**Expriment No.7**

**Code:-**

import numpy as np

import matplotlib.pyplot as plt from hmmlearn import hmm

from sklearn.preprocessing import KBinsDiscretizer, LabelEncoder from sklearn.metrics import accuracy\_score

# Simulated dataset: Temperature (Celsius), Humidity (%), and Actual Weather Labels

data = np.array([

[30, 40, 'sunny'], [32, 42, 'sunny'], [35, 50, 'sunny'],

[28, 60, 'cloudy'], [26, 65, 'cloudy'], [25, 70, 'cloudy'],

[22, 80, 'rainy'], [20, 85, 'rainy'], [18, 90, 'rainy']

])

# Extract temperature, humidity, and labels temp\_humidity = data[:, :2].astype(float) labels = data[:, 2]

# Encoding states weather\_encoder = LabelEncoder()

states = weather\_encoder.fit\_transform(labels) # sunny=2, cloudy=1, rainy=0 n\_states = len(set(states))

# Discrete HMM: Discretizing temperature and humidity into 3 bins discretizer = KBinsDiscretizer(n\_bins=3, encode='ordinal', strategy='uniform') discrete\_obs = discretizer.fit\_transform(temp\_humidity).astype(int)

# Train Discrete HMM

discrete\_hmm = hmm.MultinomialHMM(n\_components=n\_states, n\_iter=100) discrete\_hmm.fit(discrete\_obs)

predicted\_states\_discrete = discrete\_hmm.predict(discrete\_obs) accuracy\_discrete = accuracy\_score(states, predicted\_states\_discrete) # Continuous HMM: Train a Gaussian HMM

continuous\_hmm = hmm.GaussianHMM(n\_components=n\_states, covariance\_type='diag', n\_iter=100)

continuous\_hmm.fit(temp\_humidity)

predicted\_states\_continuous = continuous\_hmm.predict(temp\_humidity) accuracy\_continuous = accuracy\_score(states, predicted\_states\_continuous) # Plot Accuracy Comparison

plt.bar(['Discrete HMM', 'Continuous HMM'], [accuracy\_discrete, accuracy\_continuous])

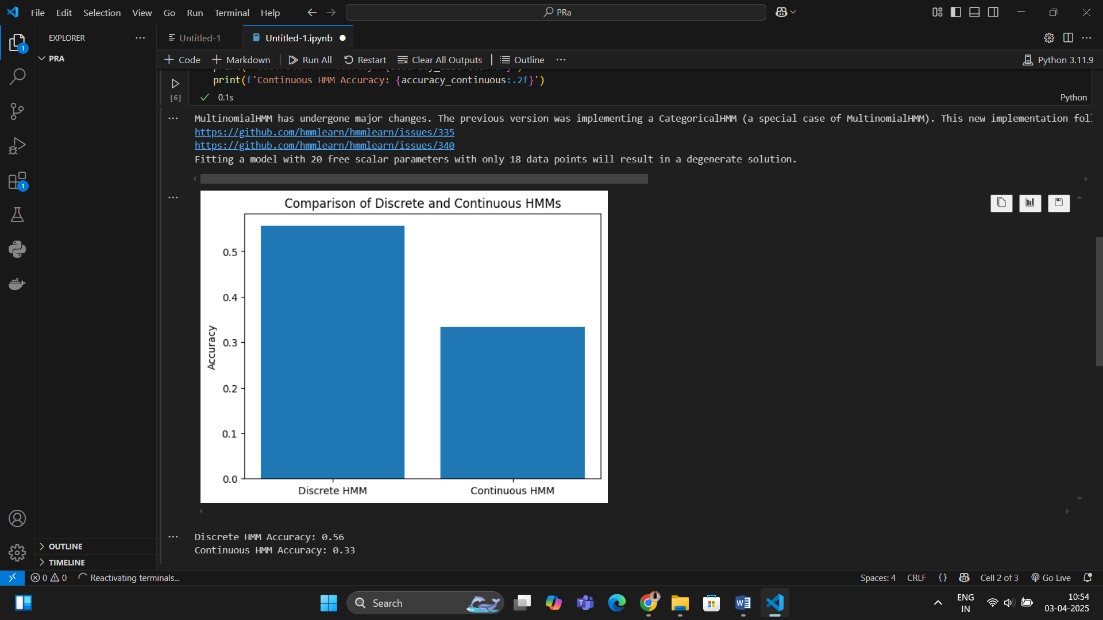
plt.ylabel('Accuracy')

plt.title('Comparison of Discrete and Continuous HMMs') plt.show()

