HPdcluster – Distributed Clustering

User Guide

Contents

User Guide

1. Introduction

The HPdcluster is a package for distributed clustering. It is written based on the distributed R infrastructure developed by HP-Labs. At the current version, it contains only distributed kmeans algorithm.The main functions of the HPdcluster package are:

- hpdkmeans
- sampleLoader: It is an example for loading a set of samples stored in a table into a darray.
- centerLoader: It is an example for loading a set of centers stored in a table into a matrix.

2. hpdkmeans

hpdkmeans function is a distributed version of kmeans function which is available in stats package of R. Syntax of hpdkmeans is designed very similar to kmeans, but the input samples should be in darray format. Moreover, hpdkmeans currently supports only Lloyd algorithm. The complete information about the syntax of hpdkmeans can be found in its manual.

Based on the nature of kmeans algorithm, its result depends upon the initial cluster centers because:

- The heuristic algorithm may trap in a local minimum based on the start-point.
- Based on the start-point, different number of iterations might be required for convergence.
- Even having the samples categorized in the same clusters, it is very likely that the order of the clusters and therefore their labels are different.

hpdkmeans and kmeans are the same in this behavior.

It should be noticed that hpdkmeans performs only on numerical values. Moreover, it has the capability to exclude the samples with invalid values; i.e., NA, NaN, or Inf.

In this user guide, a few samples are presented. The user should refer to the manual of hpdkmeans for complete information about the name and meaning of the arguments.

2.1. Notes

This section explains the meaning of several important warning or error messages which may appear after running the function.

- Under which circumstances the warning message “did not converge in 10 iterations” appears?

After each iteration, hpdkmeans compares the newly calculated set of centers with the old one. When there is no update in the centers, the algorithm is considered converged. If it reaches to the maximum number of iterations before convergence, it will throw the above warning message. The required number of iterations for convergence and the quality of the result are very closely dependent to the set of initial centers. The algorithm kmeans has the same behavior.

When there is such a warning, a data scientist can observe the value of *withinss* and other returned parameters that indicate the quality of returned clusters in order to decide whether it is good enough or not. In the case that she/he finds that clusters are in a good shape, but the a few more iterations can make the centers more accurate, she/he can use the founded centers as the initial centers of another call to the function. This way the time spent on the previous run is not wasted.

- How can *nstart* argument help to achieve better results?

When *nstart* is bigger than 1, the algorithm will be executed several times, and the best result will be returned. Therefore, some of the runs may not converge which means they will throw warning, but they may not be the one which is returned. The best result is the one with highest value of *withinss* regardless of its number of iterations. The user can check the value of *iter* to learn about the number of iterations related to the returned result.

- Under which circumstances the warning message “empty cluster: try a better set of initial centers” appears?

It may happen, especially when the initial centers are not well selected, that a cluster become empty (no data-point belonging to it) after one or more iterations. At this situation the above warning message will be thrown because an empty cluster is not very meaningful. A data scientist should decide if the algorithm should be executed again. Based on the nature of the data, she/he may decide to repeat the algorithm with the same number of centers but different values, or even a different number of centers.

- When the error message “unsuccessfully tried '10' times to randomly pick district centers. Please specify centers manually” appears?

The current implementation of hpdkmeans mimics kmeans function for selecting the initial centers. They both sample the datapoints based on uniform probability distribution which means all the samples have equal chance to be picked as a center. Nevertheless, all the centers should be distinct while it is always possible to have duplicate data points in the dataset. To address this problem, kmeans first

refines the whole dataset to its distinct data points and then samples the centers. This approach cannot be efficiently applied to a huge dataset stored in a darray. Therefore, hpdkmeans simply reruns the sampling when there are is any duplicate in the centers. Failing to randomly find 10 distinct centers in a huge number of rows indicates a lot of duplicate data points in the dataset.

