A framework to solve clustering problems through meta-heuristics

INF 2102 Final project

Daniel Lemes Gribel
dgribel@inf.puc-rio.br

Pontifical Catholic University of Rio de Janeiro

June 27, 2016

Abstract

System designed to solve the task of clustering real data through genetic algorithms. In this system, users specify some desired parameters – as the dataset, the optimization solver (which defines the clustering objective function), the number of desired clusters – and the system delivers a solution for the clustering problem.

Contents

1 Program specification

In this work, a system designed to help users in the task of clustering real data is proposed. The clustering task plays an important role in data mining, being useful in many fields as information retrieval, document extraction, image segmentation and in many domains that deal with exploratory data analysis, as medicine, engineering, logistics and biology.

User-program interactions: In this system, users specify some desired parameters – as the dataset, the kind of solver (which defines the clustering objective function), the number of desired clusters – and the system delivers a solution for the clustering problem.

Objective: The main goal of this project is to provide a tool that supports datasets from different domains and delivers a good clustering solution according to parameters and objectives defined by the user. Thus, we propose a robust framework with components that can be activated according to user-defined specifications, meeting in its core a genetic algorithm to solve the clustering problem. To our knowledge, only few genetic algorithms were proposed for this purpose. The final output must display the quality of the found solution (its cost) and a metric that indicates its accuracy when compared to the real classification (labels).

1.1 Requirements

Some requirements can be extracted from the proposed system. Here we divide them in two categories: functional and non-functional requirements.

1.1.1 Functional requirements

1. The user should specify the dataset used for clustering.
2. The user should specify the desired optimization solver (algorithm).
3. The user should insert the number of desired clusters (groups).
4. The number of desired clusters (m) must not exceed the number of points (n) in the dataset ($m \leq n$).
5. The system should support an user own dataset, provided that it met the formatting specifications (csv file). See the file format in section 5.
6. The program must deliver a solution where each point (element) of the dataset is assigned to exactly one cluster (group).
7. The program must display the total cost of the generated solution.
8. When applicable, i.e., when the dataset provides the labels, the program must display a metric that indicates the accuracy of the generated solution when compared to the real classification (labels).

1.1.2 Non-functional requirements

1. The total number of iterations inside the main loop of the program must not exceed 1000 iterations.
2. The number of iterations without improvement inside the main loop of the program must not exceed 500 iterations.
3. The solution cost delivered by the program must be displayed in a 15-decimal precision.

2 Project

2.1 Structural modelling

Figure 1 represents the classes considered in the system and how they are related to each other. For a selected dataset, an object of class `Instance` is created. It represents the properties of a dataset and defines its structure, by storing the number of elements, number of clusters, number of dimensions (attributes) for each point, etc. For each instance, an object of class `DataFrame` is associated. After the dataset loading, it stores some useful data, as a similarity matrix and a list of closest points.

The `Solver` class is a super-class which is defined to solve the optimization (clustering) problem. Thus, it is associated to a space of points defined in `DataFrame` and stores a solution, its cost, and the cardinality of each cluster (group) of the solution.

From the project criteria perspective, we can observe that **inheritance** and **polymorphism** were adopted in the module that defines the solvers (see figure 1). Methods `localSearch()` and `calculateCost()` have their own implementation on each class that inherit `Solver`. Analogously, methods `updateCentroidsRelocate()`, `updateCentroidsSwap()`, `createCenters()`, `relocate()` and `swap()` have their own implementation on each class that inherits `KCenterSolver`. Below, some important remarks regarding the solvers:

`KCenterSolver`: This class represents a centroid-based optimization solver. It is an abstract class child of `Solver` and parent of `KMeansSolver`, `KMediansSolver` and `KMedoidsSolver` classes, which have their own implementation according to their objective functions. In a general manner, given an initial solution, a centroid-based solver applies local improvements in order to minimize the distance of each point to the correspondent centroid – `KMeansSolver`, `KMediansSolver` and `KMedoidsSolver` define how the centroids are calculated.

`KMeansSolver`: This class represents an optimization solver that considers the mean point of each cluster as a centroid. Given an initial solution, it applies local improvements in order to minimize the distance of each point to the mean point inside the cluster. The mean point inside a cluster is obtained by calculating the mean value for each feature among all points belonging to the cluster. Thus, the centroid is not likely to be a representative point (point belonging to the dataset).

`KMediansSolver`: This class represents an optimization solver that considers the median point of each cluster as a centroid. Given an initial solution, it applies local improvements in order to minimize the distance of each point to the median point inside the cluster. The median point inside a cluster is obtained by calculating the median value for each feature among all points belonging to the cluster. Thus, the centroid is not likely to be a representative point (point belonging to the dataset).

