Universitat Politècnica de Catalunya

Barcelona School of Informatics

Master in Innovation and Research in Informatics

Optimization Techniques for Data Mining

- Cluster Median -

Author
Marc Mendez
Ricard Meyerhofer

Lecturer
F.-Javier Heredia

January 8, 2020

Index

1 Introduction

As we might know, clustering is a method for unsupervised learning, and a common technique applied in many fields such as machine learning or data mining. In this project, we are going to focus in a particular clustering problem which is the cluster-median problem. We are going to implement the cluster-median with AMPL and we are going to compare the results obtained, to the ones obtained by an heuristic approach based on MST solution with the Prim's algorithm that we have developed in Python.

In order to compare these two algorithms, we are going to use a given matrix $A = (a_{ij}), i = 1, ..., m, j = 1, ...n$ of m points and n variables. We decided to use two datasets to compare the Prim and cluster-median approaches with different sizes and number of variables so that we can see how each algorithm performs under different conditions.

2 Data description

As mentioned, we are going to use three different datasets: A simple dataset that we generates a dataset (we will call it Blobs) with the parameters we want, a real dataset which corresponds to the Glass dataset that contains different information regarding to the composition of 214 glasses and finally a similar version of the Glass dataset regarding the clusters and attributes but with more samples.

2.1 Blobs

This data is generated with the "dataGenerator.py" script where we chose the number of centers, instances, number of features and the deviation at each center. In the data that we are going to use to compare, we decided to generate a simple dataset with 100 instances, 4 centers, 2 centers (so that can be represented easily), and a low dispersion. Below we can see the plot of how our data looks like:

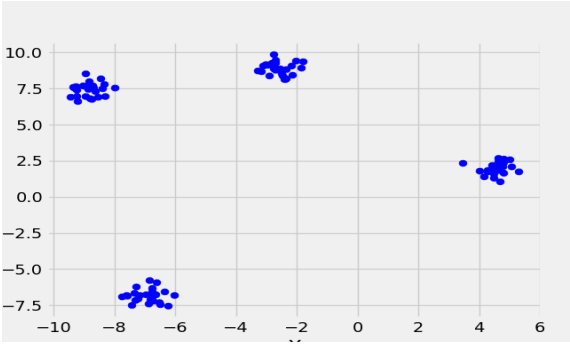

Figure 1: Blob dataset represented

2.1.1 Blobs500

In order to show how the two approaches evolve among size, we are going to create another dataset with the same code that is Blob500 which is composed by 500 samples, 6 centers and 9 features. This dataset is similar in dimensions with the Glass dataset that we will introduce now, but it has more samples.

2.2 Glass

As we previously introduced, the dataset that we are going to use, contains 214 instances of the composition of Glass with 9 attributes (we eliminated the id) and 1 response variable. Each of these samples contains the following variables:

Variable Name	**Description**
RI: Refractive index	Numeric
Na: Sodium	Numeric
Mg: Magnesium	Numeric
Al: Aluminum	Numeric
Si: Silicon	Numeric
K: Potassium	Numeric
Ca: Calcium	Numeric
Ba: Barium	Numeric
Fe: Iron	Numeric
Type of glass	Factor (1-7)

Table 1: Glass dataset attributes

This dataset contains no NA's and it has an application in determining for instance the glass from a crime scene. In order to contextualize a bit more the dataset, we are going to specify the factors where 1 corresponds to building_windows_float_processed, 2 building_windows_non_float_processed, 3 vehicle_windows_float_processed, 4 vehicle_windows_non_float_processed (none in this database), 5 containers, 6 tableware and 7 headlamps.

If we take a look at the data, we can see from a quick summary that there is very few variability in some of the attributes and also they are highly correlated. Which will probably affect to our accuracy.

