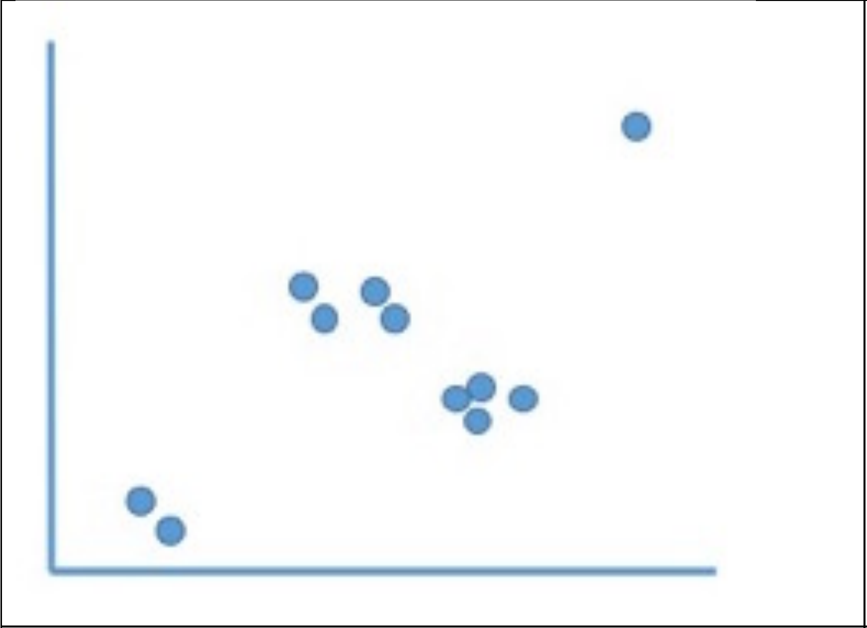
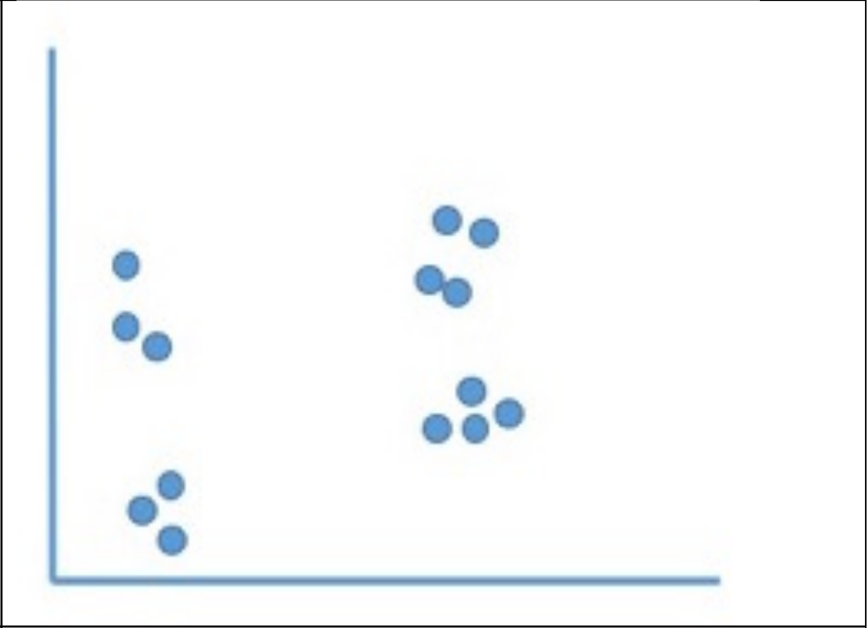
Clustering

Craig C. Douglas
University of Wyoming

Clustering is a technique that assigns data to points in some space and then groups the nearby points into clusters based on some distance measure.

	
4 clusters	4, 3, or 2 clusters?

Recall the definitions of a distance measure $d(\cdot,\cdot)$ for some space S:

1. $\forall x, y \in S$, $d(x, y) = d(y, x)$.
2. $\forall x, y \in S$, $d(x, y) \geq 0$ and $d(x, y) = 0$ iff $x = y$.
3. $\forall x, y, z \in S$, $d(x, z) \leq d(x, y) + d(y, z)$.

