Library Evolutionary Algorithms for Clustering (LEAC)

User guide
for LEAC version 1.8
7 March 2016

Hermes Robles-Berumen,
Amelia Zafra,
Sebastian Ventura.

This user guide is for Library LEAC (version 1.8, 7 March 2016), and documents commands for clustering analysis.

Table of Contents

1 Introduction

Library Evolutionary Algorithms for Clustering (LEAC) is a library for the implementation of evolutionary algorithms (EA) to solve the problem of *partitional clustering*, the purpose is to find an optimal partition of a data set X having n $d-$dimensional *objects*[1]. $X = \{x_1, x_2, ..., x_n\}$ in k subsets $C_1, C_2, ..., C_k$ $(k \leq n)$. So that the objects that are in the same group are more similar between them and the objects of the other groups are the most different. Formally under the partitioning approach the problem of clustering is defined by Nanda and Panda [NP14] as

$$\begin{aligned} C_j \neq \emptyset \quad & \forall j = 1, 2....k; \\ C_j \cap C_{j'} = \emptyset \quad & \forall j, j' = 1, ..k \quad \text{and} \quad \cup_{j=1}^{k} C_j = X. \end{aligned} \tag{1.1}$$

Clustering is useful in areas, such as exploratory *pattern-analysis*, *grouping*, *decision-making*, and *machine-learning* and *data mining* [JMF99].

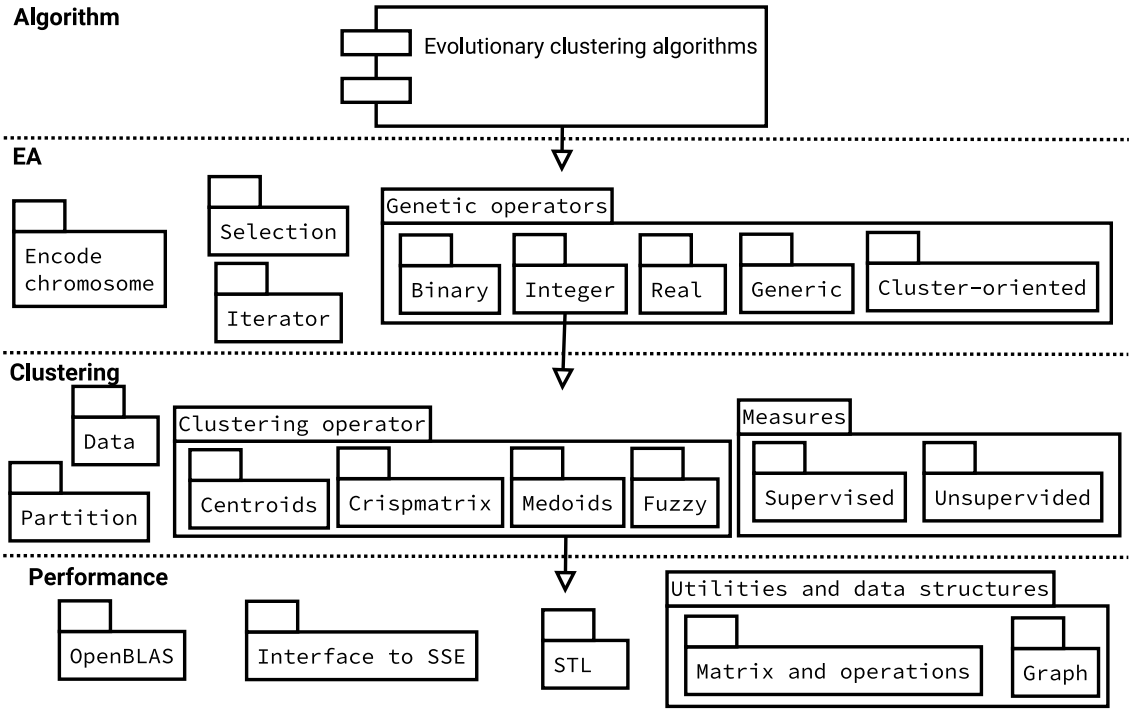

Figure 1.1: *Layered software architecture* of LEAC library

[1] The terms *object*, *instance*, *points* or *prototype* usually have the same meaning in the literature on *clustering analysis* and will be freely interchanged in this document

LEAC library it is based on a *layered software architecture* and is conceptually composed of four layers, each of which consists of a set of related packets as shown in the Figure 1.1. The description of each layer is described below:

`Algorithm`
: It consists of evolutionary algorithms, which use the layer based on the operator EA.

`EA`
: Implement what you need to configure EAs: encoding criterion, initialization of population, criterion for selecting parents, crossover and mutation operators. Along with the support of the lower layers to achieve scheme of evolution.

`Clustering`
: It refers to the domain of the problem and implements all specic clustering operators, such as, supervised and unsupervised performance measures and clustering operators based on centroids, crispmatrix or medoids. It is the layer referring to the domain of the problem.

`Performance`
: It consists of low-level functions programmed under the current CPU architectures. For example, Data Alignment and Streaming SIMD Extensions (SSE) [Int10]. It allows top layers to work with high performance.

LEAC is a modular library which make easier to develop new evolutionary algorithm proposals for solving the partitional clustering. Moreover, it contains the most representative proposals of Evolutionary Algorithms for partitional clustering. Following the taxonomy set by Hruschka et al. [HCFdC09] a first classification of these algorithms is carried out according to the use of fixed or variable number initial of clusters, as well as to the representation of the clusters of a data set. Concretely, 20 state-of-the-art paper on EA were studied, of which 25 computer programs were implemented in these paper, see Table 1.1.

LEAC is based on the current standards of the C++ language, as well as on Standard Template Library (STL) and also OpenBLAS to have a better performance. Taking advantage of the characteristics of the C++ language as hybrid language, generic programming, multi-paradigms and lambda function and the C++11 versions and C++14. It allowed the implementation of different evolutionary algorithms. The approach of LEAC to implement the particular characteristics of each algorithm is to encode the diversity of the proposed genetic operators and to use the containers and interpliers of STL to evolve the population according to the flowchart of the algorithms.

Encode	**Fixed-K**	**Variable-K**
Label	1. `gaclustering_fklabel`, Murthy and Chowdhury [MC96] 2. `gka_fklabel`, Krishna and Murty [KM99] 3. `igka_fklabel`, Lu et al. [LLF+04b] 4. `fgka_fklabel`, Lu et al. [LLF+04a]	14. `gga_vklabeldbindex` and 15. `gga_vklabelsilhouette`, Agustín-Blas et al. [ABSSJF+12] 16. `cga_vklabel`, Hruschka and Ebecken [HE03] 17. `eac_vklabel`, Hruschka et al. [HCdC06] 18. `eaci_vklabel`, 19. `eacii_vklabel`, 20. `eaciii_vklabel` and 21. `feac_vklabel`, Alves et al. [ACH06]
Crisp matrix	5. `gaclustering_fkcrispmatrix`, Bezdek et al. [BBHB94]	
Centroids	6. `gas_fkcentroid`, Maulik and Bandyopadhyay [MB00] 7. `kga_fkcentroid`, Bandyopadhyay and Maulik [BM02a] 8. `gagr_fkcentroid`, Chang et al. [CZZ09] 9. `cbga_fkcentroid_int` and 10. `cbga_fkcentroid`, Fränti et al. [FKKN97]	22. `gcuk_vkcentroid`, Bandyopadhyay and Maulik [BM02b] 23. `tgca_vkcentroid`, He and Tan [HT12]
Medoid	11. `gaprototypes_fkmedoid`, Kuncheva and Bezdek [KB97] 12. `hka_fkmedoid`, Sheng and Liu [SL04] 13. `gca_fkmedoid`, Lucasius et al. [LDK93]	
Tree		24. `gaclustering_vktreebinary`, Casillas et al. [CdLM03]
Sub-cluster		25. `clustering_vksubclusterbinary` Tseng and Yang [TY01]

Table 1.1: List of evolutionary algorithms implemented with the LEAC library, described by the taxonomy base on Hruschka et al. [HCFdC09]

2 Get and Install LEAC software

For Windows® systems, perform steps 1 through 6 and 11. For GNU/Linux systems and Mac OS X®, perform steps 1, 2 and 7 through 11.

1. Download the leac project from https://github.com/kdis-lab/leac.
2. Unzip the file `leac.zip`, we recommend you in the directory `c:\leac` for Windows® and `/home/user/leac` for GNU/Linux and Mac OS X®. Verify that the following directories exist within the main directory.

   `bin` This is the directory to store all evolutionary algorithms for clustering (EAC) binary or executable programs, which result from the compilation of using LEAC.

   `data` Directory used to store the data sets to be processed.

   `doc` In the '`doc`' directory you will find all the necessary documentation for the use of LEAC.

   `eac` It contains the source files of the implementations of the EAC algorithms implemented by the LEAC library.

   `include` Contains LEAC library header files and source code.

   `include_inout` Contains the modules for the input of parameters and output of the EAC programs.

   `openblas` Contains only the header files needed to compile a program with some functionality of the OpenBlas library.

   `sse_kernel` `sse_kernel` is a module based on OpenBlas and GotoBLAS2, own of LEAC. The functionality of this module together with that of OpenBlas is the best performance for the processing of high-dimensional data sets. For now it only works for `x86-64` architecture.

   For GNU/Linux and Mac OS X® go to [Step 7], page 6.
3. Download and install the latest version of the tdm-gcc compiler. In the step choose the components expand `gcc` and select `openmp` and also select the option Add to PATH, as shown Figure 2.1.
4. Download and install gnuplot. The recommendation for the installation is to use the file gp530-20170911-win64-mingw.zip. Unzip in the `c:\gnuplot` directory and add `c:\gnuplot\bin` to the `PATH` environment variable using the following instructions:

   Warning: Adding entries to the `PATH` is normally harmless. However, if you delete any existing entries, you may mess up your `PATH` string, and you could seriously compromise the functioning of your computer. Please be careful. Proceed at your own risk.

   a. Right-click on your `My Computer` icon and select `Properties`.
   b. Click on the `Advanced` tab, then on the `Environment Variables` button (Figure 2.2).

      You should be presented with a dialog box with two text boxes. The top box shows your user settings. The `PATH` entry in this box is the one you want to

modify. Note that the bottom text box allows you to change the system `PATH` variable. You should not alter the system path variable in any manner, or you will cause all sorts of problems for you and your computer.

c. Click on the `PATH` entry in the TOP box, then click on the `Edit` button

d. Scroll to the end of the string and at the end add '`c:\gnuplot\bin`'

e. press `OK -> OK -> OK` and you are done.

5. Optionally install the epsviewer file viewer, to visualize the data sets and the clusters created by the different programs

6. With the compiler installed see [Step 3], page 5, you can now compile the EAC applications. Open a `cmd`, you must change the directory to the `leac` directory, For example: '`cd c:\leac\eac`' and execute any of the following three options:

'`mingw32-make -k -f Makefile DEBUG=yes VERBOSE=yes`'
: To debug and analyze the detailed execution of the programs. These options allow software engineering ilities of correctness and reliability.

'`mingw32-make -k -f Makefile DEBUG=no VERBOSE=no WITHOUT_PLOT_STAT=no`'
: Optimize and obtain the evolutionary behavior of the population from the fitness function (option `WITHOUT_PLOT_STAT`) in the execution of the programs.

'`mingw32-make -k -f Makefile DEBUG=no VERBOSE=no WITHOUT_PLOT_STAT=yes`'
: Versions of optimized programs to have a good performance in the data set processing.

The compilation time of all programs varies according to the capabilities of the computers, but it can be approximately 20 minutes.

To install the applications '`mingw32-make -k -f Makefile install`'

And to eliminate the applications and use another option '`mingw32-make -k -f Makefile clean`'

Go to step [Step 11], page 8.

7. Verify that the compiler is installed, on `terminal` type, if you do not install the missing packages as system administrator (root):

'`gcc -v`' (>=4.8.5),

'`g++ -v`' (>=4.8.5),

'`make -v`' (>=4.0),

If you can not find the packages, install the missing ones as system administrator (root), with your package manager.

For GNU/Linux, e.g. run '`apt-get install gcc-4.9 g++-4.9 make`' or '`zypper install gcc gcc-c++ make`'

For Mac OS X®, if you do not have a version (> = 4.8.5), install through `MacPorts` or `Homebrew`. The procedure using `MacPorts` is described below:

a. Running in a terminal '`xcode-select --install`' and '`sudo xcodebuild -license`'

b. Install XQuartz

c. Install MacPorts for your version of the Mac operating system with `pkg` installer High Sierra, Sierra or El Capitan.

d. Add to the `PATH` variable, where MacPorts is located, e.g, typing the command '`export PATH=/opt/local/bin/port:$PATH`'

e. Install the gcc compiler by typing the following commands '`sudo port -v selfupdate`', '`sudo port install gcc5`'.

For Mac OS X® go to [Step 9], page 8.

8. If you want to use compile your applications with high-performance modules `OpenBLAS` and `sse_kernel`, for now this option only works on the `x86-64` architecture with GNU/Linux. If you do not want this option, go to [Step 9], page 8, and compile with the option `WITH_OPEN_BLAS = no`.

   First verify that you have a Fortran compiler installed

   '`gfortran -v`'
   (>=4.8.5),

   '`gfortran -print-file-name=libgfortran.so`'
   To verify that the `libgfortran` library is installed

   If you can not find the packages, install the missing ones as system administrator (root), with your package manager, e.g. run '`apt-get install gfortran-4.9 libgfortran-4.9-dev`' or '`zypper gcc-fortran libgfortran3`'.

   Then you need to compile and get the static libraries of each of the components.

   `OpenBLAS`

   a. From the http://www.openblas.net/ page, download the source code of the latest version of `OpenBLAS`.

   b. Unzip the file with the '`tar zxvf OpenBLAS-0.2.20.tar.gz`' command.

      '`cd OpenBLAS-0.2.20`'

   c. After editing `Makefile.rule` '`NO_CBLAS=1, NO_LAPACK=1, NO_LAPACKE=1`' and run '`make FC=gfortran`'

   d. Copy `libopenblas.a` static library from the `OpenBLAS-0.2.20` directory to the `openblas` directory of `leac`, e.g. '`cp libopenblas.a leac/openblas`'

   `CBLAS, LAPACK and LAPACKE`

   a. Download lapack-3.8.0.tar.gz from http://www.netlib.org/lapack/

   b. '`tar zxvf lapack-3.8.0.tar.gz`', '`cd lapack-3.8.0`'

   c. '`cp make.inc.example make.inc`'

   d. After editing `make.inc`, and change the variables '`CFLAGS = -O3 -march=native -m64 -fomit-frame-pointer -fPIC -pthread`' and the compilers that you are using '`CC`' and '`FORTRAN`'.

   e. Then you must execute the '`make cblaslib`', '`make lapacklib`' and '`make lapackelib`'.

   f. Copy static libraries '`cp libcblas.a leac/openblas/`', '`cp liblapack.a leac/openblas/`' and '`cp liblapacke.a leac/openblas/`'

`sse_kernel`

a. Change to `sse_kernel` directory, within `leac`

b. Just type `make` to compile the library and get `libssekernel.a`

9. Install `gnuplot`, as a system administrator (root), for GNU/Linux run '`apt-get install gnuplot-x1`'. For Mac OS X® '`sudo port install gnuplot`'.

10. You can now compile the EAC applications, change to the `eac` inside the `leac` home directory '`cd leac/eac`'.

    First edit the `Makefile` file and change the name of the compiler you are using in the `CXX` variable, (e.g. `g++-mp-5`), by default it is `g++`

    Select one of the following compilation options:

    '`make -k -f Makefile DEBUG=yes VERBOSE=yes`'
    : To debug and analyze the detailed execution of the programs. These options allow software engineering ilities of correctness and reliability.

    '`make -k -f Makefile DEBUG=no VERBOSE=no WITHOUT_PLOT_STAT=no`'
    : Optimize and obtain the evolutionary behavior of the population from the fitness function (option `WITHOUT_PLOT_STAT`) in the execution of the programs.

    '`make -k -f Makefile DEBUG=no VERBOSE=no WITH_OPEN_BLAS=yes WITHOUT_PLOT_STAT=yes`'
    : For the processing of the high dimensionality data set, this option is recommended to obtain good performance, for this you must complete See [Step 8], page 7,

11. If you do not want to generate the LEAC API documentation with Doxygen, we recommend that you use the integrated documentation contained in the `html.zip` file of the `doc` directory, just unzip the file. For the other case, do the following: Download a version Doxygen (>= 1.8.13) or with your package manager, install it.

    For Windows download doxygen-1.8.14.windows.x64.bin.zip, unzip the file in `c:\`.

    For GNU/Linux download doxygen-1.8.13, '`tar zxvf doxygen-1.8.13.linux.bin.tar.gz`'. You must also install the dependency '`apt-get install graphviz`'.

    For Mac OS X® '`sudo port install graphviz`' and '`sudo port install doxygen`'.

    To obtain the documentation in a terminal, type '`doxygen Doxyfile`' in t he `leac` directory, or with the full path where the `doxygen` command is located, e.g. '`C:\doxygen-1.8.14.windows.x64.bin\doxygen Doxyfile`' or '`~/doxygen-1.8.13/bin/doxygen Doxyfile`'.

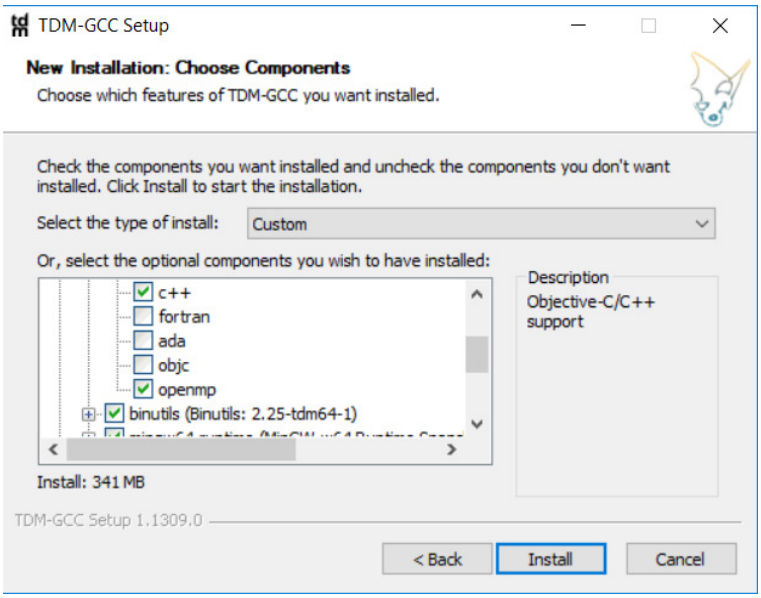

Figure 2.1: Selection of the option openmp in the installation of the tdm-gcc compiler

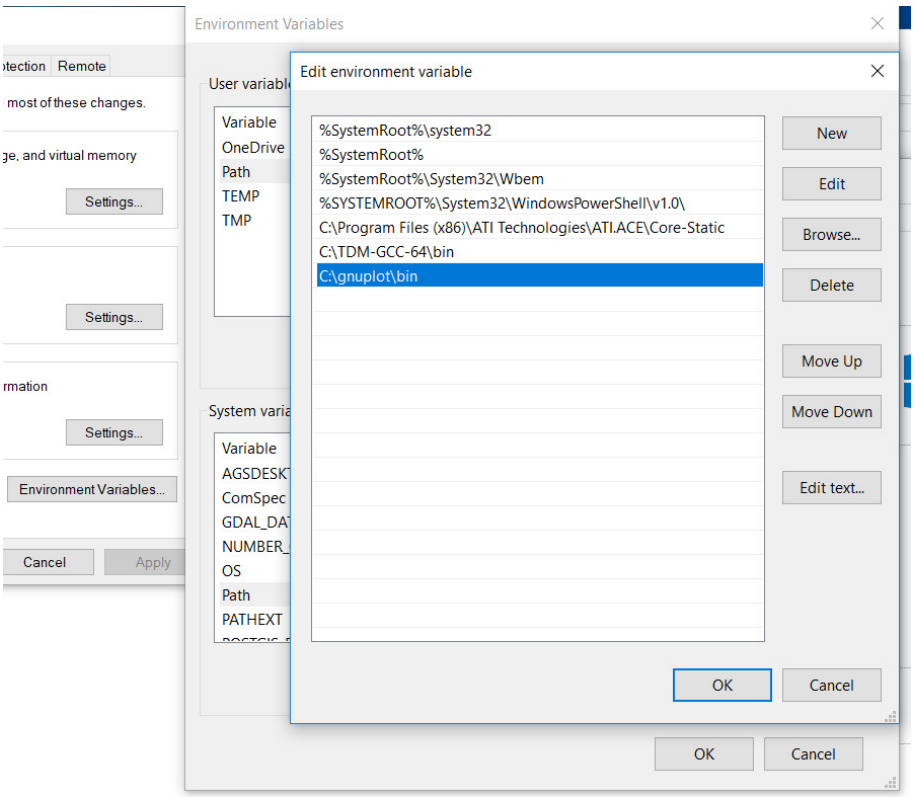

Figure 2.2: Dialog to add the `MinGW` Gnuplot to the `PATH` environment variable

3 LEAC Software

In this chapter it is composed of three sections. The first section describes the steps to implement an evolutionary algorithm with the components provided by LEAC, documented with two examples included in Appendix A [Example source code], page 75. The complete documentation of the LEAC components can be found in the doc/html folder.

If you want to build a new algorithm, you must select the components and put them together in a source file and in the next Section 3.2 [Make an executable for a new algorithm], page 35, we describe how to create the executable file. The third section (Section 3.3 [Using implemented algorithms], page 42) describes how to use the algorithms implemented with LEAC as the end user. If you want to use the implementations of the algorithms listed in Table 1.1, you can omit the first two sections.

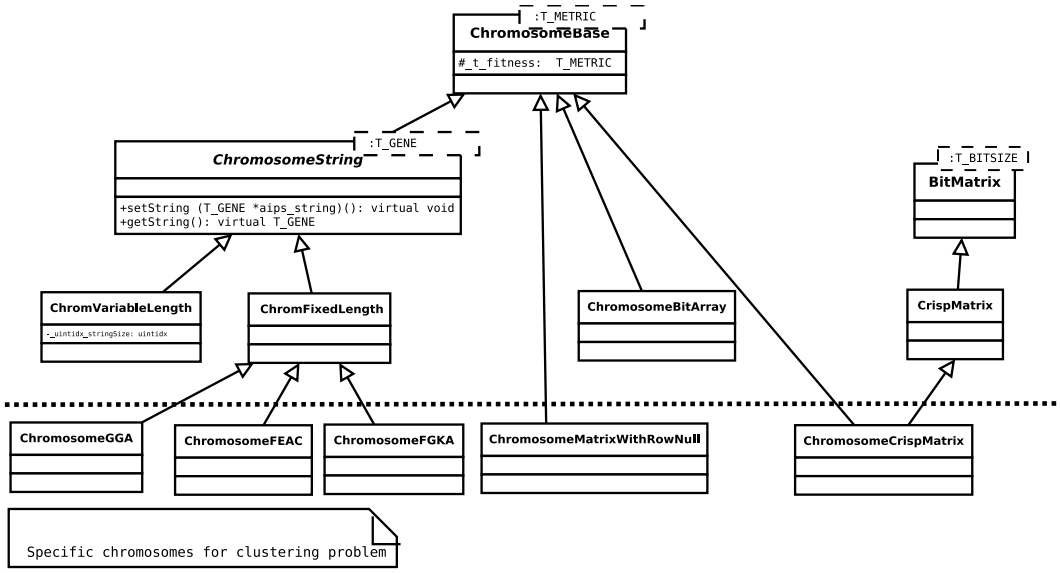

Figure 3.1: Class diagrams of the Chromosomes used for encoding

3.1 Implementation of an algorithm

LEAC provides the necessary components to implement an evolutionary algorithm to solve the clustering problem. The following subsections describe the order of the steps to construct an evolutionary algorithm and some of the functions and classes provided by LEAC for each step.

3.1.1 Encoding criterion

There are different coding schemes proposed, for the case of the clustering problem traditional encoding string as *binary*, *integer* or *real*, You can also use others based on a partition of the objects in the data set as pair of *partitioning table and cluster centroids*. The output of an evolutionary algorithms depends on the type of coding used by each for the chromosomes and its associated phenotype: *centroid*, *medoid*, *label*, *tree*, or *graph-based*

representations. The terms genotype, chromosome, and individual usually have the same meaning in the literature on evolutionary algorithms.

The different encodings are implemented in the LEAC chromosome class hierarchy shown in Figure 3.1.

In LEAC there are three general chromosomes that can be used to implement a large part of the encodings that are:

1. `ChromFixedLength`. Chromosomes with a string of characters, integer or real of fixed length. All instantiated chromosomes will have a fixed length during evolution. The constructor method is:

   [Constructor on gaencode::ChromFixedLength]
   ChromFixedLength <T_GENE, T_METRIC>()

   Where `T_GENE` it is the type of data of each gene, `T_METRIC` it es the type of data for fitness function

   Before creating the objects, you must specify the length that each chromosome will have with the following method:

   [Method on gaencode::ChromFixedLength]
   static void setStringSize (uintidx *aiuintidx_stringSize*)

   See [Example], page 77,

2. `ChromVariableLength`. It is similar to the previous one, with the difference that the length of each chromosome can vary and can also change in evolution.

   [Constructor on gaencode::ChromVariableLength]
   ChromVariableLength<T_GENE,T_METRIC>(const uintidx *aiuintidx_stringSize*)

   Where T_GENE it is the type of data of each gene, T_METRIC it es the type of data for fitness function

3. `ChromosomeBitArray`. To model the binary chromosomes. The above chromosomes could be used with the `T_GENE` parameter equal to `bool`, but to have a better efficiency in storage and performance in the application of genetic operators, bit-level management is performed.

   [Constructor on gaencode::ChromosomeBitArray]
   ChromosomeBitArray<T_BITSIZE,T_METRIC>(const uintidx *aiintidx_numBits*)

   Where `T_BITSIZE` size of the variable to store the bits, `T_METRIC` it es the type of data for fitness function

A classification based on the meaning of the coding (phenotype-genotype), used in different algorithms found in the literature on clustering is described in the following subsections.

3.1.1.1 Cluster labels

a. **String-of-group-numbers encoding**

   It consists of an integer vector of length n, where n is the number of instances. The possible values in the vector are from 1 to k and the ith element establishes a relation of belonging from the ith instance to a cluster k.

   To instantiate the *string-of-group-numbers* chromosomes, you can use the class of use [gaencode::ChromFixedLength], page 12, where `T_GENE` is integer type.

b. **Matrix-based binary encoding**

A $k \times n$ binary matrix can also be used to specify a partition of instances in clusters. Formally defined by Bezdek et al. [BEF84] called *crisp partition* or *hard partition* (3.1).

$$M_c = \{U_{k\times n} | u_{ji} \in \{0,1\}; \sum_{i=1}^{n} u_{ji} > 0, \text{for all } j, \sum_{i=1}^{n} u_{ji} = 1, \text{ for all } i\} \tag{3.1}$$

In which the rows represent clusters and the columns represent instances. In this case, if the ith object belongs to the jth cluster, then 1 is assigned to element jth rows and ith columns of the genotype, whereas the other elements of the same column receive 0. You can use the following class:

```
                                    [Constructor on gaencode::ChromosomeCrispMatrix]
ChromosomeCrispMatrix
          <T_BITSIZE,T_CLUSTERIDX, T_METRIC>
          (const uintidx aiuintidx_numRows,
           const uintidx aiuintidx_numColumns)
```

Where T_BITSIZE size of the variable to store the bits, T_CLUSTERIDX is integer index for clusters, T_METRIC it es the type of data for fitness function See [Example ChromosomeCrispMatrix], page 92.

3.1.1.2 Centroid-based

a. **Real encoding**

A chromosome in this encoding is a vector of real numbers that contains the coordinates of each centroid consecutively of the clusters. For an $d-$dimensional space, the length of a genotype is $d \times k$ words, where the first d positions (or genes) represent the d dimensions of the first cluster centre, the next d positions represent those of the second cluster centre, and so on until k cluster:

$$Ch = [g_{11}\ g_{12} \ldots g_{1d}\ g_{21}\ g_{22} \ldots g_{2d} \ldots g_{k1}\ g_{k2} \ldots g_{kd}]$$

To encode a chromosome based on a centroid you can use the same class see [gaencode::ChromFixedLength], page 12, parameterized for real numbers. See [Example gaencode::ChromFixedLength], page 77.

When a *centroid-based* partition is used, the membership of a objetc to a cluster can be derived by the *nearest object rule* equation (3.2), the rule consider the proximities between the object and centroids, such a way that the ith object is assigned to the cluster more closer (i.e., the most similar). Formally, given the centroids of the groups μ_j, so the object x_i belongs to the cluster C_j if it complies with equation (3.2).

$$x_i \in C_j \leftrightarrow \|x_i - \mu_j\| \min_k \|x_i - \mu_{j'}\|,\ j' = 1, 2, ..k, \tag{3.2}$$

b. **Binary encoding**

Another algorithm based centroids, and with a binary coding is proposed by Tseng and Yang [TY01]. First a data reduction procedure is used, which consists in calculating an adjacency matrix $A_{n\times n}$ and subsequently the *connected components*. The result is blocks $\{B_1, B_2, ..., B_m\}$, with centroid $\{V_1, V_2, ..., V_m\}$ respectively, so V_i is used as a seed to generate a higher level cluster. Consequently, the chromosomes are of length m. The ith genes with a value of '1' in a chromosome, mean to use V_i as seeds, which will group the V_i by proximity represented by a gene with value '0'.

3.1.1.3 Medoid-based

Another way to partition a data set is by selecting the most *representative object* of each cluster. In the literature there are two proposals for integer and binary encoding, which are described in the following subsections.

a. **Integer encoding**

   Integer encoding scheme involves using an array of k elements to provide a medoid-based partition of a data set in k cluster. In this case, each array element represents the index of the object x_i, for $i = 1, 2, ..., n$ where n is the number of objects in the data set.

   For implementation you can use `gaencode::ChromFixedLength` with *T_GENE* parameterized by an unsigned type as data type. See [gaencode::ChromFixedLength], page 12.

b. **Binary encoding**

   Kuncheva and Bezdek [KB97] use binary encoding to define a medoid-based partition. Each chromosome has a length equal to the number of objects n. A bit on with index i indicates that object x_i is a prototype of a cluster C_j. The members of C_j will be determined by rule (3.2), changing the centroid μ_j by the medoid m_j. For implementation you can use `gaencode::ChromosomeBitArray`. See [gaencode::ChromosomeBitArray], page 12.