2.2. Example 1

This clustering example (Figure 1) uses a small data-frame called 'iris', which is available in the standard distribution of R (datasets package).

```
> library(HPdcluster)     # loading the library
Loading required package: distributedR
Loading required package: Rcpp
Loading required package: RInside
Loading required package: MatrixHelper
> distributedR_start()    # starting the distributed environment
Workers registered - 1/1.
All 1 workers are registered.
[1] TRUE
> (ds <- distributedR_status())    # saving the status
         Workers Inst SysMem MemUsed DarrayQuota DarrayUsed
1 127.0.0.1:9090    7   7804    7134        3511          0
> nparts <- sum(ds$Inst)  # number of available distributed instances
> # loadind darray with the input data
> X <-
as.darray(as.matrix(cbind(iris$Sepal.Length,iris$Sepal.Width,iris$Petal.Length,iris$Petal.W
idth)),c(ceiling(length(iris$Sepal.Length)/nparts),4))
> # setting the column names of the darray
> colnames(X) <- colnames(iris)[1:4]
> dkm <- hpdkmeans(X,centers=3)    # finding 3 clusters
> dkm
K-means clustering with 3 clusters of sizes 50, 61, 39

Cluster means:
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1     5.006000    3.428000     1.462000    0.246000
2     5.883607    2.740984     4.388525    1.434426
3     6.853846    3.076923     5.715385    2.053846

Within cluster sum of squares by cluster:
[1] 15.15100 38.29082 25.41385
 (between_SS / total_SS =  88.4 %)

Available components:

[1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
[6] "betweenss"    "size"         "iter"
```

Figure 1. example 1 for hpdkmeans

2.3. Example 2

In this example, the samples are read from a CSV file where the first row and the first two columns are labels. Moreover, each sample is stored column-wise. Here we use only 430 samples for clustering. The script is displayed in Figure 2.

A warning message has appeared at the first run of the algorithm which indicates it is not converged in 10 iterations. Although the algorithm is not converged, a set of clusters is created. The algorithm is converged when the calculated centers do not change in two consequent iterations. hpdkmeans function has two arguments in order to increase the quality of the result (similar to kmeans function): iter.max (10 by default) and nstart (1 by default). 'iter.max' specifies the maximum number of iterations allowed. 'nstart' gives the number of times that a random set of centers is chosen and clustering is performed. There is no warning message when we run the algorithm for the second time with iter.max=100 and nstart=10.

2.4. Performance evaluation

The following experiments show the scalability of hpdkmeans function for different numbers of nodes. The test scrip displayed in Figure 3 is used for benchmarking.

Specification of each node for experiments:

- 2 CPU per Node
- Core(s) per socket: 6
- CPU model: Intel® Xeon® X5650 (12MB Cache, 2.67GHz)
- 96GB RAM
- 1Gb/s Ethernet Network
- CentOS release 6.4 (Final)
- R version 3.0.1

```
> library(HPdcluster)     # loading the library
Loading required package: distributedR
Loading required package: Rcpp
Loading required package: RInside
Loading required package: MatrixHelper
> distributedR_start()    # starting the distributed environment
Workers registered - 1/1.
All 1 workers are registered.
[1] TRUE
> (ds <- distributedR_status())   # saving the status
         Workers Inst SysMem MemUsed DarrayQuota DarrayUsed
1 127.0.0.1:9090    7   7804    6950        3511          0
> nparts <- sum(ds$Inst)  # number of available distributed instances
> N<-430  # number of samples
> # path to the file
> fname <- "path/raw.csv"
> f <- read.csv(fname)
> head(f)[1:8]   # displaying the first elements of the file
  Indx  Freq V1    V2    V3     V4     V5    V6
1    0   0Hz  0 0.010 2.291  0.431  0.223 0.972
2    1  86Hz  0 0.010 1.923  4.501  4.452 3.698
3    2 172Hz  0 0.009 1.389  8.845  6.998 3.754
4    3 258Hz  0 0.008 0.968 14.892 11.016 5.032
5    4 345Hz  0 0.007 0.540 13.916 11.348 6.037
6    5 431Hz  0 0.006 0.303  6.134  1.370 1.226
> dim(f)
[1] 128 436
> offset<-3 # The first column of useful data
> # number of rows in each block of the darray
> rowsInBlock<-ceiling(N/nparts)
> # defining the input darray
> A <- darray(dim=c(N,128), blocks=c(rowsInBlock,128))
> # loadind the darray with the input data
> foreach(i, 1:npartitions(A), function(a=splits(A,i), index=i, size=rowsInBlock,
fname=fname, offset=offset) {
+  start<-((index-1)*size)+offset
+  end<-start+nrow(a)-1
+  # reading the file
+  f <- read.csv(fname)
+  a<-as.matrix(f[,start:end],nrow=nrow(f))
+  a<-t(a)
+  update(a)
+ })
progress: 100%
[1] TRUE
> # finding 10 clusters
> mykmL <- hpdkmeans(A, centers=10)
Warning message:
did not converge in 10 iterations
> mykmL$size
 [1] 75 34 19 26 47 99 23 30 63 14
>
> mykmL <- hpdkmeans(A, centers=10, iter.max = 100, nstart = 10)
> mykmL$size
 [1]  18  34  23 149  21  12  23  13  51  86
```