Common definitions of d include

1. L_p: $d(x, y) = (\sum(x_i - y_i)^p)^{1/p}$.
2. L_∞: $max|x_i - y_i|$.
3. Jacard distance
4. Hamming distance
5. Edit distance

Clustering strategies

Hierachical (agglomerative): Each point is its own cluster initially. Combine "nearby" clusters. Keep combining clusters until some criteria required fails.

Examples: Stop when

1. Some predetermined number of clusters is reached.
2. Some maximal density is reached.
3. Maximum distance between any two points in a cluster is too great.
4. Combinations of conditions.

Point assignment: Consider each point in some order. Assign each point to a cluster that it best fits.

Notes:

1. An intial set of clusters is usually done beforehand.
2. Combining or splitting clusters may occur during the assignment procedure.

Clustering algorithms fall into two categories:

1. Euclidean versus non-Euclidean spaces:
   a. In an Euclidean space we can compute the *centroid* (average location) of the data.
   b. Have to do something else in a non_euclidean spaces.
2. Data fits or does not fit into main memory.
   a. For huge data, we take shortcuts.
   b. Always keep a summary of the clusters in main memory.

High order data dimensionality features

1. All point pairs appear "equal" distances away from each other.
2. Almost any two vectors are almost orthogonal.

Consider a d-dimensional Euclidean space, where d is very large. For $x = (x_1, \dots, x_d)$ and $y = (y_1, \dots, y_d)$ and the x_i and y_i are randomly distributed in [0,1]. Then we use the L_2 distance measure and

$$max\sqrt{d}.$$

Most point pairs will have a distance close to the average distance between all points. If no points are close, then building clusters is hard and makes no sense.

Angles between vectors

Suppose we have 3 random points A, B, and C in our d-dimensional space (d very large). Assume that $A = x$, B =origin, and $C = y$. Then

$$cos(ABC) = \frac{\sum x_i y_i}{\left(\left(\sum x_i^2\right)^{1/2}\left(\sum y_i^2\right)^{1/2}\right)}.$$

Expected value of numerator is 0.

Standard deviation is $\sqrt{d}$.

Hence, the angle between any two vectors is ~90°. When *d* is sufficiently large, then

$$\begin{aligned} \text{if } \; d(A,B) &= d_1 \text{ and} \\ d(B,C) &= d_2, \text{ then} \\ d(A,C) &\approx \sqrt{d_1^2 + d_2^2}. \end{aligned}$$

Hierarchical Clustering (Euclidean spaces)

Initialization: Every point is a cluster of size 1.

Make choices:

1. How to represent clusters.
2. Criteria to merge two clusters.
3. Stopping criteria.

Algorithm:

```
while it is not time to stop do
    1.  Choose best two clusters to merge
    2.  Combine them into a single cluster
```

Since the space is Euclidean, we can compute the centroid of each cluster. Use the merging rule of combining two clusters with the closest centroids. Break ties arbitrarily.

A common way of returning the clusters is a tree data structure showing the combination history.

This leads to terrible efficiency: the obvious implementation leads to

$$n^2, (n-1)^2, (n-2)^2, \ldots, 2^2, 1^2$$

or

$$O(n^3).$$

A slightly better implementation

1. Compute all distance pairs, $O(n^2)$.
2. Form pairs and their distances into a priority queue, $O(n^2)$.
3. When clusters C and D are merged, remove all entries in priority queue for either C or D, $O(nlogn)$.
4. Compute all distances between the new cluster and the remaining clusters, $O(nlogn)$.

Overall, this is

$$O(n^2logn) < O(n^3).$$

Other rules for merging clusters

1. Use the minimum distance between any points in each cluster.
2. Use the average distance between any points in each cluster.
3. Radius=maximum distance between all points and the centroid. Merge the two clusters that will have the smallest radius.
4. Diameter=maximum distance between all points and the centroid. Merge the two clusters that will have the smallest diameter.

Stopping criteria

1. Stop when we have *k* clusters, where *k* is predetermined.
2. Stop if the density of the merged cluster is below some threshold. The density is the number of points per unit volume. The radius or diameter can be used in the calculation.
3. Stop if the merged cluster shows evidence of being a "bad" cluster.

Hierarchical Clustering (non-Euclidean spaces)

Replace the L_2 distance with some other distance measure, e.g., Jaccard, Hamming, Edit, etc. measure.