3.1.1.4 Graph-based

a. **Binary encoding**

   Casillas et al. [CdLM03] use graph-based coding, for objects they get *minimum spanning tree* (MST). The genes represent the edges of the graph, and the vertices the n objects in the data set. As the MST have $n - 1$ edges. This is the length of the chromosomes. In the binary chromosome a value of '`0`' means that this edge remains, while a gene with value '`1`' means that this edge is eliminated. The number of elements with value '`1`' represents the value of $k - 1$, where k is the number of clusters.

3.1.2 Initialization of population

Before initializing the population, you must create a container from the `STL` library, such as `std::vector` or `std::list`, to store the individuals of the population and pool mating, see [Example a vector of chromosomes], page 77.

Several approaches are proposed for the initialization of the population. The simplest procedure to initialize the population is random, objects are randomly assigned to a cluster. Such an initialization strategy usually results in unfavorable initial partitions, since the initial clusters are likely to be mixed up to a high degree. However, it constitutes an effective approach to test the algorithms against hard evaluation scenarios [HCFdC09].

LEAC implements different forms of initialization, for the case of `ChromosomeString` objects, the following general function can use:

[Function]

```
void gagenericop::initializeGenes
          initializeGenes
          (ITERATOR       iterator_first,
           const ITERATOR iterator_last,
```

```
const FUNCTION function)
```

Iterate over the genes of the `ChromosomaString`, with the `begin` and `end` method, we obtain the `ITERATOR` respectively. The function is a lambda function that can generate random values of a distribution and the values are assigned to each gene.

He and Tan [HT12] proposes a more sophisticated initialization, see file tgca_vkcentroid.hpp.

In the case studies of this document see Section A.2 [GA algorithm], page 90, and Section A.1 [KGA algorithm], page 75. They use an initialization of centroids, selected randomly from the instances:

[Function]

```
void clusteringop::randomInitialize
          (mat::MatrixBase<T_FEATURE> &aomatrixt_centroids,
           const INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast)
```

See [Example clusteringop::randomInitialize], page 95,

so all instances are distributed in groups around these centroids by *nearest object rule* equation (3.2).

And finally, with the centroids obtained, it is possible to obtain a partition of the objects in the data set:

[Function]

```
void clusteringop::getPartition
          (mat::CrispMatrix<T_BITSIZE,T_CLUSTERIDX> &aobcrispmatrix_partition,
           mat::MatrixRow<T_FEATURE> &aimatrixt_instances,
           mat::MatrixRow<T_FEATURE> &aimatrixt_centroids,
           dist::Dist<T_DIST,T_FEATURE> &aifunc2p_dist)
```

See [Example clusteringop::getPartition], page 95,

For the initialization of the other partition representations, LEAC provides equivalent functions, found in the header files:

- `clustering_operator_centroids.hpp`
- `clustering_operator_crispmatrix.hpp`
- `clustering_operator_fuzzy.hpp`
- `clustering_operator_medoids.hpp`

3.1.3 Fitness function

The evolutionary algorithms are based on the optimization of some objective function that guides the evolutionary search, its function is known *fitness function*. In the clustering problem different measures are used for *fitness function*, these are classified as *unsupervised measures* (Section 3.1.3.2 [Unsupervised measures], page 18) and *supervised measures* (Section 3.1.3.3 [Supervised measures], page 28). Unsupervised measures usually use a distance, to describe relationships *inter cluster* and *intra cluster*. The following section lists the most common distances and their functions that implement them.

3.1.3.1 Distances

Distance measures are a key point in clustering to find the similarity between two objects x_i and x'_i. The most common is the *Euclidean distance* $\|x_i - x'_i\|$. LEAC offers a module in

the `dist_euclidean.hpp` file to calculate the distances and the class diagram is shown in Figure 3.2.

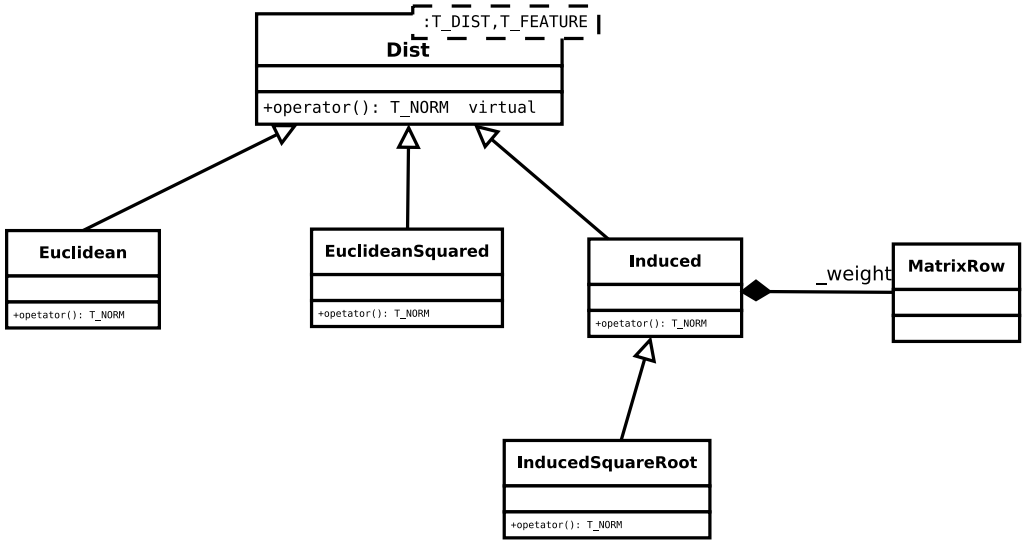

Figure 3.2: Class diagrams of the Distances

All classes derived from `Dist` define the *function call operator* that allows you to find the distance between two x_i and $x_{i'}$ objects:

[Method on dist::Dist]

```
T_DIST operator()(const T_FEATURE*, const T_FEATURE*, const uintidx)
```

To instantiate a `dist::Dist` object the following constructors are used

1. Euclidean distance

   [Constructor on dist::Euclidean]

   ```
   Euclidean<T_DIST,T_FEATURE>()
   ```

2. Euclidean square distance

   [Constructor on dist::Induced]

   ```
   Induced<T_DIST,T_FEATURE>(const mat::MatrixRow<T_DIST>& aimatrix_weight)
   ```

   See [Example dist::Induced], page 17,

3. Induced distance

   The *induced distance* is a generic measure obtained by multiplying the transposed vector of point x_i to $x_{i'}$ by the *matrix of weight* A and with the vector not transposed (3.3).

   $$D_{Ind}(x_i, x_{i'}) = (x_i - x_{i'})^T A (x_i - x_{i'}) \tag{3.3}$$

   To calculate the *matrix of weights* A we have the following functions

   a. Identity matrix

      [Function]

      ```
      mat::MatrixRow<T_FEATURE> mat::getIdentity (const uintidx aiui_dimension)
      ```

If the *identity matrix* is used the induced distance is equivalent to the *Square Euclidean distance* ($A = I$) See [Example mat::getIdentity], page 17,

b. Mahalanobis matrix

[Function]

```
mat::MatrixRow<T_FEATURE> dist::getMatrixMahalonobis
          (INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast)
```

It is the inverse of the *covariance matrix* C_x of the data set X. When used as an matrix of weights at the *induced distance* is equivalent to *Mahalanobis distance* ($A = C_x^{-1}$). See [Example dist::getMatrixMahalonobis], page 18,

c. Diagonal matrix

[Function]

```
mat::MatrixRow<T_FEATURE> dist::getMatrixDiagonal
          (T_FEATURE* aiarrayt_varianceFeactures)
```

It is the inverse matrix of the variance of the attributes in the main diagonal ($A = D_x^{-1}$). See [Example dist::getMatrixDiagonal], page 17,

The following is an example taken from the main_gas_clustering.cpp file, which shows how you can create an instance of a distance and select one of them at run time, depending on the parameters provided by the user:

```
/*Declaration of a reference to a generic object
    of type of dist::Dist.
    DATATYPE_REAL is the type of data obtained when
       calculating the distance.
    DATATYPE_FEATURE is the data type of the dimensions
       of the instances or objects
*/
dist::Dist<DATATYPE_REAL,DATATYPE_FEATURE>
  *pfunct2p_distAlg = NULL;

/*Create a dist::Dist object based on the parameter provided
     by the user, which will have a polymorphic behavior
*/
switch ( linparam_ClusteringGA.getOpDistance() ) {
case INPARAMCLUSTERING_DISTANCE_EUCLIDEAN:
  pfunct2p_distAlg =
    new dist::Euclidean<DATATYPE_REAL,DATATYPE_FEATURE>();
  break;
case INPARAMCLUSTERING_DISTANCE_EUCLIDEAN_SQ:
  pfunct2p_distAlg =
    new dist::EuclideanSquared<DATATYPE_REAL,DATATYPE_FEATURE>();
  break;
case INPARAMCLUSTERING_DISTANCE_EUCLIDEAN_INDUCED:


  pfunct2p_distAlg =
    new dist::Induced<DATATYPE_REAL,DATATYPE_FEATURE>
    (mat::getIdentity
     <DATATYPE_REAL>
     (data::Instance<DATATYPE_FEATURE>::getNumDimensions()));
  break;
case INPARAMCLUSTERING_DISTANCE_DIAGONAL_INDUCED:
```

```
  pfunct2p_distAlg =
    new dist::Induced<DATATYPE_REAL,DATATYPE_FEATURE>
    (stats::getMatrixDiagonal<DATATYPE_FEATURE>
     (larray_desvstdFeactures)
    );
  break;
case INPARAMCLUSTERING_DISTANCE_MAHALONOBIS_INDUCED:

  pfunct2p_distAlg =
    new dist::Induced<DATATYPE_REAL,DATATYPE_FEATURE>
    (stats::getMatrixMahalonobis
      (lvectorptinst_instances.begin(),
       lvectorptinst_instances.end()
      )
    );
  break;
default:
  throw std::invalid_argument("main_gas_clustering: undefined norm");
  break;
}
```

3.1.3.2 Unsupervised measures

This type of evaluation tries to determine the quality of a given obtained partition of the data without any external information available. This is why this unsupervised measure are sometimes called as internal measures [ABSSJF+12]. All *unsupervised measures* function of the library are defined in the header file `unsupervised_measures.hpp`, to do calculations of metrics you can include files or simply the `leac.hpp` library, which contains all the header files.

The measures used in the evolutionary algorithms are described below.

1. **Sum of quadratic errors (SSE)**

   A common clustering criterion or quality indicator is the *sum of squared error* (SSE), defined by Chang et al. [CZZ09] for equation (3.4).

$$\text{SSE} = \sum_{C_j} \sum_{x_i \in C_j} (x_i - \mu_j)^T (x_i - \mu_j) = \sum_{C_j} \sum_{x_i \in C_j} \|x_i - \mu_j\|^2 \tag{3.4}$$

   Where x_i represents the instance ith of the data set. We use the following convention for subscripts i; $i \in \{1, 2, ..., n\}$, for the instances. The subscript for the groups is j; $j \in \{1, 2, ..., k\}$ and the centroid of cluster C_j is denoted as μ_j.

   SSE is also defined by Krishna and Murty [KM99], using [Crisp Partition], page 12, and calling it *total within-cluster variation* (TWCV) equation (3.5).

$$\text{TWCV} = \sum_{j=1}^{k} \text{WCV}^{(j)} = \sum_{j=1}^{k} \sum_{i=1}^{n} u_{ij} \sum_{l=1}^{d} (x_{il} - \mu_{jl})^2 \tag{3.5}$$

   To refer to the dimensions of the objects we use l: $l \in \{1, 2, ..., d\}$, so lth denotes the dimension of x_{il}.

   Or with some slight variation *Sum of Euclidean Distance* SED equation (3.6).

$$\text{SED} = \sum_{C_j} \sum_{x_i \in C_j} \|x_i - \mu_j\| \tag{3.6}$$

This is probably the most straightforward and popular evaluation distance in the literature. It only considers cohesion of clusters in order to evaluate the quality of a given partition data [ABSSJF+12].

This metric generally used by different algorithms when the number of clusters k is known and is used by [BM02a] [CZZ09].

For the calculation of SSE you have three functions. Different distances can be passed as parameter *aifunc2p_dist* (See Section 3.1.3.1 [Distances], page 15), to obtain the variations of the metric:

[Function]

```
std::pair<T_METRIC,bool> um::SSE
          (const mat::MatrixRow<T_FEATURE> &aimatrixt_centroids,
           INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           const dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist)
```

The partition is derived by *aimatrixt_centroids*. See [Equation (3.2)], page 13. See [Example um::SSE], page 81,

Other functions for the calculation of *SSE* that depend on the way to specify the membership of an object to a cluster, are the following:

[Function]

```
T_METRIC um::SSE
          (const mat::MatrixRow<T_FEATURE> &aimatrixt_centroids,
           INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           T_CLUSTERIDX *aiarraymmidx_memberShip,
           const dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist)
```

[Function]

```
std::pair<T_METRIC,bool> um::SSE
          (const mat::MatrixRow<T_FEATURE> &aimatrixt_centroids,
           INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           const partition::Partition<T_CLUSTERIDX>
           &aipartition_clusters,
           dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist)
```

Depending on the encoding used, you have a set of partition clusters classes (`partition::Partition`) for calculating metrics generically. See Figure 3.3. For example, a partition based on (3.2), the constructor you can use is

[Constructor on partition::PartitionCentroids]

```
PartitionCentroids<T_FEATURE,T_CLUSTERIDX,T_DIST,INPUT_ITERATOR>
           (mat::MatrixRow<T_FEATURE> &aimatrixt_centroids,
            const INPUT_ITERATOR aiiterator_instfirst,
            const INPUT_ITERATOR aiiterator_instlast,
            const dist::Dist<T_DIST,T_FEATURE>& aifunc2p_dist)
```

And for one based on a crisp matrix (4.1):

[Constructor on partition::PartitionCrispMatrix]

```
PartitionCrispMatrix<T_BITSIZE,T_CLUSTERIDX>
          (const mat::CrispMatrix<T_BITSIZE,T_CLUSTERIDX> &aibitcrisp_matrix)
```

See [Example partition::PartitionCrispMatrix], page 98,

To avoid defining the template parameters for the case of a partition, you can use the `makePartition`:

[Function]

```
T_METRIC partition::makePartition
          (mat::MatrixRow<T_FEATURE>  &aimatrixt_centroids,
           const INPUT_ITERATOR  aiiterator_instfirst,
           const INPUT_ITERATOR  aiiterator_instlast,
           const T_CLUSTERIDX  aimcidx_numClusters,
           const dist::Dist<T_DIST,T_FEATURE>  &aifunc2p_dist)
```

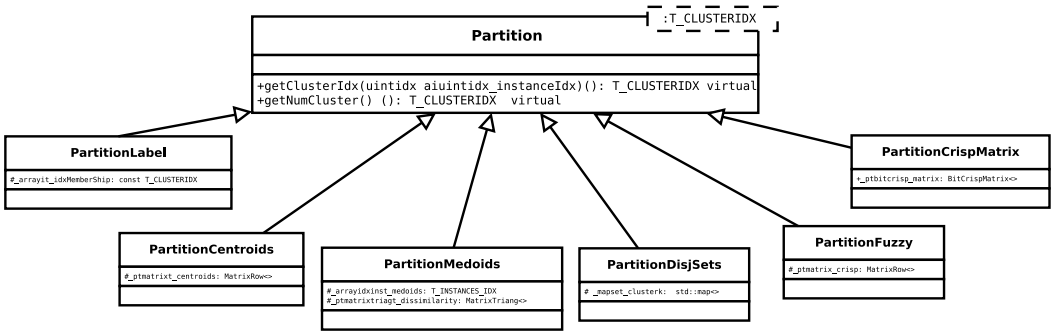

Figure 3.3: Class diagram of the partitions used to calculate the metrics

2. **Distortion**

   The distortion of the clusters is a measure define by Fränti et al. [FKKN97]. Similar to SSE but considers the number of instances and attributes (3.7).

$$\text{distortion}(C) = \frac{1}{n \cdot d} \sum_{i=j}^{n} D(x_i, f_C(x_i))^2 \tag{3.7}$$

   Where D is the Euclidean distance, $f_C(x_i)$ be a mapping which gives the closest centroid in solution C for a instance x_i, n is the number of instances, and d number of attributes of the instances.

   The function that allows to calculate the measure of the distortion is:

   [Function]

```
std::pair<T_METRIC,bool> um::distortion
          (const mat::MatrixBase<T_FEATURE> &aimatrixt_centroids,
           INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           T_CLUSTERIDX *aiarraycidx_memberShip,
           const dist::Dist<T_METRIC,T_FEATURE>  &aifunc2p_dist,
           const FUNCINSTFREQUENCY func_instfrequency)
```

3. **Sum of Euclidean Distance for Medoid**

   For *k-medoid*, replacing the centroids by the most representative instance in equation (3.6), we obtain the cost function sum of Euclidean distances to the most representative instance defined by equation (3.8).

$$\text{SEDmedoid} = \sum_{C_j} \sum_{x_i \in C_j} \|x_i - m_j\| \tag{3.8}$$

Where m_j represents the *medoid* or *prototype* of cluster C_j

[Function]

```
T_METRIC um::SSEMedoid
          (const uintidx *aiarrayidxinst_medoids,
           const T_CLUSTERIDX aimcidx_numClustersK,
           const mat::MatrixTriang<T_METRIC> &aimatrixtriagt_dissimilarity)
```

The `medoids::getMatrixDissimilarity` function calculates and returns the triangular distance matrix using a specified distance measure. Used to get other measures for example `um::SSEMedoid`. This is in `medoids_clustering.hpp` file.

[Function]

```
mat::MatrixTriang<T_DIST> medoids::getMatrixDissimilarity
          (INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           const dist::Dist<T_DIST,T_FEATURE> &aifunc2p_dista)
```

4. **Least-squared errors functional**

A *fuzzy c-partitions* is represented by a matrix (3.9), defined by Bezdek et al. [BEF84].

$$M_{fc} = \{U_{k\times n} | u_{ji} \in [0,1]; \sum_{i=1}^{n} u_{ji} > 0, \text{for all } j, \sum_{i=1}^{n} u_{ji} = 1, \text{ for all } i\}, \tag{3.9}$$

Several clustering criteria have been proposed for identifying optimal fuzzy c-partitions in X, the most popular and well-studied criterion is associated with matrices M_{fc} equation (3.9) is called *least-squared errors functional* (3.10), described in [BEF84].

$$J_m(U,\mu) = \sum_{i=1}^{n}\sum_{j=1}^{k} u_{ji}^m D_{Ind}(x_i - \mu_j), \tag{3.10}$$

Where

$U \in M_{fc}$ (3.9) fuzzy c-partition of X;

$\mu = [\mu_1, \mu_2, ..., \mu_k]$ centroids,

m weighting exponent; $1 \leq m < \infty$

$D_{Ind}(x_k, \mu_i)$ is one of the *induced distances* from x_i to μ_j. See [Induced distance], page 16,

[Function]

```
T_METRIC um::jm
          (mat::MatrixRow<T_METRIC> &aimatrixt_u,
           mat::MatrixRow<T_FEATURE> &aimatrixt_centroids,
           INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           T_METRIC aif_m,
           dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist)
```

For hard clustering will be based on $J_m(U,\mu)$ from equation (3.10), we will rewrite J_1 as (3.11).

$$J_1(U,\mu) = \sum_{i=1}^{n}\sum_{j=1}^{k} u_{ji} D_{Ind}(x_i - \mu_i) \tag{3.11}$$

[Function]

```
T_METRIC um::j1
          (mat::BitMatrix<T_BITSIZE> &aimatrix_crisp,
           mat::MatrixRow<T_FEATURE> &aimatrixt_centroids,
           INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist)
```

See [Example um::j1], page 95,

5. **Davis-Bouldin Index**

   Davies–Bouldin index (DB), defined by Davies and Bouldin [DB79], is a function of the ratio of the sum of *within-cluster scatter* to *between-cluster separation*, DB index for the partitioning of k clusters is defined by (3.12).

$$DB = \frac{1}{k}\sum_{j} R_{C_j,C_{1\leq j'\leq k}}, \quad j' = 1,2,...,k \quad \text{and} \quad j \neq j' \tag{3.12}$$

   in which the index for the jth cluster against all clusters least the same $R_{C_j,C_{1\leq j'\leq k}}$ is given by

$$R_{C_j,C_{1\leq j'\leq k}} = \max_{j',\, j' \neq j} R_{C_j,C_{j'}}$$

   where $R_{C_j,C_{j'}}$ is a measure between a pair of cluster defined by

$$R_{C_j,C_{j'}} = \frac{S_{q,C_j} + S_{q,C_{j'}}}{d_{jj',t}}$$

   and the scatter S_{q,C_j} within for jth cluster, is computed as

$$S_{q,C_j} = \left(\frac{1}{|C_j|}\sum_{x_i \in C_j} \{|x_i - \mu_j|_2^q\}\right)^{1/q}$$

   S_q is the q^{th} root of the q^{th} moment of the points in cluster j with respect to their mean, and is a measure of the dispersion of the points in cluster j. $d_{jj',t}$ is the Minkowski distance of order t between the centroids that characterize clusters j and j'.

   DB use in [BM02b].

   [Function]

```
T_METRIC um::DBindex
          (const mat::MatrixBase<T_FEATURE> &aimatrixt_centroids,
           INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           const partition::Partition<T_CLUSTERIDX> &aipartition_clusters,
           const dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist)
```

6. **Silhouette**

   This metric known as silhouette was proposed by Kaufman and Rousseeuw [KR90]. Consider an object x_i belonging to cluster C_j. So, the average dissimilarity of x_i to all other objects of C_j. is denoted by $a(x_i)$. Now let us take into account cluster $C_{j'}$. The average dissimilarity of x_i to all objects of $C_{j'}$, will be called $D(x_i, C_{j'})$. After computing $D(x_i, C_{j'})$, for all clusters $C_j \neq Cj'$, the smallest one is selected, i.e.

$b(x_i) = \min D(x_i, C_{j'})$. This value represents the dissimilarity of x_i to its neighbor cluster, and the silhouette $s(x_i)$ is defined by equation (3.13).

$$s(x_i) = \frac{b(x_i) - a(x_i)}{\max\{a(x_i), b(x_i)\}}, \tag{3.13}$$

The higher $s(x_i)$ is better the assignment of the object x_i to a given cluster and

$$-1 \leq s(x_i) \leq 1$$

Use in [HE03] and [ABSSJF+12].

[Function]

```
T_METRIC um::silhouette
          (mat::MatrixTriang<T_METRIC> &aimatrixtriagrt_dissimilarity,
           ds::PartitionLinkedNumInst<T_CLUSTERIDX,
           T_INSTANCES_CLUSTER_K> &aipartlinknuminst_memberShip)
```

7. **Simplified silhouette**

Alves et al. [ACH06] propose a metric based on [KR90], arguing that to calculate the equation of Silhouette (3.13), the computational cost is $O(n^2)$, which is often not sufficiently efficient for real-world clustering applications (e.g. data mining, text mining, gene-expression data analysis). To avoid this limitation, they propose a simplified silhouette, based on the computation of distances between objects and cluster centroids, which are the mean vectors of the clusters. More specifically, the term $a(i)$ of equation (3.13) represents the dissimilarity of object x_i to the centroid of its cluster (C_j). Similarly, instead of computing $D(x_i, C'_j)$ as the average dissimilarity of x_i to all objects of C'_j, $C_j \neq Cj'$, only the distance between x_i and the centroid of C'_j must be computed. With these modifications, the computational cost of $O(n^2)$ is reduce to $O(n)$.

[Function]

```
std::vector<T_METRIC> um::simplifiedSilhouette
          (const mat::MatrixBase<T_FEATURE> &aimatrixt_centroids,
           INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           const partition::Partition<T_CLUSTERIDX> &aipartition_clusters,
           const std::vector<T_INSTANCES_CLUSTER_K> &aivectorit_numInstClusterK,
           const dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist)
```

8. **CS measure**

The CS measure defined for the first time by Chou et al. [CSL04] and later by Das et al. [DAK08], is given by the equation (3.14).

$$CS(C) = \frac{\frac{1}{k}\sum_{j=1}^{k}\left\{\frac{1}{|C_j|}\sum_{x_i \in C_j}\max_{x_{i'} \in C_j}\{D(x_i, x_{i'})\}\right\}}{\frac{1}{k}\sum_{j=1}^{k}\left\{\min_{j \in k, j \neq j'}\{D(\mu_j, \mu_{j'})\}\right\}} \tag{3.14}$$

Where

D is a distance function,

x_i and $x_{i'}$ are instances in the same cluster,

μ_j and $\mu_{j'}$ are the centroids of two different cluster.

Chou et al. [CSL04], they establish that this measure is a function of the ratio of the sum of within-cluster scatter to between-cluster separation. The smallest $CS(C)$

indicates a valid optimal partition. The CS measure has the same rationale as the DI (See [Dunn's index], page 24) and the DB (See [Davis-Bouldin Index], page 22).

[Function]

```
T_METRIC um::CSmeasure
          (INPUT_ITERATOR aiiterator_instfirst,
           const mat::MatrixRow<T_FEATURE> &aimatrixt_centroids,
           const ds::PartitionLinkedNumInst<T_CLUSTERIDX,T_INSTANCES_CLUSTER_K>
               &aipartlinknuminst_memberShip,
           const dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist)
```

9. **Dunn's index**

A well-established hard cluster validity measure is the Dunn's index (DI) equation (3.15). This measure gives good results when the groups are well separated [CSL04].

$$DI(C) = \min_{j \in C} \left\{ \min_{j' \in C, j' \neq j} \left\{ \frac{\delta(C_j, C_{j'})}{\max_{j'' \in C} \{\Delta(C_{j''})\}} \right\} \right\} \tag{3.15}$$

Where

$$\delta(C_j, C_{j'}) = \min \{D(x_i, x_{i'}) | x_i \in C_j, x_{i'} \in C_{j'}\},$$

$$\Delta(C_j) = \max \{D(x_i, x_{i'}) | x_i, x_{i'} \in C_j\},$$

and x_i and $x_{i'}$ are instances.

The main drawback with the direct implementation of Dunn's index is its computational load because calculating $DI(C)$ becomes computationally very expensive, when the number of clusters k and the size of the data set n increase. The largest $DI(C)$ indicates a valid optimal partition [CSL04].

[Function]

```
T_METRIC um::DunnIndex
          (INPUT_ITERATOR aiiterator_instfirst,
           const ds::PartitionLinked<T_CLUSTERIDX> &aipartlink_memberShip,
           const dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist)
```

To avoid calculating distance between instances again and improve performance in the DI evaluation, use the following function. It uses a `mat::MatrixTriang` of precalculated distance between objects, and reduces the time of calculation of the distance from one object to another, especially for objects with a large dimension d:

[Function]

```
T_METRIC um::DunnIndex
          (mat::MatrixTriang<T_METRIC> &aimatrixtriagrt_dissimilarity,
           ds::PartitionLinked<T_CLUSTERIDX> &aipartlinknuminst_memberShip)
```

10. **Simplified Dunn's index**

Just as the measure of [silhouette], page 22, can simplify the complexity of the calculation also [Dunn's measure], page 24, to reduce the computational cost of $O(n^2)$ to $O(n)$ using the centroids. This measure will be called *Simplified Dunn's index* (SDI) and is defined by the equation (3.16).

$$SDI(C) = \min_{j \in C} \left\{ \min_{j' \in C, j' \neq j} \left\{ \frac{\delta(C_j, C_{j'})}{\max_{j'' \in C} \{\Delta(C_{j''})\}} \right\} \right\} \tag{3.16}$$

Where

$$\delta(C_j, C_{j'}) = \min\{D(\mu_j, \mu_{j''})|j \neq j''\}$$

$$\Delta(C_j) = \max\{D(x_i, \mu_j)|x_i \in C_j\}$$

And the function to calculate *Simplified Dunn's index*

[Function]

```
T_METRIC um::simplifiedDunnIndex
          (const mat::MatrixBase<T_FEATURE> &aimatrixt_centroids,
           INPUT_ITERATOR aiiterator_instfirst,
           const ds::PartitionLinked<T_CLUSTERIDX> &aipartlink_memberShip,
           const dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist)
```

11. **Variance ratio criterion**

The *variance ratio criterion* (VRC) sometimes called *Calinski–Harabasz index* initially proposed in [CH74] is based on the internal cluster cohesion and the external cluster isolation. The corresponding internal cohesion is calculated by the within-group sum of square distances [HT12].

The index is defined as:

$$VRC_k = \frac{SS_B}{SS_W} \cdot \frac{(n-k)}{(k-1)} \tag{3.17}$$

Where SS_B is the overall between-cluster variance, SS_W is the overall within-cluster variance, k is the number of cluster, and n is the number of instances.

The overall between-cluster variance SS_B is defined as

$$SS_B = \sum_{j=i}^{k} |C_j| \|\mu_j - M\|^2$$

Where μ_j is the centroid of cluster j, M is the overall mean of the instances. The overall within-cluster variance SS_W is defined as

$$SS_W = \sum_{j=1}^{k} \sum_{x_i \in C_j} \|x_i - \mu_j\|^2$$

The VRC should be maximized. Use in [CdLM03] and [HT12].

[Function]

```
T_METRIC um::VRC
          (const mat::MatrixRow<T_FEATURE> &aimatrixt_centroids,
           INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           const partition::Partition<T_CLUSTERIDX> &aipartition_clusters,
           const dist::Dist<T_METRIC,T_FEATURE>  &aifunc2p_dist)
```

12. **Intra-cluster and inter-cluster distance**

The definition of *intra-cluster and inter-cluster distance* (DIIC) by Tseng and Yang in [TY01] is given by the equation (3.18).

$$DIIC = \sum_{i=1}^{k} D_{inter}(C_j)w - D_{intra}(C_j) \tag{3.18}$$

Where $D_{intra}(C_j)$ is the intra-cluster distance

$$D_{intra}(C_j) = \sum_{B_l \subset C_j} \|v_l - \mu_j\| \cdot |B_l|$$

and $D_{inter}(C_j)$ is the inter-cluster distance

$$D_{inter}(C_j) = \sum_{B_l \subset C_j} \left(\begin{matrix} min \\ j \neq k \end{matrix} \|v_l - \mu_j\| \right) \cdot |B_l|$$

And w is a weight. If the value of w is small, we emphasize the importance of $D_{intra}(C_j)$. This tends to produce more clusters and each cluster tends to be compact. If the value of w is chosen to be large, we emphasize the importance of $D_{inter}(C_j)$. This tends to produce fewer clusters and each cluster tends to be loose [TY01].

