Package: GMKMcharlie (via r-universe)

August 31, 2024

Type Package

Title Unsupervised Gaussian Mixture and Minkowski and Spherical K-Means with Constraints

Version 1.1.5

Author Charlie Wusuo Liu

Maintainer Charlie Wusuo Liu <liuwusuo@gmail.com>

Description High performance trainers for parameterizing and clustering weighted data. The Gaussian mixture (GM) module includes the conventional EM (expectation maximization) trainer, the component-wise EM trainer, the minimum-message-length EM trainer by Figueiredo and Jain (2002) <doi:10.1109/34.990138>. These trainers accept additional constraints on mixture weights, covariance eigen ratios and on which mixture components are subject to update. The K-means (KM) module offers clustering with the options of (i) deterministic and stochastic K-means++ initializations, (ii) upper bounds on cluster weights (sizes), (iii) Minkowski distances, (iv) cosine dissimilarity, (v) dense and sparse representation of data input. The package improved the typical implementations of GM and KM algorithms in various aspects. It is carefully crafted in multithreaded C++ for modeling large data for industry use.

License GPL-3

Encoding UTF-8

Imports Rcpp (>= 1.0.0), RcppParallel

Suggests MASS (>= 7.3.0), plot3D (>= 1.1.1)

LinkingTo Rcpp, RcppParallel, RcppArmadillo

SystemRequirements GNU make

NeedsCompilation yes

Date/Publication 2021-05-29 06:20:02 UTC

Repository https://whateverliu.r-universe.dev

RemoteUrl https://github.com/cran/GMKMcharlie

RemoteRef HEAD

RemoteSha ee895b144744daef788e6889c13ef57d6863281a

Contents

`d2s` *Dense to sparse conversion*

Description

Convert data from dense representation (matrix) to sparse representation (list of data frames).

Usage

```
d2s(
  X,
  zero = 0,
  threshold = 1e-16,
  verbose= TRUE
  )
```

Arguments

`X`	A `d` x `N` numeric matrix where `N` is the number of data points — each column is an observation, and `d` is the dimensionality. Column-observation representation promotes cache locality.
`zero`	A numeric value. Elements in `X` satisfying `abs(X[i] - zero) <= threshold` are treated as zeros. Default 0.
`threshold`	A numeric value, explained above.
`verbose`	A boolean value. `TRUE` prints progress.

Value

A list of size N. Value[[i]] is a 2-column data frame. The 1st column is a sorted integer vector of the indexes of nonzero dimensions. Values in these dimensions are stored in the 2nd column as a numeric vector.

Examples

```
N = 2000L
d = 3000L
X = matrix(rnorm(N * d) + 2, nrow = d)
# Fill many zeros in X:
X = apply(X, 2, function(x) {
  x[sort(sample(d, d * runif(1, 0.95, 0.99)))] = 0; x})
# Get the sparse version of X.
sparseX = GMKMcharlie::d2s(X)
str(sparseX[1:5])
```

GM *Multithreaded Gaussian mixture trainer*

Description

The traditional training algorithm via maximum likelihood for parameterizing weighted data with a mixture of Gaussian PDFs. Bounds on covariance matrix eigen ratios and mixture weights are optional.

Usage

```
GM(
  X,
  Xw = rep(1, ncol(X)),
  alpha = numeric(0),
  mu = matrix(ncol = 0, nrow = 0),
  sigma = matrix(ncol = 0, nrow = 0),
  G = 5L,
  convergenceEPS = 1e-05,
  convergenceTail = 10,
  alphaEPS = 0,
  eigenRatioLim = Inf,
  embedNoise = 1e-6,
  maxIter = 1000L,
  maxCore = 7L,
  tlimit = 3600,
  verbose = TRUE,
  updateAlpha = TRUE,
  updateMean = TRUE,
  updateSigma = TRUE,
  checkInitialization = FALSE,
```

```
  KmeansFirst = TRUE,
  KmeansPPfirst = FALSE,
  KmeansRandomSeed = NULL,
  friendlyOutput = TRUE
  )
```

Arguments

X	A d x N numeric matrix where N is the number of observations — each column is an observation, and d is the dimensionality. Column-observation representation promotes cache locality.
Xw	A numeric vector of size N. Xw[i] is the weight on observation X[, i]. Users should normalize Xw such that the elements sum up to N. Default uniform weights for all observations.
alpha	A numeric vector of size K, the number of Gaussian kernels in the mixture model. alpha are the initial mixture weights and should sum up to 1. Default empty.
mu	A d x K numeric matrix. mu[, i] is the initial mean for the ith Gaussian kernel. Default empty.
sigma	Either a list of d x d matrices, or a d^2 x K numeric matrix. For the latter, each column represents a flattened d x d initial covariance matrix of the ith Gaussian kernel. In R, as.numeric(aMatrix) gives the flattened version of aMatrix. Covariance matrix of each Gaussian kernel MUST be positive-definite. Default empty.
G	An integer. If at least one of the parameters alpha, mu, sigma are empty, the program will initialize G Gaussian kernels via K-means++ deterministic initialization. See KMppIni(). Otherwise G is ignored. Default 5.
convergenceEPS	A numeric value. If the average change of all parameters in the mixture model is below convergenceEPS relative to those in the pervious iteration, the program ends. Checking convergence this way is faster than recomputing the log-likelihood every iteration. Default 1e-5.
convergenceTail	If every one of the last convergenceTail iteration produces less than a relative increase of convergenceEPS in log-likelihood, stop.
alphaEPS	A numeric value. During training, if any Gaussian kernel's weight is no greater than alphaEPS, the kernel is deleted. Default 0.
eigenRatioLim	A numeric value. During training, if any Gaussian kernel's max:min eigen value ratio exceeds eigenRatioLim, the kernel is treated as degenerate and deleted. Thresholding eigen ratios is in the interest of minimizing the effect of degenerate kernels in an early stage. Default Inf.
embedNoise	A small constant added to the diagonal entries of all covariance matrices. This may prevent covariance matrices collapsing prematurely. A suggested value is 1e-6. Covariance degeneration is detected during Cholesky decomposition, and will lead the trainer to remove the corresponding mixture component. For high-dimensional problem, setting embedNoise to nonzero may pose the illusion of massive log-likelihood, all because one or more mixture components are so close to singular, which makes the densities around them extremely high.

`maxIter` An integer, the maximal number of iterations.

`maxCore` An integer. The maximal number of threads to invoke. Should be no more than the total number of logical processors on machine. Default 7.

`tlimit` A numeric value. The program exits with the current model in `tlimit` seconds.

`verbose` A boolean value. `TRUE` prints progress.

