Danmarks Tekniske Universitet

01415
Computational Tools for Big Data

Week 2 - Exercise

Anonymous

September 2016

1 Exercise 4.1

In this exercise we have to build a program that reads "pickle" formatted files that contain highly dimensional sparse datasets in the form of compressed sparse row matrix (csr_matrix) and performs an implementation of the DBSCAN clustering algorithm working with Jaccard-distance as its metric.

The created program is divided into 3 python files:

- *pickle_lib.py*: It is a library we have created for efficiently and safely read and store "pickle" files. Since the pickle library has problems with loading big files (it runs out of RAM memory easily) we created function that check the readablity of the files and that are able to read and store Python divided in several "pickle" files.
  CDBSCAN.py: Library containing implemented functions and the class "CDBSCAN" which is a class we created to implement the DBSCAN algorithms in a better way, using the same syntax as the classes from the library scikit-learn.
- *W4.py*: It is the main file that uses the previous built libraries to load the datasets and apply the DBSCAN. algorithm.

The most important file is *CDBSCAN.py*, where the class **CDBSCAN** is defined. This class contains the data structures and methods to solve the DBSCAN clustering. We have tried to follow the main structure of the definition of the algorithm in Wikipedia but we have made some changes to improve performance. Some important keypoints to keep in mind about our implementation are:

- We work with the data using only built in compressed sparse functions, so we do not need to transform the sample points into a dense representation. More on this later when we explain how we compute the Jaccard distances. We reference points using their index in the compressed matrix, we only use the actual points when we calculate distances.
  We do not have an explicit "Visited" vector because we use the "cluster_P" vector instead, that tells us, for every point, the cluster it belongs to, if its value is -1, then it means that is either an outlier or it has not been assigned to any cluster yet.
- In the iterative process of finding the points belonging to a cluster, once we find a new Density Point, we have a queue where we keep adding all the neighbours that have not been assigned yet to any cluster and that we have not checked already (they are not already in the queue)
- There is an option in the program "reuse_computed" for storing computed Regions of unassigned points. If set to 1, when we calculate the Region of a point (Neighbours), we keep it in memory until that point is assigned to a cluster. Due to the parameters used in this assignment, it is not very useful because the Region of a points is not calculated many times but it could happen that in the worst case, it would be calculated "N" times, so storing it could save time.

The important methods of the class are:

- `def __init__(self, eps = 0.3, MinPts = 2, reuse_computed = 0):`
  This method initializes the object, assigning the configuration parameters. Ther is one additional paramter `reuse_computed` that if set to 1, then it stores the calculated regions of the visited points until they are assigned to a cluster. This way we can reuse those Regions without having to recalculate them.
  `def set_X(self, X):`
  This function intialize Global Variables for the process. The main data structures initialized here are:
  - `self.cluster_P = -1 * np.ones((self.Nsam,1))`:
    It assings to every point a cluster (-1 = Not assigned). It is used to check if we have to consider a point for a cluster. If it is noise, then we do, if not, we do not.

    `self.samples_K = []`
    Lists of lists. Each list constains the list of points belonging to cluster i
  - `self.Nelem = self.X.getnnz(axis = 1)`
    Number of elements in each sample point.
  - `self.Already_Calculated_Regions = [[-1]] * self.Nsam`
    Structure for precalculated distances. It will be reused. When a sample is assigned, then its entry is removed.

- `def regionQuery(self,i, mode = 1, only_same = 1):`
  This functions outputs the neighbour Points of Point i. With the variable `mode` we can choose the way the Region is calculated. With the variable `only_same` we can choose that we only consider neighbours those points that have not been assigned to any cluster yet or those that belong to the same cluster as point i. It is an optimization improvement.

  The first mode in which the Region can be calculated is the one we implemented first and it is very unefficient. For a given point i, identified by its index in the matrix, it converts the point to a dense array and then, for each other point, it also converts it into a dense array and it computes the union and intersection of both arrays. This method is unefficient in time and memory since it has to perform the transformation to a dense form and then operate in that domain.

  The second implementation `mode = 1` is much more efficient. It calculates the Jaccard Distante using only built in functions of the csr_matrix in the following way:

  - It obtain the insertection between the given smaple and the rest by multiplicating the sample by all samples and summing up the results.
    `inters = self.X[i,:].multiply(self.X[:,:]).sum(axis = 1)`

    It obtain the unions by means of the intersection and the number of elements of each sample. This last vector is only computed once in the initialization of the process.
    `unions = float(self.Nelem[i]) + self.Nelem - inters.T`
  - It computes the Jaccard distance to all samples in a vectorial form
    `JaDis_all = 1 - inters /unions.T`
  - It selects only those samples that are closer than the threshold (or equal) in a vectorial form.
    `RegionSamples = np.where(JaDis_all <= self.eps )[0].tolist()`

  `def fit(self, X):`
  It is the main function for performing the clustering, it starts the process by creating the data structures needed and then it starts looking for density points (that are not in a cluster yet). Whenever he finds a new Density point it call and then calls the function expandCluster() to obtain all the points belonging to that cluster.