# Print Results

print(f'Discrete HMM Accuracy: {accuracy\_discrete:.2f}') print(f'Continuous HMM Accuracy: {accuracy\_continuous:.2f}')

**Output:-**



**Expriment No.8**

**Code:-**

import numpy as np

import matplotlib.pyplot as plt from hmmlearn import hmm

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

import seaborn as sns

# Define DNA sequence encoding: A=0, C=1, G=2, T=3 dna\_mapping = {'A': 0, 'C': 1, 'G': 2, 'T': 3}

state\_mapping = {'Exon': 0, 'Intron': 1}

reverse\_state\_mapping = {0: 'Exon', 1: 'Intron'}

# Sample training dataset (Observed DNA sequences and their corresponding states)

sequences = ['ATGCGT', 'CGTTAG', 'GGATCC', 'TACGTA']

states = [['Exon', 'Exon', 'Intron', 'Intron', 'Exon', 'Exon'],

['Intron', 'Intron', 'Exon', 'Exon', 'Intron', 'Exon'],

['Exon', 'Exon', 'Exon', 'Intron', 'Intron', 'Exon'],

['Intron', 'Intron', 'Exon', 'Exon', 'Exon', 'Exon']] # Convert sequences and states to numerical format

encoded\_sequences = [np.array([dna\_mapping[n] for n in seq]) for seq in sequences]

encoded\_states = [np.array([state\_mapping[s] for s in state]) for state in states]

# Flatten the sequences for training

X = np.concatenate(encoded\_sequences).reshape(-1, 1) lengths = [len(seq) for seq in encoded\_sequences]

y\_true = np.concatenate(encoded\_states) # Train Discrete HMM

model = hmm.MultinomialHMM(n\_components=2, n\_iter=100, tol=1e-4, random\_state=42)

model.fit(X, lengths)

# Predict hidden states using Viterbi Algorithm

logprob, y\_pred = model.decode(X, algorithm="viterbi") # Evaluate model accuracy

accuracy = accuracy\_score(y\_true, y\_pred) print(f"Model Accuracy: {accuracy:.2%}")

print("Classification Report:\n", classification\_report(y\_true, y\_pred, target\_names=['Exon', 'Intron']))

# Confusion Matrix

cm = confusion\_matrix(y\_true, y\_pred)

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Exon', 'Intron'], yticklabels=['Exon', 'Intron'])

plt.xlabel('Predicted') plt.ylabel('Actual') plt.title('Confusion Matrix') plt.show()

# Visualization of predicted gene regions plt.figure(figsize=(10, 2))

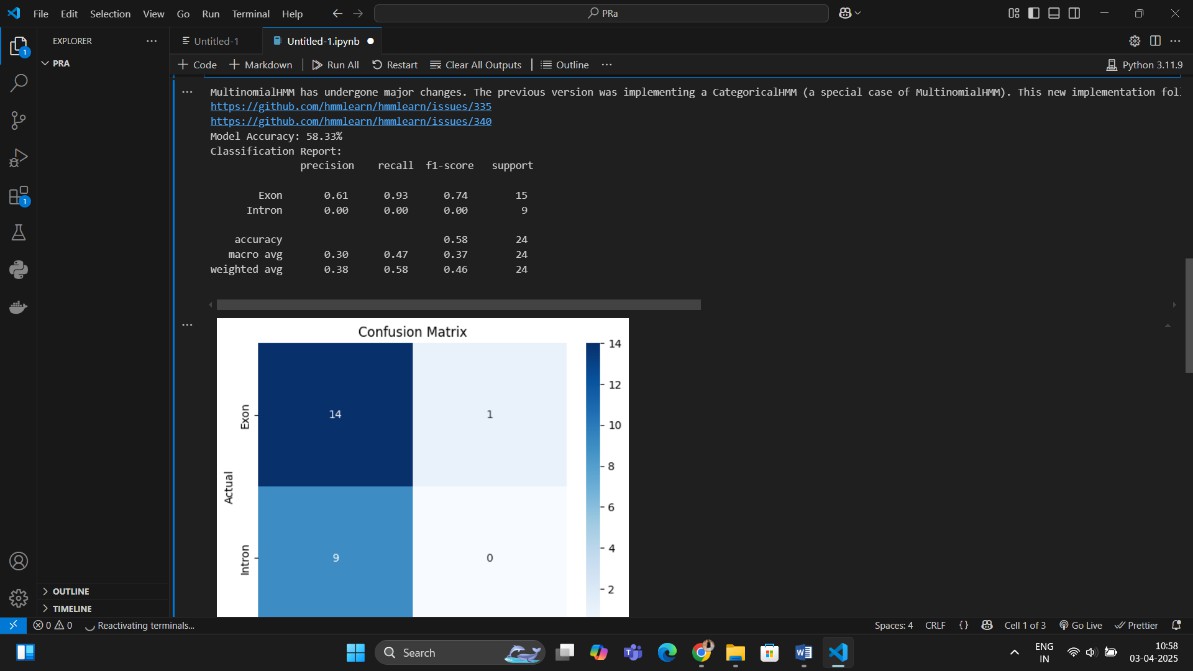
plt.plot(y\_pred, label='Predicted States', marker='o', linestyle='-', color='b') plt.plot(y\_true, label='True States', marker='x', linestyle='--', color='r') plt.xlabel('Position in Sequence')

