PATTERN PROCESSING USING AI

PRACTICAL FILE

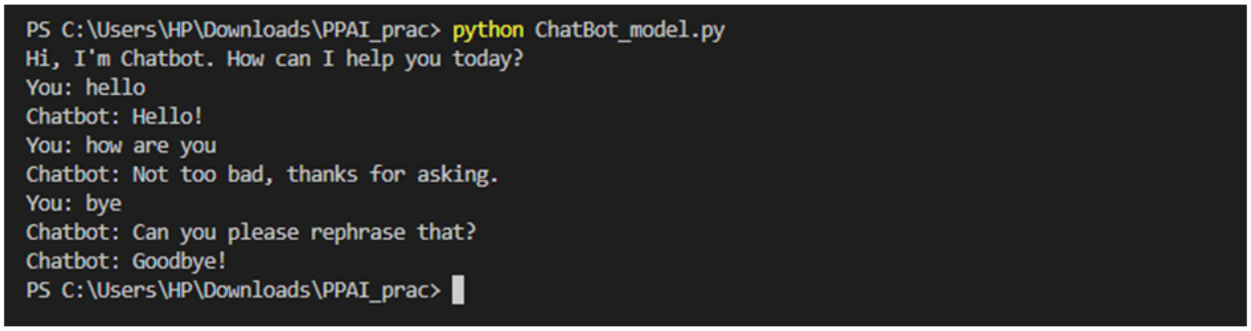
Name	:	Ankit Kumar Yadav
Roll No.	:	2019UCO1712
Subject Code	:	COCSE60
Branch	:	Computer Engineering

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



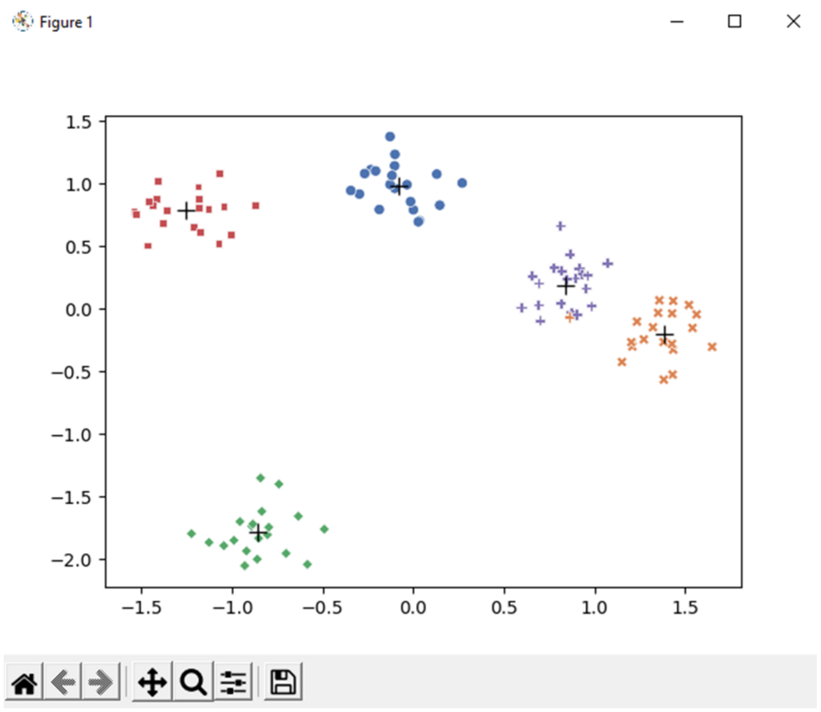
```
PS C:\Users\HP\Downloads\PPAI_prac> python ChatBot_model.py
Hi, I'm Chatbot. How can I help you today?
You: hello
Chatbot: Hello!
You: how are you
Chatbot: Not too bad, thanks for asking.
You: bye
Chatbot: Can you please rephrase that?
Chatbot: Goodbye!
PS C:\Users\HP\Downloads\PPAI_prac>
```

