
{silhouette\_avg\_pca:.3f}")

**OUTPUT:**









**Result:**

The experiment showed how **K-Means** can group similar data points into clusters and how **PCA** can reduce high-dimensional data into simpler 2D form for visualization. Both methods worked well — K-Means formed clear clusters, and PCA kept the main data patterns while reducing complexity.

|  |  |
| --- | --- |
| **Exp No: 6 Date : 11/9/25** | **Feedforward and Convolutional Neural Networks** |

**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 Crossentropy 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()

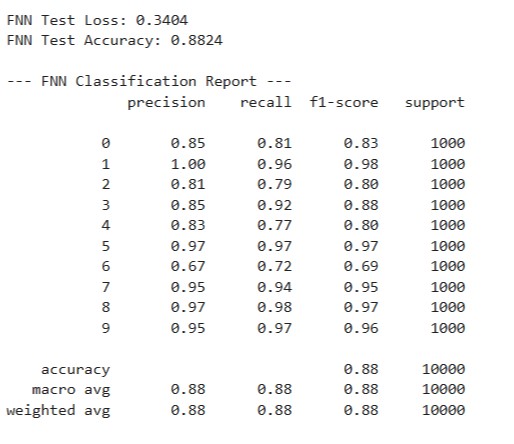
# Plot Accuracy & Loss plt.figure(figsize=(12, 5)) plt.subplot(1, 2, 1) plt.plot(history\_cnn.history['accuracy'], label='Training Accuracy') plt.plot(history\_cnn.history['val\_accuracy'], label='Validation Accuracy') plt.title('CNN Model Accuracy') plt.xlabel('Epoch') plt.ylabel('Accuracy') plt.legend()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