`updateAlpha` A boolean value or boolean vector. If a boolean value, `TRUE` implies weights on all mixture components are subject to update, otherwise they should stay unchanged during training. If a boolean vector, its size should equal the number of mixture components. `updateAlpha[i] == TRUE` implies the weight on the `i`th component is subject to update. Regardless of `updateAlpha`, the output will have normalized mixture weights.

`updateMean` A boolean value or a boolean vector. If a boolean value, `TRUE` implies means of all mixture components are subject to update, otherwise they should stay unchanged during training. If a boolean vector, its size should equal the number of mixture components. `updateMean[i] == TRUE` implies the mean of the `i`th component is subject to update.

`updateSigma` A boolean value or a boolean vector. If a boolean value, `TRUE` implies covariances of all mixture components are subject to update, otherwise they should stay unchanged during training. If a boolean vector, its size should equal the number of mixture components. `updateSigma[i] == TRUE` implies the covariance of the `i`th component is subject to update.

`checkInitialization` Check if any of the input covariance matrices are singular.

`KmeansFirst` A boolean value. Run K-means clustering for finding means.

`KmeansPPfirst` A boolean value. Run stochastic K-means++ for K-means initialization.

`KmeansRandomSeed` An integer or `NULL`, the random seed for K-means++.

`friendlyOutput` `TRUE` returns covariance matrices in a list rather than a single matrix of columns of flattened covariance matrices.

Details

An in-place Cholesky decomposition of covariance matrix is implemented for space and speed efficiency. Only the upper triangle of the Cholesky decomposition is memorized for each Gaussian kernel, and it is then applied to a forward substitution routine for fast Mahalanobis distances computation. Each of the three main steps in an iteration — Gaussian density computation, parameter inference, parameter update — is multithreaded and hand-scheduled using Intel TBB. Extensive efforts have been made over cache-friendliness and extra multithreading overheads such as memory allocation.

If `eigenRatioLim` is finite, eigen decomposition employs routines in `RcppArmadillo`.

Value

A list of size 5:

`alpha` a numeric vector of size `K`. The mixture weights.

mu	a d x K numeric matrix. Each column is the mean of a Gaussian kernel.
sigma	a d^2 x K numeric matrix. Each column is the flattened covariance matrix of a Gaussian kernel. Do matrix(sigma[, i], nrow = d) to recover the covariance matrix of the ith kernel.
fitted	a numeric vector of size N. fitted[i] is the probability density of the ith observation given by the mixture model.
clusterMember	a list of K integer vectors, the hard clustering inferred from the mixture model. Each integer vector contains the indexes of observations in X.

Warning

For one-dimensional data, X should still follow the data structure requirements: a matrix where each column is an observation.

Examples

```
# ============================================================================
# Examples below use 1 thread to pass CRAN check. Speed advantage of multiple
# threads will be more pronounced for larger data.
# ============================================================================


# ============================================================================
# Parameterize the iris data. Let the function initialize Gaussian kernels.
# ============================================================================
X = t(iris[1:4])
# CRAN check only allows 2 threads at most. Increase `maxCore` for
# acceleration.
gmmRst = GMKMcharlie::GM(X, G = 4L, maxCore = 1L, friendlyOutput = FALSE)
str(gmmRst)




# ============================================================================
# Parameterize the iris data given Gaussian kernels.
# ============================================================================
G = 3L
d = nrow(X) # Dimensionality.
alpha = rep(1, G) / G
mu = X[, sample(ncol(X), G)] # Sample observations as initial means.
# Take the average variance and create initial covariance matrices.
meanVarOfEachDim = sum(diag(var(t(X)))) / d
covar = diag(meanVarOfEachDim / G, d)
covars = matrix(rep(as.numeric(covar), G), nrow = d * d)


# Models are sensitive to initialization.
gmmRst2 = GMKMcharlie::GM(
  X, alpha = alpha, mu = mu, sigma = covars, maxCore = 1L,
  friendlyOutput = FALSE)
```

```
str(gmmRst2)



# ============================================================================
# For fun, fit Rosenbrock function with a Gaussian mixture.
# ============================================================================
set.seed(123)
rosenbrock <- function(x, y) {(1 - x) ^ 2 + 100 * (y - x ^ 2) ^ 2}
N = 2000L
x = runif(N, -2, 2)
y = runif(N, -1, 3)
z = rosenbrock(x, y)


X = rbind(x, y)
Xw = z * (N / sum(z)) # Weights on observations should sum up to N.
gmmFit = GMKMcharlie::GM(X, Xw = Xw, G = 5L, maxCore = 1L, verbose = FALSE,
  friendlyOutput = FALSE)


oldpar = par()$mfrow
par(mfrow = c(1, 2))
plot3D::points3D(x, y, z, pch = 20)
plot3D::points3D(x, y, gmmFit$fitted, pch = 20)
par(mfrow = oldpar)



# ============================================================================
# For fun, fit a 3D spiral distribution.
# ============================================================================
N = 2000
t = runif(N) ^ 2 * 15
x = cos(t) + rnorm(N) * 0.1
y = sin(t) + rnorm(N) * 0.1
z = t + rnorm(N) * 0.1


X = rbind(x, y, z)
d = 3L
G = 10L
gmmFit = GMKMcharlie::GM(X, G = G, maxCore = 1L, verbose = FALSE,
  KmeansFirst = TRUE, KmeansPPfirst = TRUE, KmeansRandomSeed = 42,
  friendlyOutput = TRUE)
# Sample N points from the Gaussian mixture.
ns = as.integer(round(N * gmmFit$alpha))
sampledPoints = list()
for(i in 1:G)
{
  sampledPoints[[i]] = MASS::mvrnorm(
```

```
    ns[i], mu = gmmFit$mu[, i], Sigma = matrix(gmmFit$sigma[[i]], nrow = d))
}
sampledPoints =
  matrix(unlist(lapply(sampledPoints, function(x) t(x))), nrow = d)


# Plot the original data and the samples from the mixture model.
oldpar = par()$mfrow
par(mfrow = c(1, 2))
plot3D::points3D(x, y, z, pch = 20)
plot3D::points3D(x = sampledPoints[1, ],
                 y = sampledPoints[2, ],
                 z = sampledPoints[3, ], pch = 20)
par(mfrow = oldpar)




# ============================================================================
# For fun, fit a 3D spiral distribution. Fix parameters at random.
# ============================================================================
N = 2000
t = runif(N) ^ 2 * 15
x = cos(t) + rnorm(N) * 0.1
y = sin(t) + rnorm(N) * 0.1
z = t + rnorm(N) * 0.1


X = rbind(x, y, z); dimnames(X) = NULL
d = 3L
G = 10L
mu = X[, sample(ncol(X), G)]
s = matrix(rep(as.numeric(cov(t(X))), G), ncol = G)
alpha = rep(1 / G, G)
updateAlpha = sample(c(TRUE, FALSE), G, replace = TRUE)
updateMean = sample(c(TRUE, FALSE), G, replace = TRUE)
updateSigma = sample(c(TRUE, FALSE), G, replace = TRUE)
gmmFit = GMKMcharlie::GM(X, alpha = alpha, mu = mu, sigma = s, G = G,
                         maxCore = 2, verbose = FALSE,
                         updateAlpha = updateAlpha,
                         updateMean = updateMean,
                         updateSigma = updateSigma,
                         convergenceEPS = 1e-5, alphaEPS = 1e-8,
                         friendlyOutput = TRUE)
```

GMcw *Multithreaded component-wise Gaussian mixture trainer*

Description

The component-wise variant of GM().

