ClusterLib Documentation

Ramon Ziai, Niels Ott

{rziai,nott}@sfs.uni-tuebingen.de

November 10, 2007

1 Introduction

This is the documentation of ClusterLib, a general purpose clustering library written in Java. ClusterLib does agglomerative hierarchical clustering, as described in Schulte im Walde (2003, p. 185) and thus employs a bottom-up strategy.

Clusters are merged in an iterative procedure until there are no more clusters to merge, that is, only one cluster is left. Alternatively, it is possible to specify how many clusters one would like to have. For the calculation of cluster distances, one can make use of currently five implemented *linkage methods*, all of which are again described in Schulte im Walde (2003, p. 186):

- Nearest Neighbor
- Furthest Neighbor
- Cluster Centroid Distance
- Average Cluster Distance
- Ward's Method

Section 2 provides a brief description of the two main data structures in ClusterLib, *`Cluster`* and *`DataPoint`*, while section 3 gives detailed examples of how the library can be used.

2 Data Structures

In this section, we describe how to use the two main data structures used in ClusterLib, *`Cluster`* and *`DataPoint`*. While this knowledge is not essential to using ClusterLib, it is useful if one plans to extend or customize it.

2.1 Cluster

`Cluster` is the interface subsuming both `CompositeCluster` and *`DataPoint`* (see section 2.2). Intuitively, this is due to the fact that a cluster can be either a group of data points or just a single data point. A composite cluster contains other clusters, each of which can then again be either a composite cluster or a single data point.

Typically, we are not interested in whether our cluster is a `CompositeCluster` or a *`DataPoint`*. Thus, we need a means of abstracting over this fact which is provided by the *`Cluster`* interface. In computer science literature, this particular kind of abstraction is called the COMPOSITE pattern (Gamma et al., 1995, p. 163). *`Cluster`* specifies methods which have to be implemented by all clusters, both composite and simple. These methods are:

- `boolean add(Cluster)`
- `boolean remove(Cluster)`
- `List<DataPoint> getLeaves()`
- `List<DataPoint> getLeavesOrSelf()`

Additionally, *`Cluster`* extends *`Iterable<Cluster>`*, so every *`Cluster`* can be iterated over.

Naturally, simple *`DataPoint`*s cannot add or remove *`Cluster`*s. So if these methods are called on *`DataPoint`*s, nothing will happen. `CompositeCluster`s will however do exactly what is expected here, namely add or remove the specified *`Cluster`*.

As for the leaves accessor methods, `CompositeCluster`s will return a list of all their leaves (i.e. the *`DataPoint`*s within them and all of their sub-clusters) no matter whether `getLeaves()` or `getLeavesOrSelf()` is called. *`DataPoint`*s, on the other hand, will return an empty list for `getLeaves()` and a list containing their own instance for `getLeavesOrSelf()`. This distinction exists to make it possible to iterate through clusters recursively if desired but still be able to avoid infinite recursion, i.e. repeatedly iterating over the same *`DataPoint`*.

For examples, take a look at sections 3.3 and 3.4.

2.2 DataPoint

As mentioned before, *`DataPoint`* is the interface describing a single data point in a clustering process. Also, *`DataPoint`* extends *`Cluster`* as every data point is an "atomic" cluster. In fact, the only methods added by *`DataPoint`* are `setValue(double)` and `getValue()` whose purpose is to provide read and write access to the actual double value of the data point.

A default implementation of *`DataPoint`* exists in the form of the `DefaultDataPoint` class. This class fulfills all the requirements and can be used straightaway if no further customization is needed.

One might ask now why *`DataPoint`* exists as an interface at all. After all, one could simply take `DefaultDataPoint` in most cases and extend it with a new class if more functionality is needed. While this is certainly a reasonable approach, it has one important drawback: Users cannot have custom classes which already extend another class still conform to *DataPoint* as Java only allows single inheritance (for good reasons). Thus, flexibility is provided by making *DataPoint* an interface while still retaining ease of use by offering a default implementation.

Additionally, `LabeledDataPoint` exists which extends `DefaultDataPoint` by a label in the form of a string.