  `def expandCluster(self, i, NeighborPts):`
  Given a DensityPoint and its neighbours, this function gets all the points that belong to the cluster. NeighborPts is the initial set of points to set as "belonging" to the cluster but if these points are also Density points then it also adds the nonvisited nodes to the queue in Breadth First Search fashion.

1.1 pickle_lib.py

```
import pickle
import gc
import os

# Library for loading and storing big amounts of data into
# different files because pickle takes a lot of RAM otherwise.
# If the number of partitions = 1, then it just loads like a
# regular pickle file.
# It uses gc also to remove garbage variables.

def store_pickle (filename, li, partitions = 1, verbose = 1):
    gc.collect()

    splitted = filename.split(".")
    if (len(splitted) == 1):  # If there was no extension
        fname = filename
        fext = ""
    else:
        fname = '.'.join(splitted[:-1])    # Name of the file
        fext = "." + splitted[-1]    # Extension of the file

    # li: List of variables to save.
    # It saves the variables of the list in "partitions" files.
    # This function stores the list li into a number of files equal to
    # If "partitions" = 1 then it is a regular load and store
    num = int(len(li)/partitions);

    if (partitions == 1):  # Only 1 partition
        if (verbose == 1):
```

```
            print "Creating file: " + fname + fext
        with open(fname + fext, 'wb') as f:
            pickle.dump(li, f)
    else:                       # Several partitions
        for i in range(partitions - 1):
            if (verbose == 1):
                print "Creating file: " + fname + str(i) + fext
            with open(fname + str(i)+ fext, 'wb') as f:
                pickle.dump(li[i*num:(i+1)*num], f)
                # We dump only a subset of the list
        # Last partition to create
        if (verbose == 1):
            print "Creating file: " + fname + str(partitions -1) + fext

        with open(filename + str(partitions - 1)+ fext, 'wb') as f:
                pickle.dump(li[num*(partitions - 1):], f)
                # We dump the last subset.
    gc.collect()

def load_pickle (filename, partitions = 1, verbose = 0):

    gc.collect()
    total_list = []
    splitted = filename.split(".")
    if (len(splitted) == 1):  # If there was no extension
        fname = filename
        fext = ""
    else:
        fname = '.'.join(splitted[:-1])    # Name of the file
        fext = "." + splitted[-1]     # Extension of the file

    if (partitions == 1):  # Only 1 partition
        if (verbose == 1):
            print "Loading file: " + fname + fext

        if (os.path.exists(fname + fext) == True):    # Check if file ex

            with open(fname + fext, 'rb') as f:
                total_list = pickle.load(f)     # We read the pickle fil
        else:
            print "File does not exist: " + fname + fext
            return []
    else:                               # Several partitions
        for i in range(partitions):
            if (verbose == 1):
                print "Loading file: " + fname + str(i)+ fext

            if (os.path.exists(fname + str(i)+ fext) == True):
# Check if file exists !!

                with open(fname + str(i)+ fext, 'rb') as f:
                    part = pickle.load(f)     # We read the pickle file
                total_list.extend(part)

            else:
                print "File does not exist: " + fname + str(i)+ fext
                return []

    gc.collect()
    return total_list

#n = 3
#lista = [10, 23, 43, 65, 34, 98, 90, 84, 98]
#store_pickle("lista",lista,n)
#lista2 = load_pickle("lista",n)
```

