JCL GuideBook

Version 0.9

Baptiste Lafabregue

22 January 2018

1 Introduction

The JCL library, Java Clustering Library, have been developed by the SDC team of the ICube laboratory. Its purpose is to regroup different tools to perform clustering on generic data. This document has for objective to present how to import your data in JCL format, perform a clustering and retrieve the result.
The source of the library may be dowloaded from the following git project :
https://icube-forge.unistra.fr/lafabregue/JCL.git
For more information you may contact the SDC team members involved in this project :

- Pierre Gançarski : *gancarski@unistra.fr*
- Baptiste Lafabregue : *lafabregue@unistra.fr*

2 Data Structure

The data structure is mainly based on two objects, *jcl.data.Data* and *jcl.data.DataObject*. The object Data is the main object that represent your data. The Data contains a set of DataObject, each DataObject representing an element of your data.

2.1 Creating a DataObject

A DataObject is a generic type that can represent different kind of data. A DataObject contains some *jcl.data.attribute.Attribute*. There are different types of attributes, but the main ones are :

- AttributeNumerical : for numercical values

```
// a simple attribute with value 4.5
AttributeNumerical attNum = new AttributeNumerical(4.5);
```

- AttributeMultiDimSequence : for sequences of numerical n-uplet

```
// a sequence of length three for an element with two features
double[][] sequence = {{0.0,0.1},{2.0,2.1},{4.4,5.0}};
AttributeMultiDimSequence attSeq = new AttributeMultiDimSequence(sequence);
```

After creating the attributes you can add them to a DataObject :

```
int nbAttributes = 3;
// create a new DataObject of 3 attributes
DataObject obj = new DataObject(nbAttributes);

// add the attributes to the DataObject
obj.setAttribute(0, att1);
obj.setAttribute(1, att2);
obj.setAttribute(2, att3);
```

2.2 Creating a Data

A Data is composed of a set of DataObjects and some meta data. There are different ways to construct a Data, note that Data is an Interface, so you should use one of its implementation, currently there is only one available, *jcl.data.SimpleData*.

2.2.1 Model usage

It is recommended to use a *jcl.data.Model* when creating a new Data. A model states which metric should be used by the classification algorithm. By default, the metric used is defined statically with the method *jcl.data.attribute.AttributeMultiDimSequence.setMode(int mode)*. By default DTW

is used for sequence attributes and Euclidean for other attributes. A Model is composed of an array of *jcl.data.distance.Distance*, one per attribute, and a *jcl.data.distance.MetaDistance* to combine attributes metrics.

An example with a Data composed of an AttributeNumercial and an AttributSequence :

```
// a distance is set for every attribute
Distance[] distances = new Distance[2];
// first attribute compared with an euclidean distance between numericals
distances[0] = NumericalEuclideanDistance.getInstance();
// second attribute (sequential) compared with the DTW distance
distances[1] = jcl.data.distance.sequential.DistanceDTW.getInstance();
// defines the way the two scores are combined (possibility to weight)
MetaDistance metaDistance = MetaDistanceEuclidean.getInstance();
Model model = new Model(distances, metaDistance);
```

2.2.2 From a DataObject List

The most basic way to create a Data is to create it from a *java.util.List* :

```
List<DataObject> objects = new ArrayList<DataObject>();
Data dataset = new SimpleData(objects, model);
```

2.2.3 From a Sampler

Another way to generate your Data is to use a *jcl.data.sampling.Sampler*. You can implement this interface to automatically allow learn from the sampling and do only the final classification on the whole data.

An implementation is done for images in the Mustic project (https://icube-forge.unistra.fr/lafabregue/Mustic.git) with the class *mustic.utils.jclAdapters.ImageSampler*.

A Sampler allows to select a portion of the dataset and handle all the reading operation from the source. The number of sample has to be set before initializing the Data with the Sampler instance, otherwise the sample will represent the whole data. Two sampling mods are currently available, by random choice or by interval.

Usage example with the ImageSampler implementation :

```
Vector<String> filesPaths = new Vector<String>();
filesPaths.add("myImage.tiff");
// create a Sampler from one file
Sampler sampler = new ImageSampler(filesPaths);

// possibility to add a Mask
sampler.setMask(myMask);
// select 20% of the data
sampler.setSizeByPercent(0.2);

Data dataset = new SimpleData(sampler, model);
```

Once the Data generated, we can proceed to the classification.

3 Clustering Process

The classification of a Data is represented by an instance of the abstract class *jcl.Classification*. This class manage all the operation done by the classification methods. All classifications support the following operations :

- void classify() : learn from the Data and classify it
- void merge(Vector<Integer>cr) : merge all cluster listed in cr in one
- void split(int c, int n) : split the cluster c into n clusters
- void reclass(int c) : delete the cluster c and classify all its elements in existing clusters.

Except the classify() method some operation might not be implemented by one of the method used, in this case a *jcl.utils.exceptions.MethodNotImplementedException* is thrown.

The classify() method should be launched before any of the other. After one of this methods is launched you can get the resulting clustering though the getClusteringResult() method.

Depending of the implementation used, some parameters should be added before classifying the Data. To know the type of a given Classification, you can use the getType() method. You can refer to the *jcl.learning.methods.ClassifierUtils* class for more information about each classifier and its parameters. The implementation are separated in two categories, mono and multi strategic methods, we will detail them bellow. But before we will present how an agent method is structured.

3.1 LearningMethod and ClusteringResult

Any classification in JCL use one or more Learning methods. A single-strategic methods is a simple classification, that use only one classification method (e.g. k-means, cobweb, SOM, EM). Multi-strategic methods use multiple agents to merge the results or make them collaborate (e.g. SAMARAH, MACLAW).