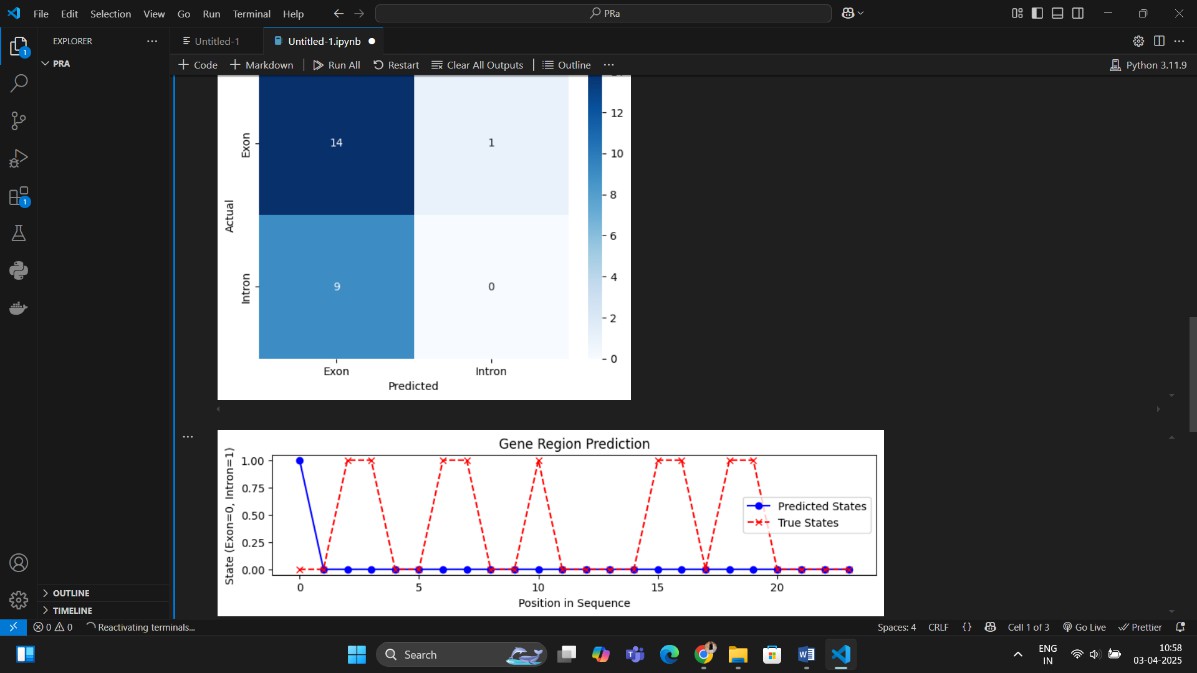
plt.ylabel('State (Exon=0, Intron=1)') plt.legend()

plt.title('Gene Region Prediction')

plt.show()

**Output:-**

****

****

**Expriment No.10**

**Code:-**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.mixture import GaussianMixture from sklearn.decomposition import PCA

from sklearn.metrics import adjusted\_rand\_score from sklearn.preprocessing import StandardScaler from tensorflow.keras.datasets import mnist

# Load MNIST dataset

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data() # Flatten images to 784-dimensional vectors

X\_train = X\_train.reshape(X\_train.shape[0], -1) X\_test = X\_test.reshape(X\_test.shape[0], -1)

# Normalize the data scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train) X\_test\_scaled = scaler.transform(X\_test)

# Reduce dimensionality for better clustering pca = PCA(n\_components=50)

X\_train\_pca = pca.fit\_transform(X\_train\_scaled) X\_test\_pca = pca.transform(X\_test\_scaled)

