Additional File 1 - ValWorkBench: an open source Java library for cluster validation, with applications to microarray data analysis

Raffaele Giancarlo\textsuperscript{1*}, Davide Scaturro\textsuperscript{1}, Filippo Utro\textsuperscript{2}

\textsuperscript{1} Dipartimento di Matematica ed Informatica, University of Palermo, Italy.
\textsuperscript{2} Computational Biology Center, IBM T. J. Watson Research, Yorktown Heights, NY 10598, USA.

Email: raffaele@math.unipa.it; davide.scaturro@gmail.com; futro@us.ibm.com;

*Corresponding author

The datatypes package
The DataMatrix.java class

It allows to store the input data matrix $D$. The class can be instantiated by one of the following two class constructors:

\begin{verbatim}
public DataMatrix()
public DataMatrix(int nOfItems, int nOfFeatures)
\end{verbatim}

When the empty class constructor method is used to create an instance of the class, one can load the fields of a DataMatrix object via a text or spreadsheet file by means of one of the following two methods:

\begin{verbatim}
public void loadFromFile(String fName)
public void loadFromExcelFile(String fName)
\end{verbatim}

An example of input data matrix is given in Fig. 1. In general, with reference to Fig. 2, the format for a data matrix with $n$ items, each with $m$ features, is as follows:

1. The first line contains two strings indicating the number of items and features in the dataset, respectively.

2. The second line contains header information, namely, the strings $<$item name$>$<item description$>$, $<$feature name 1$>$...$<$feature name m$>$.

3. The $n$ lines that follow correspond each to an item. Each line consists of $m + 2$ strings, $m$ of which must be numeric quantities indicating the value of the corresponding feature. Except for the $<$item
Figure 1: An example of data matrix, with 112 items and 17 features.

Figure 2: The input data matrix format. The first two rows correspond to header information. Entry i denotes a generic entry of the data matrix.
description> field, no other field may have a missing value, which is indicated by *. It is important to note that, with this constraint, the data matrix must be fully specified.

4. The end of each line is marked by the ASCII character <LF> (code 010).

5. Within each line, fields are separated by the ASCII character <TAB> (code 009).

6. The decimal point is the ASCII character dot <.> (code 046).

**The SimilarityMatrix.java class**

It allows to handle and store similarity matrix data files. The class can be instantiated with the use of one of the following four class constructors:

```java
public SimilarityMatrix()
public SimilarityMatrix(int numberOfRow)
public SimilarityMatrix(DataMatrix patternMatrix)
public SimilarityMatrix(ClusterMatrix clusterMatrix, DataMatrix patternMatrix)
```

In analogy with the DataMatrix.java class, when the empty class constructor is used, one can load the fields of a SimilarityMatrix object via either the loadFromFile or the loadFromExcelFile method.

The second constructor allows to instantiate an empty object from the class with a specific number of elements, and to fill it with data recovered from either a DataMatrix or a ClusterMatrix object. The corresponding methods are the following:

```java
public void fillSMatrix(DataMatrix patternMatrix)
public void fillSMatrix(ClusterMatrix clusterMatrix, DataMatrix patternMatrix)
```

The third class constructor allows to instantiate an object from the class filled with data obtained from a DataMatrix object. Finally, the fourth constructor allows to instantiate a SimilarityMatrix object that represents the similarity matrix of a given clustering solution.

**The ClusterMatrix.java class**

It allows to store and handle clustering and gold solutions. The class can be instantiated with one of the following two class constructor methods:

```java
public ClusterMatrix()
public ClusterMatrix(int nOfCluster)
```
Figure 3: An example of a clustering solution data file. The dataset has 112 items and it is partitioned into 7 clusters. Entry 1 corresponds to cluster number zero.

Figure 4: Gold/Clustering solution data format.

In analogy with the `DataMatrix.java` class, when the empty class constructor is used to create an instance of the class, one can load the fields of a `ClusterMatrix` object via either the `loadFromFile` or the `loadFromExcelFile` method.

An example of either a gold or clustering solution data file is given in Fig. 3. In general, with reference to Fig. 4, clustering and gold solution data files have to satisfy the following format:

1. The first line gives the number of clusters in the partition according to the following format:
   
   No<SPACE>of<SPACE>Clusters<SPACE>=<SPACE>K<CR>
   
   K must be a string corresponding to a numeric value less than or equal to n.

