//Program

```
from itertools import combinations
def get_frequent_itemsets(transactions, min_support):
    item_counts = {}
    for transaction in transactions:
        for itemset_size in range(1, len(transaction) + 1):
            for itemset in combinations(transaction, itemset_size):
                itemset = frozenset(itemset)
                item_counts[itemset] = item_counts.get(itemset, 0) + 1
    return {itemset: count for itemset, count in item_counts.items() if count >= min_support}
def generate_association_rules(frequent_itemsets, min_confidence):
    rules = []
    for itemset, support in frequent_itemsets.items():
        if len(itemset) > 1:
            for i in range(1, len(itemset)):
                for antecedent in combinations(itemset, i):
                    antecedent = frozenset(antecedent)
                    consequent = itemset - antecedent
                    if frequent_itemsets[antecedent] > 0:
                        confidence = support / frequent_itemsets[antecedent]
                        if confidence >= min_confidence:
                            rules.append((antecedent, consequent, confidence))
    return rules
if __name__ == "__main__":
    transactions = [
        ["apple", "banana", "cherry"],
        ["apple", "banana"],
        ["apple", "cherry"],
        ["banana", "cherry", "apple"],
        ["banana"]
    ]
    min_support = 2
    min_confidence = 0.5
    frequent_itemsets = get_frequent_itemsets(transactions, min_support)
    print("Frequent Itemsets:")
    for itemset, support in frequent_itemsets.items():
        print(f"Itemset: {list(itemset)}, Support: {support}")

    association_rules = generate_association_rules(frequent_itemsets, min_confidence)

    print("\nAssociation Rules:")
    for antecedent, consequent, confidence in association_rules:
```

```
print(f"Rule: {list(antecedent)} -> {list(consequent)}, Confidence: {confidence:.2f}")
```

Output

```
C:\Users\Asus\Desktop\Data Warehouse and Data Mining>python -u "c:\Users\Asus\Desktop\Data Warehouse and Data Mining\Apriori.py"
Frequent Itemsets:
Itemset: ['apple'], Support: 4
Itemset: ['banana'], Support: 4
Itemset: ['cherry'], Support: 3
Itemset: ['apple', 'banana'], Support: 3
Itemset: ['apple', 'cherry'], Support: 3
Itemset: ['banana', 'cherry'], Support: 2
Itemset: ['apple', 'banana', 'cherry'], Support: 2

Association Rules:
Rule: ['apple'] -> ['banana'], Confidence: 0.75
Rule: ['banana'] -> ['apple'], Confidence: 0.75
Rule: ['apple'] -> ['cherry'], Confidence: 0.75
Rule: ['cherry'] -> ['apple'], Confidence: 1.00
Rule: ['banana'] -> ['cherry'], Confidence: 0.50
Rule: ['cherry'] -> ['banana'], Confidence: 0.67
Rule: ['apple'] -> ['banana', 'cherry'], Confidence: 0.50
Rule: ['banana'] -> ['apple', 'cherry'], Confidence: 0.50
Rule: ['cherry'] -> ['apple', 'banana'], Confidence: 0.67
Rule: ['apple', 'banana'] -> ['cherry'], Confidence: 0.67
Rule: ['apple', 'cherry'] -> ['banana'], Confidence: 0.67
Rule: ['banana', 'cherry'] -> ['apple'], Confidence: 1.00
```