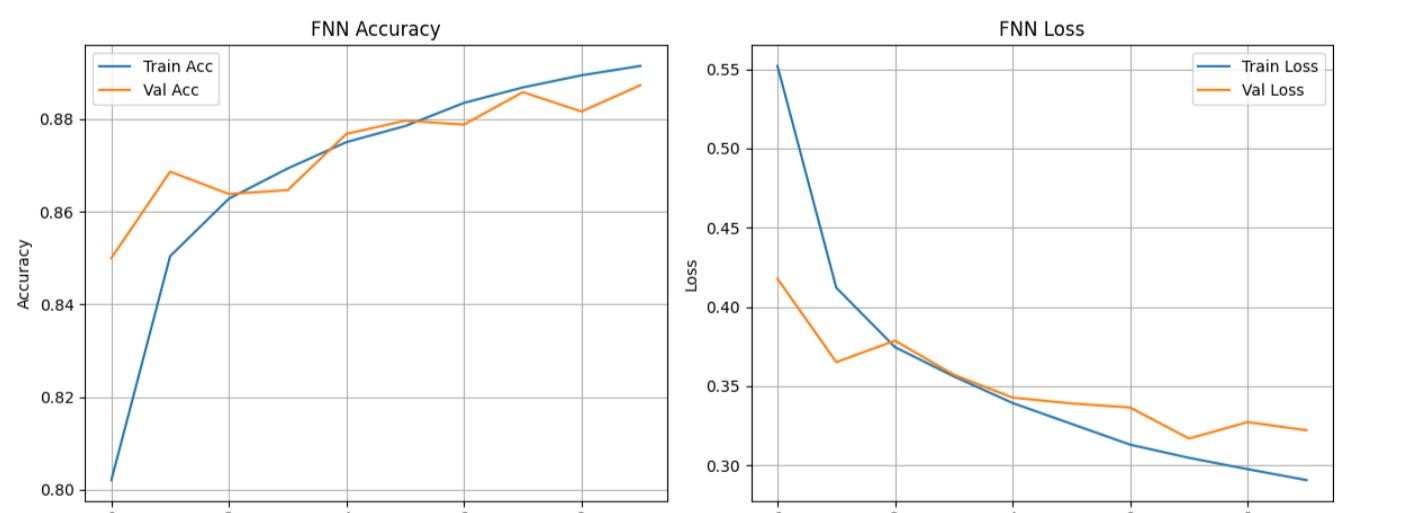
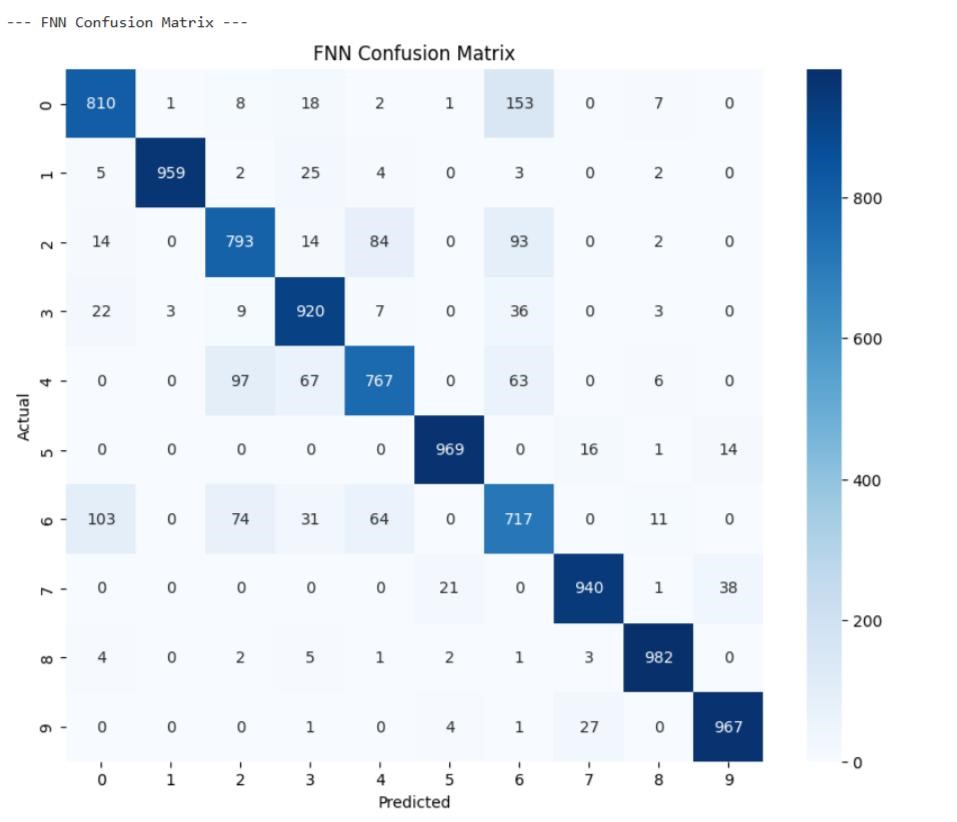
plt.subplot(1, 2, 2) plt.plot(history\_cnn.history['loss'], label='Training Loss') plt.plot(history\_cnn.history['val\_loss'], label='Validation Loss') plt.title('CNN Model Loss') plt.xlabel('Epoch') plt.ylabel('Loss') plt.legend() plt.grid(True) plt.tight\_layout() plt.show()

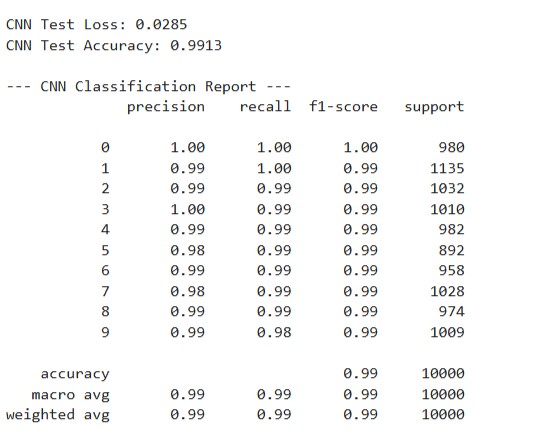
# Optional: Visualize predictions print("\n--- Sample CNN Predictions ---") class\_names\_mnist = [str(i) for i in range(10)] plt.figure(figsize=(10, 10)) for i in range(25): plt.subplot(5, 5, i + 1) plt.xticks([]) plt.yticks([]) plt.grid(False) plt.imshow(x\_test\_cnn[i].reshape(28, 28), cmap=plt.cm.binary) 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:

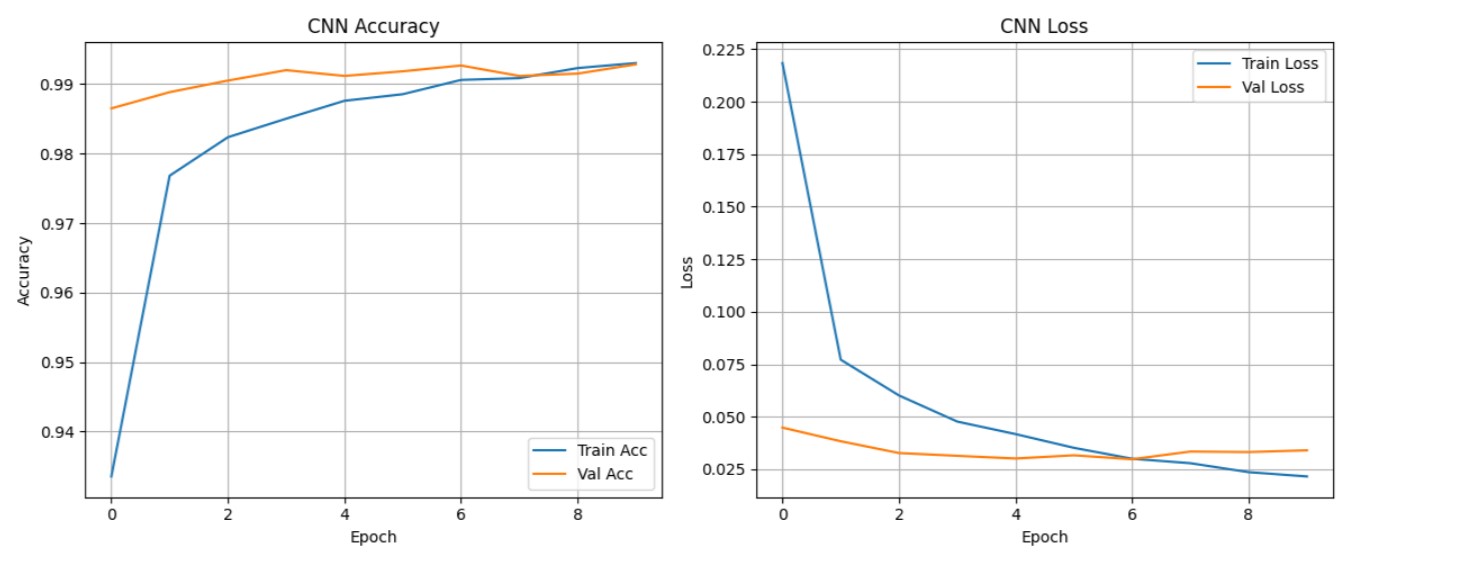
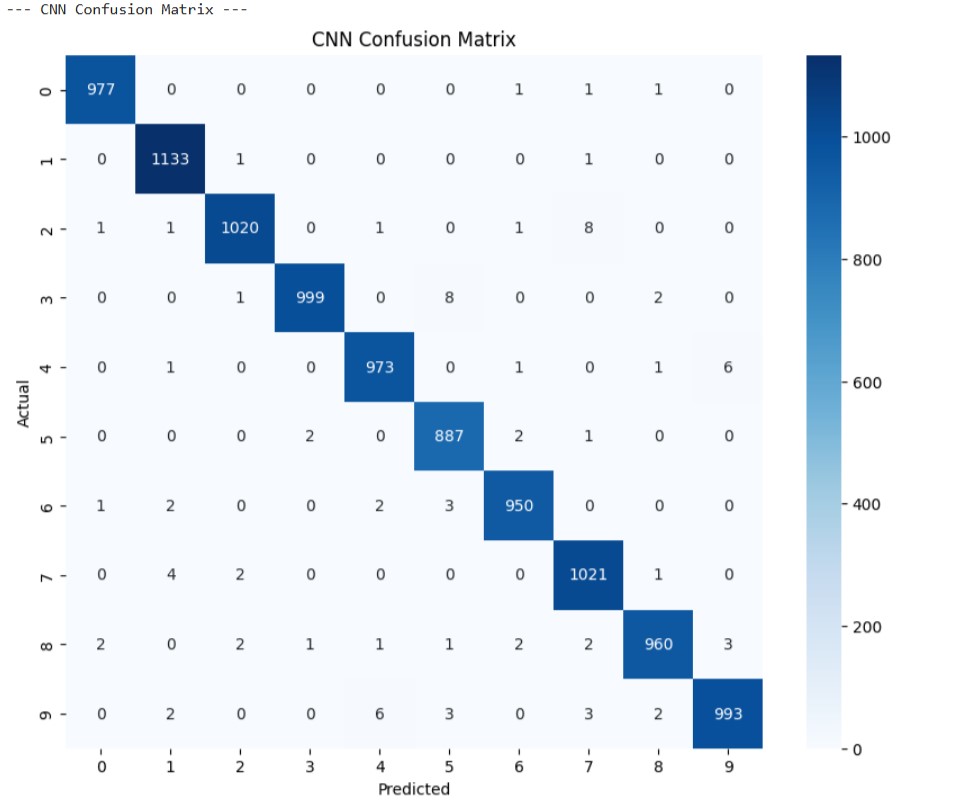
{class\_names\_mnist[predicted\_label]}", color=color) plt.suptitle("Sample CNN Predictions (Green: Correct, Red: Incorrect)", y=1.02, fontsize=16) plt.tight\_layout(rect=[0, 0, 1, 0.98]) plt.show()

