

**PATTERN PROCESSING USING AI**

**PRACTICAL FILE**

**Name: Chirag Gupta**

**Roll No.: 2019UCO1689**

**Subject Code: COCSE60**

1. **Write a python program to implement a simple Chatbot.**

Code:

import random

responses = {

"hello": ["Hi there!", "Hello!", "Hey!"],

"how are you": ["I'm doing well, thank you.", "Not too bad, thanks for asking.", "I'm just fine."],

"what's your name": ["My name is Chatbot.", "I go by Chatbot.", "You can call me Chatbot."],

"default": ["I'm sorry, I didn't understand what you said.", "Can you please rephrase that?", "I'm not sure what you mean."],

}

def get\_response(user\_input):

user\_input = user\_input.lower().strip()

if user\_input in responses:

return random.choice(responses[user\_input])

else:

return random.choice(responses["default"])

def run\_chatbot():

print("Hi, I'm Chatbot. How can I help you today?")

while True:

user\_input = input("You: ")

bot\_response = get\_response(user\_input)

print("Chatbot: " + bot\_response)

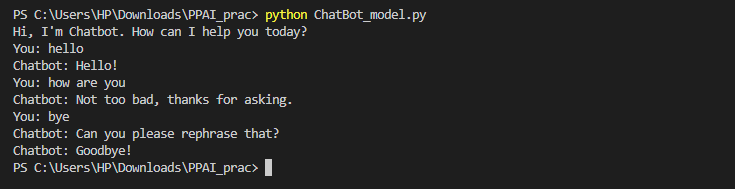
if user\_input.lower().strip() == "bye":

print("Chatbot: Goodbye!")

break

run\_chatbot()

Output:



1. **Write a program to implement k-means clustering from scratch.**

Code:

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

from numpy.random import uniform

from sklearn.datasets import make\_blobs

import seaborn as sns

import random

def euclidean(point, data):

"""

Euclidean distance between point & data.

Point has dimensions (m,), data has dimensions (n,m), and output will be of size (n,).

"""

return np.sqrt(np.sum((point - data)\*\*2, axis=1))

class KMeans:

def \_\_init\_\_(self, n\_clusters=8, max\_iter=300):

self.n\_clusters = n\_clusters

self.max\_iter = max\_iter

def fit(self, X\_train):

self.centroids = [random.choice(X\_train)]

for \_ in range(self.n\_clusters-1):

# Calculate distances from points to the centroids

dists = np.sum([euclidean(centroid, X\_train) for centroid in self.centroids], axis=0)

# Normalize the distances

dists /= np.sum(dists)

# Choose remaining points based on their distances

new\_centroid\_idx, = np.random.choice(range(len(X\_train)), size=1, p=dists)

self.centroids += [X\_train[new\_centroid\_idx]]

iteration = 0

prev\_centroids = None

while np.not\_equal(self.centroids, prev\_centroids).any() and iteration < self.max\_iter:

# Sort each datapoint, assigning to nearest centroid

sorted\_points = [[] for \_ in range(self.n\_clusters)]

for x in X\_train:

dists = euclidean(x, self.centroids)

centroid\_idx = np.argmin(dists)

sorted\_points[centroid\_idx].append(x)

prev\_centroids = self.centroids

self.centroids = [np.mean(cluster, axis=0) for cluster in sorted\_points]

for i, centroid in enumerate(self.centroids):

if np.isnan(centroid).any():

self.centroids[i] = prev\_centroids[i]

iteration += 1

def evaluate(self, X):

centroids = []

centroid\_idxs = []

for x in X:

dists = euclidean(x, self.centroids)

centroid\_idx = np.argmin(dists)

centroids.append(self.centroids[centroid\_idx])

centroid\_idxs.append(centroid\_idx)

return centroids, centroid\_idxs

centers = 5

X\_train, true\_labels = make\_blobs(n\_samples=100, centers=centers, random\_state=42)