2. 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

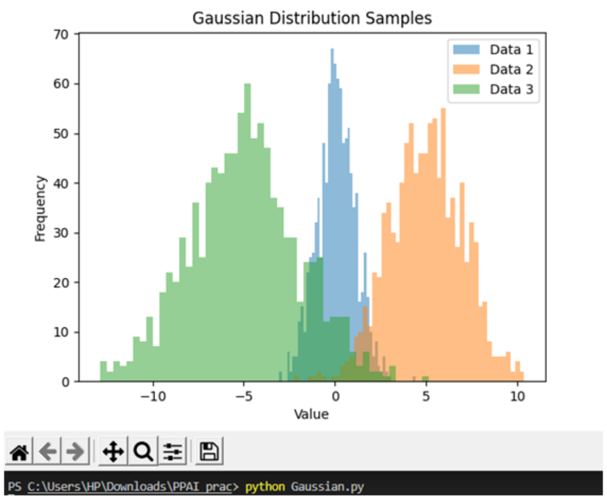


3. Generating samples of Gaussian (normal) distributions and plotting them for visualization.

Code

```python
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



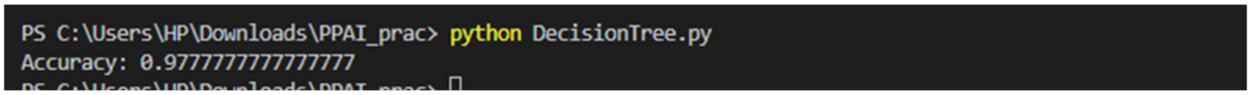
```
PS C:\Users\HP\Downloads\PPAI_prac> python Gaussian.py
```

4. 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