Program

```
class FPNode:
    def __init__(self, item, count, parent):
        self.item, self.count, self.parent = item, count, parent
        self.children, self.next = {}, None
    def increment(self, count):
        self.count += count
def build_tree(transactions, min_support):
    header_table = {}
    for transaction in transactions:
        for item in transaction:
            header_table[item] = header_table.get(item, 0) + 1
    header_table = {k: [v, None] for k, v in header_table.items() if v >= min_support}
    if not header_table: return None, None
    root = FPNode(None, None, None)
    for transaction in transactions:
        transaction = sorted([item for item in transaction if item in header_table], key=lambda x:
header_table[x][0], reverse=True)
        current_node = root
        for item in transaction:
            current_node = update_tree(item, current_node, header_table)
    return root, header_table
def update_tree(item, node, header_table):
    if item in node.children:
        node.children[item].increment(1)
    else:
        node.children[item] = FPNode(item, 1, node)
        update_header(item, node.children[item], header_table)
    return node.children[item]
def update_header(item, target_node, header_table):
    if not header_table[item][1]:
        header_table[item][1] = target_node
    else:
        current = header_table[item][1]
        while current.next: current = current.next
        current.next = target_node
def ascend_fp_tree(node, path):
    if node.parent:
        path.append(node.item)
        ascend_fp_tree(node.parent, path)

def find_frequent_itemsets(tree, header_table, min_support, prefix, frequent_itemsets):
```

```
    for item, _ in sorted(header_table.items(), key=lambda x: x[1][0]):
        new_prefix, support = prefix + [item], header_table[item][0]
        frequent_itemsets.append((new_prefix, support))
        cond_tree, cond_header = build_conditional_tree(item, tree, header_table, min_support)
        if cond_header:
            find_frequent_itemsets(cond_tree, cond_header, min_support, new_prefix, frequent_itemsets)
def build_conditional_tree(item, tree, header_table, min_support):
    paths, node = [], header_table[item][1]
    while node:
        path = []
        ascend_fp_tree(node, path)
        if len(path) > 1: paths.append(path[1:])
        node = node.next
    return build_tree(paths, min_support)
def fp_growth(transactions, min_support):
    tree, header_table = build_tree(transactions, min_support)
    frequent_itemsets = []
    find_frequent_itemsets(tree, header_table, min_support, [], frequent_itemsets)
    return frequent_itemsets
if __name__ == "__main__":
    transactions = [["A", "B", "C", "D"], ["A", "C", "D", "E"], ["A", "D", "E"], ["B", "D"], ["B", "C", "E"]]
    frequent_itemsets = fp_growth(transactions, 2)
    for itemset, support in frequent_itemsets:
        print(f"Itemset: {itemset}, Support: {support}")
```

Output

```
(env) C:\Users\Asus\Desktop\Data Warehouse and Data Mining>python -u "c:\Users\Asus\Desktop\Data Warehouse and Data Mining\Fp-growth.py"
Itemset: ['A'], Support: 3
Itemset: ['B'], Support: 3
Itemset: ['B', 'D'], Support: 2
Itemset: ['C'], Support: 3
Itemset: ['C', 'B'], Support: 2
Itemset: ['C', 'A'], Support: 2
Itemset: ['C', 'D'], Support: 2
Itemset: ['C', 'D', 'A'], Support: 2
Itemset: ['E'], Support: 3
Itemset: ['E', 'C'], Support: 2
Itemset: ['E', 'A'], Support: 2
Itemset: ['E', 'D'], Support: 2
Itemset: ['E', 'D', 'A'], Support: 2
Itemset: ['D'], Support: 4
```

Program

```
import pandas as pd
import numpy as np
def calc_entropy(data, label, class_list):
    total = data.shape[0]
    entropy = 0
    for c in class_list:
        p = data[data[label] == c].shape[0] / total
        if p > 0: entropy -= p * np.log2(p)
    return entropy
def calc_info_gain(feature, data, label, class_list):
    total_entropy = calc_entropy(data, label, class_list)
    values = data[feature].unique()
    weighted_entropy = sum((data[data[feature] == v].shape[0] / data.shape[0]) * calc_entropy(data[data[feature] == v], label, class_list) for v in values)
    return total_entropy - weighted_entropy
def find_best_feature(data, label, class_list):
    return max(data.columns.drop(label), key=lambda feature: calc_info_gain(feature, data, label, class_list))
def generate_tree(data, label):
    class_list = data[label].unique()
    if len(class_list) == 1: return class_list[0]
    if data.shape[1] == 1: return data[label].mode()[0]
    best_feature = find_best_feature(data, label, class_list)
    tree = {best_feature: {}}
    for value in data[best_feature].unique():
        subtree = generate_tree(data[data[best_feature] == value].drop(columns=[best_feature]), label)
        tree[best_feature][value] = subtree
    return tree
def predict(tree, instance):
    if not isinstance(tree, dict): return tree
    feature = next(iter(tree))
    value = instance[feature]
    return predict(tree[feature].get(value, None), instance)
def evaluate(tree, test_data, label):
    predictions = test_data.apply(lambda row: predict(tree, row), axis=1)
    accuracy = (predictions == test_data[label]).mean()
    return accuracy
train_data = pd.read_csv("PlayTennis.csv")
tree = generate_tree(train_data, "Play Tennis")
print(tree)
```

