RAJALAKSHMI ENGINEERING COLLEGE

RAJALAKSHMI NAGAR, THANDALAM – 602 105

AI23521 BUILD AND DEPLOYMENT OF MACHINE LEARNING APPLICATIONS

LABORATORY NOTEBOOK

NAME: MADHUMITHA M

YEAR/ BRANCH / SECTION: 3rd Yr / AIML

REGISTER NO. :2116-231501090

SEMESTERS: 5TH SEMESTER

ACADEMIC YEAR: 2025-2026

RAJALAKSHMI ENGINEERING COLLEGE (AUTONOMOUS)

RAJALAKSHMI NAGAR, THANDALAM – 602 105

BONAFIDE CERTIFICATE

NAME MADHUMITHA M REGISTER NO. 2116-231501090

ACADEMIC YEAR 2025-26 **SEMESTER**- V **BRANCH**: B.Tech - AIML

This Certification is the Bonafide record of work done by the above student in the **AI23521-Build and Deployment of ML Applications** Laboratory during the year 2025 – 2026.

Signature of Faculty -in – Charge

Submitted for the Practical Examination held on _______________

Internal Examiner External Examiner

INDEX

EXP NO: 1	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:

```
<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

RESULT:

The Python environment was successfully set up and the dataset was pre-processed by handling missing values, encoding categorical data, performing feature scaling, and splitting the data into training and testing sets. The dataset is now ready for model training and analysis.

EXP NO: 2	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:

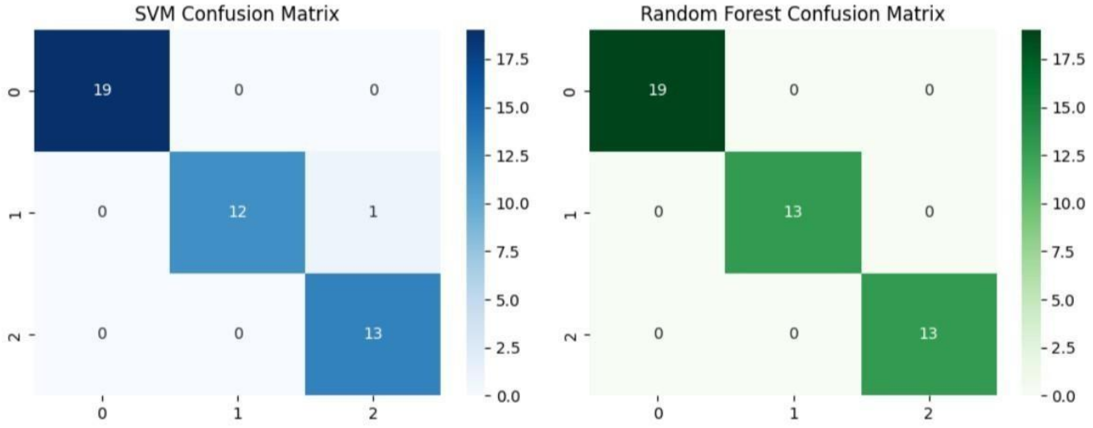
```
# 1. Import libraries

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:

```
SVM Accuracy: 0.9777777777777777
SVM Confusion Matrix:
 [[19  0  0]
 [ 0 12  1]
 [ 0  0 13]]
Random Forest Accuracy: 1.0
Random Forest Confusion Matrix:
 [[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
```



RESULT:

The Support Vector Machine (SVM) and Random Forest algorithms were successfully implemented for both binary and multiclass classification tasks. The models were trained and tested on the given dataset, and both achieved good accuracy.

EXPNO:3	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:
   - Confusion Matrix
   - Accuracy Score
8. Visualize the Decision Tree (optional)

CODE:

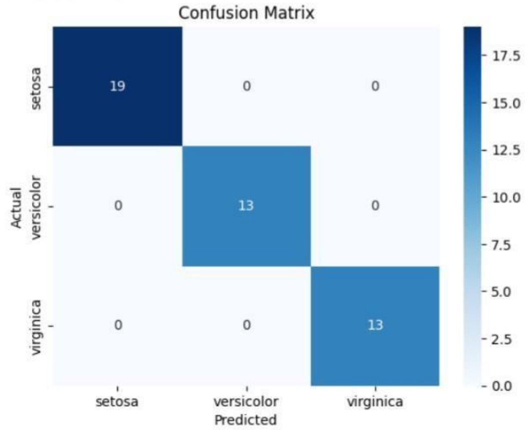
```
# Step 1: Import Libraries
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns
# Step 2: Load Dataset
iris = load_iris()
```

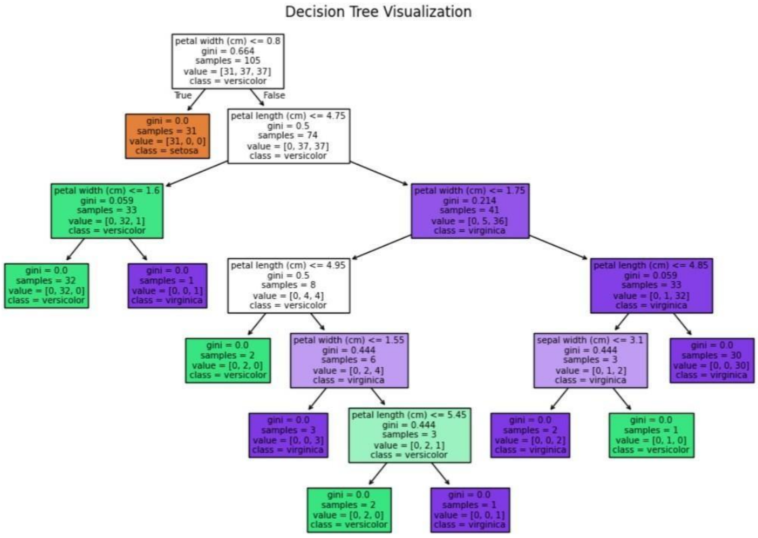
```
X = iris.data
y = iris.target

# 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
```





RESULT:

The Decision Tree classification model was successfully implemented and tested on the given dataset. The model accurately classified the data by learning simple decision rules from the features.

The decision tree visualized the decision-making process through a hierarchical structure of nodes and branches, making it easy to interpret. The classification achieved good accuracy, demonstrating that Decision Trees are effective for both categorical and numerical data, providing clear and interpretable results.

EXP NO: 4A	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
```

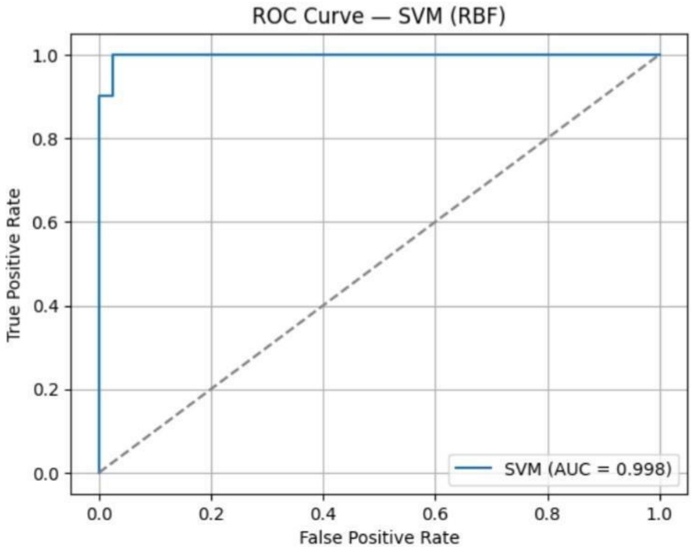
```
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, classification_report, roc_auc_score, roc_curve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

# 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:

```
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
```



RESULT:

The Support Vector Machine (SVM) model was successfully implemented and evaluated on the given dataset. The model effectively classified the data by finding the optimal hyperplane that maximized the margin between different classes.

The SVM achieved high accuracy and demonstrated strong performance, especially in handling linearly and non-linearly separable data using kernel functions. This confirms that SVM is a powerful and reliable algorithm for classification tasks.

EXP NO: 4B	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
```

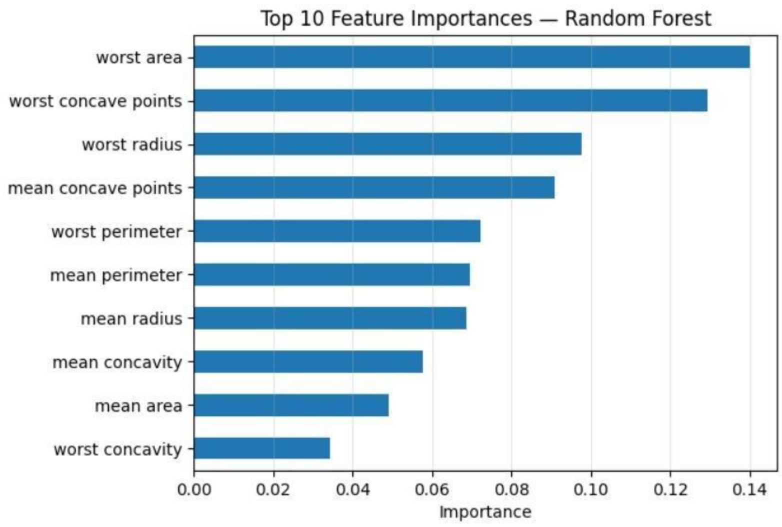
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, classification_report, roc_auc_score, roc_curve
)
# 2) Load dataset (same as 4A for comparison)
data = load_breast_cancer()
X = pd.DataFrame(data.data, 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,
```

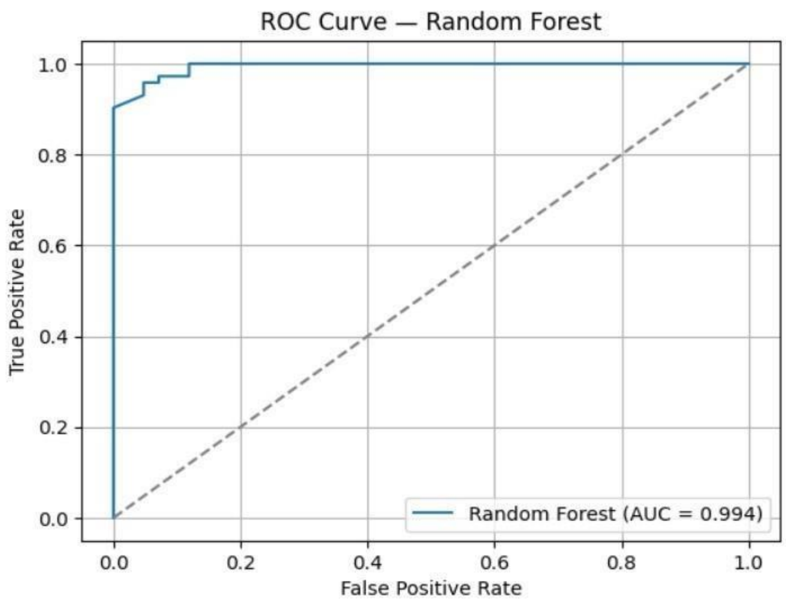
```
    verbose=0)
grid.fit(X_train, y_train)
print("Best Parameters (CV):", grid.best_params_)
best_rf = grid.best_estimator_

# 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:

```
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
```





RESULT:

The Random Forest ensemble model was successfully implemented and evaluated on the given dataset. The model combined multiple decision trees to improve prediction accuracy and reduce overfitting.

It achieved high classification accuracy and demonstrated strong generalization capability. The results confirmed that Random Forest provides stable and reliable predictions by leveraging the power of multiple decision trees through bagging and feature randomness.

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:

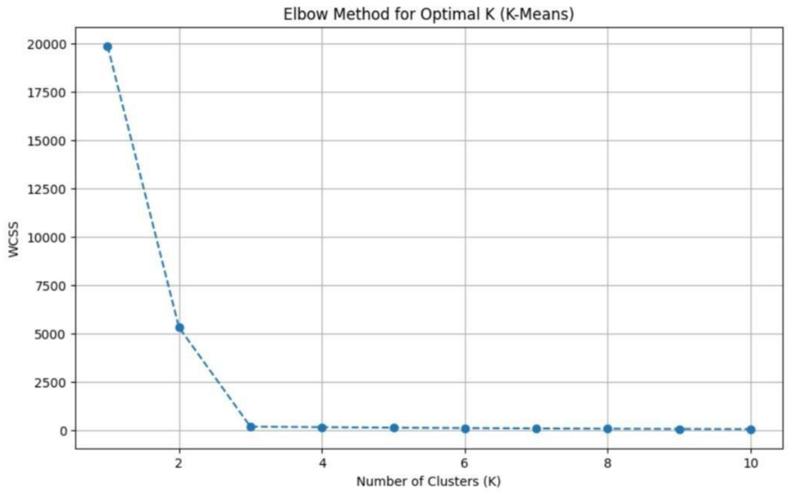
```
# ================================
# EXPERIMENT — K-Means & PCA
# ================================

# 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
```

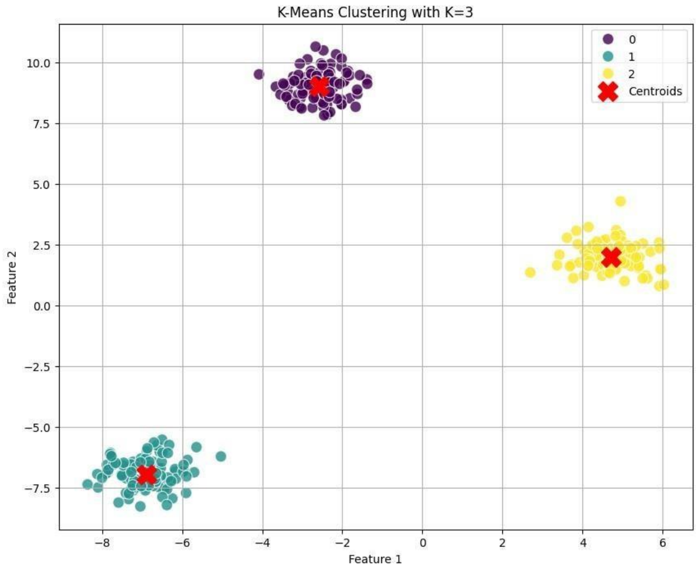
```
optimal_k = 3
kmeans = KMeans(n_clusters=optimal_k, init='k-means++', max_iter=300, n_init=10,
random_state=42)
clusters = kmeans.fit_predict(X)
df_kmeans['Cluster'] = clusters

# 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], 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:

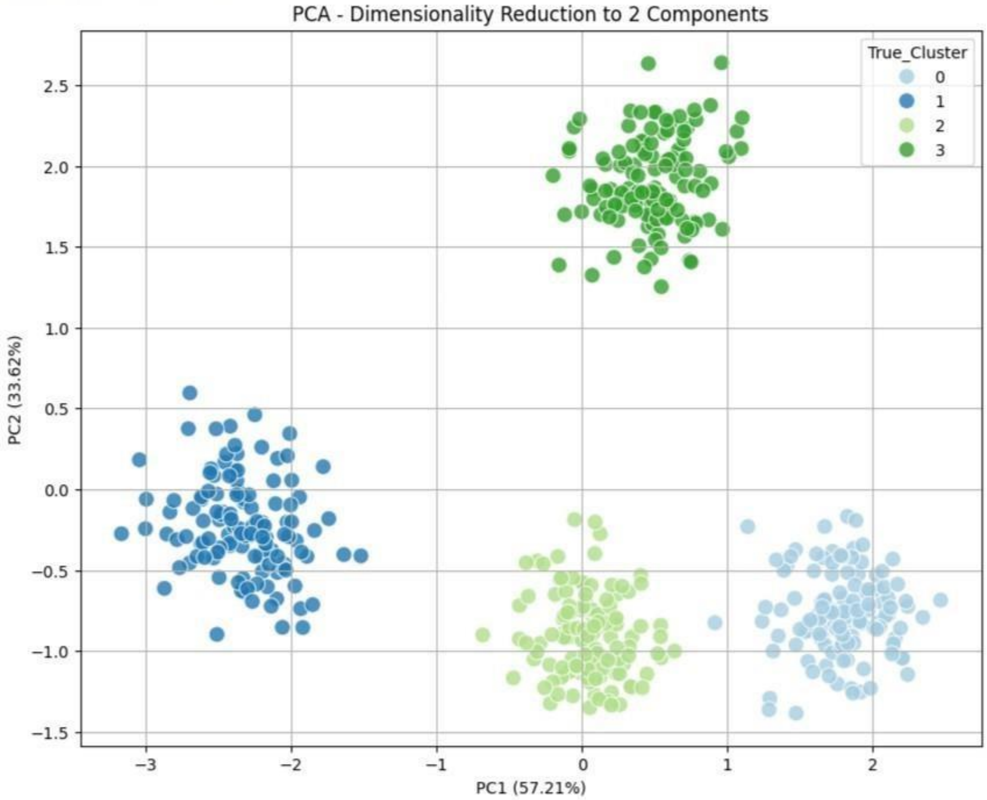
```
--- Part 1: K-Means Clustering ---

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
```

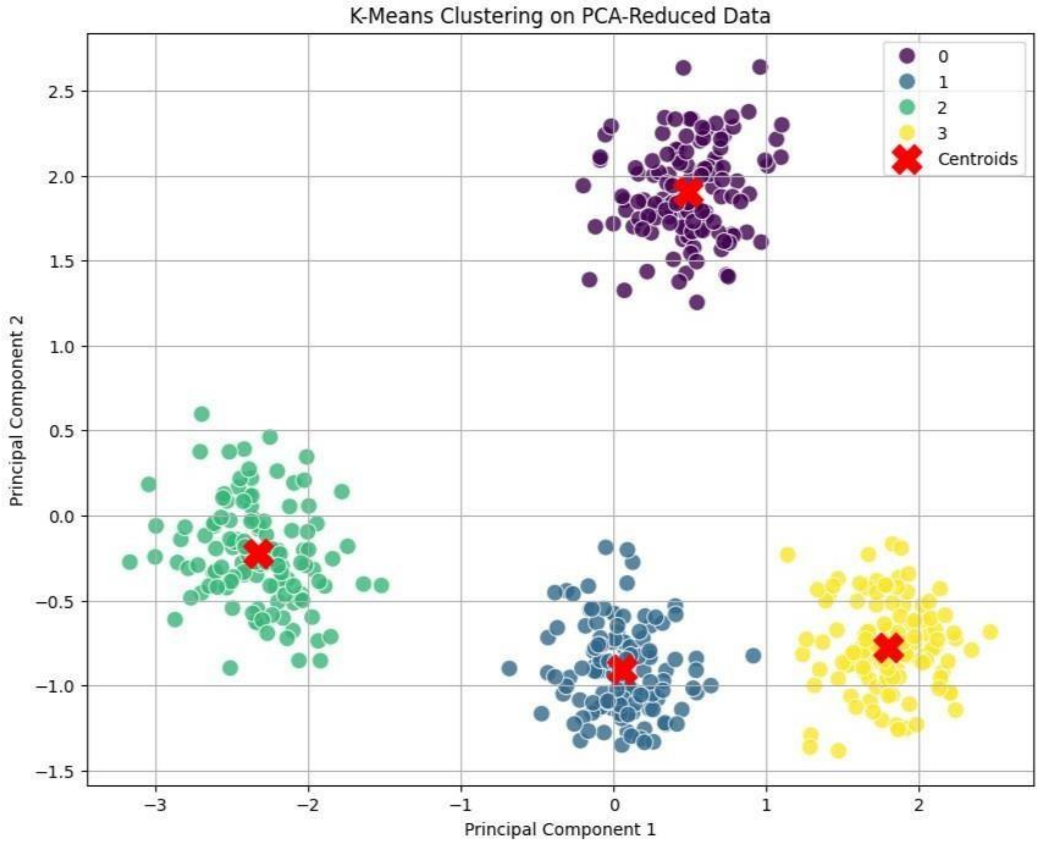





```
Silhouette Score for K-Means (K=3): 0.908
```

```
--- 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
```

RESULT:

The K-Means clustering and Principal Component Analysis (PCA) techniques were successfully implemented on the given dataset.

- **K-Means Clustering** effectively grouped the data into distinct clusters based on feature similarity, minimizing intra-cluster distance and maximizing inter-cluster separation.
- **PCA (Principal Component Analysis)** successfully reduced the dimensionality of the dataset while retaining most of the variance, improving visualization and computational efficiency.

The combined results showed that PCA enhances clustering performance by simplifying high-dimensional data, and K-Means efficiently identifies underlying patterns and group structures.