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

# Fit centroids to dataset

kmeans = KMeans(n\_clusters=centers)

kmeans.fit(X\_train)

class\_centers, classification = kmeans.evaluate(X\_train)

sns.scatterplot(x=[X[0] for X in X\_train],

y=[X[1] for X in X\_train],

hue=true\_labels,

style=classification,

palette="deep",

legend=None

)

plt.plot([x for x, \_ in kmeans.centroids],

[y for \_, y in kmeans.centroids],

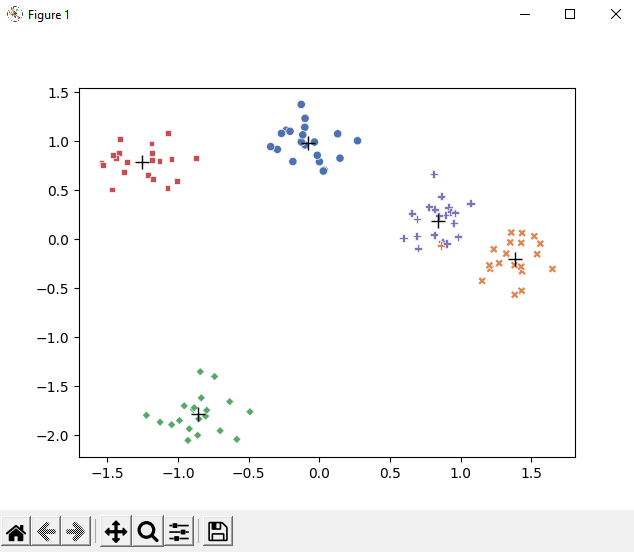
'k+',

markersize=10,

)

plt.show()

Output:



1. **Generating samples of Gaussian (normal) distributions and plotting them for visualisation**

Code:

import numpy as np

import matplotlib.pyplot as plt

mean1, mean2, mean3 = 0, 5, -5

std1, std2, std3 = 1, 2, 3

data1 = np.random.normal(mean1, std1, 1000)

data2 = np.random.normal(mean2, std2, 1000)

data3 = np.random.normal(mean3, std3, 1000)

fig, ax = plt.subplots()

ax.hist(data1, bins=50, alpha=0.5, label='Data 1')

ax.hist(data2, bins=50, alpha=0.5, label='Data 2')

ax.hist(data3, bins=50, alpha=0.5, label='Data 3')

ax.legend(loc='upper right')

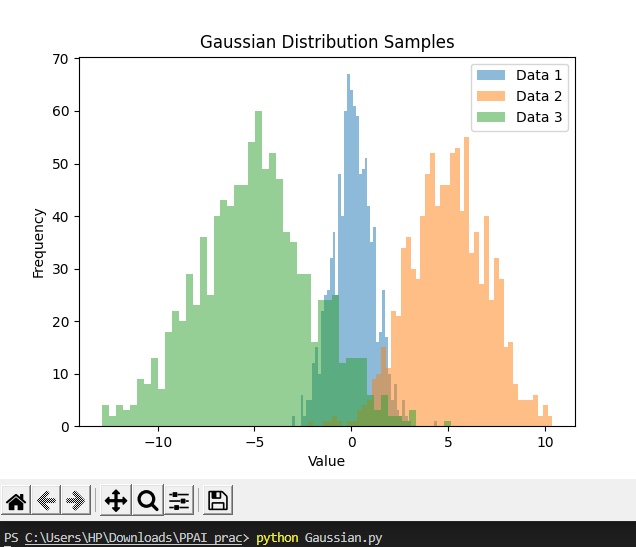
ax.set\_title('Gaussian Distribution Samples')

ax.set\_xlabel('Value')

ax.set\_ylabel('Frequency')

plt.show()

Output:



1. **Implement Decision Tree algorithms.**

Code:

import numpy as np

from sklearn.datasets import load\_iris

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

class DecisionTree:

def \_\_init\_\_(self, max\_depth=5, min\_samples\_split=2):

self.max\_depth = max\_depth

self.min\_samples\_split = min\_samples\_split

def fit(self, X, y):

class Node:

def \_\_init\_\_(self, feature\_idx=None, threshold=None, left=None, right=None, is\_leaf=False, label=None):

self.feature\_idx = feature\_idx

self.threshold = threshold

self.left = left

self.right = right

self.is\_leaf = is\_leaf

self.label = label

def entropy(y):