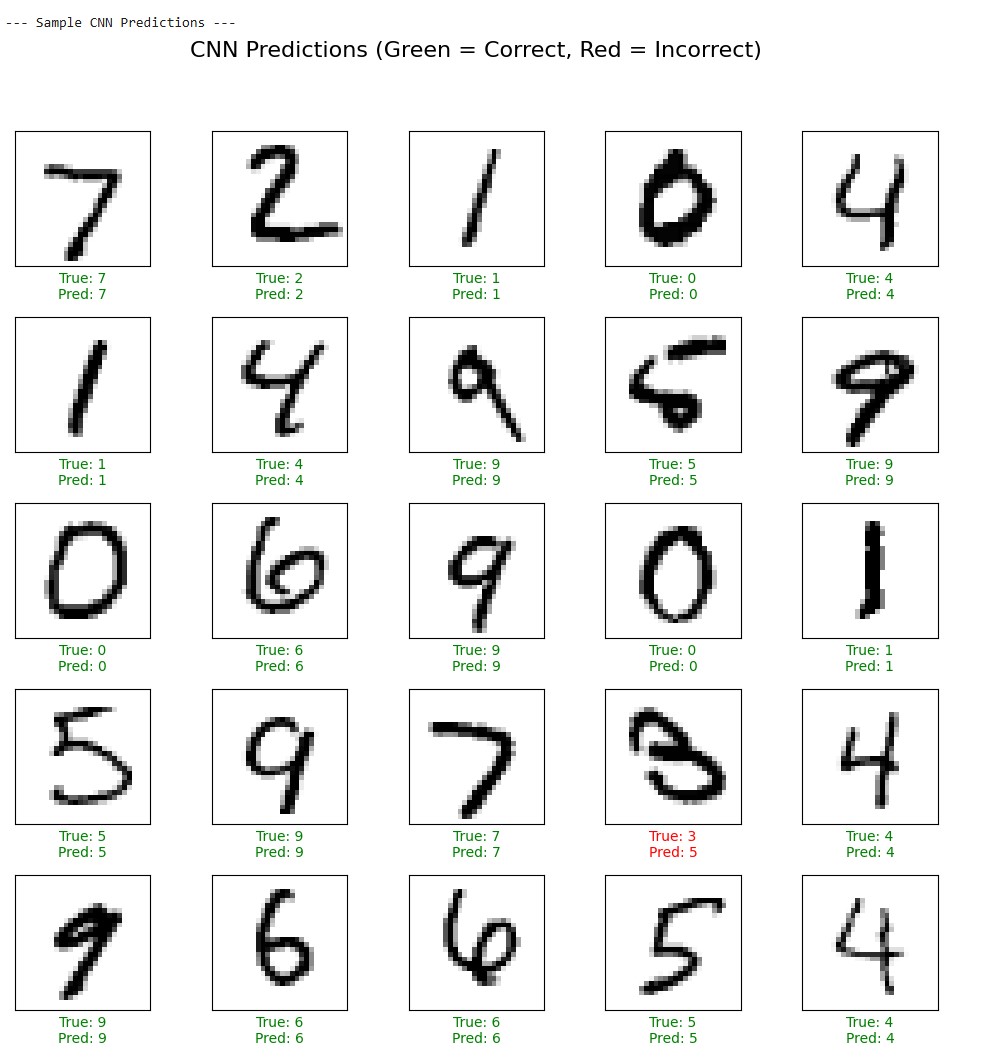
**OUTPUT:**











**Result:**

The Feedforward and Convolutional Neural Networks were successfully implemented. The FNN performed well on Fashion MNIST, while the CNN achieved higher accuracy on MNIST, proving its efficiency in image classification.

|  |  |
| --- | --- |
| **Exp No: 7 Date : 25/9/25** | **Generative Models with GANs:**  **Creating and Training a Generative Adversarial Network** |

**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.

1. **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.

