**DBSCAN**

Rubén Ventura

9305420195

**CONTENT**

[**SUMMARY OF THE PROJECT** 3](#_Toc41482922)

[**DESCRIPTION OF THE CODE** 4](#_Toc41482923)

[global data: 4](#_Toc41482924)

[read\_file(input\_data\_file): 4](#_Toc41482925)

[write\_files(n\_clusters, sorted\_cluster\_sets): 5](#_Toc41482926)

[set\_recursion\_limit( ): 5](#_Toc41482927)

[calculate\_distance(point\_1, point\_2): 6](#_Toc41482928)

[find\_close\_points(eps): 6](#_Toc41482929)

[get\_clusters(min\_pts): 7](#_Toc41482930)

[get\_density\_reachable\_points(point\_id, min\_pts, reachable\_points, label): 8](#_Toc41482931)

[main( ): 9](#_Toc41482932)

[**INSTRUCTIONS FOR COMPILING** 9](#_Toc41482933)

[**ADDITIONAL SPECIFICATIONS** 10](#_Toc41482934)

# **SUMMARY OF THE PROJECT**

For this project, we were asked to implement the DBSCAN algorithm in order to cluster three given datasets.

This project was not as challenging as the last one, since the algorithm itself is not that complicated as I understood the theoretical concepts beforehand, so the idea of how to develop the code was already in my mind.

The most difficulties I have had in this project are regarding the optimization of the runtime, which increases exponentially to the database size. I have tweaked the code a couple times to achieve an improvement of over 200x, downing the runtime for the first input from over 2 minutes to around 40 seconds.

For the code itself, I again chose Python 3 as the programming language and the sources I used for the code were the lecture materials and the algorithm pseudocode and explanation from the textbook.

# **DESCRIPTION OF THE CODE**

## global data:

# Global variable storing the data

data = {}

Since most functions in the program will be accessing the data read from the input files, I decided to define a variable in the global scope instead of using it as a parameter for every function. Since each point has a unique ID, I decided to store the data in a dictionary.

## read\_file(input\_data\_file):

# Reads the file and stores the first value in each line as dictionary key (as an int)

# Sets the following 2 values as float values for that key

# Prepares the third value as an empty set, which will be filled with neighbours

# Adds a fourth field, which will store to cluster it belongs to

def read\_file(input\_data\_file):

  global data

  with open(input\_data\_file, 'r') as f:

    for line in f.read().split('\n'):

      if line != '':

        object\_id, coord = line.split('\t', 1)

        data[int(object\_id)] = [float(value) for value in coord.split()]

        data[int(object\_id)].append(set())

        data[int(object\_id)].append(None)

Function used to read the data from the input files and store it in a global dictionary. Reads line by line and expects 3 values per line. The first value will be the point ID and will be used as the dictionary key. The next 2 values will be the X and Y values for the coordinates, which will be stored in the first and second position of the dictionary, respectively. Next, prepares a third field as an empty set, which will be filled with the IDs of all the neighbour points. Lastly, following the algorithm, we set each point as not visited, adding a “None” value to it, which will later be changed to the label of the cluster that point is assigned to.

## write\_files(n\_clusters, sorted\_cluster\_sets):

# Writes the largest N clusters to file (suggested optimization)

def write\_files(n\_clusters, sorted\_cluster\_sets):

  input\_file\_name = sys.argv[1].replace('.txt','')

  for i in range(n\_clusters):

    file\_name = input\_file\_name + '\_cluster\_' + str(i) + '.txt'

    with open(file\_name, 'w') as f:

      for point in sorted\_cluster\_sets[i]:

        f.writelines(str(int(point)) + '\n')

This function is used to write N output files (N value is passed as an argument when executing the program). It will create a txt file with the specified name format of “input\_file\_name + \_cluster\_ + cluster\_label” which contains a list of all the point IDs belonging to that cluster. As suggested in the project guidelines, it will only generate files for the N largest clusters found.

## set\_recursion\_limit( ):

# Sets recursion limit according to the db size

def set\_recursion\_limit():

  global data

  # Python's default recursion limit is not enough to handle big dbs)

  sys.setrecursionlimit(len(data))

This is a little helper function needed for the program to work when large databases are given as inputs, since Python has a default recursion limit which will cause the program to crash when recursion is called above a threshold. To solve this, we set the recursion limit to the same length of the database (in the worst case scenario, in which all the given points belong to the same cluster, the program will need to call recursion once for each point, so with this setting we solve this issue).

## calculate\_distance(point\_1, point\_2):

# Calculates pythagorean distance between two 2-dimensional points

def calculate\_distance(point\_1, point\_2):

  x1 = point\_1[0]

  y1 = point\_1[1]

  x2 = point\_2[0]

  y2 = point\_2[1]

  x = (x2 - x1)

  y = (y2 - y1)

  distance = math.sqrt(x\*\*2 + y\*\*2)

  return distance

This function was made for clarity. Calculates the Pyhtagorean distance between two given 2-dimensional points and returns that value.