Usage

```
GMcw(
  X,
  Xw = rep(1, ncol(X)),
  alpha = numeric(0),
  mu = matrix(ncol = 0, nrow = 0),
  sigma = matrix(ncol = 0, nrow = 0),
  G = 5L,
  convergenceEPS = 1e-05,
  alphaEPS = 0,
  eigenRatioLim = Inf,
  maxIter = 1000L,
  maxCore = 7L,
  tlimit = 3600,
  verbose = TRUE
  )
```

Arguments

X	A d x N numeric matrix where N is the number of observations — each column is an observation, and d is the dimensionality. Column-observation representation promotes cache locality.
Xw	A numeric vector of size N. Xw[i] is the weight on observation X[, i]. Users should normalize Xw such that the elements sum up to N. Default uniform weights for all observations.
alpha	A numeric vector of size K, the number of Gaussian kernels in the mixture model. alpha are the initial mixture weights and should sum up to 1. Default empty.
mu	A d x K numeric matrix. mu[, i] is the initial mean for the ith Gaussian kernel. Default empty matrix.
sigma	A d^2 x K numeric matrix. Each column represents a flattened d x d initial covariance matrix of the ith Gaussian kernel. In R, as.numeric(aMatrix) gives the flattened version of aMatrix. Covariance matrix of each Gaussian kernel MUST be positive-definite. Default empty.
G	An integer. If at least one of the parameters alpha, mu, sigma are empty, the program will initialize G Gaussian kernels via K-means++ deterministic initialization. See KMppIni(). Otherwise G is ignored. Default 5.
convergenceEPS	A numeric value. If the average change of all parameters in the mixture model is below convergenceEPS relative to those in the pervious iteration, the program ends. Checking convergence this way is faster than recomputing the log-likelihood every iteration. Default 1e-5.
alphaEPS	A numeric value. During training, if any Gaussian kernel's weight is no greater than alphaEPS, the kernel is deleted. Default 0.
eigenRatioLim	A numeric value. During training, if any Gaussian kernel's max:min eigen value ratio exceeds eigenRatioLim, the kernel is treated as degenerate and deleted. Thresholding eigen ratios is in the interest of minimizing the effect of degenerate kernels in an early stage. Default Inf.

maxIter	An integer, the maximal number of iterations.
maxCore	An integer. The maximal number of threads to invoke. Should be no more than the total number of logical processors on machine. Default 7.
tlimit	A numeric value. The program exits with the current model in tlimit seconds.
verbose	A boolean value. TRUE prints progress.

Details

Relevant details can be found in GM(). In GMcw(), an update of any Gaussian kernel triggers the update of the underlying weighing matrix that directs the update of all Gaussian kernels. Only after that the next Gaussian kernel is updated. See references.

In the actual implementation, the N x K weighing matrix WEI does not exist in memory. An N x K density matrix DEN instead stores each Gaussian kernel's probability density at every observation in X. Mathematically, the ith column of WEI equals DEN's ith column divided by the row sum RS. RS is a vector of size N and is memorized and updated responding to the update of each Gaussian kernel: before updating the ith kernel, the algorithm subtracts the ith column of DEN from RS; after the kernel is updated and the probability densities are recomputed, the algorithm adds back the ith column of DEN to RS. Now, to update the i+1th Gaussian kernel, we can divide the i+1th column of DEN by RS to get the weighing coefficients.

The above implementation makes the component-wise trainer comparable to the classic one in terms of speed. The component-wise trainer is a key component in Figuredo & jain's MML (minimum message length) mixture model trainer to avoid premature deaths of the Gaussian kernels.

Value

A list of size 5:

alpha	a numeric vector of size K. The mixture weights.
mu	a d x K numeric matrix. Each column is the mean of a Gaussian kernel.
sigma	a d^2 x K numeric matrix. Each column is the flattened covariance matrix of a Gaussian kernel. Do matrix(sigma[, i], nrow = d) to recover the covariance matrix of the ith kernel.
fitted	a numeric vector of size N. fitted[i] is the probability density of the ith observation given by the mixture model.
clusterMember	a list of K integer vectors, the hard clustering inferred from the mixture model. Each integer vector contains the indexes of observations in X.

Warning

For one-dimensional data, X should still follow the data structure requirements: a matrix where each column is an observation.

References

Celeux, Gilles, et al. "A Component-Wise EM Algorithm for Mixtures." Journal of Computational and Graphical Statistics, vol. 10, no. 4, 2001, pp. 697-712. JSTOR, www.jstor.org/stable/1390967.