2. Each successive line gives a cluster, arbitrarily numbered, by giving the number and a list of items in it. Each item is represented by the row number corresponding (starting from zero) to the item in the data matrix. Therefore, each line is a sequence of strings representing numeric values in the following format: <Size of clusters><TAB><Item_Index_1><SPACE><Item_Index_2><SPACE>...<SPACE><Item_Index_n><CR>.
**The ClusterList.java class**

It allows to store and handle the same type of data handled by the ClusterMatrix.java class, but with a different structure, based on linked lists. It can be instantiated by the following class constructors:

```java
public ClusterList()
public ClusterList(int nOfCluster)
```

This class is supported by the use of the class ListNode.java that allows to define nodes of a ClusterList. In analogy with the DataMatrix.java class, when the empty class constructor is used, one can load the fields of a ClusterList object via either the loadFromFile or the loadFromExcelFile method.

**The ConnetivityMatrix.java class**

It allows to store and handle a connectivity matrix. It can be instantiated by the following class constructor:

```java
public ConnetivityMatrix(int nOfItems)
```

**The IndicatorMatrix.java class**

It allows to store and handle an indicator matrix. The class can be instantiated by the following class constructor:

```java
public IndicatorMatrix(int nOfItems)
```

**The ConsensusMatrix.java class**

It allows to store and handle a consensus matrix. The class can be instantiated by the following class constructor:

```java
public ConsensusMatrix(int nOfItems)
```

**The HeaderData.java class**

It allows to store and handle the header data. It maintains all the information about the experimental set-up, e.g., for each experiment the list of the command line arguments used for the analysis of the data as well as its running time. This class can be instantiated with the use of one of the following two class constructors:

```java
public HeaderData()
public HeaderData(String algorithmName, String algParameters, String datasetName, String datasetType, String measureName, String measParameters)
```
When the empty class constructor method is used to create an instance of the class, one can load the fields of a HeaderData object via a text file by means of the following method:

```java
public void loadFromFile (String fName)
```

An example of header data file is given in Fig. 5, while its general format is given in Fig. 6. The fields have to satisfy the following format and correspond naturally to the parameters deemed essential to characterize an experiment:

1. Date: `<TAB>`<dd/mm/yyyy>`<LF>`
2. Time: `<TAB>`<hh:mm:ss>`<LF>`
3. Algorithm: `<TAB>`<Algorithm Name>`<Space>`<Algorithm parameters>`<LF>`
4. Dataset: `<TAB>`<Dataset Name>`<Space>`<Dataset Type>`<LF>`
5. Measures: `<TAB>`<Measures Name>`<Space>`<Measures Parameters>`<LF>`
The **MeasureVector.java class**

It allows to handle the measure values and store them in a data file. The class can be instantiated with the use of one of the following three class constructors:

```java
public MeasureVector()
public MeasureVector(int nOfEntries)
public MeasureVector(int nOfEntries, int[] nOfCluster, double[] measureValue)
```

In analogy with the **DataMatrix.java class**, when the empty class constructor is used, one can load the fields of a MeasureVector object via either the loadFromFile or the loadFromExcelFile method.

An example of a MeasureVector data file is given in Fig. 8, while its general format is given in Fig. 7. The file has to satisfy the following format:

1. The first line of the file has format `<No of Iterations =><Space><k><CR>`.  
2. Each of the remaining $k$ lines has format `<Cluster><Space><number><TAB><Value><LF>`, where `number` is a string corresponding to the numeric value of the iteration and `<value>` is the string corresponding to the numeric value of the measure. In case that the value of the measure is undefined, i.e., it cannot be computed, `<value>` is set to the default value NaN (short for not a number).

The **InputMeasure.java class**

It is an abstract class that encapsulates different state information fields, as well as some of the input data needed to compute an internal/external validation measure. Classes that are extensions of this one are:

- InputARand
Figure 8: An example of MeasureVector data file.
Each of them provides a constructor specific to the measure using it. Therefore, it results convenient to
describe those classes together with the corresponding class measures. However, the common inputs are
described here:

- `kMin`: an integer indicating the minimum number of clusters to be computed.
- `kMax`: an integer indicating the maximum number of clusters to be computed.
- `algCommandLine`: the clustering algorithm command line, i.e., the list of options and input
  parameters needed to execute the algorithm via shell.
- `algorithmPath`: the path of the clustering algorithm binary executable.
- `algInputPath`: the path where the algorithm take its input files.
- `algOutputPath`: the path where the algorithm stores its output files.
- `outputPath`: the path where the measure stores its output files.
- `initExtFlag`: a boolean flag indicating external initialization, it assumes value true if the clustering
  algorithm takes as input an external initialization, false otherwise.
• initExtAlgPath: the path of the clustering algorithm binary executable used to produce the external initialization.

• initExtCommandLine: the executable external clustering algorithm command line, i.e., the list of options and input parameters needed to execute the algorithm producing an external initialization.

• initExtInpPath: the input path for the external initialization clustering algorithm.

• initExtOutPath: the output path for the external initialization clustering algorithm.