import random

import numpy as np

# 7(a)

def compute\_centroids(points, cluster\_ids):

'''

Compute centroids of clusters

points : value of points (dimension 1), pyspark.rdd.RDD

clusters\_ids : ids of clusters associated with points, pyspark.rdd.RDD

'''

# create tuples (id, value)

temp = cluster\_ids.zip(points)

# sum values for each key

sum\_by\_cluster\_id = temp.reduceByKey(lambda x, y: x + y)

# .mapValues(lambda x: 1) used to count each elements by keys

count\_by\_cluster\_id = temp.mapValues(lambda x: 1).reduceByKey(lambda x, y: x+y)

# compute means by key

centroids = sum\_by\_cluster\_id.join(count\_by\_cluster\_id).mapValues(lambda x: x[0] / x[1])

return centroids

def squared\_distances(value, means):

# return [(value - mean) \*\* 2 for mean in means] # 7(b)i

return [np.sum((value - mean) \*\* 2) for mean in means] # 8.

def assign\_clusters(points, centroids):

'''

Assign each points to a cluster, return a pyspark.rdd.RDD

points : pyspark.rdd.RDD

centroids : result of compute\_centroids, pyspark.rdd.RDD

'''

# 7(b)ii

means = centroids.values().collect()

# 7(b)iii

# search index of closest centroid of each point

temp = points.map(lambda x: np.argmin(squared\_distances(x, means)))

# recover index of cluster's centroid

centroids\_keys = centroids.keys().collect() # impossible to use RDD object

# in .map, we keep the list for next line

assigned\_clusters = temp.map(lambda x: centroids\_keys[x])

return assigned\_clusters

class UnidimensionalKmeans:

def \_\_init\_\_(self, K, itermax):

self.K = K

self.itermax = itermax

self.best\_centroids = 'nothing yet'

def fit(self, points):

'''

points : pyspark.rdd.RDD

'''

# initializing by assign random clusters to each point of points

cluster\_ids = points.map(lambda x: random.choice(range(self.K)))

iteration = 0

condition = True

while condition & (iteration < self.itermax):

centroids = compute\_centroids(points, cluster\_ids)

cluster\_ids\_new = assign\_clusters(points, centroids)

# stop algorithm when clusters\_ids doesn't change

condition = (cluster\_ids.collect() != cluster\_ids\_new.collect())

# update index of clusters

cluster\_ids = cluster\_ids\_new

iteration += 1

self.best\_centroids = centroids # save centroids

print('Done ! (in {} iterations)'.format(iteration))

def predict(self, points):

'''

Return RDD with each points assigned to a cluster

points : pyspark.rdd.RDD

'''

if not isinstance(self.best\_centroids, str):

return assign\_clusters(points, self.best\_centroids)

else:

print('Model need to be fitted before !')

class MultidimensionalKmeans(UnidimensionalKmeans):

def \_\_init\_\_(self, K, itermax):

super().\_\_init\_\_(K, itermax)

def fit(self, points):

'''

points : pyspark.rdd.RDD

'''

points = points.map(lambda x : np.array(x)) # only change from super class

cluster\_ids = points.map(lambda x: random.choice(range(self.K)))

iteration = 0

condition = True

while condition & (iteration < self.itermax):

centroids = compute\_centroids(points, cluster\_ids)

cluster\_ids\_new = assign\_clusters(points, centroids)

condition = (cluster\_ids.collect() != cluster\_ids\_new.collect())

cluster\_ids = cluster\_ids\_new

iteration += 1

self.best\_centroids = centroids

print('Done ! (in {} iterations)'.format(iteration))

def cosin\_distances(value, centroids):

return [np.dot(value, centroid)/np.dot(centroid, centroid) for centroid in centroids]

def assign\_clusters\_spherical(points, centroids):

'''

Assign each points to a cluster, return a pyspark.rdd.RDD

points : pyspark.rdd.RDD

centroids : result of compute\_centroids, pyspark.rdd.RDD

'''

means = centroids.values().collect()

temp = points.map(lambda x: np.argmin(cosin\_distances(x, means)))

# recover index of cluster's centroid

centroids\_keys = centroids.keys().collect()

assigned\_clusters = temp.map(lambda x: centroids\_keys[x])

return assigned\_clusters

class SphericalKmeans(UnidimensionalKmeans):

def \_\_init\_\_(self, K, itermax):

super().\_\_init\_\_(K, itermax)

def fit(self, points):

'''

points : pyspark.rdd.RDD

'''

points = points.map(lambda x : np.array(x)/np.linalg.norm(np.array(x)))

# only change from super class, apply a normalization on vector

cluster\_ids = points.map(lambda x: random.choice(range(self.K)))

iteration = 0

condition = True

while condition & (iteration < self.itermax):

centroids = compute\_centroids(points, cluster\_ids)

# centroids may not be on the hypersphere

cluster\_ids\_new = assign\_clusters\_spherical(points, centroids)

condition = (cluster\_ids.collect() != cluster\_ids\_new.collect())

cluster\_ids = cluster\_ids\_new

iteration += 1

self.best\_centroids = centroids

print('Done ! (in {} iterations)'.format(iteration))

def predict(self, points):

if not isinstance(self.best\_centroids, str):

return assign\_clusters\_spherical(points, self.best\_centroids)

else:

print('Model need to be fitted before !')

if \_\_name\_\_ == '\_\_main\_\_':

pass