[Function]

```
T_METRIC um::Dintra
          (const mat::MatrixRow<T_FEATURE> &aimatrixrowt_S,
           const mat::MatrixRow<T_FEATURE> &aimatrixrowt_Vi,
           const std::vector<T_INSTANCES_CLUSTER_K> &aivectort_numInstBi,
           const partition::Partition<T_CLUSTERIDX> &aipartition_clustersBkinCi,
           const dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist)
```

[Function]

```
T_METRIC um::Dinter
          (const mat::MatrixRow<T_FEATURE> &aimatrixrowt_S,
           const mat::MatrixRow<T_FEATURE> &aimatrixrowt_Vi,
           const std::vector<T_INSTANCES_CLUSTER_K> &aivectort_numInstBi,
           const partition::Partition<T_CLUSTERIDX> &aipartition_clustersBkinCi,
           const dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist)
```

This metric can only be calculated when having subgroups. Given k clusters $C_1, C_2, ..., C_k$ each C_j with S_j centroid, is constructed from the subgroups B_i with V_i centroid. This way of clustering is a characteristic particular feature of the algorithm described in [TY01].

13. **Validity index I**

The *validity index I* or simply *Index I* described in Maulik and Bandyopadhyay in [MB02] and [BM07]. It is used as a metric to measure clustering performance. It was proposed as a measure to indicate the (goodness) validity of the solution in the cluster. It is defined in (3.19).

$$I(k) = \left(\frac{1}{k} \cdot \frac{E_1}{E_k} \cdot D_k \right)^p, \tag{3.19}$$

Where k is the number of clusters

$$E_k = \sum_{j=1}^{k} \sum_{i=1}^{n} u_{ji} \|x_i - \mu_j\|,$$

and

$$D_k = \max_{j,j'=1}^{k} \|\mu_{j'} - \mu_j\|$$

n is the total number of objects x_i. $U(X) = [u_{ji}]_{k\times n}$ is a partition matrix of the objects and μ_j is the centroid es el centro jth. The value of k that maximizes $I(k)$ is considered the correct number of clusters.

[Function]

```
T_METRIC um::indexI
          (const mat::MatrixRow<T_FEATURE> &aimatrixt_centroids,
           INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           const partition::Partition<T_CLUSTERIDX> &aipartition_clusters,
           const dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist,
           const T_METRIC airt_p = 2.0)
```

14. **Xie-Beni index**

The index of Xie-Beni (XB), by Xie and Beni [XB91] is defined for *fuzzy c-partitions*. See [Definition fuzzy c-partitions], page 21. This index can be extended to a crisp partition. See [crisp partition], page 12. Note that M_c is imbedded in M_{fc}.

The Xie-Beni index is defined as the quotient of the total variance σ, and minimal separation of groups d_{min} equation (3.20).

$$XB = \frac{\sigma}{n \cdot (d_{min})^2} \tag{3.20}$$

In detail

$$\sigma = \sum_{j=1}^{k} \sum_{x_i}^{n} u_{ji}^2 \|x_i - \mu_j\|^2,$$

$$d_{min} = \min_{j,j'=1,j\neq j'}^{k} \|\mu_j - \mu_{j'}\|$$

$$XB = \frac{\sum_{j=1}^{k} \sum_{x_i}^{n} u_{ji}^2 \|x_i - \mu_j\|^2}{n \cdot (d_{min})^2}$$

[Function]

```
T_METRIC um::xb
          (mat::MatrixRow<T_METRIC> &aimatrixt_u,
           mat::MatrixRow<T_FEATURE> &aimatrixt_centroids,
           INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist)
```

And the function to calculate the XB to a hard partition (See [partition::Partition], page 19).

[Function]

```
T_METRIC um::xb
          (const mat::MatrixRow<T_FEATURE> &aimatrixt_centroids,
           INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           const partition::Partition<T_CLUSTERIDX> aipartition_clusters,
           const dist::Dist<T_METRIC,T_FEATURE> &aifunc2p_dist)
```

3.1.3.3 Supervised measures

1. **Rand Index**

   The *Rand Index* by Rand [Ran71], is defined for two partitions of the same data set X, C in k cluster and R in k' known clases. Alves et al. [ACH06] indicate that these measures can be seen as an absolute criterion or referential standard that allows the use of classification data sets for performance assessment not only of *classifiers* with the same number of clusters and class ($k = k'$), if not also different ($k \neq k'$). In the same article they write *Rand Index* as $\Omega(R, C)$ given by equation (3.21).

$$\Omega(R, C) = \frac{a + d}{a + b + c + d} \tag{3.21}$$

   Where:

   - a is the number of pairs of data objects belonging to the same class in R and to the same cluster in C.
   - b is the number of pairs of data objects belonging to the same class in R yet to different clusters in C.
   - c is the number of pairs of data objects belonging to different classes in R yet to the same cluster in C.
   - d is the number of pairs of data objects belonging to different classes in R and to different clusters in C.

   The function that obtains the *Rand Index* is

   [Function]

```
T_METRIC sm::randIndex
          (const sm::ConfusionMatchingMatrix<T_INSTANCES_CLUSTER_K>
               &aimatchmatrix_confusion)
```

   To obtain the Rand Index, you must first obtain the confusion matrix:

   [Function]

```
sm::ConfusionMatchingMatrix<T_INSTANCES_CLUSTER_K> sm::getConfusionMatrix
          (INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           const partition::Partition<T_CLUSTERIDX> &aipartition_clusters,
           const FUNCINSTFREQUENCY func_instfrequency,
           const FUNCINSTCLASS func_instclass)
```

   See [Example sm::getConfusionMatrix], page 98,

2. **Purity**

   Purity equation (3.22) is a simple and transparent evaluation measure each cluster is assigned to the class which is most frequent in the cluster, and then the accuracy of this assignment is measured by counting the number of correctly assigned objects and dividing by n. For more information on Purity see [MRS08].

$$\text{purity}(C, R) = \frac{1}{n} \sum_{j} \max_{j'} |C_j \cap R_{j'}| \tag{3.22}$$

   Where C and R two partitions of the same data set X, of k cluster and k' known classes respectively.

   [Function]

```
T_METRIC sm::purity
          (const sm::ConfusionMatchingMatrix<T_INSTANCES_CLUSTER_K>
           &aimatchmatrix_confusion)
```

3. **Precision**

   To calculate the *precision* (3.23), we use the pairs of the R and C partitions, and based on [Faw06].

$$\text{precision} = a/(a+c) \tag{3.23}$$

   Where:

   - a is the number of pairs of data objects belonging to the same class in R and to the same cluster in C.
   - c is the number of pairs of data objects belonging to different classes in R yet to the same cluster in C.

[Function]

```
T_METRIC sm::precision
          (const ConfusionMatchingMatrix<T_INSTANCES_CLUSTER_K>
           &aimatchmatrix_confusion)
```

4. **Recall**

   To calculate the *recall* (3.24), we use the pairs of the R and C partitions, based on [Faw06].

$$\text{recall} = a/(a+b) \tag{3.24}$$

   Where:

   - a is the number of pairs of data objects belonging to the same class in R and to the same cluster in C.
   - b is the number of pairs of data objects belonging to the same class in R yet to different clusters in C.

[Function]

```
T_METRIC sm::recall
          (const ConfusionMatchingMatrix<T_INSTANCES_CLUSTER_K>
           &aimatchmatrix_confusion)
```

3.1.4 Stop Criterion

There exists no stopping criterion in the literature which ensures the convergence in the evolutionary algorithms, to an optimal solution. Usually, two stopping criteria are used. In the first, the process is executed for a fixed number of iterations and the best individual/solution obtained is taken to be the optimal one. In the other, the algorithm is terminated if no further improvement in the fitness value of the best individual is observed for a fixed number of iterations, and the best individual obtained is taken to be the optimal one [MC96].

3.1.5 Select individuals

In this step, the individuals of the population are selected in a mating group to apply the crossing and mutation operators. LEAC has avaliable two schemes of individual selection.

1. Roulette wheel. Based on the concept of survival used in natural genetic systems. To apply this selection after having evaluated the fitness function, the probability distribution is calculated with the function:

[Function]

```
std::vector<T_REAL> prob::makeDistRouletteWheel
          (INPUT_ITERATOR aiiterator_instfirst,
           INPUT_ITERATOR aiiterator_instlast,
           const FITNESSOPERATION fitness_func)
```

See [Example prob::makeDistRouletteWheel], page 85,

With the probability distribution, the indices of the individuals to be selected are generated:

[Function]

```
T_INTIDX gaselect::getIdxRouletteWheel
          (const std::vector<T_PROBABILITY> &aivectorrt_probDist,
           const T_INTIDX aiuintidx_begin)
```

2. Tournament. This selector carries out several "tournaments" between a small number of individuals specified by the variable *aiuintidx_orderTournament* chosen at random from the population and the individual with the best fitness of each tournament is selected:

[Function]

```
INPUT_ITERATOR gaselect::tournament
          (const INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           const uintidx  aiuintidx_orderTournament,
           const FITNESSOPERATION fitness_func)
```

Elitism. Some algorithms select the best individuals to be used in the crossing, as in [BEF84] (see Section A.2 [GA algorithm], page 90). In [BM02a] the elitism has been implemented in each generation select the worst and replace it with the best individual, (see [Example elitism replace the worst], page 83).

Also generally, the best individual is retained in the majority, of the algorithms, in this way provides the solucion al the clustering problem (see [Example elitism select the best], page 83).

Since the containers used come from the `STL library`, to store the data, it is possible to reuse the functions provided by the library `<algorithm>` specially designed to be used in ranges of elements, such as `std::max` and `std::sort` that are used to implement the elitism.

3.1.6 Crossover operator

To apply the crossover operator LEAC has two functions to iterate over the population and mating pool obtained after applying the parent selector

[Function]

```
void gaiterator::crossover
          (INPUT_ITERATOR aiiterator_instfirstParent,
           const INPUT_ITERATOR aiiterator_instlastParent,
           INPUT_ITERATOR aiiterator_instfirstChild,
           const INPUT_ITERATOR aiiterator_instlastChild,
           const GENETIC_OPERATOR genetic_operator)
```

Select a pair of parents and children consecutively from their containers. See [Example gaiterator::crossover], page 86,

[Function]

```
void gaiterator::crossoverRandSelect
          (INPUT_ITERATOR aiiterator_instfirstParent,
```

```
const INPUT_ITERATOR aiiterator_instlastParent,
INPUT_ITERATOR aiiterator_instfirstChild,
const INPUT_ITERATOR aiiterator_instlastChild,
const GENETIC_OPERATOR genetic_operator)
```

Select a pair of parents and children randomly and consecutively respectively from their containers. See [Example gaiterator::crossoverRandSelect], page 100,

Within the iterator function, the crossover operator is applied. LEAC has implemented several operators of crossover and mutation proposed in the literature. These are organized by packages as shown in the Figure 3.4. The way in which genetic operators are classified in LEAC is *cluster-oriented* or *nonoriented operators*. They are also classified according to their coding *binary*, *integer*, or *real* encodings. For a classification of operators see Hruschka et al. [HCFdC09]. From the point of view of the implementation, it is also possible to classify those that are programmed in a generic way and those that depend on the type of data.

The crossover operators used by the algorithms implemented in section Appendix A [Example source code], page 75, they use `Single-point` and `Two-point` Crossover described below:

[Function]

```
void gagenericop::onePointCrossover
          (gaencode::ChromFixedLength<T_GENE,T_METRIC> &aochrom_child1,
           gaencode::ChromFixedLength<T_GENE,T_METRIC> &aochrom_child2,
           const gaencode::ChromFixedLength<T_GENE,T_METRIC> &aichrom_parent1,
           const gaencode::ChromFixedLength<T_GENE,T_METRIC> &aichrom_parent2)
```

This crossover operator is parameterized for a chromosome of any type of data, internally generates a random number in the interval [1,length of chromosome -2], which is used to make the combination. See [Example gagenericop::onePointCrossover], page 86,

[Function]

```
void gabinaryop::onePointDistCrossover
          (mat::BitMatrix<T_BITSIZE>  &aobitmatrix_child1,
           mat::BitMatrix<T_BITSIZE> &aobitmatrix_child2,
           mat::BitMatrix<T_BITSIZE> &aibitmatrix_parent1,
           mat::BitMatrix<T_BITSIZE> &aibitmatrix_parent2)
```

This is the extension of operator `two-point crossover` for the case of `Bit-Matrix` See [Example gabinaryop::onePointDistCrossover], page 100,

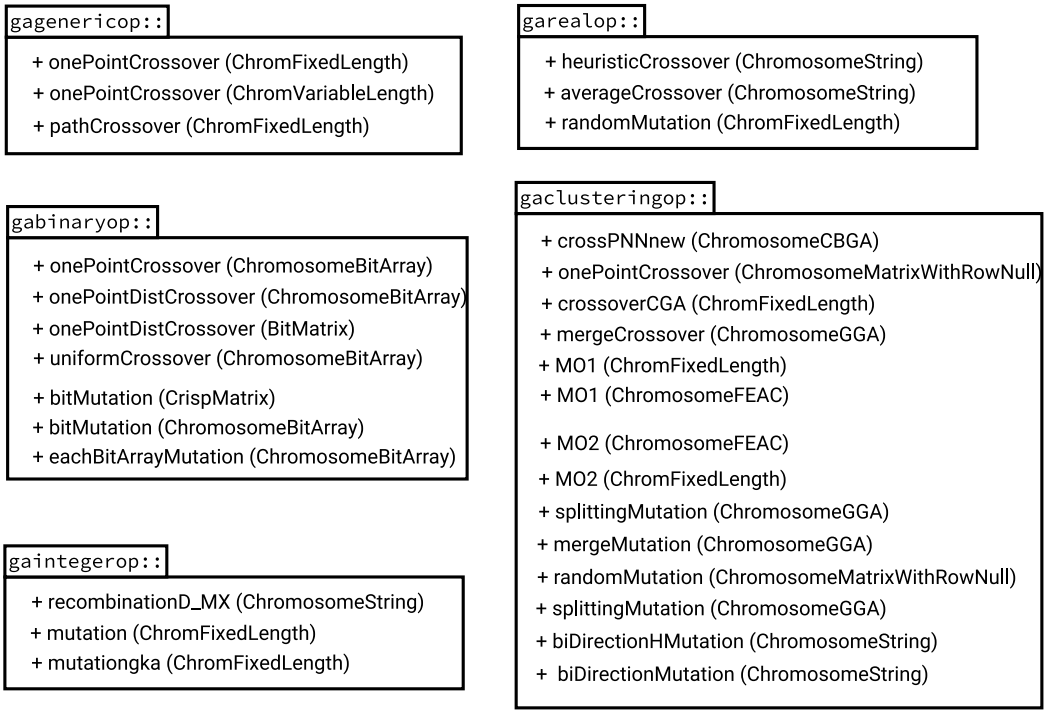

Figure 3.4: Packages of genetic operators provided by LEAC. The type of chromosome received by the operator is indicated in parentheses

3.1.7 Mutation operator

Several operators for mutation are proposed in the literature on evolutionary algorithms for clustering. In the Figure 3.4, in the same way that the crossover operators are grouped, the mutation operators are also grouped by packets.

To illustrate how the mutation operator is applied, the two operators that are used in the illustrative examples in Appendix A [Example source code], page 75, are described below.

Bezdek et al. [BBHB94] use a mutation operator to change the one bit from one row to another in an aletory manner in a *crisp matrix*:

[Function]

```
void gabinaryop::bitMutation
        (mat::CrispMatrix<T_BITSIZE,T_CLUSTERIDX> &aiobitcrispmatrix_chrom)
```

See [Example gabinaryop::bitMutation], page 101.

Bandyopadhyay and Maulik [BM02a] they propose a mutation operator for chromosomes [centroid-based], page 13. It is located in the file `ga_clustering_operator.hpp` and called here as `gaclusteringop::biDirectionHMutation`:

$$\text{mutate}(g_{jl}) = \begin{cases} g_{jl} + \delta \times (\max(x_l) - g_{jl}) & \text{if } \delta \geq 0, \text{ for } j = 1, 2, .., k \text{ and } l = 1, 2, ...d, \\ g_{jl} + \delta \times (g_{jl} - \min(x_l)) & \text{if } \delta < 0. \end{cases}$$

Where δ is a random number in the interval $[-R, +R]$:

$$R = \begin{cases} \frac{M - M_{min}}{M_{max} - M_{min}} & \text{if } M_{max} > M, \\ 1 & \text{if } M_{min} = M_{max}. \end{cases}$$

M_{min} and M_{max} are the minimum and maximum values of the clustering metric, respectively, in the current population. M is the clustering metric value of the current chromosome that must be mutated.

[Function]

```
void gaclusteringop::biDirectionHMutation
          (gaencode::ChromosomeString<T_GENE,T_METRIC> aochrom_offspring,
           const T_METRIC airt_minObjetiveFunc,
           const T_METRIC airt_maxObjetiveFunc,
           const T_GENE* aiarrayt_minFeatures,
           const T_GENE* aiarrayt_maxFeatures)
```

See [Example gaclusteringop::biDirectionHMutation], page 89,

3.1.8 Update or replacement of the population

In this step, the new individuals resulting from the crossover, the mutated and those who managed to survive without applying an operator, will form the new population.

For most of the algorithms studied in this document, the offspring automatically replace their parents. Then, in the process of crossover, individuals move from the mating pool to the population. For example, in the implementation of the KGA algorithm of [BM02a], the `gaiterator::crossover` iteration function, the input parameter is *lvectorchromfixleng_matingPool* and the output parameter is *lvectorchromfixleng_population*, in this way the replacement is achieved, whether the crossover is done or not by the assignment in the block of `else`, see [Example remplace population], page 86.

Some algorithms select the best from the previous population and the best offspring, both merge to form a new population. For example Bezdek et al. [BBHB94]. See [Example replace the population with the best], page 102.

To assign or copy individuals, the chromosome objects of the class diagram in Figure 3.1, implement assignment operator `(operator=(const class_name &)` and move assignment operator `(operator=(class_name &&)`, as well as their respective constructors, which allows container handling and code readability.

3.1.9 Other parameters

Some EAs for clustering to use local search. The `k-means algorithm` is more popular, this is a procedure of fine-tuning of maximum descent. LEAC includes an extensive list of functions for local search, some of which are discussed below:

[Function]

```
void clusteringop::updateCentroids
          (T_CLUSTERIDX &aocidx_numClusterNull,
           mat::MatrixRow<T_FEATURE> &aiomatrixt_centroids,
           mat::MatrixRow<T_FEATURE_SUM> &aomatrixt_sumInstancesCluster,
           std::vector<T_INSTANCES_CLUSTER_K> &aovectort_numInstancesInClusterK,
           INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           const dist::Dist<T_DIST,T_FEATURE> &aifunc2p_dist)
```

With rule (3.2) each point is assigned x_i a the clusters C_j. See [Equation (3.2)], page 13. Update the centroids $\mu_i^* = 1/n_j \sum_{x_i \in C_j} x_i, j = 1, 2, ..., k,$

where n_j is the number of points in cluster C_i. Returns the new centroids, the sum of instances and their number per cluster. See [Example clusteringop::updateCentroids], page 81,

Krishna and Murty [KM99], propose the use of the `k-means algorithm` as operator:

[Function]

```
T_CLUSTERIDX clusteringop::kmeansoperator
          (T_CLUSTERIDX *aioarraycidx_memberShip,
           mat::MatrixRow<T_FEATURE> &aomatrixt_centroids,
           mat::MatrixRow<T_FEATURE_SUM> &aomatrixt_sumInstancesCluster,
           std::vector<T_INSTANCES_CLUSTER_K> &aovectort_numInstancesInClusterK,
           INPUT_ITERATOR aiiterator_instfirst,
           const INPUT_ITERATOR aiiterator_instlast,
           dist::Dist<T_DIST,T_FEATURE> &aifunc2p_dist)
```

For a given partition in an array of labels, it performs an update using the `k-means algorithm`. Returns the membership labels, centroids, sum of instances and number of instances in each cluster.

Sheng and Liu [SL04], propose a *local heuristic search* for *k-medoids*, described below:

Input:

m_j medoids, where $j = 1..k$ and k number of clusters

p is the size of the search subsets (C_{subset})

Output: m_j^* medoids, with SEDmedoid(m_j^*) $\leq$ SEDmedoid(m_j), and SEDmedoid is the equation (3.8).

For each cluster C_j, it is found the most representative object:

1. Assign each object in $x_i \in X$, where X is data set to the cluster C_j with the closest medoid.
2. For each cluster C_j, repeat until the k medoids does not change:
   - Choose a subset $C_{subset} \in C_j$ the corresponds to m_j and its p nearest neighbors of m_j.
   - Calulate the new medoid

$$m_j^* = \underset{x_i \in C_{subset}}{\arg\min} \sum_{x_i' \in C_j} \|x_i - x_i'\|$$

   - if m_j is different from m_j^* replace with the new medoid
3. Repeat step 1 and 2 until k medoids do not change

The function that implements the local heuristic search for *k-medoids* is the following function:

[Function]

```
void clusteringop::updateMedoids
          (uintidx      aoarrayuiidx_medoids,
           T_CLUSTERIDX aicidx_numClusterK,
           uintidx      aiuiidx_nearestNeighborsP,
           mat::MatrixTriang<T_DIST> &aimatrixtriagt_dissimilarity)
```

Update k-medoids, based on [SL04]

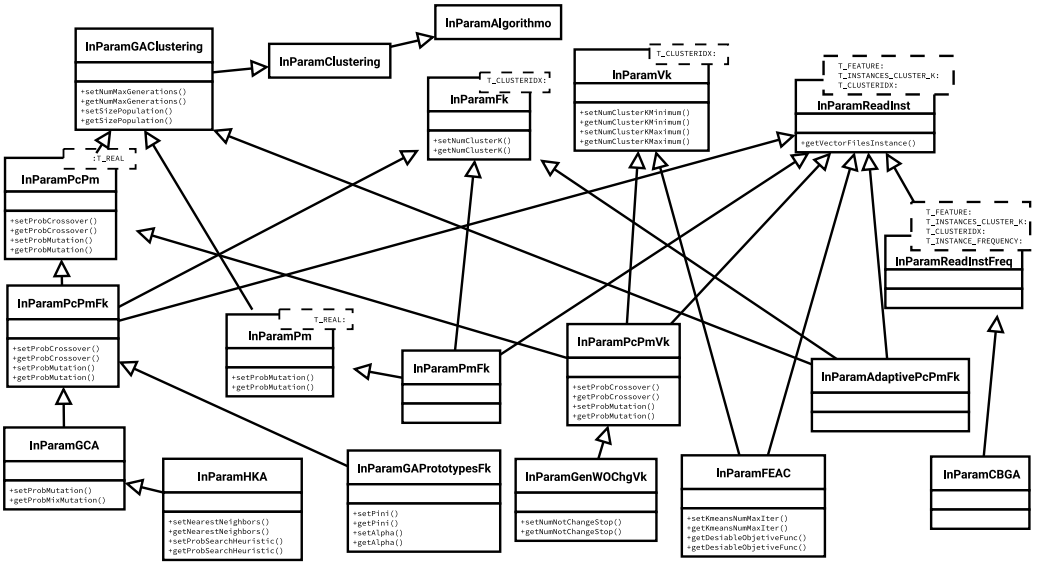

Figure 3.5: Classes used for the input of parameters for the implementation of an algorithm, Nomenclatura, crossover probability (Pc), mutation probability (Pm), fixed k-clusters (Fk), variable k-clusters (Vk).The classes in the lower level correspond to the specific parameters of some algorithms encoded with LEAC.

3.2 Make an executable for a new algorithm

There are two options to make an implementation of a new algorithm. The first is simply to include all the necessary operations in the main file. The second option is to use the scheme followed in the implementations included in the `eac` directory, you must create a file with a structure similar to the one included in Annex (Section A.1 [KGA algorithm], page 75). In both cases, the configuration scheme of an evolutionary algorithm must be followed see Section 3.1 [Implementation of an algorithm], page 11. The steps you need to compile the executable file for a new algorithm are described below:

1. Create the source file and include the LEAC library `#include <leac.hpp>`
2. Select the data type of the attributes of the instances of the data set with which you will work, this is independent of the coding of the chromosomes: datatype_instance_real.hpp or datatype_instance_integer.hpp. The files suggest the types of data you can use for the compatibility of variable types, including only one in your source file. '`#include <datatype_instance_real.hpp>`' for real or '`#include <datatype_instance_integer.hpp>`' for integer or define your own data types.
3. Select a class from Figure 3.5 to read the parameters of the new algorithm or define a new class with the parameters specified in the algorithm to be implemented, you can extend from an existing class. For example for `InParamPcPmFk` include the file '`#include <inparam_pcpmfk.hpp>`'.

```
                                                [Constructor on inout::InParamPcPmFk]
InParamPcPmFk<T_CLUSTERIDX,T_REAL,T_FEATURE,T_FEATURE_SUM,T_INSTANCES_CLUSTER_K>
            (const std::string& ais_algorithmoName,
```

```
                const std::string& ais_algorithmoAuthor,
                InParam_algTypeOut aiato_algTypeOut,
                int aii_opNorm)
```

4. Defining an instance of parameter entry class and assign the default values.

```
/*INPUT: PARAMETER
   */
  inout::InParamPcPmFk
    <DATATYPE_CLUSTERIDX,
     DATATYPE_REAL,
     DATATYPE_FEATURE,
     DATATYPE_FEATURE_SUM,
     DATATYPE_INSTANCES_CLUSTER_K
     >
    linparam_ClusteringGA
    ("Acronym of the algorithm",
     "Algorithm Authors",
     inout::CENTROIDS,
     INPARAMCLUSTERING_DISTANCE_EUCLIDEAN
     );

  linparam_ClusteringGA.setNumMaxGenerations(1000);
  linparam_ClusteringGA.setSizePopulation(50);
  linparam_ClusteringGA.setProbCrossover(0.8);
  linparam_ClusteringGA.setProbMutation(0.001);
```

When an instance of the `InParamPcPmFk` object is created, the data types used to perform all the calculations are defined, so it is not necessary to specify the data types for the functions that this `InParamPcPmFk` object is used, allowing you to avoid possible errors or redundancies.

5. Assign the default values or simply use this way to read the parameters, for example:

```
linparam_ClusteringGA.setNumMaxGenerations(1000);
linparam_ClusteringGA.setSizePopulation(50);
linparam_ClusteringGA.setProbCrossover(0.8);
linparam_ClusteringGA.setProbMutation(0.001);
```

6. Optionally, if you wish, you can read the input parameters online using the functions provided in the LEAC library. Declare the sentence '`#include <inparamclustering_getparameter.hpp>`' after the file of the parameter class.

   If you are using a class of the class diagram (Figure 3.5), go to [Step 7], page 39. Otherwise do the following. Define the directive to the preprocessor in the file of your new parameter class. The name of the directive must be different from the ones that are defined, for example:

```
#define __INPARAM_MY_PCPMFK__
```

   If you extend your parameter class from an existing one in Figure 3.5, you must override the definition of the directive that is defined in the base class, for example, instead of simply including the previous statement, you must include the following two statements in your file of parameter class:

```
#undef  __INPARAM_PCPMFK__
#define __INPARAM_MY_PCPMFK__
```

   Include similar code blocks in the inparamclustering_getparameter.hpp file, as shown in the following paragraphs, to print the help and read the parameters of the new algorithm.

a. Find the region of the `inparamclustering_usage` functions, add your directive that was previously defined, and include your definition of the function with the parameter of the new class defined by you (in bold), as shown in the following example:

```
#ifdef __INPARAM_MY_PCPMFK__
template<typename T_CLUSTERIDX,
  typename T_REAL,
   typename T_FEATURE,
   typename T_FEATURE_SUM,
   typename T_INSTANCES_CLUSTER_K
   >
void
inparamclustering_usage
(char *argv0,
 InParamMyClassPcPmFk
 <T_CLUSTERIDX,
 T_REAL,
 T_FEATURE,
 T_FEATURE_SUM,
 T_INSTANCES_CLUSTER_K
 >                    &aoipc_inParamClustering
)
#endif
```

b. Search the area to print help messages and write a block of code similar to the following:

```
#ifdef __INPARAM_MY_PCPMFK__
  std::cout << "      --number-clusters[=NUMBER]\n"
            << "                                number of clusters [NUMBER=
            << aoipc_inParamClustering.getNumClusterK() << "]\n";
  std::cout << "      --generations[=NUMBER]  number of generations or iterations\n"
            << "                                 [NUMBER="
            << aoipc_inParamClustering.getNumMaxGenerations()
            << "]\n";
  std::cout << "      --population-size[=NUMBER]\n"
            << "                                size of population [NUMBER="
            << aoipc_inParamClustering.getSizePopulation()
            << "]\n";
  std::cout << "      --crossover-probability[=NUMBER]\n"
            << "                                real number in the interval [0.25, 1]\n"
            << "                                  [NUMBER="
            << aoipc_inParamClustering.getProbCrossover()
            << "]\n";
  std::cout << "      --mutation-probability[=NUMBER]\n"
            << "                                real number in the interval [0, 0.5]\n"
            << "                                  [NUMBER="
            << aoipc_inParamClustering.getProbMutation()
            << "]\n";
#endif
```

c. Find the `inparamclustering_getParameter` functions and and defines its own function similar to the previous one:

```
#ifdef __INPARAM_MY_PCPMFK__
template<typename T_CLUSTERIDX,
         typename T_REAL,
         typename T_FEATURE,
         typename T_FEATURE_SUM,
         typename T_INSTANCES_CLUSTER_K
```

```
        >
void
inparamclustering_getParameter
(InParamMyClassPcPmFk
 <T_CLUSTERIDX,
 T_REAL,
 T_FEATURE,
 T_FEATURE_SUM,
 T_INSTANCES_CLUSTER_K
 >
 &aoipc_inParamClustering,
 int  argc,
 char **argv
 )
#endif
```

d. In the `inparamclustering_getParameter` function, find the array where the parameter labels are defined and define a new array with the labels of the parameters for the new algorithm, as shown in the following example:

```
#ifdef __INPARAM_MY_PCPMFK__
  const char   *lastr_myInParamPcPmFk[] =
    {"number-clusters",
     "population-size",
     "crossover-probability",
     "mutation-probability",
     "generations",
     (char *) NULL
    };
#endif
```

e. In `long_options` array, define your parameters with the options shown below

```
#ifdef __INPARAM_MY_PCPMFK__
      {"number-clusters",         required_argument, 0, 0},
      {"population-size",         required_argument, 0, 0},
      {"crossover-probability",   required_argument, 0, 0},
      {"mutation-probability",    required_argument, 0, 0},
      {"generations",             required_argument, 0, 0},
#endif
```

f. Finally, add the necessary code to obtain the parameters

```
#ifdef __INPARAM_MY_PCPMFK__
      else if ( strcmp /*number-clusters*/
           (long_options[option_index].name,
            lastr_inParamPcPmFk[0] ) == 0 )
        {
          T_CLUSTERIDX lmcidxT_numClusterK;
          liss_stringstream.clear();
          liss_stringstream.str(optarg);
          liss_stringstream >> lmcidxT_numClusterK;
          aoipc_inParamClustering.setNumClusterK(lmcidxT_numClusterK);
        }
      else if ( strcmp /*population-size*/
                (long_options[option_index].name,
                 lastr_inParamPcPmFk[1]) == 0
                )
        {
          liss_stringstream.clear();
          liss_stringstream.str(optarg);
```

```
            liss_stringstream >> luintidx_read;
            aoipc_inParamClustering.setSizePopulation(luintidx_read);
          }
        else if ( strcmp /*crossover-probability*/
                  (long_options[option_index].name,
                   lastr_inParamPcPmFk[2]) == 0
                  )
          {
            T_REAL  lT_readProbabilityCrossover;

            liss_stringstream.clear();
            liss_stringstream.str(optarg);
            liss_stringstream >> lT_readProbabilityCrossover;
            aoipc_inParamClustering.setProbCrossover(lT_readProbabilityCrossover);
          }
        else if ( strcmp /*mutation-probability*/
                  (long_options[option_index].name,
                   lastr_inParamPcPmFk[3]) == 0
                  )
          {
            T_REAL  lT_readProbabilityMutation;

            liss_stringstream.clear();
            liss_stringstream.str(optarg);
            liss_stringstream >> lT_readProbabilityMutation;
            aoipc_inParamClustering.setProbMutation(lT_readProbabilityMutation);
          }
        else if ( strcmp /*generations*/
                  (long_options[option_index].name,
                   lastr_inParamPcPmFk[4]) == 0
                  )
          {
            COMMON_IDOMAIN lT_readNumMaxGenerations;

            liss_stringstream.clear();
            liss_stringstream.str(optarg);
            liss_stringstream >> lT_readNumMaxGenerations;
            aoipc_inParamClustering.setNumMaxGenerations(lT_readNumMaxGenerations);
          }
        else {
          aoipc_inParamClustering.errorArgument
            (argv[0],
             long_options[option_index].name,
             lastr_inParamPcPmFk
             );
        }
    #endif
```

7. If you completed the previous step, you can now invoke the `inparamclustering_getParameter` function, to modify or complete the parameters online:

   ```
   inparamclustering_getParameter(linparam_ClusteringGA, argc, argv);
   ```

   Otherwise just read the parameters as in [Step 5], page 36.