3 Cluster-median

In the cluster-median approach we have that the criteria for similarity is that the overall distance of all the points to the median of the clusters that they belong to is minimized. The median for a subset of points $I \subseteq 1, ..., m$ is defined as the nearest point of all points of I:

$$\text{r is the median of I if } \sum_{i \epsilon I} d_{ir} = min_{j \epsilon I} \sum_{i \epsilon I} d_{ij} \tag{1}$$

where $D = d_{ij}, i = 1, ..., m, j = 1, ..., m$ is the matrix of distances for each pair of points. In our particular case, we used the Euclidean distance to compute D from A.

3.1 Formulation

For the sake of simplicity, instead of considering k clusters in our formulation, we are going to consider m clusters. Therefore, we are going to have $m - k$ of empty clusters. We have that given an element i:

$$\forall i, j \in 1, ..., m, \tag{2}$$

$$x_{ij} = \begin{cases} 1, & \text{if element } i \text{ belongs to cluster-j} \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

where the cluster whose median is the element j is denoted as cluster-j

We have that our objective function, as aforementioned, is to minimize the distance of all points to their cluster medians

$$\min \sum_{i=1}^{m} \sum_{j=1}^{m} d_{ij} x_{ij} \tag{4}$$

$$\tag{5}$$

To achieve our objective function and to solve the problem, we need to impose 3 conditions:

$$\sum_{j=1}^{m} x_{ij} = 1 \; i = 1, ...m \qquad \text{[Every point belongs to one cluster]} \tag{6}$$

$$\sum_{j=1}^{m} x_{jj} = k \qquad \text{[Exactly k clusters]} \tag{7}$$

$$m x_{jj} \sum_{i=1}^{m} x_{ij} \; j = 1, ..., m \quad \text{[A point belongs to a cluster iff the cluster exists]} \tag{8}$$

4 Minimum Spanning Tree

To compute the heuristic solution we have used Python. In this script we used some libraries which we will only introduce the 2 most important ones, we used numpy for array navigation and data reading and also used networkx for graph generation and treatment.

To introduce, a minimum spanning tree also known as MST, is a subset of edges which minimizes the total weight of the edges, this implies that there will not be cycles. In our particular case, when talking of weights it can be seen as distances. So our MST, will be a way of connecting all vertices with the minimum total distance. To calculate the Minimum Spanning Tree there are multiple algorithms, in our example, we used Prim's algorithm.

The algorithm works as following, we select an starting node and add it to S. Then we add the edge that connects a node belonging to S with a node that don't. After it, we add the node and the edge to the MST and repeat the step until having all nodes in S. We have to repeat this step until we

reach all the nodes of the graph. The figure below explains quite well how to compute it.