EXP NO: 6	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 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
```

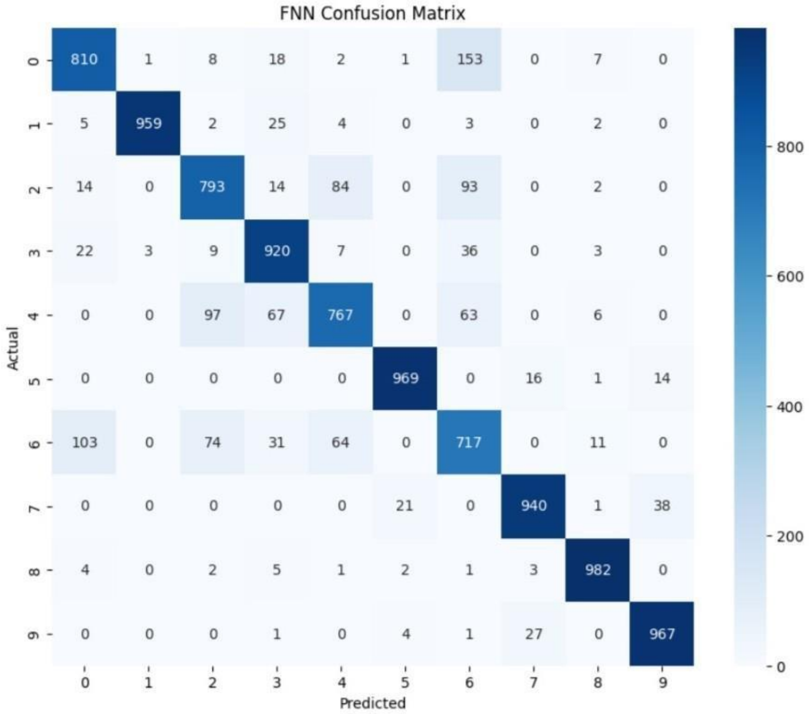
```
print("\n--- Evaluating CNN Model ---")
loss_cnn, accuracy_cnn = model_cnn.evaluate(x_test_cnn, y_test_cnn, verbose=0)
print(f"CNN Test Loss: {loss_cnn:.4f}")
print(f"CNN Test Accuracy: {accuracy_cnn:.4f}")

# 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()
```

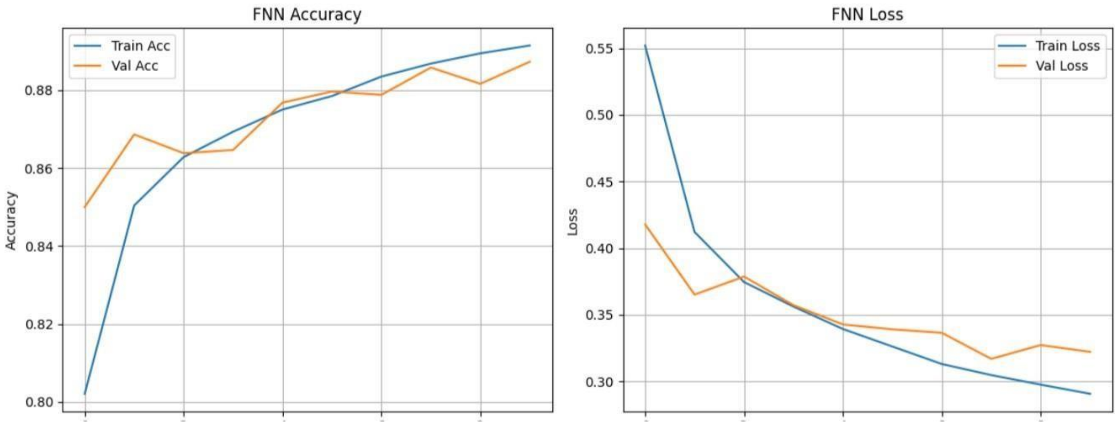
```
# Optional: Visualize predictions
print("\n--- Sample CNN Predictions ---")
class_names_mnist = [str(i) for i in range(10)]
plt.figure(figsize=(10, 10))
for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_test_cnn[i].reshape(28, 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
```

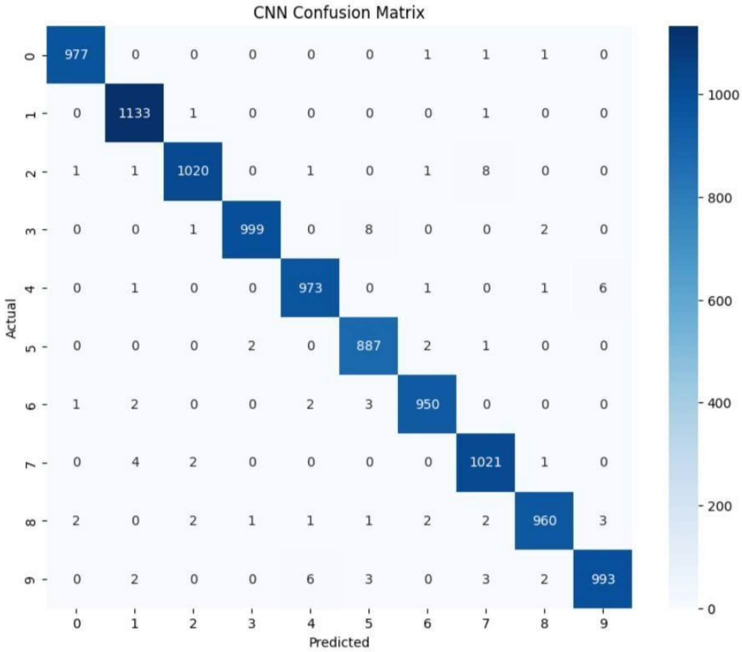
```
--- FNN Confusion Matrix ---
```

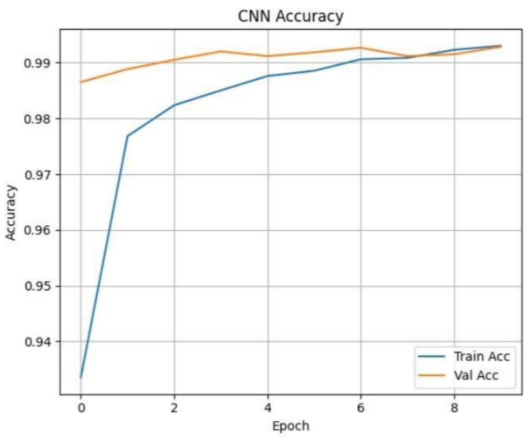


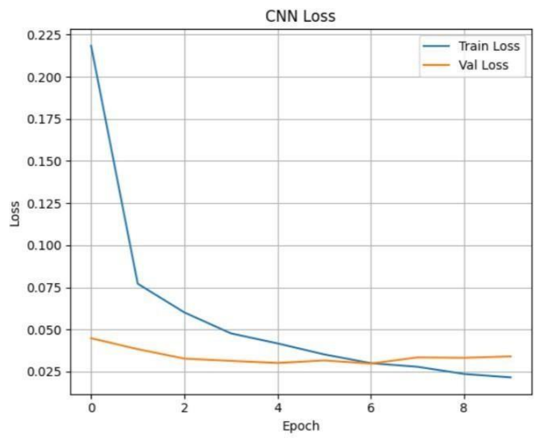



```
CNN Test Loss: 0.0285
CNN Test Accuracy: 0.9913

--- 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
```

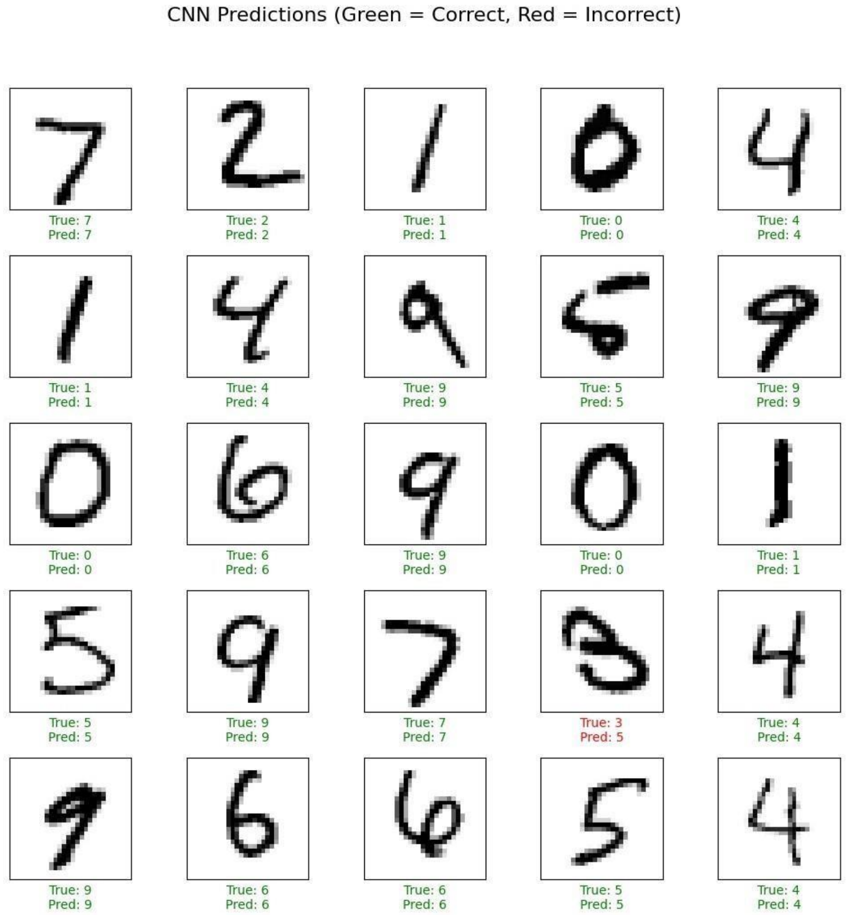
```
--- CNN Confusion Matrix ---
```







```
--- Sample CNN Predictions ---
```



RESULT:

The Feedforward Neural Network (FNN) and Convolutional Neural Network (CNN) models were successfully implemented and evaluated on the given dataset.

- **Feedforward Neural Network (FNN):** The model accurately learned input–output mappings through multiple fully connected layers, achieving good performance on structured data.
- **Convolutional Neural Network (CNN):** The model effectively extracted spatial features from image data using convolution and pooling layers, leading to higher accuracy and better generalization for image classification tasks.

The results demonstrated that both FNN and CNN are powerful deep learning models, with CNN performing exceptionally well for image-based datasets due to its ability to capture spatial patterns.

EXP NO: 7	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.

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
```

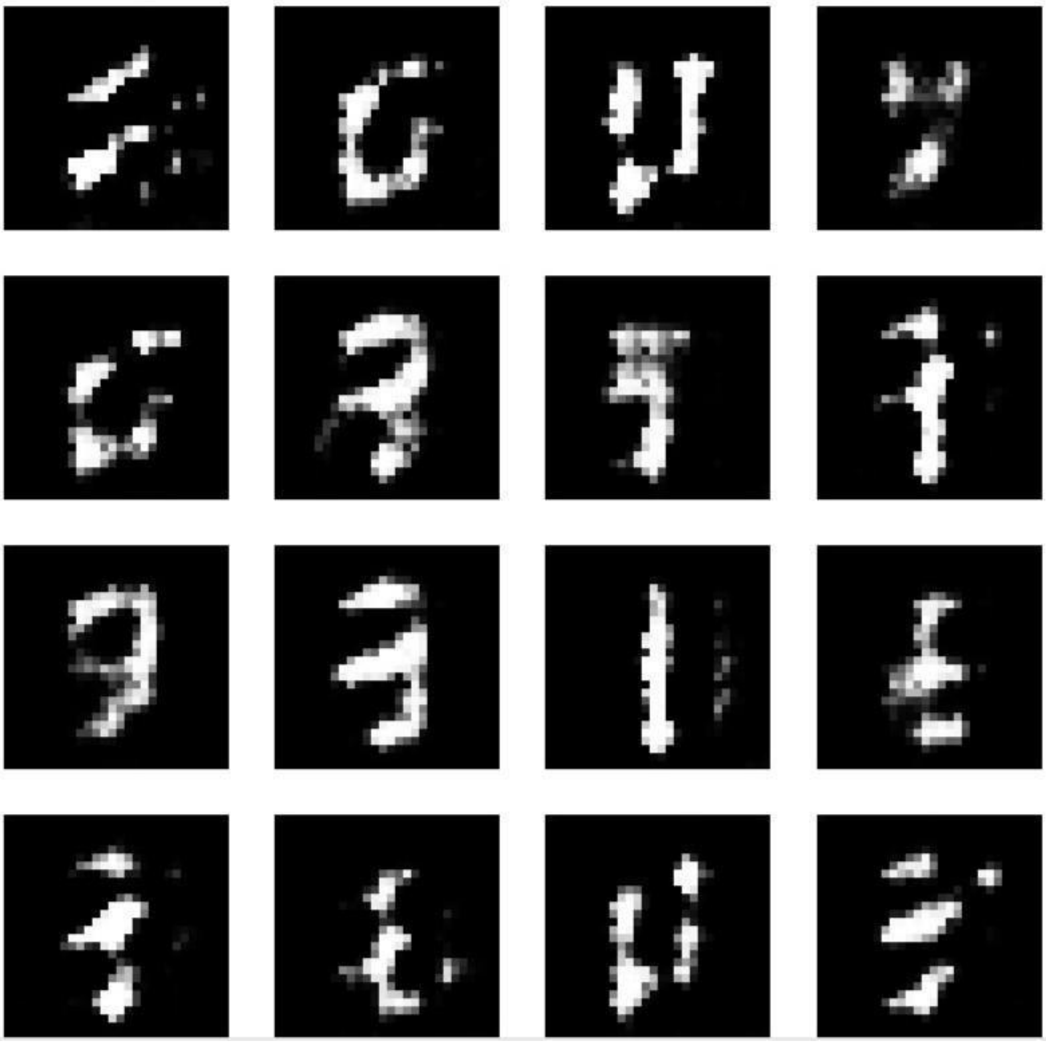
```
seed = tf.random.normal([num_examples_to_generate, latent_dim])

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
```