8. Change the seed of random numbers, with some of the following functions. The first is used to generate a new string and the second is used to repeat an experiment with an existing string.

   [Function]

   ```
   std::string randomext::setSeed (const unsigned int aiu_numSeed = 8)
   ```

Generates a random string to use as a seed, it also returns the string to be used in other executions.

[Function]

```
void randomext::setSeed (std::string aistr_seed_seq)
```

When you have a string as a seed, you assign it with this function.

The generation of random numbers is one of the most important aspects in the convergence to a solution through evolutionary algorithms. LEAC uses the improvements incorporated in the C++11 version with STL library, mersenne_twister_engine is a random number engine based on *Mersenne Twister* algorithm. It produces high quality unsigned integer random numbers of type *UIntType* on the interval $[0, 2^w - 1]$.

The file `random_ext.hpp` creates a global object *gmt19937_eng* of type mersenne_twister_engine, this object is used as a parameter to generate random numbers in a probability distribution, necessary for the implementation of several genetic operators. For your own implementations you can to the object *gmt19937_eng*. For example, first the distribution is instantiated (See [std::uniform_real_distribution], page 77), with the created distribution and with the object *gmt19937_eng* as a parameter, a random number based on the distribution is obtained (See [get a random number], page 86).

9. Read the data set to be processed with any of the following functions:

[Function]

```
std::pair<std::vector<data::Instance<T_FEATURE>* >,
          std::vector<data::Instance<T_FEATURE>* > > inout::data setRead
          (inout::InParamReadInst<T_FEATURE,T_INSTANCES_CLUSTER_K,T_CLUSTERIDX>
           &aiipri_inParamReadInst)
```

[Function]

```
std::pair<std::vector<data::Instance<T_FEATURE>* >,
          std::vector<data::Instance<T_FEATURE>* > > inout::data setReadWithFreq
          (inout::InParamReadInstFreq
           <T_FEATURE,T_INSTANCES_CLUSTER_K,T_CLUSTERIDX,T_INSTANCE_FREQUENCY>
           &aiipri_inParamReadInstWithFreq)
```

The following example shows how to read a data set, which can have a class attribute and also test data

```
auto lpairvec_data set =  inout::data setRead(linparam_ClusteringGA);
```

10. Directly include genetic operators in the main function or invoke directly the function that has the implementation of the new algorithm in the main function. At the end of writing your source code, your `main` function should have something similar to the following code:

```
#include <leac.hpp>
#include <datatype_instance_real.hpp>
#include <inparam_pcpmfk.hpp>
#include <inparamclustering_getparameter.hpp>
#include <instances_read.hpp>

#include "kga_fkcentroid.hpp"

int main(int argc, char **argv)
{
  /*INPUT: PARAMETER
   */
  inout::InParamPcPmFk
```

```
  <DATATYPE_CLUSTERIDX,
   DATATYPE_REAL,
   DATATYPE_FEATURE,
   DATATYPE_FEATURE_SUM,
   DATATYPE_INSTANCES_CLUSTER_K
   >
  linparam_ClusteringGA
  ("KGA",
   "Bandyopadhyay and Maulik 2002",
   inout::CENTROIDS,
   INPARAMCLUSTERING_DISTANCE_EUCLIDEAN
   );

linparam_ClusteringGA.setNumMaxGenerations(1000);
linparam_ClusteringGA.setSizePopulation(50);
linparam_ClusteringGA.setProbCrossover(0.8);
linparam_ClusteringGA.setProbMutation(0.001);

inparamclustering_getParameter(linparam_ClusteringGA, argc, argv);

if ( linparam_ClusteringGA.getRandomSeed().size() == 0 ) {
   std::string lstr_seed_seq = randomext::setSeed();
   linparam_ClusteringGA.setRandomSeed( lstr_seed_seq );
}
else {
   randomext::setSeed(linparam_ClusteringGA.getRandomSeed());
}

auto lpairvec_data set =  inout::data setRead(linparam_ClusteringGA);

dist::Euclidean<DATATYPE_REAL,DATATYPE_FEATURE> funct2p_dist;

inout::OutParamGAC
  <DATATYPE_REAL,
   DATATYPE_CLUSTERIDX>
  loop_outParamGAC(inout::SSE);

auto lchrom_best =
  eac::kga_fkcentroid
  (loop_outParamGAC,
   linparam_ClusteringGA,
   lpairvec_data set.first.begin(),
   lpairvec_data set.first.end(),
   funct2p_dist
   );

std::cout << "chrom_best: " << lchrom_best << std::endl;

return 0;
}
```

11. With the source files completed, compile your new algorithm, with the following command

    '`g++ -std=c++11 -D __VERBOSE_YES -I ../include -I ../include_inout/ -fopenmp kga_main.cpp -o kga`'

    The '`-D __VERBOSE_YES`' option is optional to debug your program. For the `g++` version on the Mac OS X®, it should be (`>` = 4.8.5), example, use the `g++-mp-5` version.

If you did not install LEAC as described in section Chapter 2 [Get and Install LEAC software], page 5, and you want to compile an evolutionary algorithm using LEAC like the one shown in this section, you only need to install the gcc compiler, [step 3], page 5, for for Windows® and [step 7], page 6, for GNU/Linux systems and Mac OS X®.

You can also compile your application with the other evolutionary algorithms provided by LEAC, adding the commands in the Makefile file for your new algorithm.

3.3 Using implemented algorithms

This section shows how to use the programs located in the `eac` directory to find a solution to the clustering problem. The algorithms are categorized according to three aspects: if the number the clusters is fixed or variable, the coding of the solution by the algorithm (See Section 3.1.1 [Encoding criterion], page 11) and the similarity function (See Section 3.1.3.2 [Unsupervised measures], page 18).

To complete the practical examples, you should have completed the installation described in the Chapter 2 [Get and Install LEAC software], page 5, and have access to the `leac/bin` and `leac/data` directories. Add directly to the *PATH* variable, from a terminal (`cmd`) the directory where the binary or executable files are, for example: '`export PATH=$PATH:/home/user/leac/bin`' for GNU/Linux systems and Mac OS X®, or '`SET PATH=%PATH%;C:\leac\bin;`' for Windows® systems. For data set files '`cd c:\leac\data`' for Windows® or '`cd leac/data`' for GNU/Linux. You can have the directories in a different route for this, just change the route.

The nomenclature used for the name of the programs is based on the three aspects described above. Name of the algorithm on which the program is based, fixed or variable k and the coding used:

```
name_[fk|vk]coding
```

All EAC programs are executed from a terminal with online parameters. With the parameter '`--help`', You will get a description of the different options, there are two types of options, those that are common for all the programs and the particular ones of each algorithm, these last ones are shown after the message [Particular options of the algorithm], page 43,

For example, here are shown some execution '`kga_fkcentroid --help`'. This is the output of the command

```
Usage: ./kga_fkcentroid [OPTION]
       About groups of instances for a set K as well as statistics
       of the algorithm used

  -i, --instances=FILE or DIRECTORY
                              file or directory containing data of instances
                                to be clustered
  -x, --select-instances[=PREFIX]
                              if instances is directory search files with
                                prefix for training (eg. iris-10-1tra.dat,
                                iris-10-2tra.dat,... PREFIX=tra.dat)
  -t, --test[=FILE or PREFIX] if instances is directory search files with
                                prefix for test (eg. iris-10-1tst.dat,
                                iris-10-2tst.dat,... PREFIX=tst.dat),
                                in other case only name file
  -b  --format-file[=NAME]    uci, or keel, by default uci
```

```
  -h, --with-header[=yes/no]  file contains names of instances or a header,
                                by default is no
  -u, --number-instances[=NUMBER]
                              the number of instances the file contains
                                instances, if not specified file is obtained
  -a, --select-attributes[=ARG]
                              select the attributes to be processed for
                                example, "1-2,4" by default all. Also
                                used to specify the  number of dimensions
                                of the instances, unless specified file is
                                obtained instances
  -d, --delimit-attributes=[ARG]
                              separated file by default ","
  -c, --class-column[=NUMBER] input file of instances has a class assigned
                                in the column [NUMBER=undefined]
  -e, --cluster-column[=NUMBER]
                              input file of instances has a cluster assigned
                                in the column [NUMBER=undefined]
  -l, --idinstances-column[=NUMBER]
                              the input file instance is assigned a column
                                instance identifier [NUMBER=undefined]
  -f, --freq-instances-column[=NUMBER]
                              the input file instance is assigned a column
                                frequency instances [NUMBER=undefined]
  -r, --number-runs[=NUMBER]  number of runs or repetitions of the algorithm
                                (by default [NUMBER=1])
  -R, --runtime-filename=[FILE]
                              out file of times run
  -n  --distance[=NAME]       euclidean, euclidean_sq, euclidean_induced,
                                diagonal_induced, or mahalonobis_induced,
                                by default euclidean
  -z, --random-seed[=NUMBER]  string with integer number seed by, default
                                is random
  -w, --max-execution-time[=NUMBER]
                              real number for max execution time in seconds
                                by default is 36000
  -C, --centroids-outfile[=FILE]
                              print centroids,standard output FILE=stdout
      --centroids-format[=yes/no]
                              print the matrices by rows and columns,
                                by default is no
  -M, --membership-outfile[=FILE]
                              print membership of the instances,
                                 standard output FILE=stdout
  -T, --partitionstable-outfile[=FILE]
                              print partitions table of the instances,
                                standard output FILE=stdout
      --table-format[=yes/no]
                              print the partitions table by rows and
                                columns, by default is no
  -P, --gnuplot=FILE          file of  gnuplot to graphics result
                                (compiling only with WITHOUT_PLOT_STAT)
  -y, --gnuplot-styles=WORD   plot graphics with: points, lines,
                                linespoints, and dot [ARG=linespoints]


Particular options of the algorithm KGA
  based on Bandyopadhyay and Maulik 2002
```

```
      --number-clusters[=NUMBER]
                              number of clusters [NUMBER=3]
      --generations[=NUMBER]  number of generations or iterations
                               [NUMBER=1000]
      --population-size[=NUMBER]
                              size of population [NUMBER=50]
      --crossover-probability[=NUMBER]
                              real number in the interval [0.25, 1]
                                [NUMBER=0.8]
      --mutation-probability[=NUMBER]
                              real number in the interval [0, 0.5]
                                [NUMBER=0.001]

  -v, --verbose[=NUMBER]      explain what is being done (compiled with
                                VERBOSE=yes)
                              NUMBER=[-1,..,9999] Quiet level -1 not,
                                verbose, default=-1
  -q, --bar-progress          progress bar printing,  default is not
  -?, --help                  help
```

To run any of the programs successfully, the only mandatory parameter is '`-i, --instance`' (data set that should be processed), all other parameters are taken by default proposed by the authors of the algorithms.

For all the evolutionary algorithm studied in this work, the parameters they share are:

`--generations`
`--population-size`
: They only vary in the number of generations and in the number of chromosome proposed by the different authors.

The following sections describe the evolutionary algorithm currently included in the LEAC library, based on taxonomy proposed by Hruschka et al. [HCFdC09], *k-fixed* vs. *k-variable* and the coding they use to represent the chromosomes, see Table 1.1. The description of each of the algorithms is made based on the distinctive parameters, which is significant to know how to work, for details, it is recommended to consult the included bibliofrafia. We also include practical examples of clustering applications with real data sets provided by UC Irvine Machine Learning Repository.

3.3.1 Genetic algorithms for fixed k-clusters

3.3.1.1 Based on the centroids

The GAs in this section seek to find the centroids of the cluster for a user-defined k. If you are looking for precision to locate the centroid of the clusters, these algorithms are a good option. They encode the solutions as a string with the coordinates of the centroids consecutively, see [real encoding], page 13. The GAs implemented in this class are listed below:

1. **gas_fkcentroid**

   It based is on [MB00], it optimizes SED measure (see [SED], page 18). Use `gagenericop::onePointCrossover` and a proposed mutation operator, encoded in the function `garealop::randomMutation`, both operators apply with fixed probability.

   Parameters of the algorithm:

`--crossover-probability[=NUMBER]`
real number in the interval `[0.25, 1]` `[NUMBER=0.8]`

`--mutation-probability[=NUMBER]`
real number in the interval `[0, 0.5]` `[NUMBER=0.001]`

Execution samples:

- 'gas_fkcentroid -i iris.data -a "1-4"'
- 'gas_fkcentroid -i iris.data -a "1-4" -c 5 --number-clusters 3 --generations 100 --population-size 100 --crossover-probability 0.8 --mutation-probability 0.2 -C stdout -T stdout --table-format yes -M stdout'

2. **kga_fkcentroid**

   It based is on [BM02a], it optimizes SED measure (see [SED], page 18). Uses the traditional crossover operator `gagenericop::onePointCrossover` and proposes a new mutation operator `gaclusteringop::biDirectionHMutation`, used in other algorithms.

   Parameters of the algorithm:

   `--crossover-probability[=NUMBER]`
   real number in the interval `[0.25, 1]` `[NUMBER=0.8]`

   `--mutation-probability[=NUMBER]`
   real number in the interval `[0, 0.5]` `[NUMBER=0.001]`

   Execution samples: see [Illustrative execution samples: Libras Movement Data Set], page 49.

3. **gagr_fkcentroid**

   It based is on [CZZ09], it optimizes SSE measure (see [SSE], page 18). For the crossover of the chromosomes uses two operators `gagenericop::GAGRdist` and `garealop::heuristicCrossover`, and for the mutation `gaclusteringop::biDirectionHMutation`. For the application of the operators, the algorithm uses an adaptive probability. Therefore, the parameters that it receives are only those shared by the evolutionary algorithm `--generations` and `--population-size`.

   Execution samples: see [Illustrative execution samples: Wine data set], page 46,

4. **cbga_fkcentroid_int** and **cbga_fkcentroid**

   It based is on [FKKN97], it optimizes Distortion measure (see [Distortion], page 20). This algorithm was initially applied for the processing of images, to work with integers, due to the parameterization of the functions in LEAC, it is possible to obtain another version for the domain of the real numbers.

   Use the `gaclusteringop::crossPNNnew` crossing operator and the mutation `clusteringop::randomInitialize`. In [FKKN97] they propose different selection methods, for now only `elitist1` is implemented, which is the one that gives the best results.

   Parameters of the algorithm:

   `--mutation-probability[=NUMBER]`
   real number in the interval `[0, 1.0]` `[NUMBER=0.01]`

`--select-method[=NAME]`
roulette, elitist1, elitist2, or zigzag, by default elitist1

`--gla-iterations[=NUMBER]`
number of GLA iterations `[NUMBER=0]`

Execution samples:

- '`cbga_fkcentroid -i iris.data -a "1-4"`'
- '`cbga_fkcentroid -i iris.data -a "1-4" -c 5 --number-clusters 3 --generations 100 --population-size 100 -C stdout -T stdout --table-format yes -M stdout`'

Illustrative execution samples: Wine data set

Next, we describe the steps to execute GAGR algorithm based on [CZZ09] and discuss the parameters that can be used to carry out an experimental study. Something similar would be included for the other algorithms.

For the following example we will analyze the wine data set. First you must download the wine.data file and store it in the `data` directory, all data sets used in the following illustrative examples should be stored in this directory.

Since the domain of the attributes is different for the wine.data, it is convenient to standardize them, for this the program `stdvar_milligan_cooper1988` it based is on [MC88] is available as support for the normalization of the data set. Before executing the commands, make sure you have access to the executable programs and the data, see [configuration of the PATH variable], page 42.

'`stdvar_milligan_cooper1988 -i wine.data -a "2-14" -c 1 --std-var Z1 > wine_std.data`'

The Wine data set includes 13 variables from column 2 to column 14 and the class on column 1, therefore, the `-a` or `--select-attributes` parameter is `"2-14"`, the class is in column 1, the parameter `-c` or `--class-column` it must be assigned to 1. The transformation that applies to the variables in this case is *z-score* $Z_1 = (x_{il} - \bar{x}_{*l})/s_{*l}$, where x_{il} is the original data value, and $\bar{x}_{*l}$ and s_{*l} are the sample mean and standard deviation, respectively.

Next, algorithm GAGR [CZZ09] will be run, for the normalized data

'`gagr_fkcentroid -i wine_std.data -a "1-13" -c 14 --number-clusters 3 -C stdout --centroids-format yes -M stdout`'

The attributes are now in columns `"1-13"` and the class in column `14`, because the `stdvar_milligan_cooper1988` program in the output file writes the class in the last column. Parameters with uppercase letters are used for the output. '`-C stdout`' indicates that the output of the centroids found by the program is formatted with '`--centroids-format yes`'. The parameter '`-M stdout`' generates a string of the membership labels of the instances in a cluster calculated with equation (3.2)

A possible result for a partition of three clusters ('`--number-clusters 3`') by the GAGR algorithm is

```
IN:
    Algorithmo name: GAGR
           Based on: Dong-Xia Chang and Xian-Da Zhang and Chang-Wen Zheng 2009
```

```
        Metric used: SSE

           Data set: wine_std.data
Number of instances: 178
         Dimensions: 13

       Random seed: 2330791922 722590507 2437171925 3046485563 2392633958 2970292677
       3353564338 489582404

OUT:

CROMOSOME:BEST:objetive,1274.2,fitness,0.000784804:0.160439,0.88665,0.212095,0.571924,
-0.126608,-0.99909,-1.23221,0.753954,-0.734194,0.955193,-1.17759,-1.28076,-0.396717,
0.894248,-0.293909,0.297414,-0.676197,0.580441,0.882434,0.944615,-0.589229,0.603343,
0.184902,0.44739,0.758496,1.14392,-0.890931,-0.401072,-0.46444,0.103152,-0.429416,
-0.0743665,0.0441158,-0.0771312,-0.0284121,-0.843627,0.404469,0.225602,-0.691161

       Cluster number (K): 3
                     SSE: 1274.2
                     SED: 450.099
               DB-index: 1.41154
              Silhouette: 0.284617
                     VRC: 69.9363
              CS measure: 0.412355
            Dunn's index: 0.232257
    Execution time (seg): 0.303483
Generations find the best: 46


Centroids:

  Col:    0            1            2            3            4
  Row

    0: 0.160439     0.88665      0.212095     0.571924     -0.126608
    1: 0.894248     -0.293909    0.297414     -0.676197    0.580441
    2: -0.890931    -0.401072    -0.46444     0.103152     -0.429416

  Col:    5            6            7            8            9
  Row

    0: -0.99909     -1.23221     0.753954     -0.734194    0.955193
    1: 0.882434     0.944615     -0.589229    0.603343     0.184902
    2: -0.0743665   0.0441158    -0.0771312   -0.0284121   -0.843627

  Col:    10           11           12
  Row

    0: -1.17759     -1.28076     -0.396717
    1: 0.44739      0.758496     1.14392
    2: 0.404469     0.225602     -0.691161

<MEMBERCLUSTERTRAINING,_algorithmo,GAGR,_author,Dong-Xia Chang and Xian-Da Zhang
and Chang-Wen Zheng 2009,_runnig date,20180430:185944,_number run,1,_runnig date,
20180430:185944,_number run,1,_times run,1,_dataset_training,wine_std.data,_outK,3,
PartitionCentroids[50:0x7fffac569a20],length,178>1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,1,2,2,2,2,2,
```

```
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,1,2,2,2,2,2,2,2,2,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

The '`-C -M`' uppercase options print the centroids and the membership label of each instance to a cluster, respectively. In this case, the '`stdout`' parameter makes the output the standard. The printing format for the centroids is matrixed by the option '`--centroids-format yes`', you can also specify the result in a line of text delimited by special characters, to use the output in another program, as shown below.

In addition to the measure of similarity used by the algorithm, the program calculates other ones that can be used to evaluate the goodness of the grouping. See Section 3.1.3.2 [Unsupervised measures], page 18.

To repeat the same result of the program, you can use the '`-z`' option and as parameter the string that was used as seed to generate the random numbers. Now the output will be sent to the `wine_centroids.data` and `wine_membership.data` files, without the option '`--centroids-format yes`', to visualize the results later:

'`gagr_fkcentroid -i wine_std.data -a "1-13" -c 14 --number-clusters 3 -C wine_centroids.data -M wine_membership.data -z "2330791922 722590507 2437171925 3046485563 2392633958 2970292677 3353564338 489582404"`'

The program `plot_clustering` is another EAC utility, which allows to visualize a data set, with the results obtained from the different programs. This uses Gnuplot, as an example you can run the `plot_clustering` program with the following parameters:

'`plot_clustering -i wine_std.data -a "1-13" -c 14 --projection pca --centroids-infile wine_centroids.data --member-infile wine_membership.data --graphics-outfile wine_cluster`'

You get the `eps` file `wine_cluster1.eps` and which is shown in Figure 3.6. To see the eps files in the case of Windows® you can use the See [epsviewer], page 6, program.

You can also omit the option '`--graphics-outfile`' and the drawing can be manipulated interactively in both `2D` and `3D`, try the following command:

'`plot_clustering -i wine_std.data -a "1-13" -c 14 --projection pca --centroids-infile wine_centroids.data --member-infile wine_membership.data --x-coord 1 --y-coord 2 --z-coord 3`'

Since the objects in the data set have multiple dimensions, it is advisable to use a Principal Component Analysis (PCA) with the option '`--projection pca`'

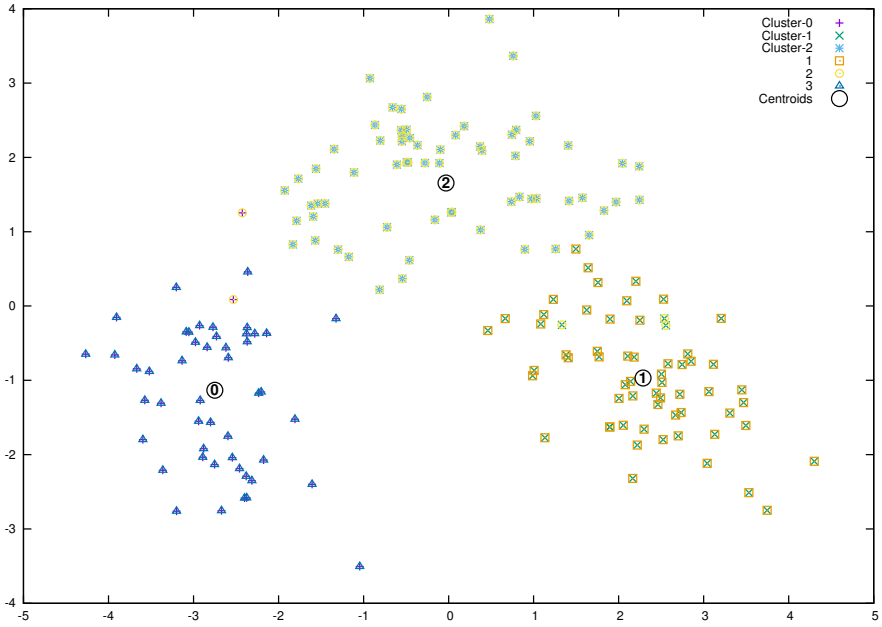

Figure 3.6: Clusters of Wine data set, obtained with `gagr_fkcentroid`

Illustrative execution samples: Libras Movement Data Set

In the following illustrative example, it has two objectives. The first is to obtain the average hand movement of the Libras Movement Data Set. data set, assuming that the data set can be partitioned into 15 clusters. The second objective is to test the quality of the grouping with a set of training data. We will use the KGA algorithm [BM02a]. Before executing the commands, make sure you have access to the executable programs and the data, see [configuration of the PATH variable], page 42.

In all programs, it is possible to use part of the data set for *training* and another part for *test*, to make easier to carry out an experimental study. To determine the ownership of the test data, the equation of the nearest centroid-instance is used equation (3.2).

To demonstrate how to process a data set with training and test data, you can use the movement_libras.data and movement_libras_1.data files.

To obtain better results in the data mining process, it is important to have knowledge about the data set. Libras Movement data set is formed by the coordinates of a two-dimensional curves of the movement of the hand with domain and rango to be betweeen -1 and 1. To have better accuracy, we center on $(0.5, 0, 5)$ each of the curves of the training data set, with the following awk script:

```
#run:
# 'awk -f movement_libras_trans.awk -F "," -v OFS="," 
   movement_libras.data  > movement_libras_trans.data'

    for(i=1;i< NF;i++) sumcoord1+=$i; i++; sumcoord2+=$i ;
    numpoint = (NF-1) / 2.0;
    xd = 0.5 - sumcoord1/numpoint;
```

```
yd = 0.5 - sumcoord2/numpoint;
sumcoord1=0; sumcoord2=0;
for(i=1;i< NF;i++) $i +=  xd;  i++; $i += yd ;
for(i=1; i<=NF; i++) printf "%s",$i (i==NF?ORS:OFS)
```

The `movement_libras_trans.awk` file of the script can be found in the `data` folder. To run the script, type:

'`awk -f movement_libras_trans.awk -F "," -v OFS="," movement_libras.data > movement_libras_trans.data`'

With the transformed data set we apply the KGA algorithm:

'`kga_fkcentroid -i movement_libras_trasn.data -t movement_libras_1.data -a "1-90" -c 91 -C c_movement_libras.data -T stdout --table-format yes --number-cluster 15`'

A possible result of the program is the following:

```
IN:
    Algorithmo name: KGA
           Based on: Bandyopadhyay and Maulik 2002
        Metric used: SED

           Data set: movement_libras_trasn.data
Number of instances: 360
         Dimensions: 90

      Data set test: movement_libras_1.data
Number of instances: 45

       Random seed: 2127474194 2277915873 2997828778 567204173 2395445691
       1861208675 35718978 101314263

OUT:

CROMOSOME:BEST:objetive,209.665,fitness,0.00476951:0.545241,0.779453,
0.544903,0.779338,0.542145,0.778586,0.539438,0.776675,0.534216,0.773666,
...
0.462735,0.320573,0.457353,0.311718,0.452307,0.306383,0.451046,0.302055

       Cluster number (K): 15
                      SED: 209.665
                DB-index: 1.15626
              Silhouette: 0.284789
                      VRC: 87.1936
              CS measure: 1.36284
            Dunn's index: 0.0855026

            Test data SED: 52.8738 Has group without objects
      Test data DB-index: 3.59388
    Test data Silhouette: -0.103324
           Test data VRC: 0.721814
    Test data CS measure: 1.76497
  Test data Dunn's index: 0.0650252
   Execution time (seg): 85.0431
Generations find the best: 56


Partition table:
```

Cluster:	0	1	2	3	4
Class					
1:	0	0	10	0	0
2:	0	0	11	0	0
3:	0	0	0	0	0
4:	0	0	0	0	0
5:	0	6	0	0	17
6:	0	0	0	7	0
7:	0	0	0	0	0
8:	14	0	0	0	0
9:	0	0	0	0	0
10:	0	0	0	0	0
11:	11	0	0	0	0
12:	0	0	0	0	0
13:	15	0	0	0	0
14:	0	6	0	14	1
15:	0	0	0	0	0
sum:	40	12	21	21	18

Cluster:	5	6	7	8	9
Class					
1:	0	0	0	8	0
2:	0	0	0	5	0
3:	0	0	23	0	0
4:	0	12	0	0	3
5:	1	0	0	0	0
6:	0	7	0	0	0
7:	0	0	0	0	20
8:	0	0	0	0	0
9:	0	0	0	0	0
10:	7	0	0	0	15
11:	0	0	0	0	0
12:	0	0	0	0	24
13:	0	0	0	0	0
14:	0	0	0	0	3
15:	12	0	0	0	4
sum:	20	19	23	13	69

Cluster:	10	11	12	13	14
Class					
1:	6	0	0	0	0
2:	6	0	0	2	0
3:	0	0	0	1	0
4:	0	0	8	1	0
5:	0	0	0	0	0
6:	0	0	9	1	0
7:	0	0	0	4	0
8:	0	0	0	1	9
9:	0	0	0	24	0
10:	0	0	0	2	0
11:	0	9	0	0	4
12:	0	0	0	0	0
13:	0	7	0	0	2
14:	0	0	0	0	0

```
     15: 0            0            0            0            8
    sum: 12           16           17           36           23

Cluster: sum
Class

      1: 24
      2: 24
      3: 24
      4: 24
      5: 24
      6: 24
      7: 24
      8: 24
      9: 24
     10: 24
     11: 24
     12: 24
     13: 24
     14: 24
     15: 24
    sum: 360

            Rand index: 0.915877
               Purity: 0.552778
            Precision: 0.385553
               Recall: 0.527295


Partition table test:

Cluster: 0            1            2            3            4
Class

      1: 0            0            2            0            0
      2: 0            0            1            0            0
      3: 0            0            0            0            0
      4: 0            0            0            0            0
      5: 0            1            0            0            2
      6: 0            0            0            0            0
      7: 0            0            0            0            0
      8: 1            0            0            0            0
      9: 0            0            0            0            0
     10: 0            0            0            0            0
     11: 1            0            0            0            0
     12: 0            0            0            0            0
     13: 0            0            0            0            0
     14: 0            0            0            0            0
     15: 0            0            0            0            0
    sum: 2            1            3            0            2