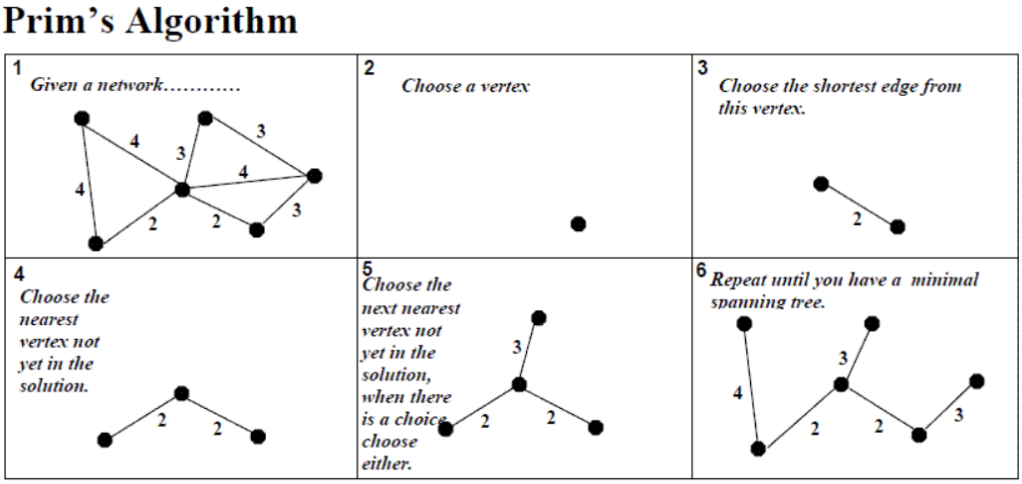

Figure 2: Prim's pseudocode

5 Conclusions

We have compared both executions with both datasets and we have seen that as expected the Integer optimization problem that we have solved with AMPL, obtains a better objective function than the MST heuristic. This is because the cluster-median is performing an integer optimization where our objective is to minimize the distance of all points to their cluster medians whereas the MST approach is performing a simpler approach by eliminating with the Prim's algorithm the k connections with higher weight. This is not better nor worse, as we can see in the executions, the optimization takes way longer than the heuristic which also was something we would expect. Below we can see the execution results:

Method	Dataset	Seconds	Objective Function
AMPL	Blobs	≈ 1.95	≈ 53.08447502
MST	Blobs	≈ 0.149	≈ 53.084475016
AMPL	Glass	≈ 61.37	≈ 215.9692
MST	Glass	≈ 1.3029	≈ 381.3650
AMPL	Blobs500	≈ 32994.9	≈ 673.251
MST	Blobs500	≈ 23.855	≈ 900.246

Table 2: Data comparing different metrics of the heuristic and ILP

What we can see, is that as the size of the dataset increases, the heuristic performs worse and the time increases linearly to the size. This is not the case of the integer optimization where we can see that the time dramatically increases where with 200 instances, it takes more than a minute to give a solution (which is optimal, unlike the heuristic) and with Blobs500 it takes around 9hours. If we put more attention to the differences between both approaches as the size increases, we can see that the number of clusters with single elements increases. For instance, when we perform with our Blobs data, we have a clear assignation of clusters:

```
set([0, 4, 15, 16, 18, 19, 21, 22, 27, 28, 33, 40, 47, 48, 54, 55,
    62, 64, 65, 67, 72, 75, 77, 83, 98])
set([1, 3, 5, 11, 12, 29, 30, 32, 42, 45, 46, 53, 60, 63, 66, 69,
    71, 74, 76, 78, 79, 81, 87, 91, 93])
set([2, 9, 10, 14, 20, 25, 34, 35, 36, 37, 43, 44, 56, 57, 59, 73,
    82, 84, 85, 86, 88, 92, 94, 95, 96])
set([6, 7, 8, 13, 17, 23, 24, 26, 31, 38, 39, 41, 49, 50, 51, 52,
    58, 61, 68, 70, 80, 89, 90, 97, 99])
```

while if we see how our distribution is for the glass dataset, we can see that there are a lot more clusters of 1-2 elements:

```
set([0, 1, 2, ... , 213])
set([106])
set([107])
set([171, 172])
set([184])
set([201])
```

If we plot the results that we have obtained, we can see that as the size increases and the problem is more complex, the gap between the optimal and the heuristic, tends to increase. Moreover, what we clearly see is that the time grows as aforementioned in a exponential manner.