A learning method is composed of the implementation of three class, *jcl.learning.LearningMethod*, *jcl.learning.LearningResult* and *jcl.learning.LearningParameters*. The LearningMethod manage the classification process, based on the parameters set in the LearningParameters, it learns a model from the Data and generate the corresponding LearningResult. The LearningResult can be used to classify the data, this is output on the *jcl.clustering.ClusteringResult* format.

The ClusteringResult is composed of a set of *jcl.clustering.Cluster*. Depending of their implementations they can retrieve different information. But mainly you can access to the list of clusters with the getClusters() method and to the result itself with getClusterMap(boolean fromSample). That last method returns an array with the cluster id assign for each element of the classified dataset. The boolean value in argument specify if the index/length of the array match the sample or the whole dataset.

3.2 SingleClassification

Every single method is executed through the generic implementation *jcl.learning.methods.monostrategy.SingleClassification*. The standard way to use it is to provide the Data used and the LearningParameters of the method used.

Here is an example with the k-means method :

```
// create Data
Data dataset = new SimpleData(sampler, model);

// create LearningParameters
ClassificationWeights weights = new GlobalWeights(dataset);
int nbClusters = 10;
int nbIters = 50;
LearningParameters lp = new ParametersKmeans(nbClusters, nbIters, weights);

// create and launch the classification
Classification classification = new SingleClassification(dataset, lp);
classification.classify();

// retrieve the result
ClusteringResult result = classification.getClusteringResult();
```

3.3 HybridClassification

As each implementation is different, this section will be focused on one example, *jcl.learning.methods.multistrategy.samarah.HybridClassification*, which is SAMARAH method.

Multi-strategic methods use single-strategic method through SingleClassifications instances. In SAMARAH this is managed with the *jcl.learning.methods.multistrategy.samarah.LearningAgent* class.

```
// create Data
Data dataset = new SimpleData(sampler, model);

// create the classification
HybridClassification classification = new HybridClassification();

// add parameters
...
classification.setParameters(nInf, nSup, minC, ps, pq, pcr);
classification.setAdvancedParameters(degradation, classRatio, solutionType, kIntern,
     kExtern, unificationType, criterion, constraintsWgt);

ClassificationWeights weights = new GlobalWeights(dataset);

// add learning agents (k-means)
classif.addAgent(new ParametersKmeans(16, 50, weights), dataset);
classif.addAgent(new ParametersKmeans(20, 50, weights), dataset);
classif.addAgent(new ParametersKmeans(24, 50, weights), dataset);
```

```
classif.setData(dataset);

// create and launch the classification
classification.classify();

// retrieve the result
ClusteringResult result = classification.getClusteringResult();
```

3.4 Other clustering options

3.4.1 Weights

It is possible to setup some weights on attributes in order to change its influence on distance computation. Weights are generally mandatory for a learning agent. A default weights can be created as shown in SingleClassification example :

```
...
// create LearningParameters
ClassificationWeights weights = new GlobalWeights(dataset);
...
```

In this case, the weights are constructed according to the Data in parameter, each attribute have the same weight. But another constructor allows to change them by giving an array with the weight of each attribute in it :

```
...
// create LearningParameters
double[] array = {0.5, 1, 0.2};
ClassificationWeights weights = new GlobalWeights(dataset);
...
```

In this case the array should have a length equal to the number of attributes in the Data.

3.4.2 Constraints

Constraints are currently managed only by SAMARAH method. Constraints can be included with the abstract class *jcl.clustering.constraints.Constraint*. There are 5 different implementations :

- MustLinkConstraint : two objects shouldn't be assigned to the same class
- CannotLinkConstraint : two objects shouldn't be assigned to the same class
- LabelConstraint : an object should be assigned to a specified class
- ClusterDiameterConstraint : the maximum diameter possible for a cluster
- NbClusterConstraint : the number of cluster should be in a specific range

To apply constraints you must add them to the Data. When object related constraints are defined we have make the choice to do it with the whole data indexes. In the case you use a sample the indexes need to be updated and the sample too, in the case constrained objects are not in the sample yet. So you have to consider that the sample size might change. To do so, you just have to use the updateAndSetConstraintsToSample(Vector<Constraint>constraints) method from your Data. You can also add some weights to the constraints with the setConstraintsWeights(Vector<Double>constraintsWeights) method.

4 Other functionalities

4.0.1 Evaluation

Different evaluation metrics are available in JCL to compare different clustering results. For example you can evaluate the compactness, the square error or the background knowledge satisfaction.

To do so you can use the getQuality(final ClusteringResult clusteringResult, final Data data, final int qualityIndex, final Vector<Constraint>constraints) static method from the class *jcl.evaluation.clustering.ClusteringEvaluation*.

For example to compute the compactness :

```
QualityIndex q;
q = ClusteringEvaluation.getQuality(a.getCR(), null, ClusteringEvaluation.WG, null);

System.out.println("Compactness : " + q.getValue());
```

4.0.2 Import/Export

To import or export a classification model you can use the class *jcl.utils.io.JCLModelExchange* with the modelToFile(String path, String classifierIndex, Classification classification) and fileToModel(String path) methods. A *jcl.utils.exceptions.MethodNotImplementedException* will be thrown if the export/import is not implemented yet for the select method.

The easiest way to use it is to directly call the *ExportResult()* and *ImportResult()* methods from the *Classification* class. Currently, only k-means and SAMARAH (with k-means agents) are supported.