Cluster: 5            6            7            8            9
Class

      1: 0            0            0            1            0
      2: 0            0            0            2            0
      3: 0            0            3            0            0
      4: 0            1            0            0            0
```

```
      5: 0            0            0            0            0
      6: 0            2            0            0            0
      7: 0            0            0            0            3
      8: 0            0            0            0            0
      9: 0            0            0            0            0
     10: 0            0            0            0            3
     11: 0            0            0            0            0
     12: 0            0            0            0            3
     13: 0            0            0            0            0
     14: 0            0            0            0            3
     15: 0            0            0            0            3
    sum: 0            3            3            3            15

Cluster: 10           11           12           13           14
Class

      1: 0            0            0            0            0
      2: 0            0            0            0            0
      3: 0            0            0            0            0
      4: 0            0            2            0            0
      5: 0            0            0            0            0
      6: 0            0            1            0            0
      7: 0            0            0            0            0
      8: 0            0            0            0            2
      9: 0            0            0            3            0
     10: 0            0            0            0            0
     11: 0            2            0            0            0
     12: 0            0            0            0            0
     13: 0            3            0            0            0
     14: 0            0            0            0            0
     15: 0            0            0            0            0
    sum: 0            5            3            3            2

Cluster: sum
Class

      1: 3
      2: 3
      3: 3
      4: 3
      5: 3
      6: 3
      7: 3
      8: 3
      9: 3
     10: 3
     11: 3
     12: 3
     13: 3
     14: 3
     15: 3
    sum: 45

    Test data Rand index: 0.879798
        Test data Purity: 0.577778
     Test data Precision: 0.227941
        Test data Recall: 0.688889
```

With the option '`-T, --partitionstable-outfile`', you get the confusion matrix and with the '`-t`' option for training and testing, as well as measures related to the previous classification of the objects, called supervised measures.

The centroids, besides serving to represent the centers of the clusters, also have a meaning that depends on the domain of the problem, in the libras data set, represent the mean movement of the hand in a two-dimensional curve made in a period of time. The centroids obtained in the execution are shown in Figure 3.7.

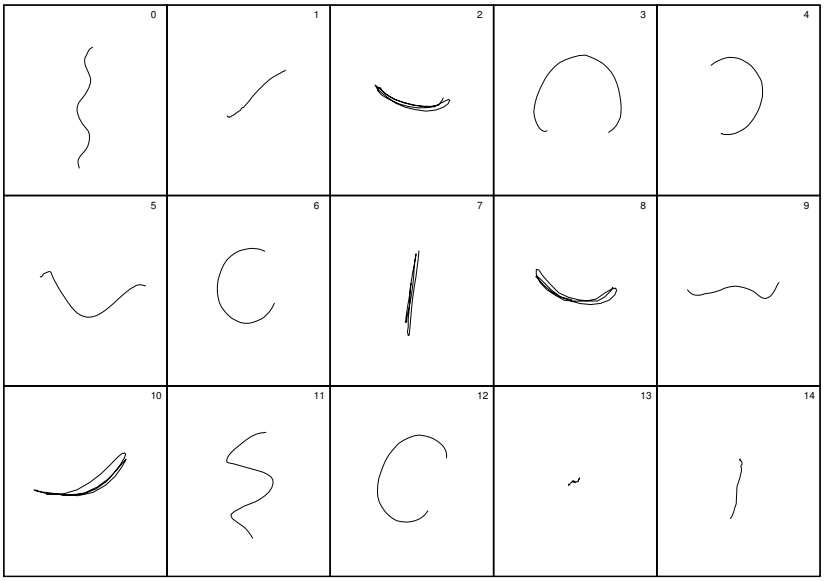

Figure 3.7: Average of the movements of the libras data set obtained with the program `kga_fkcentroid`

3.3.1.2 Based on cluster label

The GAs in this section use an [string-of-group-numbers encoding], page 12. The algorithms included in LEAC are the following:

1. **gaclustering_fklabel**

   It based is on [MC96], it optimizes SED measure (see [SED], page 18) the operators that use are `gagenericop::onePointCrossover` and `gaintegerop::mutation` (change a random gene). The classification of these operators can be considered *object-oriented*, that randomly moves objects among clusters. They propose a small population size, which makes a guided exploration around the best solution, with a fixed crossing probability and a mutation probability with a default value high 0.5 and decreasing in each generation until reaching $1/n$.

   Parameters of the algorithm:

   `--crossover-probability[=NUMBER]`
   real number in the interval `[0.25, 1]` `[NUMBER=0.8]`

`--mutation-probability[=NUMBER]`
real number in the interval `[0, 0.5] [NUMBER=0.5]`

Execution samples:

- 'gaclustering_fklabel -i iris.data -a "1-4"'
- 'gaclustering_fklabel -i iris.data -a "1-4" -c 5 --number-clusters 3 --generations 10000 --population-size 6 --crossover-probability 0.8 --mutation-probability 0.5 -C stdout -T stdout --table-format yes -M stdout'

2. **gka_fklabel**
   It based is on [KM99], it optimizes TWCV measure (see [TWCV], page 18). The authors of this document propose not to use the crossover operator, instead they use the `k-means algorithm`. [HCFdC09] state that this algorithm must be an evolutionary algorithm rather than a genetic algorithm. LEAC implements several variants of the `k-means algorithm`, including the algorithm used by this algorithm ([clusteringop::kmeansoperator], page 34). Mutation change an gene value depending on the distances of the cluster centroids from the corresponding instance. It can be assigned to another group by the following probability distribution associated with each gene.
   Parameters of the algorithm:

   `--mutation-probability[=NUMBER]`
   real number in the interval `[0, 1.0] [NUMBER=0.05]`

   Execution sample:

   'gka_fklabel -i iris.data -a "1-4" -c 5 --number-clusters 3 --generations 200 --population-size 50 --mutation-probability 0.05 -C stdout -T stdout --table-format yes -M stdout'
3. **igka_fklabel** it based is on [LLF+04b], and **fgka_fklabel** it based is on [LLF+04a]. These two proposals are inspired by [KM99]. These optimize TWCV measure (see [TWCV], page 18). Define a legality ratio $e(Ch_i)$ for each Ch_i chromosome, associate with the number of non-empty clusters in the solution obtained by Ch_i divided by number of clusters k. $e(Ch_i)$ is legal if $e(Ch_i) = 1$, and illegal otherwise. The legality ratio together with the value of TWCV defines the fitness function, that will guide the evolution of the algorithm.
   Parameters of the algorithm:

   `--mutation-probability[=NUMBER]`
   real number in the interval `[0, 1.0] [NUMBER=0.05]`

   Execution sample:

   'igka_fklabel -i iris.data -a "1-4" -c 5 --number-clusters 3 --generations 200 --population-size 50 --mutation-probability 0.05 -C stdout -T stdout --table-format yes -M stdout'

3.3.1.3 Based on the most representative

K-medoid algorithms attempt to partition the data by assigning each object to a representative and then optimizing a unsupervised measures (Section 3.1.3.2 [Unsupervised measures], page 18).

1. **gaprototypes_fkmedoid**

   It based is on [KB97], it optimizes J_1 measure (see [J_1], page 21). Use a binary encoding, see [Binary encoding for medoid-based], page 14.

   Parameters of the algorithm:

   `--crossover-probability[=NUMBER]`
   : real number in the interval `[0.25, 1]` `[NUMBER=0.5]`

   `--mutation-probability[=NUMBER]`
   : real number in the interval `[0, 0.5]` `[NUMBER=0.015]`

   `--probability-ini[=NUMBER]`
   : real number in the interval `[0.01, 0.1]` depending n and k, if -1 is eq k/n `[NUMBER=-1]`

   `--alpha[=NUMBER]`
   : real number `alpha > 0` to force the algorithm desired number cluster `[NUMBER=10]`

   The parameter `--probability-ini` serves for the initialization of the chromosomes, each gene in a chromosome has the value 1 with a prespecified probability `--probability-ini`. The genetic operators that it uses are `gabinaryop::uniformCrossover` the parent chromosomes swap their i-th genes with a certain probability (`--crossover-probability`) and `gabinaryop::eachBitArrayMutation` each gene de each offspring chromosome alternates with a predefined probability (`--mutation-probability`). To avoid generating invalid offspring for fixed k, the parameters must be adjusted together with `--alpha`.

   Execution sample:

   ‘`gaprototypes_fkmedoid -i iris.data -a "1-4" -c 5 --number-clusters 3 --generations 500 --population-size 20 --crossover-probability 0.5 --mutation-probability 0.015 --probability-ini 0.02 --alpha 100 -C stdout -T stdout --table-format yes -M stdout`’

2. **gca_fkmedoid**

   It based is on [LDK93], optimize to SEDmedoid (see [SEDmedoid], page 20). Codes individuals as a string of integers of length k, each integer number of the string corresponds to the index of the most representative instance of the cluster. This algorithm uses two new genetic operators *mix subset recombination* D_MX and *point mutation*, D_PM, D in the names serves to remind of the fact that direct encoding rather than binary. In the LEAC library they are coded with the `gaintegerop::recombinationD_MX` and `gaintegerop::mutationD_PM` functions.

   Parameters of the algorithm:

   `--mix-recombination-probability[=NUMBER]`
   : real number in the interval `[0.5, 0.9]` `[NUMBER=0.9]`

   `--point-mutation-probability[=NUMBER]`
   : real number in the interval `[0.2, 0.4]` `[NUMBER=0.4]`

   `--mix-mutation-probability[=NUMBER]`
   : real number in the interval `[0, 0.125]` `[NUMBER=0.125]`

The first two parameters establish the frequency of application of genetic operators D_MX and D_PM. The third parameter adds new material in the application of D_MX.

Execution sample:

‘`gca_fkmedoid -i iris.data -a "1-4" -c 5 --number-clusters 3 --generations 200 --population-size 200 --mix-recombination-probability 0.9 --point-mutation-probability 0.4 --mix-mutation-probability 0.125 -C stdout -T stdout --table-format yes -M stdout`’

3. **hka_fkmedoid**

   It based is on [SL04], it optimizes SEDmedoid measure (see [SEDmedoid], page 20). This algorithm is inspired [LDK93], to accelerate convergence, it contains one step of a `local heuristic search` to accelerate convergence, he proposes a heuristic search operator equivalent to `k-means algorithm`, but for medoids. The local heuristic search is implemented in the `clusteringop::updateMedoids` function. See [local heuristic search], page 34.

   Parameters of the algorithm:

   `--mix-recombination-probability[=NUMBER]`
   real number in the interval `[0.5, 0.9]` `[NUMBER=0.95]`

   `--point-mutation-probability[=NUMBER]`
   real number in the interval `[0.2, 0.4]` `[NUMBER=0.02]`

   `--mix-mutation-probability[=NUMBER]`
   real number in the interval `[0, 0.125]` `[NUMBER=0.05]`

   `--order-tournament[=NUMBER]`
   order of tournament `[NUMBER=2]`

   `--nearest-neighbors[=NUMBER]`
   number of the nearest neighbors (p) `[NUMBER=3]`

   `--search-heuristic-probability[=NUMBER]`
   real number in the interval `[0.0, 1.0]` `[NUMBER=0.2]`

Execution sample: see [Illustrative execution samples: Iris Data Set], page 57.

Illustrative execution samples: Iris Data Set

As an illustrative example of algorithms that finds the most representative instances for each cluster, we describe the `hka_fkmedoid` program based on the HKA algorithm [SL04] Before executing the commands, make sure you have access to the executable programs and the data, see [configuration of the PATH variable], page 42.

Now let’s illustrate how to find the most representative instances of the iris.data data set. You can run the program with the following parameters:

‘`hka_fkmedoid -i iris.data -a "1-4" -c 5 --number-clusters=3 -C c_hka_iris.dat -M m_hka_iris.dat`’

A possible exit from the program would be:

```
IN:
    Algorithmo name: HKA
           Based on: Weiguo Sheng and Xiaohui Liu
```

```
        Metric used: SED

           Data set: iris.data
Number of instances: 150
         Dimensions: 4

       Random seed: 4253005715 70818531 1631842517 1223368670 2252683652
       1178029056 3404574059 1048346743

OUT:

CROMOSOME:BEST:objetive,98.2137,fitness,0.0101819:7,78,112

       Cluster number (K): 3
                      SED: 98.2137
                DB-index: 0.811606
              Silhouette: 0.552592
                      VRC: 494.895
              CS measure: 0.155418
            Dunn's index: 0.0988074
     Execution time (seg): 0.072937
Generations find the best: 69
```

For this execution, the most representative instances for each Iris cluster are:

ID	SepalLength	SepalWidth	PetalLength	PetalWidth	Class
7	5	3.4	1.5	0.2	Iris-setosa
78	6	2.9	4.5	1.5	Iris-versicolor
112	6.8	3	5.5	2.1	Iris-virginica

To visualize the results:

‘`plot_clustering -i iris.data -a "1-4" -c 5 --centroids-infile c_hka_iris.dat --member-infile m_hka_iris.dat --centroids-title "Medoid" --graphics-outfile hka_iris`’

And graphically show the prototypes and groups in Figure 3.8.

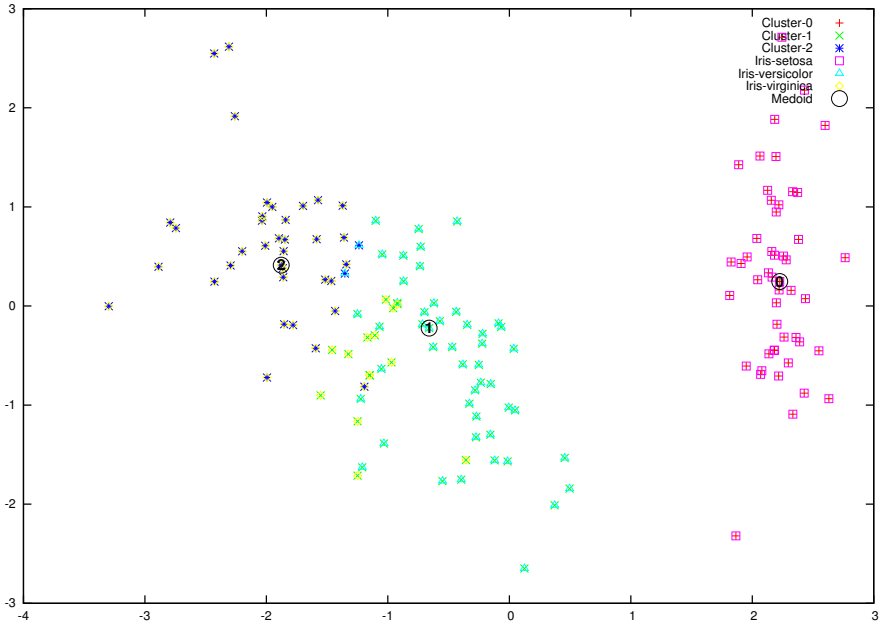

Figure 3.8: The most representative instances of the Iris data set, obtained with the program '`hka_fkmedoid`'

3.3.2 Genetic algorithms for variable k-clusters

The GAs described in the following sections obtain both the best cluster number (k^*) and cluster formation.

The parameters that are common in these algorithms are:

`--k-minimum[=NUMBER]`
: number of clusters by default `[NUMBER=2]`

`--k-maximum[=NUMBER]`
: Some authors leave it to the user's criteria, others propose a predetermined value based on the size of the data set, such as $n/2$ or $n^{1/2}$, most agree with the latter.

3.3.2.1 Based on the centroids

The GAs in this section use a real encoding (see [real encoding], page 13) with some variant to allow the search of the optimum k. The algorithms included in LEAC are the following:

1. **gcuk_vkcentroid**

   It based is on [BM02b], it optimizes SSE measure [DB], page 22. Encoding the chromosomes as strings of real numbers that represent the coordinates of the centroids, combined with `#` symbols, allow the diversity of chromosomes with different k, through the application of genetic operators `gaclusteringop::onePointCrossover` and `gaclusteringop::randomMutation`.

   Parameters of the algorithm:

`--crossover-probability[=NUMBER]`
real number in the interval `[0.25, 1]` `[NUMBER=0.8]`

`--mutation-probability[=NUMBER]`
real number in the interval `[0, 0.5]` `[NUMBER=0.001]`

Execution sample: see [Illustrative execution samples: Zoo Data Set], page 60.

2. **tgca_vkcentroid**

It based is on [HT12], it optimizes VRC measure ([VRC], page 25). It is a novel genetic algorithm, called by its authors as *two-stage genetic clustering algorithm*. First, TGCA focuses on the search of the best number of clusters, and then gradually transfers towards finding the globally optimal cluster centers. Furthermore, a maximum attribute range partition approach is used in the population initialization so as to overcome the sensitivity of clustering algorithms to initial partitions [HT12].

Parameters of the algorithm:

`--num-subpopulations-cross[=NUMBER]`
number of subpopulations for parallel crossover. It must be less than half the size compared to the population [NUMBER=4]

`--crossover-probability[=NUMBER]`
real number in the interval [0.25, 1] [NUMBER=0.8]

`--kmeans-iterations[=NUMBER]`
maximum number of iterations for k-means algorithm [NUMBER=100]

`--kmeans-threshold[=NUMBER]`
threshold value for k-means algorithm [NUMBER=0]

Execution sample:

‘`tgca_vkcentroid -i iris.data -a "1-4" -c 5 --population-size 200 --num-subpopulations-cross 4 --crossover-probability 0.8 --generations 200 --kmeans-iterations 100 --kmeans-threshold 0 -C stdout -T stdout --table-format yes -M stdout`’

Illustrative execution samples: Zoo Data Set

Next we show with the `gcuk_vkcentroid` program, how can you find an automatic classification of the Zoo data set. The data set zoo.data has 7 classes with 17 attributes. All attributes are binary, with the exception of number 14, which can be seen as nominal. Before executing the commands, make sure you have access to the executable programs and the data, see [configuration of the PATH variable], page 42.

To process it can be transformed into binary with the following `awk` script:

```
#run:
# awk -f zoo_binary.awk -F ',' -v OFS=','  zoo.data > zoo_bin.csv
BEGIN {
    h1 = "animalname";
    h2 = "hair";
    h3 = "feathers";
    h4 = "eggs";
    h5 = "milk";
    h6 = "airborne";
```

```
    h7 = "aquatic";
    h8 = "predator";
    h9 = "toothed";
    h10 = "backbone";
    h11 = "breathes";
    h12 = "venomous";
    h13 = "fins";
    h14 = "legs_0,legs_2,legs_4,legs_5,legs_6,legs_8";
    h15 = "tail";
    h16 = "domestic";
    h17 = "catsize";
    h18 = "type";
    print h1,h2,h3,h4,h5,h6,h7,h8,h9,h10,h11,h12,h13,h14,h15,h16,h17,h18;
}
{
# legs:Numeric (set of values: 0,2,4,5,6,8)
    if ( $14 == 0)
        $14 = "1,0,0,0,0,0";
    else if ($14 == 2)
        $14 = "0,1,0,0,0,0";
    else if ($14 == 4)
        $14 = "0,0,1,0,0,0";
    else if ($14 == 5)
        $14 = "0,0,0,1,0,0";
    else if ($14 == 6)
        $14 = "0,0,0,0,1,0";
    else if ( $14 == 8)
        $14 = "0,0,0,0,0,1";
    print $1,$2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12,$13,$14,$15,$16,$17,$18
}
```

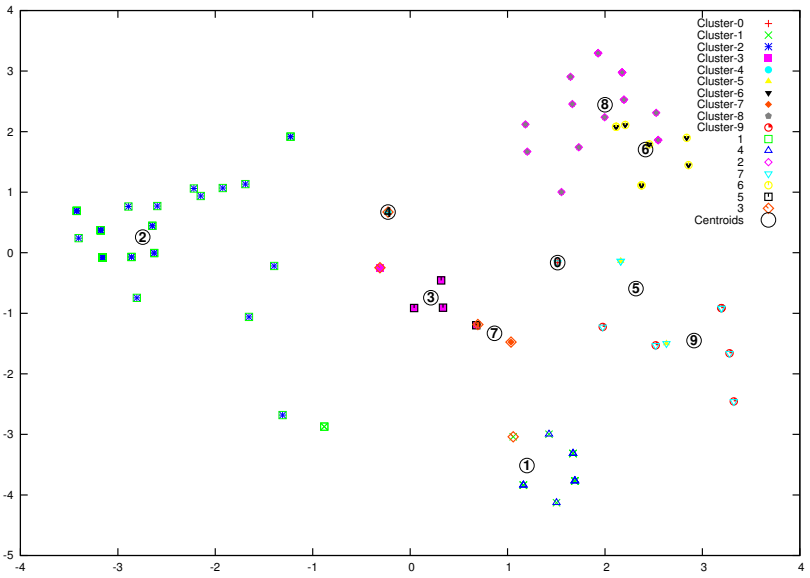

Figure 3.9: Clusters obtained in the data set Zoo with `gcuk_vkcentroid`

'`awk -f zoo_binary.awk -F ',' -v OFS=',' zoo.data > zoo_bin.csv`'

```
  ‘gcuk_vkcentroid -i zoo_bin.csv -h yes -a "2-22" -c 23 --k-minimum=2
--k-maximum=20 -C c_zoo_gcuk.data -M m_zoo_gcuk.data -T stdout --table-format
yes’
      IN:
          Algorithmo name: GCUK
                 Based on: Bandyopadhyay and Maulik 2002
              Metric used: DB-index

                 Data set: zoo_bin.csv
      Number of instances: 101
               Dimensions: 21

             Random seed: 728997733 111590912 682175903 3140775393 958797699
             412503851 3032481805 703125621

      OUT:

      CROMOSOME:BEST:rows,9,columns,21>0,1,1,0,0.8,0.3,0.45,0,1,1,0,0,0,1,0,0,0,0,
      1,0.15,0.3;0,0,0,0,0,0,1,0,0,1,1,0,0,0,0,0,0,1,1,0,0;0.0588235,0,0.764706,
      0.176471,0,1,0.764706,1,1,0.176471,0.117647,0.941176,1,0,0,0,0,0,0.941176,
      0.0588235,0.411765;1,0,0.0263158,1,0.0526316,0.0789474,0.5,0.973684,1,1,0,
      0.0263158,0,0.184211,0.815789,0,0,0,0.868421,0.210526,0.763158;0,0,1,0,0,0.8,
      0.8,1,1,1,0.2,0,0,0,1,0,0,0,0.4,0,0;0.4,0,1,0,0.6,0,0.1,0,0,1,0.2,0,0.2,0,0,
      0,0.8,0,0,0.1,0;0,0,1,0,0,0,0,0,1,1,0,0,0,0,1,0,0,0,1,0,1;0,0,1,0,0,0,1,1,1,
      1,0.5,0,1,0,0,0,0,0,1,0,0;0,0,1,0,0,0.857143,1,0,0,0,0.142857,0,0.285714,0,
      0.142857,0.142857,0.285714,0.142857,0,0,0.142857

             Cluster number (K): 9
                       DB-index: 0.855135
                            SED: 98.3379
                     Silhouette: 0.355542
                            VRC: 26.5558
                     CS measure: 1.72969
                   Dunn’s index: 0.57735
           Execution time (seg): 0.354636
      Generations find the best: 53


      Partition table:

      Cluster: 0            1            2            3            4
      Class

            1: 0            0            3            38           0
            4: 0            0            13           0            0
            2: 20           0            0            0            0
            7: 0            1            0            0            0
            6: 0            0            0            0            0
            5: 0            0            0            0            4
            3: 0            0            1            0            1
          sum: 20           1            17           38           5

      Cluster: 5            6            7            8            sum
      Class

            1: 0            0            0            0            41
            4: 0            0            0            0            13
            2: 0            0            0            0            20
```

```
 7: 2          0          0          7          10
 6: 8          0          0          0          8
 5: 0          0          0          0          4
 3: 0          1          2          0          5
sum: 10        1          2          7          101

          Rand index: 0.956238
              Purity: 0.930693
           Precision: 0.932188
              Recall: 0.875956
```

$$C = \begin{bmatrix}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\text{hair} & 0.00 & 0.00 & 0.06 & 1.00 & 0.00 & 0.40 & 0.00 & 0.00 & 0.00 \\
\text{feathers} & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
\text{eggs} & 1.00 & 0.00 & 0.76 & 0.03 & 1.00 & 1.00 & 1.00 & 1.00 & 1.00 \\
\text{milk} & 0.00 & 0.00 & 0.18 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
\text{airborne} & 0.80 & 0.00 & 0.00 & 0.05 & 0.00 & 0.60 & 0.00 & 0.00 & 0.00 \\
\text{aquatic} & 0.30 & 0.00 & 1.00 & 0.08 & 0.80 & 0.00 & 0.00 & 0.00 & 0.86 \\
\text{predator} & 0.45 & 1.00 & 0.76 & 0.50 & 0.80 & 0.10 & 0.00 & 1.00 & 1.00 \\
\text{toothed} & 0.00 & 0.00 & 1.00 & 0.97 & 1.00 & 0.00 & 0.00 & 1.00 & 0.00 \\
\text{backbone} & 1.00 & 0.00 & 1.00 & 1.00 & 1.00 & 0.00 & 1.00 & 1.00 & 0.00 \\
\text{breathes} & 1.00 & 1.00 & 0.18 & 1.00 & 1.00 & 1.00 & 1.00 & 1.00 & 0.00 \\
\text{venomous} & 0.00 & 1.00 & 0.12 & 0.00 & 0.20 & 0.20 & 0.00 & 0.50 & 0.14 \\
\text{fins} & 0.00 & 0.00 & 0.94 & 0.03 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
\text{legs 0} & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.20 & 0.00 & 1.00 & 0.29 \\
\text{legs 2} & 1.00 & 0.00 & 0.00 & 0.18 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
\text{legs 4} & 0.00 & 0.00 & 0.00 & 0.82 & 1.00 & 0.00 & 1.00 & 0.00 & 0.14 \\
\text{legs 5} & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.14 \\
\text{legs 6} & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.80 & 0.00 & 0.00 & 0.29 \\
\text{legs 8} & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.14 \\
\text{tail} & 1.00 & 1.00 & 0.94 & 0.87 & 0.40 & 0.00 & 1.00 & 1.00 & 0.00 \\
\text{domestic} & 0.15 & 0.00 & 0.06 & 0.21 & 0.00 & 0.10 & 0.00 & 0.00 & 0.00 \\
\text{catsize} & 0.30 & 0.00 & 0.41 & 0.76 & 0.00 & 0.00 & 1.00 & 0.00 & 0.14
\end{bmatrix}
\quad
W = \begin{bmatrix} 0.20 \\ 0.01 \\ 0.17 \\ 0.38 \\ 0.05 \\ 0.10 \\ 0.01 \\ 0.02 \\ 0.07 \end{bmatrix}$$

Figure 3.10: Zoo Clusters with `gcuk_vkcentroid`

From the run of the program, 9 clusters were obtained, using the *DB-index* similarity measure, with very good measured values obtained. For this data set it is possible to use the centroids to obtain an association between items in the same row ([AIS93], [HPY00]) as shown in the Figure 3.10 and the summary in the Table 3.1.

'`plot_clustering -i zoo_bin.csv -h yes -a "2-22" -c 23 --centroids-infile c_zoo_gcuk.data --member-infile m_zoo_gcuk.data --graphics-outfile zoo_gcuk`'

C_j	W_j	**Frequent dimension**	**Outliers**
0	20%	95-100%: feathers eggs backbone breathes legs 2 tail 80-90%: airborne	
1	1%	90-100%: predator breathes venomous legs 8 tail	
2	17%	90-100%: aquatic backbone fins legs 0 tail	{eggs hair domestic}
3	38%	90-100%: hair milk toothed backbone breathes legs 4	{eggs airborne fins legs 4}
4	5%	90-100%: eggs toothed backbone breathes legs 4 80-90%: aquatic predator	
5	10%	90-100%: eggs breathes 80-90%: legs 6	{legs 6 domestic}
6	1%	90-100%: eggs backbone breathes legs 4 tail catsize	
7	2%	90-100%: eggs predator toothed backbone breathes legs 0 tail	
8	7%	90-100%: eggs predator 80-90%: aquatic	{aquatic venomous legs 4 legs 5 legs 8 catsize }

Table 3.1: Zoo clusters summary table

3.3.2.2 Based on cluster label

The GAs in this section use a string of group numbers encoding (see [string-of-group-numbers encoding], page 12) with some variant to allow the search for the optimum k. The algorithms included in LEAC are the following:

1. **gga_vklabeldbindex** and **gga_vklabelsilhouette**

   It based is on [ABSSJF+12], it optimizes SSE measure [DB], page 22, and [Silhouette], page 22. Chromosome is formed by two section [*element* | *group*]. The element section each position (gene) corresponds to the belongings of the object to a cluster. The group section corresponds to the alphabet of possible values of the genes $\{1, 2, 3, ..., k\}$. An island model is used, where operators `gaclusteringop::mergeCrossover`, `gaclusteringop::mergeMutation` and `gaclusteringop::splittingMutation` are applied.

   Parameters of the algorithm:

   `--sub-population-size[=NUMBER]`
   : size of sub-populations (islands) `[NUMBER=20]`

   `--number-island[=NUMBER]`
   : number of sub-populations or islands `[NUMBER=4]`

   `--pe[=NUMBER]`
   : probability of migration good individuals between islands `[0,1]` `[NUMBER=0.5]`

   `--pci[=NUMBER]`
   : initial probability crossover, real number in the interval `[0,1]` must be high in the first stages `[NUMBER=0.8]`

`--pcf[=NUMBER]`
: final probability crossover, real number in the interval `[0,1]` must moderate in the last stages `[NUMBER=0.4]`

`--pci[=NUMBER]`
: initial probability mutation, real number in the interval `[0,1]` is smaller in the first generations `[NUMBER=0.05]`

`--pcf[=NUMBER]`
: final probability mutation, real number in the interval `[0,1]` is larger in the last ones `[NUMBER=0.2]`

`--pbi[=NUMBER]`
: initial probability local search, real number in the interval `[0,1]` must be high in the first stages `[NUMBER=0.1]`

`--pbf[=NUMBER]`
: final probability local search, real number in the interval `[0,1]` must moderate in the last stages `[NUMBER=0.05]`

Execution sample: see [Illustrative execution samples: Ionosphere Data Set], page 67.

2. **cga_vklabel**

   It based is on [HE03], it optimizes Silhouette measure (see [Silhouette], page 22). Chromosome is represented as a string of label of $(n + 1)$ positions. Each position corresponds to an instance, i.e., the i-th position (gene) represents the $(n + 1)$ object, whereas the last gene represents the number of clusters (k). Crossover operator combines clustering solutions coming from different chromosome (`gaclusteringop::crossoverCGA`). Two operators for mutation are used in MO_1 (`gaclusteringop::MO1`) and MO_2 (`gaclusteringop::MO1`). MO_1 works only on genotypes that encode more than two clusters. It eliminates a randomly chosen cluster, placing its objects to the nearest remaining clusters (according to their centroids). MO_2 divides a randomly selected cluster into two new ones. The first cluster is formed by the objects closer to the original centroid, whereas the other cluster is formed by those objects closer to the farthest object from the centroid.

   Parameters of the algorithm:

   `--crossover-probability[=NUMBER]`
   : real number in the interval `[0.25, 1]` `[NUMBER=0.5]`

   `--mutation-probability[=NUMBER]`
   : for operator 1 and 2 real number in the interval `[0, 0.5]` `[NUMBER=0.25]`

   Execution sample:

   ‘`cga_vklabel -i iris.data -a "1-4" -c 5 --generations 200 --population-size 200 --crossover-probability 0.5 --mutation-probability 0.25 -C stdout -T stdout --table-format yes -M stdout`’

3. **eac_vklabel**

   It based is on [HCdC06], it optimizes Simplified silhouette (see [Simplified silhouette], page 23). Chromosome is represented as a string of label of n, Each position corresponds to an instance, i.e., the i-th position (gene) represents the n object, whereas the last

gene represents the number of clusters k. Mutation two operators for mutation MO_1 and MO_2 similar to [HE03]. Also `k-means` operator is applied.

Parameters of the algorithm:

`--desiable-objfunc[=NUMBER]`
: value desiable of objetive function (eg. silhouette `[-1,1]` rand index[0,1]) `[NUMBER=1]`

`--kmeans-iterations[=NUMBER]`
: maximum number of iterations for k-means algorithm `[NUMBER=5]`

`--kmeans-difference[=NUMBER]`
: maximum absolute difference between centroids in two consecutive iterations is less than or equal to `[NUMBER=0.001]`

Execution sample:

‘`cga_vklabel -i iris.data -a "1-4" -c 5 --generations 200 --population-size 200 --crossover-probability 0.5 --mutation-probability 0.25 -C stdout -T stdout --table-format yes -M stdout`’

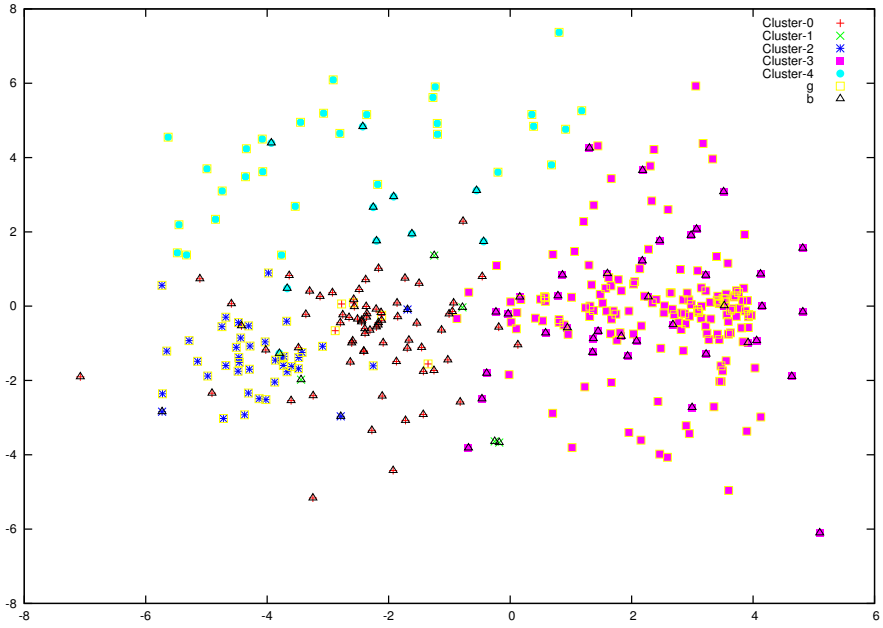

Figure 3.11: Clusters obtained with `gga_vklabelsilhouette`

4. **feac_vklabel**, and its variants **eaci_vklabel**, **eacii_vklabel** and **eaciii_vklabel**

   It based is on [ACH06], it optimizes Simplified silhouette [Simplified silhouette], page 23. Chromosome is represented as a string of label of n, Each position corresponds to an instance, i.e., the i-th position (gene) represents the n object, whereas the last gene represents the number of clusters k. For implementation purposes, the chromosome includes the centroids associated with the string of group numbers encoding `gaencode::ChromosomeFEAC`. Operator of crossover does not

use. Mutation two operators for mutation MO_1 (`gaclusteringop::MO1`) and MO_2 (`gaclusteringop::MO2`) similar to [HCdC06]. In addition, a `k-means` operator is applied (`gaclusteringop::kmeansfeac`). The only difference between `eaci_vklabel` and FEAC `feac_vklabel` is rate of each mutation operator according to its performance during the *evolutionary search*.

Parameters of the algorithm:

`--desiable-objfunc[=NUMBER]`
: value desiable of objetive function (eg. silhouette `[-1,1]` rand index `[0,1]`) `[NUMBER=1]`

`--kmeans-iterations[=NUMBER]`
: maximum number of iterations for k-means algorithm `[NUMBER=5]`

`--kmeans-difference[=NUMBER]`
: maximum absolute difference between centroids in two consecutive iterations is less than or equal to `[NUMBER=0.001]`

Execution sample:

'`feac_vklabel -i iris.data -a "1-4" -c 5 --generations 500 --population-size 20 --kmeans-iterations 5 --kmeans-difference 0.001 -C stdout -T stdout --table-format yes -M stdout`'

Illustrative execution samples: Ionosphere Data Set

As an illustrative example, the Ionosphere, data set is processed with the program `gga_vklabelsilhouette`, this data set has the complexity that the instances of different classes overlap. For this case, the See [silhouette], page 22, metric is appropriate. Before executing the commands, make sure you have access to the executable programs and the data, see [configuration of the PATH variable], page 42.

'`gga_vklabelsilhouette -i ionosphere.data -a "1-34" -c 35 -M m_gga_ionosphere.data -T stdout --table-format yes`'