```
PS C:\Users\HP\Downloads\PPAI_prac> python DecisionTree.py
Accuracy: 0.9777777777777777
PS C:\Users\HP\Downloads\PPAI_prac>
```

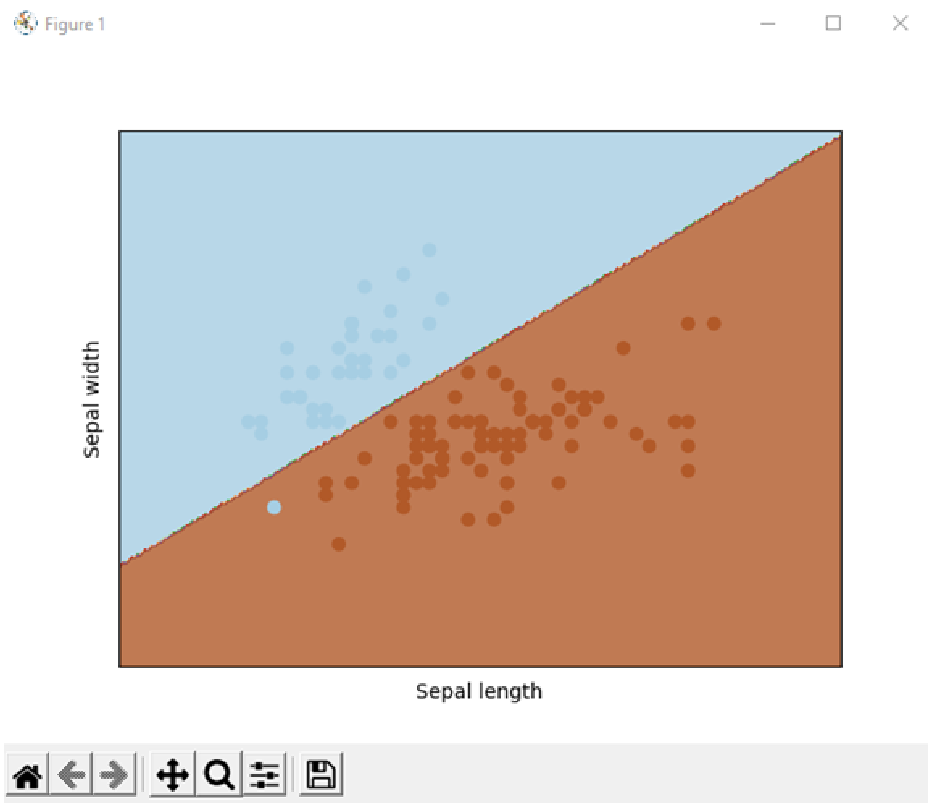
5. Implement SVM.

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

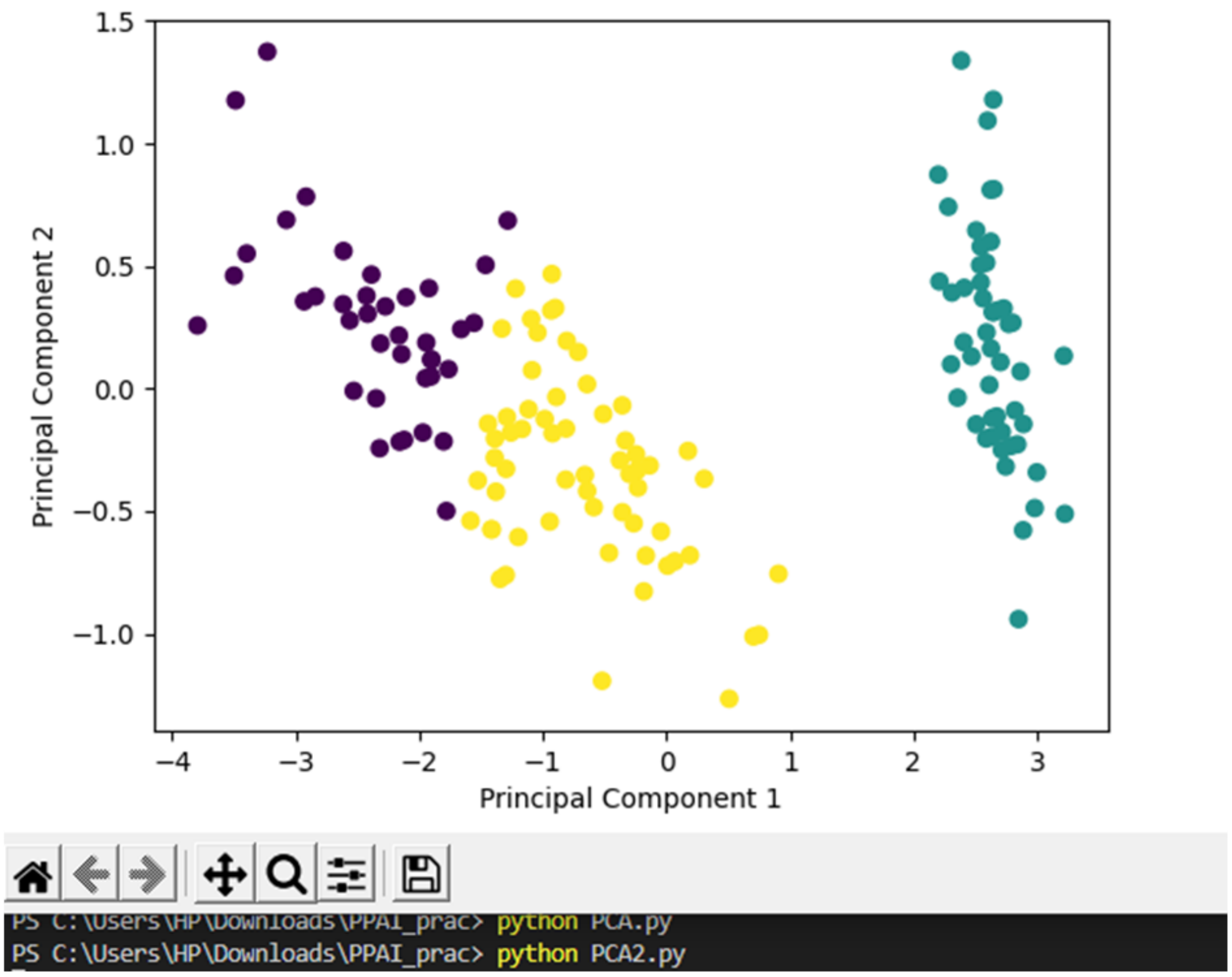


6. Implement Principal component analysis and use it for unsupervised learning

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



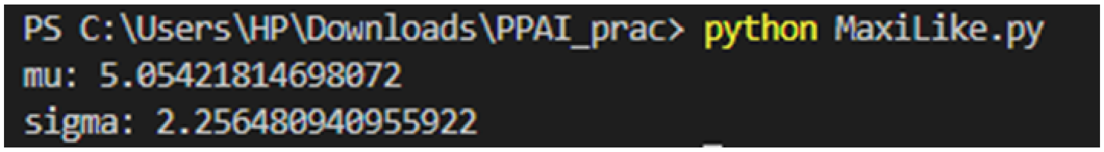
```
PS C:\Users\HP\Downloads\PPAI_prac> python PCA.py
PS C:\Users\HP\Downloads\PPAI_prac> python PCA2.py
```

7. 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



