15BCE0517

M.S.SANJAY

L7+L8

MACHINELEARNINGLABACTIVITY1

I have taken all inputsfrom user including kvalue, cutoff,no of points, number of dimensions ,linkage,cutoff and points

code:

# -\*- coding: utf-8 -\*-

"""

Created on Tue Dec 12 09:10:31 2017

@author: 15bce0517

"""

# clustering.py contains classes and functions that cluster data points

import sys, math, random

# -- The Point class represents points in n-dimensional space

class Point:

# Instance variables

# self.coords is a list of coordinates for this Point

# self.n is the number of dimensions this Point lives in (ie, its space)

# self.reference is an object bound to this Point

# Initialize new Points

def \_\_init\_\_(self, coords, reference=None):

self.coords = coords

self.n = len(coords)

self.reference = reference

# Return a string representation of this Point

def \_\_repr\_\_(self):

return str(self.coords)

# -- The Cluster class represents clusters of points in n-dimensional space

class Cluster:

# Instance variables

# self.points is a list of Points associated with this Cluster

# self.n is the number of dimensions this Cluster's Points live in

# self.centroid is the sample mean Point of this Cluster

# Initialize new Clusters

def \_\_init\_\_(self, points):

# We forbid empty Clusters (they don't make mathematical sense!)

if len(points) == 0: raise Exception("ILLEGAL: EMPTY CLUSTER")

self.points = points

self.n = points[0].n

# We also forbid Clusters containing Points in different spaces

# Ie, no Clusters with 2D Points and 3D Points

for p in points:

if p.n != self.n: raise Exception("ILLEGAL: MULTISPACE CLUSTER")

# Figure out what the centroid of this Cluster should be

self.centroid = self.calculateCentroid()

# Return a string representation of this Cluster

def \_\_repr\_\_(self):

return str(self.points)

# Update function for the K-means algorithm

# Assigns a new list of Points to this Cluster, returns centroid difference

def update(self, points):

old\_centroid = self.centroid

self.points = points

self.centroid = self.calculateCentroid()

return getDistance(old\_centroid, self.centroid)

# Calculates the centroid Point - the centroid is the sample mean Point

# (in plain English, the average of all the Points in the Cluster)

def calculateCentroid(self):

centroid\_coords = []

# For each coordinate:

for i in range(self.n):

# Take the average across all Points

centroid\_coords.append(0.0)

for p in self.points:

centroid\_coords[i] = centroid\_coords[i]+p.coords[i]

centroid\_coords[i] = centroid\_coords[i]/len(self.points)

# Return a Point object using the average coordinates

return Point(centroid\_coords)

# Return the single-linkage distance between this and another Cluster

def getSingleDistance(self, cluster):

ret = getDistance(self.points[0], cluster.points[0])

for p in self.points:

for q in cluster.points:

distance = getDistance(p, q)

if distance < ret: ret = distance

return ret

# Return the complete-linkage distance between this and another Cluster

def getCompleteDistance(self, cluster):

ret = getDistance(self.points[0], cluster.points[0])

for p in self.points:

for q in cluster.points:

distance = getDistance(p, q)

if distance > ret: ret = distance

return ret

# Return the centroid-linkage distance between this and another Cluster

def getCentroidDistance(self, cluster):

return getDistance(self.centroid, cluster.centroid)

# Return the fusion of this and another Cluster

def fuse(self, cluster):

# Forbid fusion of Clusters in different spaces

if self.n != cluster.n: raise Exception("ILLEGAL FUSION")

points = self.points

points.extend(cluster.points)

return Cluster(points)

# -- Return Clusters of Points formed by K-means clustering

def kmeans(points, k, cutoff):

# Randomly sample k Points from the points list, build Clusters around them

initial = random.sample(points, k)

clusters = []

for p in initial: clusters.append(Cluster([p]))

# Enter the program loop

while True:

# Make a list for each Cluster

lists = []

for c in clusters: lists.append([])

# For each Point:

for p in points:

# Figure out which Cluster's centroid is the nearest

smallest\_distance = getDistance(p, clusters[0].centroid)

index = 0

for i in range(len(clusters[1:])):

distance = getDistance(p, clusters[i+1].centroid)

if distance < smallest\_distance:

smallest\_distance = distance

index = i+1

# Add this Point to that Cluster's corresponding list

lists[index].append(p)

# Update each Cluster with the corresponding list

# Record the biggest centroid shift for any Cluster

biggest\_shift = 0.0

for i in range(len(clusters)):

shift = clusters[i].update(lists[i])

biggest\_shift = max(biggest\_shift, shift)