```
IN:
    Algorithmo name: GGA_SILHOUETTE
           Based on: Agustin-Blas L.E. and Salcedo-Sanz S. and
           Jimenez-Fernandez S. and Carro-Calvo L. and Del Ser J.
           and Portilla-Figueras, J.A.
        Metric used: Silhouette

           Data set: ionosphere.data
Number of instances: 351
         Dimensions: 34

       Random seed: 389887919 3739475900 1204968903 2830340246 1365531614
       2641306820 2495066055 1120010892

OUT:

CROMOSOME:BEST:objetive,0.320094,fitness,0.3200943,0,3,1,3,0,3,1,3,0,3,3,3,
3,3,0,3,0,3,4,3,0,3,0,3,3,3,0,3,1,3,3,3,3,3,3,3,4,3,3,4,2,4,3,3,4,3,0,3,0,3,
0,2,0,3,0,3,0,3,0,3,0,3,2,3,3,3,3,2,3,3,1,4,0,3,3,3,4,4,4,3,3,4,4,3,3,3,3,3,
4,3,0,3,0,3,4,4,4,0,3,3,3,0,3,3,3,0,3,0,3,3,4,3,4,0,3,3,3,0,3,3,3,0,3,0,3,3,
3,0,3,3,3,3,4,3,4,2,3,0,3,0,3,3,0,3,0,0,3,3,3,0,3,0,2,0,3,0,2,0,2,0,3,0,2,3,
```

```
2,0,2,0,3,0,3,0,3,0,3,0,3,0,3,0,3,3,2,0,2,1,3,0,0,0,3,0,3,3,3,0,3,3,3,0,
3,0,3,0,3,0,3,0,3,0,3,3,2,3,4,4,4,0,4,3,4,0,4,0,2,3,3,0,3,0,3,3,3,4,3,0,3,0,
3,0,3,3,3,0,3,0,2,0,2,0,2,2,2,3,3,3,3,3,4,4,4,4,4,4,3,3,3,3,3,3,3,3,3,3,3,4,
4,4,4,4,2,0,3,3,3,3,3,3,3,3,3,3,3,3,4,2,2,2,2,2,2,3,3,3,3,3,3,4,4,2,2,2,3,3,2,2,2,
3,3,3,3,3,3,3,3,2,2,2,2,2,3,3,3,3,3,3,3,3,2,2,2,2,2,0,3,3,3,3,3,3,3,3,3,3|0,1,2,3,4

       Cluster number (K): 5
               Silhouette: 0.320094
                      SED: 705.646
                DB-index: 1.93699
                      VRC: 54.7797
               CS measure: 1.9306
             Dunn's index: 0.0709856
     Execution time (seg): 3.59483
Generations find the best: 77


Partition table:

Cluster: 0            1            2            3            4
Class

      g: 5            0            38           152          30
      b: 70           5            3            38           10
    sum: 75           5            41           190          40

Cluster: sum
Class

      g: 225
      b: 126
    sum: 351

                Rand index: 0.612291
                    Purity: 0.840456
                 Precision: 0.707252
                    Recall: 0.477702
```

'`plot_clustering -i ionosphere.data -a "1-34" -c 35 --member-infile m_gga_ionosphere.data --graphics-outfile gga_ionosphere`'

3.3.2.3 Based on other encoding schemes

1. **gaclustering_vktreebinary**

   It is based is on [CdLM03] it optimizes VRC measure (see [VRC], page 25). Adopt an encoding scheme based on minimum spanning tree (MST), the tree nodes represent the n instances of the data set and the edges $(n-1)$ correspond to the nearest instances. The algorithm first calculates the MST and creates the partitions of the data set by preserving or deleting the edges, represented by a binary string. The value 0 means that the corresponding edge remains, while the value 1 means that it is deleted. The number of elements with value 1 is equal to $(k-1)$, where k is the number of clusters.

   Parameters of the algorithm:

   `--notchangestop[=NUMBER]`
   : after a number x of iteratios, the best chromosome does not change. (We have fixed x = 3) `[NUMBER=3]`.

`--crossover-probability[=NUMBER]`
real number in the interval `[0.25, 1]` `[NUMBER=0.8]`

`--mutation-probability[=NUMBER]`
real number in the interval `[0, 0.5]` `[NUMBER=0.008]`

Execution sample: see [Illustrative execution samples: Ecoli Data Set], page 69.

2. **clustering_vksubclusterbinary**

It based is on [TY01] it optimizes intra-cluster and inter-cluster distance measure (see [Intra-cluster and inter-cluster distance], page 25). It uses a binary coding scheme centroid-based (see [Centroid-based binary encoding], page 13), with traditional genetic operators, `gabinaryop::onePointDistCrossover` and `gabinaryop::bitMutation`. It also uses a *heuristic strategy* to find a "good grouping".

Parameters of the algorithm:

`--u-parameter[=NUMBER]`
parameter u `[NUMBER=1.4]`

`--lambda[=NUMBER]`
parameter lambda `[NUMBER=0.125]`

`--w1[=NUMBER]`
smallest value w1 `[NUMBER=1]`

`--w2[=NUMBER]`
largest value w2 `[NUMBER=3]`

`--crossover-probability[=NUMBER]`
real number in the interval `[0.25, 1]` `[NUMBER=0.8]`

`--mutation-probability[=NUMBER]`
real number in the interval `[0, 0.5]` `[NUMBER=0.05]`

Execution sample:

‘`clustering_vksubclusterbinary -i iris.data -a "1-4" -c 5 --u-parameter 1.4 --lambda 0.125 --w1 1 --w2 3 --generations 100 --population-size 50 --crossover-probability 0.8 --mutation-probability 0.5 -C stdout -T stdout --table-format yes -M stdout`’

Illustrative execution samples: Ecoli Data Set

As an illustrative example, the Ecoli, data set was processed, in order to obtain a cluster number automatically and compare it with the classification proposed in the data set. Before executing the commands, make sure you have access to the executable programs and the data, see [configuration of the PATH variable], page 42.

‘`gaclustering_vktreebinary -i ecoli.data -a "2-8" -c 9 -d " " -C ecoli_centroids.data -M ecoli_membership.data -G ecoli_tree.data -T stdout --table-format yes --generations=500 --notchangestop=100`’

```
IN:
    Algorithmo name: GA_CASILLAS2003
           Based on: Casillas and Gonzalez and Martinez 2003
        Metric used: Variance Ratio Criterion
```

```
           Data set: ecoli.data
Number of instances: 336
         Dimensions: 7

       Random seed: 3034846250 3476018755 2194011041 1038797822 4055928898
       2745797188 3302303888 3051852958

OUT:

CROMOSOME:BEST:objetive,215.739,fitness,215.739,0000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000010000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000010

       Cluster number (K): 3
 Variance Ratio Criterion: 215.739
                      SED: 80.6089
                DB-index: 0.976877
              Silhouette: 0.39575
              CS measure: 0.0297727
            Dunn's index: 0.038597
     Execution time (seg): 12.9314
Generations find the best: 10


Partition table:

Cluster: 0            1            2            sum
Class

     cp: 142          0            1            143
     im: 7            68           2            77
    imS: 0            1            1            2
    imL: 0            2            0            2
    imU: 1            34           0            35
     om: 0            1            19           20
    omL: 0            5            0            5
     pp: 5            1            46           52
    sum: 155          112          69           336

              Rand index: 0.86672
                  Purity: 0.761905
               Precision: 0.687857
                  Recall: 0.927444
```

And the results can be visualized with the following command. Figure 3.12

```
'plot_clustering -i ecoli.data -a "2-8" -c 9 -d " " --centroids-infile
ecoli_centroids.data --member-infile ecoli_membership.data --graph-infile
ecoli_tree.data --graphics-outfile ecoli_tree --centroids-size 1.5
--member-size 0.5 --size-instance 0.6'
```

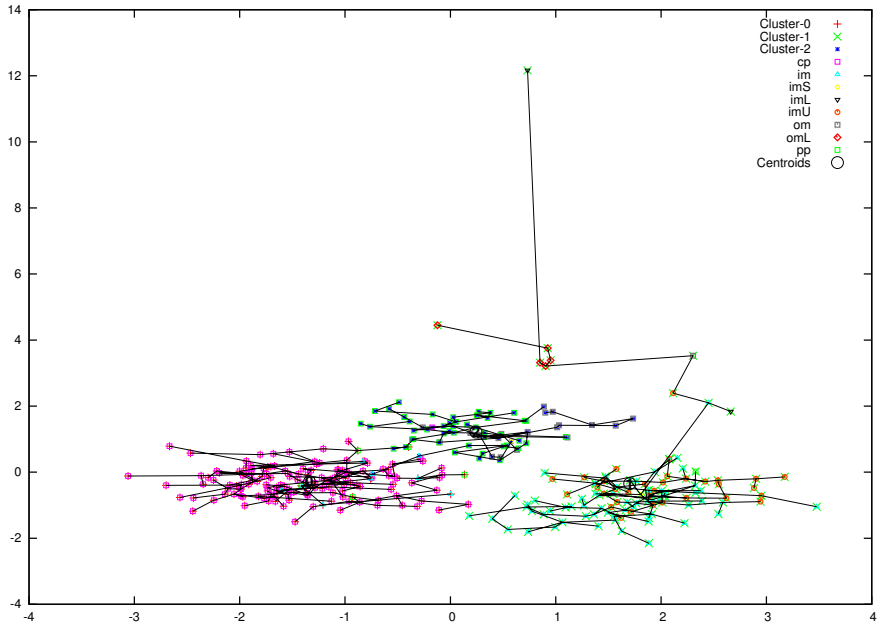

Figure 3.12: Minimum spanning tree (MST) and clusters obtained with `gaclustering_vktreebinary` program for the ecoli data set

4 Reporting Bugs

If you find a bug in LEAC, please send electronic mail to hermes@uaz.edu.mx.

Appendix A Example source code

The following source code show the use of the LEAC library. The files are in `eac` directory.

A.1 KGA algorithm

The following encoded algorithm is the KGA (`kga_fkcentroid.hpp`), described in the paper [BM02a].

```
/*! \file kga_fkcentroid.hpp
 *
 * \brief KGA \cite Bandyopadhyay:Maulik:GAclustering:KGA:2002
 *
 * \details This file is part of the LEAC.\n\n
 * Implementation of the KGA algorithm based on the paper:\n
 * S. Bandyopadhyay and U. Maulik. An evolutionary technique based\n
 * on k-means algorithm for optimal clustering in rn. Inf. Sci. Appl.,\n
 * 146(1-4):221--237, 2002.\n
 * URL: http://www.sciencedirect.com/science/article/pii/S0020025502002086,\n
 * <a href="http://dx.doi.org/10.1016/S0020-0255(02)00208-6">
 * doi:http://dx.doi.org/10.1016/S0020-0255(02)00208-6</a>\n.
 * \n
 * Library Evolutionary Algorithms for Clustering (LEAC) is a library\n
 * for the implementation of evolutionary algorithms\n
 * focused on the partition type clustering problem. Based on the\n
 * current standards of the <ahref="http://en.cppreference.com">C++</a>
 * language, as well as on Standard\n
 * Template Library <a href="http://en.cppreference.com/w/cpp/container">STL</a>
 * and also  <a href="http://www.openblas.net/">OpenBLAS</a> to have a better performance.\n
 * \version 1.0
 * \date 2015-2017
 * \authors Hermes Robles-Berumen <hermes@uaz.edu.mx>\n
 * Sebastian Ventura <sventura@uco.es>\n
 * Amelia Zafra <azafra@uco.es>\n
 * <a href="http://www.uco.es/kdis/">KDIS</a>
 * \copyright <a href="https://www.gnu.org/licenses/gpl-3.0.en.html">GPLv3</a> license
 */

#ifndef __KGA_FKCENTROID_HPP__
#define __KGA_FKCENTROID_HPP__

#include <vector>
#include <algorithm>

#include <leac.hpp>
#include "inparam_pcpmfk.hpp"
#include "outparam_gac.hpp"

#include "plot_runtime_function.hpp"

/*! \namespace eac
  \brief Evolutionary Algorithms for Clustering
  \details Implementation of evolutionary algorithms used to solve the clustering problem

  \version 1.0
  \date   2015-2017
  \copyright GPLv3 license
*/
```

```
namespace eac {

/*! \fn gaencode::ChromFixedLength<T_FEATURE,T_REAL> kga_fkcentroid
    (inout::OutParamGAC<T_REAL,T_CLUSTERIDX> &aoop_outParamGAC,
    inout::InParamPcPmFk<T_CLUSTERIDX,T_REAL,T_FEATURE,T_FEATURE_SUM,T_INSTANCES_CLUSTER_K>
    &aiinp_inParamPcPmFk,
    const INPUT_ITERATOR aiiterator_instfirst,
    const INPUT_ITERATOR aiiterator_instlast,
    const dist::Dist<T_REAL,T_FEATURE> &aifunc2p_dist)
  \brief  KGA \cite Bandyopadhyay:Maulik:GAclustering:KGA:2002
  \details Implementation of the KGA algorithm based on
  \cite Bandyopadhyay:Maulik:GAclustering:KGA:2002.
  \returns A partition of a data set, encoded on a chromosome where
    each gene is the coordinate of a centroid. Base to following equation:
  \f[
  x_i \in C_j \leftrightarrow  \| x_i - \mu_j \|  \beginarraycmin\\  k \endarray
  \| x_i - \mu_k \|,\; j=1,2,..k,
  \f]
  where \f$mu_j\f$, represents the centroid of cluster \f$C_j\f$
  \param aoop_outParamGAC a inout::OutParamGAC with the
    output parameters of the algorithm
  \param aiinp_inParamPcPmFk a inout::InParamPcPmFk parameters
    required by the algorithm
  \param aiiterator_instfirst an InputIterator to the initial
    positions of the sequence of instances
  \param aiiterator_instlast an InputIterator to the final positions
    of the sequence of instances
  \param aifunc2p_dist an object of type dist::Dist to calculate distances
*/
template < typename T_FEATURE,
           typename T_REAL,
           typename T_FEATURE_SUM,
           typename T_INSTANCES_CLUSTER_K,
           typename T_CLUSTERIDX, //-1, 0, 1, .., K
           typename INPUT_ITERATOR
           >
gaencode::ChromFixedLength<T_FEATURE,T_REAL>
kga_fkcentroid
(inout::OutParamGAC
 <T_REAL,
 T_CLUSTERIDX>                       &aoop_outParamGAC,
 inout::InParamPcPmFk
 <T_CLUSTERIDX,
 T_REAL,
 T_FEATURE,
 T_FEATURE_SUM,
 T_INSTANCES_CLUSTER_K>              &aiinp_inParamPcPmFk,
 const INPUT_ITERATOR                aiiterator_instfirst,
 const INPUT_ITERATOR                aiiterator_instlast,
 const dist::Dist<T_REAL,T_FEATURE>  &aifunc2p_dist
 )
{
  const uintidx lconstui_numClusterFixedK =
    (uintidx) aiinp_inParamPcPmFk.getNumClusterK();
```

Defines the size of the chromosome. Specifically, each chromosome is described by a sequence of $length(Ch) = l \times k$ real-valued numbers where l is the dimension of the instances,

and k is the number of clusters [BM02a]. That is to say, the chromosome of the algorithm is written as (4.2) (See [centroid-based], page 13)

```
  /*ASSIGN SIZE FOR ALL CHROMOSOMES
   */
  gaencode::ChromFixedLength<T_FEATURE,T_REAL>::setStringSize
    ( lconstui_numClusterFixedK * data::Instance<T_FEATURE>::getNumDimensions() );


  gaencode::ChromFixedLength<T_FEATURE,T_REAL> lochromfixleng_best;

  /*VARIABLE NEED FOR POPULATION AND MATINGPOOL GENETIC
   */

  /*POPULATION CREATE
   */
  std::vector<gaencode::ChromFixedLength<T_FEATURE,T_REAL> >
    lvectorchromfixleng_population
    (aiinp_inParamPcPmFk.getSizePopulation());

  /*CREATE SPACE FOR STORE MATINGPOOL
   */
  std::vector<gaencode::ChromFixedLength<T_FEATURE,T_REAL> >
    lvectorchromfixleng_matingPool
    (aiinp_inParamPcPmFk.getSizePopulation());

  std::uniform_real_distribution<T_REAL> uniformdis_real01(0,1);

#ifdef __VERBOSE_YES

  /*ID PROC
   */
  geverboseui_idproc = 1;

  ++geiinparam_verbose;
  const char* lpc_labelAlgGA = "kga_fkcentroid";
  if ( geiinparam_verbose <= geiinparam_verboseMax ) {
    std::cout
      << lpc_labelAlgGA
      << ":  IN(" << geiinparam_verbose << ")\n"
      << "\t(output Chromosome: lochromfixleng_best["
      << &lochromfixleng_best << "]\n"
      << "\t output outparam::OutParamGAC&: "
      << "aoop_outParamGAC["
      << &aoop_outParamGAC << "]\n"
      << "\t input  InParamPcPmFk&: "
      <<"aiinp_inParamPcPmFk["
      << &aiinp_inParamPcPmFk << "]\n"
      << "\t input aiiterator_instfirst[" << *aiiterator_instfirst << "]\n"
      << "\t input aiiterator_instlast[" <<  &aiiterator_instlast << "]\n"
      << "\t input  dist::Dist<T_REAL,T_FEATURE> &aifunc2p_dist["
      << &aifunc2p_dist << ']'
      << "\n\t\tPopulation size = "
      << aiinp_inParamPcPmFk.getSizePopulation()
      << "\n\t\tProbCrossover = "
      << aiinp_inParamPcPmFk.getProbCrossover()
      << "\n\t\tProbMutation  = "
      << aiinp_inParamPcPmFk.getProbMutation()
```

```
      << "\n\t)"
      << std::endl;
  }
#endif /*__VERBOSE_YES*/

  runtime::ListRuntimeFunction<COMMON_IDOMAIN>
    llfh_listFuntionHist
    (aiinp_inParamPcPmFk.getNumMaxGenerations(),
     "Iterations",
     "Clustering metrics"
     );

  /*DECLARATION OF VARIABLES: COMPUTING STATISTICAL AND METRIC OF THE ALGORITHM*/
#ifndef __WITHOUT_PLOT_STAT
  std::ofstream               lfileout_plotStatObjetiveFunc;
  runtime::RuntimeFunctionValue<T_REAL> *lofh_SSE = NULL;
  runtime::RuntimeFunctionStat<T_REAL>
    *lofhs_statObjectiveFunc[STATISTICAL_ALL_MEASURES];
  std::vector<T_REAL>         lvectorT_statfuncObjetiveFunc;

  if ( aiinp_inParamPcPmFk.getWithPlotStatObjetiveFunc() ) {

    lvectorT_statfuncObjetiveFunc.reserve
      ( aiinp_inParamPcPmFk.getSizePopulation());
    //DEFINE FUNCTION
    lofh_SSE  = new runtime::RuntimeFunctionValue<T_REAL>
      ("SSE",
       aiinp_inParamPcPmFk.getAlgorithmoName(),
       RUNTIMEFUNCTION_NOT_STORAGE
       );

    llfh_listFuntionHist.addFuntion(lofh_SSE);

    //DEFINE FUNCTION STATISTICAL
    for  (int li_i = 0; li_i < STATISTICAL_ALL_MEASURES; li_i++) {
      lofhs_statObjectiveFunc[li_i] =
        new runtime::RuntimeFunctionStat<T_REAL>
        ( (char) li_i,
          aiinp_inParamPcPmFk.getAlgorithmoName(),
          RUNTIMEFUNCTION_NOT_STORAGE
        );
      llfh_listFuntionHist.addFuntion(lofhs_statObjectiveFunc[li_i]);
    }

    //OPEN FILE STRORE FUNCTION
    aoop_outParamGAC.setFileNameOutPlotStatObjetiveFunc
      (aiinp_inParamPcPmFk.getFileNamePlotStatObjetiveFunc(),
       aiinp_inParamPcPmFk.getTimesRunAlgorithm()
       );

    lfileout_plotStatObjetiveFunc.open
      (aoop_outParamGAC.getFileNameOutPlotStatObjetiveFunc().c_str(),
       std::ios::out | std::ios::app
       );

    lfileout_plotStatObjetiveFunc.precision(COMMON_COUT_PRECISION);

    //FUNCTION HEADER
```

```
    lfileout_plotStatObjetiveFunc
      <<  llfh_listFuntionHist.getHeaderFuntions()
      << "\n";
  }

#endif /*__WITHOUT_PLOT_STAT*/

  /*WHEN CAN MEASURE STARTS AT ZERO INVALID OFFSPRING
   */
  aoop_outParamGAC.setTotalInvalidOffspring(0);

  /*OUT: GENETIC ALGORITHM CHARACTERIZATION*/
  runtime::ExecutionTime let_executionTime = runtime::start();

  T_FEATURE *larray_maxFeactures =
    new T_FEATURE[data::Instance<T_FEATURE>::getNumDimensions()];

  T_FEATURE *larray_minFeactures =
    new T_FEATURE[data::Instance<T_FEATURE>::getNumDimensions()];

  stats::maxFeatures
    (larray_maxFeactures,
     aiiterator_instfirst,
     aiiterator_instlast
     );

  stats::minFeatures
    (larray_minFeactures,
     aiiterator_instfirst,
     aiiterator_instlast
     );
```

Population initialization. Chosen distict points from the data set are used to initialize the K cluster centers encoded in each choromosome. This is similar to the initialization od the centers in K-Means algorithm. This process es repeat for each chromosome in the population [BM02a]

```
  {/*BEGIN INITIALIZE POPULATION P(t)*/

#ifdef __VERBOSE_YES
    geverbosepc_labelstep = "(0) POPULATION INITIAL";
    ++geiinparam_verbose;
    if ( geiinparam_verbose <= geiinparam_verboseMax ) {
      std::cout
        << geverbosepc_labelstep
        << ": IN(" << geiinparam_verbose << ')'
        << std::endl;
    }
#endif /*__VERBOSE_YES*/

    for ( auto& lchromfixleng_iter: lvectorchromfixleng_population ) {

      /*DECODE CHROMOSOME
       */
      mat::MatrixRow<T_FEATURE>
        lmatrixrowt_centroidsChrom
        (lconstui_numClusterFixedK,
         data::Instance<T_FEATURE>::getNumDimensions(),
         lchromfixleng_iter.getString()
```

```
        );

      clusteringop::randomInitialize
        (lmatrixrowt_centroidsChrom,
         aiiterator_instfirst,
         aiiterator_instlast
        );

      lchromfixleng_iter.setFitness
        (-std::numeric_limits<T_REAL>::max());
      lchromfixleng_iter.setObjetiveFunc
        (std::numeric_limits<T_REAL>::max());

    }

#ifdef __VERBOSE_YES
    if ( geiinparam_verbose <= geiinparam_verboseMax ) {
      std::cout
       << geverbosepc_labelstep
       << ": OUT(" << geiinparam_verbose << ')'
       << std::endl;
    }
    --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

  } /*END INITIALIZE POPULATION P(t)*/

while ( 1 ) {

    /*BEGIN ITERATION
     */
    llfh_listFuntionHist.increaseDomainUpperBound();
```

Clustering. In this step, the cluster are formed according to the center encoded in the chromosome. This is done by assigning each point $x_i, i = 1, 2, ..., n$ to one of the clusters Cj with center z_i^* such that

$$\|x_i - \mu_j\| \leq \|x_i - \mu'_j\|,\ j' = 1, 2, ..., k,\ and\ j \neq j'$$

. All ties are resolved arbitrarily. As like the K-Means algorithm, for each cluster C_i, its new center μ^* is computed as $\mu_i^* = 1/n_j \sum_{x_i \in C_j} x_i,\ j = 1, 2, ..., k$, where n_j is the number of points in cluster C_i. These μ^* now replace the preious μ_i's in the chromosome [BM02a].