Output

```
(env) C:\Users\Asus\Desktop\Data Warehouse and Data Mining>python -u "c:\Users\Asus\Desktop\Data Warehouse and Data Mining\Id3.py"
{'Outlook': {'Sunny': 'No', 'Overcast': 'Yes', 'Rain': 'Yes'}}
```

Program

```
import numpy as np
import pandas as pd
def accuracy_score(y_true, y_pred):
    return round(float(sum(y_pred == y_true)) / len(y_true) * 100, 2)
def pre_processing(df):
    X = df.drop([df.columns[-1]], axis=1)
    y = df[df.columns[-1]]
    return X, y
class NaiveBayes:
    def __init__(self):
        self.likelihoods = {}
        self.class_priors = {}
        self.pred_priors = {}
    def fit(self, X, y):
        self.X_train, self.y_train = X, y
        self.train_size, self.features = X.shape[0], X.columns
        for feature in self.features:
            self.likelihoods[feature] = {feat_val: {outcome: 0 for outcome in y.unique()} for feat_val in
X[feature].unique()}
            self.pred_priors[feature] = X[feature].value_counts() / self.train_size
        self.class_priors = y.value_counts() / self.train_size
        self._calc_likelihoods()
    def _calc_likelihoods(self):
        for feature in self.features:
            for outcome in self.y_train.unique():
                outcome_indices = self.y_train[self.y_train == outcome].index
                outcome_count = len(outcome_indices)
                for feat_val, count in self.X_train[feature].iloc[outcome_indices].value_counts().items():
                    self.likelihoods[feature][feat_val][outcome] = count / outcome_count
    def predict(self, X):
        results = []
        for query in np.array(X):
            probs = {outcome: self.class_priors[outcome] * np.prod([self.likelihoods[feat][feat_val][outcome]
for feat, feat_val in zip(self.features, query)]) for outcome in self.y_train.unique()}
            results.append(max(probs, key=probs.get))
        return np.array(results)
if __name__ == "__main__":
    # Weather Dataset
    df = pd.read_table("./weather.txt")
    X, y = pre_processing(df)
    nb_clf = NaiveBayes()
```

```
nb_clf.fit(X, y)
print("Train Accuracy:", accuracy_score(y, nb_clf.predict(X)))
queries = [np.array([["Rainy", "Mild", "Normal", "Weak"]]),
        np.array([["Overcast", "Cool", "Normal", "Strong"]]),
        np.array([["Sunny", "Hot", "High", "Weak"]])]
for i, query in enumerate(queries, 1):
    print(f"Query {i}:- {query} ---> {nb_clf.predict(query)}")
```

Output

```
(env) C:\Users\Asus\Desktop\Data Warehouse and Data Mining>python -u "c:\Users\Asus\Desktop\Data Warehouse and Data Mining\Naive.py"
Train Accuracy: 20.0
Query 1:- [['Rainy' 'Mild' 'Normal' 'Weak']] ---> ['Sunny, Hot, High,weak,No']
Query 2:- [['Overcast' 'Cool' 'Normal' 'Strong']] ---> ['Sunny, Hot, High,weak,No']
Query 3:- [['Sunny' 'Hot' 'High' 'Weak']] ---> ['Sunny, Hot, High,weak,No']
```

Program

```
import numpy as np
class SVM:
    def __init__(self, learning_rate=0.001, lambda_param=0.01, n_iters=1000):
        self.lr = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None
    def fit(self, X, y):
        n_samples, n_features = X.shape
        # Initialize weights and bias
        self.w = np.zeros(n_features)
        self.b = 0
        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                condition = y[idx] * (np.dot(x_i, self.w) + self.b) >= 1
                if condition:
                    self.w -= self.lr * (2 * self.lambda_param * self.w)
                else:
                    self.w -= self.lr * (2 * self.lambda_param * self.w - np.dot(x_i, y[idx]))
                    self.b -= self.lr * y[idx]
    def predict(self, X):
        approx = np.dot(X, self.w) + self.b
        return np.sign(approx)
if __name__ == "__main__":
    # Example dataset
    X = np.array([[-2, 4], [4, 1], [1, 6], [2, 4], [6, 2]])  # Features
    y = np.array([-1, -1, 1, 1, 1])  # Labels
    svm = SVM(learning_rate=0.001, lambda_param=0.01, n_iters=1000)
    svm.fit(X, y)
    predictions = svm.predict(X)
    print("Predictions:", predictions)
```