Example: Strings and Edit measure. Consider

$$d(abcd, aecdb) = 3.$$

If we merge them, what is the new cluster?

1. There is no average string.
2. Transform string, e.g., *aebcd*, but there are many options and results.
3. Once several mergers have taken place there is no realistic transformation method.

So we pick one point in non_Euclidean space to represent the merged cluster. It should represent the “center” of the cluster (a *clustroid*). Common methods of computing the clustroid are

1. Sum of distances to the other points.
2. Maximum distance to any point.
3. Sum of squares of the distances to the other points.

Example: distance=Edit. Consider 4 unique strings:

	ecdab	*abecb*	*aecdb*
abcd	5	3	3
aecdb	2	2	
abecb	4		

Apply all 3 criteria to the 4 unique strings:

Point	Sum	Max	Sum^2
abcd	11	5	43
aecdb	7	3	17
abecb	9	4	29
ecdab	11	5	45

The clustroid is *aecdb* by all 3 criteria.

How do we assign point values to data? The best known clustering method is known as *k-means*. It assumes

1. *K* is chosen *a priori*.
2. Euclidean space.

Algorithm:

1. Initially choose *k* points that should be in different clusters. Call each a centroid(/clustroid).
2. For each remaining point *p*,
    a. Find the closest centroid *c*.
    b. Add *p* to *c*'s cluster.
    c. Recompute the centroid in *c*'s cluster.
3. (Optional) Fix the final centroids and rerun step 2.

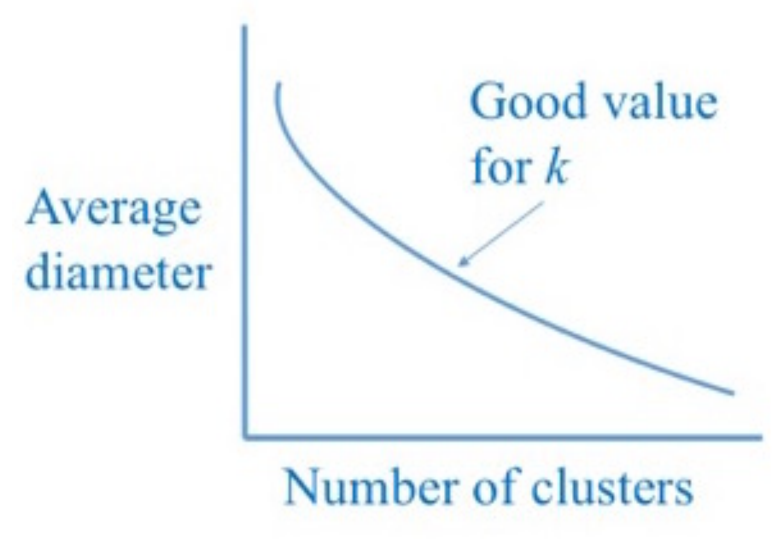
Normally step 3 makes little or no difference in the cluster assignments for points. However, if the centroids moved radically in the first pass of step 2, then there can be quite different assignments in the second pass of step 2.

Step 1 has a number of variations. There are two primary variations:

1. Pick points away from each other.
2. Cluster a small sample of the data. Choose points close to the centroids.

Choosing k

Consider the following graph, which shows typical behavior for clustering:



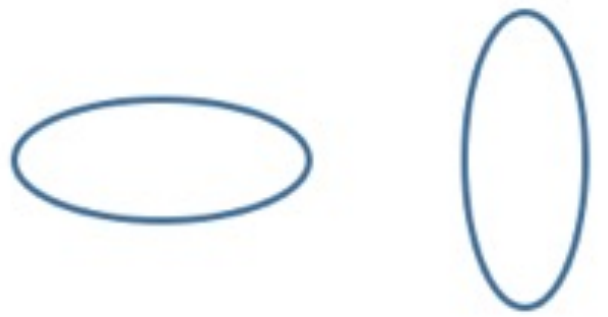
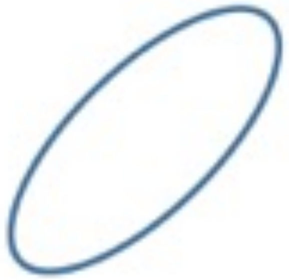
As soon as there are too few clusters, the average diameter (or radius) will grow very quickly.

We can use values $k = 1,2,4,8, \ldots, 2^j$ and evaluate the average diameter growth. When the growth is "not too much" we have a good estimate of k.

BFR Algorithm (Bradley, Fayyad, Reina)

Designed for very high dimensional data. The shape of clusters must be normally distributed about a centroid. The mean and standard deviation for a cluster may differ for different dimensions, but must be independent.

Example: *k=2*

	
Cluster shape must align with the *x* or *y* axes only.	Skew fails.

BFR assumes that the data does *not* fit into main memory.

1. Pick *k* initial points.
2. Read data in chunks. Each chunk easily fits in main memory.
3. Store in main memory.
   a. Summaries of each cluster (Discard Set).
   b. Summaries of sets of points close to one another, but not of another cluster (Compressed Set).
   c. Points not represented in points a or b (Retained Set). These points remain in main memory.

The Discard and Compressed Sets are represented by $2d+1$ numbers for a d-dimensional set:

1. N points
2. Vector of SUM=sum of all components in each of the dimensions.
3. Vector $SUMSQ$=square of sum of all components in each of the dimensions.

The goal is to represent

1. A set of points by their count: N.
2. Their centroid: in the ith dimension, SUM_i/N.
3. Standard deviation: in the ith dimension,
   $SUMSQ_i/N\text{-}(SUM_i/N)^2)$.

Example: *d*=*2* and data points $(5,1)$, $(6,-2)$, and $(7,0)$. Then

$$N = 3$$
$$SUM = (18,-1)$$
$$SUMSQ = (110,5)$$

So,

$$\text{Centroid}=(6,-1/3)$$
$$\text{Standard deviation}=(0.816,1.25)$$

Chief advantage of BFR: It is trivial to modify *N*, *SUM*, and *SUMSQ* as each point is processed. This is not true if we use centroids and standard deviations instead.

How chunks are processed

1. If a point *p* is "close enough" to a cluster, it is added to the cluster and *N*, *SUM*, and *SUMSQ* are modified for that cluster.
2. If *p* is not close enough to a cluster, it goes into the Retained Set. We use clustering algorithms within the Retained Set. Clusters of more than 1 point are moved to the Compressed Set.
3. We have miniclusters with their own (*N*,*SUM*, *SUMSQ*) summaries. These outliers are easily analyzed.
4. The miniclusters are saved to disk eventually.

Once the last chunk is processed we have to decide what to do with the Compressed and Retained Sets. We can

1. Never cluster them.
2. Add them to the nearest clusters based on centroid locations.

Ways to decide which cluster a point *p* belongs in (if any):

1. Select the cluster whose centroid is closest to *p*. Guarantee statistically that this will still be the cluster of choice after all points are processed.
2. Measure the probability that if *p* belongs to a cluster that is as far from the centroid as it is.

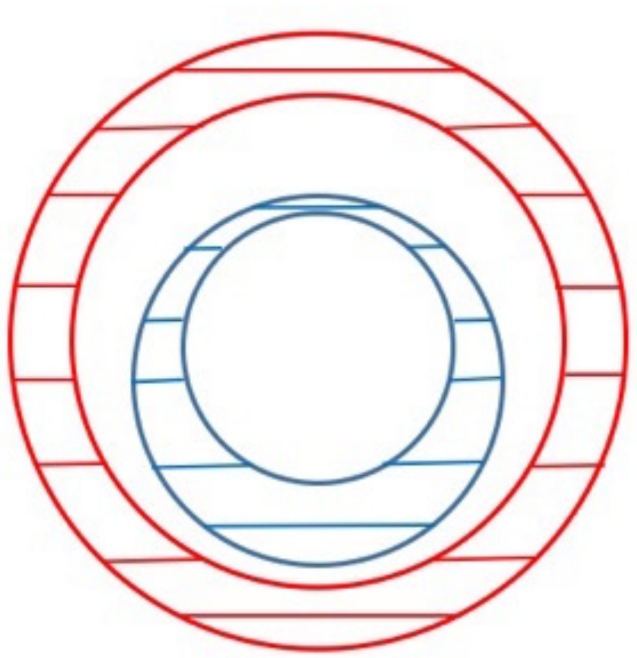
Point 2 uses our belief that each cluster consists of normally distributed points with axes of distribution aligned with the axes of the space.

The *Mahalanobis distance* is defined by the distance between a point $p = (p_1, \dots, p_d)$ and the centroid $c = (c_1, \dots, c_d)$ and scaled by the standard deviation $\sigma = (\sigma, \dots, \sigma_d)$ is

$$d(p, c) = \left(\sum \left(\frac{p_i - c_i}{\sigma_i} \right)^2 \right)^{1/2}.$$

So, when choosing a cluster to assign *p* to, choose the one with the smallest Mahalanobis distance.

Clustering Using Representatives (CURE Algorithm)



Outer cluster

Inner cluster

We now study clusters in Euclidean space that have weird shapes and cannot be presented using centroids.

Instead we have to determine a set of points that can be used to define the "center" of the clusters.

CURE Algorithm

1. Take a small subset of data and apply a hierarchical clustering algorithm in main memory.
2. Select a smaller set of representative points that are as far apart as possible.
3. Move each representative point a fixed (20% typically) distance from its location and the centroid for the representative points.
4. Merge clusters: if 2 clusters have a representative point in each cluster that are "close enough" then merge the two clusters. Keep merging until there are no more candidates.

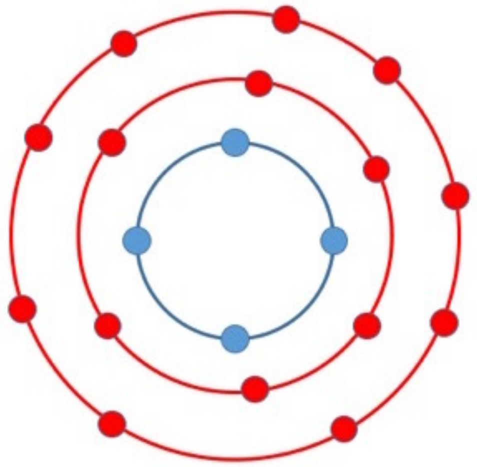
5. Point assignment: each point *p* is brought in and compared with representative points. Place *p* in whatever cluster has the closest representative point.

Steps 1-3 are the initialization steps of CURE.
Steps 4-5 are the clustering step.

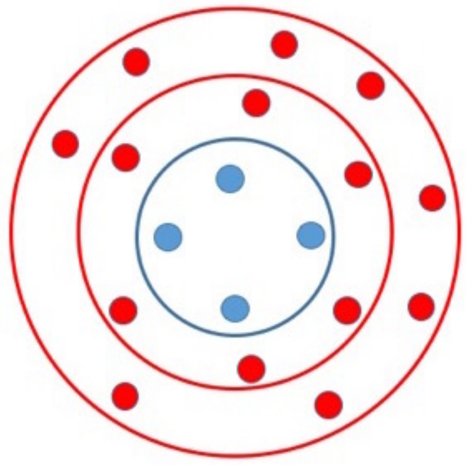
Note: Based on the data, the percentage used in Step 3 has to be picked carefully. Further, it also has an impact on how far apart representative points must be to determine if they are or are not in the same cluster.

Note: Step 3 does not work in non-Euclidean spaces in which there is no concept of a line.

Example:



After Steps 1 and 2 we could have representative points like on the left.



The representative points move towards the centroid in Step 3. The clusters are constructed in Step 4.

All of the data points are assigned to one of the clusters in Step 5.

Clustering in non-Euclidean Spaces

GRGPF Algorithm (Ganti, Ramakrishnan, Gehrke, Powell, and French)

The data representation is complicated: For each cluster,

1. N, the number of of points in the cluster.
2. The clustroid, which is defined as the point that minimizes the sum of the squares of the distances to the other points.
3. $ROWSUM(clustroid\ c) = \sum(d(c,p))^2$.
4. For some chosen k, $\{p_1, p_2, \ldots, p_k\}$, where the p_i are closest to the clustroid c. Also store $\{ROWSUM(p_i)\}$.

5. $\{q_1, q_2, \dots, q_k\}$, where the q_i are furthest from the clustroid c. Also store $\{ROWSUM(q_i)\}$.

The clusters are organized into trees. Both B-trees and R-trees are common forms.

Recall, a B-tree

- Keeps keys in sorted order for sequential traversing.
- Uses a hierarchical index to minimize the number of disk reads.
- Uses partially full blocks to speed insertions and deletions
- Keeps the index balanced with a recursive algorithm.