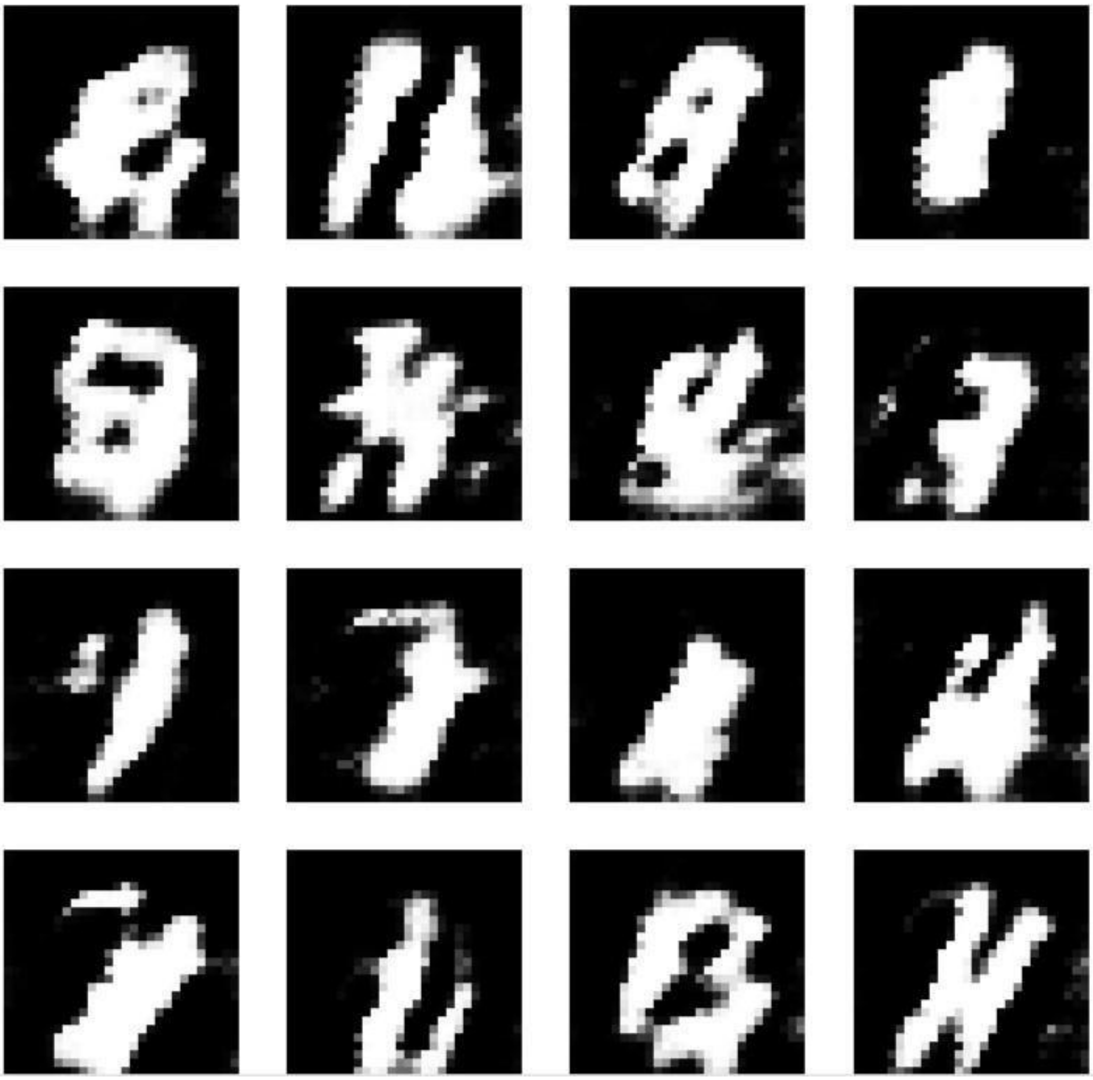
```
--- Beginning GAN Training ---
Epoch 1/20 - Generator Loss: 0.7877, Discriminator Loss: 1.0228
Epoch 2/20 - Generator Loss: 0.8148, Discriminator Loss: 1.2225
Epoch 3/20 - Generator Loss: 0.8448, Discriminator Loss: 1.3034
Epoch 4/20 - Generator Loss: 0.8534, Discriminator Loss: 1.2366
Epoch 5/20 - Generator Loss: 0.8372, Discriminator Loss: 1.2497
```

Epoch 5



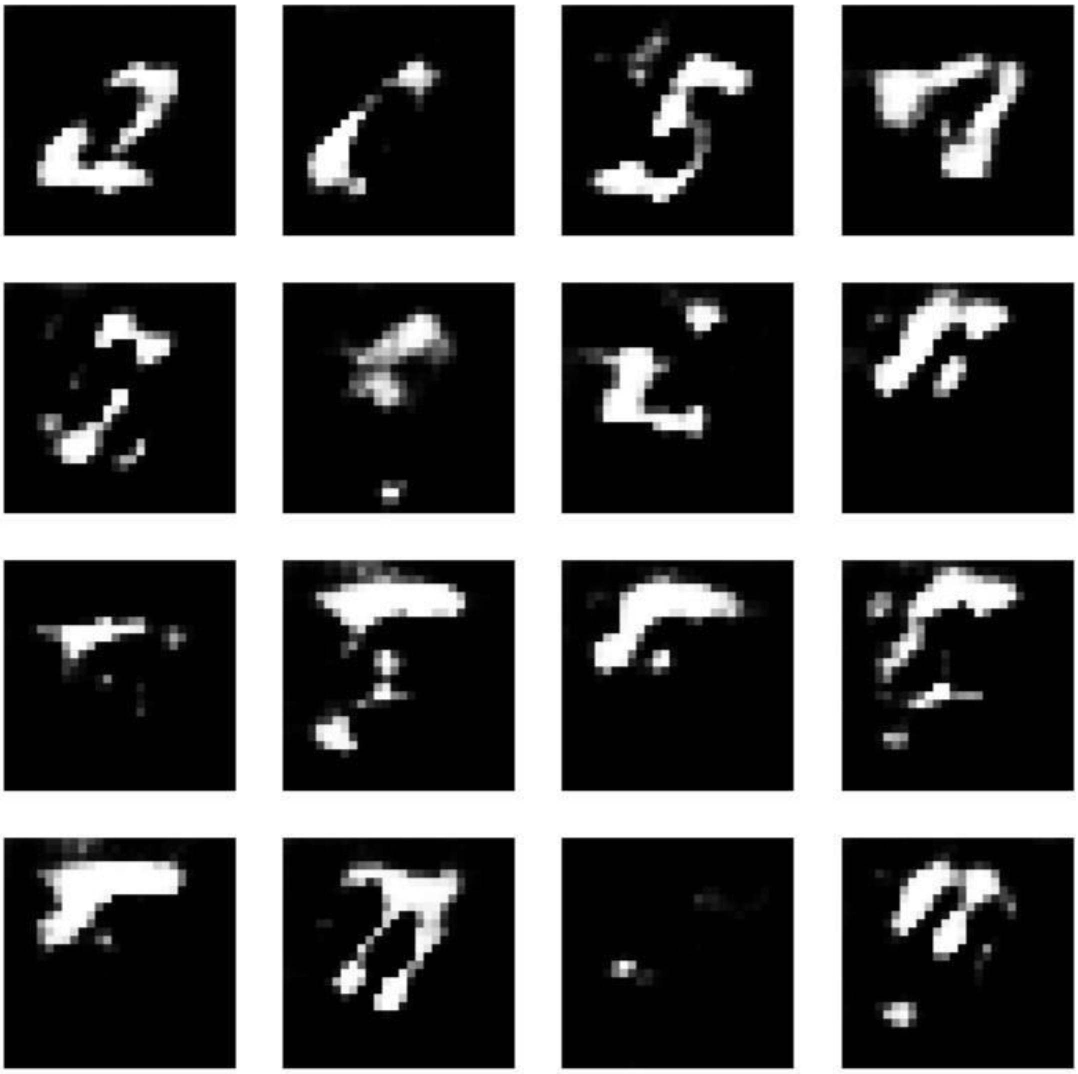
```
Epoch 6/20 - Generator Loss: 0.8516, Discriminator Loss: 1.2705
Epoch 7/20 - Generator Loss: 0.8888, Discriminator Loss: 1.3028
Epoch 8/20 - Generator Loss: 0.8739, Discriminator Loss: 1.2512
Epoch 9/20 - Generator Loss: 0.8691, Discriminator Loss: 1.3130
Epoch 10/20 - Generator Loss: 0.8862, Discriminator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

Epoch 10



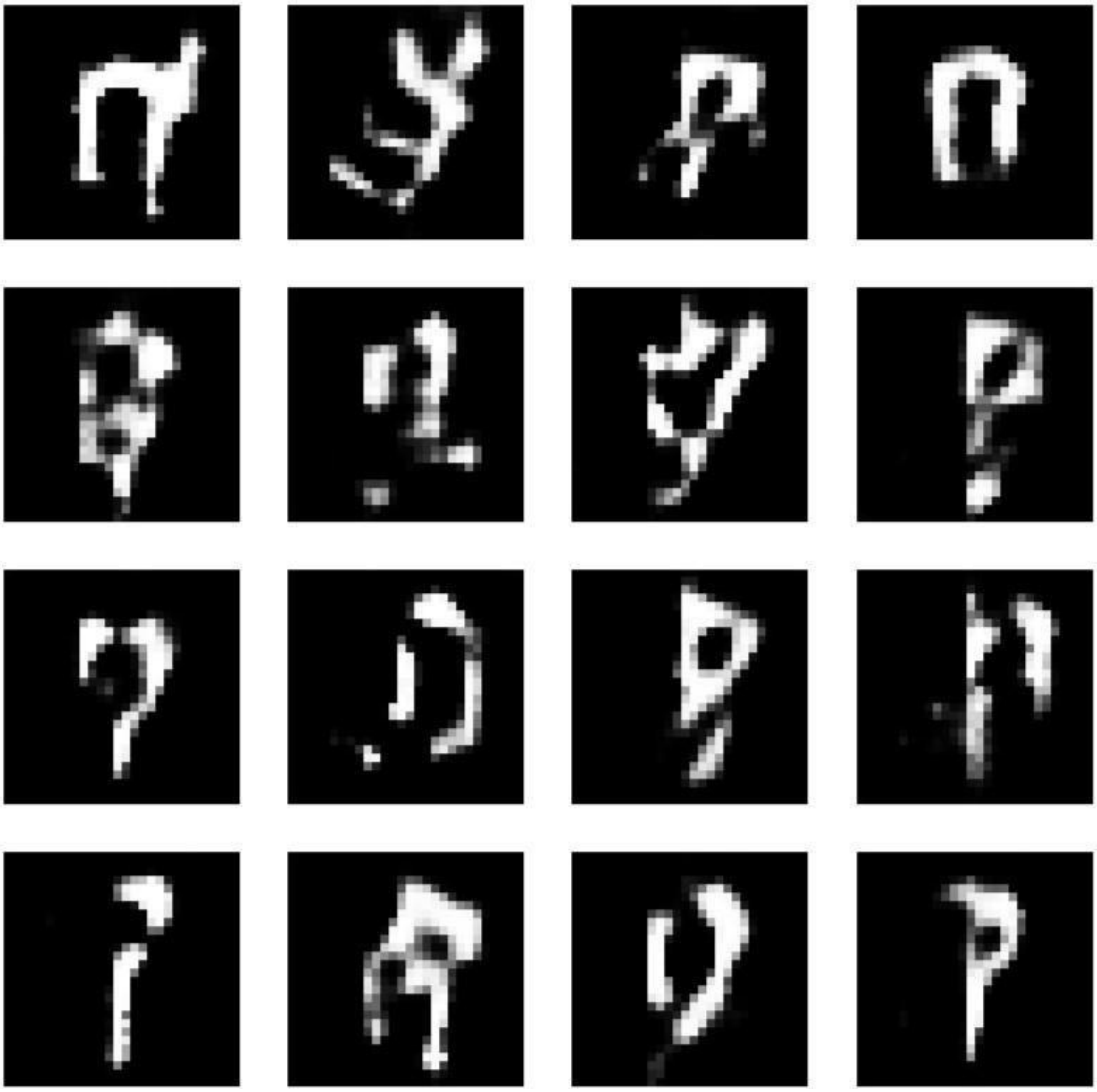
```
Epoch 11/20 - Generator Loss: 0.9361, Discriminator Loss: 1.2244
Epoch 12/20 - Generator Loss: 0.9946, Discriminator Loss: 1.1719
Epoch 13/20 - Generator Loss: 0.9948, Discriminator Loss: 1.1944
Epoch 14/20 - Generator Loss: 0.9786, Discriminator Loss: 1.1809
Epoch 15/20 - Generator Loss: 1.0420,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