\_, counts = np.unique(y, return\_counts=True)

p = counts / len(y)

return -np.sum(p \* np.log2(p))

def info\_gain(X, y, feature\_idx, threshold):

left\_idx = X[:, feature\_idx] < threshold

left\_y = y[left\_idx]

right\_y = y[~left\_idx]

p\_left = len(left\_y) / len(y)

p\_right = 1 - p\_left

ig = entropy(y) - p\_left \* entropy(left\_y) - p\_right \* entropy(right\_y)

return ig

def split(X, y, depth):

if depth >= self.max\_depth or len(X) < self.min\_samples\_split or len(np.unique(y)) == 1:

label = np.bincount(y).argmax()

return Node(is\_leaf=True, label=label)

best\_feature\_idx, best\_threshold, best\_ig = None, None, 0

for feature\_idx in range(X.shape[1]):

thresholds = np.unique(X[:, feature\_idx])

for threshold in thresholds:

ig = info\_gain(X, y, feature\_idx, threshold)

if ig > best\_ig:

best\_feature\_idx, best\_threshold, best\_ig = feature\_idx, threshold, ig

left\_idx = X[:, best\_feature\_idx] < best\_threshold

right\_idx = ~left\_idx

left\_node = split(X[left\_idx], y[left\_idx], depth+1)

right\_node = split(X[right\_idx], y[right\_idx], depth+1)

return Node(feature\_idx=best\_feature\_idx, threshold=best\_threshold, left=left\_node, right=right\_node)

self.root = split(X, y, depth=0)

def predict(self, X):

def traverse(node, x):

if node.is\_leaf:

return node.label

if x[node.feature\_idx] < node.threshold:

return traverse(node.left, x)

else:

return traverse(node.right, x)

y\_pred = np.array([traverse(self.root, x) for x in X])

return y\_pred

iris = load\_iris()

X, y = iris.data, iris.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

dt = DecisionTree()

dt.fit(X\_train, y\_train)

y\_pred = dt.predict(X\_test)

accuracy = np.sum(y\_pred == y\_test) / len(y\_test)

print("Accuracy:", accuracy)

Output:



1. **Implement SVM.**

Code:

import numpy as np

from sklearn.datasets import load\_iris

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

import matplotlib.pyplot as plt

class SVM:

def \_\_init\_\_(self, learning\_rate=0.001, regularization=0.01, num\_iterations=1000):

self.lr = learning\_rate

self.reg = regularization

self.num\_iters = num\_iterations

self.w = None

self.b = None

def fit(self, X, y):

self.w = np.zeros(X.shape[1])

self.b = 0

for i in range(self.num\_iters):

margins = y \* (np.dot(X, self.w) + self.b)

hinge\_loss = np.maximum(0, 1 - margins)

dw = self.reg \* self.w - np.dot(X.T, y \* (hinge\_loss > 0))

db = -np.sum(y \* (hinge\_loss > 0))

self.w -= self.lr \* dw

self.b -= self.lr \* db

def predict(self, X):

scores = np.dot(X, self.w) + self.b

return np.sign(scores)

iris = load\_iris()

X = iris.data[:, :2] # Select only the first two features

y = np.where(iris.target == 0, -1, 1) # Convert labels to -1 or 1

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

svm = SVM()

svm.fit(X\_train, y\_train)

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.02),

np.arange(y\_min, y\_max, 0.02))

Z = svm.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.8)

plt.scatter(X\_train[:, 0], X\_train[:, 1], c=y\_train, cmap=plt.cm.Paired)

plt.xlabel('Sepal length')

plt.ylabel('Sepal width')

plt.xlim(xx.min(), xx.max())