In addition, a B-tree minimizes waste by making sure the interior nodes are at least half full. A B-tree can handle an arbitrary number of insertions and deletions.
A R-tree is like a B-tree, but is for objects with (nearby) locations.

Each leaf in the tree holds as many clusters as is feasible. The size of a cluster is independent of the actual number of points in it.

An interior node of the cluster tree holds a sample of the clustroids of the clusters represented by each of its subtrees, plus pointers to the subtree roots.

The number of children an interior node has is independent of its level in the tree.
As we go up in the tree, the probability that a given cluster's clustroid is part of the sample decreases.

Initialization:

1. Cluster hierarchically into main memory a sample of the dataset. We now have a tree T.
2. Select from T some nodes with some desired size n (or close to n). These become the initial clusters for GRGPF. Place them in leafs of the cluster representing tree (or *CRT*).
3. Group clusters with a common ancestor in T into interior nodes of the *CRT*.

Notes:

1. Step 3 clusters descended from one interior node are as close as possible.
2. Rebalancing of the *CRT* is probable. It is similar to reorganizing a B-tree.

Adding points in GRGPF: We have to process each point *p* in the dataset.

1. Start at the root of the *CRT*. Choose the child cluster whose clustroid is closest to *p*. Examine that cluster's node next.
2. Repeat the inspection and choosing a next node until we get to a leaf. Choose the cluster in the leaf whose clustroid is closest to *p*.

3. Adjust the cluster to account for p.
   a. Add 1 to N.
   b. For all points q in the cluster representation, add $(d(p,q))^2$ to $ROWSUM(q)$.
   c. Estimate $ROWSUM(p) \approx ROWSUM(c) + N(d(p,q))^2$, where N and $ROWSUM(c)$ are the values before we included p in this cluster.
   d. Check if p is the new clustroid c.
   e. Check if p should replace one of the closest or furthest points q in the cluster representation.

Notes:

1. Typically, non-Euclidean spaces have similarities to high dimensional Euclidean spaces. If we assume that the angle between p, c, q is a right angle, the Pythagorean theorem says
$$(d(p,q))^2 = (d(p,c))^2 + (d(c,q))^2$$
Hence, the *ROWSUM* updates make sense.
2. As you walk down the *CRT*, some clustroids may be in more than one level in the *CRT*.

Splitting and merging clusters in GRGPF: GRGPF assumes that the radius of each cluster satisfies some maximum limit. Define

$$radius = \sqrt{ROWSUM(c)/N}.$$

Splitting:

1. Bring all of the cluster's points back into main memory.
2. Divide the cluster into two to minimize the *ROWSUM* values.
3. Recompute each new cluster's representation.

The *CRT* is managed like a B-tree.

If we overflow main memory within the *CRT*, then we have to start over and raise the radius limit. ☹☹

Merging: $C = C_1 \cup C_2$, clustroids c_1 and c_2.

1. Assume clustroid *c* for *C* is one of the *k* points far away from c_1 in C_1's features.
2. $ROWSUM(p) = ROWSUM_{C_1}(p) + N_{C_2}((d(p, c_1)^2 + (d(c_1, c_2)^2) + ROWSUM_{C_2}(c_2)$.
3. Recompute the features of *C*. This is based solely on the features of C_1 and C_2.

Clustering for Streams

Use a sliding window and keep $N \gg 0$ "points" of data.

For any $m \leq N$, do clustering on the m points. The clustering algorithm depends on the data space.

A simple way of keeping track of the data is store data in buckets containing 2^k points. Allow up to 2 buckets of size 2^k, k fixed.

The bucket contents are
1. Size of the bucket.

2. Timestamp of the most recent point added to the bucket.
3. Cluster representations with each having
    a. Size of cluster.
    b. Centroid or clustroid.
    c. Any feature needed to merge clusters.

When a new point arrives it must go into a bucket. This causes bucket management issues. Buckets also time out and can be deleted.

Answering queries

Given $m \leq N$, we pool all of the points and clusters from smallest set of buckets with at least m total points. The worst case $2m$.

We merge clusters from the buckets.

Parallel Computing

See Wooyoung Kim, Parallel Clustering Algorithms: Survey
http://www.cs.gsu.edu/~wkim/index_files/SurveyParallelClustering.html.