# Define and fit Gaussian Mixture Model (GMM) n\_components = 10 # Number of clusters (digits 0-9)

gmm = GaussianMixture(n\_components=n\_components, covariance\_type='full', random\_state=42)

gmm.fit(X\_train\_pca) # Predict cluster labels

train\_clusters = gmm.predict(X\_train\_pca) test\_clusters = gmm.predict(X\_test\_pca)

# Evaluate clustering performance using Adjusted Rand Index (ARI) ari\_score = adjusted\_rand\_score(y\_train, train\_clusters) print(f"Adjusted Rand Index (ARI) on training data: {ari\_score:.4f}") # Visualize the clusters using PCA (first two components)

def plot\_clusters(data, labels, title): plt.figure(figsize=(8, 6))

scatter = plt.scatter(data[:, 0], data[:, 1], c=labels, cmap='viridis', alpha=0.5) plt.colorbar(scatter, label='Cluster Index')

plt.title(title)

plt.xlabel('PCA Component 1')

plt.ylabel('PCA Component 2') plt.show()

plot\_clusters(X\_train\_pca, train\_clusters, 'GMM Clustering of Handwritten Digits (Training Data)')

# Visualize cluster means (reshaped to 28x28 images) def plot\_gmm\_means(gmm, pca):

mean\_images = pca.inverse\_transform(gmm.means\_).reshape(n\_components, 28, 28)

plt.figure(figsize=(10, 5))

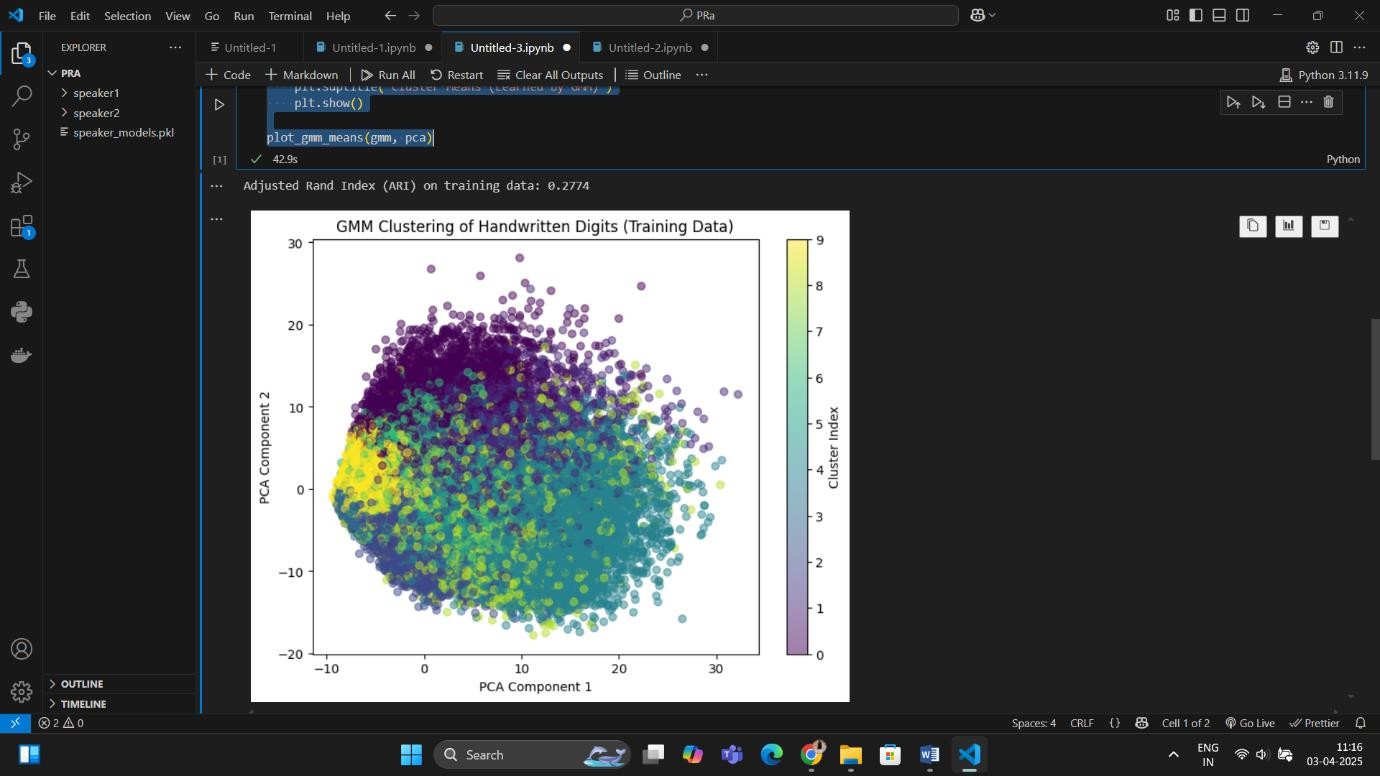
for i in range(n\_components):

