```
'''
  Bardia Mojra

  @link https://github.com/ZwEin27/Hierarchical-Clustering/blob/master/hclust.py
'''

import numpy as np
import matplotlib.pyplot as plt
import sys
import math
import os
import heapq
import itertools
import pdb


D= [(170,57,32),(190,95,28),(150,45,35),(168,65,29),(175,78,26),(185,90,32),
(171,65,28),(155,48,31),(165,60,27),(182,80,30),(175,69,28),(178,80,27),
(160,50,31),(170,72,30)]

Y = ['W', 'M', 'W', 'M', 'M', 'M', 'W', 'W', 'W', 'M', 'W', 'M', 'W', 'M']


def get_data(data, label):
  Xlist = list()
  for x in data:
    Xlist.append(np.asarray(x))
  X = np.asarray(Xlist)
  print(label+':')
  print(X)
  print('\n')
  return X

'''
def euclidean_distance(self, row_A, row_B):
  dist = 0.0
  diffList = list()
  for i in range(len(row_A[0])):
    diff = 0.0
    diff = row_A[0][i]-row_B[i]
    diffList.append(diff)
    dist += (diff)**2
  dist = math.sqrt(dist)
  return round(dist, self.precision)
'''

class Hierarchical_Clustering:
  def __init__(self, ipt_data, k, linkage='single'):
    self.input_file_name = ipt_data
    self.k = k
    self.dataset = None
    self.linkage = linkage
    self.dataset_size = 0
    self.dimension = 0
    self.heap = []
    self.clusters = []
    #self.gold_standard = {}

  def initialize(self):
    """ Initialize and check parameters
    """
    # check file exist and if it's a file or dir
    if not os.path.isfile(self.input_file_name):
      self.quit("Input file doesn't exist or it's not a file")
    self.dataset, self.clusters = self.load_data(self.input_file_name)
    self.dataset_size = len(self.dataset)
    if self.dataset_size == 0:
      self.quit("Input file doesn't include any data")
    if self.k == 0:
      self.quit("k = 0, no cluster will be generated")
    if self.k > self.dataset_size:
      self.quit("k is larger than the number of existing clusters")
    self.dimension = len(self.dataset[0]["data"])
    if self.dimension == 0:
      self.quit("dimension for dataset cannot be zero")


  def euclidean_distance(self, data_point_one, data_point_two):
    """
    euclidean distance: https://en.wikipedia.org/wiki/Euclidean_distance
    assume that two data points have same dimension

    """
    size = len(data_point_one)
    result = 0.0
    for i in range(size):
      f1 = float(data_point_one[i])   # feature for data one
      f2 = float(data_point_two[i])   # feature for data two
      tmp = f1 - f2
      result += pow(tmp, 2)
    result = math.sqrt(result)
    return result

  def compute_pairwise_distance(self, dataset):
    result = []
    dataset_size = len(dataset)
    for i in range(dataset_size-1):    # ignore last i
      for j in range(i+1, dataset_size):     # ignore duplication
        dist = self.euclidean_distance(dataset[i]["data"], dataset[j]["data"])

        # duplicate dist, need to be remove, and there is no difference to use t
uple only
        # leave second dist here is to take up a position for tie selection
        result.append( (dist, [dist, [[i], [j]]]) )
    return result

  def build_priority_queue(self, distance_list):
    if self.linkage == 'single':
      heapq.heapify(distance_list)
    elif self.linkage == 'complete':
      heapq._heapify_max(distance_list)
    else:
      print("error - heapq not properly assigned.")
    self.heap = distance_list
    return self.heap

  def compute_centroid_two_clusters(self, current_clusters, data_points_index):
    size = len(data_points_index)
    dim = self.dimension
    centroid = [0.0]*dim
    for index in data_points_index:
      dim_data = current_clusters[str(index)]["centroid"]
      for i in range(dim):
        centroid[i] += float(dim_data[i])
    for i in range(dim):
      centroid[i] /= size
    return centroid

  def compute_centroid(self, dataset, data_points_index):
    size = len(data_points_index)
    dim = self.dimension
    centroid = [0.0]*dim
    for idx in data_points_index:
      dim_data = dataset[idx]["data"]
      for i in range(dim):
        centroid[i] += float(dim_data[i])
    for i in range(dim):
      centroid[i] /= size
    return centroid

  def hierarchical_clustering(self):
    """
    Main Process for hierarchical clustering
    """
    dataset = self.dataset
    current_clusters = self.clusters
    old_clusters = []
    heap = hc.compute_pairwise_distance(dataset)
    heap = hc.build_priority_queue(heap)

    while len(current_clusters) > self.k:
      if self.linkage == 'single':
        dist, m_item = heapq.heappop(heap)  # get min distance
      elif self.linkage == 'complete':
        dist, m_item = heapq._heappop_max(heap)  # get max distance
      else:
        print("error - heapq not properly assigned.")

      # pair_dist = m_item[0]
      pair_data = m_item[1]
      #pdb.set_trace()
      # judge if include old cluster
      if not self.valid_heap_node(m_item, old_clusters):
        continue

      print('> merging '+str(m_item[1][0][0])+' & '+str(m_item[1][1][0]) \
        +' with distance of '+str(rnd(m_item[0], 3)))
      new_cluster = {}
      new_cluster_elements = sum(pair_data, [])
      new_cluster_cendroid = self.compute_centroid(dataset, new_cluster_elements
)
      new_cluster_elements.sort()
      new_cluster.setdefault("centroid", new_cluster_cendroid)
      new_cluster.setdefault("elements", new_cluster_elements)
      for pair_item in pair_data:
        old_clusters.append(pair_item)
        del current_clusters[str(pair_item)]
      self.add_heap_entry(heap, new_cluster, current_clusters)
      current_clusters[str(new_cluster_elements)] = new_cluster
    current_clusters = sorted(current_clusters)
    return current_clusters

  def valid_heap_node(self, heap_node, old_clusters):
    pair_dist = heap_node[0]
    pair_data = heap_node[1]
    for old_cluster in old_clusters:
      if old_cluster in pair_data:
        return False
    return True

  def add_heap_entry(self, heap, new_cluster, current_clusters):
    for ex_cluster in current_clusters.values():
      new_heap_entry = []
      dist = self.euclidean_distance(ex_cluster["centroid"], new_cluster["centro
id"])
      new_heap_entry.append(dist)
      new_heap_entry.append([new_cluster["elements"], ex_cluster["elements"]])
      heapq.heappush(heap, (dist, new_heap_entry))

  '''
  def evaluate(self, current_clusters):
    gold_standard = self.gold_standard
    current_clustes_pairs = []

    for (current_cluster_key, current_cluster_value) in current_clusters.items()
:
      tmp = list(itertools.combinations(current_cluster_value["elements"], 2))
      current_clustes_pairs.extend(tmp)
    tp_fp = len(current_clustes_pairs)

    gold_standard_pairs = []
    for (gold_standard_key, gold_standard_value) in gold_standard.items():
      tmp = list(itertools.combinations(gold_standard_value, 2))
      gold_standard_pairs.extend(tmp)
    tp_fn = len(gold_standard_pairs)
    tp = 0.0
    for ccp in current_clustes_pairs:
      if ccp in gold_standard_pairs:
        tp += 1
    if tp_fp == 0:
      precision = 0.0
    else:
      precision = tp/tp_fp
    if tp_fn == 0:
      precision = 0.0
    else:
      recall = tp/tp_fn
    return precision, recall
  '''

  ''' Helper Functions
  '''
  def load_data(self, input_file_name):
    """
    load data and do some preparations

    """
    input_file = open(input_file_name, 'r')
    dataset = []
    clusters = {}
    gold_standard = {}
    id = 0
    for line in input_file:
      line = line.strip('\n')
      row = str(line)
      row = row.split(",")
      iris_class = row[-1]

      data = {}
      data.setdefault("id", id)   # duplicate
      data.setdefault("data", row[:-1])
      data.setdefault("class", row[-1])
      dataset.append(data)

      clusters_key = str([id])
      clusters.setdefault(clusters_key, {})
      clusters[clusters_key].setdefault("centroid", row[:-1])
      clusters[clusters_key].setdefault("elements", [id])

      #gold_standard.setdefault(iris_class, [])
      #gold_standard[iris_class].append(id)
      id += 1
    return dataset, clusters #, gold_standard

  def quit(self, err_desc):
    raise SystemExit('\n'+ "PROGRAM EXIT: " + err_desc + ', please check your in
put' + '\n')

  def loaded_dataset(self):
    """
    use for test only
    """
    return self.dataset

  def display(self, current_clusters):
    print()
    print('final clusters:')
    clusters = current_clusters
    for cluster in clusters:
      print(cluster)

def get_clusters(clus_set):
  clusters = list()
  for cluster in clus_set:
    clus = cluster.replace('[','')
    clus = clus.replace(']','')
    clus = clus.replace(' ','')
    clusList = clus.split(',')
    clusList = [ int(n) for n in clusList]
    clusters.append(clusList)
  return clusters

def rnd(num, precision):
  return math.floor(num * 10**precision)/10**precision

def get_GT(y):
  pdb.set_trace()

  return y
```

```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""
"""                                 Main Method
 """
"""
 """
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
""""
if __name__ == '__main__':

  '''Pseudo code for part 2:
    - Mix the two datasets to create a new combined dataset, lets call it allD.
    - Use previous implementation to cluster similar data points into two classe
s,
    M, and W.
    - For the two clusters, count the majority class to label the entire cluster
.
    - Then, for each cluster, remove unlabeled data and use logistic regression
    to train each cluster on labeled data.



  '''
```