Output

```
(env) C:\Users\Asus\Desktop\Data Warehouse and Data Mining>python -u "c:\Users\Asus\Desktop\Data Warehouse and Data Mining\SVm.py"
Predictions: [1. 1. 1. 1. 1.]
```

Program

```
import numpy as np
class LinearRegression:
   def __init__(self, learning_rate=0.01, n_iters=1000):
      self.lr = learning_rate
      self.n_iters = n_iters
      self.weights = None
      self.bias = None
   def fit(self, X, y):
      n_samples, n_features = X.shape
      # Initialize weights and bias
      self.weights = np.zeros(n_features)
      self.bias = 0
      for _ in range(self.n_iters):
         y_predicted = np.dot(X, self.weights) + self.bias
         dw = (1 / n_samples) * np.dot(X.T, (y_predicted - y))
         db = (1 / n_samples) * np.sum(y_predicted - y)
         self.weights -= self.lr * dw
         self.bias -= self.lr * db
   def predict(self, X):
      return np.dot(X, self.weights) + self.bias
if __name__ == "__main__":
   # Sample dataset
   X = np.array([[1], [2], [3], [4], [5]])  # Features
   y = np.array([1, 2, 3, 4, 5])  # Target values (for simplicity, it's a linear relationship)
   regressor = LinearRegression(learning_rate=0.01, n_iters=1000)
   regressor.fit(X, y)
   predictions = regressor.predict(X)
   print("Predictions:", predictions)
```

Output

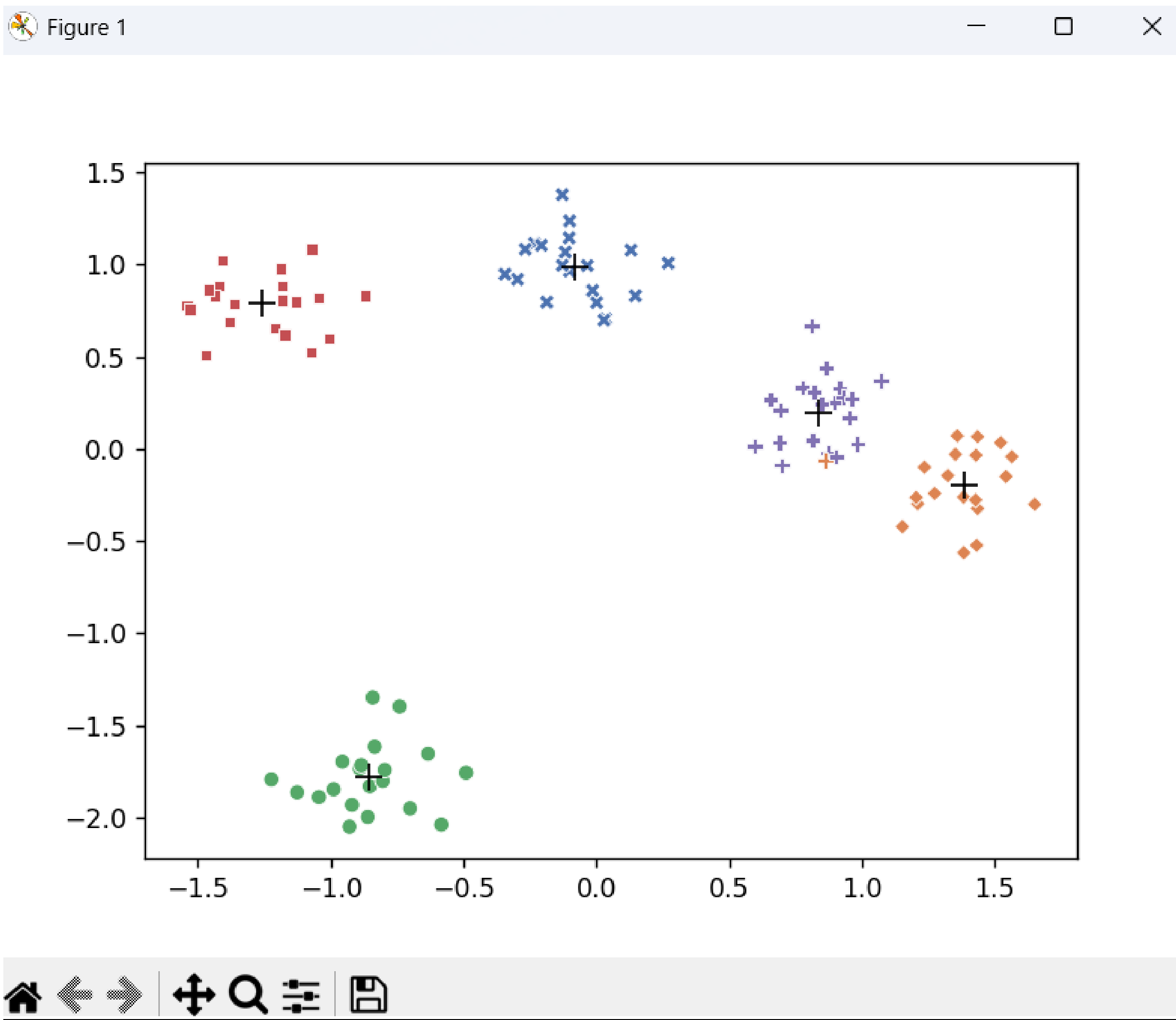
```
(env) C:\Users\Asus\Desktop\Data Warehouse and Data Mining>python -u "c:\Users\Asus\Desktop\Data Warehouse and Data Mining\linearregression.p
Predictions: [1.03425405 2.02113149 3.00800893 3.99488637 4.9817638 ]
```