`KMedoidsSolver`: This class represents an optimization solver that considers the medoid point of each cluster as a centroid. Given an initial solution, it applies local improvements in order to minimize the distance of each point to the medoid point inside the cluster. The medoid point inside a cluster is the representative (point belonging to the dataset) that minimizes the distance for each other point within the cluster.

`CsgSolver`: This class represents an optimization solver that considers the total within clusters distances as the objective for clustering. Given an initial solution, it applies local improvements aiming to minimize the total within group distances, i.e., the sum of distances of all points belonging to the same cluster.

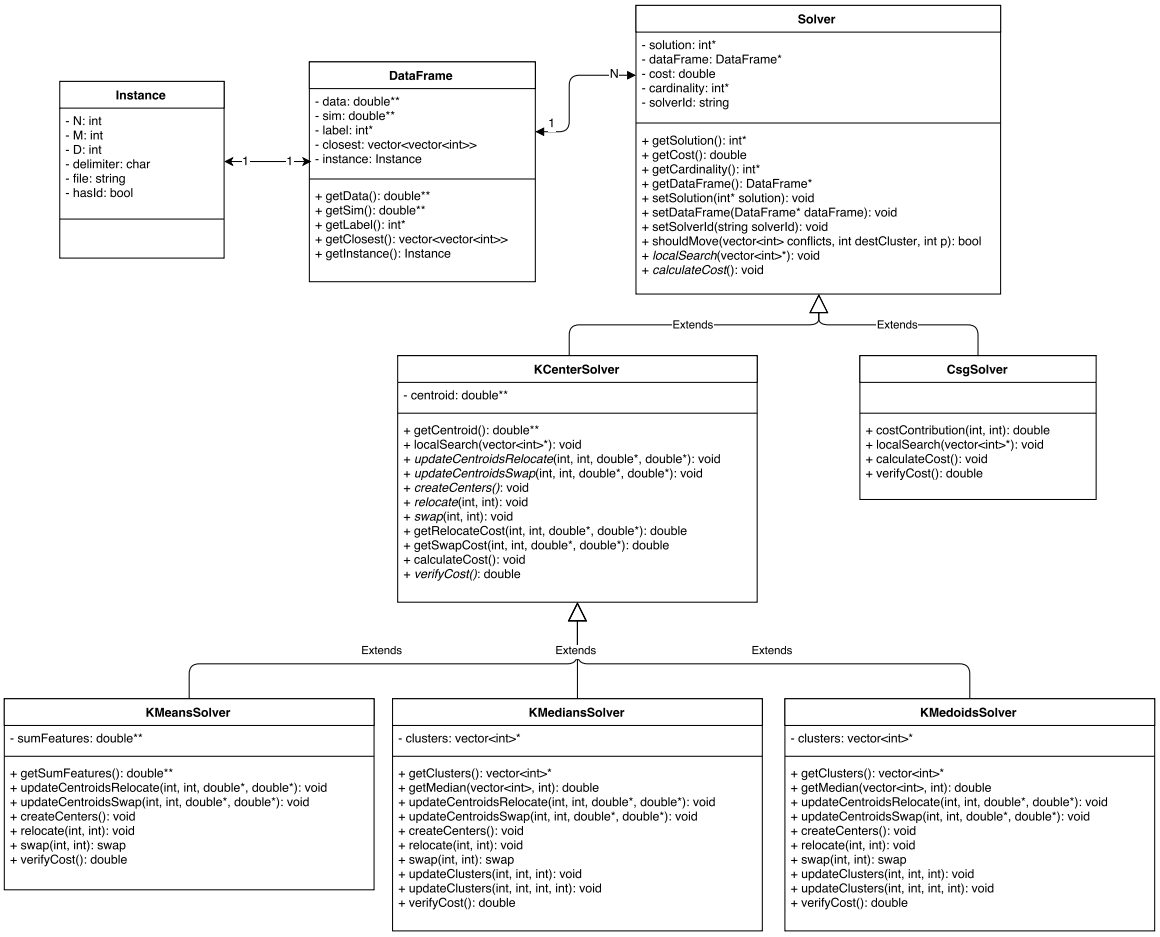

Figure 1: Classes diagram

3 Source code documentation

Please, see the source files placed in the same folder of this document.

Alternatively, access: https://github.com/danielgribel/ga-clustering.