```
PS C:\Users\HP\Downloads\PPAI_prac> python MaxiLike.py
mu: 5.05421814698072
sigma: 2.256480940955922
```

8. Implement agglomerative Hierarchical clustering.

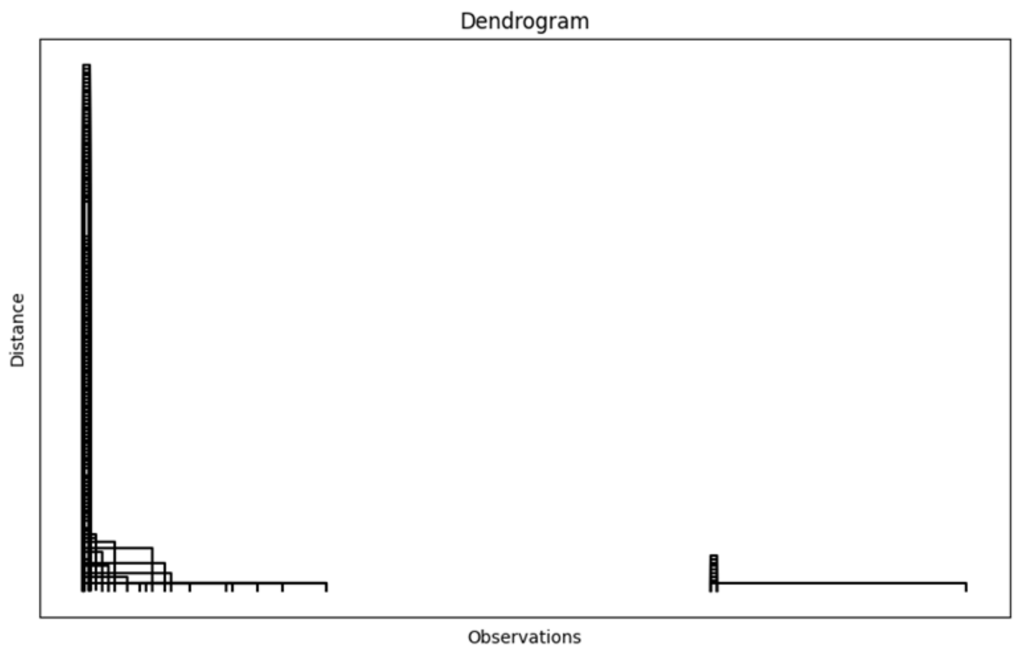
Code

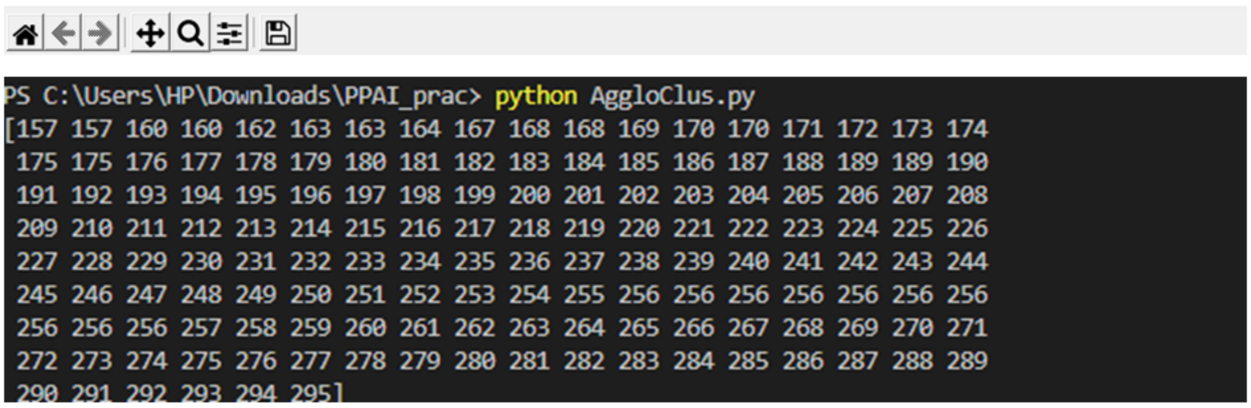
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris

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





```
PS C:\Users\HP\Downloads\PPAI_prac> python AggloClus.py
[157 157 160 160 162 163 163 164 167 168 168 169 170 170 171 172 173 174
 175 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 189 190
 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208
 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226
 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244
 245 246 247 248 249 250 251 252 253 254 255 256 256 256 256 256 256 256
 256 256 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271
 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289
 290 291 292 293 294 295]
```