3 Examples

3.1 FirstClusteringTool

This section describes the `FirstClusteringTool` class from the *de.linuxusers.clustering.examples* package. This class demonstrates the most basic usage of ClusterLib. The general procedure can be described in four steps:

1. Organize data for clustering.
2. Chose a linkage method.
3. Execute the clustering.
4. Look at the outcome of the clustering.

In our simple case, we have an `ArrayList`[1] of `Double` as data:

```
ArrayList<Double> data = new ArrayList<Double>();
data.add(1.0);
data.add(2.0);
data.add(3.0);
// and so on
```

Choosing a linkage method is straightforward. The *de.linuxusers.clustering.linkage* packages provides five classes that represent five different methods.[2] They all implement the *LinkageMethod* interface. The example uses the popular nearest neighbor method by creating a new instance of the corresponding class:

```
LinkageMethod method = new NearestNeighbour();
```

Now all relevant pieces of information for running the clustering algorithm are complete. As we have a list of doubles, we use the static method `ClusterDoubles`:

```
Cluster topCluster = HierarchicalClusterer.clusterDoubles(data,
          method);
```

`topCluster` now contains one cluster which again contains all smaller clusters (cf. section 2.1). Clustering can be stopped earlier by providing the number of clusters as third argument. In this case a *List* of clusters will be returned.

[1]Everything implementing *List* can be used.

[2]See also: Strategy pattern, Gamma et al. (1995, p. 315)

As a last step, we want to see the outcome of our clustering experiment. This can be simply done by printing a cluster, internal functions will take care of the rest (more on accessing the bits and pieces of a cluster in sections 3.3 and 3.4).

```
System.out.println(topCluster);
```

3.2 WeatherClusteringTool

With the `WeatherClusteringTool` class from the *de.linuxusers.clustering.examples* package, we add several concepts to the example from the previous section: firstly, the data is no longer a list of `Double` but of more specialized types. Secondly, we want to have three clusters instead of one large cluster.

ClusterLib comes with two types of data points. The `DefaultDataPoint` simply holds a double value. In addition, the `LabeledDataPoint` can also hold a label, e.g. a site or place name (cf. section 2.2). In the example, we have temperatures in Germany, taken from `http://www.wetter24.de` on July 26, 2007 (all in degrees Celsius). From those, we construct an `ArrayList` of *`DataPoint`* which we fill with a `LabeledDataPoint` for each city:

```
ArrayList<DataPoint> temperatures = new ArrayList<DataPoint>();
temperatures.add( new LabeledDataPoint( 28, "Berlin" ));
temperatures.add( new LabeledDataPoint( 25, "Hamburg" ));
temperatures.add( new LabeledDataPoint( 27, "Cologne" ));
temperatures.add( new LabeledDataPoint( 27, "Munich" ));
temperatures.add( new LabeledDataPoint( 29, "Frankfurt/M." ));
temperatures.add( new LabeledDataPoint( 27, "Leipzig" ));
temperatures.add( new LabeledDataPoint( 27, "Nuremberg" ));
temperatures.add( new LabeledDataPoint( 29, "Stuttgart" ));
temperatures.add( new LabeledDataPoint( 25, "Rostock" ));
```

Let's say we have the feeling that German temperatures are grouped into three regions. However, we are not sure which linkage method will show this. So we use two methods and set a limit to three clusters:

```
LinkageMethod ward = new WardsMethod();
LinkageMethod neighbour = new FurthestNeighbour();

int clusterCount = 3;
```

Similarly to the example in the previous section, we ask the `HierarchicalClusterer` to perform the clustering. This time we run it with two linkage methods using the `cluster` method which is intended for the use with lists of *`DataPoint`*. As there is a limit given, the result will be a list of clusters for each run.

```
List<Cluster> wardResults = HierarchicalClusterer.cluster(
    temperatures, ward, clusterCount);
List<Cluster> neighbourResults = HierarchicalClusterer.cluster(
    temperatures, neighbour,clusterCount);
```