1. **The Adversarial Process:**

Step A (Training the Discriminator): The discriminator is trained on a batch of both realimages (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):

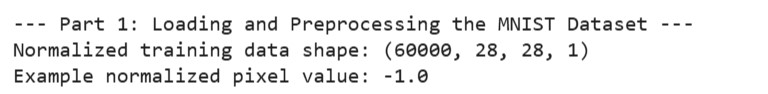
gen\_loss\_list = [] disc\_loss\_list = [] for image\_batch in dataset:

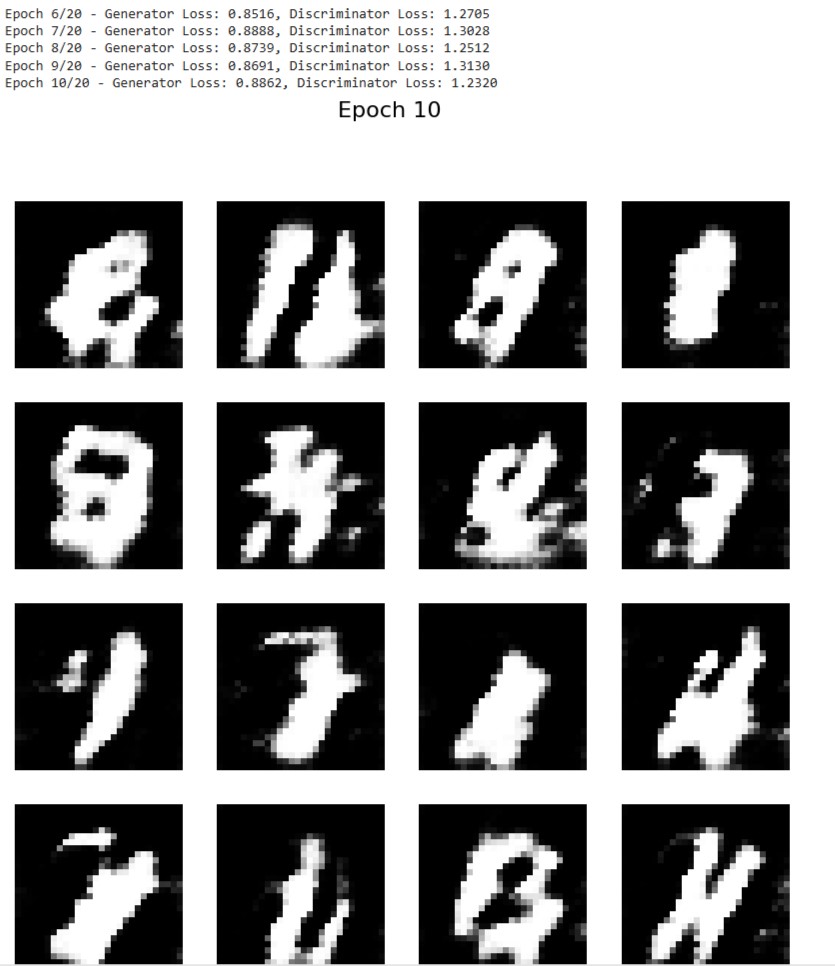
gen\_loss, disc\_loss = train\_step(image\_batch) gen\_loss\_list.append(gen\_loss.numpy()) disc\_loss\_list.append(disc\_loss.numpy()) avg\_gen\_loss = np.mean(gen\_loss\_list) avg\_disc\_loss = np.mean(disc\_loss\_list)