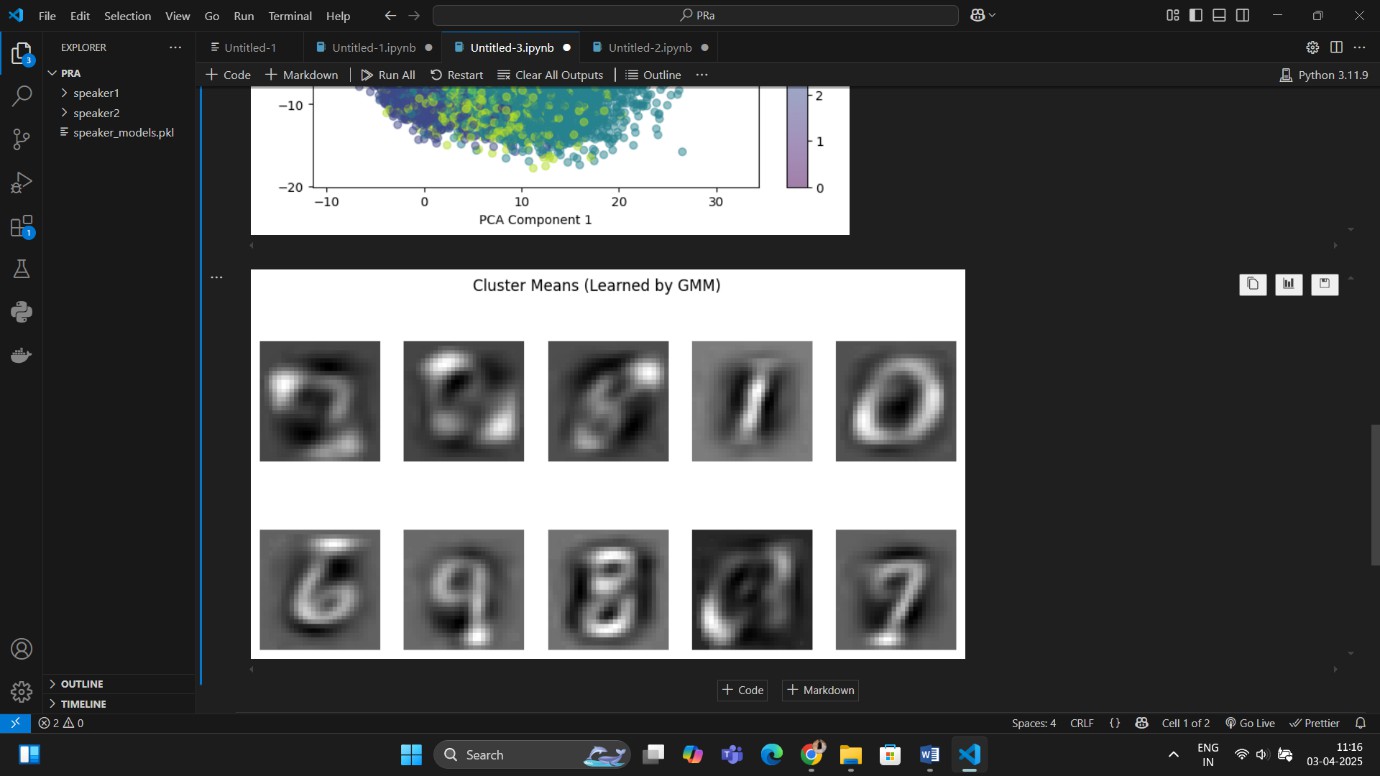
plt.subplot(2, 5, i + 1) plt.imshow(mean\_images[i], cmap='gray') plt.axis('off')

plt.suptitle('Cluster Means (Learned by GMM)') plt.show()

plot\_gmm\_means(gmm, pca)

**Output:-**



****

**Expriment No.11**

**Code:-**

import numpy as np import cv2

import os

# Define dataset directory dataset\_path = "shapes\_dataset" shapes = ["circle", "square", "triangle"]

num\_images = 100 # Number of images per shape

# Ensure directories exist for shape in shapes:

os.makedirs(os.path.join(dataset\_path, shape), exist\_ok=True)