4 Tests

Basically, two tests were performed in order to ensure the correctness of the program:

- **Cost test**: Verify the cost of a solution from scratch. This test check if the generated cost for a solution is correct. If the calculated cost is different then the one generated by the algorithm, then the test returns `false`. Otherwise, it returns `true`.
- **Cardinality test**: Verify the cardinality (number of elements) of each cluster. If any cluster is empty, then we have a invalid solution, and the test returns `false`. Otherwise, it returns `true`.

Class `Test` contains the tests implementation (see section 3). Below, a log obtained from the tests generated for 4 runs, considering different datasets, solvers and number of clusters:

```
>> Tests:
Dataset        n         m         Solver     Cost        Cardinality
fisher.csv     150       10        mean       true        true

>> Tests:
Dataset        n         m         Solver     Cost        Cardinality
glass.csv      214       6         median     true        true

>> Tests:
Dataset        n         m         Solver     Cost        Cardinality
liver.csv      345       10        medoid     true        true

>> Tests:
Dataset        n         m         Solver     Cost        Cardinality
pima.csv       768       2         csg        true        true
```

5 User documentation

5.1 Dataset format

In order to load an input file (dataset) for the clustering task, the following format is required:

$$
\begin{array}{cccccccc}
\text{n} & \text{d} & & & & & & \\
f_{11} & f_{12} & f_{13} & f_{14} & \cdots & f_{1d} & c_1 \\
f_{21} & f_{22} & f_{23} & f_{24} & \cdots & f_{2d} & c_2 \\
f_{31} & f_{32} & f_{33} & f_{34} & \cdots & f_{3d} & c_3 \\
f_{41} & f_{42} & f_{43} & f_{44} & \cdots & f_{4d} & c_4 \\
\cdots & & & & & & \\
f_{n1} & f_{n2} & f_{n3} & f_{n4} & \cdots & f_{nd} & c_n
\end{array}
$$

where n is the number of points in the dataset (dataset size), d is the number of dimensions (features) of the points, f_{ij} is the value of the j-th feature of the i-th point and c_i is the class (label) of the i-th point.

5.2 Program building

To build the program, just go to `/src` folder and type:

```
make
```

This command compiles all `.cpp` files. Alternatively, if the user uses an integrated development environment (IDE), it is possible to import the `/src` folder and then build it.

5.3 Program runnning

The program is designed to take commands from standard input. It is expected that the user passes three parameters. To run it, type:

```
./Driver input_file solver m
```

where `input_file` is the file name of the dataset, `solver` is the optimization solver used to perform the clustering task and `m` is the number of desired clusters. Alternatively, if the user uses an integrated development environment (IDE), it is possible to run the `./Driver.cpp` file with the three parameters. Important:

- The input file (dataset) must be placed on `/data` folder.
- The `solver` parameter admits 4 values: `mean`, `median`, `medoid` and `csg`, which are the commands to call `KMeansSolver`, `KMediansSolver`, `KMedoidsSolver` and `CsgSolver` respectively.

Here is an example on running the program:

```
./Driver fisher.csv csg 3
```

Then, the program will start and validate the inputs. If some input parameter given by the user is incorrect or inconsistent, the program will print a message informing where the problem is. For example, if the user enter a file that does not exist, the program will alert:

```
Error: File does not exist or is corrupted.
```

and then finish its execution. Otherwise, if everything is ok regarding the parameters, the program will run and at the end of processing will display the results. Here we have an example:

```
>> Best solution found:
0 2 2 2 2 5 2 5 2 5 5 5 5 5 5 5 5 0 0 2 5 0 5 5 2 5 2 5 5 5 5 5 5 5 5 2 0 5 0 0
5 5 5 0 5 0 5 0 0 0 0 5 5 5 5 5 5 5 2 2 2 0 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 5 5 5 5 2 5 2 5 5 5 5 5 5 5 0 0 4 4 4 4 4 4 4 4 2 2 2 2 2 2 2
2 2 2 2 2 2 5 0 0 4 4 4 2 2 2 2 5 5 5 5 2 2 5 2 5 5 2 2 2 5 2 0 2 2 5 5 2 0 2 2
2 2 0 3 5 4 4 4 4 4 4 4 3 3 4 5 4 0 0 0 0 2 1 4 4 1 3 3 2 0 0 5 1 1 1 1 1 1 1 1
1 4 1 1 1 1 1 1 1 1 1 1 1 1 1

>> Summary of results:
Dataset         n      m     Solver     Obj. function     C-rand     Time (s)
glass.csv       214    6     median     217.36913         0.2311     269.1
```

Fields `Obj. function` and `C-rand` are the final outputs of the program. They represent the value of the objective function (cost) of the solution and the partitioning accuracy, respectively.