1.2 CDBSCAN.py

```
import numpy as np
import copy
import scipy.sparse as sps
import gc as gc
```

```
def remove_from_list(L,RL):
    # This function removes the elements RL from list L
    Lc = copy.copy(L)  # Create a copy of L
    for rl in RL:  # For every element to remove
        try:
            Lc.remove(rl)
        except ValueError:
            pass
    return Lc  # Return removed list

def Jaccard_Dis(s1,s2):
    # s1 and s2 are two sets.
    interS = list(set(s1) & set(s2))
    unionS =  list(set(s1) | set(s2))

    JaSim = float(len(interS))/len(unionS)
    JaSDis = 1 - JaSim
    return JaSDis

class CDBSCAN:
    # DBSCAN for discrite binary points
    def __init__(self, eps = 0.3, MinPts = 2, reuse_computed = 1):
        self.eps = eps
        self.MinPts = MinPts
        self.reuseC = reuse_computed

    def set_X(self, X):
        self.X = X
        self.Nsam = X.shape[0]  # Number of samples
        self.Ndim = X.shape[1]  # Number of datapoints

        ### Intialize Global Variables for the process
        # It assings to every point a cluster (-1 = Noise)
        self.cluster_P = -1 * np.ones((self.Nsam,1))
        # Number of clusters so far.
        self.K = -1;
        self.samples_K = []  # Lists of lists.
        # Each list inside constains the list of points of cluster i
        # We find new clusters when we find unassigned density points

        ################ OPTIMIZATION ###########
        # Number of elements in each sample. For obtaining the Jaccard
        self.Nelem = self.X.getnnz(axis = 1)

        if (self.reuseC == 1): # If we store precalculated distances
            self.Already_Calculated_Regions = [[-1]] * self.Nsam
            # When we calculate the region of a point, we also put here
            # If the samples is assigned to a cluster, then we remove i

        self.Nregion = 0;  # Number of times regions are calculated.
        # Just to have estimation of execution time.

    def regionQuery(self,i, mode = 1, only_same = 1):
        # This functions outputs the neighbourPoints of Point P given t
        # With the variable mode we can choose the way the Region is ca
        # With the variable only_same we can choose that we only consid
        # for the cluster, those points that havent been assigned to an

        eps = self.eps
        Nsam = self.Nsam

        if (self.reuseC == 1): # If we store precalculated distances
            ## If the region is precalculated !
            if (self.Already_Calculated_Regions[i][0] != -1):
                RegionSamples = self.Already_Calculated_Regions[i]
                print "Region Computation reused !!!"
                return RegionSamples

        # If we dont have precomputed ! We compute it :)

        self.Nregion += 1     # Regions calculated so far.
        if (self.Nregion % 50 == 0):    # Every 50 regions computed we p
            print "Nregion: " + str(self.Nregion)

        ## Firt mode to calculate the Region
        if (mode == 0):  # Not efficient mode
            RegionSamples = []
```

```
            for j in range(Nsam):                    # For every sample
                JaDis = self.JaDis_samples(i,j) # Get its metric
                if (JaDis <= eps):                  # If the point j is in the
                    RegionSamples.append(j)

        ### Efficient way to calculate the Jaccard distance
        # This function uses compresses sparse functions.
        elif (mode == 1):
            # Obtain the insertection by multiplicating the sample by a
            # and summing the results.
            inters = self.X[i,:].multiply(self.X[:,:]).sum(axis = 1)
            # Obtain the unions by means of the intersection and the nu
            unions = float(self.Nelem[i]) + self.Nelem - inters.T
            # Compute JaDis_all to all samples.
            JaDis_all = 1 - inters /unions.T
            # Only get those ones closer than the threshols
            RegionSamples = np.where(JaDis_all <= self.eps )[0].tolist(

        if (self.reuseC == 1): # If we store precalculated distances
            self.Already_Calculated_Regions[i] = RegionSamples

        if (only_same == 0): # If we just give all the neighbours
            return RegionSamples

        ##########################################################
        #####  Optimization    ###################################
        ##########################################################
        # We could retrieve only the unassigned neighbors and the neigh
        # of the same cluster as the samples. This will be the real nei
        # to consider the point as a density point.
        cluster_P_Region = self.cluster_P[RegionSamples]
#           print cluster_P_Region
        non_assigned_indx = np.where(cluster_P_Region == -1)[0]

#           print non_assigned_indx
        non_assigned_points = np.array(RegionSamples)[non_assigned_indx
        if (self.cluster_P[i] != -1):
            same_cluster_indx = np.where(cluster_P_Region == self.clust
            same_cluster_points = np.array(RegionSamples)[same_cluster_
            if(len(same_cluster_points) > 0 ):
                non_assigned_points.extend(same_cluster_points)

        return non_assigned_points


    def fit(self, X):  # Fit points X
        # This function obtaines the clusters for the compressed sparce

        self.set_X(X) # Initialize data structures

        for i in range(self.Nsam):  # For every sample