Figure 2. example 2 for hpdkmeans

Strong Scalability

In order to test scalability with respect to increase in the processing resources with fixed input data size, we used the following setting. The measured speedup follows Amdahl's law.

- Number of samples: 10,000,000
- Number of features in each sample: 100
- Number of partitions in darray the same as the total number of instances

A set of experiments was performed on a single node with changing the number of instances from 1 to 23. Table 1 shows the results of these experiments. Figure 4 and Figure 5 also display corresponding charts for running-time and speedup.

Another set of experiments was performed on multiple nodes, from 1 to 16 nodes, once with 12 instances and the other time 23 instances per node. Table 2 shows the results of these experiments. Figure 6 and Figure 7 also display corresponding charts for running-time and speedup.

```
args <- commandArgs(TRUE)
library(HPdcluster)
distributedR_start(cluster_conf=args[1], inst=args[2])
(drs <- distributedR_status())

(N <- 1e7 * as.numeric(args[3]))
(npredictors <- 100)
(nCenters <- 1000)
iniCenter <- matrix(nrow=nCenters, ncol=npredictors, data=runif(nCenters*npredictors, min=-1000,max=1000))
npart <- sum(drs$Inst)
s <- ceiling(N/npart)

X <- darray(dim=c(N, npredictors), blocks=c(s,npredictors), FALSE)
foreach(i, 1:numSplits(X), initArrays <- function(x = splits(X,i), iniCenter=iniCenter) {
   row <- nrow(x)
   col <- ncol(x)
   nCenters <- nrow(iniCenter)
   x <- matrix(nrow=row, ncol=col, data=rnorm(row*col, t(iniCenter[ sample(nCenters,nCenters,replace=TRUE),
]), 1), byrow=TRUE)
   update(x)
})
distributedR_status()

mykm <- hpdkmeans(X,centers = iniCenter, trace=TRUE)
distributedR_status()
distributedR_shutdown()
```

Figure 3. test script for performance evaluation

Table 1. Strong scalability test on a single node, times in second (tt: total time, it: iteration time)

Instance	tt	average (it)	# iteration	Speedup (tt)	Speedup (it)
1	4395.948	2035.573	2	1.00	1.00
2	2462.588	1149.988	2	1.79	1.77
4	927.581	421.979	2	4.74	4.82
8	451.55	201.889	2	9.74	10.08
12	329.573	142.222	2	13.34	14.31
16	381.025	171.865	2	11.54	11.84
20	459.467	211.647	2	9.57	9.62
23	472.233	219.16	2	9.31	9.29

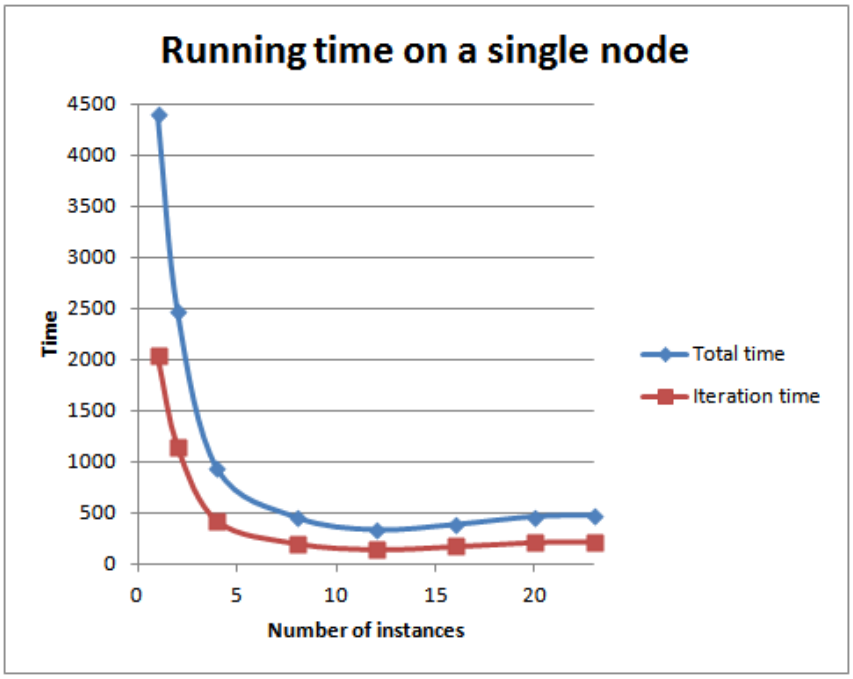

Figure 4. running time on a single node versus increase in the number of instances

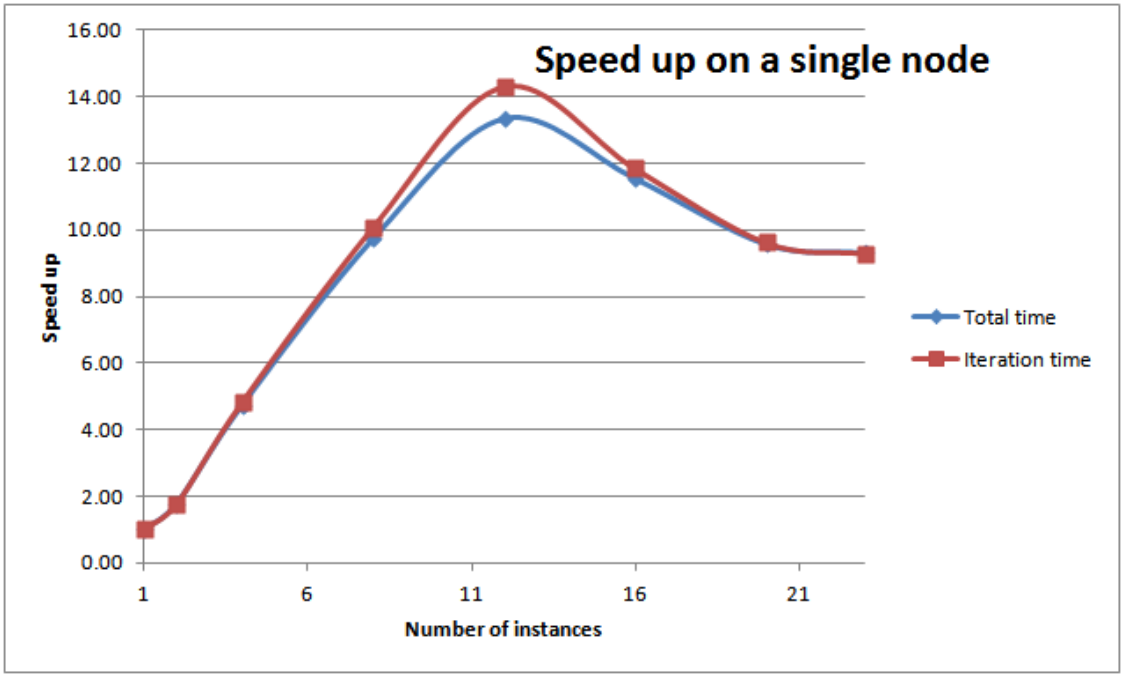

Figure 5. speedup on a single node versus increase in the number of instances

Table 2. Strong scalability test on multiple nodes, times in second (tt: total time, it: iteration time)