Now the two lists of clusters can be traversed and printed to compare the outcome of the methods. The single data points are queried from each ***Cluster*** object using the `getLeavesOrSelf` method (more on that in section 3.3). The formatting and printing of the resulting list of ***DataPoint*** is left to the internal magic of Java and ClusterLib.

```
System.out.println("Ward's Method:");
for ( Cluster cl : wardResults ) {
    System.out.println(cl.getLeavesOrSelf());
}

System.out.println("Furthest Neighbour Method:");
for ( Cluster cl : neighbourResults ) {
    System.out.println(cl.getLeavesOrSelf());
}
```

The results show that the three clusters are identical for the two linkage methods. To our disappointment, the temperature regions found by our experiment are not connected geographically:

```
Ward's Method:
[Hamburg:25.0, Rostock:25.0]
[Cologne:27.0, Munich:27.0, Leipzig:27.0, Nuremberg:27.0]
[Berlin:28.0, Frankfurt/M.:29.0, Stuttgart:29.0]
Furthest Neighbour Method:
[Hamburg:25.0, Rostock:25.0]
[Cologne:27.0, Munich:27.0, Leipzig:27.0, Nuremberg:27.0]
[Berlin:28.0, Frankfurt/M.:29.0, Stuttgart:29.0]
```

3.3 Retrieving Data Points of a Cluster

The examples in the previous two sections did not deal with the retrieval of the data points from clusters. As a ***Cluster*** is a hierarchical data structure containing other objects implementing ***Cluster***, there is a simple convenience method called `getLeaves-OrSelf()`. This method retrieves all single data points from a cluster. Due to the Composite data structure (sections 2.2 and 2.1), this method must also be implemented by ***DataPoint*** objects. These are supposed to return a singleton list containing themselves when queried for their leaves.

The previous section indicated the use of the `getLeavesOrSelf()` method. The following example is a replacement of the previously given printing strategy. The first list of clusters used earlier is traversed manually. Additionally, the returned points are casted back to `LabeledDataPoint`. This must be done in order to access the label.

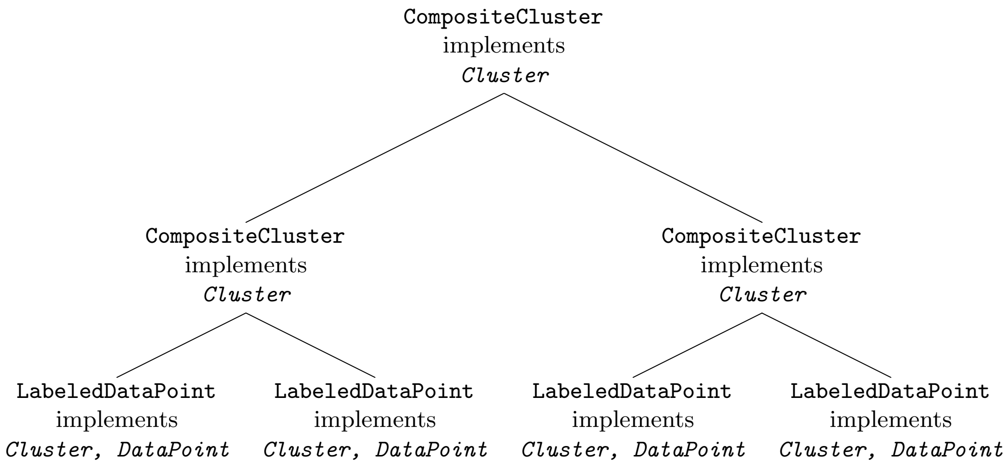
```
System.out.println("Ward's Method:");
int clusterNum = 0;
for ( Cluster cl : wardResults ) {
    clusterNum++;
    System.out.println("Cluster " + clusterNum +
        " contains:");

    // retrieve the data points and print their labels
    List<DataPoint> dpList = cl.getLeavesOrSelf();
    for ( DataPoint dp : dpList ) {
        System.out.println("     " +
            ((LabeledDataPoint)dp).getLabel());
    }
}
```

The resulting output of the code snippet given above:

```
Ward's Method:
Cluster 1 contains:
    Hamburg
    Rostock
Cluster 2 contains:
    Cologne
    Munich
    Leipzig
    Nuremberg
Cluster 3 contains:
    Berlin
    Frankfurt/M.
    Stuttgart
```

3.4 Recursive Cluster Traversal

In the previous section, we have discussed how to retrieve all leaves from a *`Cluster`* object and how to do something nice with them. While this is useful if one wants to get the data points from a collection of clusters, there will also be the case where the recursive structure of a cluster is of interest. The COMPOSITE structure of *`Cluster`* holds the data similar to a dendrogram representation.