Program

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs
import seaborn as sns
import random
def euclidean(point, data):
    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):
            dists = np.sum([euclidean(centroid, X_train) for centroid in self.centroids], axis=0)
            dists /= np.sum(dists)
            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):
            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_idx
```

```
centers = 5
X_train, true_labels = make_blobs(n_samples=100, centers=centers, random_state=42)
X_train = StandardScaler().fit_transform(X_train)
kmeans = KMeans(n_clusters=centers)
kmeans.fit(X_train)
class_centers, classification = kmeans.evaluate(X_train)
sns.scatterplot(x=X_train[:, 0], y=X_train[:, 1], 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



Program

```
import numpy as np
from typing import List, Tuple
def k_medoids_clustering(data: List[Tuple[float, float]], k: int, max_iter=100, random_seed=42) -> Tuple[List[int], List[Tuple[float, float]]]:
    np.random.seed(random_seed)
    data = np.array(data)
    N = data.shape[0]
    medoids_idx = np.random.choice(N, k, replace=False)
    medoids = data[medoids_idx]
    for _ in range(max_iter):
        distances = np.abs(data[:, np.newaxis] - medoids).sum(axis=2)
        labels = np.argmin(distances, axis=1)
       best_swap = (-1, -1, float('inf'))
        for i in range(k):
            for j in range(N):
                if j not in medoids_idx:
                    new_medoids = np.copy(medoids)
                    new_medoids[i] = data[j]
                    new_distances = np.abs(data[:, np.newaxis] - new_medoids).sum(axis=2)
                    cost_change = np.sum(new_distances[labels == i]) - np.sum(distances[labels == i, i])
                    if cost_change < best_swap[2]:
                        best_swap = (i, j, cost_change)
        if best_swap[2] == float('inf'): break
        i, j, _ = best_swap
        medoids[i] = data[j]
    return labels.tolist(), medoids.tolist()
if __name__ == "__main__":
    points = [(1, 2), (2, 3), (3, 4), (10, 11), (11, 12), (12, 13)]
    k = 2
    clusters, medoids = k_medoids_clustering(points, k)
    print("Clusters:")
    for point, cluster in zip(points, clusters):
        print(f"Point {point} belongs to cluster {cluster}")
    print("\nMedoids:")
    for i, medoid in enumerate(medoids):
        print(f"Cluster {i}: Medoid {medoid}")
```

Output

```
(env) C:\Users\Asus\Desktop\Data Warehouse and Data Mining>python -u "c:\Users\Asus\Desktop\Data Warehouse and Data Mining\k-medoid.py"
Clusters:
Point (1, 2) belongs to cluster 0
Point (2, 3) belongs to cluster 0
Point (3, 4) belongs to cluster 0
Point (10, 11) belongs to cluster 0
Point (11, 12) belongs to cluster 0
Point (12, 13) belongs to cluster 0

Medoids:
Cluster 0: Medoid [3, 4]
Cluster 1: Medoid [3, 4]
```