Nodes(Instances)	tt	average (it)	# iteration	Speedup (tt)	Speedup (it)
1 (12)	329.573	142.222	2	1.00	1.00
2 (12)	171.331	72.127	2	1.92	1.97
4 (12)	91.251	39.083	2	3.61	3.64
8 (12)	60.054	25.156	2	5.49	5.65
16 (12)	46.168	18.658	2	7.14	7.62
1 (23)	472.233	219.16	2	1.00	1.00
2 (23)	290.753	135.276	2	1.62	1.62
4 (23)	182.6	85.029	2	2.59	2.58
8 (23)	98.862	44.42	2	4.78	4.93
16 (23)	69.548	27.481	2	6.79	7.97

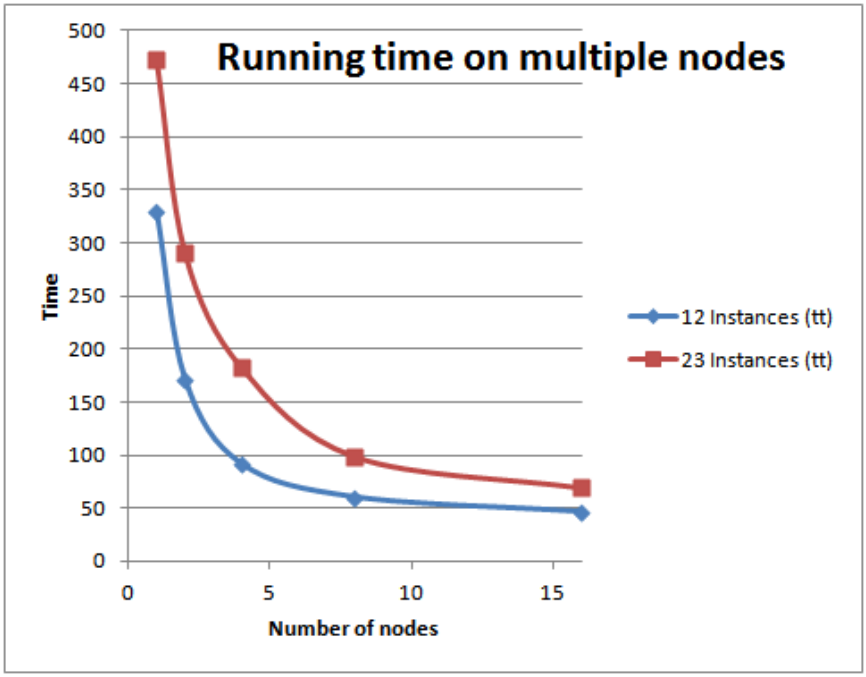

Figure 6. running time on multiple nodes (tt: total time)

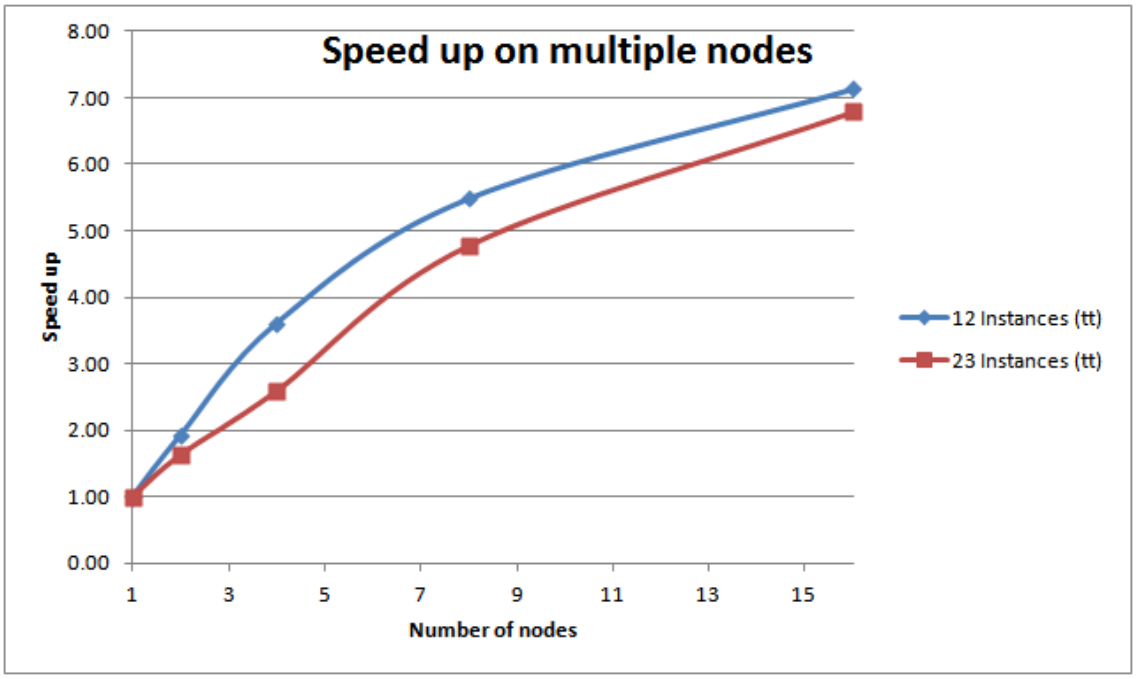

Figure 7. speedup on multiple nodes (tt: total time)

Weak Scalability

In order to test scalability with respect to the same increase rate in the processing resources and the input data size, we used the following setting. The measured speedup follows Gustafson's law.

- Number of samples on a single node: 10,000,000
- Number of features in each sample: 100
- Number of partitions in darray the same as the total number of instances
- Number of instances per node: 12

Table 3 shows the results of these experiments. Figure 8 also displays corresponding chart for speedup.

Table 3. Weak scalability test on multiple nodes, times in second (tt: total time, it: iteration time)

Nodes(Instances)	# samples	tt	average (it)	# iteration	Speedup (tt)	Speedup (it)
1 (12)	10M	329.573	142.222	2	1.00	1.00
2 (12)	2 * 10M	387.929	173.055	2	0.85	0.82
4 (12)	4 * 10M	329.712	143.847	2	1.00	0.99
8 (12)	8 * 10M	414.04	185.361	2	0.80	0.77
16 (12)	16 * 10M	367.797	156.397	2	0.90	0.91

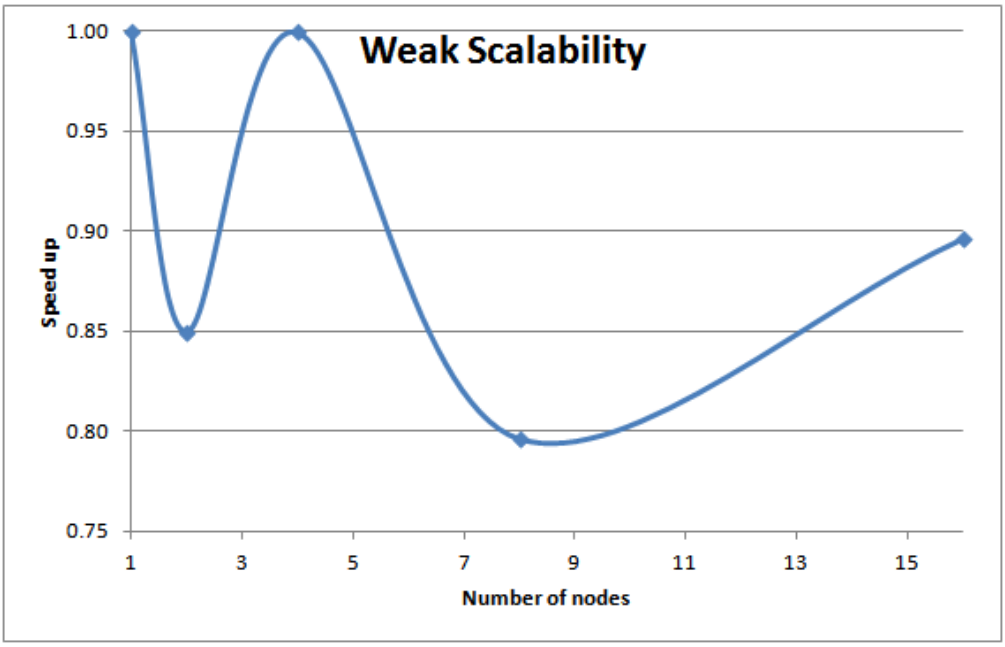

Figure 8. Weak scalability speedup

3. Loader functions

There are two loader functions provided for this package.