# If the biggest centroid shift is less than the cutoff, stop

if biggest\_shift < cutoff: break

# Return the list of Clusters

return clusters

# -- Return a distance matrix which captures distances between all Clusters

def makeDistanceMatrix(clusters, linkage):

ret = dict()

for i in range(len(clusters)):

for j in range(len(clusters)):

if j == i: break

if linkage == 's':

ret[(i,j)] = clusters[i].getSingleDistance(clusters[j])

elif linkage == 'c':

ret[(i,j)] = clusters[i].getCompleteDistance(clusters[j])

elif linkage == 't':

ret[(i,j)] = clusters[i].getCentroidDistance(clusters[j])

else: raise Exception("INVALID LINKAGE")

return ret

# -- Return Clusters of Points formed by agglomerative clustering

def agglo(points, linkage, cutoff):

# Currently, we only allow single, complete, or average linkage

if not linkage in [ 's', 'c', 't' ]: raise Exception("INVALID LINKAGE")

# Create singleton Clusters, one for each Point

clusters = []

for p in points: clusters.append(Cluster([p]))

# Set the min\_distance between Clusters to zero

min\_distance = 0

# Loop until the break statement is made

while (True):

# Compute a distance matrix for all Clusters

distances = makeDistanceMatrix(clusters, linkage)

# Find the key for the Clusters which are closest together

min\_key = distances.keys()[0]

min\_distance = distances[min\_key]

for key in distances.keys():

if distances[key] < min\_distance:

min\_key = key

min\_distance = distances[key]

# If the min\_distance is bigger than the cutoff, terminate the loop

# Otherwise, agglomerate the closest clusters

if min\_distance > cutoff or len(clusters) == 1: break

else:

c1, c2 = clusters[min\_key[0]], clusters[min\_key[1]]

clusters.remove(c1)

clusters.remove(c2)

clusters.append(c1.fuse(c2))

# Return the list of Clusters

return clusters

# -- Get the Euclidean distance between two Points

def getDistance(a, b):

# Forbid measurements between Points in different spaces

if a.n != b.n: raise Exception("ILLEGAL: NON-COMPARABLE POINTS")

# Euclidean distance between a and b is sqrt(sum((a[i]-b[i])^2) for all i)

ret = 0.0

for i in range(a.n):

ret = ret+pow((a.coords[i]-b.coords[i]), 2)

return math.sqrt(ret)

# -- Create a random Point in n-dimensional space

def makeRandomPoint(n):

coords = []

for i in range(n): coords.append(input())

return Point(coords)

# -- Plot Clusters using Tkinter

def plot(clusters):

root = Tk()

cp = ClusterPlot(root)

root.mainLoop()

# -- Main function

def main(args):

num\_points=input()

n=input()

k=input()

kmeans\_cutoff=input()

linkage=input()

agglo\_cutoff=input()

# Create num\_points random Points in n-dimensional space, print them

print "\nPOINTS:"

points = []

for i in range(num\_points):

p = makeRandomPoint(n)

points.append(p)

print "P:", p

# Cluster the points using the K-means algorithm, print the results

clusters = kmeans(points, k, kmeans\_cutoff)

print "\nK-MEANS\nCLUSTERS:"

for c in clusters: print "C:", c

# Cluster the points using the agglomerative algorithm, print the results

clusters = agglo(points, linkage, agglo\_cutoff)

print "\nAGGLOMERATIVE\nCLUSTERS:"

for c in clusters: print "C:", c

# -- The following code executes upon command-line invocation

if \_\_name\_\_ == "\_\_main\_\_": main(sys.argv)

OUTPUT:

>>> runfile('/home/likewise-open/VITUNIVERSITY/15bce0517/untitled2.py', wdir=r'/home/likewise-open/VITUNIVERSITY/15bce0517')

10

2

3

0.5

's'

20

POINTS:

1

2

P: [1, 2]

3

4

P: [3, 4]

5

6

P: [5, 6]

7

8

P: [7, 8]

9

56

P: [9, 56]

34

32

P: [34, 32]

32

32

P: [32, 32]

3

23

P: [3, 23]

2

32

P: [2, 32]

32

332

P: [2, 332]

K-MEANS

CLUSTERS:

C: [[1, 2], [3, 4], [5, 6], [7, 8]]

C: [[9, 56], [34, 32], [32, 32], [3, 23], [2, 32]]

C: [[2, 332]]

AGGLOMERATIVE

CLUSTERS:

C: [[9, 56]]

C: [[2, 332]]

C: [[32, 32], [34, 32]]

C: [[2, 32], [3, 23], [5, 6], [3, 4], [7, 8], [1, 2]]

>>>