Examples

```
# ============================================================================
# Examples below use 1 thread to pass CRAN check. Speed advantage of multiple
# threads will be more pronounced for larger data.
# ============================================================================


# ============================================================================
# Parameterize the iris data. Let the function initialize Gaussian kernels.
# ============================================================================
X = t(iris[1:4])
# CRAN check only allows 2 threads at most. Increase `maxCore` for
# acceleration.
gmmRst = GMKMcharlie::GMcw(X, G = 3L, maxCore = 1L)
str(gmmRst)




# ============================================================================
# Parameterize the iris data given Gaussian kernels.
# ============================================================================
G = 3L
d = nrow(X) # Dimensionality.
alpha = rep(1, G) / G
mu = X[, sample(ncol(X), G)] # Sample observations as initial means.
# Take the average variance and create initial covariance matrices.
meanVarOfEachDim = sum(diag(var(t(X)))) / d
covar = diag(meanVarOfEachDim / G, d)
covars = matrix(rep(as.numeric(covar), G), nrow = d * d)


# Models could be different given a different initialization.
gmmRst2 = GMKMcharlie::GMcw(
  X, alpha = alpha, mu = mu, sigma = covars, maxCore = 1L)
str(gmmRst2)




# ============================================================================
# For fun, fit Rosenbrock function with a Gaussian mixture.
# ============================================================================
set.seed(123)
rosenbrock <- function(x, y) {(1 - x) ^ 2 + 100 * (y - x ^ 2) ^ 2}
N = 2000L
x = runif(N, -2, 2)
y = runif(N, -1, 3)
z = rosenbrock(x, y)


X = rbind(x, y)
```

```
Xw = z * (N / sum(z)) # Weights on observations should sum up to N.
gmmFit = GMKMcharlie::GMcw(X, Xw = Xw, G = 5L, maxCore = 1L, verbose = FALSE)


oldpar = par()$mfrow
par(mfrow = c(1, 2))
plot3D::points3D(x, y, z, pch = 20)
plot3D::points3D(x, y, gmmFit$fitted, pch = 20)
par(mfrow = oldpar)




# ============================================================================
# For fun, fit a 3D spiral distribution.
# ============================================================================
N = 2000
t = runif(N) ^ 2 * 15
x = cos(t) + rnorm(N) * 0.1
y = sin(t) + rnorm(N) * 0.1
z = t + rnorm(N) * 0.1


X = rbind(x, y, z)
d = 3L
G = 10L
gmmFit = GMKMcharlie::GMcw(X, G = G, maxCore = 1L, verbose = FALSE)
# Sample N points from the Gaussian mixture.
ns = as.integer(round(N * gmmFit$alpha))
sampledPoints = list()
for(i in 1L : G)
{
  sampledPoints[[i]] = MASS::mvrnorm(
    ns[i], mu = gmmFit$mu[, i], Sigma = matrix(gmmFit$sigma[, i], nrow = d))
}
sampledPoints =
  matrix(unlist(lapply(sampledPoints, function(x) t(x))), nrow = d)


# Plot the original data and the samples from the mixture model.
oldpar = par()$mfrow
par(mfrow = c(1, 2))
plot3D::points3D(x, y, z, pch = 20)
plot3D::points3D(x = sampledPoints[1, ],
                 y = sampledPoints[2, ],
                 z = sampledPoints[3, ],
                 pch = 20)
par(mfrow = oldpar)
```

GMfj *Multithreaded minimum message length Gaussian mixture trainer*

Description

Figueiredo and Jain's Gaussian mixture trainer with all options in `GM()`.

Usage

```
GMfj(
  X,
  Xw = rep(1, ncol(X)),
  alpha = numeric(0),
  mu = matrix(ncol = 0, nrow = 0),
  sigma = matrix(ncol = 0, nrow = 0),
  G = 5L,
  Gmin = 2L,
  convergenceEPS = 1e-05,
  alphaEPS = 0,
  eigenRatioLim = Inf,
  maxIter = 1000L,
  maxCore = 7L,
  tlimit = 3600,
  verbose = TRUE
  )
```

Arguments

`X`	A `d` x `N` numeric matrix where `N` is the number of observations — each column is an observation, and `d` is the dimensionality. Column-observation representation promotes cache locality.
`Xw`	A numeric vector of size `N`. `Xw[i]` is the weight on observation `X[, i]`. Users should normalize `Xw` such that the elements sum up to `N`. Default uniform weights for all observations.
`alpha`	A numeric vector of size `K`, the number of Gaussian kernels in the mixture model. `alpha` are the initial mixture weights and should sum up to 1. Default empty.
`mu`	A `d` x `K` numeric matrix. `mu[, i]` is the initial mean for the `i`th Gaussian kernel. Default empty.
`sigma`	A `d^2` x `K` numeric matrix. Each column represents a flattened `d` x `d` initial covariance matrix of the `i`th Gaussian kernel. In R, `as.numeric(aMatrix)` gives the flattened version of `aMatrix`. Covariance matrix of each Gaussian kernel MUST be positive-definite. Default empty.
`G`	An integer. If at least one of the parameters `alpha`, `mu`, `sigma` are empty, the program will initialize `G` Gaussian kernels via K-means++ deterministic initialization. See `KMppIni()`. Otherwise `G` is ignored. Default 5.
`Gmin`	An integer. The final model should have at least `Gmin` kernels.
`convergenceEPS`	A numeric value. If the average change of all parameters in the mixture model is below `convergenceEPS` relative to those in the pervious iteration, the program ends. Checking convergence this way is faster than recomputing the log-likelihood every iteration. Default 1e-5.

alphaEPS	A numeric value. During training, if any Gaussian kernel's weight is no greater than alphaEPS, the kernel is deleted. Default 0.
eigenRatioLim	A numeric value. During training, if any Gaussian kernel's max:min eigen value ratio exceeds eigenRatioLim, the kernel is treated as degenerate and deleted. Thresholding eigen ratios is in the interest of minimizing the effect of degenerate kernels in an early stage. Default Inf.
maxIter	An integer, the maximal number of iterations.
maxCore	An integer. The maximal number of threads to invoke. Should be no more than the total number of logical processors on machine. Default 7.
tlimit	A numeric value. The program exits with the current model in tlimit seconds.
verbose	A boolean value. TRUE prints progress.

Details

Although heavily cited, the paper has some misleading information and the algorithm's performance does not live up to its reputation. See <https://stats.stackexchange.com/questions/423935/figueiredo-and-jains-gaussian-mixture-em-convergence-criterion>. Nevertheless, it is a worthwhile algorithm to try in practice.

Value

A list of size 5:

alpha	a numeric vector of size K. The mixture weights.
mu	a d x K numeric matrix. Each column is the mean of a Gaussian kernel.
sigma	a d^2 x K numeric matrix. Each column is the flattened covariance matrix of a Gaussian kernel. Do matrix(sigma[, i], nrow = d) to recover the covariance matrix of the ith kernel.
fitted	a numeric vector of size N. fitted[i] is the probability density of the ith observation given by the mixture model.
clusterMember	a list of K integer vectors, the hard clustering inferred from the mixture model. Each integer vector contains the indexes of observations in X.

Warning

For one-dimensional data, X should still follow the data structure requirements: a matrix where each column is an observation.

References

Mario A.T. Figueiredo & Anil K. Jain (2002): "Unsupervised learning of finite mixture models." IEEE Transactions on Pattern Analysis and Machine Intelligence 24(3): 381-396.

Examples

```
# ============================================================================
# Examples below use 1 thread to pass CRAN check. Speed advantage of multiple
# threads will be more pronounced for larger data.
# ============================================================================


# ============================================================================
# Parameterize the iris data. Let the function initialize Gaussian kernels.
# ============================================================================
X = t(iris[1:4])
# CRAN check only allows 2 threads at most. Increase `maxCore` for
# acceleration.
system.time({gmmRst = GMKMcharlie::GMfj(
  X, G = 25L, Gmin = 2L, maxCore = 1L, verbose = FALSE)})
str(gmmRst)




# ============================================================================
# Parameterize the iris data given Gaussian kernels.
# ============================================================================
G = 25L
d = nrow(X) # Dimensionality.
alpha = rep(1, G) / G
mu = X[, sample(ncol(X), G)] # Sample observations as initial means.
# Take the average variance and create initial covariance matrices.
meanVarOfEachDim = sum(diag(var(t(X)))) / d
covar = diag(meanVarOfEachDim / G, d)
covars = matrix(rep(as.numeric(covar), G), nrow = d * d)


# Models are sensitive to initialization.
system.time({gmmRst2 = GMKMcharlie::GMfj(
  X, alpha = alpha, mu = mu, sigma = covars, maxCore = 1L, verbose = FALSE)})
str(gmmRst2)




# ============================================================================
# For fun, fit Rosenbrock function with a Gaussian mixture.
# ============================================================================
set.seed(123)
rosenbrock <- function(x, y) {(1 - x) ^ 2 + 100 * (y - x ^ 2) ^ 2}
N = 2000L
x = runif(N, -2, 2)
y = runif(N, -1, 3)
z = rosenbrock(x, y)
```

```
X = rbind(x, y)
Xw = z * (N / sum(z)) # Weights on observations should sum up to N.
system.time({gmmFit = GMKMcharlie::GMfj(
  X, Xw = Xw, G = 5L, maxCore = 1L, verbose = FALSE)})


oldpar = par()$mfrow
par(mfrow = c(1, 2))
plot3D::points3D(x, y, z, pch = 20)
plot3D::points3D(x, y, gmmFit$fitted, pch = 20)
par(mfrow = oldpar)




# ============================================================================
# For fun, fit a 3D spiral distribution.
# ============================================================================
N = 2000
t = runif(N) ^ 2 * 15
x = cos(t) + rnorm(N) * 0.1
y = sin(t) + rnorm(N) * 0.1
z = t + rnorm(N) * 0.1


X = rbind(x, y, z)
d = 3L
G = 10L
system.time({gmmFit = GMKMcharlie::GMfj(
  X, G = G, maxCore = 1L, verbose = FALSE)})
# Sample N points from the Gaussian mixture.
ns = as.integer(round(N * gmmFit$alpha))
sampledPoints = list()
for(i in 1L : G)
{
  sampledPoints[[i]] = MASS::mvrnorm(
    ns[i], mu = gmmFit$mu[, i], Sigma = matrix(gmmFit$sigma[, i], nrow = d))
}
sampledPoints =
  matrix(unlist(lapply(sampledPoints, function(x) t(x))), nrow = d)


# Plot the original data and the samples from the mixture model.
oldpar = par()$mfrow
par(mfrow = c(1, 2))
plot3D::points3D(x, y, z, pch = 20)
plot3D::points3D(x = sampledPoints[1, ],
                 y = sampledPoints[2, ],
                 z = sampledPoints[3, ], pch = 20)
par(mfrow = oldpar)
```

KM *K-means over dense representation of data*

Description

Multithreaded weighted Minkowski and spherical K-means via Lloyd's algorithm over dense representation of data.

Usage

```
KM(
  X,
  centroid,
  Xw = rep(1, ncol(X)),
  minkP = 2,
  maxIter = 100L,
  maxCore = 7L,
  verbose = TRUE
  )
```

Arguments

X	A d x N numeric matrix where N is the number of data points — each column is an observation, and d is the dimensionality. Column-observation representation promotes cache locality.
centroid	A d x K numeric matrix where K is the number of clusters. Each column represents a cluster center.
Xw	A numeric vector of size N. Xw[i] is the weight on observation X[, i]. Users should normalize Xw such that the elements sum up to N. Default uniform weights for all observations.
minkP	A numeric value or a character string. If numeric, minkP is the power p in the definition of Minkowski distance. If character string, "max" implies Chebyshev distance, "cosine" implies cosine dissimilarity. Default 2.
maxIter	An integer. The maximal number of iterations. Default 100.
maxCore	An integer. The maximal number of threads to invoke. No more than the total number of logical processors on machine. Default 7.
verbose	A boolean value. TRUE prints progress.

Details

Implementation highlights include:

(i) In Minkowski distance calculation, integer power no greater than 30 uses multiplications. Fractional powers or powers above 30 call std::pow().

(ii) Multithreaded observation-centroid distance calculations. Distances are memorized to avoid unnecessary recomputations if centroids did not change in the last iteration.

(iii) A lookup table is built for storing observation - centroid ID pairs during the assignment step. Observation IDs are then grouped by centroid IDs which allows parallel computing cluster means.

(iv) Function allows non-uniform weights on observations.

(v) Meta-template programming trims branches over different distance functions and other computing methods during compile time.

Value

A list of size `K`, the number of clusters. Each element is a list of 3 vectors:

`centroid` a numeric vector of size `d`.

`clusterMember` an integer vector of indexes of elements grouped to `centroid`.

`member2centroidDistance`
a numeric vector of the same size of `clusterMember`. The `i`th element is the Minkowski distance or cosine dissimilarity from `centroid` to the `clusterMember[i]`th observation in `X`.

Empty `clusterMember` implies empty cluster.

Note

Although rarely happens, divergence of K-means with non-Euclidean distance `minkP != 2` measure is still a theoretical possibility.

Examples

```
# ============================================================================
# Play random numbers. See speed.
# ============================================================================
N = 5000L # Number of points.
d = 500L # Dimensionality.
K = 50L # Number of clusters.
dat = matrix(rnorm(N * d) + runif(N * d), nrow = d)


# Use kmeans++ initialization.
centroidInd = GMKMcharlie::KMppIni(
  X = dat, K, firstSelection = 1L, minkP = 2, stochastic = FALSE,
  seed = sample(1e9L, 1), maxCore = 2L, verbose = TRUE)


centroid = dat[, centroidInd]


# Euclidean.
system.time({rst = GMKMcharlie::KM(
  X = dat, centroid = centroid, maxIter = 100,
  minkP = 2, maxCore = 2, verbose = TRUE)})


# Cosine dissimilarity.
```

```
dat = apply(dat, 2, function(x) x / sum(x ^ 2) ^ 0.5)
centroid = dat[, centroidInd]
system.time({rst2 = GMKMcharlie::KM(
  X = dat, centroid = centroid, maxIter = 100,
  minkP = "cosine", maxCore = 2, verbose = TRUE)})


# ============================================================================
# Test against R's inbuilt km()
# ============================================================================
dat = t(iris[1:4])
dimnames(dat) = NULL


# Use kmeans++ initialization.
centroidInd = GMKMcharlie::KMppIni(
  X = dat, K = 3, firstSelection = 1L, minkP = 2, stochastic = FALSE,
  seed = sample(1e9L, 1), maxCore = 2L, verbose = TRUE)
centroid = dat[, centroidInd]


rst = GMKMcharlie::KM(X = dat, centroid = centroid, maxIter = 100,
                      minkP = 2, maxCore = 2, verbose = TRUE)
rst = lapply(rst, function(x) sort(x$clusterMember))


rst2 = kmeans(x = t(dat), centers = t(centroid), algorithm = "Lloyd")
rst2 = aggregate(list(1L : length(rst2$cluster)),
                 list(rst2$cluster), function(x) sort(x))[[2]]


setdiff(rst, rst2)
```

KMconstrained *K-means over dense data input with constraints on cluster weights*

Description

Multithreaded weighted Minkowski and spherical K-means via Lloyd's algorithm over dense representation of data given cluster size (weight) constraints.

Usage

```
KMconstrained(
  X,
  centroid,
  Xw = rep(1, ncol(X)),
  clusterWeightUB = rep(ncol(X) + 1, ncol(centroid)),
  minkP = 2,
  convergenceTail = 5L,
```

```
  tailConvergedRelaErr = 1e-04,
  maxIter = 100L,
  maxCore = 7L,
  paraSortInplaceMerge = FALSE,
  verbose = TRUE
  )
```

Arguments

X	A d x N numeric matrix where N is the number of data points — each column is an observation, and d is the dimensionality. Column-observation representation promotes cache locality.
centroid	A d x K numeric matrix where K is the number of clusters. Each column represents a cluster center.
Xw	A numeric vector of size N. Xw[i] is the weight on observation X[, i]. Users should normalize Xw such that the elements sum up to N. Default uniform weights for all observations.
clusterWeightUB	An integer vector of size K. The upper bound of weight for each cluster. If Xw are all 1, clusterWeightUB upper-bound cluster sizes.
minkP	A numeric value or a character string. If numeric, minkP is the power p in the definition of Minkowski distance. If character string, "max" implies Chebyshev distance, "cosine" implies cosine dissimilarity. Default 2.
convergenceTail	An integer. The algorithm may end up with "cyclical convergence" due to the size / weight constraints, that is, every few iterations produce the same clustering. If the cost (total in-cluster distance) of each of the last convergenceTail iterations has a relative difference less than tailConvergedRelaErr against the cost from the prior iteration, the program stops.
tailConvergedRelaErr	A numeric value, explained in convergenceTail.
maxIter	An integer. The maximal number of iterations. Default 100.
maxCore	An integer. The maximal number of threads to invoke. No more than the total number of logical processors on machine. Default 7.
paraSortInplaceMerge	A boolean value. TRUE let the algorithm call std::inplace_merge() (std refers to C++ STL namespace) instead of std::merge() for parallel-sorting the observation-centroid distances. In-place merge is slower but requires no extra memory.
verbose	A boolean value. TRUE prints progress.

Details

See details in KM() for common implementation highlights. Weight upper bounds are implemented as follows:

In each iteration, all the (observation ID, centroid ID, distance) tuples are sorted by distance. From the first to the last tuple, the algorithm puts observation in the cluster labeled by the centroid ID,

if (i) the observation has not already been assigned and (ii) the cluster size has not exceeded its upper bound. The actual implementation is slightly different. A parallel merge sort is crafted for computing speed.

Value

A list of size K, the number of clusters. Each element is a list of 3 vectors:

centroid	a numeric vector of size d.
clusterMember	an integer vector of indexes of elements grouped to centroid.
member2centroidDistance	a numeric vector of the same size of clusterMember. The ith element is the Minkowski distance or cosine dissimilarity from centroid to the clusterMember[i]th observation in X.

Empty clusterMember implies empty cluster.

Note

Although rarely happens, divergence of K-means with non-Euclidean distance minkP != 2 measure is still a theoretical possibility. Bounding the cluster weights / sizes increases the chance of divergence.

Examples

```
N = 3000L # Number of points.
d = 500L # Dimensionality.
K = 50L # Number of clusters.
dat = matrix(rnorm(N * d) + runif(N * d), nrow = d)


# Use kmeans++ initialization.
centroidInd = GMKMcharlie::KMppIni(
  X = dat, K, firstSelection = 1L, minkP = 2, stochastic = FALSE,
  seed = sample(1e9L, 1), maxCore = 2L, verbose = TRUE)


centroid = dat[, centroidInd]


# Each cluster size should not be greater than N / K * 2.
sizeConstraints = as.integer(rep(N / K * 2, K))
system.time({rst = GMKMcharlie::KMconstrained(
  X = dat, centroid = centroid, clusterWeightUB = sizeConstraints,
  maxCore = 2L, tailConvergedRelaErr = 1e-6, verbose = TRUE)})


# Size upper bounds vary in [N / K * 1.5, N / K * 2]
sizeConstraints = as.integer(round(runif(K, N / K * 1.5, N / K * 2)))
system.time({rst = GMKMcharlie::KMconstrained(
  X = dat, centroid = centroid, clusterWeightUB = sizeConstraints,
  maxCore = 2L, tailConvergedRelaErr = 1e-6, verbose = TRUE)})
```

`KMconstrainedSparse` *K-means over sparse data input with constraints on cluster weights*

Description

Multithreaded weighted Minkowski and spherical K-means via Lloyd's algorithm over sparse representation of data given cluster size (weight) constraints.

Usage

```
KMconstrainedSparse(
  X,
  d,
  centroid,
  Xw = rep(1, length(X)),
  clusterWeightUB = rep(length(X) + 1, length(centroid)),
  minkP = 2,
  convergenceTail = 5L,
  tailConvergedRelaErr = 1e-04,
  maxIter = 100L,
  maxCore = 7L,
  paraSortInplaceMerge = FALSE,
  verbose = TRUE
  )
```

Arguments

`X`	A list of size `N`, the number of observations. `X[[i]]` is a 2-column data frame. The 1st column is a sorted **integer vector** of the indexes of nonzero dimensions. Values in these dimensions are stored in the 2nd column as a **numeric vector**. Internally the algorithm sets a 32-bit *int* pointer to the beginning of the 1st column and a 64-bit *double* pointer to the beginning of the 2nd column, so it is critical that the input has the correct type.
`d`	An integer. The dimensionality of `X`. `d` MUST be no less than the maximum of all index vectors in `X`.
`centroid`	A list of size `K`, the number of clusters. `centroid[[i]]` can be in dense or sparse representation. If dense, a numeric vector of size `d`. If sparse, a 2-column data frame in the same sense as `X[[i]]`.
`Xw`	A numeric vector of size `N`. `Xw[i]` is the weight on observation `X[[i]]`. Users should normalize `Xw` such that the elements sum up to `N`. Default uniform weights for all observations.
`clusterWeightUB`	An integer vector of size `K`. The upper bound of weight for each cluster. If `Xw` are all 1s, `clusterWeightUB` upper-bound cluster sizes.

`minkP` A numeric value or a character string. If numeric, `minkP` is the power `p` in the definition of Minkowski distance. If character string, `"max"` implies Chebyshev distance, `"cosine"` implies cosine dissimilarity. Default 2.

`convergenceTail`
An integer. The algorithm may end up with "cyclical convergence" due to the size / weight constraints, that is, every few iterations produce the same clustering. If the cost (total in-cluster distance) of each of the last `convergenceTail` iterations has a relative difference less than `tailConvergedRelaErr` against the cost from the prior iteration, the program stops.

`tailConvergedRelaErr`
A numeric value, explained in `convergenceTail`.

`maxIter` An integer. The maximal number of iterations. Default 100.

`maxCore` An integer. The maximal number of threads to invoke. No more than the total number of logical processors on machine. Default 7.

`paraSortInplaceMerge`
A boolean value. `TRUE` let the algorithm call `std::inplace_merge()` (`std` refers to C++ STL namespace) instead of `std::merge()` for parallel-sorting the observation-centroid distances. In-place merge is slower but requires no extra memory.

`verbose` A boolean value. `TRUE` prints progress.

Details

See details for `KMconstrained()` and `KM()`

Value

A list of size `K`, the number of clusters. Each element is a list of 3 vectors:

`centroid` a numeric vector of size `d`.

`clusterMember` an integer vector of indexes of elements grouped to `centroid`.

`member2centroidDistance`
a numeric vector of the same size of `clusterMember`. The `i`th element is the Minkowski distance or cosine dissimilarity from `centroid` to the `clusterMember[i]`th observation in `X`.

Empty `clusterMember` implies empty cluster.

Note

Although rarely happens, divergence of K-means with non-Euclidean distance `minkP != 2` measure is still a theoretical possibility. Bounding the cluster weights / sizes increases the chance of divergence.

Examples

```
N = 5000L # Number of points.
d = 500L # Dimensionality.
K = 50L # Number of clusters.


# Create a data matrix, about 95% of which are zeros.
dat = matrix(unlist(lapply(1L : N, function(x)
{
  tmp = numeric(d)
  # Nonzero entries.
  Nnz = as.integer(max(1, d * runif(1, 0, 0.05)))
  tmp[sample(d, Nnz)] = runif(Nnz) + rnorm(Nnz)
  tmp
})), nrow = d); gc()


# Convert to sparse representation.
# GMKMcharlie::d2s() is equivalent.
sparsedat = apply(dat, 2, function(x)
{
  nonz = which(x != 0)
  list(nonz, x[nonz])
}); gc()


centroidInd = sample(length(sparsedat), K)


# Test speed using sparse representation.
sparseCentroid = sparsedat[centroidInd]
# Size upper bounds vary in [N / K * 1.5, N / K * 2]
sizeConstraints = as.integer(round(runif(K, N / K * 1.5, N / K * 2)))
system.time({sparseRst = GMKMcharlie::KMconstrainedSparse(
  X = sparsedat, d = d, centroid = sparseCentroid,
  clusterWeightUB = sizeConstraints,
  tailConvergedRelaErr = 1e-6,
  maxIter = 100, minkP = 2, maxCore = 2, verbose = TRUE)})
```

`KMppIni` *Minkowski and spherical, deterministic and stochastic, multithreaded K-means++ initialization over dense representation of data*

Description

Find suitable observations as initial centroids.

Usage

```
KMppIni(
  X,
  K,
  firstSelection = 1L,
  minkP = 2,
  stochastic = FALSE,
  seed = 123,
  maxCore = 7L,
  verbose = TRUE
  )
```

Arguments

X	A d x N numeric matrix where N is the number of data points — each column is an observation, and d is the dimensionality. Column-observation representation promotes cache locality.
K	An integer, the number of centroids.
firstSelection	An integer, index of the observation selected as the first initial centroid in X. Should be no greater than N.
minkP	A numeric value or a character string. If numeric, minkP is the power p in the definition of Minkowski distance. If character string, "max" implies Chebyshev distance, "cosine" implies cosine dissimilarity. Default 2.
stochastic	A boolean value. TRUE runs the stochastic K-means++ initialization by Arthur and Vassilvitskii (2007). Roughly speaking, the algorithm is stochastic in the sense that each of the remaining observations has a probability of being selected as the next centroid, and the probability is an increasing function of the minimal distance between this observation and the existing centroids. In the same context, the deterministic version selects as the next centroid with probability 1 the observation that has the longest minimal distance to the existing centroids.
seed	Random seed if stochastic.
maxCore	An integer. The maximal number of threads to invoke. No more than the total number of logical processors on machine. Default 7.
verbose	A boolean value. TRUE prints progress.

Details

In each iteration, the distances between the newly selected centroid and all the other observations are computed with multiple threads. Scheduling is homemade for minimizing the overhead of thread communication.

Value

An integer vector of size K. The vector contains the indexes of observations selected as the initial centroids.

Examples

```
N = 30000L
d = 300L
K = 30L
X = matrix(rnorm(N * d) + 2, nrow = d)
# CRAN check allows examples invoking 2 threads at most. Change `maxCore`
# for acceleration.
kmppSt = KMppIni(X, K, firstSelection = 1L, minkP = 2,
                 stochastic = TRUE, seed = sample(1e9L, 1), maxCore = 2L)
kmppDt = KMppIni(X, K, firstSelection = 1L, minkP = 2,
                 stochastic = FALSE, maxCore = 2L)
str(kmppSt)
str(kmppDt)
```

`KMppIniSparse` *Minkowski and spherical, deterministic and stochastic, multithreaded K-means++ initialization over sparse representation of data*

Description

Find suitable observations as initial centroids.

Usage

```
KMppIniSparse(
  X,
  d,
  K,
  firstSelection = 1L,
  minkP = 2,
  stochastic = FALSE,
  seed = 123,
  maxCore = 7L,
  verbose = TRUE
  )
```

Arguments

`X`	A list of size `N`, the number of observations. `X[[i]]` is a 2-column data frame. The 1st column is a sorted **integer vector** of the indexes of nonzero dimensions. Values in these dimensions are stored in the 2nd column as a **numeric vector**. Internally the algorithm sets a 32-bit *int* pointer to the beginning of the 1st column and a 64-bit *double* pointer to the beginning of the 2nd column, so it is critical that the input has the correct type.
`d`	An integer. The dimensionality of `X`. `d` MUST be no less than the maximum of all index vectors in `X`.
`K`	An integer, the number of centroids.

`firstSelection`	An integer, index of the observation selected as the first initial centroid in `X`. Should be no greater than `N`.
`minkP`	A numeric value or a character string. If numeric, `minkP` is the power `p` in the definition of Minkowski distance. If character string, `"max"` implies Chebyshev distance, `"cosine"` implies cosine dissimilarity. Default 2.
`stochastic`	A boolean value. `TRUE` runs the stochastic K-means++ initialization by Arthur and Vassilvitskii (2007). Roughly speaking, the algorithm is stochastic in the sense that each of the remaining observations has a probability of being selected as the next centroid, and the probability is an increasing function of the minimal distance between this observation and the existing centroids. In the same context, the deterministic version selects as the next centroid with probability 1 the observation that has the longest minimal distance to the existing centroids.
`seed`	Random seed if `stochastic`.
`maxCore`	An integer. The maximal number of threads to invoke. No more than the total number of logical processors on machine. Default 7.
`verbose`	A boolean value. `TRUE` prints progress.