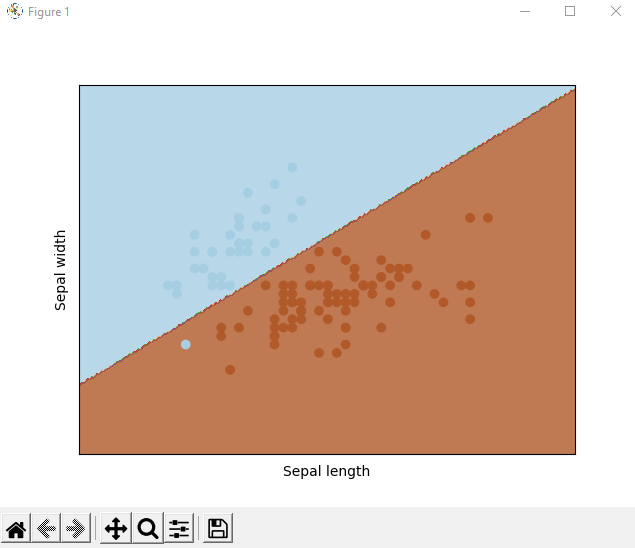
plt.ylim(yy.min(), yy.max())

plt.xticks(())

plt.yticks(())

plt.show()

Output:



1. **Implement Principal component analysis and use it for unsupervised learning**

Code:

import numpy as np

def pca(X, num\_components):

mean\_X = np.mean(X, axis=0)

X\_centered = X - mean\_X

cov\_X = np.cov(X\_centered, rowvar=False)

eigenvalues, eigenvectors = np.linalg.eigh(cov\_X)

sorted\_indices = np.argsort(eigenvalues)[::-1]

sorted\_eigenvalues = eigenvalues[sorted\_indices]

sorted\_eigenvectors = eigenvectors[:, sorted\_indices]

top\_eigenvectors = sorted\_eigenvectors[:, :num\_components]

X\_transformed = np.dot(X\_centered, top\_eigenvectors)

return X\_transformed, top\_eigenvectors

from sklearn.datasets import load\_iris

iris = load\_iris()

X = iris.data

X\_transformed, top\_eigenvectors = pca(X, num\_components=2)

from sklearn.cluster import KMeans

kmeans = KMeans(n\_clusters=3, random\_state=42)

kmeans.fit(X\_transformed)

labels = kmeans.labels\_

import matplotlib.pyplot as plt

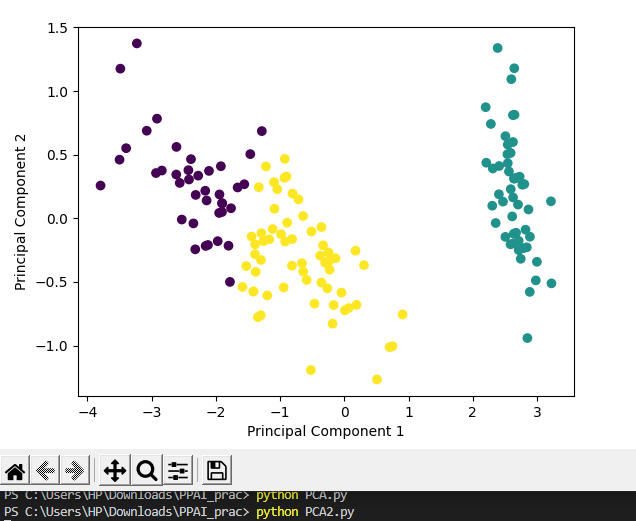
plt.scatter(X\_transformed[:, 0], X\_transformed[:, 1], c=labels)

plt.xlabel('Principal Component 1')

plt.ylabel('Principal Component 2')

plt.show()

Output:



1. **Implement Maximum-Likelihood estimation.**

Code:

import numpy as np

def likelihood(x, mu, sigma):

n = len(x)

log\_likelihood = -n/2\*np.log(2\*np.pi\*sigma\*\*2) - np.sum((x-mu)\*\*2)/(2\*sigma\*\*2)

return log\_likelihood

def d\_likelihood\_mu(x, mu, sigma):

n = len(x)

d\_log\_likelihood\_mu = np.sum((mu-x)/(sigma\*\*2))

return d\_log\_likelihood\_mu

def d\_likelihood\_sigma(x, mu, sigma):

n = len(x)

d\_log\_likelihood\_sigma = -n/(2\*sigma\*\*2) + np.sum((x-mu)\*\*2)/(2\*sigma\*\*4)

return d\_log\_likelihood\_sigma

def maximum\_likelihood\_estimation(x):

mu = np.mean(x)

sigma = np.std(x)

alpha = 0.1

epsilon = 1e-5

while True:

d\_mu = d\_likelihood\_mu(x, mu, sigma)

d\_sigma = d\_likelihood\_sigma(x, mu, sigma)

mu -= alpha\*d\_mu

sigma -= alpha\*d\_sigma

if np.abs(d\_mu) < epsilon and np.abs(d\_sigma) < epsilon:

break

return mu, sigma

np.random.seed(123)

x = np.random.normal(loc=5, scale=2, size=100)

mu, sigma = maximum\_likelihood\_estimation(x)

print('mu:', mu)

print('sigma:', sigma)

Output:



1. **Implement agglomerative Hierarchical clustering.**

Code:

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

class AgglomerativeClustering:

def \_\_init\_\_(self, n\_clusters):

self.n\_clusters = n\_clusters

def fit(self, X):

clusters = [[i] for i in range(X.shape[0])]

dist\_matrix = np.zeros((X.shape[0], X.shape[0]))

for i in range(X.shape[0]):

for j in range(i+1, X.shape[0]):

dist\_matrix[i,j] = np.linalg.norm(X[i,:] - X[j,:])

dendrogram = np.zeros((X.shape[0]-1, 4))

for i in range(X.shape[0]-1):

min\_dist = np.inf

for j in range(len(clusters)):

for k in range(j+1, len(clusters)):

dist = np.min(dist\_matrix[clusters[j],:][:,clusters[k]])

if dist < min\_dist:

min\_dist = dist

merge\_clusters = (j,k)

dendrogram[i,0] = merge\_clusters[0]

dendrogram[i,1] = merge\_clusters[1]

dendrogram[i,2] = min\_dist

dendrogram[i,3] = len(clusters[merge\_clusters[0]]) + len(clusters[merge\_clusters[1]])

clusters[merge\_clusters[0]] += clusters[merge\_clusters[1]]

del clusters[merge\_clusters[1]]

# Update the distance matrix

for j in range(len(clusters)-1):

for k in range(j+1, len(clusters)):

min\_dists = []

for l in clusters[j]:

for m in clusters[k]:

min\_dists.append(dist\_matrix[l,m])

dist\_matrix[j,k] = min(min\_dists)

dist\_matrix[k,j] = dist\_matrix[j,k]

self.labels\_ = np.zeros(X.shape[0], dtype=np.int32)

for i in range(X.shape[0]):

for j in range(dendrogram.shape[0]):

if dendrogram[j,0] <= i < dendrogram[j,3]:

self.labels\_[i] = j + X.shape[0] - self.n\_clusters

break

self.dendrogram = dendrogram

def plot\_dendrogram(self):

# Plot the dendrogram

plt.figure(figsize=(10,6))

plt.title('Dendrogram')

plt.xlabel('Observations')

plt.ylabel('Distance')

plt.xticks([])

plt.yticks([])

for i in range(self.dendrogram.shape[0]):

x1 = self.dendrogram[i,0]

x2 = self.dendrogram[i,1]

y1 = self.dendrogram[i,2]

y2 = self.dendrogram[i,3]

plt.plot([x1,x1,x2,x2], [y1,y2,y2,y1], 'k-')

plt.show()

iris = load\_iris()

X = iris.data

n\_clusters = 3

model = AgglomerativeClustering(n\_clusters)

model.fit(X)

model.plot\_dendrogram()

print(model.labels\_)

Output:

