import random

import numpy as np

import pandas as pd

class ClaraClustering:

"""The clara clustering algorithm.

Basically an iterative guessing version of k-mediods that makes things a lot faster

for bigger data sets.

"""

def \_\_init\_\_(self, max\_iter):

"""Class initialization.

:param max\_iter: The default number of max iterations

"""

self.max\_iter = max\_iter

self.dist\_cache = dict()

def clara(self, \_df, \_k, \_fn):

"""The main clara clustering iterative algorithm.

:param \_df: Input data frame.

:param \_k: Number of medoids.

:param \_fn: The distance function to use.

:return: The minimized cost, the best medoid choices and the final configuration.

"""

size = len(\_df)

if size > 100000:

niter = 1000

runs = 1

else:

niter = self.max\_iter

runs = 5

min\_avg\_cost = np.inf

best\_choices = []

best\_results = {}

for j in range(runs):

sampling\_idx = random.sample([i for i in range(size)], (40+\_k\*2))

sampling\_data = []

for idx in sampling\_idx:

sampling\_data.append(\_df.iloc[idx])

sampled\_df = pd.DataFrame(sampling\_data)

pre\_cost, pre\_choice, pre\_medoids = self.k\_medoids(sampled\_df, \_k, \_fn, niter)

tmp\_avg\_cost, tmp\_medoids = self.average\_cost(\_df, \_fn, pre\_choice)

if tmp\_avg\_cost <= min\_avg\_cost:

min\_avg\_cost = tmp\_avg\_cost

best\_choices = list(pre\_choice)

best\_results = dict(tmp\_medoids)

return min\_avg\_cost, best\_choices, best\_results

def k\_medoids(self, \_df, \_k, \_fn, \_niter):

"""The original k-mediods algorithm.

:param \_df: Input data frame.

:param \_k: Number of medoids.

:param \_fn: The distance function to use.

:param \_niter: The number of iterations.

:return: Cluster label.

Pseudo-code for the k-mediods algorithm.

1. Sample k of the n data points as the medoids.

2. Associate each data point to the closest medoid.

3. While the cost of the data point space configuration is decreasing.

1. For each medoid m and each non-medoid point o:

1. Swap m and o, recompute cost.

2. If global cost increased, swap back.

"""

print('K-medoids starting')

# Do some smarter setting of initial cost configuration

pc1, medoids\_sample = self.cheat\_at\_sampling(\_df, \_k, \_fn, 17)

prior\_cost, medoids = self.compute\_cost(\_df, \_fn, medoids\_sample)

current\_cost = prior\_cost

iter\_count = 0

best\_choices = []

best\_results = {}

print('Running with {m} iterations'.format(m=\_niter))

while iter\_count < \_niter:

for m in medoids:

clust\_iter = 0

for item in medoids[m]:

if item != m:

idx = medoids\_sample.index(m)

swap\_temp = medoids\_sample[idx]

medoids\_sample[idx] = item

tmp\_cost, tmp\_medoids = self.compute\_cost(\_df, \_fn, medoids\_sample, True)

if (tmp\_cost < current\_cost) & (clust\_iter < 1):

best\_choices = list(medoids\_sample)

best\_results = dict(tmp\_medoids)

current\_cost = tmp\_cost

clust\_iter += 1

else:

best\_choices = best\_choices

best\_results = best\_results

current\_cost = current\_cost

medoids\_sample[idx] = swap\_temp

iter\_count += 1

if best\_choices == medoids\_sample:

print('Best configuration found!')

break

if current\_cost <= prior\_cost:

prior\_cost = current\_cost

medoids = best\_results

medoids\_sample = best\_choices

return current\_cost, best\_choices, best\_results

def compute\_cost(self, \_df, \_fn, \_cur\_choice, cache\_on=True):

"""A function to compute the configuration cost.

:param \_df: The input data frame.

:param \_fn: The distance function.

:param \_cur\_choice: The current set of medoid choices.

:param cache\_on: Binary flag to turn caching.

:return: The total configuration cost, the mediods.

"""

size = len(\_df)

total\_cost = 0.0

medoids = {}

for idx in \_cur\_choice:

medoids[idx] = []

for i in range(size):

choice = -1

min\_cost = np.inf

for m in medoids:

if cache\_on:

tmp = self.dist\_cache.get((m, i), None)

if not cache\_on or tmp is None:

if \_fn == 'manhattan':

tmp = self.manhattan\_distance(\_df.iloc[m], \_df.iloc[i])

elif \_fn == 'cosine':

tmp = self.cosine\_distance(\_df.iloc[m], \_df.iloc[i])

elif \_fn == 'euclidean':

tmp = self.euclidean\_distance(\_df.iloc[m], \_df.iloc[i])

elif \_fn == 'fast\_euclidean':

tmp = self.fast\_euclidean(\_df.iloc[m], \_df.iloc[i])

else:

print('You need to input a distance function.')

if cache\_on:

self.dist\_cache[(m, i)] = tmp

if tmp < min\_cost:

choice = m

min\_cost = tmp

medoids[choice].append(i)

total\_cost += min\_cost

return total\_cost, medoids

def average\_cost(self, \_df, \_fn, \_cur\_choice):

"""A function to compute the average cost.

:param \_df: The input data frame.

:param \_fn: The distance function.

:param \_cur\_choice: The current medoid candidates.

:return: The average cost, the new medoids.

"""

\_tc, \_m = self.compute\_cost(\_df, \_fn, \_cur\_choice)

avg\_cost = \_tc / len(\_m)

return avg\_cost, \_m

def cheat\_at\_sampling(self, \_df, \_k, \_fn, \_nsamp):

"""A function to cheat at sampling for speed ups.

:param \_df: The input data frame.

:param \_k: The number of medoids.

:param \_fn: The distance function.

:param \_nsamp: The number of samples.

:return: The best score, the medoids.

"""

size = len(\_df)

score\_holder = []

medoid\_holder = []

for \_ in range(\_nsamp):

medoids\_sample = random.sample([i for i in range(size)], \_k)

prior\_cost, medoids = self.compute\_cost(\_df, \_fn, medoids\_sample, True)

score\_holder.append(prior\_cost)

medoid\_holder.append(medoids)

idx = score\_holder.index(min(score\_holder))

ms = medoid\_holder[idx].keys()

return score\_holder[idx], ms

def euclidean\_distance(self, v1, v2):

"""Slow function for euclidean distance.

:param v1: The first vector.

:param v2: The second vector.

:return: The euclidean distance between v1 and v2.

"""

dist = 0

for a1, a2 in zip(v1, v2):

dist += abs(a1 - a2)\*\*2

return dist

def fast\_euclidean(self, v1, v2):

"""Faster function for euclidean distance.

:param v1: The first vector.

:param v2: The second vector.

:return: The euclidean distance between v1 and v2.

"""

return np.linalg.norm(v1 - v2)

def manhattan\_distance(self, v1, v2):

"""Function for manhattan distance.

:param v1: The first vector.

:param v2: The second vector.

:return: The manhattan distance between v1 and v2.

"""

dist = 0

for a1, a2 in zip(v1, v2):

dist += abs(a1 - a2)

return dist

def cosine\_distance(self, v1, v2):

"""Function for cosine distance.

:param v1: The first vector.

:param v2: The second vector.

:return: The cosine distance between v1 and v2.

"""

xx, yy, xy = 0, 0, 0

for a1, a2 in zip(v1, v2):

xx += a1\*a1

yy += a2\*a2

xy += a1\*a2

return float(xy) / np.sqrt(xx\*yy)

data = pd.read\_csv("E:\Kak Lia\koordinat.csv") #Load data berformat .csv berdasarkan lokasi file

print data

ridesharing = data.drop(["Name"], axis=1) #Menghapus attribute data koordinat yang tidak diperlukan

dist = 'euclidean' #metode distance ['euclidean','fast\_euclidean','manhattan','cosine']

max\_i=100000

CLARA = ClaraClustering(max\_i)

out = CLARA.clara(ridesharing, 5, dist)

for x in out:

print x #Memapilkan hasil Cluster dari algoritma Clara