# Function to draw shapes

def generate\_shape(shape, size=(64, 64)):

img = np.zeros(size, dtype=np.uint8) # Black background center = (size[0] // 2, size[1] // 2)

radius = 20

if shape == "circle":

cv2.circle(img, center, radius, 255, -1) elif shape == "square":

cv2.rectangle(img, (center[0] - radius, center[1] - radius), (center[0] + radius, center[1] + radius), 255, -1)

elif shape == "triangle":

pts = np.array([[center[0], center[1] - radius], [center[0] - radius, center[1] + radius],

[center[0] + radius, center[1] + radius]], np.int32) pts = pts.reshape((-1, 1, 2))

cv2.fillPoly(img, [pts], 255)

return img

# Generate and save images for shape in shapes:

for i in range(num\_images): img = generate\_shape(shape)

img\_path = os.path.join(dataset\_path, shape, f"{shape}\_{i}.png") cv2.imwrite(img\_path, img)

print("✅ Synthetic dataset generated successfully!") import os

import numpy as np import cv2

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KernelDensity, KNeighborsClassifier from sklearn.metrics import accuracy\_score, confusion\_matrix import matplotlib.pyplot as plt

import seaborn as sns

# Load dataset

dataset\_path = "shapes\_dataset" shapes = ["circle", "square", "triangle"] img\_size = (64, 64)

X = [] # Feature vectors y = [] # Labels

# Load images and extract features

for label, shape in enumerate(shapes): shape\_path = os.path.join(dataset\_path, shape) for img\_name in os.listdir(shape\_path):

img\_path = os.path.join(shape\_path, img\_name)

img = cv2.imread(img\_path, cv2.IMREAD\_GRAYSCALE) # Load as grayscale img = cv2.resize(img, img\_size) # Resize (optional)

X.append(img.flatten()) # Flatten image to 1D feature vector y.append(label)

# Convert to NumPy arrays X = np.array(X)

y = np.array(y)

# Split dataset (80% train, 20% test)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# --- Parzen-Window Classifier ---

kde\_models = {} # Dictionary to store KDE models per class bandwidth = 0.2 # Smoothing parameter

for label in np.unique(y\_train):

kde = KernelDensity(kernel="gaussian", bandwidth=bandwidth) kde.fit(X\_train[y\_train == label])

kde\_models[label] = kde

# Classification using Parzen-Window method y\_pred\_parzen = []

for x in X\_test:

probs = {label: kde\_models[label].score\_samples([x]) for label in kde\_models}

y\_pred\_parzen.append(max(probs, key=probs.get)) # Compute accuracy

accuracy\_parzen = accuracy\_score(y\_test, y\_pred\_parzen)

print(f"🎯 Parzen-Window Classification Accuracy: {accuracy\_parzen:.2f}")

# --- K-Nearest Neighbors Classifier ---

knn = KNeighborsClassifier(n\_neighbors=5) knn.fit(X\_train, y\_train)

y\_pred\_knn = knn.predict(X\_test) # Compute accuracy

accuracy\_knn = accuracy\_score(y\_test, y\_pred\_knn) print(f"🎯 KNN Classification Accuracy: {accuracy\_knn:.2f}") # --- Confusion Matrices ---

fig, axes = plt.subplots(1, 2, figsize=(12, 5)) # Parzen-Window Confusion Matrix

sns.heatmap(confusion\_matrix(y\_test, y\_pred\_parzen), annot=True, fmt="d", cmap="Blues",

xticklabels=shapes, yticklabels=shapes, ax=axes[0]) axes[0].set\_title("Parzen-Window Confusion Matrix") axes[0].set\_xlabel("Predicted") axes[0].set\_ylabel("Actual")