Program

```
import numpy as np
import string
def dbscan(D, eps, MinPts):
   labels = [0] * len(D)
   core_points, outliers = [], []
   C = 0
   for P in range(len(D)):
      if labels[P] != 0: continue
      NeighborPts = region_query(D, P, eps)
      if len(NeighborPts) < MinPts:
         labels[P] = -1
         outliers.append(P)
      else:
         C += 1
         core_points.append(P)
         grow_cluster(D, labels, P, NeighborPts, C, eps, MinPts)
   return labels, core_points, outliers
def grow_cluster(D, labels, P, NeighborPts, C, eps, MinPts):
   labels[P] = C
   i = 0
   while i < len(NeighborPts):
      Pn = NeighborPts[i]
      if labels[Pn] == -1: labels[Pn] = C
      elif labels[Pn] == 0:
         labels[Pn] = C
         PnNeighborPts = region_query(D, Pn, eps)
         if len(PnNeighborPts) >= MinPts:
            NeighborPts += PnNeighborPts
      i += 1
def region_query(D, P, eps):
   return [Pn for Pn in range(len(D)) if np.linalg.norm(np.array(D[P]) - np.array(D[Pn])) < eps] points
points = [(2, 10), (2, 5), (8, 4), (5, 8), (7, 5), (6, 4), (1, 2), (4, 9)]
alphabet_labels = list(string.ascii_uppercase)
point_labels = {i: label for i, label in enumerate(alphabet_labels)}

labels, core_points, outliers = dbscan(np.array(points), eps=2, MinPts=2)
print("Core Points:")
for idx in core_points: print(f"{point_labels[idx]}: {points[idx]}")
          print("Outliers:")
for idx in outliers:
          print(f"{point_labels[idx]}: {points[idx]}")
```

Output

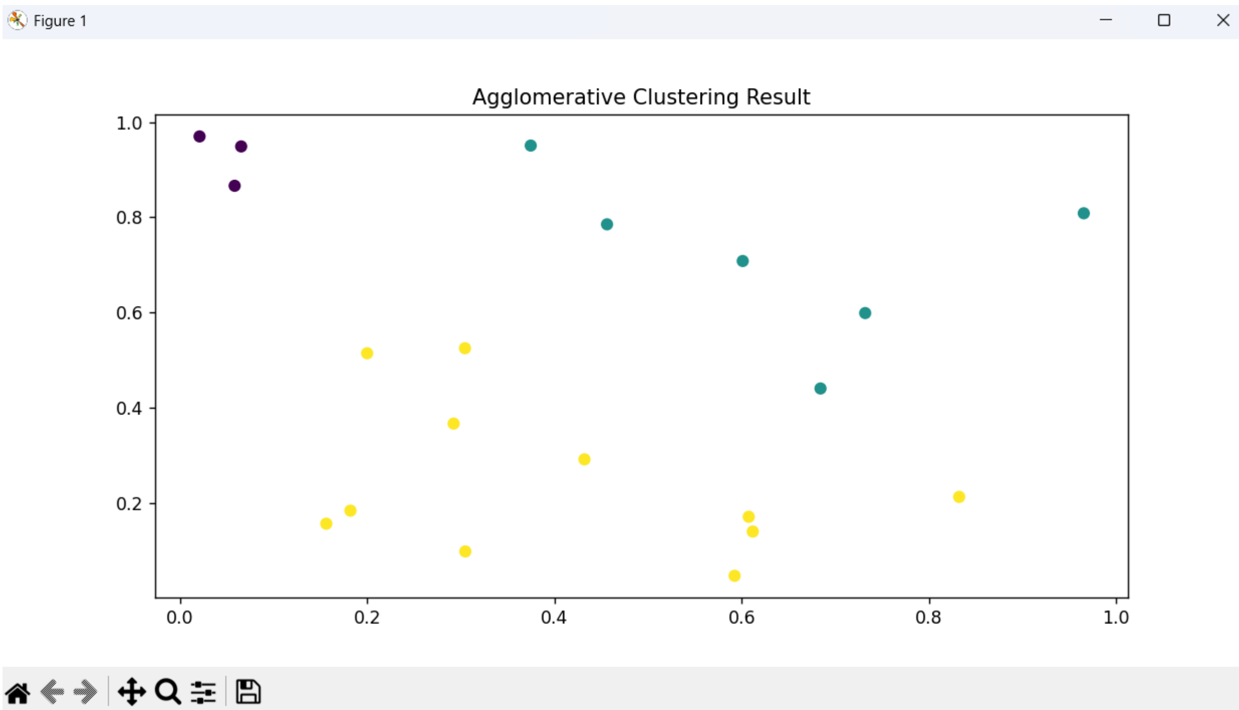
```
(env) C:\Users\Asus\Desktop\Data Warehouse and Data Mining>python -u "c:\Users\Asus\Desktop\Data Warehouse and Data Mining\DBSCAN.py"
Core Points:
C: (8, 4)
D: (5, 8)
Outliers:
A: (2, 10)
B: (2, 5)
G: (1, 2)
```