3.1. sampleLoader

sampleLoader function is an example for loading a set of samples stored in a table to a darray. It is assumed that samples are stored in a single table, and the table contains a column called 'rowid'. It is also assumed that 'rowid' starts from 0, and there is no missed 'rowid'. The user is encouraged to modify this example for more sophisticated cases.

3.2. centerLoader

centerLoader function is an example for loading a set of centers stored in a table to a matrix. It is assumed that centers are stored in a single table. All the rows of the table will be read, and each row will be a center. The user is encouraged to modify this example for more sophisticated cases.

3.3. Examples

As an example, assume that samples are stored in a table named *mortgage*; while 6 columns should be used as the features for clustering: *mltvspline1*, *mltvspline2*, *agespline1*, *agespline2*, *hpichgspline*, and *ficospline* (see Table 4). Also assume that the name of configuration for the database connection is "*Test*". The command to load all the samples stored in this table is:

```
loadedSamples <- sampleLoader ("mortgage", list("mltvspline1", "mltvspline2",
"agespline1", "agespline2", "hpichgspline", "ficospline"), conf="Test")
```

The darray for samples will become available as `loadedSamples`. Partitions (blocks) of the darray are row-wise and their number is the same as the number of active instances (R-executors). Please note that the number of rows in the table must be more than the number of active instances (R-executors).

Table 4. Example of a table in the database called mortgage

rowid	def	mltvspline1	mltvspline2	agespline1	agespline2	hpichgspline	ficospline
1	1	0.760777	0.006632	0.948052	0.906403	0.058021	0.960328
2	0	0.135741	0.205449	0.516031	0.013455	0.827438	0.659125
3	0	0.021796	0.138996	0.862165	0.034211	0.150524	0.345917
4	1	0.271257	0.543280	0.940978	0.891880	0.993050	0.000160
5	1	0.986207	0.053896	0.119611	0.646744	0.819753	0.663289

```
> loadedSamples <- sampleLoader ("mortgage", list("mltvspline1", "mltvspline2",
"agespline1", "agespline2", "hpichgspline", "ficospline"))
progress: 100%
> getpartition(loadedSamples)
      mltvspline1 mltvspline2 agespline1 agespline2 hpichgspline ficospline
 [1,]    0.760777    0.006632   0.948052   0.906403     0.058021   0.960328
 [2,]    0.135741    0.205449   0.516031   0.013455     0.827438   0.659125
 [3,]    0.021796    0.138996   0.862165   0.034211     0.150524   0.345917
 [4,]    0.271257    0.543280   0.940978   0.891880     0.993050   0.000160
 [5,]    0.986207    0.053896   0.119611   0.646744     0.819753   0.663289
```

Figure 9. An example for sampleLoader

Let's assume that two centers as the initial set of centers for the samples is stored in a table called *centers*. This table should have 6 columns corresponding to different features of the samples. Each row of this table is read as a center; apparently, the number of clusters would be equal to the number of centers. Table and figure show an example for this function.

Table 5. Example of a table in the database called centers

mltvspline1	mltvspline2	agespline1	agespline2	hpichgspline	ficospline
0.76	0.006	0.948	0.906	0.058	0.960
0.135	0.205	0.516	0.013	0.827	0.659

```
> loadedCenters <- centerLoader ("centers", list("mltvspline1", "mltvspline2",
"agespline1", "agespline2", "hpichgspline", "ficospline"))
> loadedCenters
      mltvspline1 mltvspline2 agespline1 agespline2 hpichgspline ficospline
 [1,]    0.76        0.006      0.948      0.906        0.058      0.960328
 [2,]    0.135       0.205      0.516      0.013        0.827      0.659125
```

Figure 10. An example for centerLoader