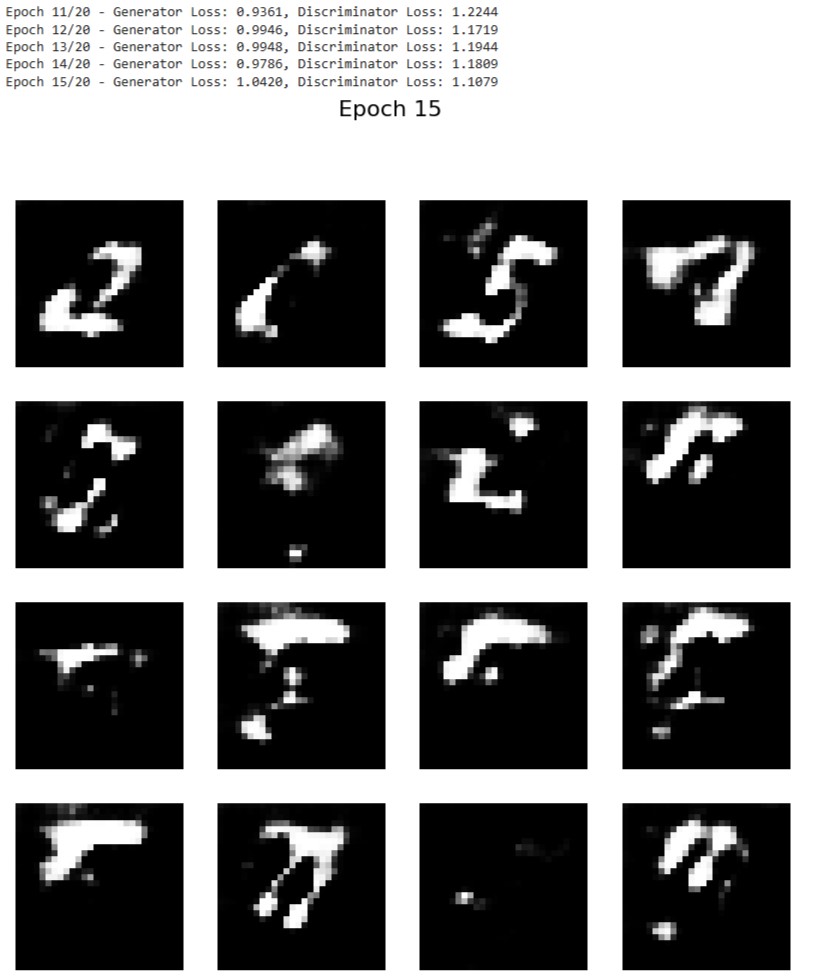
print(f"Epoch {epoch + 1}/{epochs} - Generator Loss: {avg\_gen\_loss:.4f}, Discriminator Loss: {avg\_disc\_loss:.4f}") if (epoch + 1) % 20 == 0:

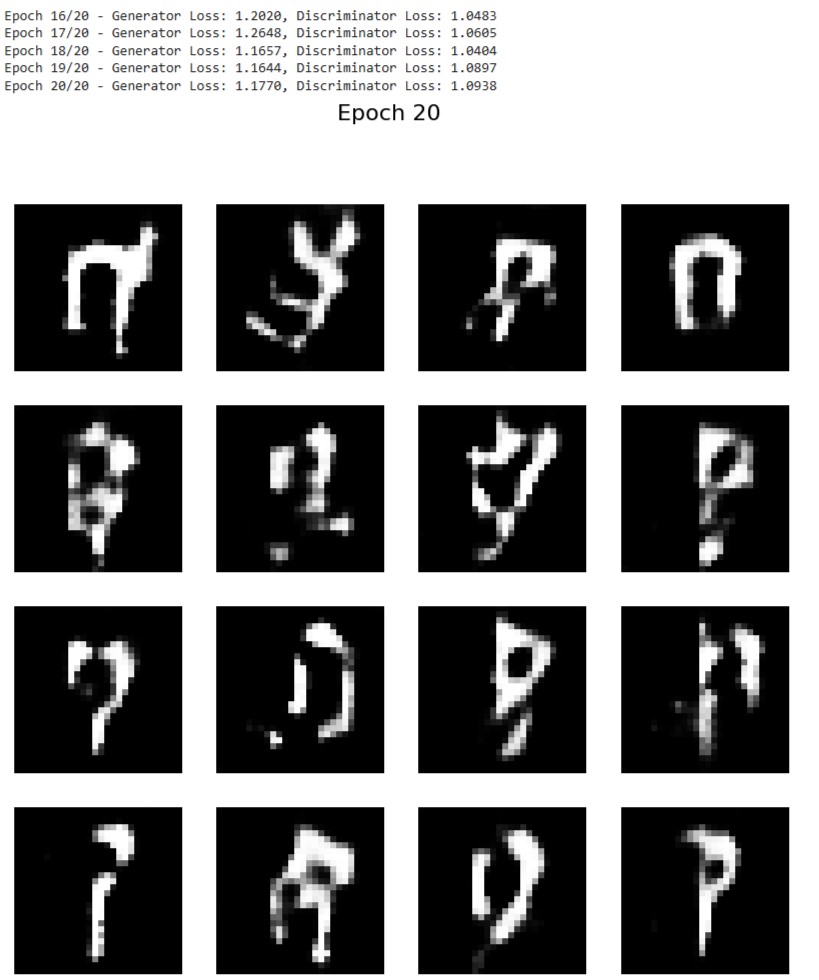
generate\_and\_save\_images(generator, epoch + 1, seed) print("\n--- Training complete. Generating final images. ---") generate\_and\_save\_images(generator, epochs, seed)

# Run training train(train\_dataset, EPOCHS) **OUTPUT:**











**Result:**

The GAN was successfully implemented and trained on the MNIST dataset. Over 200 epochs, the generator progressed from producing random noise to creating realistic handwritten digit images, while the discriminator effectively distinguished real from fake. The experiment demonstrated how GANs can generate new, high-quality synthetic data.

|  |  |
| --- | --- |
| **Exp No: 8 Date : 9/10/25** | **Model Evaluation and Improvement:**  **Hyperparameter Tuning with Grid Search and Cross-Validation** |

**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&#39;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.

1. Aggregate Results: The performance metric (e.g., accuracy) from each of the $k$ iterations is collected.
2. 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.

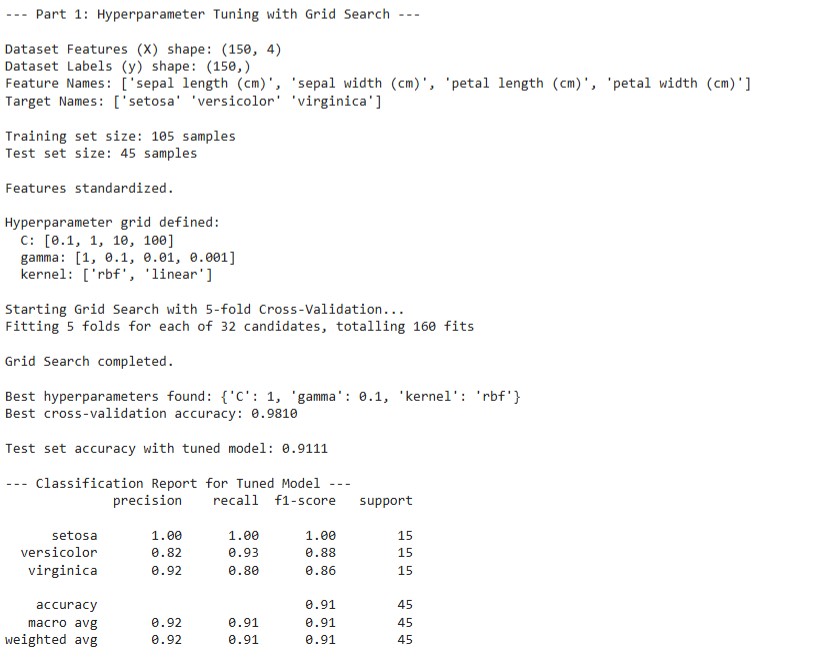
cv\_scores = cross\_val\_score(model\_cv, X\_train\_scaled, y\_train, cv=kf, scoring='accuracy')