Details

In each iteration, the distances between the newly selected centroid and all the other observations are computed with multiple threads. Scheduling is homemade for minimizing the overhead of thread communication.

Value

An integer vector of size `K`. The vector contains the indexes of observations selected as the initial centroids.

Examples

```
N = 2000L
d = 3000L
X = matrix(rnorm(N * d) + 2, nrow = d)
# Fill many zeros in X:
X = apply(X, 2, function(x) {
  x[sort(sample(d, d * runif(1, 0.95, 0.99)))] = 0; x})
# Get the sparse version of X.
sparseX = GMKMcharlie::d2s(X)


K = 30L
seed = 123L
# Time cost of finding the centroids via dense representation.
# CRAN check allows only 2 threads. Increase `maxCore` for more speed.
system.time({kmppViaDense = GMKMcharlie::KMppIni(
  X, K, firstSelection = 1L, minkP = 2, stochastic = TRUE, seed = seed,
  maxCore = 2L)})
```

```
# Time cost of finding the initial centroids via sparse representation.
system.time({kmppViaSparse = GMKMcharlie::KMppIniSparse(
  sparseX, d, K, firstSelection = 1L, minkP = 2, stochastic = TRUE,
  seed = seed, maxCore = 2L)})


# Results should be identical.
sum(kmppViaSparse - kmppViaDense)
```

KMsparse *K-means over sparse representation of data*

Description

Multithreaded weighted Minkowski and spherical K-means via Lloyd's algorithm over sparse representation of data.

Usage

```
KMsparse(
  X,
  d,
  centroid,
  Xw = rep(1, length(X)),
  minkP = 2,
  maxIter = 100L,
  maxCore = 7L,
  verbose = TRUE
  )
```

Arguments

X	A list of size N, the number of observations. X[[i]] is a 2-column data frame. The 1st column is a sorted **integer vector** of the indexes of nonzero dimensions. Values in these dimensions are stored in the 2nd column as a **numeric vector**. Internally the algorithm sets a 32-bit *int* pointer to the beginning of the 1st column and a 64-bit *double* pointer to the beginning of the 2nd column, so it is critical that the input has the correct type.
d	An integer. The dimensionality of X. d MUST be no less than the maximum of all index vectors in X.
centroid	A list of size K, the number of clusters. centroid[[i]] can be in dense or sparse representation. If dense, a numeric vector of size d. If sparse, a 2-column data frame in the same sense as X[[i]].
Xw	A numeric vector of size N. Xw[i] is the weight on observation X[[i]]. Users should normalize Xw such that the elements sum up to N. Default uniform weights for all observations.

minkP	A numeric value or a character string. If numeric, minkP is the power p in the definition of Minkowski distance. If character string, "max" implies Chebyshev distance, "cosine" implies cosine dissimilarity. Default 2.
maxIter	An integer. The maximal number of iterations. Default 100.
maxCore	An integer. The maximal number of threads to invoke. No more than the total number of logical processors on machine. Default 7.
verbose	A boolean value. TRUE prints progress.

Details

See details in KM() for implementation highlights. There are some other optimizations such as, except for the maximum norm, cost of computing the distance between a dense centroid vector and a sparse observation is linear to the size of the sparse observation, which should be largely less than the size of the dense vector. This is done by letting every centroid memorize its before-root Minkowski norm. The full distance can then be inferred from adding the residual norm to the partial distance.

Value

A list of size K, the number of clusters. Each element is a list of 3 vectors:

centroid	a numeric vector of size d.
clusterMember	an integer vector of indexes of elements grouped to centroid.
member2centroidDistance	a numeric vector of the same size of clusterMember. The ith element is the Minkowski distance or cosine dissimilarity from centroid to the clusterMember[i]th observation in X.

Empty clusterMember implies empty cluster.

Note

Although rarely happens, divergence of K-means with non-Euclidean distance minkP != 2 measure is still a theoretical possibility.

Examples

```
# ============================================================================
# Play random numbers. See speed.
# ============================================================================
N = 10000L # Number of points.
d = 500L # Dimensionality.
K = 100L # Number of clusters.


# Create a data matrix, about 95% of which are zeros.
dat = matrix(unlist(lapply(1L : N, function(x)
{
  tmp = numeric(d)
  # Nonzero entries.
```

```
  Nnz = as.integer(max(1, d * runif(1, 0, 0.05)))
  tmp[sample(d, Nnz)] = runif(Nnz) + rnorm(Nnz)
  tmp
})), nrow = d); gc()


# Convert to sparse representation.
# GMKMcharlie::d2s() acheives the same.
sparsedat = apply(dat, 2, function(x)
{
  nonz = which(x != 0)
  list(nonz, x[nonz])
}); gc()


centroidInd = sample(length(sparsedat), K)


# Test speed using dense representation.
centroid = dat[, centroidInd]
system.time({rst = GMKMcharlie::KM(
  X = dat, centroid = centroid, maxIter = 100,
  minkP = 2, maxCore = 2, verbose = TRUE)})


# Test speed using sparse representation.
sparseCentroid = sparsedat[centroidInd]
system.time({sparseRst = GMKMcharlie::KMsparse(
  X = sparsedat, d = d, centroid = sparseCentroid,
  maxIter = 100, minkP = 2, maxCore = 2, verbose = TRUE)})
```

---

s2d *Sparse to dense conversion*

Description

Convert data from sparse representation (list of data frames) to dese representation (matrix).

Usage

```
s2d(
  X,
  d,
  zero = 0,
  verbose = TRUE
  )
```

Arguments

X	A list of size N, the number of observations. X[[i]] is a 2-column data frame. The 1st column is a sorted integer vector of the indexes of nonzero dimensions. Values in these dimensions are stored in the 2nd column as a numeric vector.
d	An integer. The dimensionality of X. d MUST be no less than the maximum of all index vectors in X.
zero	A numeric value. In the result matrix, entries not registered in X will be filled with zero.
verbose	A boolean value. TRUE prints progress.

Value

A d x N numeric matrix.

Examples

```
N = 2000L
d = 3000L
X = matrix(rnorm(N * d) + 2, nrow = d)
# Fill many zeros in X:
X = apply(X, 2, function(x) {
  x[sort(sample(d, d * runif(1, 0.95, 0.99)))] = 0; x})
# Get the sparse version of X.
sparseX = GMKMcharlie::d2s(X)
# Convert it back to dense.
X2 = GMKMcharlie::s2d(sparseX, d)
range(X - X2)
```

Index