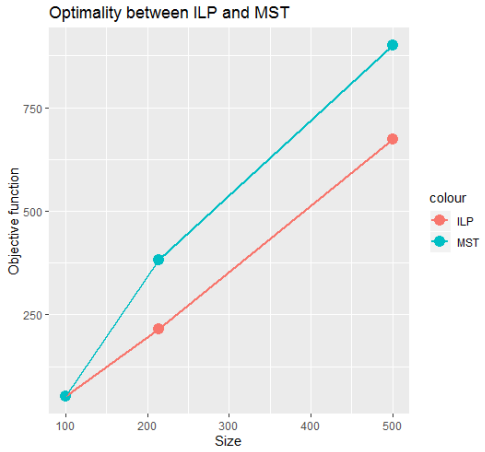

Figure 3: Optimality among size

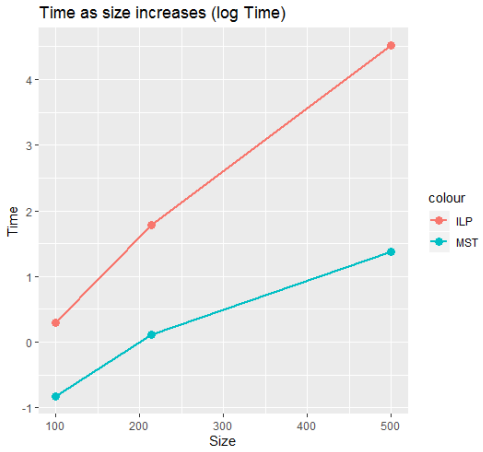

Figure 4: Time among size

6 AMPL Code

```
#Run
reset;
cd('C:/Users/Meyerhofer/Desktop/UNI/OTDM/cluster-median-project/');
model 'code/primal.mod';
data 'code/data.dat';
#display nu;

option solver "ampl_mswin64/cplex";
solve;

display x > 'data/outputs/x.txt';

var total{i in 1..m};

for{i in 1..m} {
   for{j in 1..m}{
      let total[i] := total[i] + x[j,i];
   }
}

display total > 'data/outputs/total.txt';
display _total_solve_elapsed_time;
#.Mod
param m;
param n;
param k;

param A{i in 1..m, j in 1..n};
param D{i in 1..m, j in 1..m};

var x{i in 1..m, j in 1..m} binary;

minimize obj_function:
   sum{i in 1..m, j in 1..m} D[i,j]*x[i,j];

#every point belongs to one cluster
subject to Constraint1 {i in 1..m}:
```

```
   sum{j in 1..m} x[i,j] = 1;

#exactly k clusters
subject to Constraint2:
   sum{i in 1..m} x[i,i] = k;

#a point may belong to a cluster iff cluster exists
subject to Constraint3 {j in 1..m}:
   m*x[j,j] >= sum{i in 1..m} x[i,j];
#.Dat
data;

read m < "data/inputs/m.txt";
read n < "data/inputs/n.txt";
read k < "data/inputs/k.txt";

read {i in 1..m, j in 1..n} A[i,j] < "data/inputs/A2.txt";
read {i in 1..m, j in 1..m} D[i,j] < "data/inputs/D2.txt";
```