# KNN Confusion Matrix

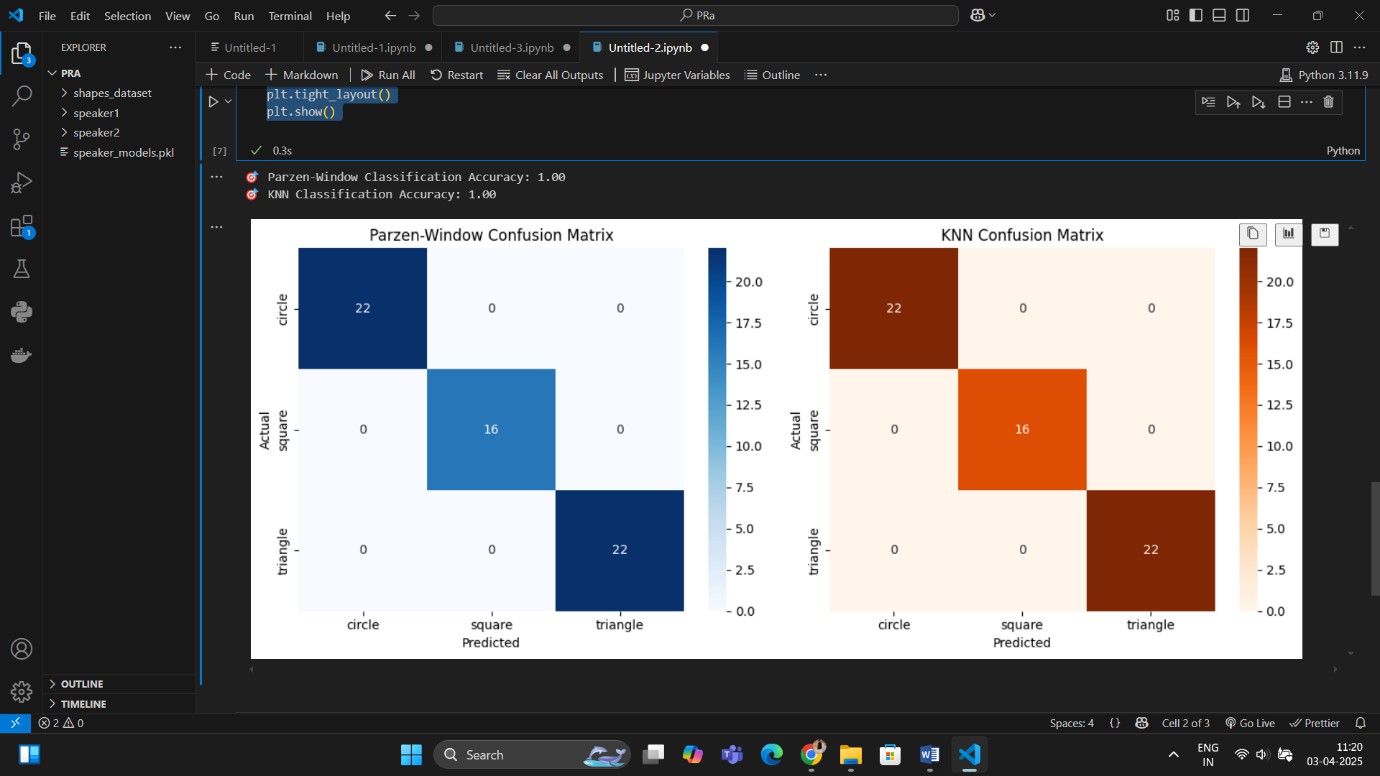
sns.heatmap(confusion\_matrix(y\_test, y\_pred\_knn), annot=True, fmt="d", cmap="Oranges",

xticklabels=shapes, yticklabels=shapes, ax=axes[1]) axes[1].set\_title("KNN Confusion Matrix") axes[1].set\_xlabel("Predicted") axes[1].set\_ylabel("Actual")

plt.tight\_layout()

plt.show()

**Output:-**

****

**Expriment No.12**

**Code:-**

import numpy as np import cv2

import os

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split, GridSearchCV from sklearn.neighbors import KNeighborsClassifier

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix, make\_scorer

import seaborn as sns

def create\_shape\_image(shape, size=(50, 50)): img = np.zeros(size, dtype=np.uint8)

if shape == "circle":

cv2.circle(img, (25, 25), 20, 255, -1) elif shape == "square":

cv2.rectangle(img, (10, 10), (40, 40), 255, -1) elif shape == "triangle":

points = np.array([[25, 5], [5, 45], [45, 45]], np.int32)

cv2.fillPoly(img, [points], 255) return img

def generate\_dataset(samples\_per\_class=100): shapes = ["circle", "square", "triangle"] images, labels = [], []

for shape in shapes:

for \_ in range(samples\_per\_class): img = create\_shape\_image(shape) images.append(img.flatten()) labels.append(shape)

return np.array(images, dtype=np.float32), np.array(labels) # Generate synthetic dataset

X, y = generate\_dataset(samples\_per\_class=200) # Encode labels

label\_encoder = LabelEncoder()

y = label\_encoder.fit\_transform(y) # Normalize features

scaler = StandardScaler() X = scaler.fit\_transform(X)

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)