Epoch 15

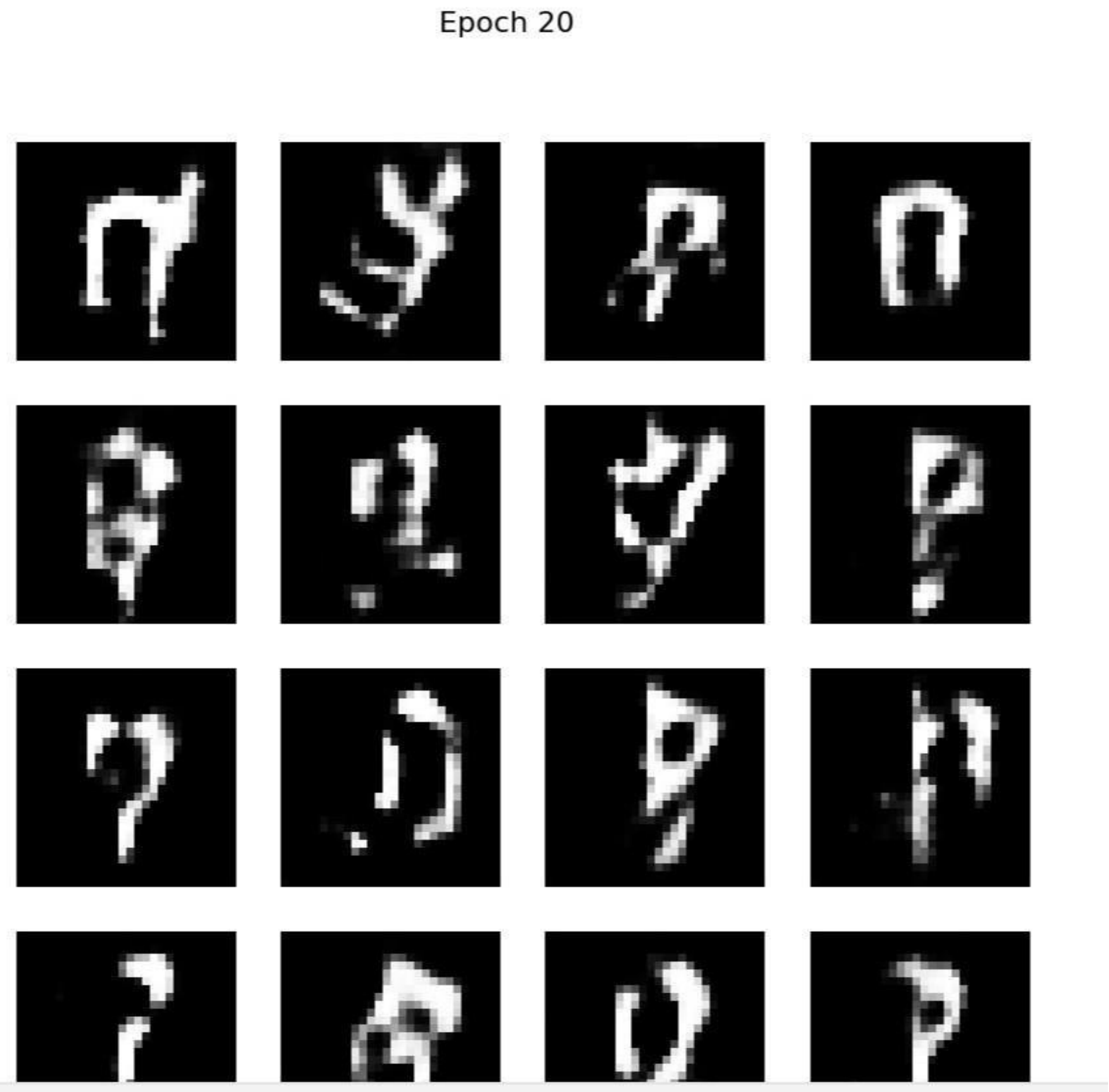


```
Epoch 16/20 - Generator Loss: 1.2020, Discriminator Loss: 1.0483
Epoch 17/20 - Generator Loss: 1.2648, Discriminator Loss: 1.0605
Epoch 18/20 - Generator Loss: 1.1657, Discriminator Loss: 1.0404
Epoch 19/20 - Generator Loss: 1.1644, Discriminator Loss: 1.0897
Epoch 20/20 - Generator Loss: 1.1770,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

Epoch 20



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



RESULT:

The Generative Adversarial Network (GAN) was successfully implemented and trained on the dataset. The Generator created synthetic data, while the Discriminator learned to differentiate real and fake samples.

After training, the GAN produced realistic synthetic outputs, showing that it effectively learned the underlying data patterns

Exp No: 8	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'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}")
```

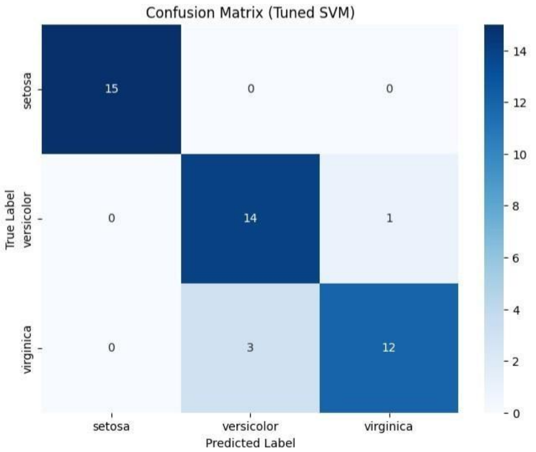
```
# 5. Perform Grid Search with Cross-Validation
# GridSearchCV automatically performs k-fold cross-validation for each combination.
# cv=5 means 5-fold cross-validation.
# scoring='accuracy' means we want to optimize for accuracy.
grid_search = GridSearchCV(SVC(), param_grid, cv=5, scoring='accuracy',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

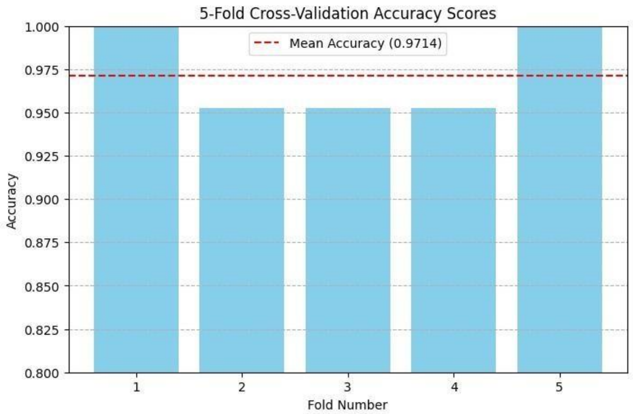
```
({np.mean(cv_scores):.4f})')
plt.title(f'{k_folds}-Fold Cross-Validation Accuracy Scores')
plt.xlabel('Fold Number')
plt.ylabel('Accuracy')
plt.ylim(0.8, 1.0) # Set y-axis limits for better visualization
plt.legend()
plt.grid(axis='y', linestyle='--')
plt.show()

# 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.
```

RESULT:

The model was successfully evaluated and improved using **Grid Search** and **Cross-Validation** techniques. Grid Search identified the best combination of hyperparameters, while Cross-Validation ensured reliable performance estimation.

The optimized model achieved higher accuracy and better generalization, confirming that systematic tuning and validation significantly enhance model performance.

EXP NO : 9	MINI PROJECT : AI RESUME SCREENER USING TF-IDF AND COSINE SIMILARITY

AIM:

To develop a secure, AI-powered hiring assistant web application that **automates resume screening** and ranking against job descriptions. The system will leverage Natural Language Processing (NLP) techniques to compute similarity scores, boosting efficiency and ensuring a data-driven, unbiased selection process for recruiters.

ALGORITHM:

1. **User Interface & Authentication (Streamlit/Python/SQL):** Implement the Login and Registration UIs using Streamlit. Securely store user credentials (hashed passwords) in a **SQL database**.
2. **Job Information & Upload (Streamlit/Python):** Create the Dashboard UI for inputting the **Job Title** and **Job Description**. Implement file handling to allow recruiters to **upload multiple PDF resumes**.
3. **Resume Parsing & Embedding (Python/NLP):** Use NLP libraries (like spaCy or NLTK) to **parse the text from the uploaded PDFs** and the job description. Convert the text data into numerical **vector embeddings** (e.g., using techniques like TF-IDF or BERT).
4. **Similarity Calculation & Ranking (Python/NLP):** Calculate the **cosine similarity** between the job description embedding and each resume embedding. **Rank the resumes** based on their similarity scores and store the ranking history (Job Title, Date, Results) in the **SQL database**.

CODE:

App.py:

```
import streamlit as st
import pandas as pd
import os
import io
from pypdf import PdfReader
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import hashlib
import uuid
from datetime import datetime
import sqlite3

# --- Streamlit Page Config ---
```

```
st.set_page_config(
    page_title="HireSense AI",
    page_icon="", layout="wide",
    initial_sidebar_state="expanded"
)

# --- Database Setup ---
def init_db():
    """Initialize SQLite database with necessary tables"""
    # Using a local SQLite file for demonstration
    conn = sqlite3.connect('Resume.db')
    c = conn.cursor()

    # Create users table
    c.execute('''
    CREATE TABLE IF NOT EXISTS users (
        email TEXT PRIMARY KEY,
        password TEXT NOT NULL,
        name TEXT,
        job_title TEXT,
        company TEXT,
        date_joined TEXT,
        last_login TEXT
    )
    ''')

    # Create ranking history table
    c.execute('''
    CREATE TABLE IF NOT EXISTS ranking_history (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        email TEXT NOT NULL,
        timestamp TEXT NOT NULL,
        job_title TEXT,
        description TEXT,
        results TEXT,
        FOREIGN KEY (email) REFERENCES users (email)
    )
    ''')

    conn.commit()
    conn.close()

# --- Initialize Session State ---
if "authenticated" not in st.session_state:
    st.session_state["authenticated"] = False
    st.session_state["user_email"] = None
    st.session_state["user_name"] = None
```

```
    st.session_state["current_page"] = "login"  # Default page: login, register, dashboard, profile

# --- Security Functions ---
def hash_password(password, salt=None):
    """Hash a password for storing."""
    if salt is None:
        salt = uuid.uuid4().hex
    # Hash the salt and password combination
    hashed = hashlib.sha256(salt.encode() + password.encode()).hexdigest()
    return f"{salt}${hashed}"

def verify_password(stored_password, provided_password):
    """Verify a stored password against one provided by user"""
    try:
        salt, hashed = stored_password.split('$')
        return hashed == hashlib.sha256(salt.encode() + provided_password.encode()).hexdigest()
    except ValueError:
        # Handle cases where the stored password is not in the expected format (e.g., legacy or malformed)
        return False

# --- User Management Functions ---
def save_user(email, password, name=""):
    """Registers a new user in the database."""
    conn = sqlite3.connect('Resume.db')
    c = conn.cursor()

    # Check if user exists
    c.execute("SELECT email FROM users WHERE email = ?", (email,))
    if c.fetchone():
        conn.close()
        return False  # User already exists

    # Hash the password
    hashed_password = hash_password(password)

    # Create new user with timestamp
    current_date = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    c.execute(
        "INSERT INTO users (email, password, name, job_title, company, date_joined, last_login) VALUES (?,
?, ?, ?, ?, ?, ?)",
        (email, hashed_password, name, "", "", current_date, current_date)
    )

    conn.commit()
    conn.close()
    return True

def authenticate_user(email, password):
```

```
    """Authenticate a user with email and password."""
    conn = sqlite3.connect('Resume.db')
    c = conn.cursor()

    c.execute("SELECT password FROM users WHERE email = ?", (email,))
    result = c.fetchone()

    if not result:
        conn.close()
        return False

    stored_password = result[0]

    if verify_password(stored_password, password):
        # Update last login time
        current_date = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        c.execute("UPDATE users SET last_login = ? WHERE email = ?", (current_date, email))
        conn.commit()
        conn.close()
        return True

    conn.close()
    return False

def update_profile(email, name, job_title, company):
    """Update user profile information."""
    conn = sqlite3.connect('Resume.db')
    c = conn.cursor()

    c.execute(
        "UPDATE users SET name = ?, job_title = ?, company = ? WHERE email = ?",
        (name, job_title, company, email)
    )

    conn.commit()
    conn.close()
    return True

def get_user_profile(email):
    """Get user profile data."""
    conn = sqlite3.connect('Resume.db')
    c = conn.cursor()

    c.execute(
        "SELECT email, name, job_title, company, date_joined, last_login FROM users WHERE email = ?",
        (email,)
    )
```

```
    result = c.fetchone()
    conn.close()

    if not result:
        return None

    return {
        "email": result[0],
        "name": result[1],
        "job_title": result[2],
        "company": result[3],
        "date_joined": result[4],
        "last_login": result[5]
    }

def change_password(email, current_password, new_password):
    """Change user password."""
    conn = sqlite3.connect('Resume.db')
    c = conn.cursor()

    c.execute("SELECT password FROM users WHERE email = ?", (email,))
    result = c.fetchone()

    if not result:
        conn.close()
        return False, "User not found"

    stored_password = result[0]

    if not verify_password(stored_password, current_password):
        conn.close()
        return False, "Current password is incorrect"

    # Hash the new password
    hashed_password = hash_password(new_password)

    # Update password
    c.execute("UPDATE users SET password = ? WHERE email = ?", (hashed_password, email))
    conn.commit()
    conn.close()

    return True, "Password changed successfully"

# --- Resume History Functions ---
def save_ranking_history(email, job_title, description, results_df):
    """Save resume ranking history for the user. Results DataFrame is converted to JSON."""
    conn = sqlite3.connect('Resume.db')
    c = conn.cursor()
```

```
    # Convert DataFrame to JSON string for storage
    results_json = results_df.to_json()

    # Create new history entry
    c.execute(
        "INSERT INTO ranking_history (email, timestamp, job_title, description, results) VALUES (?, ?, ?, ?,
?)",
        (
            email,
            datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
            job_title,
            description,
            results_json
        )
    )

    conn.commit()
    conn.close()

def get_user_history(email):
    """Get resume ranking history for the user."""
    conn = sqlite3.connect('Resume.db')

    # Get all history records for the user
    query = "SELECT id, timestamp, job_title, description, results FROM ranking_history WHERE email = ?
ORDER BY timestamp DESC"
    history_df = pd.read_sql_query(query, conn, params=(email,))

    conn.close()

    return history_df

# --- Resume Processing Functions ---
def extract_text_from_pdf(file):
    """Extracts text from an uploaded PDF file."""
    try:
        # Reset file pointer to the beginning
        file.seek(0)
        pdf = PdfReader(file)
        text = ""
        for page in pdf.pages:
            page_text = page.extract_text()
            if page_text:
                text += page_text + "\n"
        return text.strip() if text else "No readable text found."
    except Exception as e:
        # Return error message instead of failing silently
```

```
        return f"Error extracting text: {str(e)}"

def rank_resumes(job_description, resumes):
    """Ranks resumes based on their similarity to the job description using TF-IDF and Cosine Similarity."""

    # Create the list of documents: Job Description first, then all Resumes
    documents = [job_description] + resumes

    # Initialize and fit the TF-IDF Vectorizer
    vectorizer = TfidfVectorizer()
    vector_matrix = vectorizer.fit_transform(documents)

    # Get the vectors (Sparse matrix to dense array)
    vectors = vector_matrix.toarray()

    # The first vector is the Job Description
    job_description_vector = vectors[0]
    # The remaining vectors are the Resumes
    resume_vectors = vectors[1:]

    # Calculate Cosine Similarity between the Job Description and all Resumes
    # cosine_similarity expects a list of vectors for the first argument, even if it's one.
    cosine_similarities = cosine_similarity([job_description_vector], resume_vectors).flatten()

    return cosine_similarities

# Add custom CSS for better styling and responsive design
st.markdown("""
    <style>
        .stButton>button {
            background-color: #1E90FF; /* Dodger Blue */
            color: white;
            font-size: 16px;
            border-radius: 8px; /* Slightly more rounded */
            padding: 10px 20px;
            transition: all 0.3s ease;
            box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1); /* Subtle shadow */
            border: none;
        }
        .stButton>button:hover {
            background-color: #4682B4; /* Steel Blue */
            box-shadow: 0 6px 8px rgba(0, 0, 0, 0.15);
        }
        .stTextInput>div>div>input, .stTextArea>div>div>textarea {
            font-size: 16px;
            border-radius: 8px;
            border: 1px solid #ccc;
            padding: 10px;
```

```
        }
        /* Style for the sidebar title */
        .sidebar .stMarkdown h2 {
            text-align: center;
            font-weight: bold;
            font-size: 40px;
            margin-bottom: 20px;
        }
        /* Responsive main content container */
        .block-container {
            padding-top: 2rem;
            padding-bottom: 5rem; /* Space for the footer */
        }
        /* Gradient title style */
        .gradient-title {
            background: -webkit-linear-gradient(45deg, #1FA2FF, #12D8FA, #A6FFCB);
            -webkit-background-clip: text;
            -webkit-text-fill-color: transparent;
            font-weight: 800;
            text-align: center;
            font-size: 3rem;
        }
        /* Footer style */
        .footer {
            position: fixed;
            left: 0;
            bottom: 0;
            width: 100%;
            background-color: #f1f1f1;
            color: #555;
            text-align: center;
            padding: 8px 0;
            font-size: 12px;
            border-top: 1px solid #ccc;
            z-index: 100;
        }
        /* Info/Warning/Error boxes */
        [data-testid="stAlert"] {
            border-radius: 8px;
        }
    </style>
""", unsafe_allow_html=True)

# --- Login / Register Pages ---
def show_login_page():
    st.sidebar.title("� User Login")
    st.sidebar.markdown("### Please enter your credentials.")
```

```
    login_email = st.sidebar.text_input("? Email", key="login_email", placeholder="Enter your email")
    login_password = st.sidebar.text_input("? Password", type="password", key="login_password",
placeholder="Enter your password")

    st.sidebar.markdown("---")

    col1, col2 = st.sidebar.columns(2)

    with col1:
        if st.button("? Login", use_container_width=True):
            if authenticate_user(login_email, login_password):
                st.session_state["authenticated"] = True
                st.session_state["user_email"] = login_email
                profile = get_user_profile(login_email)
                st.session_state["user_name"] = profile["name"] if profile and profile["name"] else
login_email.split('@')[0]
                st.session_state["current_page"] = "dashboard"
                st.rerun()
            else:
                st.sidebar.error("❌Invalid email or password")

    with col2:
        if st.button("? Register", use_container_width=True):
            st.session_state["current_page"] = "register"
            st.rerun()

def show_register_page():
    st.sidebar.title("? User Registration")
    st.sidebar.markdown("### Create a new account.")