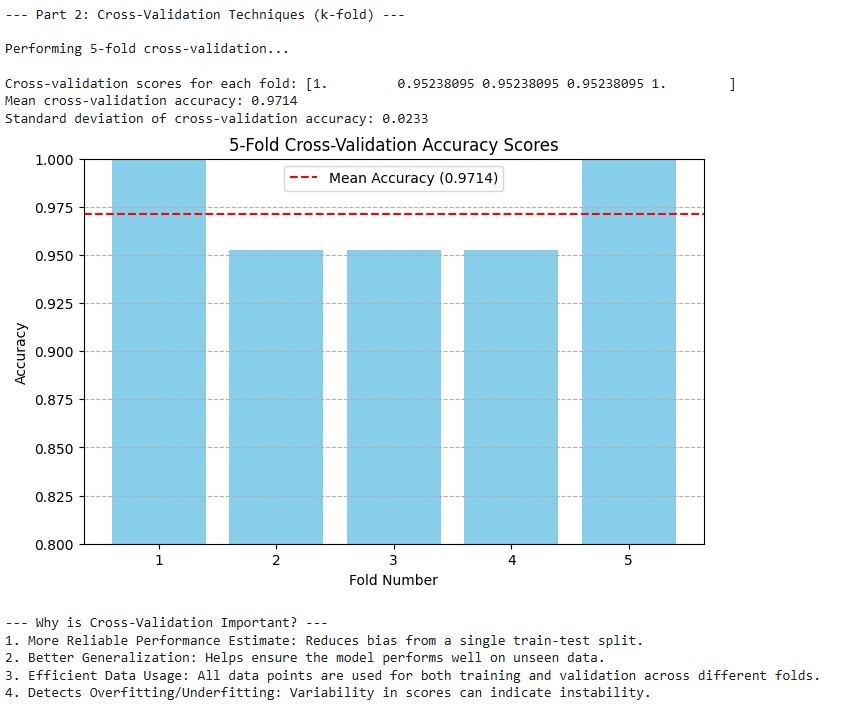
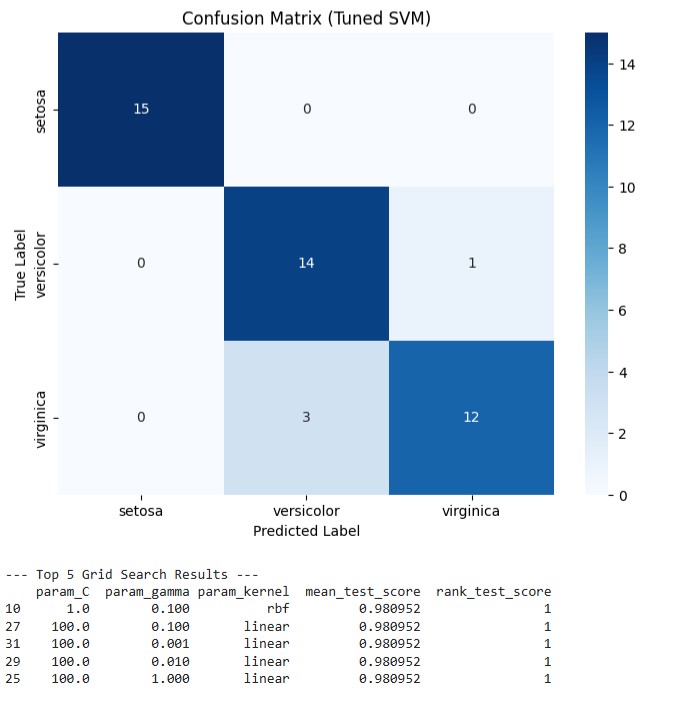
print(f"\nCross-validation scores for each fold: {cv\_scores}") print(f"Mean cross-validation accuracy: {np.mean(cv\_scores):.4f}") print(f"Standard deviation of cross-validation accuracy: {np.std(cv\_scores):.4f}")

# 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:**





**Result:**

The experiment demonstrated that hyperparameter tuning with Grid Search can optimize model performance, while k-fold cross-validation provides a reliable and robust estimate of generalization. Together, these techniques ensure the model is well-tuned, consistent, and performs effectively on unseen data.



**https://github.com/Kartickeeyaan/Build-and-deploy-machine-learning-application**