7 Python Code

7.1 Prim's Algorithm

```
import sys
import os
import numpy as np
import networkx as nx
from networkx import Graph
import matplotlib.pyplot as plt
from scipy.spatial import distance_matrix
from scipy.spatial.distance import pdist
from scipy.spatial.distance import squareform

import time

def takeDistance(element):
   return element[2]
```

```
def connected_components(G):
    seen = set()
    for v in G:
        if v not in seen:
            c = set(_plain_bfs(G, v))
            yield c
            seen.update(c)



def number_connected_components(G):

    return sum(1 for cc in connected_components(G))

def is_connected(G):

    if len(G) == 0:
        raise nx.NetworkXPointlessConcept('Connectivity is undefined
            ',
                                          'for the null graph.')
    return sum(1 for node in _plain_bfs(G, arbitrary_element(G))) ==
        len(G)


def node_connected_component(G, n):

    return set(_plain_bfs(G, n))

def _plain_bfs(G, source):
    """A fast BFS node generator"""
    G_adj = G.adj
    seen = set()
    nextlevel = {source}
    while nextlevel:
        thislevel = nextlevel
        nextlevel = set()
        for v in thislevel:
            if v not in seen:
                yield v
                seen.add(v)
```

```
                nextlevel.update(G_adj[v])


class PrintGraph(Graph):
    """
    Example subclass of the Graph class.

    Prints activity log to file or standard output.
    """

    def __init__(self, data=None, name='', file=None, **attr):
        Graph.__init__(self, data=data, name=name, **attr)
        if file is None:
            import sys
            self.fh = sys.stdout
        else:
            self.fh = open(file, 'w')

    def add_node(self, n, attr_dict=None, **attr):
        Graph.add_node(self, n, attr_dict=attr_dict, **attr)
        self.fh.write("Add node: %s\n" % n)

    def add_nodes_from(self, nodes, **attr):
        for n in nodes:
            self.add_node(n, **attr)

    def remove_node(self, n):
        Graph.remove_node(self, n)
        self.fh.write("Remove node: %s\n" % n)

    def remove_nodes_from(self, nodes):
        for n in nodes:
            self.remove_node(n)

    def add_edge(self, u, v, attr_dict=None, **attr):
        Graph.add_edge(self, u, v, attr_dict=attr_dict, **attr)
        self.fh.write("Add edge: %s-%s\n" % (u, v))

    def add_edges_from(self, ebunch, attr_dict=None, **attr):
        for e in ebunch:
```

```
            u, v = e[0:2]
            self.add_edge(u, v, attr_dict=attr_dict, **attr)

    def remove_edge(self, u, v):
        Graph.remove_edge(self, u, v)
        self.fh.write("Remove edge: %s-%s\n" % (u, v))

    def remove_edges_from(self, ebunch):
        for e in ebunch:
            u, v = e[0:2]
            self.remove_edge(u, v)

    def clear(self):
        Graph.clear(self)
        self.fh.write("Clear graph\n")

def main():
   start_time = time.time()
   #Reads d and a
   d = np.loadtxt("../data/inputs/D.txt")
   a = np.loadtxt("../data/inputs/A.txt")
   vertex = 100
   numClusters = 4
   T = set()
   S = set()

   #Prims algorithm:
   S.add(0)
   while len(S) != vertex:
      edges = set()
      minEdge = [99999,99999,99999]
      for x in S:
         for k in range(vertex):
            if k not in S and d[x][k] != 0:
               if d[x][k] < minEdge[2]:
                  minEdge[0] = x
                  minEdge[1] = k
                  minEdge[2] = d[x][k]

      T.add((minEdge[0], minEdge[1], minEdge[2]))
```

```
    S.add(minEdge[1])
#we drop the k-1 first elements which will lead to disconnect the
    graph into clusters:
sortedEdges = sorted(T, reverse = False,
    key=takeDistance)[0:(vertex-numClusters)]

#we create a graph containing x-y-dist
G = Graph()
for i in range(vertex):
    G.add_node(i)
for i in sortedEdges:
    G.add_edge(i[0], i[1], weight=i[2])

clusters = nx.connected_components(G)
print(d)
#we calculate per each cluster the obj function
iterator = 0
medoids = []
objectiveFunction = 0
for c in clusters:
    points = []
    distancesMatrix = []
    for i in c:
        #Agafa les dades del cluster
        points.append([a[i][0], a[i][1]])
    #calcules la matriu de distancies D
    mean = np.mean(points,axis=0)

    distancesMatrix = distance_matrix([mean],np.array(points))
    print(distancesMatrix)
    minValue = np.array(distancesMatrix[0]).argmin()
    #el obj value es la suma d'aquests valors
    connectedComponentsList = list(c)
    #print(connectedComponentsList)
    for i in c:
        objectiveFunction += d[connectedComponentsList[minValue]][i]

    iterator +=1
```

```
    print(objectiveFunction)



    end_time = time.time()
    print(end_time-start_time)


if __name__ == '__main__':
    main()
```

7.2 Data generation

```
# Creating Test DataSets using sklearn.datasets.make_blobs
from sklearn.datasets.samples_generator import make_blobs
from matplotlib import pyplot as plt
from matplotlib import style
import numpy as np

style.use("fivethirtyeight")


X, y = make_blobs(n_samples = 500, centers = 6, cluster_std = 0.45,
    n_features = 9, random_state=42)

plt.scatter(X[:, 0], X[:, 1], s = 40, color = 'b')
plt.xlabel("X")
plt.ylabel("Y")

#plt.show()
#plt.clf()
np.set_printoptions(threshold=np.inf)
np.savetxt("../data/inputs/A3.txt", X)
```