    reg_email = st.sidebar.text_input("? Email*", key="reg_email", placeholder="Enter your work email")
    reg_name = st.sidebar.text_input("? Full Name", key="reg_name", placeholder="Enter your full
name")
    reg_password = st.sidebar.text_input("? Password*", type="password", key="reg_password",
placeholder="Create a strong password")
    reg_confirm_password = st.sidebar.text_input("? Confirm Password*", type="password",
key="reg_confirm_password", placeholder="Confirm your password")

    st.sidebar.markdown("---")

    col1, col2 = st.sidebar.columns(2)

    with col1:
        if st.button("✅Register", use_container_width=True):
            if not reg_email or not reg_password or not reg_confirm_password:
                st.sidebar.error("❌Email, password, and confirmation are required")
            elif "@" not in reg_email or "." not in reg_email:
                st.sidebar.error("❌Invalid email format")
```

```
            elif reg_password != reg_confirm_password:
                st.sidebar.error("❌Passwords do not match")
            else:
                if save_user(reg_email, reg_password, reg_name):
                    st.sidebar.success("✅Registration successful! You can now log in.")
                    st.session_state["current_page"] = "login"
                    st.rerun()
                else:
                    st.sidebar.warning("⚠Email already registered. Please log in instead.")
                    st.session_state["current_page"] = "login"
                    st.rerun()

    with col2:
        if st.button("↩ Back to Login", use_container_width=True):
            st.session_state["current_page"] = "login"
            st.rerun()

# --- Authenticated Pages ---

def show_profile_page():
    st.title("� User Profile")
    st.markdown("### Manage your profile and review history.")

    profile = get_user_profile(st.session_state["user_email"])
    if not profile:
        st.error("❌Error loading profile data. Please try logging in again.")
        return

    # Profile tabs
    profile_tab, password_tab, history_tab = st.tabs(["☒ Edit Profile", "� Change Password", "� History"])

    with profile_tab:
        st.subheader("Personal Information")

        # Display read-only info
        st.info(f"**Email:** `{profile['email']}`")
        st.info(f"**Member Since:** {profile['date_joined'].split(' ')[0]}")

        # Editable fields
        name = st.text_input("Full Name", value=profile["name"] if profile["name"] else "")
        job_title = st.text_input("Job Title", value=profile["job_title"] if profile["job_title"] else "")
        company = st.text_input("Company", value=profile["company"] if profile["company"] else "")

        if st.button("� Save Profile"):
            if update_profile(profile["email"], name, job_title, company):
                # Update session state with new name
                st.session_state["user_name"] = name if name else profile["email"].split('@')[0]
```

```
            st.success("✅Profile updated successfully! Rerunning application...")
            st.rerun()
        else:
            st.error("❌Error updating profile")

    with password_tab:
        st.subheader("Change Password")

        # Use unique keys to avoid conflict with login page
        current_password = st.text_input("Current Password", type="password",
key="p_current_password")
        new_password = st.text_input("New Password", type="password", key="p_new_password")
        confirm_new_password = st.text_input("Confirm New Password", type="password",
key="p_confirm_new_password")

        if st.button("� Update Password"):
            if not current_password or not new_password or not confirm_new_password:
                st.error("❌All fields are required.")
            elif new_password != confirm_new_password:
                st.error("❌New passwords do not match.")
            else:
                success, message = change_password(profile["email"], current_password, new_password)
                if success:
                    st.success(f"✅{message}")
                else:
                    st.error(f"❌{message}")

    with history_tab:
        st.subheader("Resume Ranking History")

        history = get_user_history(profile["email"])
        if history.empty:
            st.info("� No ranking history found. Run a job screening on the Dashboard to start collecting
history.")
        else:
            st.success(f"Found {len(history)} past screening sessions.")
            for idx, row in history.iterrows():
                # Use the database ID to ensure unique key for expander
                history_id = row['id']

                with st.expander(f"**{row['job_title']}** - Ran on: {datetime.strptime(row['timestamp'], '%Y-
%m-%d %H:%M:%S').strftime('%b %d, %Y %I:%M %p')}"):

                    st.text_area("Job Description Used", value=row["description"], height=100, disabled=True,
key=f"job_desc_hist_{history_id}")
                    st.markdown("#### **Ranked Results**")

                    try:
```

```
                # Convert the stored JSON string back to a DataFrame
                results_df = pd.read_json(row["results"])
                # Only show the user-friendly columns in the history view
                st.dataframe(results_df.drop(columns=["Raw Score"]), hide_index=True,
use_container_width=True)
            except Exception as e:
                st.warning(f"⚠Error loading detailed results for this entry: {str(e)}")


def show_dashboard():
    # Use session state name or default to the email part
    welcome_name = st.session_state.get("user_name", st.session_state["user_email"].split('@')[0])

    # Title with gradient effect using HTML class
    st.markdown(f"""
        <h2 class='gradient-title'>� HireSense AI - Intelligent Screening</h2>
        <div style='text-align:center; font-size:18px;'>
            Welcome back, <b style='color:#4CAF50;'>{welcome_name}</b> � |
            <small>Logged in as: <i>{st.session_state["user_email"]}</i></small>
        </div>
        <hr style="margin: 20px 0;">
    """, unsafe_allow_html=True)

    # --- Job Information Section ---
    st.subheader("� Target Job Information")
    job_title = st.text_input("Job Title", placeholder="e.g., Senior Data Scientist", label_visibility="visible")

    st.markdown("---")

    # --- Job Description & Resume Upload ---
    st.subheader("� Job Description & � Resume Upload")

    col1, col2 = st.columns([1.2, 1])

    with col1:
        job_description = st.text_area(
            "Job Description (Paste or write the full details)",
            placeholder="Key skills, required experience, and responsibilities...",
            height=250,
            key="job_desc"
        )

    with col2:
        st.markdown("#### Upload Candidate Resumes (PDF)")
        uploaded_files = st.file_uploader(
            "Select one or more PDF files",
            type=["pdf"],
            accept_multiple_files=True,
```

```
        key="resume_files"
    )

    if uploaded_files:
        st.info(f"✅{len(uploaded_files)} resume(s) selected for processing.")

st.markdown("---")

# --- Processing & Ranking ---
# Button is disabled if essential inputs are missing
is_ready = uploaded_files and job_description
if st.button("□ Rank Resumes Now", disabled=not is_ready, use_container_width=True): if
    not job_title.strip():
        st.warning("Please enter a **Job Title** for better history tracking.")

    with st.spinner("□ Processing resumes and calculating similarity scores..."):
        resumes = []
        file_names = []
        error_files = []

        # 1. Process each resume
        for file in uploaded_files:
            text = extract_text_from_pdf(file)
            if "Error extracting text" in text or "No readable text found" in text:
                error_files.append(file.name)
            else:
                resumes.append(text)
                file_names.append(file.name)

        if error_files:
            st.warning(f"⚠Could not process {len(error_files)} files: {', '.join(error_files)}. They will be
excluded from ranking.")

        if resumes:
            # 2. Rank resumes using NLP
            scores = rank_resumes(job_description, resumes)

            # 3. Combine results and sort (Ranked Resumes)
            ranked_resumes = sorted(zip(file_names, scores), key=lambda x: x[1], reverse=True)

            # Create final results dataframe
            results_df = pd.DataFrame({
                "Rank": range(1, len(ranked_resumes) + 1),
                "Resume Name": [name for name, _ in ranked_resumes],
                "Match Score": [f"{round(score * 100, 2)}%" for _, score in ranked_resumes],
                "Raw Score": [round(score, 4) for _, score in ranked_resumes] # Raw score for chart
            })
```

```
        # 4. Display results
        st.success("✅Ranking complete! Displaying results below.")
        st.subheader("□ Ranked Candidate Summary")

        # Drop the raw score column for display
        st.dataframe(results_df.drop(columns=["Raw Score"]), hide_index=True,
use_container_width=True)

        # 5. Visualize top candidates
        st.subheader("□ Top Match Score Visualization")

        # Prepare data for chart: use Raw Score for numerical plotting
        chart_data = results_df.head(min(len(results_df), 10)).copy()
        chart_data['Resume Name'] = chart_data['Resume Name'].apply(lambda x: x.split('.')[0])

        st.bar_chart(chart_data, x="Resume Name", y="Raw Score", color="#1E90FF")

        # 6. Save ranking history
        save_ranking_history(
          st.session_state["user_email"],
          job_title if job_title else "Unnamed Job",
          job_description,
          results_df
        )

        st.caption("Results saved to your **History** tab in My Profile.")

        # 7. Download options
        col1, col2 = st.columns(2)
        with col1:
          csv = results_df.drop(columns=["Raw Score"]).to_csv(index=False).encode('utf-8')
          st.download_button("□ Download Results (CSV)", csv, f"{job_title.replace(' ',
'_')}_ranked_resumes.csv", "text/csv", use_container_width=True)
        with col2:
          buffer = io.BytesIO()
          # Use ExcelWriter to save the DataFrame to the in-memory buffer
          with pd.ExcelWriter(buffer, engine='openpyxl') as writer:
            results_df.drop(columns=["Raw Score"]).to_excel(writer, index=False,
sheet_name='Ranked Resumes')
          buffer.seek(0)
          st.download_button("□ Download Results (Excel)", buffer, f"{job_title.replace(' ',
'_')}_ranked_resumes.xlsx",
                    "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",
use_container_width=True)
      else:
        st.error("❌No valid resumes could be processed. Please check your uploaded files.")

# --- App Sidebar ---
```

```
def        render_sidebar():
    st.sidebar.markdown("""
<h2 style="
    text-align: center;
    font-weight: bold;
    font-size: 40px;
    background: linear-gradient(90deg, #4CAF50, #2196F3);
    -webkit-background-clip: text;
    -webkit-text-fill-color: transparent;
">
    HireSense AI
</h2>
            """, unsafe_allow_html=True)

    if st.session_state["authenticated"]:
        # Sidebar status for authenticated user
        name_display = st.session_state.get("user_name", st.session_state["user_email"].split('@')[0])
        st.sidebar.subheader(f"� Hello, {name_display}")

        # Navigation
        st.sidebar.markdown("---")
        st.sidebar.subheader("� Navigation")

        # Use a list of tuples to handle button clicks and rerunning logic
        pages = [
            ("� Dashboard", "dashboard"),
            ("� My Profile", "profile")
        ]

        # Dynamic button rendering and logic
        for label, page_name in pages:
            if st.sidebar.button(label, use_container_width=True, key=f"nav_{page_name}",
                          # Highlight the current page button
                          type="primary" if st.session_state["current_page"] == page_name else "secondary"):
                st.session_state["current_page"] = page_name
                st.rerun()

        # Logout Button
        st.sidebar.markdown("---")
        if st.sidebar.button("� Logout", use_container_width=True):
            st.session_state["authenticated"] = False
            st.session_state["user_email"] = None
            st.session_state["user_name"] = None
            st.session_state["current_page"] = "login"
            st.sidebar.success("� Logged out successfully! Rerunning application...")
            st.rerun()

# --- Global Footer (outside sidebar) ---
```

```
def render_footer():
    st.markdown("""
        <div class="footer">
            © 2025 HireSense AI | Built with Streamlit, Python, and SQLite
        </div>
    """, unsafe_allow_html=True)