## find\_close\_points(eps):

# Finds which points are within the minimum radius for each point

def find\_close\_points(eps):

  global data

  # Used to reduce the number of operations from n^n to n!

  last\_visited\_point = 0

  for point\_id in range(len(data)):

    for candidate\_point in range(last\_visited\_point, len(data)):

      # If we don't have the point already in the set, we check

      if candidate\_point not in data[point\_id][2]:

        distance = calculate\_distance(data[point\_id][:2], data[candidate\_point][:2])

        # If the two points are within distance, updates both sets

        if distance <= eps:

          data[point\_id][2].add(candidate\_point)

          data[candidate\_point][2].add(point\_id)

    last\_visited\_point += 1

This is one of the core functions of the algorithm, used to get all the neighbour points for each point in the database. It starts with the first point (ID = 0) and checks all the remaining unvisited points in the database (this way we reduce the number of iterations the function needs to do). For each of the candidate points, if it is not already in the neighbour list of the current point (has not been updated by a previous iteration of this function) we calculate the distances between the two points, and if they are within the defined threshold, we update them in both neighbour sets. We repeat the process for each point in the database until all points have been visited and all the neighbours have been found and assigned.

## get\_clusters(min\_pts):

# Starts clustering, starting from a random point

def get\_clusters(min\_pts):

  global data

  cluster\_sets = []

  # Initialize labels

  label\_id = 0

  # Check if point has been assigned to a cluster. If it's not, add it to

  # current cluster and repeat process for nearby points.

  # When done, change cluster label and move to next unassigned point.

  for point\_id in range(len(data)):

    if data[point\_id][3] == None:

      reachable\_points = set()

      cluster\_sets.append(get\_density\_reachable\_points(point\_id, min\_pts, reachable\_points, label\_id))

      label\_id += 1

  return cluster\_sets

Another of the main algorithm functions. Once all the neighbours for each point have been found, we proceed to label the different clusters. Starting with the first point in data, we check if the point is assigned to a cluster (by default it is not). If it is not, we start assigning all the density-reachable points to that cluster, recursively. When we are done, we proceed to the next cluster.

## get\_density\_reachable\_points(point\_id, min\_pts, reachable\_points, label):

# Returns all the density-reachable points from one point

# Recursively looks for all direct-reachable points

def get\_density\_reachable\_points(point\_id, min\_pts, reachable\_points, label):

  global data

  # Each reachable point

  for neighbour in data[point\_id][2]:

    if neighbour not in reachable\_points:

      # Add it to the set of reachable points in the cluster and label it

      reachable\_points.add(neighbour)

      data[neighbour][3] = label

      # If it's core, repeat

      if len(data[neighbour][2]) >= min\_pts:

        reachable\_points.union(get\_density\_reachable\_points(neighbour, min\_pts, reachable\_points, label))

  return reachable\_points

Last function of the main algorithm. Called in the get\_clusters( ), it searches all the direct-reachable points from one given point recursively , and effectively returns the density-reachable points. Starting from the given point, it checks all its neighbours. If they are not in the given reachable\_points set, we add them to the set and update their labels with the current cluster label. It then checks if current neighbour point is a core point itself, in which case the function is called again to repeat the process with this point. Every layer of recursion will add every new reachable point to the set and label it in the process, and when the function ends, it returns a set of all the density-reachable point.

## main( ):

def main():

  input\_data\_file = sys.argv[1]

  n\_clusters = int(sys.argv[2])

  eps = int(sys.argv[3])

  min\_pts = int(sys.argv[4])

  # Dictionary with point\_id as key, coordinates as values

  read\_file(input\_data\_file)

  # Set the new recursion limit according to the data size

  set\_recursion\_limit()

  # Get close points for each point

  print("Clustering...")

  find\_close\_points(eps)

  # Get clusters

  cluster\_sets = get\_clusters(min\_pts)

  # Sort clusters in descending length order (larger ones first)

  sorted\_cluster\_sets = sorted(cluster\_sets, reverse=True,  key=len)

  print("Completed")

  # Print the N first clusters

  write\_files(n\_clusters, sorted\_cluster\_sets)

Main function of the program. It prepares the data and executes the DBSCAN algorithm.

# **INSTRUCTIONS FOR COMPILING**

The whole program is written in Python 3.x, in a single .py file, so no compilation is needed. It expects four arguments as input, in this order: 1) input file, 2) number of clusters (n\_clusters), 3) maximum radius of the neighbourhood (eps) and 4) minimum number of points belonging to a cluster (min\_pts). So, going to the directory where the program and the input file are, the command for executing would be:

python clustering.py [input\_file] [n\_clusters] [eps] [min\_pts]

ex:

python clustering.py input1.txt 8 15 22

# **ADDITIONAL SPECIFICATIONS**

All the files share the same directory, including the program files, the input files and all the newly generated output files.