```
    {/*BEGIN CLUSTERING*/
#ifdef __VERBOSE_YES
      geverbosepc_labelstep = "A. THE CLUSTERS ARE FORMED";
      ++geiinparam_verbose;
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ":  IN(" << geiinparam_verbose << ')'
          << std::endl;
     }
#endif /*__VERBOSE_YES*/

      for ( auto& liter_iChrom: lvectorchromfixleng_population ) {
```

```
          /*DECODE CHROMOSOME*/
          mat::MatrixRow<T_FEATURE>
            lmatrixrowt_centroidsChrom
              (lconstui_numClusterFixedK,
               data::Instance<T_FEATURE>::getNumDimensions(),
               liter_iChrom.getString()
              );

          mat::MatrixRow<T_FEATURE_SUM>
            llmatrixrowt_sumInstancesCluster
              (lconstui_numClusterFixedK,
               data::Instance<T_FEATURE>::getNumDimensions(),
               T_FEATURE_SUM(0)
              );

          std::vector<T_INSTANCES_CLUSTER_K>
            lvectort_numInstancesInClusterK
              (lconstui_numClusterFixedK,
               T_INSTANCES_CLUSTER_K(0)
              );

          T_CLUSTERIDX lmcidx_numClusterNull;


          clusteringop::updateCentroids
            (lmcidx_numClusterNull,
             lmatrixrowt_centroidsChrom,
             llmatrixrowt_sumInstancesCluster,
             lvectort_numInstancesInClusterK,
             aiiterator_instfirst,
             aiiterator_instlast,
             aifunc2p_dist
            );
      }
#ifdef __VERBOSE_YES
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": OUT(" << geiinparam_verbose << ')'
          << std::endl;
      }
      --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

    } /*END CLUSTERING*/

    /*FITNESS FUNCTION
```

Fitness computation. For each chromosome, the clusters formed in the previous step are utilized computing the clustering metric, SSE, as follows:

$$SSE = \sum_{j=1}^{k} \sum_{x_i \in C_j} ||x_i - \mu_j||$$

For finding the appropriate clusters SSE has to be minimized. The fitness function of a chromosome is defined as $1/SSE$. Therefore, maximization of the fitness function will lead to minimization of the clustering metric SSE [BM02a].

```
    { /*BEGIN COMPUTED METRIC M AND FITNESS*/

#ifdef __VERBOSE_YES
      geverbosepc_labelstep = "B. COMPUTED METRIC M AND FITNESS";
      ++geiinparam_verbose;
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": IN(" << geiinparam_verbose << ')'
          << std::endl;
      }
#endif /*__VERBOSE_YES*/

      long ll_invalidOffspring = 0;

      for ( auto& lchromfixleng_iter: lvectorchromfixleng_population ) {

        /*DECODE CHROMOSOME*/
        mat::MatrixRow<T_FEATURE>
          lmatrixrowt_centroidsChrom
            (lconstui_numClusterFixedK,
             data::Instance<T_FEATURE>::getNumDimensions(),
             lchromfixleng_iter.getString()
            );

        std::pair<T_REAL,bool> lpair_SSE =
          um::SSE
            (lmatrixrowt_centroidsChrom,
             aiiterator_instfirst,
             aiiterator_instlast,
             aifunc2p_dist
            );

        lchromfixleng_iter.setObjetiveFunc(lpair_SSE.first);
        lchromfixleng_iter.setFitness(1.0 / lpair_SSE.first);
        lchromfixleng_iter.setValidString(lpair_SSE.second);

        if ( lchromfixleng_iter.getValidString() == false )
          ++ll_invalidOffspring;

#ifndef __WITHOUT_PLOT_STAT
        lvectorT_statfuncObjetiveFunc.push_back
          (lchromfixleng_iter.getObjetiveFunc());
#endif /*__WITHOUT_PLOT_STAT*/

      } //End for

      aoop_outParamGAC.sumTotalInvalidOffspring
        (ll_invalidOffspring);

#ifdef __VERBOSE_YES
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": OUT(" << geiinparam_verbose << ')'
          << std::endl;
      }
      --geiinparam_verbose;
```

```
#endif /*__VERBOSE_YES*/

    } /*END COMPUTED METRIC M AND FITNESS*/
```

Elitism. Has been implemented in each generation by replacing the worst chromosome of the population with the best one seen up to the previous generation [BM02a].

```
    { /*BEGIN ELITISM REPLACING THE WORST CHROMOSOME*/
#ifdef __VERBOSE_YES
      geverbosepc_labelstep = "ELITISM REPLACING THE WORST CHROMOSOME";
      ++geiinparam_verbose;
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": IN(" << geiinparam_verbose << ')'
          << std::endl;
      }
#endif /*__VERBOSE_YES*/

      auto lit_chromMin =
        std::min_element
          (lvectorchromfixleng_population.begin(),
           lvectorchromfixleng_population.end(),
           [](const gaencode::ChromFixedLength<T_FEATURE,T_REAL>& x,
              const gaencode::ChromFixedLength<T_FEATURE,T_REAL>& y
             )
           { return x.getFitness() < y.getFitness(); }
          );

      if ( lit_chromMin->getFitness() < lochromfixleng_best.getFitness() ) {
        *lit_chromMin = lochromfixleng_best;
      }

#ifdef __VERBOSE_YES
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": OUT(" << geiinparam_verbose << ')'
          << std::endl;
      }
      --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

    } /*END ELITISM REPLACING THE WORST CHROMOSOME*/
```

The best string or chromosome seen up to the last generation provides the solution to the clustering problem [BM02a].

```
    { /*BEGIN PRESERVING THE BEST STRING*/


      auto lchromfixleng_iterMax  =
        std::max_element
          (lvectorchromfixleng_population.begin(),
           lvectorchromfixleng_population.end(),
           [](const gaencode::ChromFixedLength<T_FEATURE,T_REAL>& x,
              const gaencode::ChromFixedLength<T_FEATURE,T_REAL>& y
           )
            {  return x.getFitness() < y.getFitness(); }
          );
```

```
#ifdef __VERBOSE_YES
      geverbosepc_labelstep = "ELITISM PRESERVING THE BEST";
      ++geiinparam_verbose;
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": IN(" << geiinparam_verbose << ")\tmax fitness = "
          << lchromfixleng_iterMax->getFitness()
          << std::endl;
      }
#endif /*__VERBOSE_YES*/

      if ( lochromfixleng_best.getFitness() <
           lchromfixleng_iterMax->getFitness() ) {

        /*CHROMOSOME ONE WAS FOUND IN THIS ITERATION*/
        lochromfixleng_best  = *lchromfixleng_iterMax;

        aoop_outParamGAC.setIterationGetsBest
          (llfh_listFuntionHist.getDomainUpperBound());
        aoop_outParamGAC.setRunTimeGetsBest
          (runtime::elapsedTime(let_executionTime));
      }

#ifdef __VERBOSE_YES
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": OUT(" << geiinparam_verbose << ')'
          << std::endl;
      }
      --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

    } /*END PRESERVING THE BEST STRING*/


    /*MEASUREMENT BEST: COMPUTING STATISTICAL AND METRIC OF THE
      ALGORITHM
    */
#ifndef __WITHOUT_PLOT_STAT
    if ( aiinp_inParamPcPmFk.getWithPlotStatObjetiveFunc() ) {

      lofh_SSE->setValue(lochromfixleng_best.getObjetiveFunc());

      functionhiststat_evaluateAll
        (lofhs_statObjectiveFunc,
         lvectorT_statfuncObjetiveFunc
         );
      lfileout_plotStatObjetiveFunc << llfh_listFuntionHist;
      lvectorT_statfuncObjetiveFunc.clear();
    }
#endif /*__WITHOUT_PLOT_STAT*/

    /*TERMINATION CRITERION
      3.1.5 TERMINATION CRITERION
      [BM02a]     */
```

```
#ifdef __VERBOSE_YES
    /*ID PROC
     */
    ++geverboseui_idproc;

    ++geiinparam_verbose;
    if ( geiinparam_verbose <= geiinparam_verboseMax ) {
      std::cout
        << "TERMINATION CRITERION ATTAINED?: "
        << llfh_listFuntionHist.getDomainUpperBound()
        << std::endl;
    }
    --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

    if ( !(llfh_listFuntionHist.getDomainUpperBound()
           < aiinp_inParamPcPmFk.getNumMaxGenerations() )
         )
      break;
    /*3.1.4 GENETIC OPERATIONS
      [BM02a]      */
```

Selection. The selection process selects chromosomes from the mating pool directed by the survival of the fittest concept of natural genetic systems. In the proportional selection strategy adopted in this paper, a chromosome is assigned a number of copies, which is proportional to its fitness in the population [BM02a].

```
    { /*BEGIN SELECTION*/
#ifdef __VERBOSE_YES
      geverbosepc_labelstep = "SELECTION";
      ++geiinparam_verbose;
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": IN(" << geiinparam_verbose << ')'
          << std::endl;
      }
#endif /*__VERBOSE_YES*/


      const std::vector<T_REAL>&& lvectorT_probDistRouletteWheel =
        prob::makeDistRouletteWheel
        (lvectorchromfixleng_population.begin(),
         lvectorchromfixleng_population.end(),
         [](const gaencode::ChromFixedLength<T_FEATURE,T_REAL>&
            lchromfixleng_iter) -> T_REAL
         {
           return lchromfixleng_iter.getFitness();
         }
         );

      /*COPY POPULATION TO STRING POOL FOR ROULETTE WHEEL
       */
      for ( auto& lchromfixleng_iter: lvectorchromfixleng_matingPool ) {

        uintidx lstidx_chrom =
          gaselect::getIdxRouletteWheel
          (lvectorT_probDistRouletteWheel,
           uintidx(0)
```

```
          );

        lchromfixleng_iter = lvectorchromfixleng_population.at(lstidx_chrom);
      }

#ifdef __VERBOSE_YES
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": OUT(" << geiinparam_verbose << ')'
          << std::endl;
      }
      --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

    } /*END SELECTION*/
```

Crossover. Is a probabilistic process that exchanges information between two parent chromosomes for generating two offspring. Here, single-point crossover with a fixed crossover probability of p_c is used. For chromosomes of length $l \times k$, a random integer, called the crossover point, is generated in the range $[1, l-1]$. The portions of the chromosomes lying to the right of the crossover point are exchanged to produce two offspring [BM02a].

```
    { /*BEGIN CROSSOVER*/
#ifdef __VERBOSE_YES
      geverbosepc_labelstep = "CROSSOVER";
      ++geiinparam_verbose;
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout << geverbosepc_labelstep
                  << ": IN(" << geiinparam_verbose << ')'
                  << std::endl;
      }
#endif /*__VERBOSE_YES*/

      long ll_invalidOffspring = 0;

      gaiterator::crossover
        (lvectorchromfixleng_matingPool.begin(),
         lvectorchromfixleng_matingPool.end(),
         lvectorchromfixleng_population.begin(),
         lvectorchromfixleng_population.end(),
         [&](const gaencode::ChromFixedLength<T_FEATURE,T_REAL>&
             aichrom_parent1,
             const gaencode::ChromFixedLength<T_FEATURE,T_REAL>&
             aichrom_parent2,
             gaencode::ChromFixedLength<T_FEATURE,T_REAL>&
             aochrom_child1,
             gaencode::ChromFixedLength<T_FEATURE,T_REAL>&
             aochrom_child2
             )
         {


           if ( uniformdis_real01(gmt19937_eng) <
                aiinp_inParamPcPmFk.getProbCrossover()  ) {


             gagenericop::onePointCrossover
               (aochrom_child1,
```

```
        aochrom_child2,
        aichrom_parent1,
        aichrom_parent2
        );

      /*DECODE CHROMOSOME CHILD1*/
      mat::MatrixRow<T_FEATURE>
        lmatrixrowt_centroidsChromChild1
        (lconstui_numClusterFixedK,
         data::Instance<T_FEATURE>::getNumDimensions(),
         aochrom_child1.getString()
         );

      std::pair<T_REAL,bool>
        lpair_SSE1 =
        um::SSE
        (lmatrixrowt_centroidsChromChild1,
         aiiterator_instfirst,
         aiiterator_instlast,
         aifunc2p_dist
         );
      aochrom_child1.setObjetiveFunc(lpair_SSE1.first);
      aochrom_child1.setFitness(1.0 / lpair_SSE1.first);
      aochrom_child1.setValidString(lpair_SSE1.second);

      if ( aochrom_child1.getValidString() == false )
        ++ll_invalidOffspring;

      /*DECODE CHROMOSOME CHILD1*/
      mat::MatrixRow<T_FEATURE>
        lmatrixrowt_centroidsChromChild2
        (lconstui_numClusterFixedK,
         data::Instance<T_FEATURE>::getNumDimensions(),
         aochrom_child2.getString()
         );

      std::pair<T_REAL,bool>
        lpair_SSE2 =
        um::SSE
        (lmatrixrowt_centroidsChromChild2,
         aiiterator_instfirst,
         aiiterator_instlast,
         aifunc2p_dist
         );

      aochrom_child2.setObjetiveFunc(lpair_SSE2.first);
      aochrom_child2.setFitness(1.0 / lpair_SSE2.first);
      aochrom_child2.setValidString(lpair_SSE2.second);

      if ( aochrom_child2.getValidString() == false )
        ++ll_invalidOffspring;

    } //if  Crossover
    else {
      aochrom_child1 =  aichrom_parent1;
      aochrom_child2 =  aichrom_parent2;
    }
  }
```

```
          );

      aoop_outParamGAC.sumTotalInvalidOffspring
        (ll_invalidOffspring);

#ifdef __VERBOSE_YES
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": OUT(" << geiinparam_verbose << ')'
          << std::endl;
      }
      --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

    } /*END CROSSOVER*/
```

Mutation. Each *liter_Chrom* chromosome undergoes mutation with a fixed probability p_m (*lr_mutationProbability*). Let M_{min} (*lchrom_minObjFunc*) and M_{max} (*lrt_maxClusteringMetric*) be the minimum and maximum values of the clustering metric, respectively, in the current population. See [Definition gaclusteringop::biDirectionHMutation], page 32,

```
    { /*BEGIN MUTATION*/
#ifdef __VERBOSE_YES
      geverbosepc_labelstep = "MUTATION";
      ++geiinparam_verbose;
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout << geverbosepc_labelstep
                  << ": IN(" << geiinparam_verbose << ')'
                  << std::endl;
      }
#endif /*__VERBOSE_YES*/

      auto lchrom_minObjFunc  =
        std::min_element
        (lvectorchromfixleng_population.begin(),
         lvectorchromfixleng_population.end(),
         [](const gaencode::ChromFixedLength<T_FEATURE,T_REAL>& x,
            const gaencode::ChromFixedLength<T_FEATURE,T_REAL>& y
            )
         {  return x.getObjetiveFunc() < y.getObjetiveFunc(); }
         );
      T_REAL  lrt_minClusteringMetric =
        lchrom_minObjFunc->getObjetiveFunc();

      auto lchrom_maxObjFunc  =
        std::max_element
        (lvectorchromfixleng_population.begin(),
         lvectorchromfixleng_population.end(),
         [](const gaencode::ChromFixedLength<T_FEATURE,T_REAL>& x,
            const gaencode::ChromFixedLength<T_FEATURE,T_REAL>& y
            )
         {  return x.getObjetiveFunc() < y.getObjetiveFunc(); }

         );

      T_REAL lrt_maxClusteringMetric =
        lchrom_maxObjFunc->getObjetiveFunc();
```

```
      for ( auto& lchromfixleng_iter: lvectorchromfixleng_population ) {

        if ( uniformdis_real01(gmt19937_eng)
             < aiinp_inParamPcPmFk.getProbMutation() )
          { //IF MUTATION
            gaclusteringop::biDirectionHMutation
              (lchromfixleng_iter,
               lrt_minClusteringMetric,
               lrt_maxClusteringMetric,
               larray_minFeactures,
               larray_maxFeactures
               );
            lchromfixleng_iter.setFitness
              (-std::numeric_limits<T_REAL>::max());
            lchromfixleng_iter.setObjetiveFunc
              (std::numeric_limits<T_REAL>::max());
          } //END IF MUTATION
      }
#ifdef __VERBOSE_YES
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": OUT(" << geiinparam_verbose << ')'
          << std::endl;
      }
      --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

    } /*END MUTATION*/


  } /*END EVOLUTION While*/

  /*FREE MEMORY
   */
  delete [] larray_maxFeactures;
  delete [] larray_minFeactures;

  runtime::stop(let_executionTime);
  aoop_outParamGAC.setNumClusterK
    (aiinp_inParamPcPmFk.getNumClusterK());
  aoop_outParamGAC.setMetricFuncRun
    (lochromfixleng_best.getObjetiveFunc());
  aoop_outParamGAC.setAlgorithmRunTime
    (runtime::getTime(let_executionTime));
  aoop_outParamGAC.setFitness
    (lochromfixleng_best.getFitness());
  aoop_outParamGAC.setNumTotalGenerations
    (llfh_listFuntionHist.getDomainUpperBound());

#ifndef __WITHOUT_PLOT_STAT
  if ( aiinp_inParamPcPmFk.getWithPlotStatObjetiveFunc() ) {
    runtime::plot_funtionHist
      (llfh_listFuntionHist,
       aiinp_inParamPcPmFk,
       aoop_outParamGAC
       );
```

```
  }

#endif /*__WITHOUT_PLOT_STAT*/

#ifdef __VERBOSE_YES
  if ( geiinparam_verbose <= geiinparam_verboseMax ) {
    geverbosepc_labelstep = lpc_labelAlgGA;
    std::cout
      << lpc_labelAlgGA
      << ": OUT(" << geiinparam_verbose << ")\n";
    lochromfixleng_best.print();
    std::cout << std::endl;
  }
  --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

  return lochromfixleng_best;

} /* END kga_fkcentroid */

} /*END eac */

#endif /*__KGA_FKCENTROID_HPP__*/
```

A.2 GA algorithm

```
/*! \file gaclustering_fkcrispmatrix.hpp
 *
 * \brief GA CLUSTERING \cite Bezdek:etal:GAclustering:GA:1994
 *
 * \details This file is part of the LEAC.\n\n
 * Implementation of the GA algorithm based on the paper:\n
 * J.C. Bezdek, S. Boggavarapu, L.O. Hall, and A. Bensaid.\n
 * Genetic algorithm guided clustering. In Evolutionary Computation,\n
 * 1994. IEEE World Congress on Computational Intelligence., Proceed-\n
 * ings of the First IEEE Conference on, pages 34--39 vol.1, Jun 1994.\n
 * <a href="http://dx.doi.org/10.1109/ICEC.1994.350046">doi:10.1109/ICEC.1994.350046</a>\n.
 * \n
 * Library Evolutionary Algorithms for Clustering (LEAC) is a library\n
 * for the implementation of evolutionary algorithms\n
 * focused on the partition type clustering problem. Based on the\n
 * current standards of the <a href="http://en.cppreference.com">C++</a>
 * language, as well as on Standard\n
 * Template Library <a href="http://en.cppreference.com/w/cpp/container">STL</a>
 * and also  <a href="http://www.openblas.net/">OpenBLAS</a> to have a better performance.\n
 * \version 1.0
 * \date 2015-2017
 * \authors Hermes Robles-Berumen <hermes@uaz.edu.mx>\n
 * Sebastian Ventura <sventura@uco.es>\n
 * Amelia Zafra <azafra@uco.es>\n
 * <a href="http://www.uco.es/kdis/">KDIS</a>
 * \copyright <a href="https://www.gnu.org/licenses/gpl-3.0.en.html">GPLv3</a> license
 */

#ifndef __GACLUSTERING_FKCRISPMATRIX_HPP__
#define __GACLUSTERING_FKCRISPMATRIX_HPP__
```

```
#include <iostream>
#include <iomanip>
#include <vector>

#include <leac.hpp>

#include "plot_runtime_function.hpp"
#include "inparam_withoutpcpmfk.hpp"
#include "outparam_gac.hpp"

/*! \namespace eac
  \brief Evolutionary Algorithms for Clustering
  \details Implementation of evolutionary algorithms used to solve the clustering problem

  \version 1.0
  \date   2015-2017
  \copyright GPLv3 license
*/

namespace eac {

/*! \fn gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>
gaclustering_fkcrispmatrix
(inout::OutParamGAC<T_REAL,T_CLUSTERIDX> &aoop_outParamGAC,
inout::InParamWithoutPcPm<T_CLUSTERIDX,T_BITSIZE,T_FEATURE,
T_FEATURE_SUM,T_INSTANCES_CLUSTER_K> &aiinp_inParamWithoutPcPmFk,
const INPUT_ITERATOR aiiterator_instfirst,
const INPUT_ITERATOR aiiterator_instlast, dist::Dist<T_REAL,T_FEATURE> &aifunc2p_dist)
  \brief  gaclustering_fkcrispmatrix
  \details GA clustering based on [BBHB94]   Returns a crisp matrix, which encodes a par-
tition of a data set, for a defined k.
  \param aoop_outParamGAC a inout::OutParamGAC that contains
  information relevant to program execution
  \param aiinp_inParamWithoutPcPmFk a inout::InParamWithoutPcPm with
  the input parameters for the program configuration
  \param aiiterator_instfirst an InputIterator to the initial positions of the
   sequence of instances
  \param aiiterator_instlast an InputIterator to the final positions of the
  sequence of instances
  \param aipartition_clusters a partition of instances in clusters
  \param aifunc2p_dist an object of type dist::Dist to calculate distances
*/

template < typename T_BITSIZE,
           typename T_REAL,
           typename T_FEATURE,
           typename T_FEATURE_SUM,
           typename T_INSTANCES_CLUSTER_K, //0, 1, .., N
           typename T_CLUSTERIDX,          //-1, 0, 1, .., K
           typename INPUT_ITERATOR
   >
gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>
gaclustering_fkcrispmatrix
(inout::OutParamGAC
 <T_REAL,
 T_CLUSTERIDX>                &aoop_outParamGAC,
 inout::InParamWithoutPcPm
```

```
 <T_CLUSTERIDX,
 T_BITSIZE,
 T_FEATURE,
 T_FEATURE_SUM,
 T_INSTANCES_CLUSTER_K>       &aiinp_inParamWithoutPcPmFk,
 const INPUT_ITERATOR         aiiterator_instfirst,
 const INPUT_ITERATOR         aiiterator_instlast,
 dist::Dist<T_REAL,T_FEATURE> &aifunc2p_dist
 )
{
#ifdef __VERBOSE_YES
  /*ID PROC
   */
  geverboseui_idproc = 1;

  ++geiinparam_verbose;
  const char* lpc_labelAlgGA = "gaclustering_fkcrispmatrix";
  if ( geiinparam_verbose <= geiinparam_verboseMax ) {
    std::cout
      << lpc_labelAlgGA
      << "  IN(" << geiinparam_verbose << ")\n"
      << "\t(output outparam::OutParamGAC&: aoop_outParamGAC["
      << &aoop_outParamGAC << "]\n"
      << "\t input  InParamClusteringBezdekGA1994&: aiinp_inParamWithoutPcPmFk["
      << &aiinp_inParamWithoutPcPmFk << "]\n"
      << "\t input aiiterator_instfirst[" << *aiiterator_instfirst << "]\n"
      << "\t input aiiterator_instlast[" <<  &aiiterator_instlast << "]\n"
      << "\t input  dist::Dist<T_REAL,T_FEATURE>  &aifunc2p_dist["
      << &aifunc2p_dist << ']'
      << "\n\t\tPopulation size = "
      << aiinp_inParamWithoutPcPmFk.getSizePopulation()
      << "\n\t\tMatingPool size = "
      << aiinp_inParamWithoutPcPmFk.getSizeMatingPool()
      << "\n\t\t Generations  = "
      << aiinp_inParamWithoutPcPmFk.getNumMaxGenerations()
      << "\n\t\trandom-seed = "
      << aiinp_inParamWithoutPcPmFk.getRandomSeed()
      << "\n\t)"
      << std::endl;
  }
#endif /*__VERBOSE_YES*/

  const uintidx  luintidx_numClusterK =
    (uintidx) aiinp_inParamWithoutPcPmFk.getNumClusterK();
  const uintidx  luintidx_numIntances =
    uintidx(std::distance(aiiterator_instfirst,aiiterator_instlast));

  /*CONVERT INSTANCES TO FORMAT MATRIX
   */
  mat::MatrixRow<T_FEATURE>&& lmatrixt_y =
    data::toMatrixRow
    (aiiterator_instfirst,
     aiiterator_instlast
     );

  std::uniform_int_distribution<T_CLUSTERIDX> uniformdis_mmcidx0K
    (0,aiinp_inParamWithoutPcPmFk.getNumClusterK()-1);
```

```
  gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>
    lochrombitcrispmatrix_best(luintidx_numClusterK,luintidx_numIntances);

  /*STL container for storing the chromosome population
   */
  std::vector<gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>* >
    lvectorchrombitcrispmatrix_population;

  /*Vector for matingpool
   */
  std::vector<gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>* >
    lvectorchrombitcrispmatrix_matingPool;

  /*Vector for temporary storage when applying generic operators
   */
  std::vector<gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>* >
    lvectorchromfixleng_childR;

  if ( aiinp_inParamWithoutPcPmFk.getSizePopulation()
       <= aiinp_inParamWithoutPcPmFk.getSizeMatingPool() )
    throw std::invalid_argument
      ("gaclustering_fkcrispmatrix: "
       "size population should be greater than size matingpool"
       );

  runtime::ListRuntimeFunction<COMMON_IDOMAIN>
    llfh_listFuntionHist
    (aiinp_inParamWithoutPcPmFk.getNumMaxGenerations(),
     "Iterations",
     "Clustering metrics"
     );

  /*Declaration of variables: computing statistical
    and metric of the algorithm
   */
#ifndef __WITHOUT_PLOT_STAT
  std::ofstream               lfileout_plotStatObjetiveFunc;
  runtime::RuntimeFunctionValue<T_REAL> *lofh_J1 = NULL;
  runtime::RuntimeFunctionValue<T_INSTANCES_CLUSTER_K>
    *lofh_misclassified = NULL; /*function extra*/
  runtime::RuntimeFunctionStat<T_REAL>
    *lofhs_statObjectiveFunc[STATISTICAL_ALL_MEASURES];
  std::vector<T_REAL>       lvectorT_statfuncObjetiveFunc;

  if ( aiinp_inParamWithoutPcPmFk.getWithPlotStatObjetiveFunc() ) {

    lvectorT_statfuncObjetiveFunc.reserve
      ( aiinp_inParamWithoutPcPmFk.getSizePopulation());
    //Variable to monitor in the execution of the program
    lofh_J1  = new runtime::RuntimeFunctionValue<T_REAL>
      ("J1",
       aiinp_inParamWithoutPcPmFk.getAlgorithmoName(),
       RUNTIMEFUNCTION_NOT_STORAGE
       );

    llfh_listFuntionHist.addFuntion(lofh_J1);
```

```
    if ( aiinp_inParamWithoutPcPmFk.getClassInstanceColumn()  ) {
      lofh_misclassified =
        new runtime::RuntimeFunctionValue<T_INSTANCES_CLUSTER_K>
        ("Misclassified",
         aiinp_inParamWithoutPcPmFk.getAlgorithmoName(),
         RUNTIMEFUNCTION_NOT_STORAGE
         );
      llfh_listFuntionHist.addFuntion(lofh_misclassified);
    }

    //Statistics of variable J1 in runtime
    for  (int li_i = 0; li_i < STATISTICAL_ALL_MEASURES; li_i++) {
      lofhs_statObjectiveFunc[li_i] =
        new runtime::RuntimeFunctionStat
        <T_REAL>
        ( (char) li_i,
          aiinp_inParamWithoutPcPmFk.getAlgorithmoName(),
          RUNTIMEFUNCTION_NOT_STORAGE
          );
      llfh_listFuntionHist.addFuntion(lofhs_statObjectiveFunc[li_i]);
    }

    //OPEN FILE STRORE FUNCTION
    aoop_outParamGAC.setFileNameOutPlotStatObjetiveFunc
      (aiinp_inParamWithoutPcPmFk.getFileNamePlotStatObjetiveFunc(),
       aiinp_inParamWithoutPcPmFk.getTimesRunAlgorithm()
       );

    lfileout_plotStatObjetiveFunc.open
      (aoop_outParamGAC.getFileNameOutPlotStatObjetiveFunc().c_str(),
       std::ios::out | std::ios::app
       );

    lfileout_plotStatObjetiveFunc.precision(COMMON_COUT_PRECISION);

    //Header function
    lfileout_plotStatObjetiveFunc
      <<  llfh_listFuntionHist.getHeaderFuntions()
      << "\n";
  }
#endif /*__WITHOUT_PLOT_STAT*/


  runtime::ExecutionTime let_executionTime  = runtime::start();

  /*Create space for store population
   */
  lvectorchrombitcrispmatrix_population.reserve
    (aiinp_inParamWithoutPcPmFk.getSizePopulation() + 1);
  for (uintidx lui_i = 0;
       lui_i < aiinp_inParamWithoutPcPmFk.getSizePopulation();
       lui_i++)
    {
      lvectorchrombitcrispmatrix_population.push_back
        (new gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>
         (luintidx_numClusterK,luintidx_numIntances)
         );
    }
```

```
  /*Space for store matingpool
   */
  lvectorchrombitcrispmatrix_matingPool.reserve
    (aiinp_inParamWithoutPcPmFk.getSizeMatingPool());

  /*Space for chromosomes R
   */
  lvectorchromfixleng_childR.reserve
    (aiinp_inParamWithoutPcPmFk.getSizeMatingPool() + 1 );
```

Initialization of population. Initial population of size P, consisting of U matrices is pseudo randomly generated such that each has one at least one 1 in every row $\sum_{j=1}^{n} U_{ij} \geq 1 \forall i$) and each column sums to 1, i.e. $\sum_{i=1}^{c} U_{ij} = 1, \forall j$. The partly random initialization is obtained as follows. For each cluster center v_i, we choose the k^{th} element of the cluster center to be the k^{th} feature of a randomly chosen pattern to be clustered. This is done for each of the s elements of a cluster center. The process is repeated for each cluster center. An initial U matrix is then generated from the cluster centers. For a GA, population P (the population size) U matrices are generated in this manner [BBHB94].

```
{/*BEGIN INITIALIZE POPULATION*/

#ifdef __VERBOSE_YES
    geverbosepc_labelstep = "POPULATION INITIALIZATION";
    ++geiinparam_verbose;
    if ( geiinparam_verbose <= geiinparam_verboseMax ) {
      std::cout
        << geverbosepc_labelstep
        << ":  IN(" << geiinparam_verbose << ')'
        << std::endl;
    }
#endif /*__VERBOSE_YES*/

    mat::MatrixRow<T_FEATURE>
      lmatrixt_v
      ( luintidx_numClusterK,
        data::Instance<T_FEATURE>::getNumDimensions()
        );

    for ( auto lchrombitcrispmatrix_iter: lvectorchrombitcrispmatrix_population) {

      clusteringop::randomInitialize
        (lmatrixt_v,
         aiiterator_instfirst,
         aiiterator_instlast
         );

      clusteringop::getPartition
        (*lchrombitcrispmatrix_iter,
         lmatrixt_y,
         lmatrixt_v,
         aifunc2p_dist
         );

      T_REAL lT_j1 =
        um::j1
        (*lchrombitcrispmatrix_iter,
         lmatrixt_v,
         aiiterator_instfirst,
```

```
          aiiterator_instlast,
          aifunc2p_dist
          );

      lchrombitcrispmatrix_iter->setObjetiveFunc(lT_j1);

    }

#ifdef __VERBOSE_YES
    if ( geiinparam_verbose <= geiinparam_verboseMax ) {
      std::cout
        << geverbosepc_labelstep
        << ": OUT(" << geiinparam_verbose << ')'
        << std::endl;
    }
    --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

  } /*END INITIALIZE POPULATION*/
```

Population sort by J_1 The U matrices are sorted by J_1 value. and the R with the lowest J_1, values are choses to reproduce [BBHB94].