Program

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram
class AgglomerativeClustering:
  def __init__(self, n_clusters=2, linkage='ward'):
    self.n_clusters = n_clusters
    self.linkage = linkage
    self.labels_ = None
    self.distances_ = None
  def fit(self, X):
    n_samples = X.shape[0]
    clusters = [[i] for i in range(n_samples)]
    dist_matrix = self._compute_distance_matrix(X)
    while len(clusters) > self.n_clusters:
      # Find closest clusters
      min_dist = float('inf')
      merge_i, merge_j = 0, 0
      for i in range(len(clusters)):
        for j in range(i+1, len(clusters)):
          dist = self._cluster_distance(dist_matrix, clusters[i], clusters[j])
          if dist < min_dist:
            min_dist = dist
            merge_i, merge_j = i, j
      merged_cluster = clusters[merge_i] + clusters[merge_j]
      clusters.pop(max(merge_i, merge_j))
      clusters.pop(min(merge_i, merge_j))
      clusters.append(merged_cluster)
    self.labels_ = np.zeros(n_samples, dtype=int)
    for cluster_id, cluster in enumerate(clusters):
      for idx in cluster:
        self.labels_[idx] = cluster_i
    return self
  def _compute_distance_matrix(self, X):
    return np.sqrt(((X[:, np.newaxis, :] - X[np.newaxis, :, :]) ** 2).sum(axis=2))
  def _cluster_distance(self, dist_matrix, cluster1, cluster2):
    # Ward's method (minimum variance method)
    distances = [dist_matrix[i][j] for i in cluster1 for j in cluster2]
    return np.mean(distances)
np.random.seed(42)
X = np.random.rand(20, 2)  # 20 random 2D points
clustering = AgglomerativeClustering(n_clusters=3)
```

```
clustering.fit(X)
plt.figure(figsize=(10, 5))
plt.scatter(X[:, 0], X[:, 1], c=clustering.labels_, cmap='viridis')
plt.title('Agglomerative Clustering Result')
plt.show()
```

Output



Program

```
import pandas as pd
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split
def preprocess_data(data):
    # Load data into DataFrame
    df = pd.DataFrame(data)
    # Handle missing numeric values with mean
    df.fillna(df.select_dtypes(include='number').mean(), inplace=True)

    # Handle missing categorical values with mode
    most_frequent_category = df['category'].mode()[0]
    df['category'].fillna(most_frequent_category, inplace=True)
    label_encoder = LabelEncoder()
    df['category'] = label_encoder.fit_transform(df['category'])
    scaler = StandardScaler()
    df[['feature1', 'feature2']] = scaler.fit_transform(df[['feature1', 'feature2']])
    X = df.drop('target', axis=1)
    y = df['target']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    return X_train, X_test, y_train, y_test
data = {
    'feature1': [1.0, 2.0, None, 4.0, 5.0],
    'feature2': [10.0, None, 30.0, 40.0, 50.0],
    'category': ['A', 'B', 'A', None, 'B'],
    'target': [0, 1, 0, 1, 0]
}
X_train, X_test, y_train, y_test = preprocess_data(data)
print("Training Data Shapes:")
print(f"X_train: {X_train.shape}")
print(f"y_train: {y_train.shape}")
print("\nTesting Data Shapes:")
print(f"X_test: {X_test.shape}")
print(f"y_test: {y_test.shape}")
```