# Custom scoring function to handle potential errors scorer = make\_scorer(accuracy\_score)

# Hyperparameter tuning using GridSearchCV

param\_grid = {'n\_neighbors': [3, 5, 7, 9], 'metric': ['euclidean', 'manhattan', 'minkowski']}

knn = KNeighborsClassifier()

grid\_search = GridSearchCV(knn, param\_grid, cv=5, scoring=scorer, error\_score='raise')

grid\_search.fit(X\_train, y\_train) # Best model selection

best\_knn = grid\_search.best\_estimator\_

y\_pred = best\_knn.predict(X\_test) # Model evaluation

accuracy = accuracy\_score(y\_test, y\_pred) print(f'Best KNN Model: {grid\_search.best\_params\_}') print(f'Accuracy: {accuracy:.2f}')

print('Classification Report:\n', classification\_report(y\_test, y\_pred)) # Confusion matrix visualization

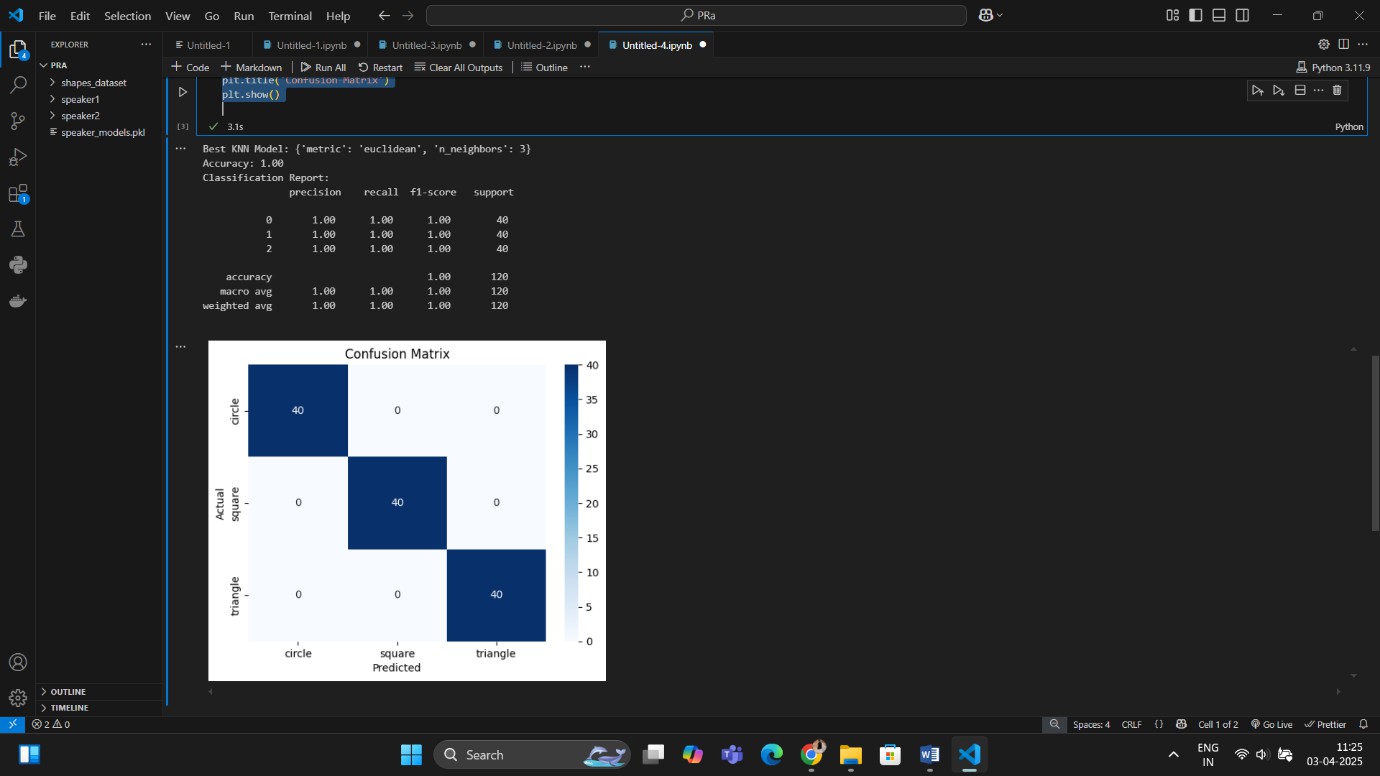
conf\_matrix = confusion\_matrix(y\_test, y\_pred)

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=label\_encoder.classes\_, yticklabels=label\_encoder.classes\_)

plt.xlabel('Predicted') plt.ylabel('Actual') plt.title('Confusion Matrix')

plt.show()

**Output:-**

****