# --- Main App Logic ---
def main():
    # 1. Initialize database at the start of the application
    init_db()

    # 2. Render the sidebar (which includes navigation and user status)
    render_sidebar()

    # 3. Determine which main content page to display
    if not st.session_state.get("authenticated", False):
        # Unauthenticated Pages
        if st.session_state["current_page"] == "register":
            st.title("User Registration")
            st.markdown("### Create a new account")
            show_register_page()
            st.markdown("---")
            # Display landing info next to the registration form
            st.markdown("""
            ### ? Intelligent Hiring Starts Here
            HireSense AI automates the tedious screening process, letting you focus on the best candidates.
            """)
        else: # Default to login page
            st.markdown("""
<h1 style='
    text-align: left;
    font-weight: bold;
    font-size: 48px;
    background: linear-gradient(90deg, #4CAF50, #2196F3);
    -webkit-background-clip:  text;
    -webkit-text-fill-color:  transparent;
'>
? Welcome to HireSense AI
</h1>
""", unsafe_allow_html=True)
            st.subheader("Your AI-powered hiring assistant")

            st.markdown("""
            ### ? Key Features:
            - **Intelligent Matching**: Uses TF-IDF and Cosine Similarity for highly accurate skill-to-job
matching.
```

```
        - **Secure Profiles**: User accounts are secured with hashed passwords (SQLite backend).
        - **History Tracking**: Saves every screening session for future reference and comparison.
        """)

        # Display login form next to the info panel
        st.markdown("---")
        show_login_page()

    else:
        # Authenticated Pages
        if st.session_state["current_page"] == "dashboard":
            show_dashboard()
        elif st.session_state["current_page"] == "profile":
            show_profile_page()

    # 4. Render the fixed footer
    render_footer()

if __name___ == "__main__":
    main()
```

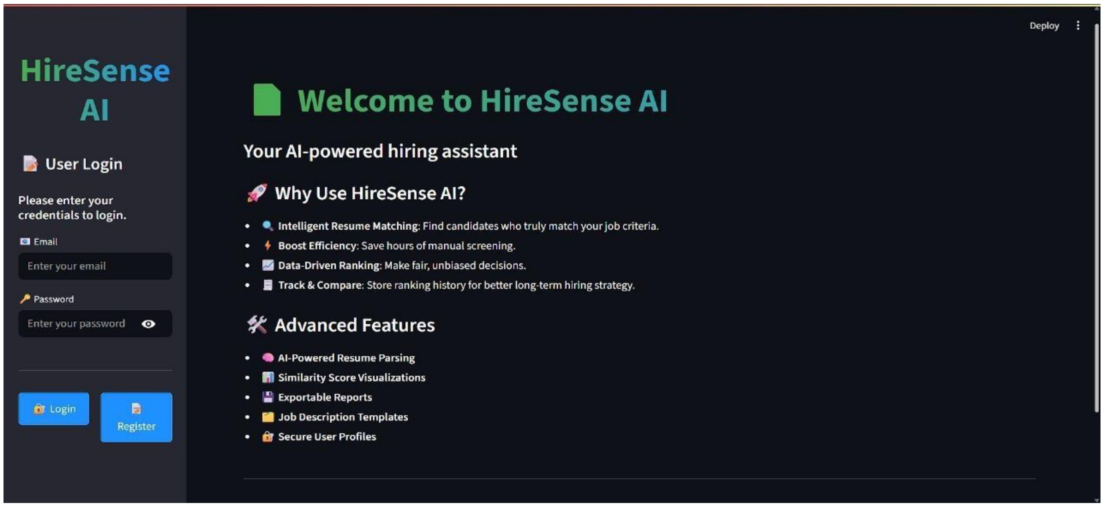
Requirements.txt:

```
streamlit==1.32.2

pandas==2.2.2

pypdf==4.1.0

scikit-learn==1.4.2
```

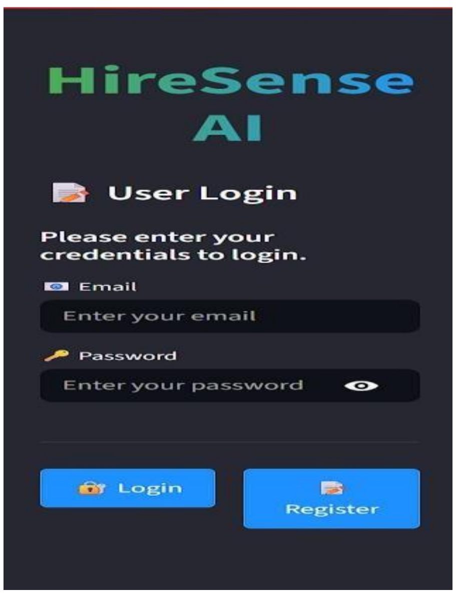
settings.json:

```
{

  "git.ignoreLimitWarning":  true

}
```

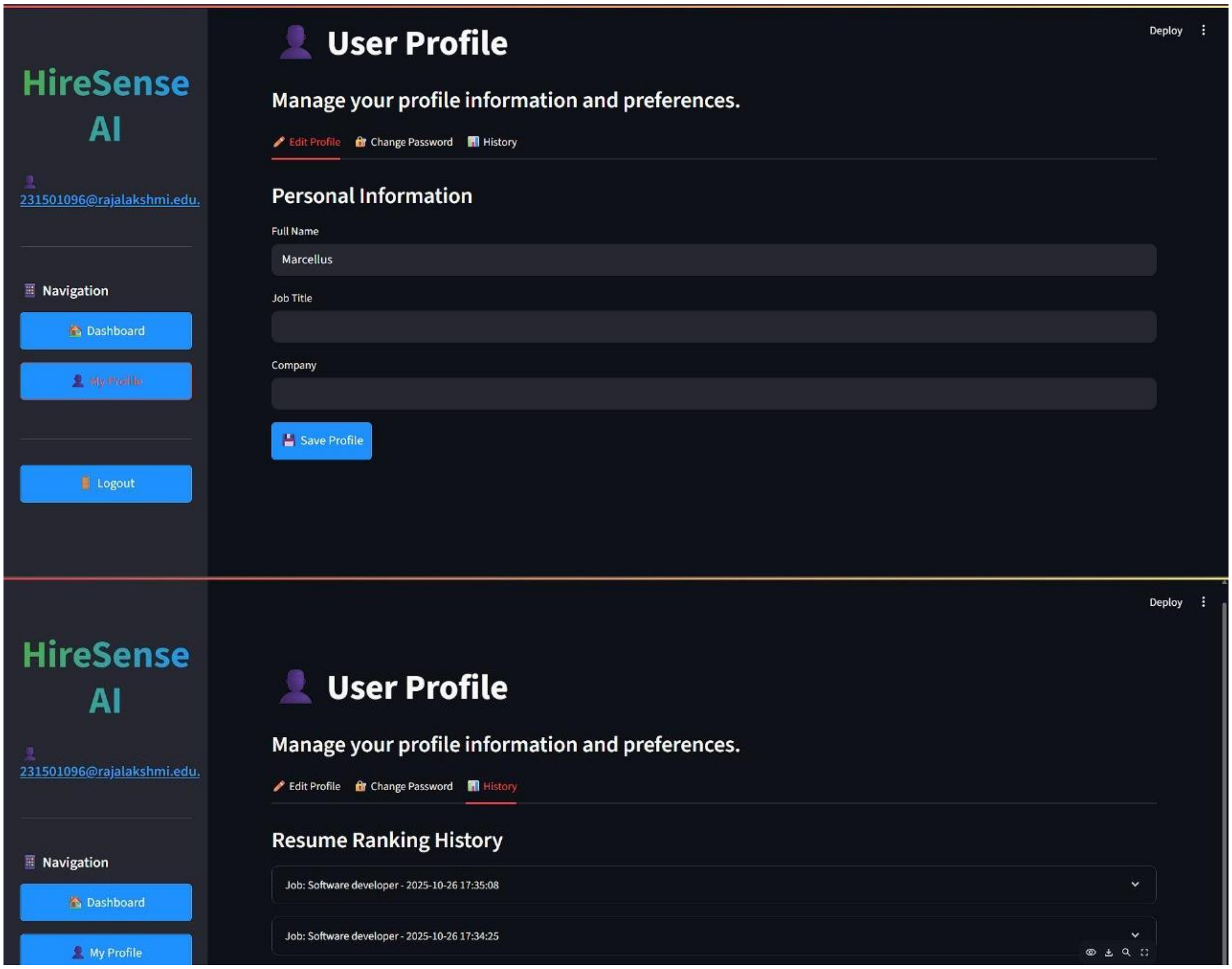
OUTPUT:



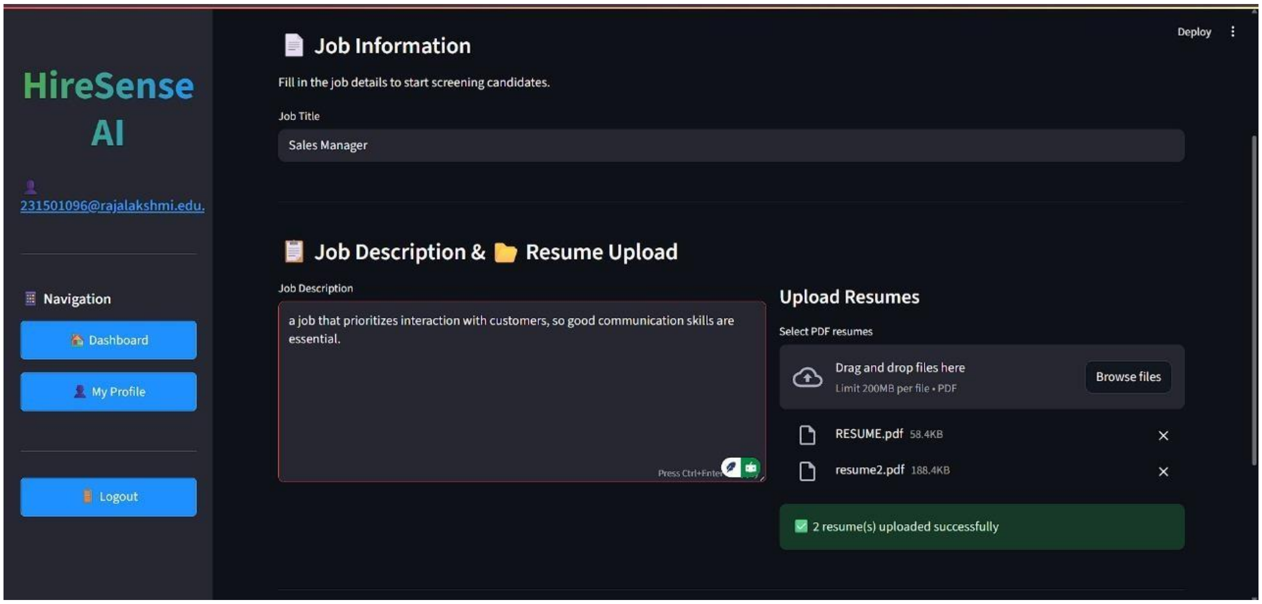
USER LOGIN:



USER PROFILE ENTERING :



INSIGHTS:



RESULT:

This application provides a **multi-page, authenticated experience** for human resources and recruiters to quickly screen candidates against a job description using a simple NLP technique.