        # If point already assigned to cluster, we dont analyse it
          if (self.cluster_P[i] != -1):
              continue;  # Get out of the loop and process next point

          # Get the samples that are inside the region of the point
          NeighborPts = self.regionQuery(i)

          # If there are not enough data points
          if (len(NeighborPts) < self.MinPts):
              self.cluster_P[i] = -1;  # Mark as noise for now

          else:  # Otherwise we start a Breath First Search of the node
              self.K += 1;     # Increase the number of clusters.
              self.cluster_P[i] = self.K  # Set the point to the cluster

              if (self.reuseC == 1): # If we store precalculated distanc
                  self.Already_Calculated_Regions[i] = [-1] # Remove its
                  gc.collect()
              print "New cluster density point found: " + str(i)
              print "Obtaining cluster..."

              # Expand cluster !! Look for the nodes belonging to that c
              self.expandCluster(i, NeighborPts)
              print "Cluster finished " + str(self.K) + " found: "+ \
                  str(len(self.samples_K[self.K])) + " samples."
```

```
        if (self.K == -1):
            print "No Clusters found"

        ## Obtain the samples that outliers.
        self.K += 1
        outliers = np.where(self.cluster_P == -1)[0].tolist()
        self.samples_K.append(outliers)

    def print_clusters_sizes(self):
        # This function prints the sizes of all clusters
        size_clus = []
        for clus in self.samples_K:
            size_clus.append(len(clus))
        print "Sizes of the clusters"
        print size_clus

    def JaDis_samples (self,i,j, mode = 1):
        # Jaccard Distance between two samples i and j in an unefficien
        # self.X[i,:].todense()  # To convert to dense
        # Shouldnt be used in the final representation.
        if (mode == 0):
            set_i = np.where(self.X[i,:].toarray() == 1)[1] # Indexes o
            set_j = np.where(self.X[j,:].toarray() == 1)[1]
            JaDis = Jaccard_Dis(set_i,set_j)

        if (mode == 1):
            x1 = self.X[i,:].toarray()
            x2 = self.X[j,:].toarray()
            union = np.sum(np.logical_or(x1, x2))
            inter = x1.dot(x2.T)
            JaDis = 1- float(inter)/union
        return JaDis

    def expandCluster(self, i, NeighborPts):
        # Given a DensityPoint and its neighbours, this function
        # gets all the points that belong to the cluster.
        # NeighborPts is the initial set of points to check
       K_samples = []; # Indexes of the points that belong to the clust
       K_samples.append(i)

       N_neighbors = len(NeighborPts)
       # Number of total neighBours we have to look at.
       # If the neighbour has not been visited we add it to the cluster
       # If it is a density point, we also add its neighbours to this s
       j = 0;  # Index on the neighbors. It will iterate over new poins

       # Breath First Search of the nodes in the cluster  !!!
       while (j < N_neighbors):
          Neigh_i = NeighborPts[j]
           # For every neighbour point (sample) to the density point

          # If a neighbour is in a cluster, it means that all its neigh
          # are also already in a cluster... So no need to recalculate
          if (self.cluster_P[Neigh_i] == -1): # if P' neighbour does no
             Neighb_ij = self.regionQuery(Neigh_i)  # Obtain the neighb

             if (len(Neighb_ij) >= self.MinPts):  # If this point is al
                # We add its Neighbours to the NeighborPts.
                # Only the neighbour that we were not considering alrea
                # We get the points in common between these new neighbo
                 InterNeigh = list(set(NeighborPts) & set(Neighb_ij))
                 NewNeigh = remove_from_list(Neighb_ij,InterNeigh)
                 if (len(NewNeigh) > 0): # If new neighbors found
                     NeighborPts.extend(NewNeigh)
#                    print NeighborPts
                 N_neighbors = len(NeighborPts)  # Recalculate number o

             # if P' is not yet member of any cluster
             if ((self.cluster_P[Neigh_i]) == -1):  #
                 K_samples.append(Neigh_i)   # We assing it to the curr
                 self.cluster_P[Neigh_i] = self.K
                 # We remove the precalculated region !!
                 if (self.reuseC == 1): # If we store precalculated dis
                     self.Already_Calculated_Regions[Neigh_i] = [-1]
                     gc.collect()
          j+= 1; # Increase the index
```

```
        # Now we append all the samples of the cluster to the global st
        self.samples_K.append(K_samples)
```

1.3 W4.py

```
import pickle_lib as pklib
import CDBSCAN as CDBSCAN

problem_indx = 3
files_list = ["./data/data_10points_10dims.dat",
              "./data/data_100points_100dims.dat",
              "./data/data_1000points_1000dims.dat",
              "./data/data_10000points_10000dims.dat",
              "./data/data_100000points_100000dims.dat"]

eps_list = [0.4, 0.3, 0.15, 0.15, 0.15]
data = pklib.load_pickle (files_list[problem_indx],verbose = 1);

myDBSCAN = CDBSCAN.CDBSCAN(eps=eps_list[problem_indx], MinPts=2, reuse_
myDBSCAN.fit(data)

#print myDBSCAN.Already_Calculated_Regions[1]
print "Number of clusters " + str(myDBSCAN.K + 1)
myDBSCAN.print_clusters_sizes()
```