CompositeCluster implements Cluster
CompositeCluster implements Cluster
CompositeCluster implements Cluster
LabeledDataPoint implements Cluster, DataPoint
LabeledDataPoint implements Cluster, DataPoint
LabeledDataPoint implements Cluster, DataPoint
LabeledDataPoint implements Cluster, DataPoint

So how to traverse this data structure recursively? It is worth mentioning at this point, that *`Cluster`* extends *`Iterable<Cluster>`*. In other words: A *`Cluster`* always is a collection of *`Cluster`* which makes it easy to iterate through using the *foreach*-alike construct introduced with Java 1.5. Here is the general algorithm in pseudo-code:

```
FUNCTION TraverseCluster(c):
   FOR EACH element e in c:
      IF e is a data point: PRINT e, // a leaf in the tree
      ELSE: TraverseCluster(e).      // recursion into
                                     // sub-cluster
```

An example implementation can be found in the `LatexQtreeBuilder` class from the *de.linuxusers.clustering.diagrams* package. This class is designed to create a tree from a cluster[3]. The output format serves as input of the QTree package[4] available for LaTeX:

```
private String getQTreeRecursively(Iterable<Cluster>
        clusterIterable) {

    String res = "";

    res += " [ ";

    // collect all items in the cluster
    for ( Cluster item : clusterIterable ) {

        // add leaves directly or go recursive for composites
        if ( item instanceof LabeledDataPoint ) {
            res += " {"+
                ((LabeledDataPoint)item).getLabel() +
                ":" +
                ((LabeledDataPoint)item).getValue()
                +"} ";
        } else if ( item instanceof DataPoint ) {
            res += " {"+ ((DataPoint)item).getValue() +"} ";
        } else {
            res += getQTreeRecursively(item);
        }

    }

    res += " ] ";

    return res;
}
```

The reason why this looks slightly more complicated than the pseudo-code is the following: in general, it would be enough to distinguish between *`DataPoint`* and *not* *`DataPoint`* elements of the cluster. As we want to have pretty printing of labels and values in case the element is a `LabeledDataPoint`, there is another level of distinction. Note that the *`Cluster`* always contains *`Cluster`*, so the elements must be explicitly casted down to *`DataPoint`* or `LabeledDataPoint`. The code above yields the following results when called with the `wardResults` cluster from the examples used before[5]:

[3]Or, more precisely, from any *`Iterable<Cluster>`*, which is not only implemented by *`Cluster`* but also by *`List<Cluster>`*, allowing the class to operate both on clusters and lists or vectors etc. containing clusters.

[4]`http://www.ctan.org/tex-archive/macros/latex/contrib/qtree/`

[5]With nice formatting done manually, LaTeXdoes not consider beauty of its input to be relevant.

```
[
   [  {Hamburg:25.0}  {Rostock:25.0}  ]
   [
      [  {Cologne:27.0} {Munich:27.0}  ]
      [  {Leipzig:27.0}  {Nuremberg:27.0}  ]
   ]
   [  {Berlin:28.0}
      [  {Frankfurt/M.:29.0}  {Stuttgart:29.0}  ]
   ]
]
```

References

Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston: Addison-Wesley.

Schulte im Walde, Sabine (2003). *Experiments on the Automatic Induction of German Semantic Verb Classes.* Ph.D. thesis, Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart. Published as AIMS Report 9(2).
URL `http://www.ims.uni-stuttgart.de/~schulte/Theses/PhD-Thesis/phd-thesis.pdf`

Contents

1 Introduction **1**

2 Data Structures **1**
2.1 Cluster 2
2.2 DataPoint 2

3 Examples **3**
3.1 FirstClusteringTool 3
3.2 WeatherClusteringTool 4
3.3 Retrieving Data Points of a Cluster 5
3.4 Recursive Cluster Traversal 6