```
{/*BEGIN POPULATION SORT BY J_1*/

#ifdef __VERBOSE_YES
    geverbosepc_labelstep = "SORT POPULATION";
    ++geiinparam_verbose;
    if ( geiinparam_verbose <= geiinparam_verboseMax ) {
      std::cout
        << geverbosepc_labelstep
        << ":  IN(" << geiinparam_verbose << ')'
        << std::endl;
    }
#endif /*__VERBOSE_YES*/


    std::sort
      (lvectorchrombitcrispmatrix_population.begin(),
       lvectorchrombitcrispmatrix_population.end(),
       [](const gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>* x,
          const gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>* y
          )
       {  return x->getObjetiveFunc() < y->getObjetiveFunc(); }
       );


#ifdef __VERBOSE_YES

    ++geiinparam_verbose;
    if ( geiinparam_verbose <= geiinparam_verboseMax ) {

      for ( auto lchrombitcrispmatrix_iter: lvectorchrombitcrispmatrix_population) {

        lchrombitcrispmatrix_iter->print
          (std::cout,
           geverbosepc_labelstep,
           ',',
           ';'
```

```
            );
        std::cout << '\n';
      }
    }
    --geiinparam_verbose;

    if ( geiinparam_verbose <= geiinparam_verboseMax ) {
      std::cout
        << geverbosepc_labelstep
        << ": OUT(" << geiinparam_verbose << ')'
        << std::endl;
    }
    --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

  } /*END POPULATION SORT BY J_1*/


  while( true ) {

    {/*BEGIN PRESERVING THE CHROMOSOME BEST
      */
#ifdef __VERBOSE_YES
      geverbosepc_labelstep = "ELITISM PRESERVING THE BEST";
      ++geiinparam_verbose;
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ":  IN(" << geiinparam_verbose << ')'
          << std::endl;
      }
#endif /*__VERBOSE_YES*/

      if ( lvectorchrombitcrispmatrix_population[0]->getObjetiveFunc()
           < lochrombitcrispmatrix_best.getObjetiveFunc() ) {
        lochrombitcrispmatrix_best =
          *lvectorchrombitcrispmatrix_population[0];
         /*A better chromosome is found in this iteration
          */
        aoop_outParamGAC.setIterationGetsBest
          (llfh_listFuntionHist.getDomainUpperBound());
        aoop_outParamGAC.setRunTimeGetsBest
          (runtime::elapsedTime(let_executionTime));
      }

#ifdef __VERBOSE_YES
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": OUT(" << geiinparam_verbose << ')'
          << std::endl;
      }
      --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

    } /*END PRESERVING THE CHROMOSOME BEST*/

    /*COMPUTING STATISTICAL OF THE ALGORITHM
```

```
     */
#ifndef __WITHOUT_PLOT_STAT

    if ( aiinp_inParamWithoutPcPmFk.getWithPlotStatObjetiveFunc() ) {

      for ( auto lchrombitcrispmatrix_iter:
              lvectorchrombitcrispmatrix_population) {
        lvectorT_statfuncObjetiveFunc.push_back
          (lchrombitcrispmatrix_iter->getObjetiveFunc());
      }

      lofh_J1->setValue
        (lvectorchrombitcrispmatrix_population[0]->getObjetiveFunc());

      if ( lofh_misclassified != NULL ) {

        partition::PartitionCrispMatrix
          <T_BITSIZE,T_CLUSTERIDX>
          lpartitionCrispMatrix_classifierU
          (*lvectorchrombitcrispmatrix_population[0]);


        sm::ConfusionMatchingMatrix<T_INSTANCES_CLUSTER_K>&&
          lmatchmatrix_confusion =
          sm::getConfusionMatrix
          (aiiterator_instfirst,
           aiiterator_instlast,
           lpartitionCrispMatrix_classifierU,
           [](const data::Instance<T_FEATURE>* aiinst_iter )
           -> T_INSTANCES_CLUSTER_K
           {
             return  T_INSTANCES_CLUSTER_K(1);
           },
           [](const data::Instance<T_FEATURE>* aiinst_iter )
           -> T_CLUSTERIDX
           {
             data::InstanceClass
               <T_FEATURE,
                T_INSTANCES_CLUSTER_K,
                T_CLUSTERIDX>
               *linstclass_iter =
               (data::InstanceClass
                <T_FEATURE,
                T_INSTANCES_CLUSTER_K,
                T_CLUSTERIDX>*)
               aiinst_iter;

             return linstclass_iter->getClassIdx();

           }
           );
        lofh_misclassified->setValue
          (lmatchmatrix_confusion.getMisclassified());
      }
      functionhiststat_evaluateAll
        (lofhs_statObjectiveFunc,
         lvectorT_statfuncObjetiveFunc
         );
```

```
      lfileout_plotStatObjetiveFunc << llfh_listFuntionHist;
      lvectorT_statfuncObjetiveFunc.clear();
    }
#endif /*__WITHOUT_PLOT_STAT*/


#ifdef __VERBOSE_YES

    /*ID PROC
     */
    ++geverboseui_idproc;

    ++geiinparam_verbose;
    if ( geiinparam_verbose <= geiinparam_verboseMax ) {
      std::cout
        << "END ITERATION: "
        << llfh_listFuntionHist.getDomainUpperBound()
        << "\tobjetivoFunc = "
        << lochrombitcrispmatrix_best.getObjetiveFunc()
        << std::endl;
    }
    --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

    /*Termination criterion attained?
     */
    if ( (llfh_listFuntionHist.getDomainUpperBound()
          >= aiinp_inParamWithoutPcPmFk.getNumMaxGenerations()) ||
         (runtime::elapsedTime(let_executionTime) >
          aiinp_inParamWithoutPcPmFk.getMaxExecutiontime())
         )
      break;
```

Selection. R matrices with the lowest J_1, values are choses to reproduce [BBHB94].

```
    {/*BEGIN SELECTION
      */
      auto ichrom_population = lvectorchrombitcrispmatrix_population.begin();

      for (uintidx lui_i = 0;
           lui_i < aiinp_inParamWithoutPcPmFk.getSizeMatingPool();
           lui_i++)  {
        lvectorchrombitcrispmatrix_matingPool.push_back
          (*ichrom_population);
        ++ichrom_population;
      }

    }/*END SELECTION*/
```

Crossover operator. The crossover point and number of columns in the two U matrices chosen for reproduction are randomly chosen. The columns of the matrices are combined to create the children matrices [BBHB94].

```
    {/*BEGIN CROSSOVER OPERATORS*/

#ifdef __VERBOSE_YES
      geverbosepc_labelstep = "CROSSOVER OPERATORS";
      ++geiinparam_verbose;
```

```
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ":  IN(" << geiinparam_verbose << ')'
          << std::endl;
      }
#endif /*__VERBOSE_YES*/

      for (uintidx lui_i = 0;
           lui_i < aiinp_inParamWithoutPcPmFk.getSizeMatingPool();
           lui_i++) {
        lvectorchromfixleng_childR.push_back
          (new gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>
           (luintidx_numClusterK,luintidx_numIntances)
           );
      }

      gaiterator::crossoverRandSelect
      (lvectorchrombitcrispmatrix_matingPool.begin(),
       lvectorchrombitcrispmatrix_matingPool.end(),
       lvectorchromfixleng_childR.begin(),
       lvectorchromfixleng_childR.end(),
       [&](gaencode::ChromosomeCrispMatrix
           <T_BITSIZE,T_CLUSTERIDX,T_REAL>* aichrom_parent1,
           gaencode::ChromosomeCrispMatrix
           <T_BITSIZE,T_CLUSTERIDX,T_REAL>* aichrom_parent2,
           gaencode::ChromosomeCrispMatrix
           <T_BITSIZE,T_CLUSTERIDX,T_REAL>* aochrom_child1,
           gaencode::ChromosomeCrispMatrix
           <T_BITSIZE,T_CLUSTERIDX,T_REAL>* aochrom_child2
           )
       {

         gabinaryop::onePointDistCrossover
           (*aochrom_child1,
            *aochrom_child2,
            *aichrom_parent1,
            *aichrom_parent2
            );

         aochrom_child1->setObjetiveFunc(std::numeric_limits<T_REAL>::max());
         aochrom_child2->setObjetiveFunc(std::numeric_limits<T_REAL>::max());

       }
       );

      lvectorchrombitcrispmatrix_matingPool.clear();

#ifdef __VERBOSE_YES
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": OUT(" << geiinparam_verbose << ')'
          << std::endl;
      }
      --geiinparam_verbose;
#endif /*__VERBOSE_YES*/
```

```
    }/*END CROSSOVER OPERATORS*/
```

Mutation. Consists of randomly choosing an element of a column to have the value 1, such that it is a different element than the one currently having a value of 1 [BBHB94].

```
    {/*BEGIN MUTATION OPERATOR*/
#ifdef __VERBOSE_YES
      geverbosepc_labelstep = "MUTATION OPERATOR";
      ++geiinparam_verbose;
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ":  IN(" << geiinparam_verbose << ')'
          << std::endl;
      }
#endif /*__VERBOSE_YES*/

      for ( auto ichrom_childR: lvectorchromfixleng_childR ) {
        gabinaryop::bitMutation(*ichrom_childR);
      }

#ifdef __VERBOSE_YES
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": OUT(" << geiinparam_verbose << ')'
          << std::endl;
      }
      --geiinparam_verbose;
#endif /*__VERBOSE_YES*/
    } /*END MUTATION OPERATOR*/
```

Evaluate J_1 for childr

```
    {/*BEGIN EVALUATE J1 FOR CHILDR*/
#ifdef __VERBOSE_YES
      geverbosepc_labelstep = "EVALUATE J1 FOR CHILDR";
      ++geiinparam_verbose;
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ":  IN(" << geiinparam_verbose << ')'
          << std::endl;
      }
#endif /*__VERBOSE_YES*/

      mat::MatrixRow<T_FEATURE>
        lmatrixt_v
        (luintidx_numClusterK,
         data::Instance<T_FEATURE>::getNumDimensions()
         );

      mat::MatrixRow<T_FEATURE_SUM>
        lmatrixT_sumWX
        (lmatrixt_v.getNumRows(),
         lmatrixt_v.getNumColumns()
         );
      std::vector<T_INSTANCES_CLUSTER_K>
        lvectorT_sumWik(lmatrixt_v.getNumRows());

      for ( auto ichrom_childR: lvectorchromfixleng_childR ) {
```

```
         /*Calculate the centroid associated with U_i
          */
        clusteringop::getCentroids
          (lmatrixt_v,
           lmatrixT_sumWX,
           lvectorT_sumWik,
           *ichrom_childR,
           lmatrixt_y
           );

        T_REAL lT_j1 =
          um::j1
          (*ichrom_childR,
           lmatrixt_v,
           aiiterator_instfirst,
           aiiterator_instlast,
           aifunc2p_dist
           );

        ichrom_childR->setObjetiveFunc(lT_j1);

      }

#ifdef __VERBOSE_YES
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": OUT(" << geiinparam_verbose << ')'
          << std::endl;
      }
      --geiinparam_verbose;
#endif /*__VERBOSE_YES*/
    } /*END EVALUATE J1 FOR CHILDR*/
```

Replace. The R chuild U matrices are added to the population with the $P - R$ U matrices with the greatest J_1 values dropped from the population [BBHB94].

```
    { /*BEGIN ADD P-R U MATRICES TO POPULATION*/
#ifdef __VERBOSE_YES
      geverbosepc_labelstep = "ADD P-R U MATRICES TO POPULATION";
      ++geiinparam_verbose;
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ":  IN(" << geiinparam_verbose << ')'
          << std::endl;
      }
#endif /*__VERBOSE_YES*/

      std::sort
        (lvectorchromfixleng_childR.begin(),
         lvectorchromfixleng_childR.end(),
         [](const gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>* x,
            const gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>* y
            )
         {  return x->getObjetiveFunc() < y->getObjetiveFunc(); }
         );
```

```
      std::vector<gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>* >
        lvectorchrombitcrispmatrix_tmpL;

      lvectorchrombitcrispmatrix_tmpL.swap(lvectorchrombitcrispmatrix_population);

      /*Insert a sentinel to merge the two vectors
       */
      lvectorchrombitcrispmatrix_tmpL.push_back
        (new gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>());
      lvectorchromfixleng_childR.push_back
        (new gaencode::ChromosomeCrispMatrix<T_BITSIZE,T_CLUSTERIDX,T_REAL>());

      lvectorchrombitcrispmatrix_population.reserve
        (aiinp_inParamWithoutPcPmFk.getSizePopulation() + 1);

      uintidx luintidx_l = 0;
      uintidx luintidx_r = 0;

      for (uintidx lui_i = 0;
           lui_i < aiinp_inParamWithoutPcPmFk.getSizePopulation();
           lui_i++)
        {

          if (  lvectorchrombitcrispmatrix_tmpL[luintidx_l]->getObjetiveFunc() <
                lvectorchromfixleng_childR[luintidx_r]->getObjetiveFunc() )
            {

#ifdef __VERBOSE_YES
              ++geiinparam_verbose;
              if ( geiinparam_verbose <= geiinparam_verboseMax ) {
                std::cout
                  << " lvectorchrombitcrispmatrix_population[" << lui_i <<  ']'
                  << " <--  lvectorchrombitcrispmatrix_tmpL[" << luintidx_l << ']'
                  << '[' << & lvectorchrombitcrispmatrix_population[luintidx_l] << ']'
                  << " Fitness: "
                  <<  lvectorchrombitcrispmatrix_tmpL[luintidx_l]->getObjetiveFunc()
                  << '\n';
              }
              --geiinparam_verbose;
#endif //__VERBOSE_YES

              lvectorchrombitcrispmatrix_population.push_back
                (lvectorchrombitcrispmatrix_tmpL[luintidx_l]);
              lvectorchrombitcrispmatrix_tmpL[luintidx_l] = NULL;
              ++luintidx_l;

            }
          else {

#ifdef __VERBOSE_YES
            ++geiinparam_verbose;
            if ( geiinparam_verbose <= geiinparam_verboseMax ) {
              std::cout
                << " lvectorchrombitcrispmatrix_population[" << lui_i <<  ']'
                << " <--  lvectorchromfixleng_childR[" << luintidx_r << ']'
                << "[" << & lvectorchrombitcrispmatrix_population[luintidx_r] << ']'
                << " Fitness: "
                <<  lvectorchromfixleng_childR[luintidx_r]->getObjetiveFunc()
```

```
                << '\n';
            }
            --geiinparam_verbose;
#endif //__VERBOSE_YES

            lvectorchrombitcrispmatrix_population.push_back
              (lvectorchromfixleng_childR[luintidx_r]);
            lvectorchromfixleng_childR[luintidx_r] = NULL;
            ++luintidx_r;

          }

        }

      for (uintidx lui_i = 0;
           lui_i < lvectorchromfixleng_childR.size();
           ++lui_i) {
        if ( lvectorchromfixleng_childR[lui_i] != NULL )
          delete lvectorchromfixleng_childR[lui_i];
      }
      lvectorchromfixleng_childR.clear();

      for (uintidx lui_i = 0;
           lui_i < lvectorchrombitcrispmatrix_tmpL.size();
           ++lui_i) {
        if ( lvectorchrombitcrispmatrix_tmpL[lui_i] != NULL )
          delete lvectorchrombitcrispmatrix_tmpL[lui_i];
      }
      lvectorchrombitcrispmatrix_tmpL.clear();

#ifdef __VERBOSE_YES
      if ( geiinparam_verbose <= geiinparam_verboseMax ) {
        std::cout
          << geverbosepc_labelstep
          << ": OUT(" << geiinparam_verbose << ')'
          << std::endl;
      }
      --geiinparam_verbose;
#endif /*__VERBOSE_YES*/
    } /*END ADD P-R U MATRICES TO POPULATION*/

    /*The reproduction and survival of fittest process
      continues for some set number of generations
    */

    llfh_listFuntionHist.increaseDomainUpperBound();

  } /*while*/

  /*FREE MEMORY*/
  {/*BEGIN FREE MEMORY OF POPULATION*/

#ifdef __VERBOSE_YES
    geverbosepc_labelstep = "DELETEPOPULATION";
    ++geiinparam_verbose;
    if ( geiinparam_verbose <= geiinparam_verboseMax ) {
      std::cout
        <<  geverbosepc_labelstep
```

```
        << ":  IN(" << geiinparam_verbose << ')'
        << std::endl;
    }
#endif /*__VERBOSE_YES*/

    for (uintidx lui_i = 0;
         lui_i < lvectorchrombitcrispmatrix_population.size();
         ++lui_i) {
      delete lvectorchrombitcrispmatrix_population[lui_i];
    }

#ifdef __VERBOSE_YES
    if ( geiinparam_verbose <= geiinparam_verboseMax ) {
      std::cout
        << geverbosepc_labelstep
        << ": OUT(" << geiinparam_verbose << ')'
        << std::endl;
    }
    --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

  }/*END FREE MEMORY OF POPULATION*/

  runtime::stop(let_executionTime);
  aoop_outParamGAC.setNumClusterK
    (aiinp_inParamWithoutPcPmFk.getNumClusterK());
  aoop_outParamGAC.setMetricFuncRun
    (lochrombitcrispmatrix_best.getObjetiveFunc());
  aoop_outParamGAC.setAlgorithmRunTime
    (runtime::getTime(let_executionTime));

  aoop_outParamGAC.setFitness
    (lochrombitcrispmatrix_best.getObjetiveFunc());
  aoop_outParamGAC.setNumTotalGenerations
    (llfh_listFuntionHist.getDomainUpperBound());

  /*FREE: COMPUTING STATISTICAL AND METRIC OF THE ALGORITHM
   */
#ifndef __WITHOUT_PLOT_STAT

  if ( aiinp_inParamWithoutPcPmFk.getWithPlotStatObjetiveFunc() ) {
    plot_funtionHist
      (llfh_listFuntionHist,
       aiinp_inParamWithoutPcPmFk,
       aoop_outParamGAC
       );
  }

#endif /*__WITHOUT_PLOT_STAT*/



#ifdef __VERBOSE_YES
  geverbosepc_labelstep = lpc_labelAlgGA;
  if ( geiinparam_verbose <= geiinparam_verboseMax ) {
    std::cout
      << lpc_labelAlgGA
      << " OUT(" << geiinparam_verbose << ")\n";
```

```
    std::setprecision(COMMON_COUT_PRECISION);

    mat::MatrixRow<T_FEATURE>
      lmatrixt_vBestChrom
      ( luintidx_numClusterK,
        data::Instance<T_FEATURE>::getNumDimensions()
        );

    mat::MatrixRow<T_FEATURE_SUM>
      lmatrixT_sumWX
      (lmatrixt_vBestChrom.getNumRows(),
       lmatrixt_vBestChrom.getNumColumns()
       );

    std::vector<T_INSTANCES_CLUSTER_K>
      lvectorT_sumWik(lmatrixt_vBestChrom.getNumRows());

    clusteringop::getCentroids
      (lmatrixt_vBestChrom,
       lmatrixT_sumWX,
       lvectorT_sumWik,
       lochrombitcrispmatrix_best,
       lmatrixt_y
       );

    lochrombitcrispmatrix_best.print
      (std::cout,
       geverbosepc_labelstep,
       ',',
       ';'
       );

    std::cout << '\n';
    um::j1
      (lochrombitcrispmatrix_best,
       lmatrixt_vBestChrom,
       aiiterator_instfirst,
       aiiterator_instlast,
       aifunc2p_dist
       );

    std::cout << std::endl;

    std::setprecision(COMMON_VERBOSE_COUT_PRECISION);

  }
  --geiinparam_verbose;
#endif /*__VERBOSE_YES*/

  return lochrombitcrispmatrix_best;

} /*END gaclustering_fkcrispmatrix */

} /*END namespace alg*/

#endif /*__GACLUSTERING_FKCRISPMATRIX_HPP__*/
```

Bibliography

[**ABSSJF+12**] L. E. Agustín-Blas, S. Salcedo-Sanz, S. Jiménez-Fernández, L. Carro-Calvo, J. Del Ser, and J. A. Portilla-Figueras. *A new grouping genetic algorithm for clustering problems*. Expert Syst. Appl., 39(10):9695–9703, August 2012. doi:http://dx.doi.org/10.1016/j.eswa.2012.02.149.

[**ACH06**] V. S. Alves, R. J. G. B. Campello, and E. R. Hruschka. *Towards a fast evolutionary algorithm for clustering*. In IEEE International Conference on Evolutionary Computation, CEC 2006, part of WCCI 2006, Vancouver, BC, Canada, 16-21 July 2006, pages 1776–1783. IEEE, 2006. doi:http://dx.doi.org/10.1109/CEC.2006.1688522.

[**AIS93**] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. SIGMOD Rec., 22(2):207–216, June 1993. doi:http://doi.acm.org/10.1145/170036.170072, doi:10.1145/170036.170072.

[**BBHB94**] J. C. Bezdek, S. Boggavarapu, L. O. Hall, and A. Bensaid. Genetic algorithm guided clustering. In Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on, pages 34–39 vol.1, Jun 1994. doi:http://dx.doi:10.1109/ICEC.1994.350046.

[**BEF84**] J. C. Bezdek, R. Ehrlich, and W. Full. Fcm: *The fuzzy c-means clustering algorithm*. Computers & Geosciences, 10(2):191–203, 1984. http://www.sciencedirect.com/science/article/pii/0098300484900207, doi:http://dx.doi.org/10.1016/0098-3004(84)90020-7.

[**DB79**] David L. Davies and Donald W. Bouldin. A cluster separation measure. Pattern Analysis and Machine Intelligence, IEEE Transactions on, PAMI-1(2):224–227, April 1979. doi:http://dx.doi.org/10.1109/TPAMI.1979.4766909.

[**BM02a**] S. Bandyopadhyay and U. Maulik. *An evolutionary technique based on k-means algorithm for optimal clustering in rn*. Inf. Sci. Appl., 146(1-4):221–237, 2002. ttp://www.sciencedirect.com/science/article/pii/S0020025502002086, doi:http://dx.doi.org/10.1016/S0020-0255(02)00208-6.

[**BM02b**] S. Bandyopadhyay and U. Maulik. *Genetic clustering for automatic evolution of clusters and application to image classification*. Pattern Recognition, 35(6):1197 – 1208, 2002. http://www.sciencedirect.com/science/article/pii/S003132030100108X, doi:http://dx.doi.org/10.1016/S0031-3203(01)00108-X.

[**BM07**] S. Bandyopadhuay and U. Maulik. *Multiobjective genetic clustering for pixel classification in remote sensing imagery*. IEEE Trans. Geosci. Remote Sensing, 45:1506–1511, May 2007

[**CdLM03**] A. Casillas, M.T. González de Lena, and R. Martínez. *Document clustering into an unknown number of clusters using a genetic algorithm*. In Václav Matoušek and Pavel Mautner, editors, Text, Speech and Dialogue, volume 2807 of Lecture Notes in Computer Science, pages 43–49. Springer Berlin Heidelberg, 2003. doi:http://dx.doi.org/10.1007/978-3-540-39398-6_7.

[**CH74**] T. Caliński and J. Harabasz. *A dendrite method for cluster analysis*. Communications in Statistics, 3(1):1–27, January 1974. doi:http://dx.doi.org/10.1080/03610927408827101.

[**CSL04**] C.-H. Chou, M.-C. Su, and E. Lai. *A new cluster validity measure and its application to image compression*. *Pattern Analysis and Applications*, 7(2):205–220, 2004. doi:http://dx.doi.org/10.1007/s10044-004-0218-1.

[**CZZ09**] Dong-Xia Chang, Xian-Da Zhang, and Chang-Wen Zheng. *A genetic algorithm with gene rearrangement for k-means clustering*. Pattern Recogn., 42(7):1210–1222, 2009. doi:http://dx.doi.org/10.1016/j.patcog.2008.11.006.

[**DAK08**] S. Das, A. Abraham, and A. Konar. *Automatic clustering using an improved differential evolution algorithm*. *Systems, Man and Cybernetics*, Part A: Systems and Humans, IEEE Transactions on, 38(1):218–237, Jan 2008. doi:http://dx.doi.org/10.1109/TSMCA.2007.909595.

[**Faw06**] Tom Fawcett. *An introduction to roc analysis*. Pattern Recogn. Lett., 27(8):861–874, June 2006. doi:http://dx.doi.org/10.1016/j.patrec.2005.10.010.

[**FKKN97**] Pasi Fränti, Juha Kivijärvi, Timo Kaukoranta, and Olli Nevalainen. *Genetic algorithms for large-scale clustering problems*. The Computer Journal, 40(9):547–554, Jan 1997. URL: https://academic.oup.com/comjnl/article-abstract/40/9/547/343025?redirectedFrom=fulltext, doi:http://dx.doi.org/:10.1093/comjnl/40.9.547.

[**Gol89**] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[**HCdC06**] E. R. Hruschka, R. J. G. B. Campello, and L. N. de Castro. *Evolving clusters in gene-expression data*. Inf. Sci., 176(13):1898–1927, July 2006. doi:http://dx.doi.org/10.1016/j.ins.2005.07.015.

[**HCFdC09**] E.R. Hruschka, R.J.G.B. Campello, A.A. Freitas, and A.C.P.L.F. de Carvalho. *A survey of evolutionary algorithms for clustering*. IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews, 39(2):133–155, March 2009. http://www.cs.kent.ac.uk/pubs/2009/2884.

[**HE03**] E. R. Hruschka and N. F. F. Ebecken. *A genetic algorithm for cluster analysis*. Intell. Data Anal., 7(1):15–25, January 2003. http://dl.acm.org/citation.cfm?id=1293920.1293922.

[**HK17**] Emrah Hancer and Dervis Karaboga. *A comprehensive survey of traditional, merge-split and evolutionary approaches proposed for determination of cluster number*. Swarm and Evolutionary Computation, 32:49 – 67, 2017. http://www.sciencedirect.com/science/article/pii/S2210650216300475, https://dx.doi.org/10.1016/j.swevo.2016.06.004.

[**HPY00**] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. SIGMOD Rec., 29(2):1–12, May 2000. URL: http://doi.acm.org/10.1145/335191.335372, doi:doi:10.1145/335191.335372.

[**HT12**] Hong He and Yonghong Tan. *A two-stage genetic algorithm for automatic clustering*. Neurocomput., 81:49–59, April 2012. doi:http://dx.doi.org/10.1016/j.neucom.2011.11.001.

[**Int10**] Intel, Santa Clara, CA, USA. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide*, Part 1, jun 2010.

[**JMF99**] A. K. Jain, M. N. Murty, and P. J. Flynn. *Data clustering: A review*. ACM Comput. Surv., 31(3):264–323, September 1999. doi:http://doi.acm.org/10.1145/331499.331504.

[**KB97**] L. I. Kuncheva and J. C. Bezdek. *Selection of cluster prototypes from data by a genetic algorithm*. In inProc. 5th Eur. Congr. Intell. Tech. Soft Comput., pages 1683–1688, 1997.

[**KM99**] K. Krishna and M. Narasimha Murty. *Genetic k-means algorithm*. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 29(3):433–439, Jun 1999. http://ieeexplore.ieee.org/document/764879/, doi:http://dx.doi.org/10.1109/3477.764879.

[**KR90**] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. John Wiley and Sons, New York, 1990.

[**LDK93**] C.B. Lucasius, A.D. Dane, and G. Kateman. *On k-medoid clustering of large data sets with the aid of a genetic algorithm: background, feasibility and comparison*. Analytica Chimica Acta, 282:647–669, 1993. http://www.sciencedirect.com/science/article/pii/000326709380130D, doi:https://doi.org/10.1016/0003-2670(93)80130-D.

[**LLF+04a**] Yi Lu, Shiyong Lu, Farshad Fotouhi, Youping Deng, and Susan J. Brown. *Fgka: a fast genetic k-means clustering algorithm*. In Proceedings of the 2004 ACM symposium on Applied computing, SAC '04, pages 622–623, New York, NY, USA, 2004. ACM. doi:http://doi.acm.org/10.1145/967900.968029.

[**LLF+04b**] Yi Lu, Shiyong Lu, Farshad Fotouhi, Youping Deng, and Susan J. Brown.*Incremental genetic k-means algorithm and its application in gene expression data analysis*. BMC Bioinformatics, 5:172, 2004.

[**MB00**] U. Maulik and S. Bandyopadhyay. *Genetic algorithm-based clustering technique*. Pattern Recognition, 33(9):1455–1465, 2000.

[**MB02**] U. Maulik and S. Bandyopadhyay. *Performance evaluation of some clustering algorithms and validity indices*. IEEE Trans. Pattern Anal. Mach. Intell., 24:1650–1654, December 2002. doi:http://dx.doi.org/10.1109/TPAMI.2002.1114856.

[**MC88**] G.W. Milligan and M.C. Cooper. *A study of standardization of variables in cluster analysis*. J. Classification, 5:181–204, 1988. doi:http://www.springerlink.com/content/t588424722r23031.

[**MC96**] C. A. Murthy and Nirmalya Chowdhury. In search of optimal clusters using genetic algorithms. Pattern Recogn. Lett., 17(8):825–832, 1996. doi:http://dx.doi.org/10.1016/0167-8655(96)00043-8.

[**Mic92**] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, Berlin,1992.

[**MRS08**] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK, 2008. http://nlp.stanford.edu/IR-book/information-retrieval-book.html.

[**NP14**] Satyasai Jagannath Nanda and Ganapati Panda. *A survey on nature inspired metaheuristic algorithms for partitional clustering*. Swarm and Evolutionary Computation, 16:1–18, 2014. http://www.sciencedirect.com/science/article/pii/S221065021300076X, doi:http://dx.doi.org/10.1016/j.swevo.2013.11.003.

[**Ran71**] William M. Rand. *Objective criteria for the evaluation of clustering methods*. Journal of the American Statistical Association, 66(336):846–850, 1971. doi:http://dx.doi.org/10.2307/2284239.

[**SL04**] Weiguo Sheng and Xiaohui Liu. *A hybrid algorithm for k-medoid clustering of large data sets*. In Evolutionary Computation, 2004. CEC2004. Congress on, volume 1, pages 77–82 Vol.1, June 2004. doi:http://dx.doi.org/10.1109/CEC.2004.1330840.

[**XB91**] Xuanli Lisa Xie and Gerardo Beni. *A validity measure for fuzzy clustering*. IEEE Trans. Pattern Anal. Mach. Intell., 13(8):841–847, August 1991. doi:http://dx.doi.org/10.1109/34.85677.

[**TY01**] Lin Yu Tseng and Shiueng Bien Yang. *A genetic approach to the automatic clustering problem*. Pattern Recognition, 34(2):415 – 424, 2001. doi:http://dx.doi.org/10.1016/S0031-3203(00)00005-4.

Appendix B GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix C Concept index